Recognition of Java Source Code by Graph Matching Algorithm

Tomáš Bublík
Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague
Prague, Czech Republic
e-mail: tomas.bublik@gmail.com

Miroslav Virius
Faculty of Nuclear Sciences and Physical Engineering
Czech Technical University in Prague
Prague, Czech Republic
e-mail: miroslav.virius@fjfi.cvut.cz

***Abstract*— This paper describes an option how to detect a desired Java code snippet in a large number of Java source files. The scripting language Scripthon is used to describe the desired section. Next, from this piece, an abstract tree is created, and it is compared to the other trees which are created from the Java source codes. The Java Compiler API is used to obtain the trees from the Java source codes. The final result of tree matching process is presented to a user.**

Keywords-abstract syntax tree; Java; Scripthon; trees matching; compiler API

I. Introduction

Searching source code is an easy task. Nevertheless, this applies only in the case of a simple text or simple structure names. This feature is supported in most of the current Java development environments. Some integrated development environments (hereinafter IDE) [11] [12] support an advanced searching with the regular expressions. But, what if a user wants to know, whether a program contains the singleton? Or, whether the specific method (with three concrete parameters) is somewhere in a program? It is very difficult to find such information; however, using the mathematical and programming knowledge, it is possible.

When using the Scripthon language [1], these special structures can be described very precisely. On the other hand, by using the Java Compiler application programming interface (hereinafter API), the abstract syntax trees (hereinafter AST) can be obtained and compared with Scripthon output. This paper is about using these trees for searching the desired code snippet. This task is similar to the graph matching and isomorphic sub-graphs finding in a large set of trees. An additional problem arises in the applications where an input graph needs to be matched not only to another graph, but to an entire database of graphs under a given matching paradigm. Therefore, some complexity reducing algorithms are proposed in this paper.

There are several reasons to consider graphs to be very advantageous tool for the representation of source code of some language. One reason is that there is no unnecessary material like spaces, comments, etc. Another reason is that there are many well described mathematical algorithms to work with graphs. Some of the algorithms are known for decades. Representing the code as a graph has also the disadvantage: it has large demands on a computer power and memory; especially for larger programs.

The first section compares existing similar solutions with this one. Several tools with the similar function are mentioned there. The next section introduces necessary graph theory concepts. The definitions of a graph, a sub-graph and a graph isomorphism are given. The Scripthon language is introduced briefly in fourth chapter. Because the language has been described already in another paper [1], only the important properties are mentioned here. The next two sections are about graphs generation, optimizations, and the comparison of graphs generated by the Compiler API. An algorithm for trees matching can be found in Section 6. Finally, several results are presented in the conclusion.

II. Comparation with Similar Solutions

There are many approaches to the code search area. These approaches can be classified as textual, lexical, tree-based, metrics-based and graph-based. This distribution depends on how the source code is expressed. More on this topic can be found in [6]. Scripthon belongs to the tree-based solutions.

A number of similar solutions for all the mentioned tasks have been proposed in [6]; however, Scripthon is quite different tool. This tool is not supposed to detect the clones automatically. However, it is possible with the assistance of the user,. Our previous work dealt with automatic detection and removal of clones in Java source code [2]. Finally, with respect to other solutions and a complexity of this topic, we decided to try another way. In addition, we considered that the detection and removal of the so-called “non-ideal” clones is very difficult without some additional information from a user. (The “non-ideal” clones are repeated pieces of source code that are not exactly the same, but execute similar operations.) The Scripthon is primarily designed to search known patterns in source code. It means that the user must approximately know how the clone looks like. Then, he or she creates a script based on his or her ideas which finds the desired patterns.

A similar solution is described in [7]. Refactoring NG is an interesting tool which allows defining a refactoring operation programmatically; however, it cannot be used for defining the searching patterns.

In addition, Scripthon is not aimed to detect design patters. With Scripthon, it is possible to find a simple design pattern within one class (for example, the above mentioned Singleton), but it is not its main purpose. It is not possible to find a design pattern composed of multiple classes.

Unlike regular expressions, Scripthon offers an interesting alternative to search a shape or properties of a given Java source code.

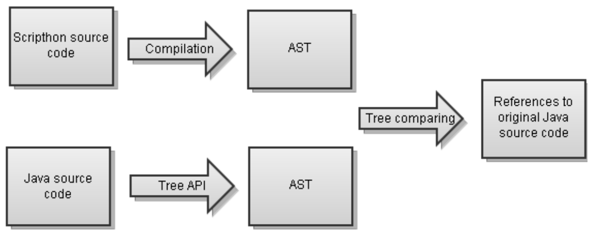

Figure 1. Complete process

Figure 1 shows the whole process of searching Java source snippets using Scripthon. The tool runs in two threads. The first one is a Scripthon compiler. Its output is a modified AST. The second thread is aimed to create a Java AST. Both trees are compared with the matching algorithm. A result of the process is the references to a given Java source code. Typically, it is the name of a Java class and the line number where the match occurred.

III. Basic Graph Theory Concepts

A graph is a four-tuple $g = (V, E, \alpha, \beta)$, where V denotes a finite set of nodes, $E \subseteq V \times V$ is a finite set of edges, $\alpha : V \longrightarrow L_V$ is a node labeling function, and $\beta : E \longrightarrow L_E$ is an edge labeling function. L_V and L_E are finite of infinite sets of node and edge labels, respectively.

All the graphs in this work are considered to be directed. A subgraph $g_s = (V_S, E_S, \alpha_S, \beta_S)$ of a graph g is a subset of its nodes and edges, such that $V_S \subseteq V, E_S \subseteq E \cap (V_S \times V_S)$ Two graphs g and g' are isomorphic to each other if there exists a bijective mapping u from the nodes of g to the nodes of g, such that the structure of the edges as well as all node and edge labels are preserved under u . Similarly, an isomorphism between a graph g and a subgraph g' of a graph g' is called subgraph-isomorphism form g to g' .

A tree is a connected and undirected graph with no simple circuits. Since a tree cannot have a circuit, a tree cannot contain multiple edges or loops. Therefore, any tree must be a simple graph. An undirected graph is a tree if and only if there is a unique simple path between any two of its vertices.

The graph matching problem is actually the same as the problem of finding the isomorphism between the graphs. Moreover, matching the parts of a graph with a pattern is the same challenge as the finding the isomorphic subgraph.

IV. Scriphon Description

The Scripthon language is widely described in [1]. The following text will present only the summary of important properties of this language. Scripthon is a simple-to-learn language which is able to describe a Java source code structure. Because of its simple syntax, it is very easy to learn. The syntax of the Scripthon language is similar to the syntax of Java, and it is very intuitive. Basically, the keywords represent the structures in Java language. Thus, a Scripthon program is built only with these words and its properties. Each keyword has a special set of its own properties. There are three sets defining the usable keyword, its properties and the properties values. For example, this is the set of structural keywords (Str):

$$\mathrm{Str} = \{\mathrm{Meth}, \mathrm{Init}, \mathrm{Block}, \mathrm{Class}, \dots\}$$

For a Class() keyword, the set of parameters (SAtr) looks like:

$$\mathrm{SAtr} = \{\mathrm{Name}, \mathrm{Rest}\}$$

For these parameters, the set of available values (AVal) is:

$$\mathrm{AVal} = \{\mathrm{public}, \mathrm{static}, \mathrm{private}, \mathrm{interface}\}$$

For example, a class is represented by a Class() keyword. The parameters of this keyword can be in the parentheses, however, if the brackets include no parameters, each class is a candidate for searching and each class of a given program corresponds to this structure. For example, the following command:

```
Class(Name = "Main"; Rest = public)
```

means that the wanted structure is a public class with the name Main. The options of the parameters are specified in the Scripthon documentation. The structure nesting it is denoted only by the line separators.

```
Meth(Rest = private; ParamsNum = 2)
  Block()
    Init(Type = int; Value = ""; Name = "sum")
```

This example means that the searched structure is a private method with two parameters. Inside the method is a block with two statements. The first statement is a variable named sum of type int. The second statement is a return statement with a parameter of the previously specified variable.

The big advance of the Scripthon language is the ability to describe the elements with a variable depth of details. This means that the searched structures can be described in a detail or very loosely. For example, this is a very detailed description:

```
Class(Name = "TestDecompile"; Rest = public)
  Meth(Name = "main"; Ret = void; Rest = public)
    Init(Name = "toPrintValue"; Type = String)
      MethCall(Name = "System.out.println")
```

The same script without details follows:

```
Class()
  Meth()
    Init()
      MethCall()
```

Therefore, a searched subject can be found on the base of a very inaccurate description. The results can be obtained with the iterative refinement of the input conditions. In the end, the user can get better results.

Furthermore, Scriphon contains a special keyword Any(). It is not a structural keyword, but it is information for the matching algorithm to act as anything. When used, it means that a searching structure could be anything (even with the sub-trees) or nothing. With respect to the previous example, the desired structure can be described even with this script:

```
Class()
  Meth()
```

However, because of the generality of this script, the number of results found will be very high. (Actually, any class corresponds to this script)

The level of detail which can be described by the current version of Scripthon is up to – but not including – the expression. In addition, Scripthon can describe a lot of Java structures, but it cannot describe the individual elements of an expression statement. For example, while describing the if statement, it is possible to address the inner block, or the else block with inner statements; however, the if-expression in the parentheses cannot be described. Moreover, Scripthon is not able to describe the mathematical operations. If a variable i is declared such as:

```
int i = a + b;
```

The most accurate Scripthon statement to find is:

```
Init(Name = "i"; Type = int)
```

In the current version of Scripthon, nothing more can be described. On the other hand, this language is designed to be extensible. The main program consists of several modules appropriate to corresponding stages of searching process.

Current version of the language cannot describe all the Java language structures. For example, annotations, generics, diamond operators, and many others are omitted, but they can be easily added in future versions. It would be necessary to introduce new structural word, define its properties, and define the rules to the searching algorithm. There is no need to change syntax, or even the compiler.

V. Graph Generation with Java Compiler API

The Java Compiler API is used to get a graph for the searching algorithm. This API is free, and it is included in

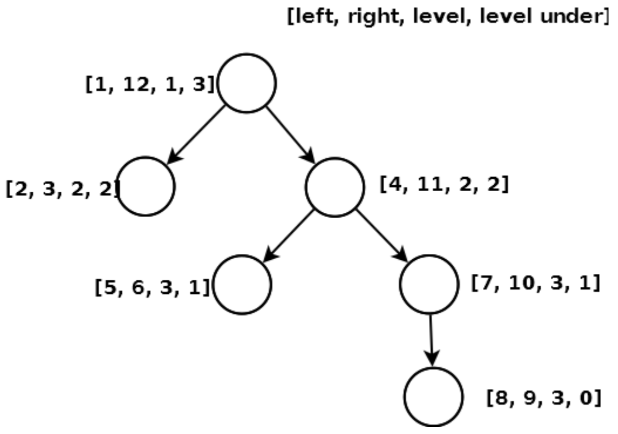

Figure 2. Tree with optimizations

the Java distribution. Basically, the Java Compiler API serves to the advanced control of a compilation process. This API uses the AST and the visitor design pattern. Unfortunately, this design pattern is not suitable for searching purposes. This is because the Scripthon language cannot to describe such a quantity of structures, and also because the searching algorithm is not suitable for the implementation with the visitor design pattern. Therefore, the more advanced graph is created from a Java AST. This graph is based on the AST, but it has a several benefits.

The first benefit is the replacement of the visitor pattern with the classic approach. The second one is that some additional information is included, which significantly facilitates the searching.

While browsing the Java source code, the tree with the nodes enhanced by four numbers is created. These numbers are the natural numbers named left, right, level and level under. The first and the second number (left, right) denote the order index of a node in the tree preorder traversal. Therefore, an ancestor's left index is always smaller than its children left index, while the right index is always bigger than any children's right index. The level number denotes the level in a tree hierarchy of vertices, and the level under number denotes a number of levels under the current node (compare with the method described in [4]).

Suppose that x and y are two nodes from a tree; the following rules are valid for these values.

- The y node is an ancestor of x and x is a descendant of y if $\mathrm{y.left} < x.\mathrm{left} < y.right$
- The y node is a parent of x and x is a child of y if 1) $\mathrm{y.left} < x.\mathrm{left} < y.right$ and 2) $\mathrm{y.level} = \mathrm{x.level} - 1$
- The node x has $((\mathrm{x.left} - \mathrm{x.right}) - 1)$ sub-nodes.

All these data are acquired during a single pass through the tree. Obtaining this information is not a time consuming operation, because it is made during the tree production process. On the other hand, the number of comparisons can be significantly reduced with these numbers. Moreover, while comparing the trees, it is very easy to detect:

- How many elements have a given structure
- Whether a node is a leaf
- How many sub-statements are included in a given structure

The comparison of two trees is much more time consuming without this information. In summary, this information is used in cases where the shape of the given structures and its coupling is considered more than its properties.

A line reference to source code is important information which is also added to the tree as a metadata. Therefore, it is easy to link the results with the original source position and show it to the user. There are some more elements in a node metadata. For example, some of the other metadata information is a filename of the source file.

Because the number of the comparisons is a key indicator for the algorithm speed, it is necessary to keep the number of nodes as small as possible. Therefore, only the supported structures and their properties are considered while creating a tree from source code. Thus, the same Scripthon definition set is used during the tree creation process. Other elements are omitted.

VI. GRAPH MATCHING

The simple and many times described backtracking algorithm is used for the graph matching. Basically, it is the problem of finding an isomorphic tree to the given tree from a large database of trees. Comparing to the common tree matching, there are two differences. The first one is that the node properties need to be considered during the process. The second difference is that not every Scripthon node corresponds exactly to one Java structure node. For example, the already mentioned keyword Any() could correspond to more nodes.

The source trees are created from the corresponding classes. The classes and the trees are mapped one-by-one. Each tree corresponds to exactly one class. In the first step, the algorithm checks whether the shape of the structure match, and then the properties are compared. This is because the properties matching is much more time consuming operation than shape detection. Many structures are eliminated very quickly from the process in the case that the shape does not fit.

If the shape of the structure corresponds to the required shape, the structure parameters are compared. All the parameters of a given node must be met. The node properties are provided by the Java compiler.

```
1.for (Class c) //iterate over all classes from given sources
2.   match = true
3.   for (Statement s)
4.      match = compare(s, c.parentNode)
5.      if (match == false)
6.         break
7.      if (match)
8.         add it to the list of founded structures
9.
10.boolean compare(Statement s, Node n)
11.   match = true
12.   if (s.properties match n.properties)
13.      for (s.children, n.children)
14.         compare(s.child, n.child)
15.      if (match == false)
16.         return false
17.      else
18.         return true
19.   else
20.      return false
21.   return true
```

Figure 3. Simplified tree matching algorithm.

Figure 3 shows the simplified matching algorithm. It is written in Java pseudo-code. The algorithm skeleton is similar to the algorithm described in [8]. The main difference is that in our solutions are compared not two Java trees, but a Java tree and a Scriphon tree. The whole program iterates over all given Java classes (line 1in the figure 3). Instead of finding a corresponding sub-tree, the algorithm tries to exclude quickly a mismatching part. It can be seen from line 2.

At the beginning, it is assumed that the given source matches. The rest of the algorithm iterates over Scripthon statements (line 3) and tries to find a match between a statement and a node of a Java AST (line 4). A matching method (line 10) is called recursively as the sub-nodes are traversed. If a result of this method is false, the loop over Scripthon statements is interrupted (line 6), because even the first statement does not correspond to anything of a Java class. The result of the "compare" method is true (line 21) or false (line 20). A statement and a node are equal if all their corresponding properties are equal (line 12) and all the children are equal (line 13). Therefore, all children are iterated and compared recursively (line 14). If a match is found, this method returns true (line 18). Otherwise it returns false (line 16). If true, the result is added to the result list.

Many aspects are considered during properties matching process. Not only keywords and Java nodes properties are considered. According to the previous section, it is possible to exclude quickly the mismatched parts, because some additional data are known about a shape of the sub-tree.

The typical size of a class graph depends on the source size and on the number of supported structures. About 80 nodes of the graph are created from a Java class with length about 200 lines nodes in the current version of Scripthon. In future versions, when more structures will be supported, may the number of the nodes significantly increase.

Unfortunately, because all the Java classes with all their nodes must be compared with all the Scripthon statements, the number of complexity rapidly grows. According to [3], the sub-graph isomorphism problem has $O(N^3)$ complexity in worst case. Since the number of occurrences can be more than one, each class must be browsed more than once. Each class needs to be traversed until the number of results is 0. According to [9, 10] the graph isomorphism problem is polynomial. Therefore, even in this case, the complexity of our algorithm remains polynomial. On the other hand, with the above outlined optimizations, the number of node comparisons is significantly decreased. More on the similar graph matching techniques can be found in [5].

VII. Measurements Results

The used algorithm modifications substantially reduced the time needed to find the requested Java structures. Moreover, also the time of the tree generation procedure has been shortened. According to the measurements, the meta-information counting does not significantly affect the time of a graph creation.

The searching with optimization is much faster. The following tables show the measured time results. The small program means a program consisting of approximately 20 to 30 classes, while the larger program is a program with approximately 100 to 150 classes. There are also the results before and after the described optimizations.

TABLE I. Graph creation times

Program type	Time
Small program (no optimizations)	412 ms
Larger program (no optimizations)	4 423 ms
Small program (optimized)	132 ms
Larger program (optimized)	337 ms

TABLE II. Searching times

Program type	Time
Small program (no optimizations)	2 345 ms
Larger program (no optimizations)	11 236 ms
Small program (optimized)	753 ms
Larger program (optimized)	1 986 ms

TABLE III. Total times

Program type	Time
Small program (no optimizations)	2 757 ms
Larger program (no optimizations)	15 659 ms
Small program (optimized)	886 ms
Larger program (optimized)	2 323 ms

The measurements were performed on the quite common computer. The computer configuration was: 4GB of memory, the Intel Core I5 processor with a frequency of 2.4 GHz and Windows 7 as an operating system. The individual results represent the averages of several consecutive measurements. The first column indicates the time needed to the AST generation, while the second one represents the time required to find a piece of the sample code described by the Scripthon language. The last column is the sum of both times. The lines represent the sizes of programs on which the measurements were performed.

As it is shown in the tables, in case of the small program, the graph assembling is not significantly different. On contrary, better results can be obtained in the case of larger programs. Probably, this is because some time is needed for the overhead services related to the starting and initializing the own search.

VIII. Conclusion

With the described solution, we proved that the proposed concept of searching is possible. Moreover, it is also very effective. With used optimizations, the algorithm significantly improved performance of a whole process. Next, the Scripthon project is designed as a modular system. Therefore, as will the functionality requirements grow, it is not difficult to add more supported Java structures. Even the language itself could be enhanced by new syntax elements very easily. There are many possibilities of how Scripthon could be used. One of the planned usage areas is a student's work controlling task. With Scripthon, it is easy to detect whether a student's work contains prescribed programming structures.

The Scripthon language improved. The Scripthon compiler is available as a command line tool now. We suppose to develop the Scripthon plug-in for some popular integrated development environments in the future.

Acknowledgment

This work is supported by the SGS 11/167 grant of the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1] T. Bublík and M. Virius.: “New language for searching Java code snippets,” in: ITAT 2012. Proc. of the 12th national conference ITAT. diar, Sep 17 – 21 2012. Pavol Jozef Safrik University in Kosice, pp. 35 – 40.

[2] T. Bublík and M. Virius: “Automatic detecting and removing clones in Java source code,” in: Software Development 2011. Proc. of the 37th national conference Software Development. Ostrava, May 25 – 27 2011. Ostrava: Technical University of Ostrava 2011. ISBN 978-80-248-2425-3, pp. 10 – 18.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. “Clone Detection Using Abstract Syntax Trees,” in Proceedings of the International Conference on Software Maintenance (ICSM '98). IEEE Computer Society, Washington, DC, USA, pp. 368-377.

[4] J. T. Yao and M. Zhang. 2004. “A Fast Tree Pattern Matching Algorithm for XML Query,” in Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI '04). IEEE Computer Society, Washington, DC, USA, pp. 235-241.

[5] H. Bunke, Ch. Irniger, and M. Neuhaus. 2005. “Graph matching – challenges and potential solutions,” in Proceedings of the 13th international conference on Image Analysis and Processing (ICIAP'05), Fabio Roli and Sergio Vitulano (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 1-10.

[6] Ch. K. Roy, J. R. Cordy, and R. Koschke. 2009. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” Sci. Comput. Program. 74, 7 (May 2009), pp. 470-495.

[7] Z. Troníček. 2012. “RefactoringNG: a flexible Java refactoring tool,” in Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12). ACM, New York, NY, USA, pp. 1165-1170.

[8] W. Yang. 1991. “Identifying syntactic differences between two programs,” Softw. Pract. Exper. 21, 7 (June 1991), pp. 739-755.

[9] J. Köbler and J. Torán. 2002. “The Complexity of Graph Isomorphism for Colored Graphs with Color Classes of Size 2 and 3,” In Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS '02), Helmut Alt and Afonso Ferreira (Eds.). Springer-Verlag, London, UK, UK, pp. 121-132.

[10] I. S. Filotti and J. N. Mayer. 1980. “A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus,” In Proceedings of the twelfth annual ACM symposium on Theory of computing (STOC '80). ACM, New York, NY, USA, pp. 236-243.

[11] T. Boudreau, J. Glick, and V. Spurlin, “NetBeans: The Definitive Guide,” Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2002.

[12] S. Holzner, “Eclipse,” O'Reilly Media, April 2004.