Relationship-Aware Code Search for JavaScript Frameworks

Xuan Li[1], Zerui Wang[1], Qianxiang Wang[1], Shoumeng Yan[2], Tao Xie[3], Hong Mei[1]
[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing, China
[2]Intel China Research Center, Beijing, China
[3]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
{lixuan12, wangzr13, wqx, meih}@sei.pku.edu.cn, shoumeng.yan@intel.com
taoxie@illinois.edu

ABSTRACT
JavaScript frameworks, such as jQuery, are widely used for developing web applications. To facilitate using these JavaScript frameworks to implement a feature (e.g., functionality), a large number of programmers often search for code snippets that implement the same or similar feature. However, existing code search approaches tend to be ineffective, without taking into account the fact that JavaScript code snippets often implement a feature based on various relationships (e.g., sequencing, condition, and callback relationships) among the invoked framework API methods. To address this issue, we present a novel Relationship-Aware Code Search (RACS) approach for finding code snippets that use JavaScript frameworks to implement a specific feature. In advance, RACS collects a large number of code snippets that use some JavaScript frameworks, mines API usage patterns from the collected code snippets, and represents the mined patterns with method call relationship (MCR) graphs, which capture framework API methods' signatures and their relationships. Given a natural language (NL) search query issued by a programmer, RACS conducts NL processing to automatically extract an action relationship (AR) graph, which consists of actions and their relationships inferred from the query. In this way, RACS reduces code search to the problem of graph search: finding similar MCR graphs for a given AR graph. We conduct evaluations against representative real-world jQuery questions posted on Stack Overflow, based on 308,294 code snippets collected from over 81,540 files on the Internet. The evaluation results show the effectiveness of RACS: the top 1 snippet produced by RACS matches the target code snippet for 46% questions, compared to only 4% achieved by a relationship-oblivious approach.

CCS Concepts
• **Software and its engineering → Software creation and management;** • **Information system → Information retrieval;**

Keywords
Code search; JavaScript code mining; natural language processing

FSE'16, November 13-19, 2016, Seattle, WA, USA
 ISBN 978-1-4503-4218-6/16/11$15.00
DOI: http://dx.doi.org/10.1145/2950290.2950341

1. INTRODUCTION
JavaScript frameworks are widely used for developing web applications. A recent survey [16] has shown that 72.5% of the top 10 million websites use JavaScript frameworks, such as jQuery, MooTools, Prototype, YUI, and ExtJS. Meanwhile, among all these frameworks, jQuery has a share of 95.9%. When using these JavaScript frameworks, a large number of programmers are often in great need of help. For example, 9.5% of 11,245,425 questions in Stack Overflow (the most well-known programming Q&A website) are tagged with "JavaScript", which is the top 1 tag.

When using these JavaScript frameworks to implement a feature (e.g., functionality), programmers can benefit from existing code snippets that implement the same or similar feature [1]. To search for such code snippets, the programmers can read Application Programming Interface (API) documentation or tutorials, post questions on Q&A websites [2], use code search engines and so on. However, these existing approaches of code search face various limitations for JavaScript frameworks. For example, API documentation contains only a very limited number of hand-crafted code snippets. Existing code search engines, such as Ohloh Code (https://code.openhub.net/) and Krugle (http://www.krugle.com/), mainly use text similarity to find code snippets in open source code repositories (e.g., GitHub, SourceForge), and tend to be inaccurate. Recent research contributes new approaches that leverage code analysis and code mining, e.g., PARSEWeb [5], MAPO [6], SNIFF [7]; they take into account code characteristics, such as API usage patterns [6] and encoded code patterns [8]. However, none of these approaches considers the characteristics of JavaScript code snippets or search queries related to using a JavaScript framework API.

Searching for code snippets using JavaScript frameworks has three main unique characteristics. First, in JavaScript, relationships between method calls are complex, beyond sequencing relationships (e.g., `open()` should be invoked before `read()`) among method calls, as commonly captured by existing approaches [5][6][22][24]. For example, many API methods in JavaScript frameworks are asynchronous: although the call sites of these asynchronous methods are sequentially listed in a code snippet, there can be a large number of concurrent executions of these methods at runtime. In addition, callback is often used in JavaScript code to enforce strict execution order of some method calls. For example, Lines 4-7 in the lower part of Figure 1 define an anonymous function, in which API methods `$('#loader_img')` (jQuery uses "$" as a shortcut for "jQuery") and `hide()` are called (Line 6). This anonymous function is passed as a callback parameter of the method `load()` (Line 4). The code's runtime behavior is that these two API methods are called only after the method `load()` has completed.

Second, JavaScript is mainly used for client-side scripting in web browsers. Many typical search queries for JavaScript framework API usage describe user interaction, browser control, asynchronous

communication, and altering of a displayed document content. Thus these queries usually consist of simple actions, conjuncted with relationship-describing words (e.g., "when" and "after"). For example, the underlined sentence in the upper part of Figure 1 illustrates such a query, which includes multiple actions: "*show a busy image*", "*image is downloaded*", and "*busy image is removed*". Table 2 in Section 4.1 provides more query examples from Stack Overflow. Many short descriptions for a simple action can be mapped to the corresponding action-implementing methods in JavaScript frameworks, by leveraging their API documentation. In addition, the conjunction words between short descriptions (e.g., "when" as shown in Figure 1) reflect the relationships of these actions. These relationship-describing words may also be mapped to aforementioned relationships among API method calls, "sequencing" and "callback", respectively.

Third, JavaScript frameworks such as jQuery are usually used to select and manipulate Document Object Model (DOM) elements. The types (e.g., "*img*", "*div*") and attributes of DOM elements (e.g., "*class*", "*id*") are usually defined in HTML code. API methods in JavaScript code use CSS selectors (e.g. ".child", "#option") to select DOM elements and are generally applicable to manipulate various types of DOM elements without directly referring to these elements' types. Therefore, it is undesirable to directly use elements' types that appear in an NL search query (e.g., "*div*" in the "*hide div*" query) to search code snippets for the target code snippet. Special care needs to be taken to process these elements' types in a search query before being used in code search.

Based on the observations of these unique characteristics, we propose a Relationship-Aware Code Search (RACS) approach for finding code snippets that use JavaScript frameworks to implement a specific feature, being described in the given search query. RACS emphasizes the utility of semantic information, especially the relationships between API method calls in code snippets and relationships between actions in search queries. RACS abstracts a code snippet as an API method call relationship (MCR) graph, which consists of the signatures of the API methods invoked in the code snippet along with the relationships among these methods. Given a natural language (NL) search query, RACS conducts NL processing to automatically abstract the query to an action relationship (AR) graph. In this way, RACS reduces code search to the problem of graph search: finding similar MCR graphs for a given AR graph.

This paper makes the following main contributions:

- The first approach for finding relevant JavaScript-framework-based code snippets given a search query in NL.
- A technique for mining framework API usage patterns expressed formally as MCR graphs from large-scale JavaScript code snippets.
- A technique for abstracting an NL search query to an AR graph.
- A technique for reducing the code search problem as a graph search problem.
- Evaluations conducted against representative real-world jQuery questions (posted on Stack Overflow), based on 308,294 code snippets collected from over 81,540 files on the Internet. The evaluation results show the effectiveness of RACS: the top 1 snippet produced by RACS matches the target code snippet for 46% questions, compared to only 4% achieved by a relationship-oblivious approach (existing state-of-the-art code search approaches [7][31] are relationship-oblivious approaches).

The rest of the paper is organized as follows. Section 2 explains our RACS approach through an example. Section 3 elaborates RACS. Section 4 discusses evaluation results. Section 5 discusses the applicability and limitations of RACS. Section 6 presents related work. Finally, Section 7 concludes this paper.

2. MOTIVATING EXAMPLE

In this section, using an example, we elaborate characteristics of both JavaScript code snippets and search queries related to using a JavaScript framework API. Figure 1 shows a real-world question (the upper part) and one accepted answer (the lower part) from Stack Overflow. This question describes a typical scenario in developing web applications. The underlined sentence is an NL description for a feature implemented in JavaScript or jQuery. The accepted answer contains a code snippet implementing the feature with jQuery.

Stack Overflow Question and Description

How to display loading image while actual image is downloading

Some time images take some time to render in the browser. I want show a busy image while the actual image is downloading, and when image is downloaded, the busy image is removed and actual image is be shown there. How can I do this with JQuery or any javascript?

Accepted Answer

You can do something like this:

```
1| // show loading image
2| $('#loader_img').show();
3| // main image loaded ?
4| $('#main_img').load(function(){
5| // hide/remove the loading image
6| $('#loader_img').hide();
7| });
```

You assign `load` event to the image which fires when image has finished loading. Before that, you can show your loader image.

From an earlier version of http://stackoverflow.com/questions/4635388[1]

Figure 1. Example from Stack Overflow

The code snippet in the accepted answer shows a callback relationship in JavaScript code (Lines 4-7). An anonymous callback function is defined and passed as a parameter of the jQuery API method call `load()`. Two other jQuery API methods, `$('#loader_img')` and `hide()` (Line 6), are called inside the anonymous function. Existing approaches [5][6][22][24] mainly extract method-call sequences as the abstract representation of the code snippet and apply mining algorithms on the sequences. In this code snippet, Line 4 with the callback not only represents the occurrence order in the code snippet, but also reflects the strict execution order for asynchronous methods (as explained by the comments in Lines 3 and 5). RACS analyzes the JavaScript code snippet, extracts method signatures for the API methods invoked in the code snippet, and identifies different relationships between the method calls (see Section 3.1 for details). In this example, `show()` and `load()` have a sequencing relationship, while `load()` and `hide()` have a callback relationship, enforcing a strict order. We represent the signatures of the invoked methods and their relationships as an API method call relationship (MCR) graph, the abstract representation of the code snippet.

In the upper part of Figure 1, the underlined sentence is an NL description of a feature. The feature consists of multiple actions in each clause ("*show a busy image*", "*image is downloaded*", and "*busy image is removed*"), and there are structural relationships between clauses (implied by relationship-describing words "when" and "after"). No existing approach considers such structural information. In some existing code search tools, the users need to manually extract query terms based on the NL description. For example, in Keivanloo et al.'s approach [8], the users manually select candidate terms from Koder's query log dataset. Then the users manually map the description "successfully login and logout"

[1] The latest version of the accepted answer includes the updated code being compatible with a more recent version of jQuery.

to query term "FtpClient". Programmers with little knowledge of the names of the target framework API methods can hardly write a query as specific terms. Some other approaches, such as SNIFF [7], directly take short descriptions as the query after preliminary preprocessing, e.g., stop-word removal and stemming. Our RACS approach uses NL processing to extract semantic descriptions for actions in each clause. RACS analyzes the sentence structure and identifies different relationships between actions (see Section 3.2 for details). In addition, RACS constructs a mapping between a method signature and its API documentation description, and uses this mapping to connect a given action description to its corresponding API method. For a given action description, RACS seeks to find a matching API documentation description and then the method signature. The matching between an action description and API documentation description is based on text semantic similarity, instead of keyword matching, to address NL complications.

3. APPROACH

Given an NL search query for snippets using a JavaScript framework API, RACS returns multiple highly relevant code snippets. As shown in Figure 2, RACS is composed of three major components that conduct three steps:

(1) Mining API usage patterns. This component mines JavaScript code snippets for framework API usage patterns, and represents the patterns as Method Call Relationship (MCR) graphs. This process is offline.

(2) Abstracting NL query. This component analyzes the given NL query's description and generates an Action Relationship (AR) graph to reflect the user's search intention.

(3) Searching snippets. This component searches all the MCR graphs for the top ones that match the AR graph (produced by Step 2). This component leverages the API documentation description to bridge the NL query and the API methods invoked in code snippets. The component then presents to the user the ranked code snippets associated with the top matched MCR graphs.

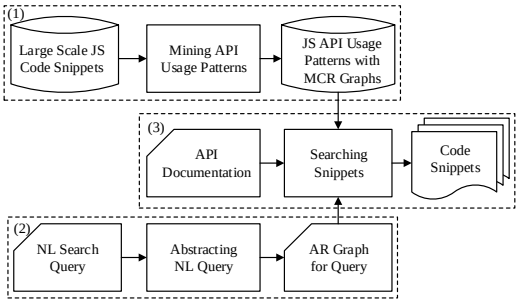

Figure 2. Overview of RACS

RACS emphasizes both relationships between statements in programs and relationships between sentences in an NL query. Based on the observations of the JavaScript language and search queries for JavaScript frameworks, RACS focuses on three main relationships: sequencing, callback, and condition.

Before presenting RACS in detail, we give major definitions of important concepts used in the rest of this paper.

Definition 1. Method Call Relationship (MCR) Graph for a code snippet

A method call relationship (MCR) graph for a code snippet is a Directed Acyclic Graph (DAG) as a tuple $< M, R >$, where

- M is a non-empty vertex set represented as $\{m_1, m_2, \dots, m_x\}$. Every element in M is a method signature including its name and parameter type list.
- R is an edge set represented as $\{r_1, r_2, \dots, r_y\}$. Every element in R is a triple $< m_i, m_j, t >$, indcating that relationship t exists from vertex m_i to vertex m_j; t is one of the three relationships: sequencing, callback, and condition. In particular, the detailed meanings of $< m_i, m_j, sequencing >$, $< m_i, m_j, callback >$, $< m_i, m_j, condition >$ for JavaScript are further elaborated in Section 3.1.1.

An MCR graph is an abstract representation of a code snippet including one or more framework API method calls, focusing on essential behaviors involving these framework API method calls.

Definition 2. Action Relationship (AR) Graph for a query

An action relationship (AR) graph for a query is a DAG as a tuple <A,R>, where

- A is a non-empty vertex set represented as $\{a_1, a_2, \dots, a_n\}$. Every element in A is an action that implements a feature reflected by the query.
- R is an edge set represented as $\{r_1, r_2, \dots, r_m\}$. Every element in R is a triple $< a_i, a_j, t >$, indicating that relationship t exists from vertex a_i and vertex a_j; t is one of the three relationships: sequencing, callback, and condition. In particular, the detailed meanings of $< a_i, a_j, sequencing >$, $< a_i, a_j, callback >$, $< a_i, a_j, condition >$ for JavaScript are further elaborated in Section 3.2.3.

An AR graph is an abstract representation of an NL search query including one or more actions, focusing on essential behaviors involving these actions.

3.1 Mining API Usage Patterns

The component of mining API usage patterns consists of two sub-processes. First, it analyzes large-scale JavaScript code snippets, and extracts an MCR graph as an abstract representation of each snippet. Second, it analyzes each MCR graph, and groups the code snippets with the same MCR graph as one API usage pattern.

3.1.1 Abstracting Code Snippets

RACS constructs a snippet base from an initial JavaScript code base. In particular, from the JavaScript and HTML files collected in the initial JavaScript code base, RACS first extracts sequences of framework API methods being invoked in each JavaScript function in the files. Then each contiguous subsequence of such sequence forms a code snippet. For a function consisting of n API method calls, we obtain $n * (n + 1)/2$ snippets: n snippets each include 1 API method call, $n - 1$ snippets each include 2 API method calls, …, and 1 snippet includes n API method calls.

For each code snippet (in the snippet base), RACS analyzes its Abstract Syntax Tree (AST) and constructs an MCR graph. In our implementation, we use the Rhino JavaScript engine (https://www.mozilla.org/rhino/) for JavaScript code analysis. When visiting the AST nodes, RACS identifies framework API method calls (e.g., according to the list of API methods documented in the jQuery API documentation). Meanwhile, RACS identifies relationships among these API method calls according to relationships among AST nodes. Currently, RACS considers three common relationships in JavaScript: sequencing, callback, and condition. Figure 3a shows a code snippet involving all three kinds of relationships. Figure 3b shows the correspondent AST (simplified with leaf nodes and part of other non-critical nodes such as *block statement* being removed) of the code snippet listed in Figure 3a.

Sequencing relationship. If method B is called immediately after method A is called, there is a sequencing relationship from A to B, formally represented as a triple <A, B, sequencing>. In the AST, parent-child method call nodes are method chains, having a

sequencing relationship. For example, in Figure 3b, there is a sequencing relationship from method call `$('#loader_img')` (Node 1) to method call `show()` (Node 2) in one statement, and there is a sequencing relationship from `show()` (Node 2) to `$('#main_img')` in two statements.

Callback relationship. If method A is called via an anonymous function as its parameter and method B is a method called inside the anonymous function, there is a callback relationship from A to B, formally represented as a triple <A, B, callback>. In Figure 3b, there is a callback relationship from method call `load()` (Node 4) to method call `hide()`(Node 6).

Condition relationship. If method A appears in a predicate of a conditional statement, such as IfStatement, and method B is the first method called in one of the branches, then there is a condition relationship from A to B, formally represented as a triple <A, B, condition>. For a method C being called after B in the same conditional block, we do not record a condition relationship between A and C, but we do record a sequencing relationship between B and C. In Figure 3b, there is a condition relationship from method call `width()` (Node 8) to method call `show()` (Node 10).

```
$('#loader_img').show();
$('#main_img').load(function(){
        $('#loader_img').hide();
});
if($(window).width() < 960 ){
        $('#warning_img').show();
}
```

Figure 3a. Code snippet extended from code in Figure 1.

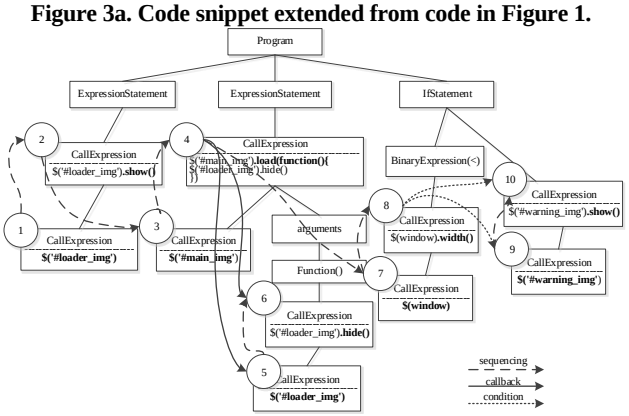

Figure 3b. Simplified AST for code snippet in Figure 3a.

In jQuery, method "`$`" (in nodes 1, 3, 5, and 7) is used for selecting elements by string selectors. All method calls with signature "`$(STRING)`" are considered to be semantically equivalent. "`$(STRING)`" often appears in code and provides limited semantics. Therefore, RACS ignores all method "`$`" calls to simplify the MCR graph. This process removes method calls with low degree of differentiation, sharing similar ideas with stop-word removal and TF-IDF. The MCR graph for the snippet in Figure 3a is shown in Figure 4.

3.1.2 Clustering Code Snippets

Many MCR graphs from different snippets are equivalent. To avoid unnecessary duplicated graph matching in the next section, and thus improve the querying speed, RACS clusters code snippets in its base according to their MCR graphs. To capture precise semantics of code snippets falling into the same cluster, we use a relatively conservative way of clustering: if two code snippets are abstracted to the same MCR graph, they fall into one cluster represented by the MCR graph.

After this step, we attain a large number of clusters represented as MCR graphs, reflecting different usage patterns of a JavaScript framework API. Each MCR graph (i.e., each cluster) is associated with one or more code snippets.

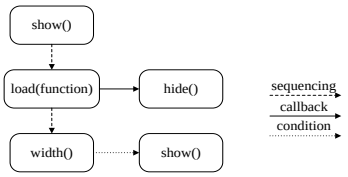

Figure 4. MCR graph for the example in Figure 3.

3.2 Abstracting NL Query

This section presents the techniques for extracting an AR graph that accurately captures essential semantics of the given NL query.

3.2.1 Preprocessing

The step of preprocessing accepts the given NL query's sentences and processes these sentences to get a more accurate result of NL analysis. In particular, this step performs two major tasks.

First, the step transforms some special identifiers to plain English text. Identifiers in an NL query may pose complication in NL analysis. For example, the character period ('`.`') in identifiers (e.g., "`jQuery.Ajax`") may be recognized as the end of a sentence, leading to wrong parsing (see Section 3.2.2 for detail). In addition, some identifiers are the combination of multiple English words (e.g., "`toggleClass`"), negatively influencing the calculation of sentence-semantic similarity in a later process (See Section 3.3.1 for detail). To deal with these complications, we replace the period (".") character with a space, split identifiers concatenated with hyphen ("-") or underscore ("_"), and split "camelCase" style identifiers following the JavaScript naming convention.

Second, the step transforms abbreviations into full terms (e.g., "attr" to "attribute") based on a domain-specific dictionary. The domain-specific dictionary contains the top 20 most frequent expansions in programs found in Hill's et al.'s research [30]. Besides a short list of abbreviations, we automatically extract terms from DOM/JavaScript documentation using Python scripts. The domain-specific dictionary includes a word list of DOM elements (e.g., "div", marked by <td title="Name"> in https://www.w3.org/TR/1999/REC-html401-19991224/index/elements.html) and JavaScript events (e.g., "mouseenter", marked by <a class="eventtype"> in https://www.w3.org/TR/DOM-Level-3-Events/#event-types).

3.2.2 Action Identification

The step of action identification uses NL processing techniques, such as POS (Parts Of Speech) tagging and parsing, to attain the structure of a given NL query sentence, and identifies the actions in the sentence.

POS tagging determines the part of speech (e.g., a singular or mass noun's POS tag is NN) for each word in the given sentence. Parsing determines the parse tree of the given sentence. Each word in the sentence is represented as a leaf node in the parse tree, and each grammatical unit (e.g., verb phrase) corresponds to a sub-tree. We use the Stanford Parser [19] to generate the POS tags and parse tree. RACS traverses the parse tree and identifies the actions in the given NL query. In the language structure, the description of an action consists of Verb Phrase (VP), Noun Phrase (NP), and optional Prepositional Phrase (PP). A sub-tree with this structure can be identified as one action. Figure 5 shows the parse tree of the NL query in Figure 1. RACS identifies five actions in this example (see the rectangle in Figure 5). Note that these actions are not the final elements of the action set; they may be modified or even discarded by the post-processing step described in Section 3.2.4.

3.2.3 Relationship Identification

The step of relationship identification identifies potential relationships among the actions, by mapping relationship-describing

words to the three aforementioned relationships. These relationship-describing words are the most commonly used in a lot of Stack Overflow questions' descriptions. For each adjacent action pair found in the given NL query's description, we check whether it corresponds to one of the following three relationships.

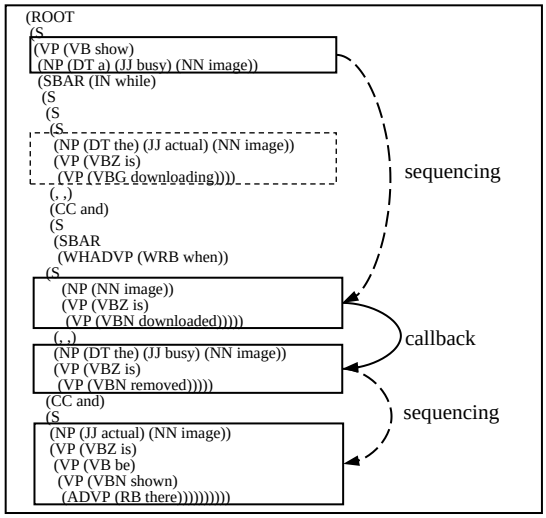

Figure 5. Parse Tree of NL query in Figure 1

Sequencing relationship. A sequencing relationship from action a_i to action a_j exists when two actions' descriptions are connected by a connecting word being a preposition ("before" and "after"), a conjunction ("and" and "then"), or a punctuation (comma, semicolon, and period). Note that the direction of the sequencing relationship is properly determined based on the semantics of the connecting word. Only the action after connecting word "after" is the starting point of the sequencing relationship. In other cases, the later action is the end point. An example of such relationship is in sentence "*add class 'checked' to element and fade in the element."*

Condition relationship. A condition relationship from action a_i to action a_j exists when two actions' descriptions are connected by a connecting word being a preposition ("if") indicating that whether action a_j will happen or not depends on the result of action a_i. The action after the connecting word is the starting point of the condition relationship. An example of such relationship is in sentence "*show a warning image if screen width is less than 960px."*

Callback relationship. A callback relationship from action a_i to action a_j exists when not only (1) two actions' descriptions are connected by a connecting word being a conjunction or a preposition indicating point-in-time ("when", "after", and "if"), but also (2) the action happening first, i.e., action a_i, should imply an event or describe a completion status; in other words, the description is supposed to contain a word from the word list for JavaScript events in the domain-specific dictionary (described in Section 3.2.1), or the POS of a verb in the description should be a gerund/present participle (VBG) or a past participle (VBN). The action after the connecting word is the starting point of the callback relationship. An example of such relationship is in sentence "*when image is downloaded, the busy image is removed."*

The arrowed lines in Figure 5 show three relationships between actions (two being sequencing and one being callback).

3.2.4 Post-processing

The step of post-processing further processes the identified actions, and forms an AR graph for the given NL query. RACS adds an inferred notional verb "get" to a description containing only the link verb "be". Consider an example description "screen width is less than 960px". This description does not contain a notional verb. Adding "get" as "get the screen width" makes the description more precise. This process is similar to Hill et al.'s technique [21] for inferring an action for a method name that does not begin with a verb.

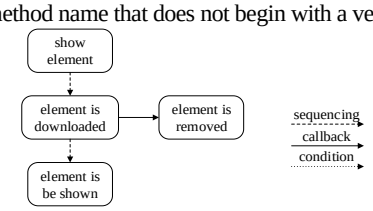

Figure 6. AR graph for NL query in Figure 1

Elements selected by the jQuery selector are usually defined in HTML code, and too little detailed information about the elements could be found only in the JavaScript code. For example, consider three code snippets where *$(".child").hide()* and *$("#option").hide()*[2] are used to hide a selected "div" element[3], respectively, in two JavaScript code snippets; *$("div").show()* is used to show the "*div*" element in the third code snippet. If the given user query is "hide div", although the three code snippets would have the equal number of terms matched by the query, the first two snippets should be returned but not the last one. The reason is that in NL queries, terms for describing the detailed types (e.g., "*div"*) of DOM elements are not as important as other terms (e.g., "*hide", "show"*) for jQuery-API identification. In addition, code snippets with the same usage pattern but with different element types as the target code snippet are still often useful to the developers. For example, in Figure 1, if we change all the "*img*" element type in the accepted answer to the "*div*" element type, the code snippet still gives valuable hints to complete the target task.

Based on this insight, our step of post-processing replaces some specific nouns (e.g., "*image*", "*div*" indicating element types) with more general ones (e.g., "element") based on the word list for DOM elements in the domain-specific dictionary (described in Section 3.2.1), and removes determiners and adjectives. For example, RACS transforms "the busy image" to "element". At the same time, RACS stores these changed words as *keywords*, which are used for later ranking code snippets (as described in Section 3.3.3). The AR graph for the NL query in Figure 5 is shown in Figure 6.

3.3 Searching Snippets

Given an AR graph generated from the NL query, this step first derives multiple AR-derived MCR (A-MCR) graphs for the AR graph based on text semantic similarity. Then it uses the A-MCR graphs as the graph query to search the set of MCR graphs (produced by the technique described in Section 3.1) for the most relevant MCR graphs. Finally, it selects top code snippets associated with the most relevant MCR graphs as the snippets returned to the user.

3.3.1 Deriving A-MCR graphs from the AR graph

The names of actions (NL by nature) in an AR graph are often quite different from the names of framework API methods (programming languages by nature) in an MCR graph. Such differences pose barriers for searching MCR graphs with an AR graph as the graph query. To address such challenge, we refine the AR graph to

[2] In CSS, *".child"* selects all elements with *class="child"*, and *"#option"* selects all elements with *id="option"*. An element's *class* and *id* attributes and their values are described in HTML code, e.g., * *.

[3] The "div" element indicated by a tag defines a division or a section in an HTML document.

multiple AR-derived MCR (A-MCR) graphs by replacing the actions in the AR graph with the actions' corresponding framework API methods, determined by our technique of text semantic similarity. Given an action, RACS searches for API methods whose descriptions in the API documentation share high text semantic similarity with the description of the action (i.e., the text description of the action in the NL query).

To match actions and API methods, matching based on text semantic similarity [13] in RACS has advantages over keyword (lexical) matching, which has been widely used in existing approaches such as SNIFF [7]. Keyword matching is fast and strict, while ignoring semantic similarity, but may miss to match many cases, e.g., "I own a dog" and "I have an animal". On the other hand, matching based on text semantic similarity is based on word-to-word similarity metrics, addressing such issue and achieving a higher recall. In particular, matching based on text semantic similarity computes the similarity between two texts (T_1 and T_2) using the following equation:

$$sim(T_1, T_2) = \frac{1}{2}\left(\frac{\sum_{w\in\{T_1\}}(maxSim(w,T_2)*idf(w))}{\sum_{w\in\{T_1\}} idf(w)} + \frac{\sum_{w\in\{T_2\}}(maxSim(w,T_1)*idf(w))}{\sum_{w\in\{T_2\}} idf(w)}\right)$$

where $maxSim(w,T)$ computes the maximum word-to-word similarity between word w and any word with the same POS in text T; $idf(w)$ denotes the inverse document frequency of word w. The word-to-word similarity metrics can be either knowledge-based (e.g., WordNet similarity: https://wordnet.princeton.edu/) or corpus-based (e.g., latent semantic analysis: http://lsa.colorado.edu/). In our implementation, we use the knowledge-based MCS technique implemented by the SEMILAR toolkit [10].

For each action, RACS chooses the top *K* framework API methods as the candidate methods corresponding to the action. Thus, for a given AR graph with *n* actions, RACS replaces each action of AR graph with multiple corresponding candidate methods, and attain K^n A-MCR graphs.

3.3.2 Searching MCR graphs with A-MCR graphs

This step uses the A-MCR graphs as the graph query to search for the most relevant MCR graphs, based on graph similarity.

We first define "matched relationships" (i.e., edges) between two graphs. Given two graphs G_1(an A-MCR graph) and G_2 (an MCR graph), relationship $< m_i, m_j, t >$ from G_1 and relationship $< m'_k, m'_l, t' >$ from G_2 are matched when $(m_i == m'_k)$ && $(m_j == m'_l)$ && $(t == t')$. Two method signatures are equal when they have the same method name and parameter type list.

Then we define the "graph similarity score" between A-MCR graph G_1 and MCR graph G_2 as

$$sim(G_1, G_2) = \frac{mrn(G_1,G_2)}{rn(G_1)+rn(G_2)-mrn(G_1,G_2)}$$

where $mrn(G_1, G_2)$ denotes the number of matched relationships between G_1 and G_2 ; $rn(G_i)$ denotes the total number of relationships (i.e., edges) in G_i. This graph similarity score has a value between 0 and 1, with the score of 1 indicating that two MCR graphs are identical, and the score of 0 indicating no matched relationship between the graphs. RACS selects the top *K'* MCR graphs ranked based on the graph similarity scores for further selection of snippets (as described in the subsequent subsection).

3.3.3 Selecting Code Snippets

Each MCR graph (indicating a usage pattern) may be associated with more than one code snippet, and not all the associated code snippets are of the same importance to the user. To help the user locate the desired code snippets more efficiently, this step selects one code snippet for each of top *K'* ranked MCR graphs based on two metrics: the number of the keywords in the NL query matched by a code snippet, and the length of the code snippet.

Recall that the *keywords* in the NL query are those words changed by the technique described in Section 3.2.4. RACS gives higher ranks to code snippets matching more keywords in the NL query. This ranking heuristic gives preference to those code snippets more similar to the NL query. If two code snippets have the same number of matched keywords, RACS ranks the shorter code snippet higher. This ranking heuristic has been widely used in previous approaches, such as Prospector [3] and PARSEWeb [5], making the returned code snippets concise. After applying the two ranking heuristics, RACS finally returns the ranked code snippets as the final search results for the given NL query.

4. EVALUATIONS

We conducted evaluations to assess the effectiveness of RACS. In our evaluations, we implemented RACS and addressed two main research questions:

RQ1: How effectively can RACS search JavaScript code snippets for a given NL search query?

RQ2: How much can RACS outperform a relationship-oblivious code search approach?

4.1 Evaluation Setup

We constructed a JavaScript snippet base consisting of snippets using the popular jQuery framework. The snippet base was constructed from Ohloh (currently OpenHub https://www.openhub.net/) and Amazon.com. Ohloh is a popular public directory of open source software projects. We attained the code locations of 620 jQuery-tagged projects from Ohloh and downloaded the source code of these open source projects. Amazon.com is one of the world's top ten web sites [28], which uses jQuery heavily. We ran a web crawler on Amazon to download jQuery-related files. We constructed the code base in this way to ensure the diversity and quality of our snippet base. The snippet base consists of 81,540 JavaScript files, from which we obtained 308,294 code snippets. Table 1 shows the details of the snippet base. We mined 9,905 API usage patterns from the snippet base, according to jQuery core API documentation (which contains over 700 method signatures) within 115s.

Table 1. JavaScript snippet base

Source	#Project	Date of	#files	#snippets
Ohloh	620	06/28/2014	51,949	226,099
Amazon.com	1	08/16/2014	29,591	82,195
Total	621	--	81,540	308,294

We also constructed a benchmark, which contains 50 real-world representative jQuery related queries from Stack Overflow. We manually checked the list of highest voted jQuery-tagged questions sequentially, and unless a question satisfies any of the following exclusion conditions, it was added to the benchmark.

(1) The accepted answer of the question does not contain a JavaScript code snippet. (2) The code snippet in the accepted answer contains only JavaScript built-in method(s), without using jQuery or other JavaScript frameworks. (3) The code snippet in the accepted answer is implemented by using only a jQuery plugin or a non-jQuery JavaScript library. (4) The question is only about the setting of parameters, such as the writing of selectors. (5) The description of the question is vague. The questioner and the respondent discussed the details later in the comments. (6) The description of the question is at too high abstraction level. For example, for the question "*get all descendant text nodes of an element*", the accepted answer mentions *"jQuery does not have a convenient function for this. You need to combine* `contents()`*, which will give just child nodes but includes text nodes, with* `find()`*, which gives all descendant elements but no text nodes*". (7) The question is not related to code search, such as discussing programming experiences. For example, a search query cannot be extracted from a question with title "`.prop()` vs. `attr()`". (8) The code snippet in the accepted answer

contains only one jQuery method (instead of two or more jQuery methods as focused by the work in this paper).

We manually checked the selected questions and further filtered out redundant questions: some questions have similar descriptions and thus have the same or very similar code snippets in their accepted answers. Table 2 shows these 50 search queries. We acquired a "NL search query" directly from the corresponding question's title and description. All the questions corresponding to these search queries have accepted answers with code snippets. For each question, we used the code snippet in the accepted answer as the target snippet, and checked whether a snippet returned by RACS hits the target snippet (i.e., both snippets have the same jQuery API method calls with the same relationships); if so, we consider such snippet as a hit snippet. Instead of user studies, our evaluations use real-world snippets from answers accepted by developers to validate our results. To good extent, the "accept" reflects the real-world developers' feedback of usefulness. Our validation is conservative: some "failed" cases could actually be reasonable snippets that help users for their tasks at hand.

In our evaluations, parameters K and K' introduced in Section 3.3 are set to 5 and 50, respectively, by default. In initial investigation of sample cases (outside of the benchmark), we evaluated different settings of K, and found that when K is set as 5, we could achieve a good balance between acceptable query time and satisfactory recall. Note that when K is set higher, the query time is longer and the recall is higher. Setting K' to 50 is to limit the total number of results needed to be checked. All of our evaluations were conducted on a Linux machine with an Intel i7 3.0 GHz CPU and 4 GB of RAM. The details of our evaluation subjects and results can be found on our project website: https://sites.google.com/site/racsproject/.

Table 2. Selected Stack Overflow queries, search results of RACS, and characteristic of accepted answers

No.	Question ID	NL Search Query	Query AR Graph		Target Snippet MCR Graph		T1 (ms)	T2 (ms)	Top Rank
			#Meth	#Rela	#Meth	#Rela			
1	1854556	If a field is click into, check if input is empty, display a red background.	3	2	4	3	209	463	1
2	6677035	When the user clicks on that input subject, the page should scroll to the last element of the page with a nice animation to scroll to bottom and not to top.	3	1	3	2	247	471	44
3	554273	When someone clicks on an image, change the image source.	2	1	2	1	188	78	3
4	986120	Get the value of the selected radio button when any of these three are clicked.	2	1	2	1	195	100	1
5	1423561	Hide the container if focus is lost.	2	1	2	1	181	80	1
6	699065	When I press Enter on the form, the form is submitted.	2	1	2	1	194	79	1
7	901712	If the age checkbox is checked, then I need to show a textbox to enter age, else hide the textbox.	3	2	2	1	214	356	1
8	152975	Show HTML menus completely when a user clicks on the head of these menus. Hide these elements when the user clicks outside the menus' area.	4	3	3	2	287	2633	NF
9	169506	When I catch the submit form event with jQuery, get all the input fields of that form in an associative array.	2	1	2	1	211	101	1
10	1594952	When the text field is empty the submit button should be "disabled". When something is typed in the text field to remove the "disabled" attribute. If the text field becomes empty again the submit button should be "disabled" again.	6	5	4	3	302	70155	NF
11	24816	Escaped an arbitrary string and display in an HTML page.	2	1	2	1	191	90	NF
12	303767	Grab the height of the window and the scrolling offset in jQuery.	2	1	2	1	200	88	NF
13	1216114	Make a div stick to the top of the screen once it's been scrolled enough to contact its top boundary	2	0	2	1	236	90	NF
14	253689	Change the background image of a div when it is clicked.	2	1	2	1	193	88	1
15	480735	Select all contents of textbox when it receives focus.	2	1	2	1	189	89	3
16	164085	Execute a callback when an IFRAME has finished loading.	2	1	2	1	196	91	NF
17	376081	Loop though the table, and get the value of the "Customer Id" column for each row.	2	1	2	1	192	83	NF
18	4551175	Before the AJAX request if the previous request is not completed I've to abort that request and make a new request.	2	1	2	1	224	85	NF
19	912711	Load javascript file only if the user clicks on a certain button.	2	1	2	1	202	99	1
20	47824	Remove all the options of a select box, then add one option.	3	2	3	2	288	498	NF
21	540349	Hide the rollover image when the onmouseout event happen	2	1	2	1	194	90	1
22	3709597	Wait for all Ajax requests to be done before I execute the next	2	1	2	1	167	85	NF
23	34830973	If a field is clicked, display a background image	2	1	2	1	209	463	1
24	3044573	Determine the size of the browser viewport, and to redetect this if the page is resized?	3	2	3	2	245	590	NF
25	8423217	An event to fire client side when a checkbox is checked	2	1	2	1	202	86	NF
26	5797539	When you click inside a textarea, its entire content gets selected	2	1	2	1	231	417	4
27	871063	Check radio option whether no default is set and then set a default.	2	1	2	1	211	376	1
28	4177159	When element clicked, toggle between checked and unchecked.	2	1	3	2	223	98	NF
29	1064089	When someone clicks a link, a word or two to be inserted where the cursor is.	2	1	2	1	320	1687	1
30	437958	When one of these links is clicked, hide the links that are not clicked.	2	1	2	1	230	79	1
31	1212500	Create a CSS class and add it to DOM at runtime with jQuery.	2	1	2	1	184	83	1
32	7717527	JQuery smooth scrolling when clicking an anchor link.	2	1	2	1	179	80	NF
33	9398870	Remove the top and left attribute from the inline style on the div when clicked.	2	1	2	1	199	345	3
34	946534	Insert text into a text area using jquery, upon the click of an anchor tag.	2	1	2	1	189	78	1
35	1925614	Get the value selected from a dropdown menu and change the form action	1	0	2	1	201	79	NF
36	360491	Strip white space when grabbing text with jQuery?	2	1	2	1	178	83	NF
37	2358205	Trigger an event after any other type of iterative callback has completed.	2	1	2	1	234	345	NF
38	4687579	I want just the new "blah" div to fade in after the content gets appended.	2	1	2	1	169	76	1
39	3024391	Get child elements and iterate through each of those elements.	2	1	2	1	197	85	NF
40	2380230	Get the selected option from a dropdown and populate another item with that text.	2	1	2	1	203	98	1
41	316278	Have an element fade in, then in 5000 ms fade back out again	2	1	2	1	187	80	NF
42	2330209	If the "Check Me" checkbox is checked, all the other 3 checkboxes should be enabled.	2	1	2	1	223	485	1
43	4613261	Get the position of layer1 and set the same position to layer2.	2	1	2	1	202	87	1
44	5176803	When the radio button is selected I enable an edit box.	2	1	2	1	198	89	NF
45	4996002	Get the index of the child li relative to it's parent, when clicking that li	2	1	2	1	188	104	1
46	13626517	Disable inputs at first and then enable them when click a link	3	2	3	2	174	84	NF
47	2230704	Get the value of the hidden field when the select is changed.	2	1	2	1	210	93	1
48	6658752	Generate a new tag with class name "test" in h2 by clicking the button	2	1	2	1	186	88	8
49	4076770	When the <select> dropdown is changed, get the value before change.	2	1	2	1	213	80	1
50	1314450	Capture the TAB keypress, cancel the default action.	2	1	2	1	204	479	1

4.2 Metrics

To assess the effectiveness of a code search approach with respect to *a single query*, our evaluations used the metric of the *best hit rank*, i.e., the highest rank of the hit snippets for the query. A higher best hit rank implies lower user effort for inspection to find the hit snippet.

To assess the effectiveness of a code search approach with respect to a set of queries, our evaluations used the metric of *success percentage at k*, i.e., the success percentage among the set of queries considering only the top *k* results returned by a search approach. In particular, the *success percentage at k* (P_k) in our evaluations is calculated using the following formula:

$$P_k = \frac{\#\ best\ hit\ ranks\ that\ are\ less\ than\ k}{total\ \#\ queries}$$

We investigated P_k with k's respective values as 1, 5, and 10 returned snippets, reflecting the typical sizes of snippets that various users would invest to inspect. Such metric has been popularly used to assess the effectiveness of a code search approach [8][27]. Note that we do not use MRR (Mean Reciprocal Rank), which is popular for assessing navigational search and question answering but is not appropriate for assessing code search.

4.3 Effectiveness of RACS (RQ1)

We first evaluated the effectiveness of RACS. Table 2 also shows the results of RACS and some characteristics of a question's accepted answer (including its sample code snippet). Columns "#Meth" and "#Rela" under "Target Snippet MCR Graph" show the number of the jQuery API method calls and relationships in the accepted answer, respectively. Columns "#Act" and "#Rela" under "Query AR Graph" show, for each NL search query, the total number of actions and relationships that are identified, respectively. Columns "T1" and "T2" represent the time (in millisecond) for deriving A-MCR graphs (Section 3.3.1) and searching MCR graphs (Section 3.3.2), respectively. The last column shows the best hit rank, i.e., the highest rank of hit snippets that answered the question. "NF" denotes "Not Found".

For 23 of the queries (46% of 50), top 1 of the snippet list returned by RACS is a hit snippet, i.e., one that matches the target code snippet. For 28 queries (56% of 50), top 10 of the snippet list returned by RACS include at least one hit snippet. Once RACS constructed a very precise MCR graph, which is the same as the MCR graph of the accepted answer's code, RACS returns the right snippet in the top 1 rank. As shown in Figure 6, RACS accurately returned code snippets for queries 6 and 7 in Table 2. The jQuery API method calls (marked with a rectangle box) meet the semantics of each action, and the code structures meet the relationships implied in the NL query.

RACS did not return good results for some queries as shown in Table 2. There are three main reasons. First, our current snippet base is not sufficiently large to contain their required sample snippets (queries 8, 10, 16, 18, 20, 24, 39, 41, and 46). When we added (to our current snippet base) the code snippet from the accepted answer for each query, all of these queries got the target snippet in top 10 results. Second, the AR graph generated from a query may not exactly reflect the semantics (queries 2, 13, and 35). Queries 2 and 35 miss one relationship, and query 13 includes an incorrectly identified callback relationship. Third, an NL search query is not similar to the required method's API documentation description (queries 11, 12, 17, 22, 25, 28, 36, 37, and 44). Thus, given limited API documentation description, relying on semantic-similarity-based method searching, RACS cannot identify the candidate methods for these nine queries unless these queries are rewritten.

We also investigated the influence of replacing some specific nouns with more general ones as done in post-processing (in Section 3.2.4). The detailed results can be found on our project website: https://sites.google.com/site/racsproject/.

Query 6: "*When I press Enter on the form, the form is submitted*"

```
$("[id^='relExInput']").keypress(function(A) {
  if (A.which == 13)
  {
    $("#i").val(this.value);
    $("#calculate").submit();
  }
});
```

Query 7: "*If the age checkbox is checked, then I need to show a textbox to enter age, else hide the textbox.*"

```
$('.hide-postbox-tog').bind('click.postboxes', function() {
  var box = $(this).val();
  if ($(this).prop('checked'))
  {
    $('#' + box).show();
    if ($.isFunction(postboxes.pbshow))
    self.pbshow(box);
  } else {
    $('#' + box).hide();
    if ($.isFunction(postboxes.pbhide))
    self.pbhide(box);
  }
  self.save_state(page);
  self._mark_area();
});
```

Figure 6. Top 1 ranked code snippet for queries 6 and 7.

4.4 Comparison with Relationship-oblivious Approach (RQ2)

We next present the evaluation results of comparing RACS with a relationship-oblivious approach. We implemented a relationship-oblivious code search approach (ROCS) that uses keyword matching between the given query text and the API documentation text related with a particular code snippet. This implemented relationship-oblivious approach shares key ideas with two existing state-of-the-art code search approaches (SNIFF [7] and Exemplar [31]). Similar to SNIFF, this relationship-oblivious approach applies stop-word removal and stemming to a user query, and retrieves usage patterns (in the form of MCR graphs for direct comparison with RACS) from the snippet base based on *keyword matching* (while taking no account of the order of keywords). The existing relationship-oblivious approaches typically use *support* for ranking each usage pattern (along with a sample code snippet that matches the pattern) based on the number of code snippets that match the pattern. Such ranking is based on the premise that more-popularly implemented patterns tend to be more relevant for a query among all the patterns that match the query (i.e., matching the keywords in the query).

By contrast, RACS uses two different search techniques as presented in Section 3.3. In particular, RACS uses semantic similarity (instead of keyword matching) for matching a query text against text in API documentation (see Section 3.3.1 for detail). RACS uses relationship-aware ranking (instead of pattern support), which ranks the MCR graphs by the graph similarity of A-MCR graphs (derived from the query) and MCR graphs (see Section 3.3.2 for detail). To evaluate how these two key techniques in RACS contribute to the overall effectiveness of RACS, we also implemented two variant approaches that each replace one technique in RACS with the corresponding technique in the baseline approach ROCS. Then we compared the effectiveness of the four approaches:

RACS: semantic similarity + relationship-aware ranking
ROCS: keyword matching + relationship-oblivious (i.e., support-based) ranking
ROCS^{+}: semantic similarity + relationship-oblivious ranking
RACS^{-}: keyword matching + relationship-aware ranking

We investigated the *success percentage at k* with k's respective values as 1, 5, and 10 snippets. The results are shown in Table 3. The table shows that RACS could answer more questions with higher rank than ROCS. RACS could hit the target code snippet with the top 1 snippet for 46% queries, compared to only 4% achieved by ROCS. ROCS found the desired code snippet for only query 30 and query 38 in top 1

snippet. In top 5 snippets, the success percentage of RACS is 54%, while the success percentage of ROCS is only 10%. With top 10 snippets, RACS could answer 28 questions for the entire 50 questions, while ROCS could answer only 8 questions. We did t-test on the value of the best hit rank. RACS significantly outperformed the other three approaches. The detailed results can be found on our project website: https://sites.google.com/site/racsproject/.

Table 3. Comparison results

Metrics	Pattern Searching / Method Searching	Relationship-oblivious	Relationship-Aware	Typeless Relationship-Aware
P_1	Keyword Matching	4%	16%	10%
P_5		10%	22%	20%
P_{10}		16%	26%	26%
P_1	Semantic Similarity	14%	46%	36%
P_5		34%	54%	52%
P_{10}		48%	56%	56%

RACS: ROCS: $ROCS^+$: $RACS^-$:

In Table 3, the success percentage results in column "Relationship-aware" are always higher than the results in column "Relationship-oblivious", indicating that relationship-aware ranking performs better than relationship-oblivious (support-based) ranking. The results show that relationship among API method calls is very valuable when conducting code search for JavaScript frameworks. Sometimes, code snippets with the highest support may not be the target snippets. For example, for query 4 *"Get the value of the selected radio button when any of these three are clicked"*, the best hit rank of RACS is 1. The top 1 code snippet contains a callback relationship of `.click(FUNCTION)` and `.val()`. In contrast, the best hit rank of $ROCS^+$ for query 4 is 10. The $ROCS^+$ approach ranks the sequencing of `.children(STRING)` and `.find(STRING)` first, with the highest support. RACS's awareness of the method call relationship improves the effectiveness of searching.

Table 3 also shows that the approaches based on semantic similarity achieve higher success percentage than the approaches based on keyword matching. The approaches based on keyword matching are effective only if the words in an NL search query exactly match the words in API documentation. RACS uses text semantic similarity, which can overcome such shortcomings. For example, for query 3, "*When someone clicks on an image, change the image source*", RACS found a code snippet in top 1 similar to the accepted answer's code snippet, while $RACS^-$ failed to answer this query. RACS analyzed the sentence in the NL search query and generated the MCR graph with method signature set `{.click(FUNCTION),.attr(STRING,STRING)}` and *callback* relationship between them. $RACS^-$ failed in searching for a relevant method using keyword matching, because the query and API documentation description use semantic similar words ("change" and "set"), rather than exactly the same word.

We also investigated the significance of identifying different types of relationships. In the processes of mining API usage patterns and abstracting an NL query, we treated all the three kinds of relationships as one type – sequencing relationship, leading to more AR graphs that have the same similarity with the A-MCR graph. We used support to re-rank patterns with the same similarity. As shown in the last column of Table 3, not differentiating relationship types leads to reducing the effectiveness, especially for P_1. In addition, we found that the number of the relationships does not affect the effectiveness of RACS when the code corpus includes the target code snippet. For queries with 1 or 2 relationships, RACS gets better results than being relationship-obvious. Actually, >2-relationship queries are rare in Stack Overflow, and their target code snippets are also rare in the snippet base. After we added in the snippet base the target code snippets from the accepted answers for each query, all of these >2-relationship queries got their target snippets in top 10 results.

We compared RACS with Ohloh Code (https://code.openhub.net/), which is a publicly available industrial Internet-scale code search engine. All our projects for building the snippet base except Amazon are included in the underlying repositories used by Ohloh Code. We removed the Amazon snippets from the snippet base of RACS, mined 6,778 usage patterns, and searched on the smaller snippet base. For Ohloh Code, we added "jquery" to each benchmark query and filtered out non-JavaScript code snippets. If there was no hit in top 10 search results, we directly used the API names in the accepted answer as query keywords in place of the NL query. For the top 10 search results, RACS could hit the target code snippet for 48% queries, while Oholh Code could hit for 16%: RACS substantially outperformed Ohloh Code.

4.5 Threats to Validity

The threats to external validity primarily include the degree to which selected JavaScript frameworks and search queries are representative of true practice. There are many kinds of JavaScript frameworks for different purposes. In our evaluations, we selected only the most commonly used web-application related framework – jQuery. There are other frameworks with different qualities of documentation, which may influence the results. The qualities of search queries also affect the query results. To make queries used in our evaluations to reflect real-world queries, we selected representative questions from Stack Overflow based on the vote number, and directly used the question title and description as search queries. Queries written by different users have different qualities. These threats could be reduced by more experiments on more frameworks and more search queries in the future. In addition, the relationship-oblivious approach was implemented by us. To alleviate this threat, we already took great care to accomplish fair comparison and evaluation. For example, the only two modifications from RACS to produce ROCS are (1) from semantic similarity to keyword matching and (2) from relationship-aware ranking to support ranking, where the keyword matching and support ranking are common/typical techniques adopted by existing approaches. Moreover, we implemented two variant approaches ROCS+ and RACS- to represent broad comparison bases.

5. Discussion

In this section, we discuss the applicability and limitations of our current implementation of the RACS approach.

Given free-form NL descriptions, RACS can effectively search snippets (JavaScript framework client code) for relevant code snippets. RACS is very useful for beginner programmers of using a framework. The programmers do not need to know details about the framework, such as the method name and type information in the target framework API method. Our implemented tool can be integrated in programming Q&A sites and development environments for the jQuery framework.

With some modifications, our RACS approach can be applied to a wider scope. For example, when used for another JavaScript framework, RACS needs to use only the framework's corresponding API documentation. RACS focuses on a JavaScript framework, and introduces three common relationships in JavaScript code. Considering only sequencing and condition relationships, RACS could be applied to other languages. We can also define more relationships that best show these languages' features.

Our RACS approach attains the NL description for an API method directly from the API documentation's short description, which may not comprehensively capture the API method's semantics. The user may use a high-level description where one action maps to multiple API methods. Automatic techniques of comment generation [32] and NL relation classification techniques based on model neural networks [4] may alleviate this problem. We can also attain more knowledge by crowdsourcing [33] beyond API documentation.

Automatically identifying actions and relationships from an NL search query may not work well for some search queries due to the arbitrariness of NL, especially for sentences with ambiguous meanings or grammatical

mistakes. Cooperation between the user and the tool [18] can be used to address such issues. Another extension is to incorporate deep learning-based approaches to automatically characterize code features [14][36].

6. Related Work

In this section, we discuss related work to our code search approach, along with our approach's technique of mining framework API usage patterns and technique of abstracting the AR graph from an NL query.

6.1 Source Code Search

There have been various code search approaches for different forms of queries. The most common form is an NL query, which is the same form as the one in general search engines. Mica [29] augments Google Web API's search results to help programmers find the target API classes and methods given a description of desired functionality. Mica can return some web pages containing code snippets that show basic usage of API methods. RACS directly searches code snippets in a large-scale code base and can find complex usage of API methods. Keivanloo et al. [8] use code-clone detection to spot out working code snippets, with a time complexity as low as the complexity of existing code search engines. Portfolio [27] uses the PageRank and spreading activation networks to help programmers navigate and understand usages of the given methods. These approaches require users to provide good query terms and require that keywords extracted from the query terms appear in the code base. SNIFF [7] searches API document description of API methods invoked in the code base to support a query in plain English. CodeHow [35] recognizes potential APIs with the help of API documentation and applies the Extend Boolean model instead of a SVM model to retrieve code snippets that match queries. RACS supports a free-form NL query, and uses a metric to reflect semantic text similarity instead of keyword matching as used by previous related approaches.

Prospector [3] accepts a query in the form of source and target objects types. It synthesizes code fragments using both API method signatures and type cast information mined from a code base. PARSEWeb [5] interacts with the Google code search engine and suggests relevant method-invocation sequences. Semantics-based code search [26] lets users specify what they are looking for as precisely as possible using keywords, method signatures, test cases, etc. The query forms required by these preceding approaches may not be easy to formulate if the programmers are unfamiliar with the framework to be reused. RACS accepts a plain NL query, and extracts specifications from the NL query instead of requiring users to formulate a detailed query using programming keywords.

There are other code search approaches whose input query form is close to actual code. Strathcona [23] locates relevant code in a code base based on heuristically matching the structure of the code under development. XSnippet [25] makes use of the context information similar to Strathcona, but it offers improvements on reducing irrelevant code examples being matched along with using only relevant contexts. MAPO [6] mines patterns that describe a certain usage scenario and further recommends mined API usage patterns and their associated code snippets upon users' requests. PRIME [24] can answer queries focused on API usage with code showing how an API method should be used. PRIME searches code over partial programs using a relaxed inclusion matching technique. RACS can answer similar questions without requiring users to write a detailed query such as source code. Chan et al.'s approach [34] constructs an API graph from an API library's implementation code. Such API graph connects classes and methods with relationships (i.e., inheritance class, member methods, input parameter, and output parameter), and then their approach selects nodes (i.e., classes and methods) with high textual similarity on node names only. Subgraphs with higher accumulated node textual similarity are ranked higher. RACS constructs an MCR graph from an API library's client code and uses graph similarity to search MCR graphs with A-MCR graphs by considering both node types and relationship types (i.e., sequencing, callback, and condition).

6.2 JavaScript Code Analysis and Usage Pattern Mining

Code analysis and code mining are basic components of various software engineering tasks. Dealing with JavaScript code needs more specific techniques due to its language features. Our technique of mining API usage patterns for JavaScript frameworks handles JavaScript language features similar to TAJS [15] and JSMiner [9]. TAJS [15] is a whole-program dataflow analyzer for JavaScript, including the ECMAScript standard library and large parts of the W3C browser API and HTML DOM functionality. JSMiner [9] uses a graph-based representation, JSModel, for JavaScript usage and mines inter-procedural, data-oriented JavaScript usage patterns. JSModel contains non-essential information (such as data flow dependencies) that contributes little to producing query results, but reduces the search efficiency. RACS uses MCR graphs to abstractly represent JavaScript code using a certain framework. An MCR graph is more concise and contains essential information that may be reflected in a user query.

6.3 NLP-based Specification Extraction and Program Synthesis

There exist various approaches that extract specifications automatically from NL. Zhong et al. [20] infer resource specifications from API documentation and detect code bugs. Xiao et al. [18] develop a template-based approach to extract security policies from NL software documentation and resource-access information from NL scenario-based functional requirements. Pandita et al. [17] infer formal method specifications from NL text of API documentation. These approaches apply NLP techniques to analyze software documents. Other approaches analyze an NL search query from the users and synthesize programs meeting the requirements of the users. SmartSynth [12] is an end-to-end programming system for synthesizing smartphone automation scripts from NL descriptions. NLyze [11] is an Excel add-in that supports a rich user interaction model including annotating the users' NL specification and explaining the synthesized programs (written in a domain-specific language) by paraphrasing them into structured English. RACS searches for relevant code snippets from a code base based on analyzing an NL search query.

7. CONCLUSION

Existing code search approaches are not effective in finding code snippets that use JavaScript frameworks to implement a specific feature reflected by the given NL search query. In this paper, we have presented a novel Relationship-Aware Code Search (RACS) approach. RACS first collects a large number of code snippets that use some JavaScript frameworks, mines API usage patterns from the collected code snippets, and represents the mined patterns with MCR graphs. Given an NL search query, RACS conducts NL processing to automatically transform the query to an AR graph. In this way, RACS reduces code search to the problem of graph search: searching the MCR graphs for a graph similar to the given AR graph. During the graph search, RACS includes a technique based on text semantic similarity to bridge the gap between NL actions in an AR graph and framework API methods in an MCR graph. We have conducted evaluations against popular real-world jQuery questions (posted on Stack Overflow), based on 308,294 code snippets collected from over 81,540 files on the Internet. The evaluation results show the effectiveness of RACS: the top 1 snippet produced by RACS matches the target code snippet for 46% questions, compared to only 4% achieved by a relationship-oblivious approach.

8. ACKNOWLEDGMENTS

This work is supported by the National Basic Research Program of China (973) under Grant No. 2015CB352201; the National Natural Science Foundation of China under Grant Nos. 91318301, 61421091, 61529201; and US National Science Foundation under grants no. CCF-1409423, CNS-1434582, CNS-1513939, CNS-1564274.

9. REFERENCES

[1] Annie T. T. Ying and Martin P. Robillard. Selection and presentation practices for code example summarization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*, pp. 460-471, 2014.

[2] Siddharth Subramanian and Reid Holmes. Making sense of online code snippets. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pp. 85-88, 2013.

[3] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pp. 48-61, 2005.

[4] Yan Xu, Lili Mou, Ge Li, Yunchuan Chen, Hao Peng, and Zhi Jin. Classifying relations via long short term memory networks along shortest dependency paths. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP '15)*, pp. 1785–1794, 2015.

[5] Suresh Thummalapenta and Tao Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pp. 204-213, 2007.

[6] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP '09)*, pp. 318-343, 2009.

[7] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. SNIFF: A search engine for Java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09)*, pp. 385-400, 2009.

[8] Iman Keivanloo, Juergen Rilling, and Ying Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 664-675, 2014.

[9] Hung Viet Nguyen, Hoan Anh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Mining interprocedural, data-oriented usage patterns in JavaScript web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pp. 791-802, 2014.

[10] Vasile Rus, Mihai Lintean, Rajendra Banjade, Nobal Niraula, and Dan Stefanescu. SEMILAR: The semantic similarity toolkit. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL '13)*, Demo Track, 2013.

[11] Sumit Gulwani and Mark Marron. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*, pp. 803-814, 2014.

[12] Vu Le, Sumit Gulwani, and Zhendong Su. SmartSynth: synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*, pp. 193-206, 2013.

[13] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI '06)*, pp. 775-780, 2006.

[14] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In P*roceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI '16)*, pp. 1287-1293, 2016.

[15] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*, pp. 17-31, 2014.

[16] http://w3techs.com/technologies/overview/JavaScript_library/all

[17] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pp. 815-825, 2012.

[18] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, pp. 1-11, 2012.

[19] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL '03)*, pp. 423-430, 2003.

[20] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*, pp. 307-318, 2009.

[21] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pp. 232-242, 2009.

[22] Jue Wang, Yingnong Dang, Hongyu Zhang, Kai Chen, Tao Xie, and Dongmei Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*, pp. 319-328, 2013.

[23] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '13)*, pp. 237-240, 2005.

[24] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '12)*, pp. 997-1016, 2012.

[25] Naiyana Sahavechaphan and Kajal Claypool. XSnippet: mining For sample code. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pp. 413-430, 2006.

[26] Steven P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pp. 243-253, 2009.

[27] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 111-120, 2011.

[28] http://www.alexa.com/topsites

[29] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding API components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing (VLHCC '06)*, pp. 195-202, 2006.

[30] Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori Pollock, and K. Vijay-Shanker. AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*, pp. 79-88, 2008.

[31] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, pp. 475-484, 2010.

[32] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)* pp. 101-110, 2011.

[33] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. Emergent, crowd-scale programming practice in the IDE. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, pp. 2491-2500, 2014.

[34] Wing-Kwan Chan, Hong Cheng, and David Lo. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, pp.1-11, 2012.

[35] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. CodeHow: Effective code search based on API understanding and extended boolean model. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pp. 260-270, 2015.

[36] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management (KSEM '15)*, pp. 547-553, 2015.