# Structurally Heterogeneous Source Code Examples from Unstructured Knowledge Sources

Venkatesh Vinayakarao Rahul Purandare
Indraprastha Institute of Information Technology, Delhi
{venkateshv,purandare}@iiitd.ac.in

Aditya V. Nori
Microsoft Research
adityan@microsoft.com

## Abstract

Software developers rarely write code from scratch. With the existence of Wikipedia, discussion forums, books and blogs, it is hard to imagine a software developer not looking up these sites for sample code while building any non-trivial software system. While researchers have proposed approaches to retrieve relevant posts and code snippets, the need for finding variant implementations of functionally similar code snippets has been ignored. In this work, we propose an approach to automatically create a repository of structurally heterogeneous but functionally similar source code examples from unstructured sources. We evaluate the approach on stackoverflow[1], a discussion forum that has approximately 19 million posts. The results of our evaluation indicates that the approach extracts structurally different snippets with a precision of 83%. A repository of such heterogeneous source code examples will be useful to programmers in learning different implementation strategies and for researchers working on problems such as program comprehension, semantic clones and code search.

***Categories and Subject Descriptors*** H.3.3 [*Information Search and Retrieval*]: Retrieval models; D.3.3 [*Language Constructs and Features*]: Control structures

***General Terms*** Code Search, Programs, Examples, Mining, Knowledge Representation, Similarity

***Keywords*** Example Retrieval

## 1. Introduction

Structural heterogeneity is very commonly observed in functionally similar source code written in high level programming languages such as Java. For example, *factorial* can be implemented either using recursion or loops or programming constructs such as BigInteger. Listings 1, 2 and 3 show examples for different implementations of a simple factorial program.

Unstructured sources such as discussion forums contain several examples in the form of partial program snippets that implement functionality relevant to the topic discussed. We propose an approach to extract such code examples. Henceforth, we refer to such examples that are well discussed as *familiar code*. We refer to a short text that describes the familiar code as a *topic*. Listing 3 shows a familiar code for the topic, "factorial". Knowledge of variant implementations plays a significant role in several research areas such as program comprehension, semantic clone detection and code search.

```
public static int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) {
        fact *= i;
    }
    return fact;
}
```

Listing 1: Factorial using loop

```
public static long calc(long n){
    if (n <= 1)
        return 1;
    else
        return n * calc(n-1);
}
```

Listing 2: Factorial using recursion

```
public static void main(string[] args) {
    BigInteger fact = BigInteger.valueof(1);
    for (int i = 1; i <= 8785856; i++)
        fact = fact.multiply(BigInteger.valueof(i));
    system.out.println(fact);
}
```

Listing 3: Factorial using BigInteger

Familiar code snippets frequently occur in massive code repositories. For example, Apache commons' BrentOptimizer.java[2] uses Brent's algorithm. As Sridhara et al. [17] claim, spotting such familiar code in massive code bases reduces the amount of code to read, and thus supports program comprehension. Knowing just one implementation or a few structurally similar implementations limits the amount of familiar code that we can spot. Therefore, we believe, an approach to mine structurally heterogeneous code examples will help program comprehension.

Structurally different code examples that exhibit same or similar behavior are candidates for semantic clones. For a given topic, we find structurally different and functionally relevant source code. Thus, our approach directly supports semantic clone detection. The

[1] http://stackoverflow.com/

[2] https://commons.apache.org/proper/commons-math/apidocs/org/apache/commons/math3/optimization/univariate/BrentOptimizer.html



*PEPM '15*, January 13–14, 2015, Mumbai, India.
Copyright © 2015 ACM 978-1-4503-3297-2/15/01…$15.00.
http://dx.doi.org/10.1145/2678015.2682537

Table 1: Comparison of Discussion Forum characteristics with Code Repository.

| # | Discussion Forum | Code Repository |
|---|---|---|
| 1 | Coding concerns common to several projects. | Project specific concerns. |
| 2 | Language How-tos. | Domain How-tos. |
| 3 | Partial Code. | Compilable Code. |
| 4 | Discussions on snippets. | Code & few comments. |
| 5 | Alternate approaches discussed. | Single approach & mostly missing reasoning. |
| 6 | Captures popularity and correctness of discussions explicitly in the form of responses, comments and ratings. | There is no way to know how many people have read or used a specific snippet from a specific repository. |
| 7 | Multilingual Connections. For example, gives Java version of C code. | No connection or references to other language implementations. |

output of our approach is a set of semantic clones exemplifying the given topic.

Code search tools need to deal with elimination of duplicate or near-duplicate results. Those tools that search over unstructured sources typically retrieve familiar code. They can improve their results by leveraging techniques to identify structural heterogeneity or just by using a repository of familiar code snippets.

In this work, we discuss the challenges and propose an approach to extract structurally heterogeneous familiar code examples for a given topic from an unstructured source. We implement a prototype using data from a popular discussion forum for a list of 12 topics. We use light-weight program analysis and information retrieval techniques so that partial programs can be effectively retrieved from large scale data sources. Finally, we evaluate the approach and show that structural complexity of familiar code can be used to further refine the results.

## 2. Mining Familiar Code Snippets

Many researchers have worked on code search over code repositories [2, 3, 10, 13]. Discussion forums have very different characteristics when compared to code repositories. We highlight some key observations that came out of our study in Table 1. Due to these fundamental differences, the approaches used to search a code repository cannot be used to search discussion forum. Also, we believe, due to these differences, use of discussion forum for extracting sample code should not be overlooked.

We take stackoverflow posts that contain a Java code snippet. We could have used any other high level programming language since our methods can easily be modified to work with any language. We extract code snippets and tokens (keywords from the natural language discussion around the snippet) from such relevant posts. We rank code snippets based on vocabulary similarity. Here, we also include the vocabulary elements from code snippet such as identifier and method names. We transform the snippet into a form that captures the structural information. For each code snippet, we check if a snippet's structure differs from its predecessors and thus remove snippets that are structurally duplicates. Further, we use structural complexity to detect and remove outliers. Figure 1 outlines this approach to mine familiar code snippets. The result of this approach is a set of structurally heterogeneous and functionally similar source code samples.

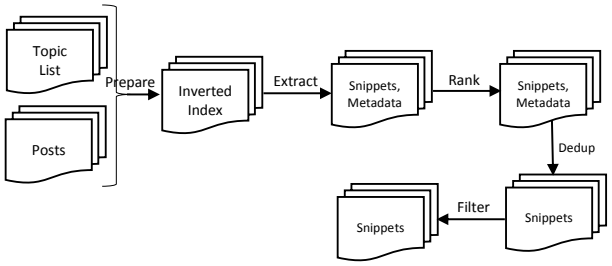



Figure 1: Steps showing the mining of sample code for a topic list from stackoverflow.

### 2.1 Identifying Posts with Java Snippets

Posts in stackoverflow are typically short text with code embedded in between. The convention in stackoverflow is to place code snippets inside *<code>... </code>* xml blocks. We retrieve such blocks that have at least one *code* element. Posts may contain partial programs i.e., the code snippet may refer to anything between uncompilable statements to fully compilable classes. Moreover, the embedded code could be of any programming language and not necessarily Java.

Research shows that method definitions play a major role in code search [15]. In this work, we limit our analysis to Java methods. We look for *public* or *private* keywords in the code snippet to identify methods. We wrap methods into a custom class template so that we can use Eclipse JDT[3] to parse them into an Abstract Syntax Tree (AST). If Eclipse JDT is able to parse it, we consider that the code snippet we have is indeed a Java code snippet. Several code samples exist without these keywords and our approach drops them. These may sound restrictive but we still ended up with 29,299 code snippets to analyze which is sufficient to pull distinct examples.

Code snippets that are not method definitions could still be useful and are acceptable as mined examples. Static analysis on partial programs [6] could help us identify partial Java code that do not have these modifiers (public or private) and thus enable us mine more examples. This is left for future work.

### 2.2 Collecting Keywords

Firstly, we tokenize text in the post with the objective of extracting representative terms. We use TF-IDF technique [1] to derive such terms. For this, we had to compute a set of unique terms in the post and the number of posts that they appeared in. In our case, since the text is very short, term frequency (TF) does not have any impact. Hence, it is reasonable to use the inverse document frequency (IDF) value to find the representative terms for a post. For any term $t$ in the entire corpus of posts $P$, *idf* is given by the following formula where $N$ is the total number of posts.

$$idf(t, P) = \log \frac{N}{|\{p \in P : t \in p\}|} \quad (1)$$

Secondly, we capture terms from the vocabulary used in the code snippet. From the ASTs, we capture vocabulary in the form of method name and variable names. There are variables that do not carry much semantic value such as the loop index variables. In this context, by semantic value, we refer to the natural language expressiveness of the variable. We observe that these variables are usually left as a single character such as *i* or *j*. There are cases where variables of length less than three carry semantic value. For

[3] http://www.eclipse.org/jdt/

Table 2: Precision before and after the use of sequence information.

| # | Case | Method#1 | Method#2 |
|---|---|---|---|
| 1 | Convert int to String | 50 | 75 |
| 2 | Factorial | 100 | 100 |

instance, $n$ typically refers to "number of ..." or "count", and $sd$ refers to "standard deviation". For this work, we ignore such short variables. Without loss of generality, we chose to drop all terms that had any non-alphabetic character. This step of filtration does not impact our approach or results. Leveraging state of the art vocabulary manipulation techniques such as Normalize [21] can help us improve our work.

### 2.3 Relevance and Ranking

Now that we have a collection of code snippets and associated metadata in the form of keywords, our next task is to rank them based on relevance to topic. We tokenize the stackoverflow post title. We also grab the variable names and method name from the code snippet. We compare the topic with these data (tokenized title and elements from code snippet) and quantify the relevance.

Firstly, we study the snippets ranked based on term frequency (TF) score. We observe three major issues:

1. **Favoring long posts**: Long posts tend to have more terms with high frequency and hence such long posts are preferred over shorter posts.
2. **Sequence information is ignored**: Sequence of terms in topic is important for retrieving relevant results.
3. **Specialization problem**: There are terms such as *Array* and *Array List* that require different results even though they contain some terms in common.

Research on quality of example programs [4] shows that users prefer shorter examples. Term Frequency has the disadvantage of bias towards longer documents [20]. To address this concern, we use augmented term frequency $tf(t,p)$ computed as follows:

$$tf(t,p) = \frac{f(t,p)}{1 + max\{f(t,p)\}} \tag{2}$$

where $f(t,p)$ represents the raw frequency of term $t$ in post $p$.

Consider the topic *convert int to string*. This is not same as *convert string to int*. If we go with just the term frequency, results relevant to either of these queries will score the same. To avoid this, we compute longest common subsequence metric for each result. The intuition is that if the terms "convert", "int" and "string" appear in the same order in the result as well, the result scores higher. Table 2 shows the precision for two topics before and after the application of this method. Note that while this approach does not impact *factorial*, it had a positive impact on *convert int to string* topic.

Our ranking model is as follows. Let $w_1, w_2..., w_n$ denote any vector of terms that represents a post $p$. Let $t_1, t_2, ..., t_k$ be the topic vector representing a topic $T$. The distance function $\phi$ is defined as follows:

$$\phi(p,T) = \lambda \sum_{i=1}^{k-1}((loc(t_{i+1},p) - loc(t_i,p)) \tag{3}$$

The function $loc(t,p)$ refers to the location (or index) of term $t$ in post $p$. $\lambda$ denotes heuristically derived weight. For a match in title, we believe the post should be more relevant and hence have a higher $\lambda = 6$ and for code snippet match, we keep $\lambda$ as 3. We add up the scores computed for title and code snippet to get the final score for a post p as:

$$\phi(p,T) = \phi_{title}(p,T) + \phi_{snippet}(p,T) \tag{4}$$

*Binary Search* is not the same as *Binary Search Tree*. Similarly, *Array* is not same as *Array List*. We observe that TF based approach is unable to differentiate these kinds of results. Judges did not like the results of *sort array* since they saw results relevant to *sort array list*. We refer to this non-trivial problem as a specialization problem and leave this as future work.

Table 3: Structural information extracted from source code.

| Code | Structure |
|---|---|
| `public static int factorial(int n) {` | `<algo>` |
| `  int fact = 1;` | `<loop>` |
| `  for (int i=1; i<=n; i++) {` | `<=` |
| `    fact *= i;` | `*=` |
| `  }` | `</loop>` |
| `  return fact;` | `</algo>` |
| `}` | |

### 2.4 De-duplication

Our assumption is that several short code snippets are heavily reused. Thus, snippets of same structure are expected to show up as the result. Current research [3] has used structural similarity to predict functional similarity. We have the reverse objective. Out of all the structurally similar examples, we wish to retain only one result.

To compute structural similarity, we flatten the structure into term-like items. Sridhara et al. [17] group structural elements into three significant categories: return, conditionals and loops. We use conditionals and loops. We leave *return* for future work. Table 3 illustrates one sample run of structure generation. Each code snippet $C_i$ is transformed into a vector of structural terms $c_j$. Structural similarity $similarity(C_1, C_2)$ between two code snippets $C_1$ and $C_2$ is computed as follows:

$$similarity(C_1, C_2) = \frac{|C1 \cap C2|}{max\{|C1|, |C2|\}} \tag{5}$$

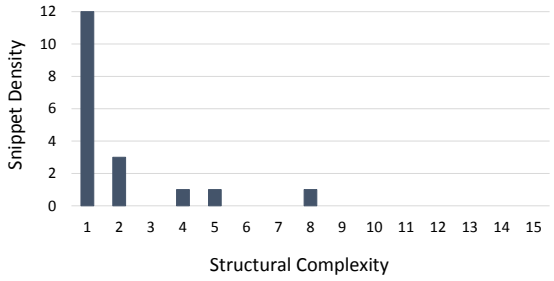
Note that for any pair of identical factorial code snippets, we arrive at the maximum possible similarity score (of one). A recursive version of factorial when compared with the iterator based version will have very few common structures, thereby resulting in a smaller similarity score.

### 2.5 Structural Complexity

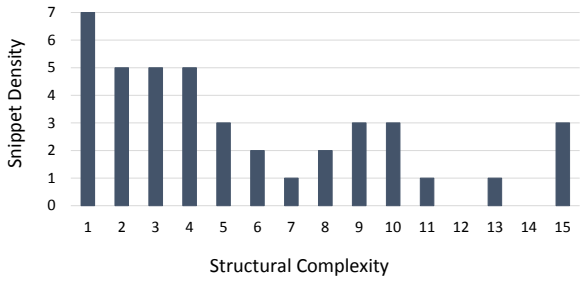
Our intuition is that the snippets that are functionally similar would have programming constructs that are similar in terms of their structural complexity even though they may differ considerably in their structures.

We observe the structural elements in familiar code and learn how they contribute to structural complexity. For instance, loops do not change the structural complexity irrespective of which form they take such as *while* or *for*. Our approach to approximate both of them to a generic *loop* as shown in Table 3 helps us to take advantage of this similarity. However, several such structural patterns exist in source code that could be leveraged to compute structural complexity. For instance, loops can be flattened to *switch* or *if* statements especially for small index values of loops. There are many familiar code examples that have a recursive equivalent of loop form.

We compute structural complexity in terms of Structural Lines of Code ($LOC_{ST}$). Here, we compute the number of statements in the flattened version of the source code (referred as structure




(a) Factorial: Most examples are of low structural complexity.




(b) Binary Search: Widespread structural complexity values.

Figure 2: Structural complexity characterizes the topics. Note that, in general, *Factorial* is less complicated structurally than *Binary Search*. Factorial code samples in the $LOC_{ST}$ buckets of 4 and above are most probably irrelevant and can be treated as outliers. For *Binary Search*, the outliers are likely to be in the first $LOC_{ST}$ bucket since there are sufficiently large number of samples in high-complexity buckets.

in Table 3). We categorize statements as Loop(l), Assignment (a), Declaration(d), Call(c), Recursive Call(rc), Return(r) and Expression(e) and assign a weight of $l = 5, a = 2, d = 2, c = 3, rc = 4$ and $r = 0$. These weights are manually derived by observing several code examples. We observe that complexity of expressions is directly proportional to the number of operators used in the expression. Hence we assign $e$ to the number of operators. A simple coarse grained model with such weights works for us. For a code snippet C that has statements $< c_1, c_2, c_3, ...c_n >$ with corresponding type dependent weight $< w_1, w_2, w_3...w_n >$, $LOC_{ST}$ is computed as follows:

$$LOC_{ST}(C) = \sum_{i=1}^{n} w_i \tag{6}$$

We compute structural complexity as discussed above on all the code snippets mined so far. Our intuition is that, in-spite of being structurally heterogeneous, the structural complexity of the code snippets (relevant to a specific topic) should not vary significantly. So, irrespective of whether recursion or loops are used, factorial code is simple by structural complexity when compared with implementations of binary search. Figure 2a suggests that most of the relevant code snippets for factorial have very low structural complexity. We learn from Figure 2b that code snippets that implement binary search tend to be more complex than those that implement factorial. At the same time, we also see several very simple implementations grouped in the first bucket of structural complexity, for binary search. These turned out to be simple snippets such as node and tree definitions that are relevant to binary search tree discussions. As we strengthen our methods to find more and more relevant code snippets, we will see concentrated regions in the structural complexity charts. We observe that the structural complexity gives a reasonable metric to detect outliers. Outliers are typically code samples that are either too simple such as empty methods, or extremely long and complicated methods. These become candidates because of high vocabulary similarity. We treat examples that lie in the sparse regions away from the skyscrapers as outliers. So, for factorial, any example with $LOC_{ST}$ of 4 and above are considered to be outliers.

## 3. Experimental Setup

We asked 13 graduate students working in the computer science department with at least 6 years of exposure to programming, to give us one query for a typically searched code snippet and one recently searched query, thereby leaving us with 26 queries. We dropped some of the responses which we believe would not have Java implementations. For example, "sort a python dictionary by value" was dropped. This resulted in 12 topics. We mine code examples for these from stackoverflow's XML dump of all posts. On this data dump, we apply the steps detailed in Section 2 i.e., preparation, keyword collection, ranking, de-duplication and structural complexity based filtering. We analyzed over 19 million posts, found 29,299 code snippets that were Java methods. We mined 523 implementations in total for the 12 topics that we started with.

## 4. Evaluation

Our evaluation criteria consist of two parts. Firstly, we check the relevance:

- The code snippet must show how to implement the given topic in Java. Sometimes, an API call to existing library will also be sufficient and hence it is considered as a good solution.
- A code snippet may not be implementing the given topic alone. For a query like *file reading*, a code snippet that shows how to print numbers from a file is considered as a good example.
- Example retrieved may partly implement the topic. For instance, there are several options while opening a file in Java. Example need not be exhaustive to be relevant. However, just the first two lines of file reading is not considered to be a relevant and good example.
- Code snippets that neither demonstrate the use nor exhibit an implementation of given topic should be marked as irrelevant.

Next, we ensure that the results are structurally diverse.

- Example retrieved should not be similar to an already retrieved example. Two examples should be marked as being structurally similar using the following guidelines:
  - if we rename the variables and remove white spaces and change the indentation, the examples become the same.
  - irrelevant code inserted or deleted in between is not considered as a variant example.
  - use of different api should be considered as a variant example.

Three graduate students evaluated all the results and marked them on a binary scale as relevant/irrelevant. We used this data to compute precision. Firstly, we checked the precision we get if we do not use any technique apart from string match of topic name in stackoverflow post vocabulary. The precision varied from 17% (convert integer to string) to 93% (for factorial) with a mean of 63% and standard deviation of 22%. Hence, the topic set has no bias towards the potential for string match.

Table 4: We are able to extract variant implementations with a precision of 83.3%. Count column shows the number of example snippets retrieved. Precision demands both functional relevance and structural heterogeneity.

| # | Topic | Count | Precision |
|---|---|---|---|
| 1 | Binary Search | 7 | 100 |
| 2 | Check file exists | 4 | 50 |
| 3 | Compare objects | 7 | 75 |
| 4 | Convert integer to string | 2 | 75 |
| 5 | Deep copy | 3 | 100 |
| 6 | Factorial | 5 | 100 |
| 7 | File reading bufferedreader | 2 | 50 |
| 8 | Keyboard input | 3 | 100 |
| 9 | Merge sort | 2 | 100 |
| 10 | Palindrome | 5 | 100 |
| 11 | Reverse string | 9 | 100 |
| 12 | Sorting array | 3 | 50 |

## 5. Results

We have extracted structurally heterogeneous and semantically similar code snippets for 12 topics with a precision of 83.3% with a mean and median of approximately four examples for each topic. Table 4 shows the number of structurally diverse results obtained from our approach and its corresponding precision, for each topic. From these results, we are convinced that the availability of multiple examples allows us to leverage dominant structures, variables and words (from post body) and score each snippet. We find stackoverflow to be a rich source for demonstrating an approach to mine heterogeneous examples. We also observe that structural complexity characterizes the topics and therefore can be used to filter outliers as shown in Figure 2 and discussed in Section 2.5.

Lexicons play a major role in identifying APIs that are frequently used to do key functions. For example, BigInteger is typically used in factorial computation in Java. Our method found that BigInteger is popular in the context of factorial and retrieved it (shown in Listing 3) as an example. When multiple people refer to a BigInteger based factorial implementation, an understanding of technicalities behind BigInteger is not required to believe that the API plays a role in factorial computation. This goes well with the natural way humans comprehend software.

Lack of context is a key challenge. Our topic *check file exists* has ambiguous context. It may refer to file on local disk or file on remote server. Most of the judges felt that former was the need and judged the latter implementations as irrelevant.

We also observe the issue of picking a heavily correlated wrong example from the post. For instance, file reader and file writer are discussed together and they end up getting similar score.

***Threats to Validity*** Our current evaluation uses 12 topics. We have showed that our topics have no bias from the context of vocabulary matches. It is still possible that we get a lower precision when we increase the number of topics. Moreover, the way the topics are worded could impact the results. Topics need to support search. We assume that users can try out a few queries to quickly get to good results. We have used a single discussion forum as our source to mine snippets. A different discussion forum might impact results. As long as the forum is large and if it reflects the same properties as listed in Table 1, we believe the results should not be very different. Our assumption that the variables of length one or two does not hold semantic value does not hold in all cases. If the topics are such that the relevant code samples use several abbreviations of length one or two, our approach may result in poor precision.

## 6. Related Work

Prompter [12] and SeaHawk [11] use stackoverflow to help developers (through an IDE extension) with relevant posts using the current code context. They construct queries based on terms in current context. They use several attributes such as textual similarity, code similarity and user reputation to suggest relevant post. However, their work stops at retrieving API names and code-like words. Rigby et al. [14] extract elements of code from informal discussions. Cordeiro et al. [5] extract relevant posts based on stack trace of exceptions. Instead of posts and code-like elements, we extract heterogeneous source code examples for given topics and thus our problem, method and techniques are different.

Sourcerer [2] is a code search engine that collects open source projects, and uses vocabulary and structural information to search for code. Holmes et al. [9] extract examples from source code repositories. Holmes and Murphy [8] use program context to find relevant source code from existing repository using structural information. They work with fully compilable code and use call hierarchy based API usage information. Their techniques cannot be used on code fragments available on discussion forums. Discussion forums have partial yet re-usable code that is better for learning when compared with examples extracted from source repositories since they have discussions associated with them. We leverage these discussions to find source code examples for a given topic.

AlgorithmSeer's [18] algorithm search works on the intuition that the algorithms discussed in scholarly documents are related. A similar idea has been applied on source code using Latent Semantic Indexing and Structural Semantic Indexing [3]. Gulwani et al.'s [7] Programming by Example (PBE) attempts algorithm discovery for automating repetitive tasks. While this uses text as input/output, they have also applied similar idea to automatically suggest corrections to student assignments which are essentially source code [16]. However, this approach works with a large example set of submitted solutions and compares with other solutions. Since the assignment is same, structural similarity serves as a good measure to solve this problem. In contrast, we look at structural dissimilarity to pull up distinct examples for the same topic. We are particularly inspired by the Zhang et al.'s [19] idea of algorithmic comparison. In our work, missing call hierarchy information makes construction of value dependency graphs, impossible. Further, we do not use input/output values or even execute the code samples. Our approach leverages light weight information retrieval and program analysis techniques. Thus, we are able to process several thousand code samples within a few seconds.

## 7. Conclusion and Future Work

Our initial contribution yields a reasonably accurate approach to mine structurally heterogeneous and functionally similar code snippets from heterogeneous sources. We exploit information retrieval and partial program analysis techniques to arrive at a repository of such code samples. We show that it is possible to retrieve such samples from unstructured sources and discuss the challenges. We have used stackoverflow posts for our research. The same approach can be applied on wikipedia, blogs and books as well.

Understanding the fundamental elements of programming from the perspective of variants of implementations and the way they show up in code opens up new ways to solve problems such as semantic clones. The ability to spot familiar code in massive code bases can not only add value directly to program comprehension but also have several other useful applications. We find the resultant code samples to demonstrate a very high educative and illustrative value. This can be used in designing programming language course texts.

We plan to conduct a study comparing our results against the benchmarks for code retrieval tools such as ohloh[4] and post retrieval tools such as Prompter [12]. We have left the following as future work:

1. Application of information retrieval techniques to address issues such as specialization problem and support non-alphabetic terms in vocabulary. We will also benefit from a vocabulary normalization algorithm such as Normalize [21].
2. Application of partial program analysis techniques towards detecting code and better extraction of structural elements.
3. Structural complexity computation should support program constructs beyond Java method definitions.
4. Examples lie in contexts. For instance, *reading a file* is not same as *reading an integer list from a file* while the latter might still be acceptable as an example of the former.
5. We use a static topic list. To increase the scale, we will need a long list which could be a result of another automated approach.
6. Concepts such as *greedy algorithm* are context specific. For example, the idea of greedy algorithm can be applied on Huffman tree construction during Huffman coding. Retrieving samples for such topics require domain knowledge.

Classic program analysis approaches fail to capture semantics at scale and thus makes a good case to apply information retrieval techniques. Our approach should scale very well to extract code examples in a language independent fashion. While precision seems to be good, recall is still an issue. State of the art program analysis and information retrieval techniques can further help to improve recall. We will need to show that ideas from information retrieval around context extraction can be extended to source code.

## Acknowledgments

This work is supported by Confederation of Indian Industries (CII) and Microsoft Research. We thank Dr. Matthew Dwyer and Dr. Sebastian Elbaum for their suggestions to improve this work.

## References

[1] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: A search engine for open source code supporting structure-based search. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 681–682, New York, NY, USA, 2006. ACM.

[3] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166, New York, NY, USA, 2010. ACM.

[4] J. Börstler, M. S. Hall, M. Nordström, J. H. Paterson, K. Sanders, C. Schulte, and L. Thomas. An evaluation of object oriented example programs in introductory programming textbooks. *SIGCSE Bull.*, 41(4):126–143, Jan. 2010.

[5] J. Cordeiro, B. Antunes, and P. Gomes. Context-based recommendation to support problem solving in software development. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 85–89, June 2012.

[6] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008. ACM.

[7] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.

[9] R. Holmes, R. J. Walker, and G. 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, pages 237–240, New York, NY, USA, 2005. ACM.

[10] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[11] L. Ponzanelli, A. Bacchelli, and M. Lanza. Leveraging crowd knowledge for software comprehension and development. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 57–66, March 2013.

[12] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 102–111, New York, NY, USA, 2014. ACM.

[13] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society.

[14] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 832–841, Piscataway, NJ, USA, 2013. IEEE Press.

[15] S. Sim, C. Clarke, and R. Holt. Archetypal source code searches: a survey of software developers and maintainers. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 180–187, Jun 1998.

[16] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM.

[17] G. Sridhara, L. 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, pages 101–110, New York, NY, USA, 2011. ACM.

[18] S. Tuarob, P. Mitra, and C. L. Giles. "Building a Search Engine for Algorithms" by Suppawong Tuarob, Prasenjit Mitra, and C. Lee Giles with Martin Vesely As Coordinator *SIGWEB Newsl.*, (Winter):5:1–5:9, Jan. 2014.

[19] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 111–121, New York, NY, USA, 2012. ACM.

[20] A. Singhal, C. Buckley, and M. Mitra. Pivoted document length normalization. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96, pages 21–29, New York, NY, USA, 1996. ACM.

[21] D. Lawrie, D. Binkley, and C. Morrell. Normalizing source code vocabulary. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 3–12, Oct 2010.

[4] https://code.ohloh.net/