Concept Extraction: A Modular Approach to Extraction of Source Code Concepts

Ritu Chaturvedi
University of Guelph
chaturvr@uoguelph.ca

Veerpal Brar
University of Toronto
v.brar@mail.utoronto.ca

Jai Geelal
University of Toronto
jai.geelal@mail.utoronto.ca

Kelvin Kong
University of Toronto
kelvin.kong@mail.utoronto.ca

***Abstract*—Code examples have always been one of the most sought after pieces of information when it comes to understanding and mastering programming concepts. Research shows extracting knowledge from such examples in any online tutoring system is a challenging task. Current methods rely upon specifically formed regular expressions that must be tailor made to the input language, or generation of an AST for the given input program. In our paper, we extend upon existing implementations in code recommendation software using a novel keyword based search tree (k-BST) method. K-BST recommends relevant code fragments by extracting existing keywords , matching with relevant coding examples by k-means clustering, and recommending the relevant coding examples back to the user. K-BSTs also address several major issues which modern knowledge extraction software often run into, like ease of use, extendibility to other domains and run time. With that in mind, K-BSTs are designed to tackle ease of use with popular recognizable file formats such as CSV while keeping the run time of extracting relevant keywords to be extremely low (compared to the more popular method that uses AST).**

I. Introduction

According to Unbehauen et al. [11], Knowledge Extraction is an automated process of knowledge creation from structured (relational databases, XML) and unstructured (text, documents, images) sources. Knowledge in online tutoring systems such as Intelligent Tutoring Systems (ITS) [2] is defined in terms of building blocks of the ITS domain such as lessons, exercises, tasks, worked-out examples, their solutions and more importantly, the learning units that define them and are acquired by domain experts.

The following definitions are adapted from an earlier work [2].

Definition 1: Domain of an ITS refers to the subject or course that the ITS is designed to teach (mostly within a limited scope) (e.g. domain of Python programming).

Definition 2: A worked-out example (WE) refers to a complete or partial worked-out solution of a question or instruction (similar to examples in textbooks).

Definition 3: A learning unit (LU), also referred to as a topic or concept, is the smallest basic unit of knowledge in that domain that a worked-out example or a task solution is divided into. For example, "scanf" is an LU in the domain of C programming. Similarly, "fraction" is an LU in the domain of Math.

In the domain of programming, the extraction of learning units from worked-out code examples involves parsing source code for different coding concepts such as loops, conditionals, and assignment. Current methods for extracting learning units from code (as discussed in related works section) tend to be restricted in their application or use some variation of an abstract syntax tree (AST). An AST is a tree representation of source code where nodes represent different syntax of the source code. An AST can be traversed in order to determine all of the learning units in the source code. However, building an AST requires the user to specify the grammar of the language they are parsing which can be difficult and time-consuming. Furthermore, while an AST returns accurate results it often contains more information than is required for successful knowledge extraction.

In the domain of beginners programming as introduced in an introductory computer science course, the extra information provided by an AST is unnecessary for extracting learning units from source code. We propose the structured syntax of a programming language means that learning units such as loops, conditionals, and assignments can all be identified based on the presence of certain keywords within the source code. Here, a keyword is defined as basic strings of characters that are essential to the identification of a certain learning unit. For example the keyword "for" is vital in identifying the for loop learning unit.

Therefore, this paper aims to determine if a specific keyword based extraction of a program's concepts be comparably accurate and/or more efficient than the commonly used AST? In particular, this paper will introduce a new method called the Keyword Based Search Tree (k-BST) in order to answer the above question. A k-BST is a tree that functions similarly to a n-ary search tree where the values of the nodes are used to traverse the tree. However, the leaf nodes of a k-BST are the names of learning units and the k-BST uses the presence of certain keywords to determine which path to follow from the root to a learning unit node.

The goal of knowledge extraction (KE) typically is to create knowledge and represent it in a machine-readable and machine-interpretable format [11]. In a typical ITS, the goal of KE is to represent worked-out examples in its domain in terms of learning units they are made of. The proposed model in this paper transforms each task and example into a binary vector of size n, where n = number of LUs defined by the domain expert. The binary vector is created by using a k-BST to parse the source code and extract all learning units.This is explained further in the proposed methodology section.

The goal of knowledge organization is to organize the knowledge contained in the ITS in an easy to understand and informative way. In our proposed method, after the learning units are extracted, k-means clustering is used to cluster the binary vectors into groups of related worked-out examples and presented a user to explore. Each cluster will contain examples that share similar learning units. Thus, when a user explores a cluster they can see how the different learning units a commonly used in programming. For example, a cluster of examples that use while loops would showcase the various uses of while loops as well as the relationship between assignment statements and while loops. The process of clustering is explained in the proposed methodology section.

II. Related Works

There is a large body of work involving the extraction of learning units or concepts from code, for various purposes. A common motivation is analyzing code to discover variation [4], [7] or similarity in student solutions [3]. The Overcode software converts a program into a set of lines and then creates stacks of similar sets [4]. However, this involves extensive data prepossessing where the code has to be traced and reformatted to contain similar variable names and formatting as other code. The keyword based search tree method does not require any prepossessing of the code before the extraction method. Another method of program analysis used is to break the program into smaller blocks using a control flow graph and then analyzing the smaller blocks by either hashing mini abstract syntax trees [7] or using equivalent expression directed acyclic graph (ee-DAG) [3]. The limitation of these methods is that they use the Jimple library specific to Java in order to generate the control flow graphs. Also, while it can be determined that two pieces of code are similar, its may not be possible to determine what exact similarities the two programs share. The k-BST algorithm is not language specific and informs the user of the exact learning units that exist in the code. A recent method of program analysis used by the Cobra tool involved dividing the program into lexical tokens which are then used to create a linked list where each node contain tags about the token [5]. Thus, the linked list can be used to query the code and analyze the code (potentially using it to find learning units). However, similar to an AST this method involves storing information about each piece of the code even if it is not relevant to identifying learning units.

In another paper, Zhu et all wanted to find the frequency occurrence of certain statements in Java, C, and C++ programs. In order to count the occurrence of each statement, they converted the source code to an XML file using a library called srcML. XML is a data storage format that has similar syntax to HTML but does not contain predefined tags. The srcML library preserves the original code but adds XML tags in the code. The tags are reflective the abstract syntax of the language. XML allows for nested tags so code structure is preserved and nested statements are easy to identify. Furthermore, since XML is easy to parse, Zhu et all were easily able to count the frequency of tags by extracting the relevant XML tags from the XML file generated by srcML [14]. However, srcML only supports Java, C, and C++ programs. In order to apply this method to other languages, one would have to create a new library that supports their language of choice. Furthermore, the generated XML file contains tags that are not necessary for statement identification. This leads to a verbose XML file compared to the original source code. Our proposed algorithm leverages the efficient parsing and extraction methods of XML while avoiding the limitations of the srcML library.

The algorithm used in Code fragment summarization [13], summarized code examples using its syntactic features in the code by using an AST. For our algorithm, instead of using the popular syntax extraction method, AST, we parsed through the source code and stored only the learning units present in a binary array. Implementing the algorithm this way, gave us the advantage to be more efficient in the sense of saving memory, and reducing complexity for users to be able to extract relevant learning units from code examples. Our algorithm is also designed to be user friendly. Users can change the specific learning units and keywords they are looking for within their scope by simply editing an XML or CSV file. This allows users to be flexible with their learning unit selection while allowing them to be able to change learning units quickly and easily. This implementation also allows users to change our algorithm to search for more distinct learning units such as syntax from other languages other than Python.

Our algorithm also offers distinct advantages over typical keyword extraction algorithms. In the article Keyword extraction and clustering for document recommendation conversations [9], they used Word frequency and TFIDF to identify Keywords. In the implementation of our knowledge extraction algorithm, we parse the code example provided by the user line by line until we extract all of the learning units present in the current example.

In order to keep our algorithm modular and scalable for future extensions, we implemented the knowledge organization without the use of external data relationship matchers. Knowledge extraction portions of recommender algorithms are often outsourced to external sources to relate to relevant key terms. In the article Sound and Music Recommendation with knowledge graphs and Keyword extraction [10] and clustering for document recommendation conversations both use online data sources to match related documents to their tagged examples. While this method has more relationships available to them it greatly limits the user's control over how the matching is done. The algorithm we designed uses our internal matching system to match Keywords from code fragments to other relevant code examples. We have full control of how and when the matching of code fragments is done.

In order to achieve a high level of accuracy in their extraction, Hosseini et al used an Abstract Syntax Tree in order to parse the given source code. Using the Eclipse Abstract Syntax Tree framework, JavaParser [6] was able to convert the given source code into a syntax tree which contained the structural elements of the code such as: super method invocations, return types, variable declarations, parameters and exceptions. JavaParser then parsed the resulting AST based to extract the desired concepts from the source code. Results of the JavaParser showed that JavaParser has 93% [6] accuracy in comparison to manual parsing of source code. JavaParser presented the benefit of extracting all fine details of an application which in this case allowed the test subjects to be exposed to all syntactical elements of the source code. However, JavaParser which developed specifically for Java source code, and the use of an Abstract Syntax Tree could be resource intensive based on the source code being examined.

The Ontology-Based Architecture with Recommendation Strategy in Java Tutoring System [12] presented several interesting ideas which were captured and improved upon in the k-BST implementation. Vesin et al proposed the idea of improving upon their previous implementation Programming Tutoring System (PROTUS 1.0) by making PROTUS 2.0 a general tutoring system which was not language dependent. Furthermore, the use of Semantic Web Standards promotes a common ground of usability and modularity which allowed for simple interfacing and use. Furthermore, the use of the AprioriAll algorithm allowed for personalized student tutoring.

Abebe et al's method in Extraction of domain concepts from the source code [1] combined the extracted solutions from both a Natural Language Parser as well as a Structure Based Extraction. Our implementation of k-BST did not use any natural language techniques, so we focused solely on the extraction from structural properties. Abebe et al concluded that based on the strict syntactical requirements of object oriented programming, they are able to extract conceptual elements(learning units) from the source code. Abebe et al's method of extraction using structural ontology uses seven rules to determine the properties of the given source code [1]. The method of extraction in this paper lacks the ability to extract specific conceptual ideas based on the seven rules presented. This method is not able to extract specific concepts which are targeted in CS1 lectures such as loops, control statements, or print statements.

III. Proposed Methodology

We propose that each learning unit can be identified through the presence of certain keywords. For the domain of beginner level python programming, we identified 14 learning units, each of which had about 2 keywords associated with it. For example, the "simple print" learning unit had the keyword "print" associated with it while the "complex print" learning unit had the keywords "print" and "format" associated with it. An example is said to contain a learning unit if all of the keywords associated with the learning unit can be found in a *single* line of the source code. In the k-BST, the learning units are contained in the leaf nodes. The path from the root to a leaf node will contain all of the keywords associated with the learning unit of that leaf node. In order to maximize the efficiency of the k-BST the first level of the k-BST is said to contain "top-level" keywords. Top-level keywords are keywords that partition the learning units into subgroups such that a line containing one top-level keyword will not contain any other top-level keyword. For example, both "for" and "if" are top level keywords as a line containing the keyword "for" will not contain the keyword "if" as for loop and if statements can be contained in the same line.

For all subsequent levels, an effort to choose keywords that partition the learning units should be made as well, however, it is not a requirement of the k-BST.

A. Building a k-BST from CSV file

The learning units and associated keywords are stored within CSV or XML files. One way to update/change the algorithm uses for learning units is to update the defined learning units within the files. The CSV file is intended for users who want to change the learning units but are not familiar with XML formatted files. The learning units and associated keywords are then extracted from the CSV and a k-BST is built to represent it. The k-BST in Figure 1 is an example of a k-BST with the learning units as its leaves and associated keywords as the parents.

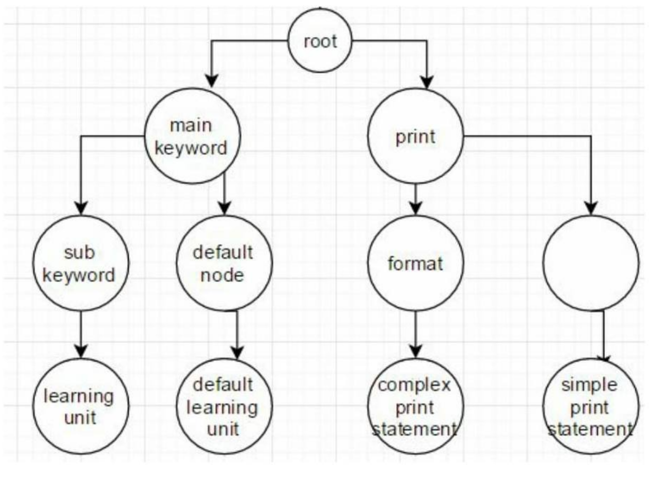

Figure 1. KBST

B. Extraction using k-BST

Once the CSV/XML file has been parsed into a k-BST, we obtain a graph that resembles Figure 1, which shows learning units(LU) followed by the keywords that define each LU. Given any piece of Python code, we can now use the k-BST to extract the LUs contained within the given code. The k-BST algorithm accepts Python source code as its input and outputs a binary vector where the presence of a 1 denotes the LU is present in the code, whereas the presence of a 0 indicates the given learning unit is not present.

The Proposed k-BST Algorithm

```
create empty vector array
open file with code
for each line in the code:
   start at root
   key = first child of root
   while key is not a learning unit node:
       if key found in line:
           key = first child of current key
       else:
           key = next sibling of current key
           if no next sibling:
               break while loop
   if learning unit found:
     set appropriate index in vector array as 1
return learning units
```

The k-BST algorithm loops through every line of code in the input source code, and for each line, based on the results of the preliminary matching, k-BST then determines if it should perform a deeper search, or move to check the presence of another LU. If a topLevel keyword is present, k-BST performs a deeper search to extract a specific learning unit whereas if the topLevel keyword is not detected, k-BST continues onto the next line of code. Upon reaching the end of the input file, k-BST knows the extraction is completed, and passes the extracted binary vector onto the Knowledge Organization module.

C. Tracing An Example Using k-BST

Sample Input Code to Add Two Numbers. Code 1.1

```
1  # This program adds two numbers
2
3  num1 = 3.14
4  num2 = 4.13
5
6  # Add two numbers
7  sum = float(num1) + float(num2)
8
9  # Display the sum
10 print('The sum of {0} and {1} is {2}'.
11 format(num1, num2, sum))
```

Tracing

For the purposes of this example, we can assume the expert has already provided a properly formatted XML/CSV with the appropriate learning units defined based on the expectations for a CS1 class. The *k-BST* algorithm will then scan the top of the application starting from line one. It is important to note that *k-BST* is designed to ignore docStrings and comments in its parsing due to the possibility of producing false positives. In the example given, Lines 1 and 2 will be ignored. Line 3 will result in the LU *Simple Assignment* being recognized. As *k-BST* moves onto Line 4, *Simple Assignment* will once again be recognized, but since it was previously detected, no changes are made. Again, lines 5 and 6 will be ignored and as *k-BST* progresses onto Line 7, it detects the assignments of the sum of two variables, which by the provided definition is a *Complex Assignment*. As *k-BST* proceeds, lines 8 and 9 are ignored, and Lines 10/11 triggers the detection of a *Complex Print* statement as the provided XML definitions define the use of the *.format* string as an indication of a complex print statement. Therefore, visually, the user will see the output as: [complexPrint, Complex Assignment, Simple Assignment], however, a binary vector representing this is also outputted which is sent to the Knowledge Organization module for further processing.

D. Knowledge Organization with Clustering

Once learning units have been extracted, the examples need to be organized such that groups of similar examples can be found and presented to the user. To do this, k-means clustering is used to divide the n binary vectors into k clusters. The k-means clustering algorithm involves first randomly choosing k binary vectors to act as the centroids for k clusters. Next, each binary vector is assigned to the same cluster as the closest centroid. The distance is measured using Jaccard distance, as it as been shown to work well with binary data [2]. The Jaccard distance between two binary vectors a and b is defined as

$$J = \frac{M_{11}}{M_{01} + M_{10} + M_{11}}$$

where M_{11} is the number of times both a and b have a value of 1, M_{01} is number of times a has a value 1 and b has the value 0, and M_{01} is number of times a has a value 0 and b has the value 1.

Once each binary vector has been assigned to a cluster using the Jaccard distance, the centroid of each cluster is recomputed to equal the mode of all the vectors in a cluster. Note that k-means usually involves computing the mean of a cluster when dealing with real valued vectors but is commonly replaced by the mode for binary vectors [2]. Once the centroids are recomputed, all the binary vectors are reassigned to new clusters based on their distance to the recomputed centroids. This process of computing centroids and assigning clusters continues until the location of the centroids no longer changes.

When a new example is added to the database, k-means clustering is performed on all the examples and the cluster assignment of each example is stored. When a user views an example, similar examples are also suggested to the users. These similar examples are suggested based on the other examples found in the same cluster as the current example.

IV. Evaluation of k-BST

Based on the results achieved from running k-BST on over 500 examples of Python code aimed at CS1 students, the k-BST algorithm was able to extract the relevant learning units with 100% accuracy. The major difference between the k-BST results and the results of an AST generator was the specific nature of the results from the AST. Having the predefined XML/CSV input file allows k-BST to match specific concepts to a general learning unit. However, an AST simply returns all syntactical elements of the program and does not classify them into learning units. Taking *Code 1.1* under consideration, the following would be the outputs from k-BST and an AST.

k-BST

Vector Array: [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

learning units: [complexPrint, Simple Arithmetic, Complex Assignment, Simple Assignment, Python Functions]

AST [8]

Syntax: Sequences,.,Function Call,Variable,Numbers,=,+ Builtins: print(),float() Tags: <Call(Call())>, Initial,<Function Call (with arguments)>,Assignment, Floats,<Arithmetic Op>, of Expression, Strings, of Literal Modules: Members: format() Flags:

For the purposes of Knowledge Extraction and Organization, the output of an AST [8] does little to assist in the creation of a structured, adaptive learning environment. The ability of k-BST to output a binary vector as well as match the binary vector to textual learning units provides users with the ability to process a parsed example in order to generate similar examples. Having the parsed learning units outputted as a simple to read text string provides novice programmers with a simple manner to understand programming concepts that are contained within a given code segment.

Furthermore, k-BST is able to produce extractions significantly quicker than an AST [8] would. Figure 2 shows that given examples in the quantities of 1, 50, 100, 500, 1000, 5000, and 20000, the k-BST algorithm was able to complete the extraction of learning units in significantly less time. This significant reduction in time can be attributed to the manner in which k-BST uses the parsed k-BST which allows input examples to be simply compared against an existing tree which contains the desired learning units, as opposed to a traditional AST which performs a complete extraction of input code, and then afterwards seeks to extract the relevant learning units. In order to have such a large set of examples to test k-BST against, a set of 1000 examples was first compiled which contained a variety of learning units. From there, the original 1000 examples was then duplicated 5 and 20 times in order to have an example set of 5000 and 20000. From a simple perspective, k-BST and AST can be compared to knowing exactly what you want and looking for it, as opposed to gathering all the data, and then extracting the concepts that you are looking for.

V. Evaluation of Clustering

To evaluate the clustering algorithm, we applied it to the benchmark zoo database (available at uci.kdd) which contains 101 instances of various animals. Each animal by a vector that contains 17 binary attributes (has hair, feathers, eggs) and 1 categorical attribute (number of legs). We transformed the non-binary attribute into 6 different binary attributes to create an binary vector representation of each animal. The dataset already clustered the animals into one of seven predefined classes. When we ran our clustering algorithm on the zoo dataset, we also created 7 clusters. Then we created a confusion matrix which compares the actual class of the animal

Number of Examples	AST	k-BST
1	0.2	0
50	0.42	0.4
100	0.74	0.07
500	3.7	0.24
1000	8.03	0.47
5000	37.34	1.51
20000	162.1	4.83

Table I

System Runtimes (in seconds) for a varied number of input examples. Times shown are the average of 5 runs on the same code sample.

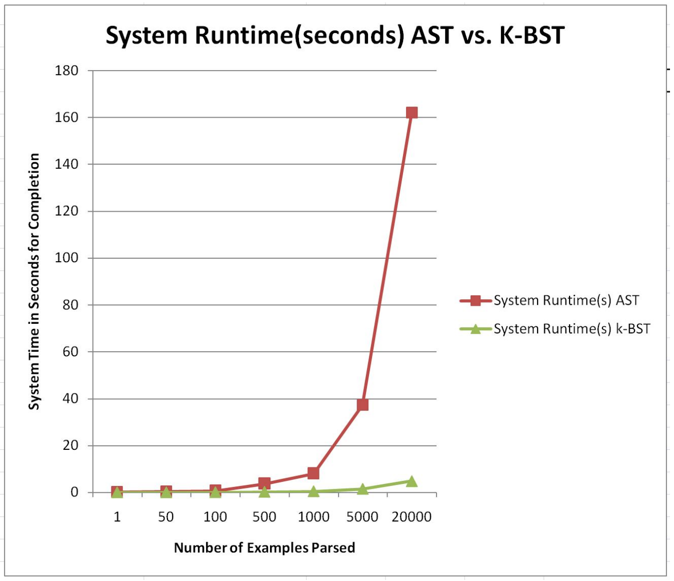

Figure 2. System Runtime: AST vs. k-BST

	1	2	3	4	5	6	7
1	32	0	7	2	0	0	0
2	0	20	0	0	0	0	0
3	2	0	0	3	0	0	0
4	0	0	0	13	0	0	0
5	0	0	1	3	0	0	0
6	0	0	0	0	0	8	0
7	1	0	0	0	3	0	6

Table II

The rows indicate the actual class of an animal and the column indicates the predicted class.

Number of Clusters	Dunn's Index	Number of Clusters	Dunn's Index
2	0.6	6	0.72
3	0.73	7	0.78
4	0.89	8	0.26
5	0.68	9	0.16

Table III

Dunn's indices for various number of clusters.

Example	learning units Found
DNASequence.py	nested statement, Function Definition, Simple If, For Loop
factorial.py	simplePrint, Simple Else If, nested statement, Simple Assignment, Else, Simple If, For Loop with Range
largestOfThree.py	simplePrint, Simple Else If, Simple Assignment, Else, Simple If

Table IV
SAMPLE OF THREE EXAMPLES AND THEIR LEARNING UNITS FOUND IN ONE CLUSTER

to the class assigned by the algorithm. Looking at table II, we can see that 79 of the animals were assigned to the correct class leading to a classification accuracy of 78%.

Another way to evaluate the clustering algorithm is to use the Dunn's index, which is a metric for evaluating clustering algorithms. The higher the Dunn's index, the better formed the clusters as a high Dunn's index implies high intra-cluster similarity and low inter-cluster similarity [2]. The Dunn's index for m clusters is defined as

$$DI_m = \frac{\min_{1 \leq i \leq j \leq m} \delta(C_i, C_j)}{\max_{1 \leq k \leq m} \Delta_k}$$

where $\delta(C_i, C_j)$ is the distance between two clusters and Δ_k is the intra-cluster distance. When we computed the Dunn's index on the zoo dataset for a various number of clusters, we found that the Dunn's index is highest when there are 4 clusters as opposed to 7 (table III). This can be attributed to the fact the random initialization of initial centroids means that certain centroids may be too close together, leading to some clusters being very large and other clusters being very small.

When the clustering algorithm was applied to the set of worked out python programming examples, 7 clusters were created and the Dunn's index was calculated to be 0.6857. This indicates each cluster contains a set of examples that are fairly similar with some overlapping learning units. This achieves the goal of creating cluster similar enough to show how each learning unit is used in programming as well as how different learning units are related to one another. Table IV shows a sample of three examples that were found in the same cluster. They each contain similar learning units related to conditional statements along with other learning units. By viewing these examples, a user could see a variety of ways conditional statements are used as well as how conditional statement related to for loop or nested statements.

VI. LIMITATIONS

Being the first iteration of k-BST, several limitations have been noted. Firstly, k-BST assumes that the input code is correct and well formed. Therefore, there exists the possibility of detecting the presence of a learning unit even though the code may be syntactically incorrect. Secondly, k-BST requires that an expert defines the relevant learning units, and the associated keywords to each learning unit prior to extraction taking place. In its current form, k-BST is also limited to basic concepts and does not go beyond the scope of the CS1 curriculum. Lastly, due to the current implementation of k-BST's searching technique, a maximum of one learning unit can be detected per line of code, however there exists the possibility to have multiple learning units per line of code.

VII. CONCLUSIONS

In conclusion, when doing knowledge extraction with beginner level python source code an AST often provides more information than is required to accurate extract learning units. The proposed k-BST method simplifies the knowledge extraction process by only requiring the domain expert to provide keywords and learning units rather than an entire grammar.The simplified nature of the k-BST allows it to provide sufficient results for our needs while extracting learning units more efficiently than an AST. The results of the k-BST than can then be used for knowledge organization using k-means clustering where it is possible to create clusters of related worked out examples.

VIII. FUTURE WORK

In the future, we will look at making modifications to the k-BST method to modify the memory usage in order to minimize the space-time tradeoff. The proof of concept shows a k-BST is able to accurately extract the same concepts as an AST, and do so in a shorter time, so the next step is to modify the existing implementation to be more conscious of memory usage, and perhaps implement in a more efficient language such as C in order to have direct control over memory usage. Furthermore, the initial cluster evaluation shows that is possible to create clusters of related examples. The next step is to introduce the clusters to students and measure the effectiveness of the clusters in teaching students about applying the various coding concepts contained within the learning units.

References

[1] Surafel Lemma Abebe and Paolo Tonella. Extraction of domain concepts from the source code. *Science of Computer Programming*, 98:680–706, 2015.

[2] Ritu Chaturvedi. Task-based example miner for intelligent tutoring systems. *Electronic Theses and Dissertations*, 2016.

[3] K Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S Sudarshan. Extracting equivalent sql from imperative code in database applications. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1781–1796. ACM, 2016.

[4] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015.

[5] Gerard J Holzmann. Cobra: a light-weight tool for static and dynamic program analysis. *Innovations in Systems and Software Engineering*, 13(1):35–49, 2017.

[6] Roya Hosseini and Peter Brusilovsky. Javaparser: A fine-grain concept indexing tool for java problems. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, pages 60–63. University of Pittsburgh, 2013.

[7] Lannan Luo and Qiang Zeng. Solminer: mining distinct solutions in programs. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 481–490. ACM, 2016.

[8] Andrew Luxton-Reilly and Andrew Petersen. The compound nature of novice programming assessments. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, pages 26–35. ACM, 2017.

[9] Maryam and Andrei Popescu-Belis. Keyword extraction and clustering for document recommendation in conversations. *IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING*, 23:746–759, 2015.

[10] TOMMASO DI NOIA XAVIER SERRA EUGENIO DI SCIASCIO SERGIO ORAMAS, VITO CLAUDIO OSTUNI. Sound and music recommendation with knowledge graphs. *ACM Transactions on Intelligent Systems and Technology*, 8:21:0–21:21, 2016.

[11] Jörg Unbehauen, Sebastian Hellmann, Sören Auer, and Claus Stadler. Knowledge extraction from structured sources. *SeCO Book*, 7538:34–52, 2012.

[12] Boban Vesin, Mirjana Ivanović, Aleksandra Klašnja-Milićević, and Zoran Budimac. Ontology-based architecture with recommendation strategy in java tutoring system. *Computer Science and Information Systems*, 10(1):237–261, 2013.

[13] Annie TT Ying and Martin P Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 655–658. ACM, 2013.

[14] Xiaoyan Zhu, E James Whitehead, Caitlin Sadowski, and Qinbao Song. An analysis of programming language statement frequency in c, c++, and java source code. *Software: Practice and Experience*, 45(11):1479–1495, 2015.