Supporting Code Search with Context-Aware, Analytics-Driven, Effective Query Reformulation

Mohammad Masudur Rahman
Department of Computer Science, University of Saskatchewan, Canada
Homepage: http://www.usask.ca/∼masud.rahman
Adviser: Chanchal K. Roy
masud.rahman@usask.ca

Abstract—**Software developers often experience difficulties in preparing appropriate queries for code search. Recent finding has suggested that developers fail to choose the right search keywords from an issue report for 88% of times. Thus, despite a number of earlier studies, automatic reformulation of queries for the code search is an open problem which warrants further investigations. In this dissertation work, we hypothesize that code search could be improved by adopting appropriate term weighting, context-awareness and data-analytics in query reformulation. We ask three research questions to evaluate the hypothesis, and then conduct six studies to answer these questions. Our proposed approaches improve code search by incorporating (1) novel, appropriate keyword selection algorithms, (2) context-awareness, (3) crowdsourced knowledge from Stack Overflow, and (4) large-scale data analytics into the query reformulation process.**

I. Introduction

Changes are inevitable in a software system. Once a software product is released, the customers (a.k.a., users) often (1) report software bugs encountered in the product, and/or (2) request for new software features. According to Cambridge University researchers, software bugs cost about **$312 billion** per year globally [1]. They also consume almost **50%** of the total development time and efforts. Addition of new software features to an existing system is also responsible for **60%** of the total maintenance costs [11]. Thus, traditional ad hoc practices for software debugging and other maintenance tasks are *not cost-effective* and *not sustainable* [11].

Software users are presumably *layman*. They generally describe their requirements in plain texts as an issue report (a.k.a., bug report). In order to satisfy customer requirements (e.g., fixing bugs, feature addition), a developer must locate the source code that needs to be changed or reused. Such code could be found either in the local codebase of a software system or even in the online software repositories (e.g., SourceForge, GitHub). When performing code level changes to an existing system, a developer first chooses a few important keywords (or concepts) from an issue report. Then she attempts to locate the source code entities (e.g., classes, methods) that deal with such concepts using a search operation. Unfortunately, preparing an appropriate search query is highly challenging even for the experienced developers [19, 24, 42]. According to Kevic and Fritz [19], developers fail to choose the right search keywords from a bug report for **88%** of times. An ad hoc alternative solution could be the use of *whole texts* from the bug report as a search query. Unfortunately, according to existing evidence [8], such texts also make poor queries. Besides local codebase, developers also frequently look for relevant code on the web and spend about **19%** of their time [5]. When searching on the web, developers often use generic natural language queries as a standard practice. However, such queries are also not sufficient enough for the code search, and almost **73%** of them need further expansion [34, 36]. Thus, appropriate query preparation for the code search is highly challenging regardless of the search contexts, and automated supports for query reformulation are in high demand.

Although there have been a number of studies that provide query reformulation supports for various *code search oriented maintenance tasks* such as *concept/concern location* [10, 12, 13, 15, 23] and *bug localization* [9, 39], they might (1) fail to determine keyword importance accurately, (2) suffer from noisy and poor bug reports, and (3) fail to efficiently use the resources at hand (e.g., bug reports, source code). There exist other studies [7, 26, 38, 41, 45] that automatically reformulate queries for *Internet-scale code search*. Unfortunately, they might also be affected badly by the low quality of given queries. Thus, automated reformulation of query for the code search is an open problem (regardless of problem contexts) that warrants further investigations. We thus hypothesize:

Through the adoption of appropriate term weighting, document contexts, crowdsourced knowledge, and large-scale data analytics in the query reformulation process, we can significantly improve a given query intended for code search.

Research Questions: In order to evaluate our hypothesis, we ask and attempt to answer three broad research questions in this dissertation work as follows:

- **$\mathbf{RQ_1}$:** How do traditional approaches (e.g., TF, TF-IDF) perform in extracting appropriate keywords from a bug report? Are they sufficient enough for keyword selection from a source code document? (**Study-1**, **Study-2**)
- **$\mathbf{RQ_2}$:** Does adoption of document contexts in the query reformulation make any difference? (**Study-2**, **Study-3**)
- **$\mathbf{RQ_3}$:** Does adoption of crowdsourced resources or data analytics derived from them in query reformulation help improve a given query? (**Study-4**, **Study-5**, **Study-6**)

Expected Contributions: The dissertation work in this paper presents six studies that answer three research questions

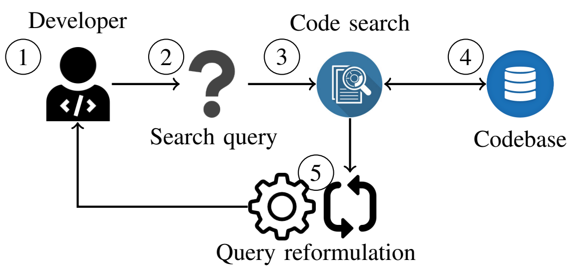

Fig. 1. Automated query reformulation intended for code search

above with empirical evidences, and then offers a suite of novel tools/techniques for improving query reformulations intended for the code search. In particular, we make three contributions to the literature as follows:

(a) **Supporting query reformulation with appropriate search keyword selection:** We propose STRICT [30] (**Study-1**) and ACER [29] (**Study-2**) for collecting appropriate search keywords from regular texts and from source code respectively. Both approaches outperformed the traditional alternatives (e.g., TF, TF-IDF [19]) in identifying important keywords and thus, in turn improved upon baseline queries significantly (e.g., 10%–**30**% [29]).

(b) **Supporting query reformulation with document contexts and structured entities:** We propose BLIZZARD [31] (**Study-3**) and ACER [29] (**Study-2**) that capture meaningful structured entities (e.g., stack traces, method signatures) from bug reports and from source documents respectively, and then leverage them in the query reformulation. Both approaches improve upon the baseline queries for code search significantly in terms of accuracy (e.g., **56%** [31]) and precision (e.g., **62%** [31]).

(c) **Supporting query reformulation with crowdsourced knowledge and data analytics:** We also propose RACK [33, 35] (**Study-4**), NLP2API [32] (**Study-5**) and BLADER (**Study-6**) that exploit **1.40** million Q & A threads of Stack Overflow and the data-analytics derived from them, and then complement poor queries with meaningful keywords. They expand a given query with either relevant API classes from Stack Overflow or relevant keywords from the project source code, and thus, improve upon the baseline accuracy by **37%** [32].

II. Background & Related Work

A. Background

Reformulation of a search query involves its (1) *expansion* with appropriate keywords, (2) *reduction* by discarding the noisy words, or (3) *replacement* with an alternative query [13]. Fig. 1 shows a typical scenario of automated query reformulation for code search. Let us assume that a developer, *Sam*, submits a query to the code search engine (e.g., *Lucene* [2]) (Steps 1–3). Unfortunately, the retrieved results are not relevant (Steps 3–4). Now, our tool captures this query, the retrieved results and other third party resources (e.g., Stack Overflow). It analyses them carefully, and then suggests a reformulated version of the query to the developer, *Sam* (Steps 5, 1). This iterative process continues until *Sam* finds out the right code that needs to be changed or could be reused.

B. Related Work

Keyword Selection for Query Reformulation: Selection of appropriate keywords is an important step of query reformulation. Kevic and Fritz [19] employ four lightweight heuristics such as *part of speech*, *notation*, *frequency* and *position* of a term within an issue report, and determine the term's eligibility as a search keyword. Unfortunately, their provided regression model was neither generalizable nor cross-validated.

Several existing studies [10, 13, 19, 28] capture apparently relevant source code documents for a given query. Then they analyse them with Rocchio's method and expand the given query. These studies use TF-IDF [18] as the *de facto* term weighting method for keyword selection. However, TF-IDF was originally designed for unstructured regular texts (e.g., news article). Thus, it might be neither appropriate nor sufficient enough for delivering appropriate search keywords from a source code document which is full of structures.

A few studies [25, 37] make use of natural language thesauri such as WordNet [27] for collecting synonyms and semantically similar words. Then they expand a given query with them. However, existing evidence [40] suggests that the same word could bear *two different semantics* when used in the regular texts (e.g., news article) and when used in the source code. Thus, keywords taken from WordNet might not be effective for query reformulation intended for code search.

Several studies [16, 20, 43] mine software repositories, extract semantically similar word pairs from them, and then use such pairs for query reformulation. That is, these studies require the source code to be well documented with high quality code comments. Thus, they might not perform well if the software repositories are poorly documented.

Document Contexts in Query Reformulation: Hill et al. [15] leverage the context of query keywords within a source code document for query reformulation. They locate query keywords in various structured entities such as *method signatures* and *field signatures*, and then suggest their co-occurring terms as candidates for query reformulation. Sisman and Kak [39] define keyword context using *spatial code proximity*, and also choose reformulation terms from the context. Thus, both approaches above exploit term co-occurrences in the query reformulation. However, semantically similar/relevant terms might always not co-occur [17, 21]. Thus, simple co-occurrence frequency might not be sufficient enough for selecting appropriate search keywords for query reformulation.

Chaparro et al. [9] divide the texts of an ideal bug report (given query) into three components– Expected Behaviour (EB), Observed Behaviour (OB), and Steps to Reproduce (S2R). Then they use only OB part as a reduced/reformulated version of the given query. However, their work is empirical in nature, which warrants significant manual analysis. Furthermore, the use of OB part as the reformulated query might not be theoretically justified. According to Kevic and Fritz [19], simultaneous occurrences of a term both in *title* and in *description* of a bug report indicates its salience as a code search keyword. Unfortunately, such hypothesis was not properly evaluated using substantial experiments.

Data Analytics for Query Reformulation: Ye et al. [44] first incorporate *semantic similarity* into Information Retrieval-based bug localization where they learn the *word embeddings* from API documentations, tutorials and wiki pages using *skip-gram algorithm*. However, such documentations might always not be available for every project under study. Zhang et al. [45] learn the word embeddings from a large corpus of $\approx$25K software projects, and then suggest semantically related API classes against a given programming task (given query). Their approach requires the presence of both query keywords and API classes in the source code. However, source code is often scarce in vocabulary and rich in structures [14]. Thus, their approach might not perform well if the corpus projects lack sufficient vocabulary. Lin et al. [23] represent each document (or query) as a collection of weighted API classes, and then leverage conceptual knowledge mined from project source code for retrieving the API learning resources. Since the API representation step relies on *lexical similarity*, their approach might also suffer from poor queries. A few studies [22, 28, 38] mine crowdsourced knowledge from Stack Overflow, and then complement the given queries with relevant program elements, user tags or technical words. Unfortunately, Stack Overflow could be noisy, and TF-IDF alone might not be sufficient enough to remove all noisy elements. Thus, their reformulated queries might also suffer from low performance.

III. Dissertation Work

To date, six different studies have been conducted in this dissertation research namely STRICT [30], ACER [29], BLIZZARD [31], RACK [33, 35], NLP2API [32] and BLADER. We categorize them into three groups based on their contribution aspects and research methodologies as follows:

(a) Answering RQ$_1$: Supporting Query Reformulation with Appropriate Search Keyword Selection: TF-IDF [18] determines the importance of a word based on its isolated frequency, and overlooks the presence of other words within a document. However, co-occurring words often depend on each other for their complete semantics. For example, the phrase *"code search"* conveys a different semantic than that of *"code"* or *"search"* alone. TF-IDF fails to capture such aspect during keyword selection for code search. We performed **Study-1** and **Study-2** in order to address this issue as follows:

Study-1 (a.k.a., **STRICT** [30]) analyses the contents of an issue report, and returns a list of appropriate keywords for code search. Unlike TF-IDF, we capture (1) *co-occurrences* [3] and (2) *syntactic dependencies* [3] among the words from an issue report. Then we represent each of these relationships as the connecting edges of a graph where the unique words from the report are being the nodes. Thus, each issue report is transformed into two different graphs based on word co-occurrences and syntactic dependencies. Then we employ two popular term weighting algorithms–*TextRank* and *POSRank*–on these graphs, and calculate the weight of each word [3]. Finally, weights of each word from these two graphs are combined, and only Top-K weighted keywords are suggested as a search query for concept location (i.e., local code search).

Study-2 (a.k.a., **ACER** [29]) analyses a list of source documents retrieved by a given query, and then suggests a list of appropriate keywords as the reformulated query. We first collect structured entities such as *method signatures*, *constructor signatures* and *field signatures* from each document, and transform them into phrase like structures with natural language preprocessing. Then, unlike TF-IDF [18], we leverage the *co-occurrences* among the terms within each phrase, and construct a source term graph. Then we employ *PageRank* [6] on this graph and calculate the weight of each of the terms. Finally, Top-K weighted terms from the graph are suggested as a reformulated version of the given query.

(b) Answering RQ$_2$: Supporting Query Reformulation with Document Contexts & Structured Entities: Two of our studies–**Study-2** and **Study-3**– employ document contexts and structured entities in the query reformulation.

Study-2 treats each source document as a connected network of *methods* and *fields* rather than plain texts [10, 13]. While each field refers to a unique attribute, method names disclose the actions available to the entity (i.e., class). The remaining parts of a source document are noisy since they deal with low level implementations [15]. We leverage these *salient contexts* (e.g., method signatures) from a source document, and construct multiple reformulation candidates. Then we apply *query difficulty analysis* [12] and *machine learning* on them, and suggest the best reformulation for a given query.

Study-3 (a.k.a., **BLIZZARD** [31]) deals with the contexts and structures from a bug report. Bug reports often contain various structured entities (e.g., *stack traces, method signatures*). We detect their presence using appropriate regular expressions, and categorize each bug report (given query) as either *noisy*, *rich* or *poor*. Noisy bug reports contain stack traces and regular texts. Rich bug reports contain both regular texts and program elements but no stack traces. On the contrary, poor bug reports contain only regular texts and no structured entities. Once a bug report is categorized, we apply such reformulation algorithm that is *appropriate* for the category, and then deliver the best reformulation to the given query.

(c) Answering RQ$_3$: Supporting Query Reformulation with Crowdsourced Knowledge and Data Analytics: Three of our studies–**Study-4**, **Study-5** and **Study-6**–exploit crowdsourced knowledge from Stack Overflow Q & A site and the data analytics derived from them in the query reformulation.

Study-4 (a.k.a., **RACK** [33, 35]) accepts a programming task description (i.e., given query), and then returns a list of API classes relevant to the task. We capture co-occurrences of *query keywords* (in question title) and *API classes* (in the accepted answer) within the same Q & A pairs, analyse them using three different heuristics, and then suggest Top-K relevant API classes. Unlike earlier approaches [7, 26], RACK does not rely on lexical similarity between query and API classes for query expansion. Thus, it has high potential for overcoming the *vocabulary mismatch problem* in code search.

Study-5 (a.k.a., **NLP2API** [32]) combines crowdsourced knowledge from Stack Overflow and large-scale data analytics in the query reformulation. We first collect Top-N candidate API classes from relevant Q & A threads for a given query. Then we determine their relevance by calculating (1) *Borda count* and (2) *semantic distance* of each API class from the given query. We then suggest Top-K relevant API classes as the reformulated query. Unlike earlier studies [44, 45], our corpus –Stack Overflow– contains sufficient regular texts and source code, and thus offers an extra-large vocabulary.

Study-6 (a.k.a., **BLADER**) also leverages large-scale data analytics in the query reformulation like NLP2API. However, it adopts the concept of *clustering tendency* rather than simple *semantic distance* between a given query and the reformulation candidates. First, we construct a *semantic hyperspace* by learning word embeddings from Stack Overflow corpus using *FastText* [4]. Second, we develop multiple reformulation candidates, and determine their clustering tendencies (e.g., *Hopkins statistic*, *Polygon area*) with the given query. Third, we deliver the best reformulated query using machine learning.

Evaluation Methodology: This dissertation addresses code search problem in two different working contexts– (1) *local code search* (e.g., bug localization, concept location), and (2) *Internet-scale code search*. In the case of concept/bug localization (Study-1, Study-2, Study-3, Study-6), we use **5000+** bug reports from eight systems for experiments. We construct *ground truth* for each bug report (query) by analysing corresponding bug fixing commits from version control history at GitHub. In the case of Internet-scale code search, we evaluate our studies (Study-4, Study-5) using **300+** queries collected from four programming tutorial sites (e.g., KodeJava, JavaDB, Java2s, CodeJava). We use the standard performance metrics such as Hit@K, MAP, MRR, Recall, and QE.

IV. Timeline & Future Work

The author of this paper is a fifth year PhD student who has passed the qualifying exam in May, 2018, and now is preparing for the comprehensive exam. Five out of six studies have already been accepted and published in top venues of Software Engineering (e.g., ICSE, ESEC/FSE, ASE, EMSE, and ICSME), and the sixth study is ready for ESEC/FSE 2019 submission. Currently, the journal version of Study-1 is under TSE review. The author hopes to graduate by August, 2019.

References

[1] Cost of software debugging. URL https://goo.gl/okoj21.

[2] Apache Lucene Core, 2019. URL https://lucene.apache.org/core.

[3] R Blanco and C Lioma. Graph-based Term Weighting for Information Retrieval. *Inf. Retr.*, 15(1):54–92, 2012.

[4] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[5] J Brandt, P J Guo, J Lewenstein, M Dontcheva, and S R Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.

[6] S Brin and L Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, 1998.

[7] W Chan, H Cheng, and D Lo. Searching Connected API Subgraph via Text Phrases. In *Proc. FSE*, pages 10:1—-10:11, 2012.

[8] O Chaparro and A Marcus. On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance. In *Proc. ICSE-C*, pages 716–718, 2016.

[9] O Chaparro, J M Florez, and A Marcus. Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization. In *Proc. ICSME*, pages 376–387, 2017.

[10] 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.

[11] R. L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–111, 2001.

[12] 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.

[13] 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.

[14] V. J. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proc. ESEC/FSE*, pages 763–773, 2017.

[15] E Hill, L Pollock, and K Vijay-Shanker. Automatically Capturing Source Code Context of NL-queries for Software Maintenance and Reuse. In *Proc. ICSE*, pages 232–242, 2009.

[16] 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.

[17] S. F. Hussain and G. Bisson. *Text Categorization Using Word Similarities Based on Higher Order Co-occurrences*, pages 1–12. 2010.

[18] K S Jones. A Statistical Interpretation Of Term Specificity And Its Application In Retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

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

[20] K Kevic and T Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.

[21] B. Lemaire and G. Denhire. Effects of high-order co-occurrences on word semantic similarities. *CoRR*, 2008.

[22] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin. Query reformulation by leveraging crowd wisdom for scenario-based software search. In *Proc. Internetware*, pages 36–44, 2016.

[23] Z. Lin, Y. Zou, J. Zhao, and B. Xie. Improving software text retrieval using conceptual knowledge in source code. In *Proc. ASE*, pages 123–134, 2017.

[24] D Liu, A Marcus, D Poshyvanyk, and V Rajlich. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proc. ASE*, pages 234–243, 2007.

[25] Meili Lu, X. Sun, S. Wang, D. Lo, and Yucong Duan. Query expansion via wordnet for effective code search. In *Proc. SANER*, pages 545–549, 2015.

[26] C McMillan, M Grechanik, D Poshyvanyk, Q Xie, and C Fu. Portfolio: Finding Relevant Functions and their Usage. In *Proc. ICSE*, pages 111–120, 2011.

[27] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38 (11):39–41, 1995.

[28] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *TSC*, 9(5):771–783, 2016.

[29] M M Rahman and C K Roy. Improved Query Reformulation for Concept Location using CodeRank and Document Structures. In *Proc. ASE*, pages 428–439, 2017.

[30] M M Rahman and C K Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.

[31] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proc. ESEC/FSE*, pages 621–632, 2018.

[32] M. M. Rahman and C. K. Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *Proc. ICSME*, pages 516–527, 2018.

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

[34] M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K T. Stolee, and Baishakhi Ray. Evaluating how developers use general-purpose web-search for code retrieval. In *Proc. MSR*, pages 465–475, 2018.

[35] M. M. Rahman, C. K. Roy, and D. Lo. Automatic query reformulation for code search using crowdsourced knowledge. *EMSE*, page 56, 2018.

[36] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: A case study. In *Proc. ESEC/FSE*, pages 191–201, 2015.

[37] D Shepherd, Z P Fry, E Hill, L Pollock, and K Vijay-Shanker. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proc. ASOD*, pages 212–224, 2007.

[38] R. Sirres, T. F. Bissyandé, D. Kim, D. Lo, J. Klein, K. Kim, and Y. L. Traon. Augmenting and structuring user queries to support efficient free-form code search. *EMSE*, pages 2622–2654, 2018.

[39] B Sisman and A C Kak. Assisting Code Search with Automatic Query Reformulation for Bug Localization. In *Proc. MSR*, pages 309–318, 2013.

[40] 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.

[41] K. T. Stolee, S. Elbaum, and M. B. Dwyer. Code search with input/output queries: Generalizing, ranking, and assessment. *JSS*, 116(C):35–48, 2016.

[42] Q Wang, C Parnin, and A Orso. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proc. ISSTA*, pages 1–11, 2015.

[43] J. Yang and L. Tan. Swordnet: Inferring semantically related words from software context. *EMSE*, 19(6):1856–1886, 2014.

[44] X Ye, H Shen, X Ma, R Bunescu, and C Liu. From Word Embeddings to Document Similarities for Improved Information Retrieval in Software Engineering. In *Proc. ICSE*, pages 404–415, 2016.

[45] F. Zhang, H. Niu, I. Keivanloo, and Y. Zou. Expanding queries for code search using semantically related api class-names. *TSE*, 44(11):1070–1082, 2018.