QUICKAR: Automatic Query Reformulation for Concept Location using Crowdsourced Knowledge

Mohammad Masudur Rahman   Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Canada
{masud.rahman, chanchal.roy}@usask.ca

ABSTRACT

During maintenance, software developers deal with numerous change requests made by the users of a software system. Studies show that the developers find it challenging to select appropriate search terms from a change request during concept location. In this paper, we propose a novel technique–QUICKAR–that automatically suggests helpful reformulations for a given query by leveraging the crowdsourced knowledge from Stack Overflow. It determines semantic similarity or relevance between any two terms by analyzing their adjacent word lists from the programming questions of Stack Overflow, and then suggests semantically relevant queries for concept location. Experiments using 510 queries from two software systems suggest that our technique can improve or preserve the quality of 76% of the initial queries on average which is promising. Comparison with one baseline technique validates our preliminary findings, and also demonstrates the potential of our technique.

CCS Concepts

•**Software and its engineering** → **Software maintenance tools; Requirements analysis;** *Traceability; Maintaining software;*

Keywords

Query reformulation, crowdsourced knowledge, semantic relevance, word co-occurrence, adjacency list, Stack Overflow

1. INTRODUCTION

Studies show that about 85%–90% of the total effort is spent in software maintenance and evolution [3, 4]. During maintenance, software developers deal with numerous change requests made by the users of a software system. Although the users might be familiar with the application domain of the software, they generally lack the idea of how a particular software feature is implemented in the source

code. Hence, the requests from them generally involve domain related concepts (e.g., application features), and they are written in an unstructured fashion using natural language texts. The developers need to prepare appropriate search query from those concepts, and then identify the relevant location(s) in the code to implement the requested change(s). Unfortunately, preparing such a query is highly challenging and error-prone for the developers [5, 13]. Based on a user study, Kevic and Fritz [13] report that developers were able to suggest good quality search terms for only 12.2% of the change tasks. Furnas et al. [5] suggest that there is a little chance (i.e., 10%–15%) that developers guess the exact words used in the source code. One way to assist the developers in this regard is to automatically suggest helpful reformulations for the initially executed query.

Existing studies use relevance feedback from developers [6] or information retrieval techniques [10], query quality [8, 9] and the context of query terms within the source code [12, 23] in suggesting reformulated queries. Gay et al. [6] make use of explicit feedback on document relevance from the software developers, and then suggest reformulated queries using Rocchio's expansion. Haiduc et al. and colleagues [7, 8, 9, 10, 11] take quality of the query into consideration, and suggest the best reformulation strategy for a given query using machine learning. Howard et al. [12] analyze leading comments and method signatures from the source code for mining semantically similar word pairs, and then suggest reformulated query using those word pairs. While these above techniques are reported to be novel or effective, they are also limited in certain aspects. First, collecting explicit feedback from the developers could be highly expensive, and such study [6] could also be hard to replicate. Second, machine learning model of Haiduc et al. is reported to be performing well in the case of *within-project* training, and only 51–72 queries are considered from each of the five projects for training and testing [10]. Given such small dataset, the reported performance possibly could not be generalized for large systems. Third, Howard et al. require the source code to be well documented for the mining of word pairs, and hence, might not perform well if the code is poorly documented [12]. Thus, we need a technique that is neither subject to the training data nor the availability of comments in the source code. One way to possibly overcome those concerns is to apply *crowd generated knowledge* in the reformulation of queries for concept location.

In this paper, we propose a novel technique–QUICKAR– that automatically identifies semantically similar words to an initial query not only from project source code but also

from crowdsourced content of Stack Overflow Q & A site, and then suggests a reformulated query. The technique collects adjacent word lists from the programming questions of Stack Overflow for any two terms, and determines their semantic similarity by comparing their corresponding adjacency lists [24]. In short, QUICKAR not only follows the essence of a *nearest neighbour classifier* [16] in the context of natural language texts but also harnesses the technical corpus developed by a large crowd over the years in estimating *semantic similarity* or *relevance*. Such simple but intuitive estimation of semantic relationship could be highly useful for suggesting an alternative version of a given query. QUICKAR also addresses the overarching *vocabulary mismatch problem* [5]. First, Stack Overflow is curated by a large crowd of four million technical users, and the millions of questions and answers posted by them are a great source for technical vocabulary (e.g., API names) [21]. Thus, QUICKAR mines semantically similar words not only from a larger corpus (i.e., compared to a single project [23]) but also from a more appropriate vocabulary (i.e., compared to WordNet [22, 23]). Second, Rahman et al. [20] reported a significant overlap (i.e., 73%) between the vocabulary of real life code search queries and that of question titles from Stack Overflow. QUICKAR mines 500K programming related questions from Stack Overflow carefully, and reaps the benefit through meaningful vocabulary extension. To the best of our knowledge, no existing studies apply crowdsourced knowledge yet in the reformulation of queries for concept location which makes our technique novel.

Experiments using 510 concept location queries from two subject systems–`ecf` and `eclipse.pde.ui`–suggest that our technique can improve or preserve the quality of 76% (i.e., improves 66% and preserves 10%) of the initial queries through reformulation which is promising according to relevant literature [6, 10]. Comparison with one baseline technique–Rocchio's expansion [1, 10]–validates our preliminary findings, and also demonstrates the potential of our technique for query reformulation. While the preliminary findings are promising, they must be validated using further experiments. In this paper, we make the following contributions:

- Construction of a word adjacency list database by mining 500K questions from Stack Overflow for the estimation of semantic similarity or relevance between words.
- A novel technique that suggests helpful reformulations for a given query for concept location by leveraging crowdsourced knowledge from Stack Overflow.

2. MOTIVATING EXAMPLE

Software change requests and project source code are often written by different people, and they use different vocabularies to describe the same technical concept. Concept location community has termed it as *vocabulary mismatch problem* [5, 10]. Table 1 shows three different questions from Stack Overflow that are marked as *duplicates* or *linked* by the users of the site. These questions use three different verbs–*'create', 'cause'* and *'track'*– to describe the same programming issue–*locating memory leaks*, and this can be considered as a real life parallel for vocabulary mismatch issue. Although, these verbs have different semantics in English literature, they share the same or almost similar semantic in this technical literature– programming questions [23]. More interestingly, their semantic similarities can also be approximated from their adjacent word lists. In graph

Table 1: Duplicate Questions from Stack Overflow

ID	Title of Question
6470651	Creating a memory leak with Java
4948529	Easiest way to cause memory leak in Java?
1071631	Tracking down a memory leak/garbage-collection issue in Java

theory, two nodes are considered to be *connected* if they share the same neighbour nodes [16]. We adapt that idea for natural language texts, and apply to the estimation of semantic connectivity between words. The co-occurred word lists (i.e., sentence as a context unit) of *'create', 'cause'* and *'track'*–{*memory, leak, Java*}, {*easiest, way, memory, leak, Java*} and {*down, memory, leak, garbage, collection, issues, Java*}–respectively share multiple words among themselves. Thus, comparison between any two such lists can potentially approximate the semantic similarity or relevance of their corresponding words [24]. In this research, we apply the above methodology in semantic similarity estimation, and then use the similar terms for the reformulation of an initial query.

## 3. QUICKAR: QUERY REFORMULATION USING CROWDSOURCED KNOWLEDGE

Programming questions and answers from Stack Overflow were previously mined for API elements [20, 21], Q & A dynamics [14, 17] or post suggestions [18, 19]. In this research, we make a novel use of such questions in query reformulation for concept location. Since these questions contain unstructured information, relevant items (i.e., potential alternative query term) should be carefully extracted and then applied to query reformulation. We first construct a database containing adjacent word list for each of the individual words taken from the title of the questions, and then leverage that information in suggesting reformulated queries. Fig. 1 shows the schematic diagram of our proposed technique–QUICKAR–for query reformulation. Thus, our technique can be divided into two major parts–(a) Construction of an adjacency list database from Stack Overflow questions, and (b) Reformulation of an initial query–as follows:

3.1 Construction of Adjacency List Database

Word co-occurrence is often considered as a proxy for semantic relevance between words in natural language texts [15]. Yuan et al. [24] first propose to use contextual words (i.e., word co-occurrence) from the programming questions and answers of Stack Overflow in identifying semantically similar software-specific word pairs. While they introduce the idea, we adapt that idea for a specific software maintenance task–search query reformulation for concept location. Given the significant overlap (i.e., 73%) between real life code search queries and titles of Stack Overflow questions [20], we collect the titles of 500K Java related questions from Stack Overflow using Stack Exchange Data Explorer[1]. We check for `<java>` tag in the tag list for identifying the Java related questions. We perform standard natural language preprocessing (i.e., word splitting, stop word removal), and turn each of those titles into a sequence of words (Step 3, 4, Fig. 1-(a)). We decompose each camel case word (e.g., `GenericContainerInstantiator`) into separate tokens (e.g., `Generic, Container, Instantiator`), and apply a standard list[2] for stop word removal. It should be noted that we avoid stemming to ensure a meaningful refor-

---

[1] http://data.stackexchange.com/stackoverflow

[2] https://code.google.com/p/stop-words/
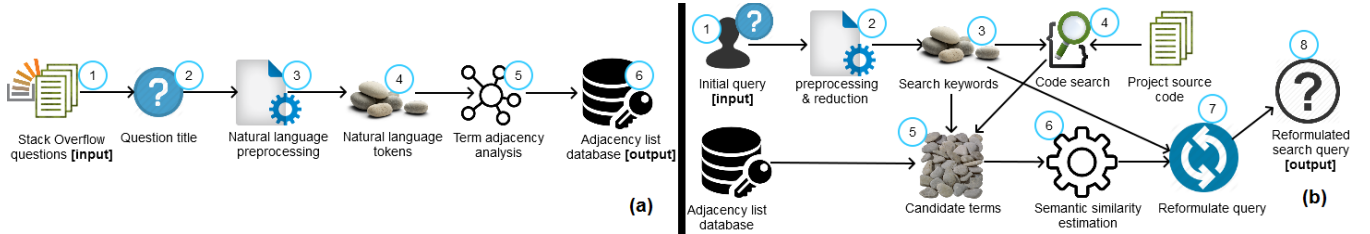
Figure 1: Proposed technique for query reformulation–(a) Construction of word adjacency list database from Stack Overflow questions, and (b) Reformulation of an initial query for concept location

mulation of the query. After preprocessing step, we found a large set of 81,394 individual words (i.e., 2,660,257 words in total) from our collected titles. Given that source code of a software project is often authored by a small group of developers, such a large set could possibly *extend* the vocabulary of the code. We then use a *sliding window* of `window size = 2` to capture co-occurred words from the titles [15], and construct an adjacency list for each of the individual words. For example, based on Table 1, the adjacency list for the word *'memory'* would be–{*creating, leak, cause, down*}. We collect adjacency list for each of the 81,394 words where each list contains 21 co-occurred words on average (Step 5, 6, Fig. 1-(a)). These lists comprise our database which is later accessed frequently for query reformulation.

3.2 Reformulation of an Initial Query

Fig. 1-(b) shows the schematic diagram and Algorithm 1 shows the pseudo code for our query reformulation technique–QUICKAR. We collect semantically similar words (i.e., with initial query) from two different but relevant sources–project source code and Stack Overflow questions, and then reformulate a given query for concept location. We discuss different intermediate steps involved in such reformulation as follows:

Collection of Candidate Terms: Existing literature mostly relies on the source code of a software system [10, 12, 23] or WordNet [22] for reformulated query suggestion. Unfortunately, source code often might not contain a rich vocabulary [12] and WordNet might also not be appropriate for technical words [22, 24]. We thus collect candidate terms for possible query expansion not only from the source code but also from another technical literature–programming questions of Stack Overflow (Line 6–Line 8, Algorithm 1). In the case of project source, we perform code search using the reduced keywords (K) from the initial query (Q). We use *Apache Lucene* [18], a popular implementation of Vector Space Model (VSM), for code search, and then collect the Top-5 (i.e., cut off) retrieved documents as the source for candidate terms. The cut-off value is chosen based on iterative experiments. We perform standard natural language preprocessing on those documents, and extract each of the terms as the reformulation candidates (T_p) (Step 3–5, Fig. 1-(b)). Relevant literature [10] also follows the same procedure for candidate term selection. In the case of questions from Stack Overflow, we collect such words as candidates (T_{so}) that frequently co-occurred in those questions with the keywords from the initial query. The underlying idea is that if two words frequently co-occur in various technical contexts, they share their semantics and thus, are possibly semantically relevant [15, 24]. We use our adjacency list database (Section 3.1) for identifying the second set of candidates.

Estimation of Semantic Similarity or Relevance: Since we target to reformulate a query using meaningful al-

---

Algorithm 1 Query Reformulation using Crowd Knowledge

1: **procedure** QUICKAR(Q)   ▷ Q: initial search query
2:     $Q' \leftarrow \{\}$ ▷ reformulated search query
3:     ▷ collecting keywords from the initial search query
4:     $K \leftarrow$ collectKeywords(Q)
5:     $K \leftarrow$ reduceKeywords(K)
6:     ▷ collecting candidate terms
7:     $T_p \leftarrow$ getCandidateTermsFromProject(K)
8:     $T_{so} \leftarrow$ getCandidateTermsFromSO(K)
9:     ▷ estimating semantic similarity of the candidates
10:     **for** Candidate $T_i \in T_p$ **do**
11:         $Adj_{T_i} \leftarrow$ getAdjacencyListFromDB(T_i)
12:         **for** Keyword $K_j \in K$ **do**
13:             $Adj_{K_j} \leftarrow$ getAdjacencyListFromDB(K_j)
14:             ▷ contextual similarity between words
15:             $S_{cos} \leftarrow$ getCosineSimilarity(Adj_{T_i}, Adj_{K_j})
16:             $R_p[T_i].score \leftarrow R_p[T_i].score + S_{cos}$
17:         **end for**
18:     **end for**
19:     **for** Candidate $T_i \in T_{so}$ **do**
20:         **for** Keyword $K_j \in K$ **do**
21:             ▷ co-occurrence between words
22:             $S_{cof} \leftarrow$ getCo-occurrenceFreq(T_i, K_j)
23:             $R_{so}[T_i].score \leftarrow R_{so}[T_i].score + S_{cof}$
24:         **end for**
25:     **end for**
26:     ▷ ranking and selection of candidates
27:     $SR_{Top_p} \leftarrow$ selectTopK(sortByScore(R_p))
28:     $SR_{Top_{so}} \leftarrow$ selectTopK(sortByScore(R_{so}))
29:     $R \leftarrow$ selectiveCombine($SR_{Top_p}, SR_{Top_{so}}$)
30:     ▷ reformulate the initial query
31:     $Q' \leftarrow$ selectiveReformulate(Q, K, R)
32:     **return** Q'
33: **end procedure**

---

ternatives, we need to choose such terms from the candidates that are either semantically similar or highly relevant to the initial query. Yuan et al. use contextual words (i.e., based on co-occurrence) from Stack Overflow for automatically identifying similar software-specific words. In the context of query reformulation for software maintenance, we similarly apply adjacency list to the estimation of semantic relevance between two terms. We collect adjacency list (i.e., from the adjacency list database) for each of the candidate terms and the keywords of the initial query, and estimate their similarities using *cosine similarity* measure (S_{cos}). Cosine similarity is frequently used in information retrieval for determining textual similarity between two given documents. It returns a value between zero (i.e., completely dissimilar) and one (i.e., completely similar). Thus, a candidate term achieves score only if it shares its context (i.e., adjacent word list, Adj_{T_i}) with that (i.e., Adj_{K_j}) of the keywords across various questions from Stack Overflow. QUICKAR iterates this

Table 2: Experimental Dataset

Subject System	Release ID	#Files	#Methods	#Queries
ecf	170_170	5,781	21,447	222
eclipse.pde.ui	I20151110-0800	7,579	31,468	288
Total	–	13,360	52,915	510

process for each of the candidates (T_p), and accumulates their similarity scores against the keywords (Line 9–Line 18, Algorithm 1, Step 6, Fig. 1-(b)). We also determine co-occurrence frequency (S_{cof}) between each candidate term and each keyword in the titles of Stack Overflow questions [20], and derive another set of scores for the candidates (T_{so}) (Line 19–Line 25, Algorithm 1, Step 6, Fig. 1-(b)). Finally, we end up with two sets of candidates (i.e., collected from two different sources), and QUICKAR determined their relevance to the initial query in terms of their context or direct co-occurrences in the Stack Overflow questions.

Candidate Term Ranking & Top-K Selection : Once relevance estimates for both candidate sets–R_p and R_{so}–are collected, they are ranked based on their estimates. We then collect the Top-$K = 5$ candidates from each ranked list, selectively choose the top candidates (R) from both lists, and then treat them as similar or relevant to the initial query (Q) (Line 26–Line 29, Algorithm 1). In particular, the *nominal terms* (i.e., nouns) from both lists are chosen [20]. Thus, we select such terms for reformulation that co-occur with the initial query keywords not only in the project source but also in the titles of the questions from Stack Overflow.

Query Reduction & Expansion: We apply both reformulation strategies–*reduction* and *expansion*–to the initial query [1]. In the case of *reduction*, we apply a conservative strategy as was also applied by Haiduc et al.. We discard the keywords from initial query that either are *non-nominal* [20] or occurred in more than 25% of the documents of the project corpus. Such keywords are not specific enough and thus, are not useful for document retrieval [10]. In the case of *expansion*, we apply the semantically similar or relevant terms (R) returned by QUICKAR to the query. If there exist M terms after the reduction step (i.e., Line 5), we append $(10-M)$ relevant terms from R to the query, and prepare an alternative query (Q') (Line 29–Line 32, Algorithm 1, Step 7, 8, Fig. 1-(b)). We also decompose each camel case term into separate tokens, and preserve both the separate and the camel case terms into the reformulated query [10]. It should be noted that if our reduction step already improves the initial query, we conveniently avoid its expansion.

Working Example: Let us consider a change request (ID: 408030) from `ecf` project. We select title of the request– *"RestClientService ignores content encoding"*–as the initial search query for the request, as was also used by the existing literature [10]. The query returns the first relevant document (i.e., Java class) at 44th position when tested using *Apache Lucene*. On the other hand, QUICKAR returns the reformulated query–*"Rest Client Service RestClientService content Web Java Executor WebService Http"*–that returns the same relevant document at 1st position in the search result. Please note that our technique expands the initial query using several relevant terms such as *"WebService"*, *"Executor"* and *"Http"* and also discards some terms such as *"encoding"* or *"ignore"*, which improved the query. We also expand the query simply using terms from project source– *"RestClientService ignores content encoding call container service http test"*–that returns the document at 11th position. This clearly demonstrates that candidates from the source

code of a project might always not be sufficient, and crowd generated vocabulary from Stack Overflow can complement them, and thus, can assist in effective query reformulation.

4. EXPERIMENT

One of the most effective ways to evaluate a query reformulation technique is to check whether the reformulated query improves the search results or not. We define improvement of search results as the bubbling up of the first relevant document to the top positions of the result list [10]. That is, a good reformulation of query provides a better rank than a baseline query for the first relevant document. We conduct experiments using 510 change requests from two Java subject systems of *Eclipse*–`ecf` and `eclipse.pde.ui`– and a well known search engine–*Apache Lucene* [10, 18]. We also compare with a baseline technique for query reformulation to validate our findings. In particular, we attempt to answer the following research questions using our experiments:

- **RQ₁:** How does QUICKAR perform in the reformulation of a query for concept location?
- **RQ₂:** Can crowdsourced knowledge from Stack Overflow improve a given query significantly?
- **RQ₃:** How does QUICKAR perform compared to the baseline technique for query reformulation?

4.1 Experimental Dataset & Corpus

Dataset Collection: Table 2 shows details of our selected subject systems. We first collect the *RESOLVED* change requests (i.e., bug reports) from *BugZilla* for each of the selected systems. Then we identify such commits that implemented those requests in their corresponding *GitHub* repositories. We consider a commit as *eligible* only if its title contains a specific request identifier (e.g., Bug: 408030). This practice is common for relevant literature for evaluation [2]. The step provides 495 and 542 requests from `ecf` and `eclipse.pde.ui` respectively. We also collect *change set* from each of the identified commits which are later used as the *solutions* for the corresponding change requests. We then consider title of each request as the baseline query, and identify such queries that return their first relevant results with poor rank (i.e., *rank*>10). That is, the baseline query needs reformulation for returning a better rank. This filtration step left us with 222 and 288 baseline queries from `ecf` and `eclipse.pde.ui` respectively for the experiments.

Corpus Preparation: Unlike an unstructured natural language document, a source code document contains items beyond regular texts such as *classes, interfaces, methods* and *constructors*. One should consider such structures for effective retrieval of the source documents from a project. We thus decompose each Java document into methods, and consider each of those methods as a single document of the corpus. This step provides 21,447 and 31,468 Java methods from `ecf` and `eclipse.pde.ui` respectively. We collect these methods using Javaparser[3], and apply natural language preprocessing to them. In particular, we remove all punctuation marks, Java programming keywords and English stop words from the body and signature of the method, and also decompose each camel case token into individual tokens.

4.2 Evaluation of QUICKAR

We execute each of our reformulated queries and baseline queries from each subject system with *Apache Lucene*,

---
[3]https://github.com/javaparser/javaparser

Table 3: Performance of QUICKAR

System	#Queries	Improvement							Worsening							Preserving
		#Improved	Mean	Q1	Q2	Q3	Min.	Max.	#Worsened	Mean	Q1	Q2	Q3	Min.	Max.	#Preserved
ecf	222	**159 (71.62%)**	194	**10**	27	156	1	2335	41 (18.47%)	764	67	245	1173	18	4330	**22 (9.90%)**
pde.ui	288	**177 (61.46%)**	219	**17**	51	171	1	4766	83 (28.82%)	558	93	244	630	17	2996	**28 (9.72%)**
	Total=510	**Avg=66.54%**							Avg=23.65%							**Avg=9.81%**

pde.ui=eclipse.pde.ui, **Mean**=Mean rank of first relevant document in the search result, $\mathbf{Q}_i$= Rank value for i^{th} quartile of all result ranks

Table 4: Role of Crowdsourced Knowledge from SO

Technique	Improved	Worsened	Preserved
Baseline query (preprocessed)	17.84%	9.90%	**72.27%**
QUICKAR$_P$	49.15%	48.41%	2.44%
QUICKAR$_{SO}$	47.83%	49.91%	2.27%
QUICKAR$_{red}$	55.55%	24.46%	19.99%
QUICKAR$_{ALL}$	**66.54**%	23.65%	**9.81**%

Table 5: Comparison with Baseline Technique

Technique	System	Improved	Worsened	Preserved
Rocchio's Expansion [1]	ecf	39.64%	59.46%	<1.00%
	pde.ui	40.63%	59.38%	0.00%
QUICKAR$_P$	ecf	53.15%	43.69%	3.15%
	pde.ui	45.14%	53.13%	1.74%
QUICKAR$_{ALL}$	ecf	**71.62%**	18.47%	**9.90%**
	pde.ui	**61.46%**	28.82%	**9.72%**

pde.ui=eclipse.pde.ui

Table 6: Result of Mann Whitney U-Tests

Technique pair	ecf		eclipse.pde.ui	
	p-value	MRD	p-value	MRD
QUICKAR$_{ALL}$ vs. QUICKAR$_P$	**0.007**<0.05	-248	**<0.001**	-369
QUICKAR$_{ALL}$ vs. QUICKAR$_{SO}$	0.115>0.05	-109	**<0.001**	-253
QUICKAR$_{ALL}$ vs. Rocchio [1]	**<0.001**	-388	**<0.001**	-332

MRD= **M**ean **R**ank **D**ifference

and compare their topmost ranks for evaluation. We identify the queries that were improved, worsened or preserved based on those ranks. Table 3 reports the outcome of our preliminary investigation. On average, QUICKAR was able to improve 66% of the baseline queries while preserving the quality of 10% which are highly promising according to relevant literature [6, 10, 12]. We see that QUICKAR can return the top results within the 10th position for 25% of 159 requests from `ecf` system. It also performs similarly for `eclipse.pde.ui`, and returns the top results within the 17th position. One might argue about the verbatim use of the title of a change request as the baseline query. However, we also experimented with preprocessed version (i.e., stop word removal, camel case decomposition) of the title. We found that the preprocessing step discarded important information, and did not provide much improvement in the query quality, which possibly justifies our choice about baseline query. Thus, to answer **RQ**$_1$, our proposed technique for query reformulation–QUICKAR–improves or preserves 76% of the 510 baseline queries which is promising.

We investigate how different reformulation decisions influence the end performance of our technique, and Table 4 reports our findings. We first experimented using a preprocessed version of the baseline queries, and found that the preprocessing step did not improve the queries much (i.e., only 18% improvement). Since candidate terms for reformulation are extracted from both project source code and Stack Overflow questions, we need to examine their impact in the reformulated queries. When we rely solely on source code, QUICKAR$_P$ can improve 49% of the queries but degrades the quality of 48%. Although the candidate terms from Stack Overflow questions alone might not be sufficient (i.e., QUICKAR$_{SO}$ improves 48% and degrades 50%), they definitely can complement the candidate terms from the project source which leads to overall query quality improvement (i.e., QUICKAR$_{ALL}$ improves 66%). We also performed *Mann Whitney U (MWU)*-test on the provided ranks by QUICKAR$_P$ and QUICKAR$_{ALL}$, and found that QUICKAR$_{ALL}$ returns the results at significantly higher ranks (i.e., *p-values* 0.007<0.05 and 0.001<0.05 for `ecf` and `eclipse.pde.ui` respectively, Table 6) in the result list. The negative *mean rank difference* (MRD) in Table 6 suggests that QUICKAR$_{ALL}$ returns the results at relatively closer to the top of the list than the counterpart, i.e., *extent of result rank improvement*. Thus, to answer **RQ**$_2$, crowdsourced knowledge from Stack Overflow questions can significantly improve the quality of a baseline query during reformulation.

Since QUICKAR involves two steps during query reformulation – *reduction* and *expansion*, an investigation is war-ranted on how these two steps impact the end performance. According to our preliminary investigation, the reduction step (i.e., QUICKAR$_{red}$) dominates over expansion step especially with `ecf` system. One possible explanation could be that we used a smaller version of the adjacency list database constructed from 50,000 (i.e., 10%) questions from the dataset. Since accessing a large database is time-consuming, we made this feasible choice during experiment. However, further investigation and experiments are essential to mitigate such concern which we consider as a future work. In short, the potential of crowdsourced knowledge is yet to be explored.

4.3 Comparison with Baseline Technique

Although the conducted evaluation demonstrates the potential of our proposed technique–QUICKAR, we still investigate to at least partially validate our performance. We compare with a baseline technique–Rocchio's expansion [1, 10]–that is reported to be effective for query reformulation. Rocchio's method first collects candidate terms from the Top-K (i.e., $K = 5$) source code documents returned by a baseline query. Then, it selects the most important candidate terms (t) for reformulation by calculating their *TF-IDF* in each of those Top-K documents (D) as follows:

$$Rocchio(t) = \sum_{d \in R} TFIDF(t, d)$$

We implemented Rocchio's method in our working environment, experimented on the same corpus and applied similar natural language preprocessing. Table 5 reports the comparative analysis between our technique and Rocchio's method. We see that our technique can improve 60%–70% of the baseline queries from each of the subject systems whereas such measure for Rocchio's method is close to 40%. More importantly, QUICKAR degrades less number of queries compared to its counterpart. While Rocchio's method worsened the quality of 60% of the baseline queries during reformulation, such measure for QUICKAR is between 18% to 29%. We also performed *Mann Whitney U*-test on the returned result ranks from both techniques, and found that QUICKAR provides significantly better ranks (i.e., *p-values* <0.001 and <0.001 for `ecf` and `eclipse.pde.ui` respectively, Table 6) than Rocchio's expansion. We also compared using a equiv-

alent variant of our technique–QUICKAR$_P$, and found that the variant still performed better than Rocchio's method for both subject systems. All these above preliminary findings clearly demonstrate the potential of our proposed technique. Thus, to answer **RQ**$_3$, our technique–QUICKAR–performs significantly better than the baseline technique [1] in query reformulation for concept location.

5. RELATED WORK

Existing studies from the literature use relevance feedback from developers [6] or information retrieval techniques [10], query quality [8, 9] or the context of a query in the source code [12, 23] for suggesting query reformulations. Gay et al. [6] capture explicit feedback on document relevance from the software developers, and then suggest reformulated queries using Rocchio's expansion. Although their adopted methodology is meaningful, capturing feedback from the developers could be expensive, and such study is often difficult to replicate. Haiduc et al. and colleagues [7, 8, 9, 10, 11] analyze quality of the query, and suggest the best reformulation strategy for any given query using machine learning. Although their reported performance is significantly higher, such performance might not be generalized for large systems given their use of small dataset (i.e., only 51–72 queries from each system). Howard et al. [12] analyze leading comments and method signatures from the source code, and suggest reformulated queries by extracting semantically similar word pairs. However, their technique requires the source code to be well documented, and thus, might not perform well with poorly documented code. On the other hand, our technique–QUICKAR–complements the source code vocabulary by capturing appropriate candidate terms from the programming questions of Stack Overflow. Carpineto and Romano [1] conduct a survey on the automatic query expansion (AQE) mechanisms applied to information retrieval. Rocchio's expansion is one of such mechanisms which was adapted by earlier studies [6, 10] in the context of software engineering. We consider this mechanism as the baseline reformulation technique, and compared with it using experiments (Section 4.3). Yuan et al. [24] first apply contextual words from Stack Overflow questions and answers for identifying semantically similar software-specific words. While they introduce the idea, we successfully adapt that idea for a software maintenance task, i.e., query reformulation for concept location. From a technical point of view, we collect candidate query terms opportunistically not only from project source code but also from questions of Stack Overflow, and determine their relevance to the initial query using crowdsourced knowledge (i.e., adjacency list database). We then apply the most relevant terms from both source code and Stack Overflow to query reformulation, and such methodology was not applied yet by any existing studies.

6. CONCLUSION AND FUTURE WORK

Studies show that software developers face difficulties in preparing an appropriate search query from a change request during concept location. In this paper, we propose a novel technique–QUICKAR–that automatically suggests effective reformulations for an initial query by leveraging the crowd generated knowledge from Stack Overflow. The technique collects candidate query terms from both project source and questions of Stack Overflow, and then determines their ap-

plicability for the reformulated query by applying the crowdsourced knowledge. Experiments using 510 change requests from two software systems suggest that our technique can improve or preserve the quality of 76% of the baseline queries which is promising. Comparison with one baseline technique also validates our preliminary findings. While the preliminary findings are promising, further experiments and investigations are warranted.

REFERENCES

[1] C. Carpineto and G. Romano. A Survey of Automatic Query Expansion in Information Retrieval. *ACM Comput. Surv.*, 44 (1):1:1–1:50, 2012.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature Location in Source Code: a Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[3] L. Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23, 2000.

[4] L. Favre. Modernizing Software & System Engineering Processes. In *Proc. ICSENG*, pages 442–447, 2008.

[5] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The Vocabulary Problem in Human-system Communication. *Commun. ACM*, 30(11):964–971, 1987.

[6] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the Use of Relevance Feedback in IR-based Concept Location. In *Proc. ICSM*, pages 351–360, 2009.

[7] S. Haiduc and A. Marcus. On the Effect of the Query in IR-based Concept Location. In *Proc. ICPC*, pages 234–237, June 2011.

[8] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus. Automatic Query Performance Assessment during the Retrieval of Software Artifacts. In *Proc. ASE*, pages 90–99, 2012.

[9] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia. Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks. In *Proc. ICSE*, pages 1273–1276, 2012.

[10] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic Query Reformulations for Text Retrieval in Software Engineering. In *Proc. ICSE*, pages 842–851, 2013.

[11] S. Haiduc, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus. Query Quality Prediction and Reformulation for Source Code Search: The Refoqus Tool. In *Proc. ICSE*, pages 1307–1310, 2013.

[12] M.J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. Automatically Mining Software-based, Semantically-Similar Words from Comment-Code Mappings. In *Proc. MSR*, pages 377–386, 2013.

[13] K. Kevic and T. Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.

[14] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann. Design Lessons from the Fastest Q & A Site in the West. In *Proc. CHI*, pages 2857–2866, 2011.

[15] R. Mihalcea and P. Tarau. TextRank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.

[16] F. Moreno-Seco, L. Mico, and J. Oncina. A New Classification Rule based on Nearest Neighbour Search. In *Proc. ICPR*, pages 408–411, 2004.

[17] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What Makes a Good Code Example?: A Study of Programming Q & A in Stack Overflow. In *Proc. ICSM*, pages 25–34, 2012.

[18] L. Ponzanelli, A. Bacchelli, and M. Lanza. Seahawk: Stack Overflow in the IDE. In *Proc. ICSE*, pages 1295–1298, 2013.

[19] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.

[20] M. M. Rahman, C. K. Roy, and D. Lo. RACK: Automatic API Recommendation using Crowdsourced Knowledge. In *Proc. SANER*, pages 349–359, 2016.

[21] P. C. Rigby and M.P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.

[22] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *Proc. ICPC*, pages 123–132, 2008.

[23] J. Yang and L. Tan. Inferring Semantically Related Words from Software Context. In *Proc. MSR*, pages 161–170, 2012.

[24] T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.