Mining Patterns in Source Code using Tree Mining Algorithms

Hoang Son Pham[1], Siegfried Nijssen[1], Kim Mens[1],
Dario Di Nucci[2], Tim Molderez[2], Coen De Roover[2],
Johan Fabry[3], and Vadim Zaytsev[3]

[1] ICTEAM, UCLouvain, Belgium
[2] Software Languages Lab, Vrije Universiteit Brussel, Belgium
[3] Raincode Labs, Belgium

Abstract. Discovering regularities in source code is of great interest to software engineers, both in academia and in industry, as regularities can provide useful information to help in a variety of tasks such as code comprehension, code refactoring, and fault localisation. However, traditional pattern mining algorithms often find too many patterns of little use and hence are not suitable for discovering useful regularities. In this paper we propose FREQTALS, a new algorithm for mining patterns in source code based on the FREQT tree mining algorithm. First, we introduce several constraints that effectively enable us to find more useful patterns; then, we show how to efficiently include them in FREQT. To illustrate the usefulness of the constraints we carried out a case study in collaboration with software engineers, where we identified a number of interesting patterns in a repository of Java code.

Keywords: Pattern Mining, Frequent Tree Mining, Source Code Regularities

1 Introduction

During software development, many design and coding conventions get encoded in program source code, either explicitly or implicitly, through regularities such as API usage protocols, design patterns, coding idioms or conventions. Being able to discover such source code regularities in software systems is of great interest to software engineers, to help understanding, analysing, transforming, improving, maintaining or evolving a particular system, or to improve best practices for the development of new systems.

A data type of particular interest in the context of source code is the Abstract Syntax Tree (AST). ASTs capture not only the textual content, but also the structure of the code. However, to analyse these trees, algorithms capable of operating on tree structures are needed. *Frequent* tree mining algorithms [1, 2] support this task. However, they typically find large numbers of patterns. This makes their output often useless in practice to software engineers. Several solutions to this problem have been proposed, ranging from constraint-based

pattern mining approaches and condensed representations, to statistically motivated pattern set mining approaches [3]. Among these, constraint-based data mining approaches are of particular interest, as they allow developers to specify easy to interpret constraints on the patterns to include in the output of the algorithm, and are guaranteed to find all patterns satisfying the constraints, contrary to sampling based approaches [4].

However, applying constraint-based data mining and condensed representations on the ASTs of software repositories is not straightforward. While frequent pattern mining has been studied extensively in the frequent tree mining literature, *constraint-based tree mining algorithms* did not receive a similar attention.

In this paper, we therefore propose a novel constraint-based tree mining algorithm, specifically designed for the analysis of software repositories. It combines two ideas: (i) *maximal frequent subtree mining* to ensure that a condensed representation of only large patterns is found, (ii) *constraint-based data mining*, in which additional constraints are imposed on the patterns to be found. Our approach is based on the addition of a number of novel constraints to the FREQT algorithm [5], combined with a new approach to find maximal subtrees.

In collaboration with software engineers we analysed in detail the quality of the patterns found. The results show (i) a significant reduction of the execution time and number of discovered patterns with respect to the original FREQT algorithm, (ii) that many of the discovered patterns highlight relevant code regularities, (iii) that some of the patterns found are significantly larger than the simpler coding idioms found in earlier studies [4].

The paper is organised as follows. Section 2 introduces frequent subtree mining and FREQT. Section 3 presents the key ideas of our solution, which is implemented by the FREQTALS algorithm described in Section 4. In Section 5 we conduct a case study to validate FREQTALS. Section 6 overviews related literature on pattern mining of semi-structured data and pattern mining applied to software. Section 7 concludes the paper.

2 Background

2.1 Frequent Subtree Mining

Abstract Syntax Trees are labelled, ordered, and rooted trees; they are produced by programming language parsers. We adopt a previously studied definition [1, 5] for ordered trees $T = (V, E, \lambda, \Sigma)$; $V = \{1, 2, \ldots, n\}$ is the set of node identifiers; $E \subseteq V \times V$ is the set of edges; $\lambda : V \mapsto \Sigma$ is a function that associates labels to nodes of V; Σ is the set of allowed labels. We assume the nodes are identified using contiguous integers listed in the order of a depth-first, left-to-right traversal of the tree; node n is called the *rightmost node* of the tree. The shortest path from node 1 to node n is its *rightmost path*.

Given two trees $T_1 = (V_1, E_1, \lambda_1, \Sigma)$ and $T_2 = (V_2, E_2, \lambda_2, \Sigma)$, T_2 is an induced subtree of T_1 ($T_1 \succeq T_2$) if there is an injective function $f : V_2 \mapsto V_1$ such that: (1) edges are preserved: for all $(v, v') \in E_2$: $(f(v), f(v')) \in E_1$; (2) labels

are preserved: for all $v \in V_2$: $\lambda_2(v) = \lambda_1(f(v))$; (3) order is preserved: if $v_1 < v_2$ for a pair of nodes in V_2, then $f(v_1) < f(v_2)$.

The *support* of a pattern tree is the number of trees in a database in which the pattern occurs. *Frequent subtree mining* is the problem of finding all patterns of which the support is higher than a given *minimum support* threshold. These patterns satisfy the *minimum support constraint*, which we will refer to as constraint **C0** in this paper.

The number of frequent subtree patterns can become large, in particular for small minimum support thresholds. To deal with this issue, one solution is to mine *condensed representations* [1, 2] and *maximal frequent subtrees*. Let $\mathcal{T}_c$ denote the set of all patterns that satisfy **C0**. We can define the problem of finding maximal frequent subtrees as the problem of finding all patterns not dominated by other patterns:

$$\max(\mathcal{T}_c) = \{T \in \mathcal{T}_c \mid \nexists T' \in \mathcal{T}_c : T' \succ T\}.$$

2.2 FREQT

FREQT was designed to mine frequent patterns from labelled ordered trees [5]. It searches for patterns using a depth-first search, where it grows patterns using *rightmost path extension*. The idea is to add new nodes only to the right of the rightmost path of a pattern. Hence, a pattern is created by adding its nodes in the order of a depth-first, left-to-right traversal.

Algorithm 1: FREQT

```
1 FP = ∅
2 C ⟵ findLabels()
3 prune(C)
4 for each c ∈ C do
5 |   add (FP, c)
6 |   expand(c)
7 output(FP)
```

Algorithm 2: `expand` procedure

```
1 function expand(f):
2 |   C ⟵ findCandidates(f)
3 |   prune (C)
4 |   for each c ∈ C do
5 |   |   add (FP, c)
6 |   |   expand(c)
```

The structure of the depth-first search algorithm is described in Algorithms 1 and 2. By default FREQT only uses minimum support (**C0**) and *maximum size* (referred to as **C1**) constraints in the `prune` function. Anti-monotonic constraints are used to effectively reduce the size of the search space. A constraint is anti-monotonic iff for all pairs of patterns with $T_1 \succeq T_2$, if T_2 does not satisfy the constraint, then T_1 does not satisfy the constraint either. In addition, to avoid undesirable patterns from being added to the set $\mathcal{FP}$, a *minimum size* constraint (referred to as **C2**) is used in the `add` function.

Algorithms for finding only maximal frequent subtrees exist [1, 2]; they usually reduce the search space by checking *all* extensions for *all* occurrences of a pattern. This is problematic for trees with a large fanout, such as ASTs.

3 Maximal Constraint-based Frequent Subtree Mining

In this section, we will show how the AST representation allow us to impose additional meaningful constraints on patterns. There are two important ideas underlying these constraints.

First, given that programming languages are typically well-structured, parts of the data have a very predictable structure. Patterns that either reflect only this predictable structure, or that only include part of it, are not useful. For instance, by definition of the Java programming language, a node with label `InfixExpression` always has the same three children, `leftOperand`, `operator` and `rightOperand`. Clearly, a pattern composed of these four nodes is a frequent pattern but it is not interesting, as it is a consequence of the language and not the particular source code that is being mined. Patterns including the `InfixExpression` label, but not its three children, are not meaningful either, as by definition, these child nodes must be present.

Second, small fragments of ASTs are hard to interpret. In practice, we found that many software engineers find easier to interpret a code fragment if it is sufficiently large, allowing to put a pattern in its context. In terms of the patterns that we find, this means that our patterns need to satisfy minimum size criteria.

To find a small set of patterns which are sufficiently large and correctly reflect interesting program structures we propose the following constraints.

Constraints on Labels. To limit the number of patterns considered, the use of labels is a straightforward solution. The key benefit of label-based constraints is that they are easy to configure by software engineers. We consider the following constraints:

C3. Limit the set of labels allowed to occur in the root of patterns;
C4. Provide labels forbidden from occurring in the pattern;
C5. Limit the number of siblings in a pattern that can have the same label.

Constraints on Leafs. It is desirable that patterns not only represent the structure of the language, but also provide program-specific information. As such specific information can be found in the leaf nodes of ASTs in the database, we add this constraint:

C6. All leaf nodes in a pattern must have a label that is included in Σ_{leaf}, where Σ_{leaf} is the set of labels that occur in the leafs of the trees in the database.

Constraint on Obligatory Children. Given a node, some of its children can be mandatory because they reflect a specific programming language construct (*e.g.,* the `InfixExpression` shown before). To avoid unnecessarily small patterns, we first need to characterise which labels are *structural*. We consider a label to be structural iff:

- in each of its occurrences, no two children have the same label;
- for all pairs of occurrences of the label, the order of the common child labels is the same.

For every label $a \in \Sigma$, we define its *obligatory child labels* $g(a)$ as the set of child labels common to all its occurrences. We added the follow constraint on obligatory child labels:

C7. Let L be the structural labels in $\mathcal{D}$. For all nodes with a label $a \in L$, we require that its set of children includes nodes with all *obligatory* labels $g(a)$. Note that in combination with the leaf constraint, this constraint enforces that all structural nodes have leaf nodes as descendant.

Maximal subtree mining. We wish to ensure that the patterns found are as large as possible, while also being nonredundant. We propose to solve this using the following new idea: in a first phase, we find all patterns under the earlier mentioned constraints, combined with a maximum size constraint. This constraint limits the size of the search space. Subsequently, we grow the patterns found under these constrains as large as possible, and return the maximal patterns among these large patterns.

More formally, let $\mathcal{T}_{cm}$ be the set of subtrees identified using constraints C0-C7, including a maximum size constraint and let $\mathcal{T}_c$ be the set of trees that satisfies constraints C0-C7, without maximum size constraint. Let $occ(T)$ be the root occurrences of a particular tree. Let $C(T) = \{T' \in \mathcal{T}_c \mid occ(T) = occ(T')\}$. Then we wish to find: $\max(\cup_{T \in \mathcal{T}_{cm}} C(T))$.

4 The FREQTALS Tree Mining Algorithm

In this section, we present FREQTALS, an extension of the FREQT algorithm that takes into account the novel ideas described in Section 3.

Constraints C3–C5 are all *anti-monotonic* in the following sense: if a tree does not satisfy the constraint, any supertree with the same root will not satisfy it either. To deal with such constraints we modified the `prune` function: extensions that do not satisfy the constraints are not added as candidates.

Constraints C6 and C7 are harder to implement, as these constraints are not *anti-monotonic*. For instance, when we start the search process, the pattern will certainly not contain leaf labels; they will only be added later. However, FREQT grows patterns only by adding nodes to the right of the rightmost path. This enables us to deal with C6 and C7 as follows.

For **C6**, we know that the only leaf that we can still add a child to, is the rightmost node. Hence, if any leaf other than the rightmost node has a label not in the set of permitted leaf labels Σ_{leaf}, the search process will not be able to resolve this violation. Hence, in `prune` we add a condition that any tree in which a leaf other than the rightmost node has a label not in Σ_{leaf} is pruned.

For **C7**, we exploit that obligatory child nodes of a structural node must occur in a specific order. Consider a structural label with three obligatory child labels $\sigma_1, \sigma_2, \sigma_3$. If a pattern already includes σ_1 and σ_3, the algorithm will not be able to add σ_2. In `prune` we add a condition so that any tree with such a situation is pruned.

Algorithm 3: FREQTALS algorithm

```
   input : 𝒟, constraints C0–C7.
   output: ℳ𝒫.
   /* Step 1: mine subtrees under constraints C0-C7 using FREQT with
      modified Add and Prune functions                                  */
1  ℱ𝒫 = FREQT(𝒟)
   /* Step 2: group the subtrees                                        */
2  ℛ𝒪ℳ ⟵ groupRootOccurrence(ℱ𝒫)
   /* Step 3: find the maximal subtrees under constraints C2-C7        */
3  ℳ𝒫 = ∅
4  for each r ∈ ℛ𝒪ℳ do
5  |  c ⟵ root label of r
6  |  mineMaximalSubtrees(c, r, ℳ𝒫)
7  output(ℳ𝒫)
```

Maximal subtree mining. The most naïve algorithm to find maximal patterns would be one in which we grow a maximal pattern for each pattern satisfying the earlier constraints. While correct, this algorithm would also be time consuming. Algorithm 3 shows an outline of FREQTALS, which solves the problem more efficiently, while finding the same set of patterns. It has three phases:

1. search frequent subtrees under constraints C0–C7;
2. group frequent subtrees by root occurrences;
3. for each set of root occurrences identified, run a search process (without C0 and C1) to identify the maximal subtrees having these root occurrences.

Delving into more detail, in Line 1 we call the FREQT algorithm, using the modified `add` and `prune` functions. Furthermore, we add an optimisation so that any tree having a frequent extension, will not be put in $\mathcal{FP}$.

In Line 2 we group the root occurrences. Essentially, for all frequent patterns found, we first compute the set: $\mathcal{RO} = \{occ(T) \,|\, T \in \mathcal{FP}\}$. Note that multiple trees in $\mathcal{FP}$ may have the same $occ(T)$. Hence, this set is smaller than the original set of patterns. Subsequently, we only keep those sets of root identifiers that are minimal: $\mathcal{ROM} = \{r \in \mathcal{RO} \,|\, \nexists r' \in \mathcal{RO} : r' \subset r\}$. This optimisation does not affect our results. The key idea is that a pattern appearing in the larger set of occurrences, will also appear in the smaller set of occurrences.
Subsequently, in line 6 we start a search for maximal patterns for each set of root occurrences $r \in \mathcal{ROM}$. Here, for reasons of simplicity we made the choice to use a modified version of FREQT:

- we start the search with the root label appearing in the root occurrences r;
- the root occurrences considered during the search are only those in r, even if the root label has more occurrences in the original data;
- instead of using the minimum support constraint, we impose the constraint that the patterns searched for appear in all the given root occurrences;
- we do not apply a maximum size constraint;

Constraint	Variable	Value
C0	Minimum Support Threshold	5
C1	Maximum # of Leaves	4
C2	Minimum # of Leaves	2
C3	Root Labels	TypeDeclaration, Block
C4	Black List Labels	Javadoc, Modifiers, Annotations, ...
C5	Maximum # of Similar Siblings	10

Table 1. FREQTALS configuration for CheckStyle

– for each pattern that is generated, we check whether it should be included in $\mathcal{MP}$ and we update $\mathcal{MP}$ accordingly.

Note that patterns considered by `mineMaximalSubtrees` may have more occurrences in the original data. This is not harmful, as any such pattern will still be maximal and frequent. The key idea is that running FREQT on a smaller set of root occurrences, with a constraint that does not allow to lose any root occurrence, makes the search more efficient.

5 Empirical Evaluation

To evaluate FREQTALS, we carried out an empirical study on source code written in Java. We analysed the results from a qualitative (Section 5.1) and a quantitative (Section 5.2) point of view. More specifically, we analyse CHECKSTYLE, a well-documented static code analysis tool for Java that was selected from the the Qualitas Corpus [6].

Table 1 reports how we configured the algorithm for our evaluation. A minimum support threshold of 5 was chosen as for lower values the number of patterns exploded. We also focused only on AST sub-trees with root nodes of type `TypeDeclaration` (*i.e.,* a Java method definition) or `Block` (*i.e.,* a Java method body), because we were interested in the program logic.

5.1 Qualitative Analysis

The main purpose of our qualitative analysis is to determine whether the patterns identified by our algorithm are indeed useful.

With the given configuration, FREQTALS found 147 patterns that we manually analysed. To illustrate the characteristics of the patterns mined by FREQTALS, below we provide a detailed analysis of some of the patterns shown in Figure 1. The CHECKSTYLE tool implements several design patterns such as the Visitor. Thus, some combinations of abstract methods are reused among many different classes (*e.g.,* `getDefaultTokens()` and `visitToken()`) and it is not surprising that our algorithm discovers many patterns with this pair of methods. Overall, 83 out of 147 mined patterns contained this pair. Pattern 34 shows an example of such a design pattern instance that contains 4 reused methods:

```
public final class ReturnCountCheck extends AbstractFormatCheck {
   ...
   @Overwrite
   public int[] getDefaultTokens(){
      return new int[] {
         TokenTypes.CTOR_DEF,
         TokenTypes.METHOD_DEF,
         TokenTypes.LITERAL_RETURN,
         };
      }
   ...
   @Overwrite
   public int[] getRequiredTokens() { … }
   ...
   @Overwrite
   public void visitToken(DetailAST aAST){
      switch (aAST.getType()) {
         case TokenTypes.CTOR_DEF:
         case TokenTypes.METHOD_DEF:
         ...
         break;
         ...
         default:
         ...
         }
   }
   ...
   @Overwrite
   public void leaveToken(DetailAST aAST){
      switch (aAST.getType()) {
         case TokenTypes.CTOR_DEF:
         case TokenTypes.METHOD_DEF:
         ...
         default:
         ...
         }
   }
   ...
}
```

Pattern 34: An instance of Checkstyle's Visitor design pattern

```
private void visitMethod(final DetailAST aMethod)
   ...
   DetailAST child = objBlock.getFirstChild();
   while (child != null) {
      if (child.getType() == TokenTypes.METHOD_DEF) {
         ... }
      child = child.getNextSibling();
}
```

Pattern 27: AST traversal.

```
@Override
public void leaveToken(DetailAST aAST) {
   switch(aAST.getType()) {
      case TokenTypes.OBJBLOCK:
      case TokenTypes.SLIST:
      case TokenTypes.LITERAL_FOR:
      ...
}
```

Pattern 9: Check for different blocks.

```
public int[] getDefaultTokens(){
    return new int[] {
       TokenTypes.ASSIGN,            // '='
       TokenTypes.DIV_ASSIGN,        // "/="
       TokenTypes.PLUS_ASSIGN,       // "+="
       …
    };
}
```

Pattern 18: Method structure.

```
private boolean checkParams(DetailAST aMethod){
   ...
   if ((aAST.getType() == TokenTypes.VARIABLE_DEF) ||
       (aAST.getType() == TokenTypes.PARAMETER_DEF))
      {
       …
      }
```

Pattern 140: IF Statement.

Fig. 1. Examples of patterns found in CHECKSTYLE

`getDefaultTokens`, `getRequiredTokens`, `visitToken`, and `leaveToken`. Pattern 27 shows a recurrent code snippet that checks every node of a given AST. Pattern 9 is an interesting example of a code structure that checks for different types of AST objects. This structure is quite frequent in CHECKSTYLE since it allows developers to customise the framework to write their own kinds of source code checks. Pattern 140 is also a typical code idiom recurring in CHECKSTYLE.

5.2 Quantitative Analysis

We limit our quantitative comparison between FREQT and FREQTALS to CHECKSTYLE, the dataset already considered in Section 5.1. We set C0 to 8, while keeping the same values, shown in Table 1, for the other settings. It is worth noting that for a more fair comparison, FREQT was modified to add a constraint on the minimum and maximum number of leaves. FREQTALS discovered 1,288 frequent patterns in 23 seconds, while the original FREQT did

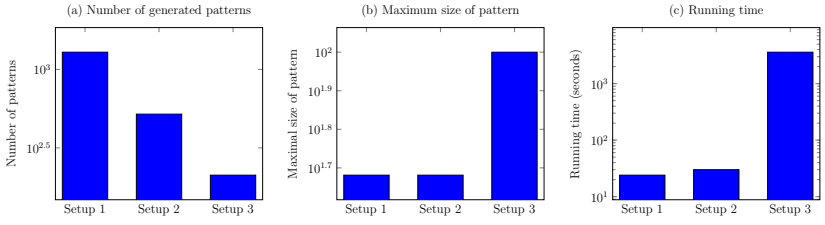

Fig. 2. Comparisons of three setups

not finish within the search budget (*i.e.,* 60 minutes); it found 717,859 patterns within these 60 minutes.

To evaluate the different steps of FREQTALS, we executed it in three setups: the first applies constraints C0-C7; the second filters the results obtained by the first to keep only the maximal frequent patterns; the third applies all the constraints (except C0 and C1) combined with maximal subtree mining. We ran these three setups on entire CHECKSTYLE project. Similar to previous experiment, C0 was set to 8, and other settings were kept the same as shown in Table 1. Figure 2 shows the results. Note that to compare the results more easily we used a logarithmic scale for the plots in this Figure. In the first plot, we observe that the number of patterns discovered by the third setup is much smaller, as intended. Similarly, in the second plot, we see that the maximum size of the patterns mined by the third setup is larger, as desired. Nevertheless, there is no free lunch, as in the third plot we can observe that the third setup is more time consuming.

6 Related Work

This section discusses related work concerning pattern mining of (i) general semi-structured data and (ii) source code regularities and idioms.

Pattern Mining of Semi-structured Data. Extensive, but rather old literature exists on frequent tree mining algorithms [1, 2]. Such algorithms can be categorised according to their input data, type of output patterns, and the approach taken for mapping patterns to data. Only algorithms designed for ordered, rooted trees, using an induced subtree relation are relevant to this work. The most well-known such algorithm is FREQT [5]. A benefit of FREQT is that it is a conceptually simple algorithm in which it is easy to add constraints. However, a major problem of frequent tree mining algorithms is that the number of output patterns is often very large. To tackle this problem, maximal frequent tree mining algorithms, i.e., CMTreeMiner [1], were developed. However, none of these algorithms operate on ordered trees. In recent years few new algorithms have been proposed, due to a lack of applications of such algorithms. Our work addresses this weakness by showing how pattern mining algorithms can indeed find useful patterns, as validated by software engineers. A notable exception is an algorithm that operates on attributed trees [7]. Our trees are not attributed, and hence we could not apply this algorithm.

Mining Software Patterns. There is an extensive literature on applying mining algorithms to software artefacts in general. Early examples include applications of formal concept analysis [8] and of association rule mining [9] for discovering design regularities. Narrowing down to the discovery of source code regularities, Allamanis *et al.* [4] describe an approach that mines for code idioms in a corpus of idiomatic code using non-parametric Bayesian methods. Similar approaches, like Bhatia *et al.* [10] mine for idioms using recurrent neural networks, aiming to correct incorrect uses of coding idioms. An advantage of FREQTALS is that the criteria used to include patterns in the output of the

algorithm remain easy to understand, even for experts without background in statistics.

7 Conclusion and Future Work

In this paper, we proposed the FREQTALS algorithm, an extension of FREQT that combines *maximal frequent subtree mining* and *constraint-based data mining* to mine structural source code regularities. Experimental results show that (i) a significant reduction of the execution time and the number of discovered patterns with respect to the original frequent tree mining algorithm, (ii) that many of the discovered patterns highlight relevant code regularities, (iii) that some of the patterns found are significantly larger than expected. However, choosing appropriate configurations for a programming language is a difficult task. We envision to replicate our empirical evaluation on a larger set of systems and to define new guidelines to help software engineers in configuring our algorithm.

Acknowledgments This work was conducted in the context of an industry-university research project between UCLouvain, Vrije Universiteit Brussel and Raincode Labs, funded by the Belgian Innoviris TeamUp project INTiMALS (2017-TEAM-UP-7).

References

1. Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent subtree mining–an overview," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 161–198, 2005.
2. A. Jiménez, F. Berzal, and J. C. C. Talavera, "Frequent tree pattern mining: A survey," *Intell. Data Anal.*, vol. 14, no. 6, pp. 603–622, 2010.
3. C. C. Aggarwal and J. Han, Eds., *Frequent Pattern Mining.* Springer, 2014.
4. M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 472–483.
5. T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," *IEICE TRANSACTIONS on Information and Systems*, vol. 87, no. 12, pp. 2754–2763, 2004.
6. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "The qualitas corpus: A curated collection of java code for empirical studies," in *Software Engineering Conference, 2010 17th Asia Pacific.* IEEE, 2010, pp. 336–345.
7. C. Pasquier, J. Sanhes, F. Flouvat, and N. Selmaoui-Folcher, "Frequent pattern mining in attributed trees: algorithms and applications," *Knowl. Inf. Syst.*, vol. 46, no. 3, pp. 491–514, 2016.
8. K. Mens and T. Tourwé, "Delving source code with formal concept analysis," *Comput. Lang. Syst. Struct.*, vol. 31, no. 3-4, pp. 183–197, Oct. 2005.
9. A. Lozano, A. Kellens, K. Mens, and G. Arevalo, "Mining source code for structural regularities," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering.* IEEE Computer Society, 2010, pp. 22–31.
10. S. Bhatia and R. Singh, "Automated correction for syntax errors in programming assignments using recurrent neural networks," *arXiv preprint arXiv:1603.06129*, 2016.