SnippetGen: Enhancing the Code Search via Intent Predicting

Qing Huang[1], Xudong Wang[2], Yangrui Yang[3], Hongyan Wan[1], Rui Wang[1], Guoqing Wu[1*],
[1] State Key Laboratory of Software Engineering, Computer School,Wuhan University, Wuhan, China
[2] International School of Software, Wuhan University, Wuhan, China
[3] College of Information Engineering of North China University of Water Resources and Electric Power, Zhengzhou, China
Email:[1]{qh, why0511, wangrui1989, wgq}@whu.edu.cn, [2]hsu@whu.edu.cn, [3]yangyangrui@ncwu.edu.cn

***Abstract*—To enable the cod sarch results to run immediately without any subsequent modification, an intent-enhanced code search approach (IECS) is proposed. It has the ability of intent predicting to guess what else a user might do after obtaining the search results. Based on the intent-relevant semantic and structural matches, IECS improves the performance of code search by incorporating the intent for expansion. To perform IECS, the code search tool SnippetGen is implemented. Compared with CodeHow and Google Code Search (CS), SnippetGen outperforms them by 28.5% with a precision score of 0.846 (i.e., 84.6% of the first results are relevant).**

Keywords-Code search; Intent predicting; Query expansion

I. Introduction

To reuse the existing method, many code search tools are proposed. Early code search engines, e.g., Google[1], Krugle[2] and Koders[3], offered only the keyword-based search with low precision. The later work did semantic search to enhance the accuracy of the search results, e.g., signature matching, type matching [15], [10]. But these approaches were impractical, because they required too little or too much specification. The current work supports query expansion to promise the better usability, such as CodeHow [2]. It considers the impact of the APIs and expands the query with the APIs. Although these existing code search tools seem to yield correct matches, the search results could not meet the user demands directly and need to be modified [3]. One major reason is that these tools lack the ability of intent predicting to guess what else a user might do after obtaining the search results.

Example: Given a query "access data in excel", "the method ExcelToDataSet" could be returned by the standard Boolean model [1], because the method contains all the query terms with the highest term frequency such as " Excel8.0; " and "Fill (*dataset*) ". Unfortunately this method is always modified, because it only accesses the data from the outdated excel2003 but fails in the most frequently-used excel2007. In this case, a user changes this method from (*"Microsoft.Ace.OLEDB.4.0", "Excel8.0"*) to (*"Microsoft.Ace.OLEDB.12.0", "Excel12.0"*). If search engine could anticipate this intent, it can generate an expansion query "Microsoft.Ace.OLEDB.12.0 Excel12.0 access data excel" and retrieve the modified method. This example shows that through intent predicting , more accurate code search could be achieved.

In this paper, we propose an intent-enhanced code search approach (*IECS*) using the intent to enhance the search. Figure 1 presents the overall structure containing intent-enriched, intent-first and intent-expanded component . In the intent-enriched component (*as shown in Fig.1a*), code modifications are recorded through the code version tracking service . Then an intent extraction algorithm is proposed to exploit the intents from modifications. This algorithm refines the commonly-used intents, extracts the intent-relevant context and enriches methods with the intents and context in turn. In the intent-first component, results are retrieved by computing the semantic and structural similarity scores between the intent and the query , and combining the two similarity scores (*as shown in Fig.1b*). If the results cannot match the query, the intent-expanded component is triggered to expand a query by considering both the intent and the text similarity (*as shown in Fig.1c*). Finally, the Extended Boolean model [17] is adopted to retrieve the more relevant results.

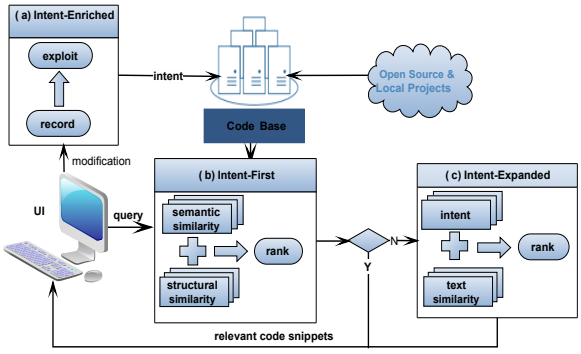

Fig. 1. Similar changes to three methods

An IECS-supported code search tool SnippetGen is implemented. The front-end is a visual studio 2010 extension. The backend is a codebase which is constructed by collecting 2,151

[1]codesearch.google.com
[2]krugle.org
[3]koders.com

*Corresponding author
DOI reference number:10.18293/SEKE2017-037

projects from Github[4] and indexing 1.16 million C# methods using Lucene[5]. SnippetGen is compared with CodeHow [2] and Google Code Search (CS) [16] by performing 70 real-world queries. The results show that SnippetGen achieves a precision score of 0.846 which outperforms them by 28.5% when the top 1 results are inspected. The results con?rm the effectiveness of SnippetGen in programming practices.

The contributions of this paper are as follows:

- An intent-enhanced code search approach (IECS) is proposed. It contains an intent extraction algorithm to exploit the potential intents of the method.
- IECS performs code retrieval preferentially within the intent scope based on semantic and structural matches, which is more effective than within the complete method scope.
- SnippetGen, that performs IECS, is implemented. The experiment results show that SnippetGen outperforms CodeHow and CS by 28.5% with a precision score of 0.846.

II. Intent-enriched component

An intent extraction algorithm is proposed to extract the intents from the past modifications. Algorithm 1 describes this process procedurally in the four steps. Note that each method may have much intent. If SnippetGen incorporates all intent for expansion, it may produce worse results than not expanding the query. Thus the refinement strategy is considered to ensure the intents that benefit the code search.

Step 1: Identifying modifications. For each method m_i, SnippetGen employs ChangeDistiller [11] to compare the AST of the m_i's old and new versions from the past modifications. Then SnippetGen characterizes modifications as a sequence of node operations Δ_i consisting of node insertions, deletions, updates and moves.

Step 2: Refining modifications. SnippetGen uses the modified Longest Common Edit Operation Subsequence (LCEOS) algorithm [18] to identify the common node operations pairs $\Delta_c = \bigcap_{i=1}^{n} \Delta_i$, such that $\forall 1 \ll i \ll n, \Delta_{c_i} \subseteq \Delta_i$, by iteratively comparing the node operations pairwise. In these common node operations pairs, if one or more concrete instances of types, methods, variables and constants have the same edit type or inheriting type, despite of the different name, SnippetGen thinks they are abstract equivalent. Then it generalizes these concrete instances with abstract identifiers \$*t*, \$$m$, \$$v$ and \$$c$, so as to enforce a consistent naming. Meanwhile, it records the mapper between the abstract identifiers and the concrete instances. If some subsequent node operations pairs are inconsistent with the current mapper, they are omitted.

Step 3: Extracting Intents. SnippetGen extracts the intents from the mapper. Meantime, it extracts the intent-relevant context with control, data and containment dependence analysis. This context comprises the unchanged AST nodes that depend on the node operations or on which node operations depend in common node operations Δ_c.

[4]https://github.com/explore

[5]http://lucene.apache.org/

Algorithm 1 IntentExtraction

Input: Original Version O, a set of Modified Version M
Output: Intents
/* step 1: identify modificaion */
foreach m_i in M
  Δ_i=m_i-o //obtain the i-th modifications(Δ_i)
  add Δ_i to Δ_s; // obtain all modifications(Δ_s)
end
/* step 2: refine modification */
// identify the common modifications
Δ_c= $\bigcap_{i=1}^{n} \Delta_i$ such that $\forall 1 \leq$ i $\leq$ n $\Delta_{ci} \subseteq \Delta_i$
foreach Δ_{c_i} in Δ_c
  // obtain the node operations pairs(nop) from Δ_{c_i}
  $\Delta_{c_i} \rightarrow$ nop:
  // obtain the concrete instances(ci) of types, methods, variables and constants from nop
  nop$\rightarrow$ ci
  if abstractMatch(ci) is true
    // ci's edit type or inheriting type is equivalent
    if ci is inconsistent with the mapper
      omit nop;
      continue;
    end
    ci=ai;//substitute the abstract identifiers(ai) for ci
    build the mapper(ai,ci);
  end
end
/* step 3 extract intents */
mapper$\rightarrow$ intents; //extract the intents from the mapper
method = method+intents; //enrich the method with the intents
Return intents

Step 4: Having obtaining intents, SnippetGen enriches each method with the intents and intent-relevant context.

Example: Given the method ExcelToDataSet's original version O, there are two modified versions F, S. SnippetGen employs ChangeDistiller to input (O, F) and output Δ_F= $\{Update\,(O_1\ F_1)\}$, which shows the variable "*Provider*" and the variable "*Extended Properties*" (line1) are changed to "*OleDb.12.0*" and "*Excel12.0; HDR=NO*" respectively (*as shown in Fig.2a*). Similarly, SnippetGen inputs (O, S) and outputs Δ_B=$\{Update\,(O_1\ S_1)\}$ (*as shown in Fig.2b*). Thus SnippetGen identifies the longest common node operations pairs, such that Δ_c=$\Delta_F \cap \Delta_S$=$\{\text{pair}_1\,(\text{Update}\,(O_1\ F_1)\,, \text{Update}\,(O_1\ S_1))\}$. In these common node operations pairs, SnippetGen substitutes the abstract identifiers \$$v_1$ and \$$v_2$ for ("*Microsoft.ACE.OLE DB.12.0*", "*Excel12.0; HDR=NO*") in Δ_A,("*Microsoft.ACE. OLEDB.12.0*", "*Excel12.0;*") in Δ_B (*as shown in Fig.2c*). Meanwhile, it records the mapper ($\$v_1, \$v_2$)=$\{$("*Microsoft. Ace.OleDb.12.0*", "*Excel12.0; HDR=NO*"), ("*Microsoft.Ace. OleDb.12.0*", "*Excel12.0*"). Finally, SnippetGen extracts the intent from this mapper , such that Intent=("*Microsoft.ACE. OLEDB.12.0, Excel12.0; HDR=NO*").

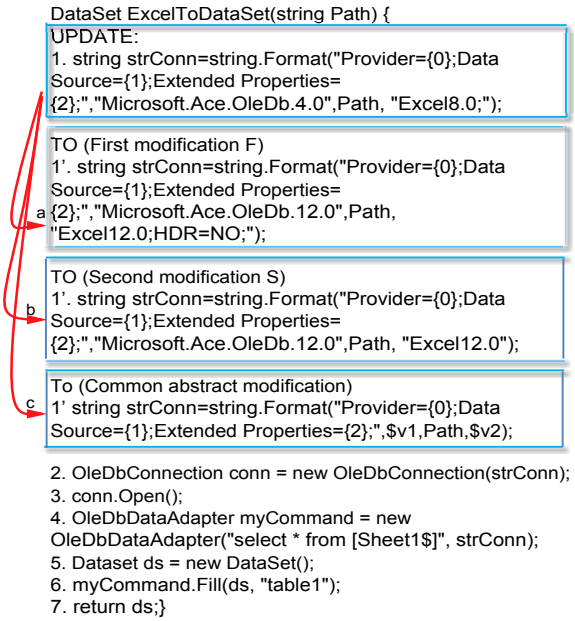

Fig. 2. Similar changes to three methods

III. Intent-first component

To retrieve the relevant methods preferentially within the method's intent scope, SnippetGen computes the semantic similarity between the query and the intents as well as the structural similarity between the query and the intent-relevant contexts, combines the two similarity values, and finally returns the relevant code. In this process, we define two method scores and an operation.

Definition 1 (Semantic Score). For each method m_i, SnippetGen views the query and m_i's intents, as a bag of words, computes their textual similarity score m_i^d. score using VSM[6] [9], and returns the top i semantic method scores denoted as m^d:

$$m^d = \{m_1^d,\ m_2^d,\ \cdots,\ m_i^d\}$$

In VSM, the query and the m_i's intent are represented by a vector. The term frequency (*tf*) and inverse document frequency (*idf*) are calculated based on the frequency of words.

Definition 2 (Structural Score). For each method m_i, SnippetGen views the query and m_i's intent-relevant context as a bag of function calls, computes their structural similarity score m_i^s. score using VSM, and returns the top i structural method scores denoted as m^s:

$$m^s = \{m_1^s,\ m_2^s,\ \cdots, m_i^s\}$$

In VSM, the query and the method are represented by a vector. *Tf* and *idf* are calculated based on the frequency of the function being called.

Definition 3 (Score Combination). Let the methods appearing in both m^d and m^s as $m_{overlap}$, and let the methods appearing only in m^d or m^s as $m_{notoverlap}$. Given m_i^d and m_i^s, the combination as follows:

$$m_i.\text{score} = \begin{cases} m_i^d.\text{score} + m_i^s.\text{score} & (\text{if } m_i \in m_{overlap}) \\ \dfrac{\text{MinOverlapScore} \times m_i^{d/s}.\text{score}}{\text{maxNotOverlapScore}+ \propto} & (\text{if } m_i \notin m_{overlap}) \end{cases} \quad (1)$$

where *MinOverlapScore* is the minimum score of all methods in $m_{overlap}$; *maxNotOverlapScore* is the maximum score of all methods in $m_{notoverlap}$. If m_i only appears in m^d, $m_i^{d/s}$.score equals to m_i^d.score. If m_i only appears in m^s, $m_i^{d/s}$.score equals to m_i^s.score. The parameter $\propto$ is an adjustment factor to make sure that the score of $m_{overlap}$ is larger than that of $m_{notoverlap}$. Empirically, we set $\propto$ to 0.1.

Actually, Equation (1) says that, if m_i appears in both m^d and m^s, its score is the sum of the two scores. Otherwise, its score is calculated based on the similarity score in m^d or m^s. SnippetGen computes the scores in above way and obtains the top i potentially relevant methods:

$$m_{relevant} = \{m_1, m_2, \cdots, m_i\}$$

Example: for the query "access data in excel", the semantic relevant methods m^d with similarity scores are {*Excel2007 ToDateSet=0.5, AccessToDateSet=0.4, ExcelTo DateSet=0.4*}. The structural methods m^s with similarity scores are {*OleDbDataAdapter.Fill=0.9, Excel2007ToDateSet=0.6, ExcelToDateSet=0.5*}.

The overlapping methods $m_{overlap}$ are "*Excel2007To DateSet*" and "*ExcelToDateSet*". We compute their score as $0.5 + 0.6 = 1.1$, and $0.4 + 0.5 = 0.9$, respectively.

The non-overlapping methods $m_{notoverlap}$ are "*OleDbDataAdapter.Fill*" and "*AccessToDateSet*". We get Min OverlapScore value 0.9 and maxNotOverlapScore value 0.9. Thus the scores for "*OleDbDataAdapter.Fill*" and "*AccessToDateSet*" are 0.81 and 0.36, respectively. Finally, the rank of potentially relevant methods $m_{relevant}$ are as follows:

- *"Excel2007ToDateSet" (score=1.1);*
- *"ExcelToDateSet" (score=0.9);*
- *"OleDbDataAdapter.Fill" (score=0.81);*
- *"AccessToDateSet" (score=0.36).*

IV. Intent-expanded component

If the methods retrieved by intent-first component cannot match the query, following the query expansion option [7], SnippetGen expand a query with intents to retrieve the relevant methods.

A query Q_t containing n terms is defined as:

$$Q_t = \ (t_1, \cdots t_n)$$

For a method, three features is defined as:

$$F = (f_1, f_2, f_3)$$

where f_1 stands for the intent ; f_2 stands for the FQN; f_3 stands for the method body .

[6]https://github.com/hcutler/tf-idf/tree/c505c72af7f3eb6e3dd5b10d9e8f54c08e1434d3

A query can be expressed in terms of f_i: t_i where $t_i \in Q_t$ and $f_i \in F_t$. It means to search for methods that contains the term t_i in a field f_i. SnippetGen constructs a Boolean query expression for retrieving methods that match the query in terms of text similarity:

$$q_{text} = (f_2 : t_1 \vee f_3 : t_1) \wedge \cdots \wedge \ \ (f_2 : t_n \vee f_3 : t_n)$$

This query expression searches for methods that contain the terms $t_1, \ldots, t_n$ in fields f_2 (*FQN*) and f_3 (*Method Body*).

After the intent-first component, SnippetGen gets k potentially relevant methods $m_{relevant}$. For each method m_i in $m_{relevant}$, SnippetGen tokenizes the intent to get a keyword list A_i. Then it constructs Boolean query expressions as follows:

$$\begin{aligned} q_{m_i} = f_1 : \ intent_i \wedge \ \ (f_2 : t_1 \vee f_3 : t_1) \\ \wedge \cdots \wedge \ \ (f_2 : t_n \vee f_3 : t_n) \end{aligned}$$

where $m_i \in m_{relevant}$ and $t_k \in (Q_t - A_i)$. Note that we remove the terms that appear in A_i from the query Q_t, since the impact of these terms have been considered in the m_i's intent . This query searches for the methods that contain the intent$_i$ in fields f_1(*intent*) as well as other query terms in fields f_2(*FQN*) and f_3(*Method Body*).

A method may be retrieved by more than one query expressions defined above . SnippetGen combines the query expressions into an expanded query for retrieving methods:

$$q_{expand} = (q_{m_1}, q_{m_2}, \cdots, q_{m_k}, q_{text})$$

Given the query: "access data in excel", the query terms Q_t=(*access, data, excel*). The potentially relevant method's intent is ("*Microsoft,Ace,OleDb,12.0,Excel12.0,HDR=NO*"). The comprehensive query expressions are as follows:

$$\begin{aligned} q_{m_1}\text{=}&(f_1{:}Microsoft, Ace, OleDb, 12.0, Excel12.0 \\ &HDR = NO) \wedge (f_2{:}access \vee f_3 : access) \\ &\wedge (f_2 : data \vee f_3 : data) \\ q_{text}\text{=}&(f_2 : access \vee f_3 : access) \wedge (f_2 : date \vee f_3 : date) \\ &\wedge (f_2 : excel \vee f_3 : excel) \end{aligned}$$

To retrieve relevant methods given the queries, we adopt the Extended Boolean model (EBM) [17], which combines the characteristics of the VSM and Boolean model. Given a query expression $q_{expand} = (q_{m_1}, \cdots, q_{m_k}, q_{text})$, it is easy to implement EMB by using Lucene[7].

V. Experiment

A. Setup

First, a codebase as the backend of SnippetGen is constructed by collecting 2,151 projects downloaded from Github and indexing 1.16 million C# methods by using Lucene. Second, 70 real-world queries are employed by Portfolio's author [8]. All these queries are formulated as set of keywords to address some programming tasks reported in Portfolio's user study. Third, the front-end of SnippetGen as a Microsoft Visual Studio 2010 extension is implemented.

[7]https://lucene.apache.org/core/2_9_4/scoring.html

To investigate the effectiveness of our approach, 20 participants are employed. Six participants are graduate students who have at least of two years of C# programming experience. The others are PhD students who have 3-6 years of C# programming experience. Each participant runs SnippetGen, CodeHow and Google Code Search (CS) to address 3-4 queries and inspect the top 20 results for each query to judge whether they are relevant or not.

CodeHow is the latest query-expanded code search tool [2]. To reprogram it, we index the online MSDN[8] document as expansion library using Lucene. Participants use CodeHow and enter the query directly to retrieve the results. CS represents conventional keywords-based code search web applications. Participants should go to the website, look for implementations and extract them by copying and pasting results into the workspace.

B. Evaluation Metrics

To evaluate the effectiveness of SnippetGen, we make use of the Precision@k[9]:

$$\text{Precision@}k = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{\text{relevant}_{i,k}}{k} \tag{2}$$

where $relevant_{i,k}$ represents the relevant methods for query i in the top k results, Q is a set of queries. Precision@k takes an average on all queries whose relevant answers could be found by inspecting the top $k(k = 1, 5, 10, 20)$ results. A better code search tool allows developers to discover the needed code by examining fewer results. The higher the metric value, the higher the accuracy is.

We also make use of Normalized Discounted Cumulative Gain (NDCG)[10] [12] to measure the ranking capability of the code search based on the graded relevance of the results of a set of queries. It varies from 0.0 to 1.0, with 1.0 representing the ideal ranking of the results. The higher the NDCG value , the better the ranking capability is.

C. Experimental Results

We compare SnippetGen with CodeHow and CS by performing the 70 queries. As Table I shows, when the top 1 results are inspected, SnippetGen achieves a precision score of 0.846, which means that 84.6% of the first results are relevant methods without any subsequent modification. When the top 5 results are inspected, SnippetGen achieves a precision score of 0.861. These results are considered satisfactory . Note that only the results, which both receive relevant feedback and need not to be modified subsequently, are labeled as relevant. Thus the precision of CodeHow and CS is lower than previous papers, such as ref [2], [10].

We pick out CodeHow for comparative analysis, as it is the latest code search tool proposed in 2015. As Table I shows,

[8]https://msdn.microsoft.com/en-us/library

[9]https://github.com/jcnewell/MyMediaLiteJava/blob/master/src/org/mymedialite/eval/measures/PrecisionAndRecall.java

[10]https://github.com/jcnewell/MyMediaLiteJava/blob/master/src/org/mymedialite/eval/measures/NDCG.java

CodeHow achieves a score of 0.658 when the top 1 results are inspected. SnippetGen achieves 28.5%, 66.2%, 70.3%, and 89.5% improvements in terms of Precision@1, Precision@5, Precision@10, and Precision@20 , respectively. In terms of NDCG, SnippetGen obtains a score of 0.873, which also outperforms the CodeHow (0.712) by 22.6%. In the same way, SnippetGen performs better than CS.

TABLE I
THE COMPARISON AMONG SNIPPETGEN, CODEHOW AND CS

	SnippetGen	CodeHow	CS
Precision@1	0.846	0.658	0.421
Precision@5	0.861	0.518	0.368
Precision@10	0.792	0.465	0.346
Precision@20	0.762	0.402	0.283
NDCG	0.873	0.712	0.682

Figure 3 shows the percentage of queries that SnippetGen performs better/worse than CodeHow. When the top 1 results are examined, SnippetGen wins in 36% of the queries and loses in 18% of the queries. In terms of the top 5 results, SnippetGen wins in 61% of the queries and loses in only 7% of the queries. The results confirm that the improvement achieved by SnippetGen is significant.

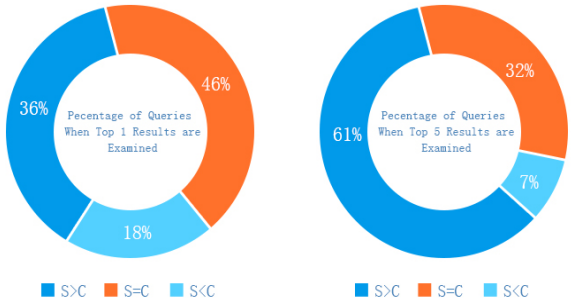

Fig. 3. The mechanism of change pattern

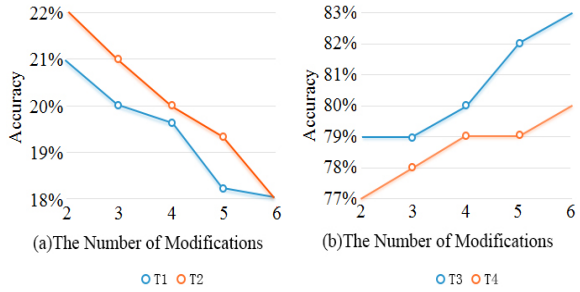

Fig. 4. The mechanism of change pattern

To analyze the reason for the lost cases, we continue to explore what factors are correlated with the accuracy of SnippetGen. We depict the two lost cases (i.e., T1 "Convert utc time to local time" and T2 "How to get Color from Hexadecimal color code") as shown in Fig.4a. They show that the more past modifications provided, the less common subset is likely to be shared among these modifications. This results in that the accuracy goes down. For the draw, the methods are never modified. In this case, SnippetGen should be similar to what other tools could achieve. Different from Fig.4a, we depict the two promising cases (i.e., T3 "access data from excel" and T4 "find regular LINQ expressions") as shown in Fig.4b. They show that the more past modifications provided, the higher accuracy is. It illustrates when methods are similar, adding modifications may not decrease the number of common modifications, but may induce more identifier abstraction and produce more sufficient intents.

These results illustrate that the accuracy of SnippetGen varies with the similarity and number of past modifications. Even the similarity takes precedence over the number. Too many modifications or too few is not good for the accuracy. The more similar modifications are, the higher accuracy is.

VI. DISCUSSIONS

Intent Source: The intent is extracted from each method's past modifications. No modifications, no intent to occu r. It results in SnppetGen not matching semantic similarity between the query and the method's intent, as well as not matching structural similarity between the query and the method's intent-relevant context. According to our user study, the methods without intent make up 25.7% to 38.4% of methods' volume in codebase. If the method carries no intent, SnippetGen uses the original FQN and Method Body.

Intent Sensitivity: The intent is closely related to past modifications, but the intent varies inconsistently with the number of the modifications. The more modifications are provided, the fewer common subset is likely to be shared, which results in the problems of over generalization. The fewer modifications are provided, the more common subset is, which results in the problems of over specification. Actually, instead of the number of the modifications, the intent strictly depends on the similarity among them. If the modifications are diverse, SnippetGen extracts the fewer common modifications and obtains the insufficient intent. If the modifications are similar, SnippetGen extracts the more common modifications and obtains the sufficient intent.

To reach the maximum similarity, we try two ways. First, we apply the heuristic algorithm to pick out the similar modifications from all modifications, so that the intent becomes more sufficient despite a big difference between a few modifications. Second, we use the threshold in LCEOS (as shown in the process of identifying common intent) to tolerate inexact matches among modifications. For example, if SnippetGen fails to find any common edit operation between two modifications, it generalizes all concrete instances of types, methods, variables and constants with abstract identifiers to match edit type or inheriting type.

VII. RELATED WORK

Early code search engines are the keywords-based information retrieval techniques [4]. For example , Google, Koders and Krugle allow a user to input keywords and perform the file-level retrieval. However, Myers [16] observed that these

code search engines always achieve inaccurate search results, because they are not designed to support programming tasks. To improve the accuracy, later work did semantic search. Originally, the work by Wing looked at matching function signatures [15]. Then it was extended to match more complete formal semantics using λ prolog and Larch-based [19]. But these techniques were impractical because either they attempted to do too little or too much. Recently, to promise better usability, several approaches have been proposed to improve the effectiveness of free-text code search via query expansion. For a vague input, adding one or more synonyms of the words appearing in the query can enhance the precision of search results. McMillan proposed Portfolio that takes natural language descriptions as "synonyms" and outputs a list of functions or code fragments along with corresponding call graphs [6]. Wang et al. [5] proposed an active code search approach which incorporates user feedback as "synonyms" to refine the query. Fei et al. [2] propose a latest code search technique that could understand the APIs a user query refers to and considers both text similarity and potential APIs. In addition, there are other query expansions either by using an appropriate ontology [20], natural language [14], or collaborative feedback [13].

These existing code search tools seem to yield semantically correct matches, but the search results might be too complex or too slow to meet the user needs. These results still has to be modified. But SnippetGen can retrieve the more relevant methods without any subsequent modification. Although our work is viewed as a query expansion, we differ from them. We give the code search engine the ability of intent predicting to guess what else the user might do after he obtains the search results. We incorporate intent as "synonyms" for expansion and consider the impact of both potential intents and text similarity on code search. Besides, we propose the refinement strategy in the intent extraction algorithm. This strategy can pick out the appropriate intents to benefit the code search.

VIII. Conclusion

In this paper, an intent-enhanced code search approach (IECS) is proposed. Based on the intent-relevant semantic and structural matches, it exploits the intent before performing code retrieval and allows a user to retrieve the relevant code by expanding the query with the intent. In the future, we plan to address the issues discussed in Section VII. For example, in the intent-enriched component, we either improve similarityCchoosing heuristic algorithm to ensure a sufficient intent, or employ the deep learning approach to make the intent become self-improvement.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under Projects No. 61373039, No. 61170022, No. 61003071 and No. 91118003.

References

[1] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," Empirical Software Engineering, pp. 1-33, 2016.

[2] F. Lv et al., "CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E)." IEEE/ACM International Conference on Automated Software Engineering pp. 260-270.2015

[3] J. Galenson et al., "CodeHint: dynamic and interactive synthesis of methods." International Conference on Software Engineering pp. 653-663.2014

[4] P. Fafalios, and Y. Tzitzikas, "Post-analysis of Keyword-Based Search Results Using Entity Mining, Linked Data, and Link Analysis at Query Time." IEEE International Conference on Semantic Computing pp. 36-43.2014

[5] S. Wang, D. Lo, and L. Jiang, "Active code search: incorporating user feedback to improve code search relevance." Acm/ieee International Conference on Automated Software Engineering pp. 677-682.2014

[6] C. Mcmillan et al., "Portfolio: Searching for relevant functions and their usages in millions of lines of code," Acm Transactions on Software Engineering & Methodology, vol. 22, no. 4, pp. 402-418, 2013.

[7] C. Carpineto, and G. Romano, "A Survey of Automatic Query Expansion in Information Retrieval," Acm Computing Surveys, vol. 44, no. 1, pp. 159-170, 2012.

[8] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In ICSE, 2011.

[9] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval: The Concepts and Technology behind Search. Addison-Wesley, 2011.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem." Communications of the Acm pp. 482-498.2010

[11] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. "Change distillingł tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering", 33 (11):18, November 2007.

[12] E. Linstead et al., "Sourcerer: mining and searching internet-scale software repositories," Data Mining & Knowledge Discovery, vol. 18, no. 2, pp. 300-336, 2009.

[13] Taciana A. Vanderlei, Frederico A. Durao, Alexandre C. Martins, Vinicius C. Garcia, Eduardo S. Almeida, and Silvio R. de L. Meira, "A cooperative classification mechanism for search and retrieval software components," Proc SAC'07, pp. 866-871 (March 2007).

[14] Christopher G. Drummond, Dan Ionescu, and Robert C. Holte, "A learning agent that assists the browsing of software libraries," IEEE Trans. on Software Engineering Vol. 26(12) pp. 1179-1196 (December 2000)

[15] A. M. Zaremski, and J. M. Wing, "Signature matching: a key to reuse," Acm Sigsoft Software Engineering Notes, vol. 18, no. 5, pp. 182-190, 1993.

[16] Y. S. Maarek, D. M. Berry, and G. E. Kaiser, "An information retrieval approach for automatically constructing software libraries," IEEE Transactions on Software Engineering, vol. 17, no. 8, pp. 800-813, 1991.

[17] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," Commun. ACM, vol. 26, pp. 1022C1036, 1983.

[18] J. W. Hunt and T. G. Szymanski. "A fast algorithm for computing longest common subsequences". CACM, 20 (5):350C353, 1977.

[19] 31. Eugene J. Rollins and Jeannette M. Wing, "Specifications as search keys for software libraries," Proc. 8th Intl. Conf. on Logic Programming, pp. 173-187 (1991).

[20] Haining Yao and Letha Etzkorn, "Towards a semanticbased approach for software reusable component classification and retrieval," ACMSE'04, pp. 110-115 (April 2004).