UCLouvain

epl

École polytechnique de Louvain

# Can Data Mining Discover Software Changes ?

Author: **Quentin Hauspie**
Supervisors: **Kims Mens, Siegfried Nijssen**
Reader: **Hélène Verhaeghe**
Academic year 2020–2021
Master [120] in Computer Science

# Abstract

Software development is a long and incremental process during which several modifications are made to the source code. Searching for regularities in the made changes can be of great interest. The approach presented in this work uses pattern mining techniques to capture these modifications' good and bad practices. Based on experiments, this work aims first to validate the approach; confirm the quality of the results, and second to verify the approach; ensure the correctness of the method.

**Keywords:** Pattern mining, Frequent subtree mining, Source code regularities

# Acknowledgments

*I would like to thank my supervisors, Prof. Kim Mens and Prof. Siegfried Nijssen, for their availability every week and their wise advice to carry out this work.*

*I would also like to thank my family for their unfailing support throughout this research.*

# Contents

**1 Introduction** **1**

**2 Background** **3**
2.1 Abstract Syntax Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Patterns in Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Frequent tree pattern mining . . . . . . . . . . . . . . . . . . . . . . . 7
2.3.1 Problem definition . . . . . . . . . . . . . . . . . . . . . . . 8
2.3.2 Strategies . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3.3 FreqT Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 15

**3 Freqtals** **17**
3.1 Implementation of FreqTals . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Extension to 2-class data . . . . . . . . . . . . . . . . . . . . . . . . 23
3.3 Framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

**4 Evaluation of FreqTals** **34**
4.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.2 Dataset . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.3 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**5 Weaknesses and Enhancement** **46**
5.1 Chi-square score . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.2 Implementation of White List Label . . . . . . . . . . . . . . . . . . . 48

**6 Validation of new implementation** **51**
6.1 Quantitative analysis comparison . . . . . . . . . . . . . . . . . . . . 52
6.2 Qualitative analysis comparison . . . . . . . . . . . . . . . . . . . . . 52

**7 Discussion** **54**
7.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
7.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**A Grammar Building** 56

**B Configuration file** 60

# Chapter 1

# Introduction

Software development is an incremental process. It means that the software goes through several iterations before arriving at the final version, the one that meets expectations. Source code repositories such as Git [1] are good examples to highlight the evolutionary character of software development.

Between two versions of software, the latter's source code evolves; bits of code have been added or deleted, while other bits of code may have been modified. We can use data mining to find patterns in these pieces of code.

This work aims to find patterns, also called regularities, in these modified code snippets that reflect good or bad programming practices. These regularities can help to understand the code, refactor the code or even locate faults in the code [2]

To achieve this goal, we start by introducing pattern mining, and more precisely, the field of frequent tree pattern mining, since it is this field that interests us in this work. In this domain, where there are many algorithms, we focus on two algorithms that fit our needs well; $FreqT$, which mines patterns in trees, and $FreqTals$, which is an extension of $FreqT$ and which can mine patterns between two versions of software. $FreqTals$ fits our needs perfectly, and that's why we use this algorithm. For ease of use, we include it in a framework that makes pattern mining more user-friendly.

Once we have presented this whole environment, we validate this approach by considering a practical software case. Depending on the results obtained, we would be able to validate or not this approach to confirm that it meets our basic needs. In addition to validating it, we also have to verify it; even if the results obtained are good, this approach must be correct.

Finally, we take a step back and have a critical opinion on this approach. In addition, we discuss some possible improvements for the future.

# Chapter 2

# Background

In this chapter, we present the domain in which this thesis is located, in a bottom-up way, starting from the most elementary data type in this domain, the tree, and in particular a data type to represent source code, the Abstract Syntax Tree.

We then explain the notion of pattern in a tree (also called subtree) and the families of subtrees that exist in current literature.

To conclude this chapter, we define the frequent tree pattern mining problem concretely. We discuss existing algorithms to solve this problem and the different strategies used by them. Primarily, we study the technique used by $FreqT$, the algorithm we are interested in since it is at the origin of $FreqTals$. After presenting the strategy used by $FreqT$, we discuss its algorithm.

Although this chapter presents the theoretical foundations underlying $FreqTals$, concrete examples will illustrate the theoretical concepts and make the chapter more understandable.

## 2.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a data structure for representing the syntactic skeleton of a program's source code. In Figure 2.1, we have a code snippet (much more straightforward than the programs we will have to manage) and the AST corresponding to this code. As this example shows, this data structure represents the structure of the code and the content of the code. The advantage of the AST over the source code is that it does not explicitly represent the text that could be characterized as unnecessary, present in the source code. An example of such text is punctuation and separations. In Figure 2.1, we can see that semicolons, brackets,

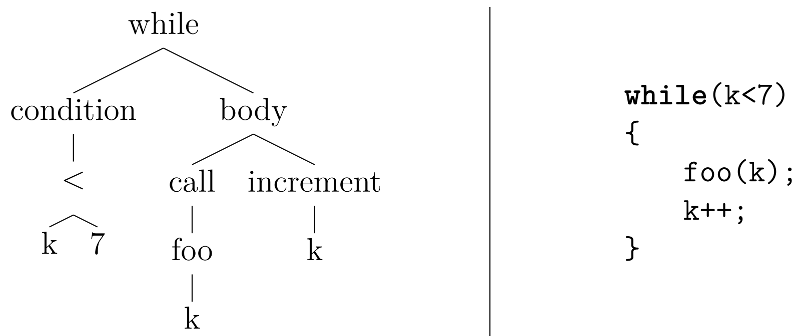



Figure 2.1: Abstract Syntax Tree example

or parentheses are not explicitly represented in the Abstract Syntax Tree.

The AST is a specific case of a tree [3]; it's a **rooted tree**, nodes have labels (**labeled tree**) and have arbitrary number of children (from 0 to $N$). The number of children does not have to be the same for all nodes, unlike full m-ary trees where every node can have either 0 children or $m$ children. In a tree, nodes with 0 children are called leaf nodes while the others are called internal nodes.

Trees are categorized according to the importance given to the order of the sibling nodes; ordered, unordered, or partially ordered. Most of the time, the order in the source code is significant, so it goes without saying that in the case of ASTs, we are in the first category, where there is a predefined order within sets of sibling nodes (**ordered tree**).

Another important aspect of trees is how they can be represented as a single and unique string; we need a one-to-one correspondence between the labeled rooted ordered tree and its string representation. Since most tree mining algorithms use trees in string format for reasons of space optimisation [4], we must be able to transform a tree into a string (algorithm input) and be able to transform a string into a tree (algorithm output). It implies that it is essential to be able to find the tree from its string representation, as well as the other way around. This property, called the canonical tree representation, allows us to compare trees and enumerate subtrees much more easily. Three solutions have been presented in literature [5, 6]:

- Depth-First codification
  We go through the tree in depth-first order. When we traverse a node, we add its label to the end of the string, and each time we move from a child to a parent (backward), we add a marker. This marker must not be present in the set of labels used.

- Breadth-First codification
  We traverse the tree in breadth-first order, and we add the label of each node traversed at the end of the string. Unlike the previous technique, as we never go back up the tree, the marker is used here to separate the sets of siblings.

- Depth-Sequence-based codification
  We go through the tree in depth-first order, as in the first technique. But instead of using a marker as we move up the tree, this technique constructs, for each node traversed, a pair $(d,l)$ where $d$ is the depth of the node in the tree and $l$ is the label of the node. The pair is added at the end of the string.

The $FreqT$ algorithm, which we present in detail in subsection 2.3.3, and in particular the implementation we consider [7], uses a small variant of Depth-First codification. Indeed, it uses marker "(" to designate the fact that one goes down in the tree, and it also uses marker ")" to designate the fact that one goes up in the tree. Considering the Abstract Syntax Tree of our example (Figure 2.1), the string representation made by $FreqT$ is the following one:

$$(while(condition(<(k)(7)))(body(call(foo(k)))(increment(k))))$$

## 2.2 Patterns in Trees

There are many ways to define a pattern (a subtree) in a tree depending on the constraints we impose on the relationships between nodes in the tree that must be preserved.
Figure 2.2 shows five subtree relationships from literature [3]. We have deliberately chosen to represent them as a set to highlight the inclusion relationships between them. Indeed, let us take a tree $T$ and another tree $T'$, *bottom_up_subtrees(T',T)*⇒ *induced_subtrees(T',T)* ⇒ *embedded_subtrees(T',T)* ⇒ *incorporated_subtrees(T',T)* ⇒ *subsumed_subtrees(T',T)*.
Another way of analysing this would be to say that the most restrictive approach is bottom-up subtrees and the least restrictive subsumed subtrees.

There is a link between the restrictiveness and the number of patterns found. A less restrictive approach, such as subsumed subtrees, will find many patterns, which could be a problem for some input trees. Conversely, a very restrictive approach such as bottom-up subtrees will find a small number of patterns.

The $FreqT$ algorithm aims to identify induced subtrees, which is a good compromise between restrictiveness and the number of patterns found.

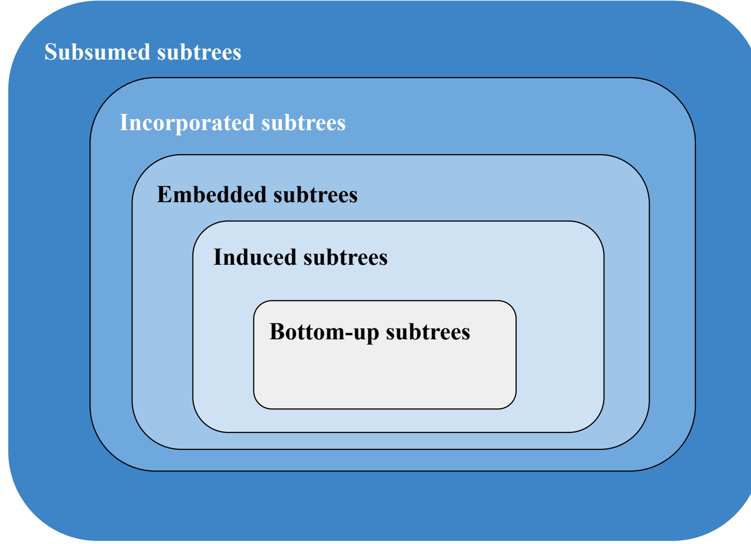



Figure 2.2: Subtree relationships

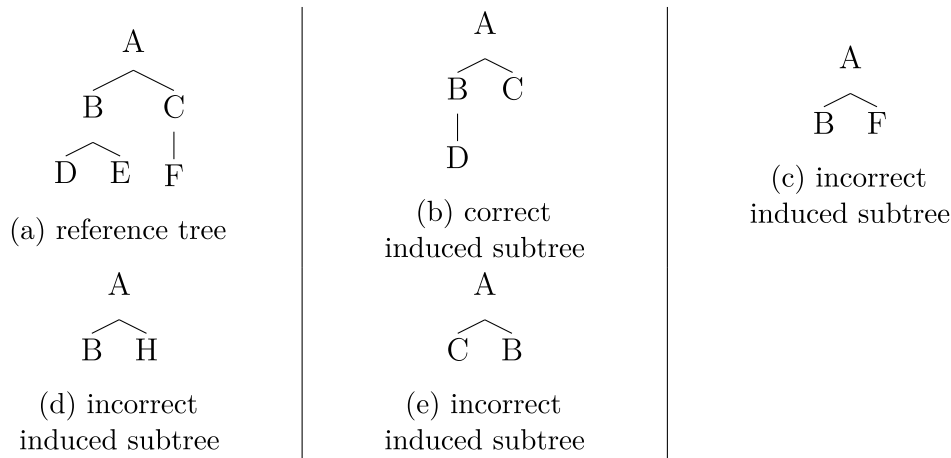
Before defining the induced subtree relationship between two trees mathematically, we need to formalize the way trees are defined [8]. An ordered tree $T$ is characterized by the tuple $(V,E,\lambda,\Sigma)$ where $V$ is the set of node identifiers (from one to $n$)[1], E is the set of edge identifiers (E $\subseteq$ V×V), $\lambda$ is the function that gives the label of a given node identifier (V $\mapsto$ $\Sigma$) and $\Sigma$ is the set of allowed labels.

In a formal way [2], given two trees $T_1 = (V_1,E_1,\lambda_1,\Sigma_1)$ and $T_2 = (V_2,E_2,\lambda_2,\Sigma_2)$, we define $T_2$ as an induced subtree of $T_1$ if and only if there exists an injective[2] function $f : V_2 \mapsto V_1$ that satisfies the three following conditions:

1. Edge are preserved : $\forall(v,v') \in E_2 : (f(v),f(v')) \in E_1$.
2. Labels are preserved : $\forall v \in V_2 : \lambda_2(v) = \lambda_1(f(v))$.
3. Order is preserved : $\forall v_1,v_2 \in V_2 : v_1 < v_2$ in $V_2 \implies f(v_1) < f(v_2)$ in $V_1$.

Figure 2.3 shows a concrete case of application of the induced subtree relationship. We have a reference tree, a correct induced subtree of this tree, and three examples of trees that are not induced subtrees of this tree because each violates one of the conditions.

[1] By convention, the node identifiers are assigned following a preorder traversal of the tree.
[2] Injective means that $\forall\ v,v' \in V_2,\ f(v) == f(v') \implies v == v'$.



(a) reference tree
(b) correct induced subtree
(c) incorrect induced subtree
(d) incorrect induced subtree
(e) incorrect induced subtree

Figure 2.3: Induced Subtrees example

Figure 2.3b is a correct induced subtree because it meets the three necessary conditions.

Figure 2.3c is not an induced subtree of the tree because it does not satisfy condition 1; the edge ($A$,$F$) is present in Figure 2.3c but is not present in the reference tree, and therefore the edges are not preserved.

Figure 2.3d is not an induced subtree of the tree because it does not satisfy condition 2; the label $H$ is present in Figure 2.3d but is not present in the reference tree, and therefore the labels are not preserved.

Figure 2.3e is not an induced subtree of the tree because it does not fulfill condition 3; $C > B$ ($C$ is before $B$) in Figure 2.3e, while $B > C$ ($B$ is before $C$) in the reference tree and so the order is not preserved.

## 2.3 Frequent tree pattern mining

Tree Pattern Mining consists in identifying subtrees (patterns) in a collection of trees. In our case, the subtrees to be identified are induced subtrees, and the collection of trees is a collection of Abstract Syntax Trees, representing the source code of a system. In this section, we define the problem formally and present some strategies used to solve it.

### 2.3.1 Problem definition

Given a collection of trees $D$, also known as the database, and a minimum threshold $\mu$, the objective is to discover all the *interesting* subtrees in $D$. To do this, we define what is the support of a subtree, with the help of the two functions *δ(S,T)* and *d(S,T)* [3].

The first function, defined in Equation 2.1, returns the number of occurrences of subtree $S$ in tree $T$. The second function, defined in Equation 2.2, returns a boolean indicating whether or not the tree $T$ contains the subtree $S$.

$$\delta(S,T) = \#occurrences\ of\ subtree\ S\ in\ tree\ T \tag{2.1}$$

$$d(S,T) = \begin{cases} 1 & if\ \delta(S,T) > 0 \\ 0 & if\ \delta(S,T) = 0 \end{cases} \tag{2.2}$$

Given these two functions we can calculate the support of a subtree $S$, which can be computed in 2 different ways.

- *support(S,D)* is the number of trees in $D$ that contain at least one occurrence of $S$. This number is always bounded ($0 \leq support(S,D) \leq |D|$)

$$support(S,D) = \sum_{T \epsilon D} d(S,T)$$

- $support_{weighted}(S,D)$ is the total number of occurrences of $S$ among all the trees of $D$. This number has no reasonable upper bound ($0 \leq support_{weighted}(S,D)$)

$$support_{weighted}(S,D) = \sum_{T \epsilon D} \delta(S,T)$$

With this definition of support, we can specify what we meant by *interesting* subtrees. The frequent tree mining problem consists in finding **all** the subtrees that are frequent, i.e. with a support greater than or equal to the threshold ($\mu$) defined by the user.

$$frequent(S,D) = \begin{cases} 1 & if\ support(S,D) \geq \mu \\ 0 & otherwise \end{cases}$$

Most of the tree mining algorithms tackle the problem as defined so far, but this problem can be made a little more complex by adding two properties on the set of

obtained subtrees. These two properties are *maximal* and *closed* [9].

Consider the result set of the frequent tree mining problem, denoted $S_{res}$.
The set $S_{res}$ is maximal if and only if there is no subtree $T_1$ and subtree $T_2$ in this set $S_{res}$ such that the first is a subtree of the second. Formally, we have this:

$$\nexists T_1, T_2 \in S_{res} : T_1 \textit{ is a subtree of } T_2$$

For example, the $Set_1$ shown in Figure 2.4 is not a maximal set, because there exists two subtrees ($t_3$ and $t_1$ or even $t_3$ and $t_2$) such that one ($t_3$) is a subtree of the other ($t_1$ or $t_2$). But on the other hand, when we remove $t_3$ from the $Set_1$, we get the $Set_2$, which is a maximal set because there do not exist two subtrees such that the first is a subtree of the second.

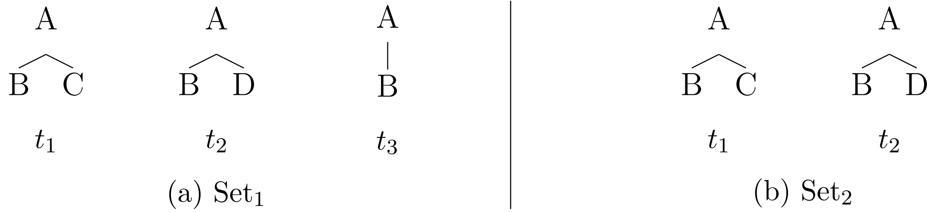



Figure 2.4: Tree dataset example

Considering the same set $S_{res}$, we can define the closed property as follow.
The set $S_{res}$ is closed if and only if there is no subtree $T_1$ and subtree $T_2$ in this set $S_{res}$ such that the first is a subtree of the second and their support is identical. Formally, we have this:

$$\nexists T_1, T_2 \in S_{res} : T_1 \textit{ is a subtree of } T_2 \wedge support(T_1, D) == support(T_2, D)$$

Considering the two datasets proposed in Figure 2.4 and adding 2,3,2 as support value for $t_1$,$t_2$ and $t_3$ respectively, the $Set_1$ is not a closed set, because there are two subtrees ($t_3$ and $t_1$) such that the first one ($t_3$) is a subtree of the second one ($t_1$) and their support value is identical (both have two as support). By removing $t_3$, we obtain the $Set_2$ which is a closed set because there do not exist anymore two subtrees such that the first is a subtree of the second and their support value is identical.

These two properties generally allow decreasing the number of subtrees in the result set by eliminating some redundancy.

### 2.3.2 Strategies

There are many frequent tree mining algorithms in literature [3]. We can categorize them along two axes:

| Algorithm | Input Trees | | | | Identified Patterns | | | |
|---|---|---|---|---|---|---|---|---|
| | Ordered trees | Partially-ordered trees | Unordered trees | Not rooted trees (free) | Induced subtrees | Embedded subtrees | Incorporated /Subsumed subtrees | Maximal/ Closed |
| FreqT | ✓ | | | | ✓ | | | |
| AMIOT | ✓ | | | | ✓ | | | |
| uFreqT | | | ✓ | | ✓ | | | |
| HybridTreeMiner | | | ✓ | ✓ | ✓ | | | |
| Unot | | | ✓ | | ✓ | | | |
| FreeTreeMiner | | | ✓ | ✓ | ✓ | | | |
| FreeTreeMiner' | | | ✓ | ✓ | ✓ | | | |
| GASTON | | | ✓ | ✓ | ✓ | | | |
| X3Miner | ✓ | | | | | ✓ | | |
| MB3Miner | ✓ | | | | | ✓ | | |
| IMB3Miner | ✓ | | | | | ✓ | | |
| TreeMiner | ✓ | | | | | ✓ | | |
| TreeMinerD | ✓ | | | | | ✓ | | |
| RETRO | ✓ | | | | ✓ | ✓ | ✓ | |
| Chopper | ✓ | | | | | ✓ | | |
| XSpanner | ✓ | | | | | ✓ | | |
| Uni3 | | | ✓ | | | ✓ | | |
| Phylominer | | | ✓ | | | ✓ | | |
| SLEUTH | | | ✓ | | | ✓ | | |
| POTMiner | ✓ | ✓ | ✓ | | ✓ | ✓ | | |
| TRIPS | ✓ | | ✓ | | ✓ | ✓ | | |
| TIDES | ✓ | | ✓ | | ✓ | ✓ | | |
| CMTreeMiner | ✓ | | ✓ | | ✓ | | | ✓ |
| PathJoin | | | ✓ | | ✓ | | | ✓ |
| DRYADE | | | ✓ | | | ✓ | | ✓ |
| TreeFinder | | | ✓ | | | | ✓ | ✓ |

Table 2.1: Frequent tree mining algorithms [3]

- Input trees
  It must be decided which type of trees can be used as input. The algorithm could use specific properties of the tree (e.g., rooted or ordered).

- Identified subtrees (patterns)
  The algorithms choose to find induced, embedded, or subsumed/incorporated subtrees. Combined with this, some algorithms decide to look for maximal and/or closed results.

Table 2.1 shows an overview of the existing algorithms in literature and their categorization.

Of all the algorithms proposed in the literature, most are based on one of the following frequent itemset mining algorithms: Apriori [10] and FP-Growth [11]. As these two algorithms work on itemsets, frequent tree mining algorithms must adapt them to trees. Apriori is the most used one, but *FreqT* adopts a pattern growth approach [12] somewhat similar to FP-Growth because Apriori has significant

drawbacks in our case. Indeed, Apriori requires many database scans to calculate the support of a subtree. In the case of Abstract Syntax Trees, we have many trees in the database, but also the size of these trees is not negligible, which makes the use of Apriori not suitable and tractable.

The approach used by *FreqT* is based on two main properties: Anti-monotonicity and Depth-first search.

- Anti-monotonicity
  A function f is said to be anti-monotonic if and only if for two elements X and Y, we have X $\subseteq$ Y, then f(X) $\geq$ f(Y).[3]
  In our case, the *support* function is indeed anti-monotone. This means that given a tree A that is frequent, we know that all subtrees of A will also be frequent. Conversely, given a tree B that is not frequent, we know that all trees of which B is a subtree will not be frequent either.

- Depth-first search
  The search space is browsed in a depth-first manner. Given a frequent subtree $F_i$ of size $i$, we generate potential candidates of size $i + 1$ that we then recursively traverse in depth-first order, from left to right.

We can cut the pattern growth approach into two distinct parts; initialization and expansion. During the initialization, we compute the itemsets of size one that are frequent. For trees, this corresponds to frequent subtrees of size one (i.e., frequent labels). In the expansion part, we iterate over those frequent subtrees of size one, and from these roots, we build larger and larger frequent subtrees in a recursive depth-first way. Each recursion of the expansion part can be split into two steps: candidate generation and support calculation.

### Candidate Generation

Starting from a frequent subtree of size $i - 1$ obtained in the previous recursion, we generate potential frequent subtree of size $i$, referred to as candidates. Thanks to the anti-monotonic property, a subtree of size $i$ that is not frequent cannot be used to generate a frequent subtree of size $i + 1$. Therefore, an infrequent subtree will not be extended.

There are many ways to generate candidates. Even in our case, where the data are trees, there are several different strategies. The one used by *FreqT* is rightmost expansion [8].
Based on a frequent subtree of size $k$, this strategy generates subtrees of size $k + 1$

[3]Since we are using trees, A $\subseteq$ B can be seen as A is a subtree of B

by adding a node on the rightmost branch of the tree.

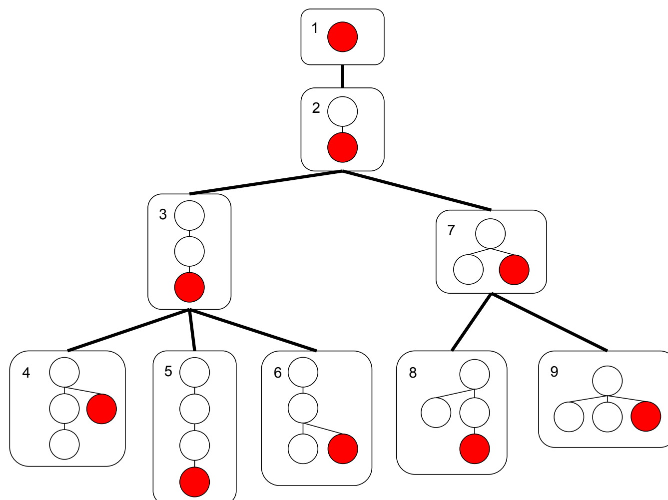



Figure 2.5: Illustration of rightmost expansion with unlabelled tree

Figure 2.5 shows, on an unlabeled tree, the search tree that we explore in a depth-first way left to right, starting from a tree of size one to a tree of size four. At each node, we first generate all the candidates generated if we use the rightmost expansion strategy (the node added to the tree is the red one). Then we go through these candidates in a depth-first way, left to right, only for the frequent ones. The indexes are explicitly set to highlight the depth-first traversal order of the search tree on the figure.

There are a few comments to make about this strategy.
First, by analyzing more closely the search tree shown in Figure 2.5 and comparing the subtrees rooted at index seven and the one rooted at index three, as Figure 2.6 does, we notice that the number of candidates generated for two trees of the same size (three), is not the same. The number of candidates generated for a frequent tree of size $i$ is an increasing function of the length of its rightmost path.

$$\#candidates\ generated \sim length(rightmost\ path)$$

Second, in Figure 2.5, we have voluntarily omitted to consider the labels of the nodes. In Figure 2.7, we compare the generation of candidates for a non-labeled tree and for a labeled tree with $\Sigma = \{$A,B,C$\}$. We see that the number of candidates

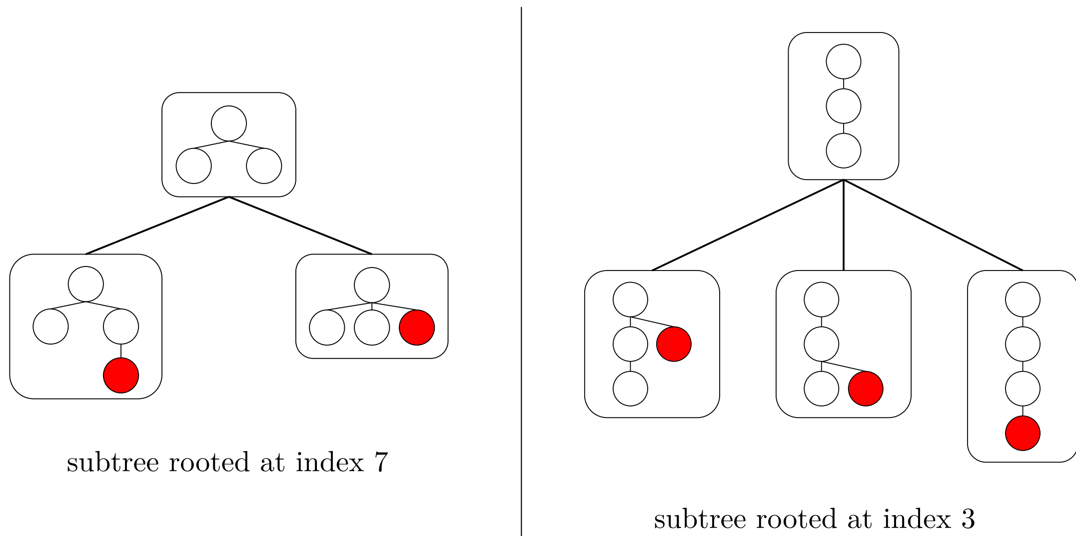



Figure 2.6: Impact of rightmost path length

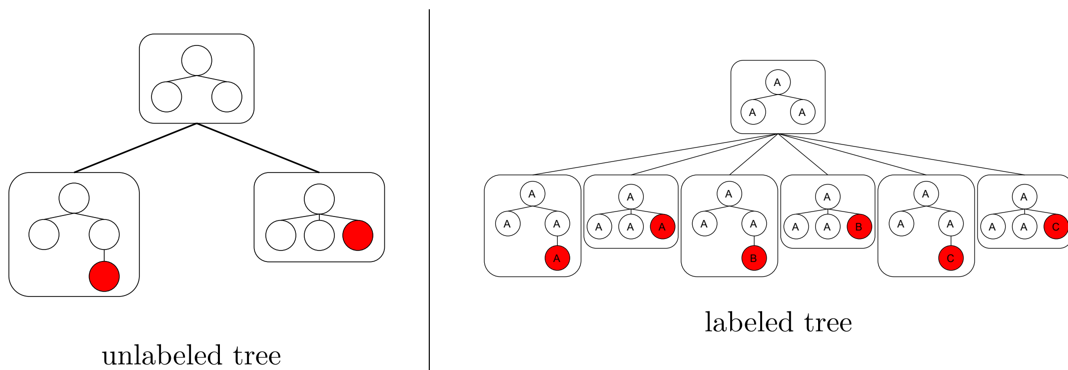



Figure 2.7: Impact of label size

generated is an increasing function of the size of the label set. More precisely, the number of generated candidates is multiplied by the size of the label set.

$$\#candidates\ generated \sim |\Sigma|$$

The first problem is inherent to tree expansion, and therefore there is nothing we can do, but it was necessary to point it out. On the other hand, for the second problem, we must find a solution because the size of the label set for a source code can be huge and cause a lot of issues in terms of the number of candidates generated.

The solution provided by *FreqT* is called collecting candidates from the data [12]. The overall idea is that, among the many candidates generated, most of them

do not appear in the database, so they have a support of zero and will be prune just after being generated. Collecting candidates from data consists in generating candidates based on what is in the database so that all generated candidates exist and have support at least equal to one. Using this technique, we do not lose any frequent candidates (since the ones that are "lost" did not exist in the database), and we significantly reduce the number of generated candidates.

### Support Calculation

The objective of this step is first to calculate the support of all generated candidates. In a second step, we have to eliminate the candidates that are not frequent. The remaining list contains frequent candidates, and thus we have the frequent patterns of iteration $i$. These are the elements that we go through one by one recursively. The algorithm does this as long as there are frequent candidates generated.

The most naive strategy to calculate the support of a pattern would be to traverse the database and check for each entry[4]. It is not difficult to convince oneself that this operation is too time consuming. That's why there are several techniques to make this step less time consuming.

The technique used by $FreqT$ is called rightmost occurrence lists (noted RMO-lists) [8]. For each candidate, we store a list of tuples (*tid*,$n$). *tid* allows us to identify an entry[4] in the database. $n$ represents the position of the candidate's rightmost leaf in the *tid* tree. Since we are using rightmost expansion strategy for candidate generation, we only need to store the rightmost occurrence of the candidate. Once RMO-lists are implemented correctly, the support computation is trivial since we have a tuple (*tid*,$n$) for each occurrence of the candidate.
Moreover, for a candidate of size $k$, its RMO-list can easily be computed from the RMO-list of the pattern of size $k-1$ from which it was generated.

Figure 2.8 represents a basic example to illustrate how the rightmost occurrence list of a pattern is managed. Consider a database containing two trees: $t1$ and $t2$. For each node of $t1$ and $t2$, we have deliberately indicated its position in the tree (index of tree preorder traversal). The right part of the figure shows how the rightmost occurrence list is modified as the pattern grows from a size of one to a size of three.

This example also convinces us that it is straightforward to compute the RMO-list of a pattern. The time saved by the support computation with RMO-lists is not

[4]In our case, an entry is an element of the tree collection $D$.

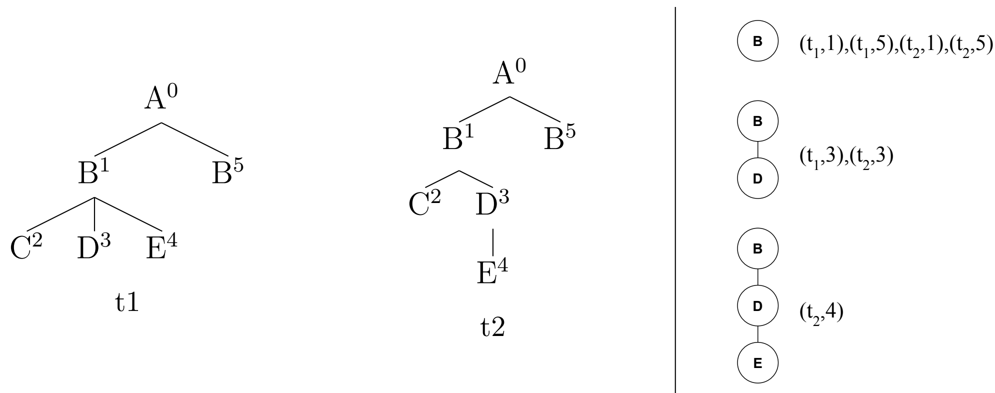


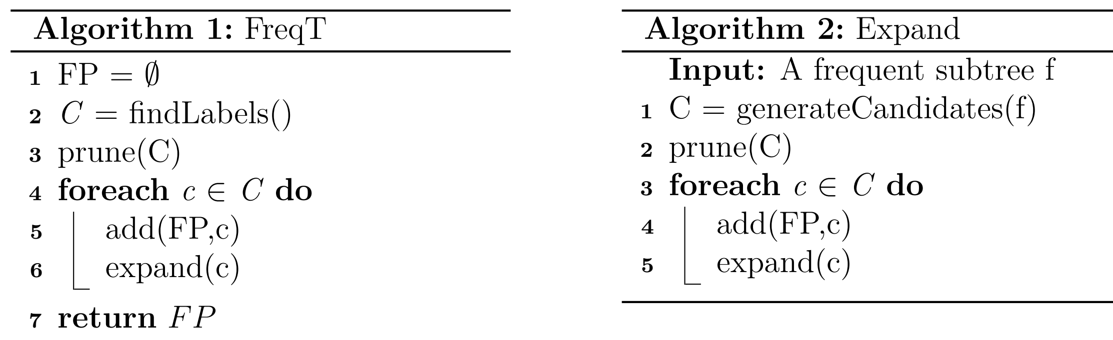

Figure 2.8: RMO-lists example

negligible compared to the naive strategy presented above.

### 2.3.3 FreqT Algorithm

In this section, we give a summary on *FreqT*, which is the frequent tree mining algorithm used by *FreqTals*.

*FreqT* algorithm aims to identify induced subtrees in rooted, labeled, and ordered trees. It generates candidates using the rightmost expansion technique by collecting candidates from the data and calculates support using the rightmost occurrence lists (RMO-lists).



**Algorithm 1:** FreqT

```
1 FP = ∅
2 C = findLabels()
3 prune(C)
4 foreach c ∈ C do
5 |   add(FP,c)
6 |   expand(c)
7 return FP
```

**Algorithm 2:** Expand

**Input:** A frequent subtree f

```
1 C = generateCandidates(f)
2 prune(C)
3 foreach c ∈ C do
4 |   add(FP,c)
5 |   expand(c)
```

Figure 2.9: *FreqT* pseudo-code [2]

The pseudo-code of *FreqT* is shown in Figure 2.9. To link with what has already been said, we have in lines 2 and 3 of algorithm 1, the initialization part which

gathers all the frequent subtrees of size one (using $findLabels$ and *prune* functions).

The $findLabels$ function generates the starting set, considering all existing labels (i.e., all subtrees of size one). For each label, $findLabels$ computes the rightmost occurrence list of this label. The function *prune*, then, eliminates from this set the labels which are not frequent, to finally have the list of frequent labels and the RMO-list linked to each of these labels.

After this initialization, we have the expansion part, which corresponds to lines 4 to 6 of algorithm 1. $FreqT$ iterates on the frequent subtrees of size one found in the initialization part. For each of these frequent subtrees, $FreqT$ recursively builds larger and larger frequent subtrees (using the *expand* function).

In this *expand* function, we have the two steps of the expansion part (candidate generation and support calculation). First, calling the $generateCandidates$ function in line 1 of algorithm 2, $FreqT$ generates the candidates of a given frequent subtree (the parameter of the *expand* function) using the rightmost expansion technique detailed earlier. In a second step, with the call to *prune* function in line 2 of algorithm 2, $FreqT$ computes the support of each generated candidate using the RMO-list of the latter and at the same time eliminates the non-frequent candidates.

And finally, as was the case for frequent subtrees of size one, $FreqT$ iterates over the frequent subtrees generated, and for each of them recursively builds larger and larger frequent subtrees (using the *expand* function) until no frequent candidates are generated.

## Summary

In this chapter, we have placed the necessary theoretical foundations of the frequent tree pattern mining field. It was important because these foundations will be helpful in the next chapter, which aims to present the $FreqTals$ algorithm in some detail.

It was also essential to focus on the $FreqT$ algorithm since it is the algorithm on which $FreqTals$ is based, to which $FreqTals$ will make some modifications. We will discuss these modifications in the next chapter.

# Chapter 3

# Freqtals

The objective of $FreqTals$ is to discover regularities, also called patterns, in the source code of programs (encoded as Abstract Syntax Trees). These regularities can be very interesting for developers, especially for code comprehension, code refactoring, and fault localization tasks [2].

A new algorithm is needed because the existing pattern mining algorithms have two significant weaknesses. Firstly, these algorithms find too many patterns, which makes the analysis of results complicated. And secondly, these algorithms find patterns that are not especially suitable for discovering interesting regularities in code.

To overcome these two problems, $FreqTals$ extends the $FreqT$ algorithm by adding two new ideas.
The first idea is maximal frequent tree mining to eliminate redundant patterns. Therefore it allows reducing the number of patterns generated and thus facilitate the analysis of results, which was a problem.
The second idea is constraint-based data mining. This mining technique allows the user to define constraints on the patterns that we will generate. These constraints can be of any type; concerning the size of the pattern, concerning its content, or other things.

This chapter presents this new algorithm in detail. Since this is a new algorithm, this chapter is based mainly on the paper showing it [2]. We first present in section 1 how the two new ideas, constraint-based data mining and maximal frequent subtree mining, are added and how $FreqTals$ extends $FreqT$. The pseudo-code of $FreqTals$ is also presented and explained in this section. Then, in section 2, we adapt the $FreqTals$ implementation to discover interesting code patterns among two versions of a single system. The resolution of this problem and other problems

arising from it are discussed in this section. Finally, in section 3, we present the framework created to mine patterns in source code, allowing a more straightforward use of the pattern mining algorithm and in which *FreqTals* is only one component.

We will use the framework in the next chapter to evaluate the efficiency of *FreqTals*.

## 3.1 Implementation of FreqTals

The original *FreqT* algorithm, as presented in subsection 2.3.3, already uses constraints to reduce the size of the search space. The first constraint used, noted **C0**, is the minimum support, and its purpose is to eliminate patterns that are not frequent enough. The second constraint used, denoted **C1**, is the maximum size. These 2 constraints are implemented in the *prune* function (line 3 of algorithm 1 and line 2 of algorithm 2). In addition to these two constraints, there is also another constraint, noted **C2**, which is the minimum size. This constraint, implemented in the *add* function (line 5 of algorithm 1 and line 4 of algorithm 2), allows us to eliminate undesirable patterns that are too small. Because of their size, the latter are difficult to interpret and most of the time do not give us any important information.

It is a brief introduction to *FreqT*. The objective is now to define *FreqTals* by extending *FreqT* with other constraints and with the concept of maximal subtree mining.

### Constraints

This section aims to add constraints to the *FreqT* algorithm to reduce the search space further and eliminate more unwanted patterns. The algorithm should focus exclusively on what we are interested in.
Since we are dealing with Abstract Syntax Trees as input data, there are several types of constraints we can apply.

The first one, the **constraint on labels** of the nodes, is the most logical and the simplest to implement. We add three constraints of this type to *FreqT*:

**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.

These three constraints on labels are anti-monotonic. In concrete terms, this means that if a tree $T$ does not respect the constraint, an extension of this tree $T$ will not

respect the constraint either. Therefore, we can implement these three constraints in the *prune* function to eliminate candidates that do not respect one of these constraints.

C3 will only be helpful during the initialization phase. This constraint ensures that all subtrees of size one (labels) generated with function $findLabels$ have a label that is allowed as a root pattern. For example, in a Java code, we might want to find only patterns in methods, in which case, the value of this constraint would be $MethodDeclaration$ so that all subtrees found by $FreqT$ have as root the node label $MethodDeclaration$.

For C4 and C5, these will be used at each iteration of the expansion phase (e.g., each time a new candidate is generated). They will check that the pattern does not contain a forbidden label (C4) and that the number of siblings with the same label is under a certain threshold (C5). For example, we could specify that we are not interested in import statements and give the value $ImportDeclaration$ to C4. In the same idea, we could want to limit the number of siblings statements in a pattern to 10, and so give the value 10 to C5.

The second type of constraints, the **constraint on leaves**, allows specifying the content of the leaves to have patterns that are program-specific. The textual information of a source code is represented in the leaves of the Abstract Syntax Tree, and indeed these leaves contain characteristic elements of the program. We add one constraint of this type to $FreqT$ :

**C6**: All leaf nodes in a pattern must have a label that is included in $\Sigma_{leaf}$, where $\Sigma_{leaf}$ is the set of labels that occur in the leaves of the trees in the database.

This constraint on leaves is not anti-monotonic, so it is more challenging to implement. Fortunately, the way $FreqT$ generates candidates (rightmost expansion) allows us to eliminate some candidates. In the *prune* function, we add a condition to eliminate patterns that have a leaf, different from the rightmost node, with a label not present in $\Sigma_{leaf}$.

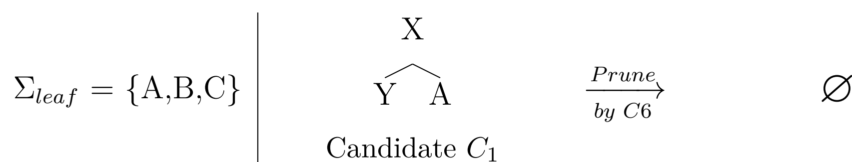



Figure 3.1: C6 example

Figure 3.1 shows us a situation where this constraint can be useful. Consider $\Sigma_{leaf}$ = {A,B,C} and a candidate $C_1$ generated by the function *generateCandidates*. The *prune* function deletes this candidate, because by the rightmost expansion, node will never be added below Y. So we will always have a pattern with a leaf node whose label is not in $\Sigma_{leaf}$. As a consequence, Y $\notin \Sigma_{leaf} \implies$ Eliminating the candidate.

The last type of constraints is the **constraint on children**. Some nodes have mandatory children and order among these children. For example, in Java, a node with label **InfixExpression** has always three mandatory children and in the same order: **leftOperand**, **operator** and **rightOperand**. It reflects language specificities.

Before introducing C7, we need to define two concepts that will be used: structural label and obligatory child labels.

- Label x is a *structural label* if it meets the two following properties :
    1. In each occurrence of x, no two children have the same label
    2. For all pairs of occurrences of x, the order of the common child labels is the same
- *obligatory child labels*, noted g(x), of label x $\in \Sigma$ (the set of all possible labels) is the set of child labels common to all occurrences of x.

The constraint on children that $FreqTals$ uses is the following one :

**C7**: Let $L$ be the structural labels in Abstract Syntax Trees of the system. For all nodes with a label $a \in L$, we require that its set of children includes nodes with all obligatory labels g(a).

This constraint on children, like the C6 constraint, is not anti-monotonic. But as with the C6 constraint, the way $FreqT$ generates candidates still eliminates some candidates.

Consider the situation presented in Figure 3.2. We have a node with structural label $A$ ($A \in L$). This node has three obligatory children ($g(A)$ = {$B$,$C$,$D$}). According to C7, these children must appear in a specific order (lexical order here). If we have candidate $C_1$ in which $A$ has as children $B$ and $D$, it is obvious, by the rightmost expansion, that the algorithm will not be able to add $C$ while respecting the predefined order between the three children. The *prune* function is modified so

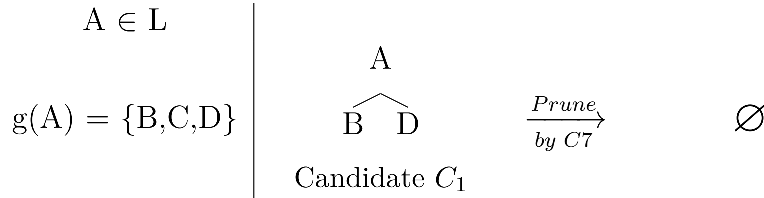



Figure 3.2: C7 example

that it prunes this kind of situation.

To know these language specificities and thus compute structural labels and obligatory child labels, *FreqTals* allows either the user to provide a grammar representing these specificities or build this grammar based on the Abstract Syntax Trees of the system. Appendix A presents in detail how *FreqTals* builds the grammar.

## Maximal

Even considering the four constraints added to *FreqT* (C4-C7), the number of patterns to manage can be large. This problem is only amplified when we decrease the value of constraint C0 (minimum support threshold). To overcome this problem, *FreqTals* mines maximal frequent subtrees. Let us consider $T_c$ the set of patterns found by *FreqT* with all the constraints defined previously. We can write the maximal frequent subtree mining problem as follows :

$$max(T_c) = \{T \in T_c \mid \nexists T' \in T_c : T \text{ is a subtree of } T'\}$$

Algorithms already exist to solve the problem of maximal frequent subtree mining. But the strategy used by the latter is problematic in the case of trees with large fanout, i.e., the number of children a node can have. In our case, Abstract Syntax Trees have large fanout, and therefore these algorithms are not suitable.

*FreqTals* proposes another strategy to solve this problem of maximal frequent subtree mining. The strategy used is divided into three steps.
The separation of the three different steps is visible in the pseudo-code of the *FreqTals* algorithm shown in algorithm 3.

- In the first step, which corresponds to line 1, we use the original *FreqT* algorithm to which we add the four defined constraints (C4-C7). The C1 constraint, the maximum size, is crucial because it is the constraint that will limit the search space size.

**Algorithm 3:** FreqTals

**Input** : Database D, Constraints C0-C7
**Output** : Maximal Frequent Patterns MP

```
1 FP = FREQT(D)
2 ROM = groupRootOccurences(FP)
3 MP = ∅
4 foreach r ∈ ROM do
5     c = root label of r
6     mineMaximalSubtrees(c,r,MP)
7 return MP
```

- In the second step, which corresponds to line 2, we group the subtrees found by root occurrences. To do this, we compute $RO = \{occ(T) \mid T \in FP\}$[1] and then $ROM = \{r \in RO \mid \nexists r' \in RO : r' \subset r\}$. ROM contains the minimal occurrence sets.

- In the third step, which corresponds to lines 4 to 6, we perform a search to find maximal patterns for each set of occurrences of $ROM$. In practice, for simplicity, we use the $FreqT$ algorithm slightly modified to have a more efficient search. It is important to note that since we are looking for maximal patterns in this third step, we do not use the C1 constraint (maximum size).

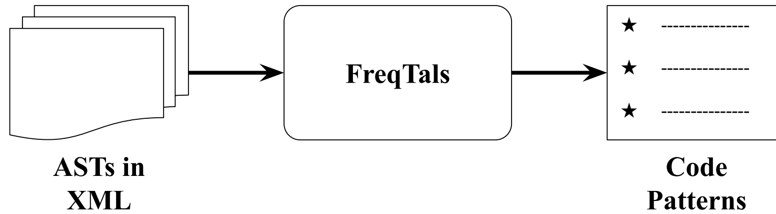



Figure 3.3: $FreqTals$ visual representation

In practice, we can represent the $FreqTals$ algorithm as in Figure 3.3; taking as input ASTs of a system (encoded in XML format) and returning as output code patterns (encoded in txt and XML format)[2].

[1] occ(T) is the occurrences set of pattern T

[2] results under txt contain the raw representations of the patterns while results under XML contain the support and the detailed representation of the patterns

The *FreqTals* algorithm presented here has been the subject of quantitative and qualitative evaluation [2]. Results showed several things:

- Compared to the original *FreqT* algorithm, they notice a decrease in execution time and the number of patterns discovered.
- The patterns found were of good quality, i.e., they highlighted interesting regularities in the source code.
- Some patterns found were larger than expected.

However, the study also highlighted that configuring the parameters of *FreqTals* is a complex task.

## 3.2 Extension to 2-class data

The *FreqTals* algorithm, as presented in the previous section, takes as input a set of ASTs corresponding to a system. The final objective of *FreqTals* being to discover software changes, it is evident that this is not possible with a single version of a system. In this section, which is primarily inspired by [13], we adapt the *FreqTals* algorithm previously developed so that we can use it in our case, where two versions of the same system are given as input.

Given the two source code of a system, we can transform them into two sets of Abstract Syntax Trees. To distinguish between the two sets, we identify them by a class label (one or two). In this way, as we can see in Figure 3.4, each tree has its id (as was already the case with the previous version of *FreqTals*) and one id linking it to its class; $t_{class_id,tree_id}$.

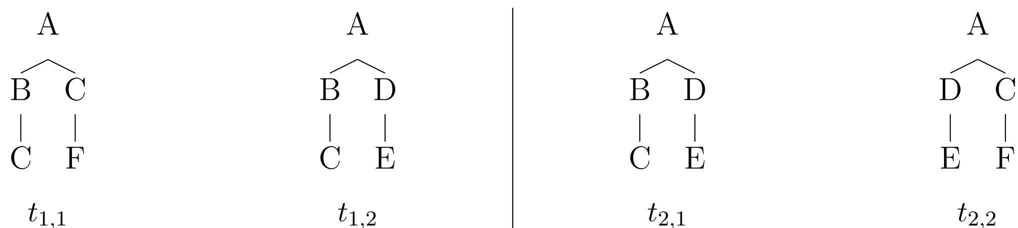



Figure 3.4: Example of 2-class dataset

Thanks to this way of identifying trees, *FreqTals* can consider the two sets of ASTs as a single set of ASTs, in which we can differentiate the trees coming from

the first set (class id = one) from those coming from the second set (class id = two). *FreqTals* no longer has to consider two sets of Abstract Syntax Trees, but only one, in which we can distinguish between trees from one set and trees from the other set.

There is another problem that we have to solve with the version presented in the previous section. Considering as input the Figure 3.4, we can derive some induced subtrees presented in Figure 3.5.

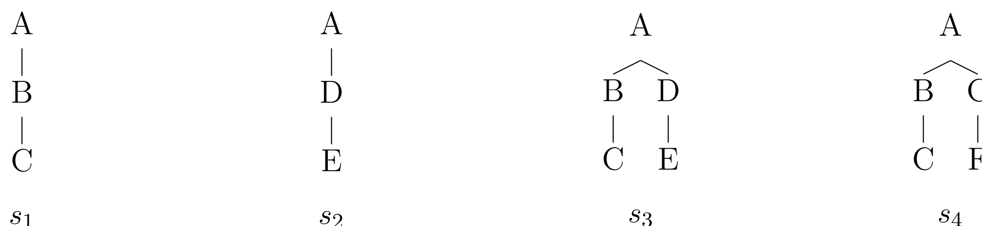



Figure 3.5: Example of induced subtrees

As we can see, $s_1$ is present two times in the first version and one time in the second. $s_2$ is present one time in the first version and two times in the second. $s_3$ is present one time in each version, and $s_4$ is only present one time in the first version.
If we calculate the support in the same way as in the previous version of *FreqTals*, it gives us respectively for $s_1$,$s_2$,$s_3$ and $s_4$ : 3,3,2,1. The problem is obvious; if we keep the same way of calculating the interestingness of patterns (only with the support value), there is no difference between $s_1$ and $s_2$.

To further highlight the problem, let's consider a pattern that has a support value $N$; it is either possible that this pattern appears in a balanced way ($N/2$ times in the first version and $N/2$ times in the second version) or that this pattern has been introduced or deleted ($N$ times in one of the two versions and zero times in the other version). Since our goal is to see how the system evolves, the introduced or deleted pattern is of great interest to us, whereas the pattern that has not moved is of no interest. We must therefore find a way to differentiate between these two cases.

The solution used by *FreqTals* is to use, in addition to the support, a measure to quantify the difference between the support in the first version and the support in the second version, such as Information Gain or Chi-Square. *FreqTals* proposes to use the chi-square value.

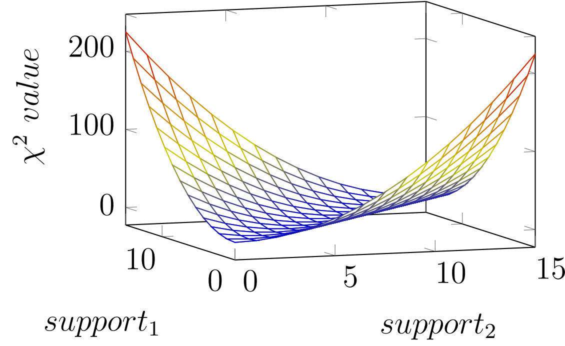



Figure 3.6: Chi-Square function

Before formally defining the constraint that $FreqTals$ uses, it is interesting to look at the chi-square function in Figure 3.6 to get an intuition of what we are looking for and what is interesting for us. What we want is a difference between version one support ($support_1$) and version two support ($support_2$). As the figure shows, the larger $|support_1 - support_2|$ is, the larger the score the chi-square function returns. The intuition would be to put a lower bound on the chi-square score to keep only patterns with a high chi-square score while keeping a high support value.

More formally now, considering a pattern $p$, ASTs set $D$ where $D_1$ and $D_2$ are the ASTs sets of the first and second version of the system and a minimum support threshold $minSup$, the support constraint can be expressed as follow:

$$support(p, D_1) + support(p, D_2) \geq minSup \tag{3.1}$$

For the chi-square value, assume the observation of pattern $p$ in the following contingency table:

| | Presence | Absence | sumRow |
|---|---|---|---|
| $D_1$ | a | b | a+b |
| $D_2$ | c | d | c+d |
| sumCol | a+c | b+d | n = a+b+c+d |

Table 3.1: contingency_table

A chi-square value of $p$ can be estimated as follow:

$$\chi^2(p, D) = \sum_{i=1}^{2} \sum_{j=1}^{2} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}$$

where $O_{i,j}$ is the observed value of column $i$ row $j$, i.e, a,b,c,d and $E_{i,j}$ is the

expected value which is calculated as follow:

$$E_{i,j} = \frac{sumCol_i \times sumRow_j}{n}$$

Given a minimum chi-square threshold $minChiSquare$, the chi-square constraint can be expressed as follow:

$$\chi^2(p, D) \geq minChiSquare \tag{3.2}$$

The objective of $FreqTals$ now is not only to look for patterns that respect support constraint, but also chi-square constraint, both expressed respectively by the Equation 3.1 and Equation 3.2.

The constraint on the chi-square value is implemented in the *add* function of the $FreqTals$ algorithm in such a way that before adding a pattern in the list of frequent patterns, we check if it does not violate the chi-square constraint.

We had a final enhancement to the version of $FreqTals$ presented in the previous section. Instead of keeping all frequent patterns in the first step of the algorithm, we take for the second step only the $k$ frequent patterns with the highest chi-square value.

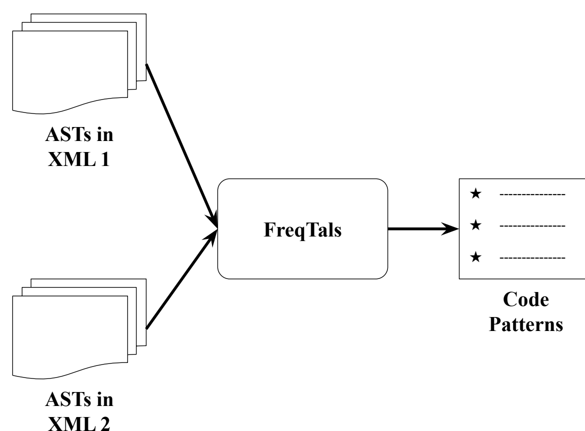



Figure 3.7: $FreqTals$ visual representation for 2-class data

Figure 3.7 provides a visual presentation of the $FreqTals$ algorithm obtained at the end of this section. The difference with the old version is mainly on the input; now $FreqTals$ can handle two versions of a system (2 sets of ASTs) as input and mines interesting patterns between these two versions.

## 3.3 Framework

The *FreqTals* algorithm developed so far is not usable yet. Since it takes Abstract Syntax Trees as input in XML format, it is not very practical for a normal user who only has source code. The same goes for the output; it is not very practical to analyze manually subtrees encoded with depth-first codification. Moreover, this algorithm, used here exclusively on Java code, is part of a larger project that aims to mine patterns in source code, independently of the language used (a language-independent framework). This section explains this framework and each of its components in more detail because it is the framework that we use for the experiment part. We mainly take our explanations on this framework from the article that presents it [14].

Figure 3.8 gives a visual presentation of this framework which has a pipeline architecture composed of 4 components: Source Code importer, Pattern Miner, Pattern Matcher, and Modernization Assistant Tool.

Figure 3.8: Framework

### Source Code Importer

The purpose of the Source Code Importer component is to transform source code (from any programming language) into a language-agnostic Abstract Syntax Tree format (AST). The format is an XML file where each AST node is an XML element whose start and end tag identifies the AST node type. The content of this tag is about the child relationships that this node has. These children are defined in the same way and can also contain their child relationships, as defined in the AST.

```
if(...){               <IfStatement>
    ...                    <expression node="Expression"/>
}else{     →               <thenStatement optional="true" node="Statement"/>
    ...                    <elseStatement optional="true" node="Statement"/>
}                      </IfStatement>
```

Figure 3.9: Source Code Importer example

Figure 3.9 shows a concrete example of correspondence between the source code and the XML representation. We have taken a relatively simple example, the If statement in Java. As we can see, the XML representation shows the existence of the three children of an If statement; the expression which is the tested condition, the thenStatement, which is the statement executed if the condition is met and the elseStatement, which is the statement executed if the condition is not met.

It is important to note that this is the only part of the framework that is language-dependent. Once we have a working importer for a language, the next steps of the framework are language-independent.

In our case, we use a working source code importer for Java code [15].

## Pattern Miner

The Pattern Miner component is the one we mainly talked about in the previous sections; it is the component that, based on two sets of Abstract Syntax Trees of a system, gives us patterns that come up often and are interesting.

The benefit of working with a framework is that we could use any algorithm for this component, as long as it respects the input and output format. In our case, the algorithm used is $FreqTals$, but we could use other frequent tree mining algorithms or even graph mining algorithms.

Figure 3.10 shows what kind of data we can get as output from the Pattern Miner component (in this case with the $FreqTals$ algorithm). The first line is the result obtained in the txt file, a raw representation of the pattern, encoded with the depth-first codification (presented in section 2.1). The second line is the result obtained in the XML file; a more detailed representation of the pattern, with its id, its support (in the first and second version of the system), its size, and its chi-square score.

```
(Block(statements(ExpressionStatement(expression(MethodInvocation(
    expression(SimpleName(identifier(*log))))(name(SimpleName(identifier(*
    log))))(arguments(QualifiedName(qualifier(SimpleName(identifier(*Level
    ))))(name(SimpleName(identifier(*SEVERE)))))(SimpleName(identifier(*e)
    ))))))))
```

```xml
<subtree id="30" support="0-22" score="25.220458553791886" size="26">
    <Block>
      <statements>
        <__directives><match-sequence/></__directives>
        <ExpressionStatement>
          <expression>
            <MethodInvocation>
              <__directives><match-sequence/></__directives>
              <expression>
                <SimpleName>
                  <identifier>log</identifier>
                </SimpleName>
              </expression>
              <name>
                <SimpleName>
                  <identifier>log</identifier>
                </SimpleName>
              </name>
              <arguments>
                <__directives><match-sequence/></__directives>
                <QualifiedName>
                  <__directives><match-sequence/></__directives>
                  <qualifier>
                    <SimpleName>
                      <identifier>Level</identifier>
                    </SimpleName>
                  </qualifier>
                  <name>
                    <SimpleName>
                      <identifier>SEVERE</identifier>
                    </SimpleName>
                  </name>
                </QualifiedName>
                <SimpleName>
                  <identifier>e</identifier>
                </SimpleName>
              </arguments>
            </MethodInvocation>
          </expression>
        </ExpressionStatement>
      </statements>
    </Block>
  </subtree>
```

Figure 3.10: Pattern Miner Output example

Figure 3.10 shows us the output for only one pattern. When the Pattern Miner finds a few tens of patterns, it quickly becomes impossible to analyze them in this format manually.

## Pattern Matcher

The Pattern Matcher component aims to find all subtrees (in the starting ASTs) matching patterns found by the Pattern Miner component. There are some features that we can use with this component to make it more efficient, like applying postprocessing on the patterns found by the Pattern Miner component to make them more general or using "*directives*" on some parts of the code to affect the semantics of subtrees found by the Pattern Matcher component.

In the *FreqTals* output example, Figure 3.10, one type of directive is present; < ___*directives*> <*match-sequence*> < /___*directives*> at line 4,8,20 and 22. This directive imposes that searched subtrees have children in the same order than the pattern, but other children can be present.

```
<match PatternID="30" FullName="../jgraphx/src/com/mxgraph/util/
    mxUtils.java">
  <node ID="23610" Name="?_root" />
  <node ID="26056" Name="?_visited" />
  <node ID="26423" Name="?_visited" />
  <node ID="25520" Name="?_visited" />
  <node ID="24786" Name="?_visited" />
  <node ID="26064" Name="?_visited" />
  <node ID="26053" Name="?_visited" />
  <node ID="26050" Name="?_visited" />
  <node ID="26420" Name="?_visited" />
</match>
```

Figure 3.11: Pattern Matcher Output Example

This component produces, as output, for each version of the system, an XML file. This file contains a list of matches between a pattern found by the Pattern Miner component and a subtree in the starting ASTs. Figure 3.11 shows us an example of a match, where the id of the pattern is specified (30, indeed it's the same pattern as in the example of Pattern Miner Output), the name of the file in which the match is made and a list of nodes for which the match is made (and the root of

the match is indicated).

In our case, we use the ForestMatcher [16] as Pattern Matcher component.

## Modernization Assistant Tool

The last component is the Modernization Assistant Tool. It provides a much more user-friendly Graphic User Interface (GUI) for analyzing results. We can split the GUI into three parts.

The first part, presented in Figure 3.12, allows visualizing the list of patterns found (with some information such as their size, their support in the two versions, and their chi-square score). By selecting a pattern, for example, the pattern with id 30 on the picture, we can visualize the files, from the first or second version, in which this pattern appears, as well as the number of occurrences in each file.

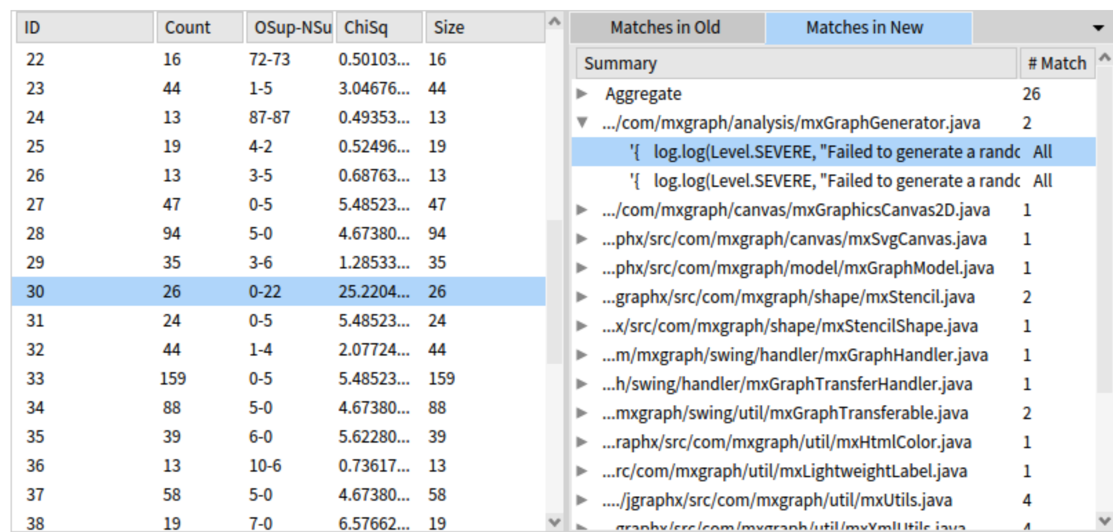



Figure 3.12: GUI of pattern selection

The second part, shown in Figure 3.13, allows visualizing a specific occurrence of a pattern in a file. In the picture, we can see a particular occurrence of the pattern with id 30 in its context.

The third part, presented in Figure 3.14, allows visualizing the Abstract Syntax Tree representation of a selected pattern. In the picture, it is the AST of the pattern with id 30.

```
{
   log.log(Level.SEVERE, "Failed to generate a random tree graph", e);
}
```

Figure 3.13: GUI of code display

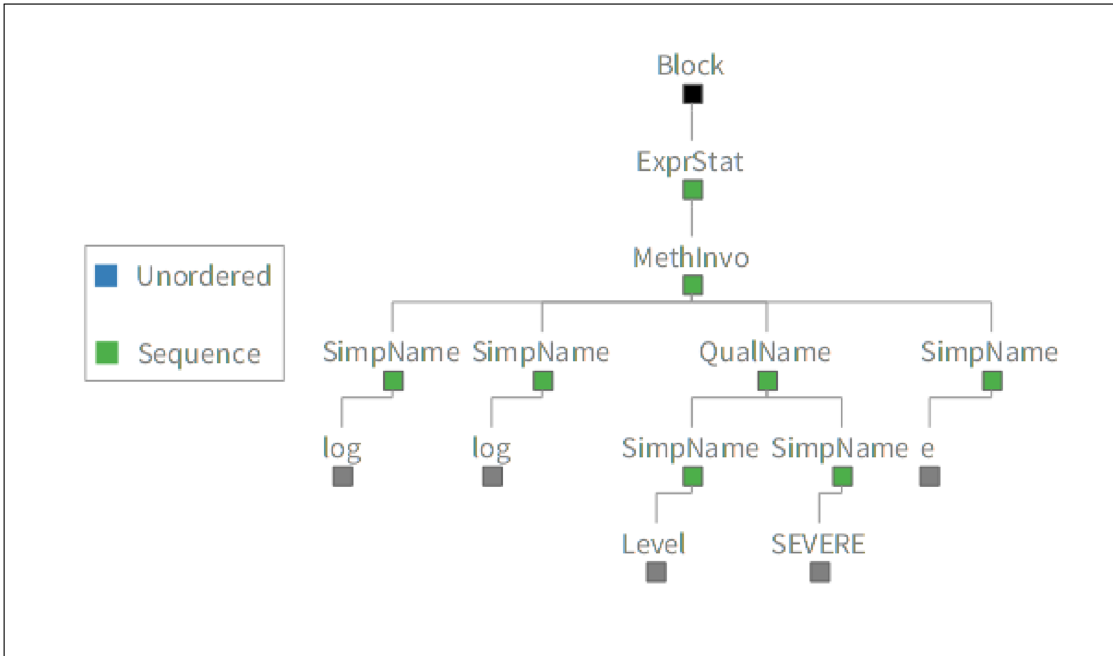



Figure 3.14: GUI of Abstract Syntax Tree display

The modular way in which the framework works is an advantage. Indeed, if we want the framework to work for a new language, we would add a new language-specific Source Code Importer. We could also change some minor details to look for patterns adapted to the language, but the overall idea and the general framework remain the same. In the same idea as a new language, we could change the algorithm used by the Pattern Miner without having to change anything else.

## Summary

The presentation of this framework concludes the first part of this work; we have presented the current situation in a theoretical way. In the second part of this work, we will use this framework to perform experiments and, in particular, to evaluate the efficiency and the correctness of the algorithm used by the Pattern Miner component in our case, $FreqTals$.

# Chapter 4

# Evaluation of FreqTals

In this chapter, we aim at using the previously defined framework on a practical case. To achieve this, we first need to determine the experimental setup; the configuration of $FreqTals$ that we use; section 1 has this role. Then we have to present the software used as a test case; this is the role of section 2. Finally, once we have the configuration and the software, we must run the algorithm and analyze the results. In section 3, we analyzed results, and very relevant analyses on some results are presented in-depth.

## 4.1 Experimental Setup

In this section, we present the configuration used for the experiments. It is an important step since, as specified at the end of section 3.1, configuring the parameters of $FreqTals$ is a complex task.

The main configuration file is fully displayed in Appendix B. We present here only some parameters that we consider essential.

- **Minimum chi-square value**
  This parameter allows us to specify the minimum chi-square value that patterns should have. In combination with the support (whose value is 5, given as an argument in the command line), it allows to define the chi-square and support constraints.

  ```
  # minimum discriminative score
  minDSScore = 0.4
  ```

  In our case, we decided to set the minimum chi-square value to 0.4.

- **Maximum number of leaves**
  We use this parameter to implement the **C1** constraint regarding the maximum size of patterns. The size is defined here in terms of the number of leaves.

  ```
  # max number of leaf
  maxLeaf = 3
  ```

  In our case, we limit the search in the first step of *FreqTals* to patterns with three leaves or less.

- **Minimum number of leaves**
  This parameter is used to characterize the constraint **C2**; the minimum size of patterns. The minimum pattern size is implemented in two ways. This parameter defines it according to the minimum number of leaves a pattern must have.

  ```
  # min number of leaf
  minLeaf = 1
  ```

  In our case, patterns with less than one leaf are not added to the result.

- **Minimum number of node**
  The second way to implement the **C2** constraint on the minimum size of patterns is to define it in terms of the number of nodes in a pattern. It is the purpose of this parameter.

  ```
  # min number of node
  minNode = 10
  ```

  In our case, we consider that a pattern with less than ten nodes should not be regarded as a valid result.

- **Number of patterns in the first step**
  This parameter is used to limit the number of patterns obtained at the end of the first step of *FreqTals*.

  ```
  # number of highest score patterns at the end
  numPatterns = 1000
  ```

  In our case, we consider the 1000 patterns with the largest chi-square value.

- **Way of calculating the support**
  As seen in subsection 2.3.1, there are two ways to compute the support. So we have to choose which way *FreqTals* calculates the support.

```
# count support: true: count support based on number of occurrences; false:
    count support based on number of files
weighted = true
```

  In our case, we have arbitrarily chosen to compute the support based on the number of occurrences rather than the number of files.

- **Root label file**
  To implement the constraint **C3**, we specify the path to a file containing labels that can be roots of patterns.

```
# file contains a list of root labels
rootLabelFile = conf/listRootLabel.txt
```

  In our case, the file *conf/listRootLabel.txt* content is shown in Figure 4.1. It defines the *MethodDeclaration* and *Block* labels as the only ones that can be patterns roots.

```
#list of root labels (AST Nodes) which should be the root labels of subtrees
#Each line corresponds to a AST node. Lines begin with # are comments.
#TypeDeclaration
MethodDeclaration
Block
```

Figure 4.1: listRootLabel.txt content

- **White label file**
  To implement the constraint **C4**, instead of specifying labels that are forbidden, it seems better and easier to select labels that we want. Indeed, if we do not know all existing labels in the ASTs, but we know what we want, it is much easier. To do this, we specify the path to a file containing the labels we want.

```
# file contains a list of label that only allow in patterns
whiteLabelFile = conf/listWhiteLabel.txt
```

In our case, the file *conf/listWhiteLabel.txt* content is shown in Figure 4.2. It defines a list of nodes and the children of those nodes we can expand. Labels that we have not specified will never be considered in the expansion process. For example, the *parameters* child of *MethodDeclaration* node is forbidden, in the sense that it will never be considered in the *FreqTals* expansion process since it is not present in the list of allowed children for the *MethodDeclaration* node (*name* and *body* are the only authorized children).

```
#This file contains a list of nodes and their children which are allowed to
    expand in the mining process
#Each line corresponds to a node and its children.

#CompilationUnit [ordered, 3, package¥true, imports¥true, types¥true]
CompilationUnit types

#TypeDeclaration [unordered, 8, javadoc¥false, modifiers¥true,
    interface¥true, name¥true, typeParameters¥false,
    superclassType¥false, superInterfaceTypes¥false,
    bodyDeclarations¥true]
TypeDeclaration bodyDeclarations

#bodyDeclarations [unordered, 1..*, ...]
bodyDeclarations MethodDeclaration

#MethodDeclaration [unordered, 12, javadoc¥false, modifiers¥true,
    constructor¥true, typeParameters¥false, returnType2¥false,
    name¥true, receiverType¥false, receiverQualifier¥false,
    parameters¥false, extraDimensions2¥false,
    thrownExceptionTypes¥false, body¥false]
MethodDeclaration name body

#SimpleName [unordered, 2, identifier¥true, var¥true]
SimpleName identifier
```

Figure 4.2: listWhiteLabel.txt content

We still define other parameters in the main configuration file (*config.properties*). Since we had to choose to present some parameters and not others, they were not shown here by arbitrary choice.

## 4.2 Dataset

The software we decided to consider for the experiments is the graph drawing open-source software jgraphx [17].
With more than 150 modifications, this software seems to be an excellent system to test our algorithm.

| Versions | Start date | End date |
|---|---|---|
| 1.0 - 2.0 | 31 Jul 2012 | 23 May 2013 |
| 2.0 - 3.0 | 23 May 2013 | 29 Aug 2014 |
| 3.0 - 4.0 | 29 Aug 2014 | 11 Mar 2019 |
| 4.0 - 4.2 | 11 Mar 2019 | 19 Jun 2020 |

Table 4.1: Dataset

We have cut out the life of this software (from the first version to the last version) into four parts. The four intervals are shown in Table 4.1 with the dates to which they correspond. On each interval, which is defined by a start version and an end version, we ran the algorithm considering the start version as version one of the system and the end version as version two of the system. We then analyzed in detail each of these four results and discussed these results in the next section.

## 4.3 Results

All the results of the 4 experiments, in a raw way, can be found on the repository [18]

| Experiment ID | Versions | # maximal patterns found | time second step (second) |
|---|---|---|---|
| 1 | 1.0 - 2.0 | 59 | 600 (timeout) |
| 2 | 2.0 - 3.0 | 8 | 0 |
| 3 | 3.0 - 4.0 | 53 | 600 (timeout) |
| 4 | 4.0 - 4.2 | 1 | 2 |

Table 4.2: Summary of results

Before presenting the results in detail from a qualitative point of view, it is interesting to have an overview from a quantitative point of view on the four experiments and some global indices such as the number of maximal patterns found or the time taken by the algorithm for the second step of $FreqTals$ (the search for maximal patterns). These indices are presented in Table 4.2, and for simplicity, we assign

an identifier to each experiment.

As we can see, experiments one and three had to be stopped because they reached the fixed timeout of 10 minutes. Despite this, these experiments found about 50 maximal patterns, which gives us material to analyze. On the other hand, experiments two and four were very fast because they found only eight and one maximal patterns, which is poor and not very useful for the analysis.

We focus in this section on the results of experiment three because, after a thorough analysis, we noticed that the results of experiment one were, for the most part, related to the creation and deletion of files (which is typical in the first versions of software).
From a qualitative point of view, experiment three gives us a lot of information. We present below three significant analyses; error management, encapsulation, and switch statement.

### Error management

There are two patterns with a high chi-square score that stand out from the 53 patterns. These are patterns 4 and 30, which are represented in a non-exhaustive way in Figure 4.3. By looking at their occurrences, these two patterns have been added to version 4.0 of the system since they do not appear in version 3.0. By doing a little more analysis, these two patterns reflect the introduction of a logger[1] in the system; this hypothesis was confirmed by manual analysis of the two versions. Looking further, asking why these patterns were introduced, what they replaced, we realize that these two patterns replaced several things.

The first thing that these patterns replace is a comment as Figure 4.4 shows. In version 3.0 of the system, the developers left some catch clauses empty, with a simple comment. Although this is an awful programming practice, these patterns correct this problem.

The second thing that these patterns replace is call to *System.out.println*(*e*) as Figure 4.5 shows. This call, in catch clauses, will always be more appropriate than a comment, but it is not the best way to handle errors. Replacing this call by these patterns is, therefore, an excellent programming practice.

The third thing that these patterns replace is call to *System.err.println*(*e*) as

[1] cfr `https://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html`

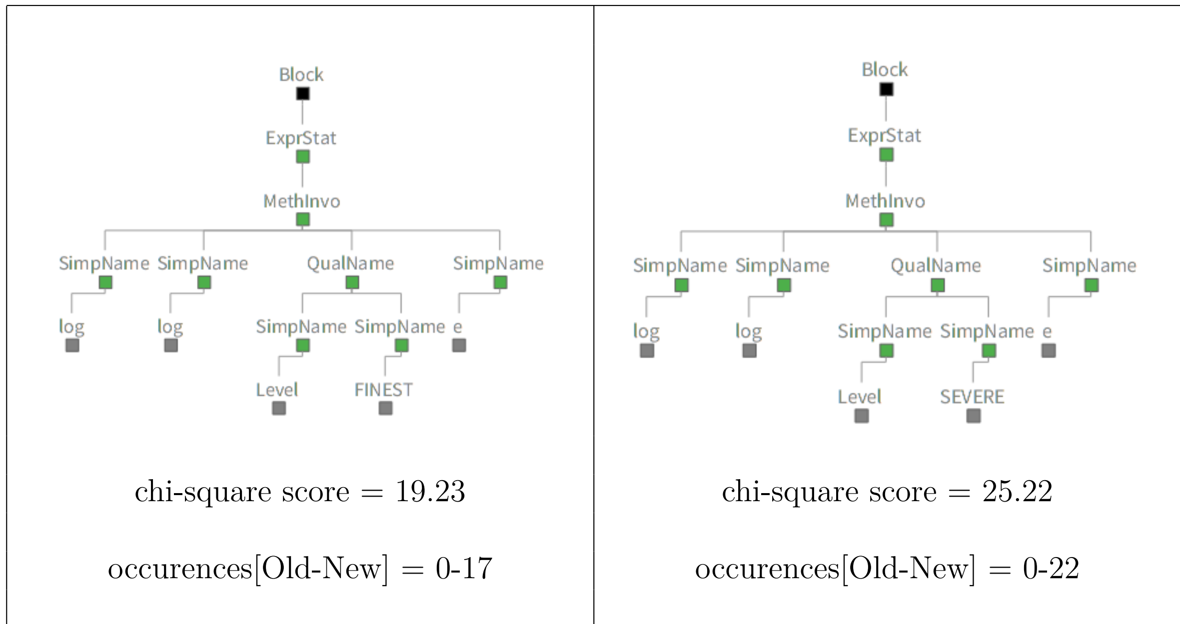



Figure 4.3: Patterns 4 and 30

```
try{
    String tmp = createDataUrl(src);
    if(tmp != null){
        src = tmp;
    }
}catch(IOException e){
    //ignore
}
```

Version 3.0

$\Longrightarrow$

```
try{
    String tmp = createDataUrl(src);
    if(tmp != null){
        src = tmp;
    }
}
catch(IOException e){
    log.log(Level.SEVERE, "Failed to create image
        data URL", e);
}
```

Version 4.0

Figure 4.4: comment replaced

```
try{
    oneSpanningTree(aGraph, true,
        true);
}
catch(StructuralException e){
    System.out.println(e);
}
```

Version 3.0

$\Longrightarrow$

```
try{
    oneSpanningTree(aGraph, true, true);
}
catch(StructuralException e){
    log.log(Level.SEVERE, "Failed to generate a
        random tree graph", e);
}
```

Version 4.0

Figure 4.5: $System.out.println(e)$ replaced

Figure 4.6 shows. This call has its place in catch clauses and is appropriate, but overall, the use of a logger standardizes the way errors are handled, which is an excellent programming practice.

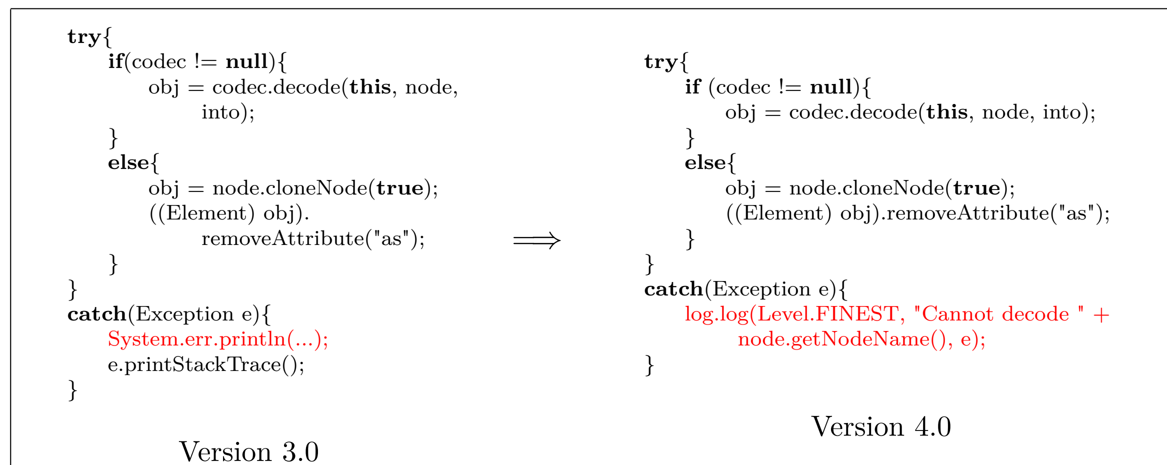



Figure 4.6: *System.err.println*(...) replaced

The last thing these patterns replace is call to *e.printStackTrace*() as Figure 4.7 shows. As for the previous element, replacing it by a logger allows to standardize the code.

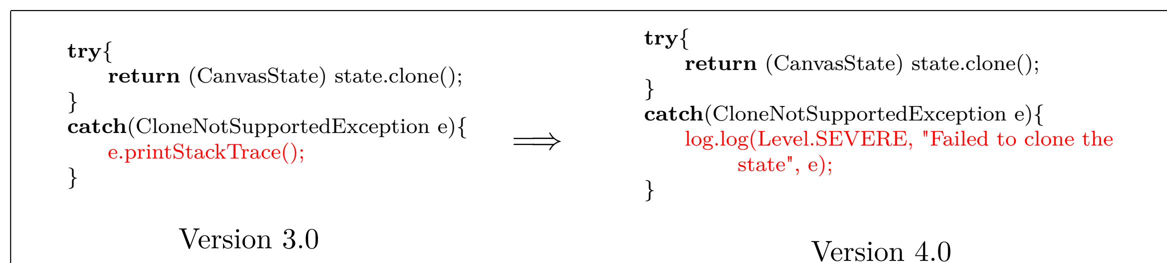


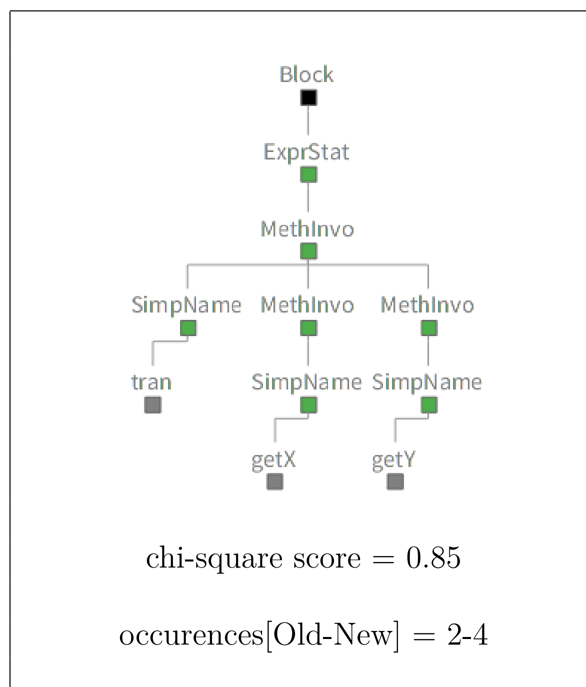

Figure 4.7: *e.printStackTrace*() replaced

Moreover, by pushing the analysis a little further, we found that the last three elements replaced by using a logger were also patterns found by the algorithm (pattern 18, 38, and 47). These are patterns that were removed between version 3.0 of the system and version 4.0 of the system since they no longer occur in version 4.0 of the system.

The two patterns related to using a logger have allowed us to see that between version 3.0 and version 4.0 of the system, the developers have standardized their way of handling errors in catch clauses. A single way has replaced four different methods.

## Encapsulation

There is a pattern whose number of occurrences and chi-square score are not very high, and yet, from a qualitative point of view, this pattern is fascinating. It is pattern 49, which is represented in Figure 4.8. This pattern already existed in the old version of the system, but its use has been increased.




chi-square score = 0.85

occurences[Old-New] = 2-4

Figure 4.8: Pattern 49

At first sight, this pattern is not exceptional; it simply uses getters in a function call. But let's take a closer look at the places in the source code where it has been added. We notice, as shown in Figure 4.9, that the use of this pattern replaces direct access to class variables, which considerably reduces encapsulation, one of the fundamental principles of object-oriented programming.

Therefore, although this pattern is not one of the best results from a quantitative

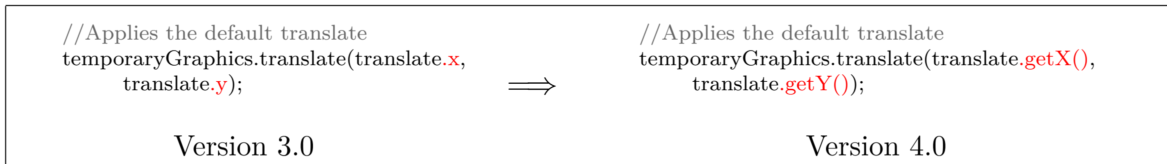



Figure 4.9: encapsulation introduced

point of view (number of occurrences and chi-square score), it shows that good programming practices are used, introduced, and democratized.

## Switch statements

There is one last pattern which, as for the previous one presented, is not exceptional in terms of results (number of occurrences and chi-square score), but from which we can draw an interesting qualitative analysis. It is pattern 1, represented in Figure 4.10. This pattern was present in the old version of the system but is no longer present in the new version of the system (the file has been deleted).

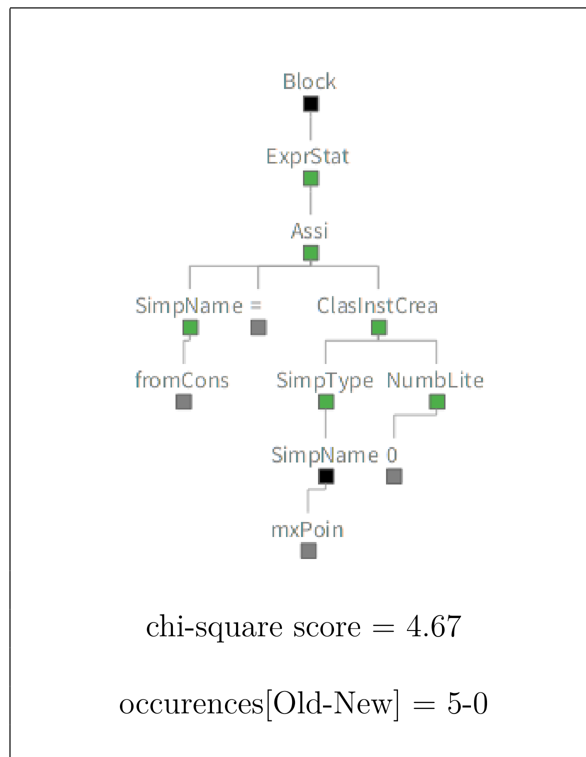



Figure 4.10: Pattern 1

What is interesting about this pattern is that all its occurrences are found in a single file. A pattern that occurs $n$ times in a single and unique file is intriguing and requires further analysis. By doing this in-depth analysis, as we can see in Figure 4.11, we found that this pattern reflects the use of switch statements (or, similarly, nested if statements).

Figure 4.11: switch statement deleted

Although this is not an error as such, it is a bad smell in programming[2]. In this particular case, we do not know how this piece of code was replaced and how the logic of this switch statement was implemented in the new system since the file was removed, but in any case, it is a good practice to not have this bad smell in the latest version of the system.

[2] https://sourcemaking.com/refactoring/smells/switch-statements

## Conclusion

In conclusion, besides these excellent results and thorough and good quality analyses, there are unfortunately some negative points (weaknesses) and problems that we noticed during the experiments. These problems and the solutions brought to solve them are the subject of the next chapter.

# Chapter 5

# Weaknesses and Enhancement

During the experiments performed in the previous chapter, we noticed two problems in the $FreqTals$ algorithm. These two errors had to be corrected, although their impact on the results is small since we had not detected them before.

The first error is related to the way we calculate the chi-square score. It is explained in detail in section 1. A solution to this problem is also provided.

The second error is related to the implementation of the White List Label (denoted as $WLL$). The latter is a much larger problem and is explained and solved in section 2.

## 5.1 Chi-square score

The first problem concerns the calculation of the chi-square score and, more particularly, the case where the **weighted** parameter of the configuration file is set to true. Therefore, $FreqTals$ calculates the support as the total number of occurrences of a pattern (and not as the number of files where the pattern appears).

As already said in subsection 2.3.1, this way of calculating the support allows to have a support value larger than the number of files in a given version of the system. For example, consider a pattern $P$ and the files collection $D_1$ for version one of the system and $D_2$ for version two of the system.

$$\begin{aligned} support(P, D_1) &= 3 \\ support(P, D_2) &= 10 \\ |D_1| = |D_2| &= 5 \end{aligned} \tag{5.1}$$

Equation 5.1 describes a possible situation. Considering the code of Figure 5.1 to

```
//return chiSquare value of a pattern in two classes
public static double chiSquare(Projected projected, int sizeClass1, int sizeClass2,
     boolean weighted){
    int[]ac = get2ClassSupport(projected, weighted);

    int a = ac[0]; //occurrences in the first class data
    int c = ac[1]; //occurrences in the second class data

    double yaminxb = sizeClass2 * a − sizeClass1 * c;
    double one = yaminxb / ((a+c) * (sizeClass1 + sizeClass2 − a − c));
    double two = yaminxb / (sizeClass1 * sizeClass2);

    return one * two * (sizeClass1 + sizeClass2);
}
```

Figure 5.1: Chi-square computation

calculate the chi-square score, this situation returns a negative value as chi-square score, which is not normal since the chi-square value should always be positive. After some mathematical development, the chi-square computation returns a negative value in all situations where we have the following properties respected.

$$
\begin{gathered}
(|D_2| \times \mathit{support}(P, D_1)) < (|D_1| \times \mathit{support}(P, D_2)) \\
\wedge \\
|D_2| + |D_1| < \mathit{support}(P, D_1) + \mathit{support}(P, D_2)
\end{gathered}
$$

To avoid this kind of situation and therefore negative values for the chi-square score, we must violate one of the properties. The easiest property to break is the second one. Indeed, using the other way of computing the support, we have an upper bound on the support values. Therefore, as said in subsection 2.3.1, by using this way, we have the following properties on the support values.

$$
\begin{aligned}
&\mathit{support}(P, D_1) \leq |D_1| \wedge \mathit{support}(P, D_2) \leq |D_2| \\
&\quad \implies |D_2| + |D_1| \geq \mathit{support}(P, D_1) + \mathit{support}(P, D_2)
\end{aligned}
$$

In practice, to correct this error, we impose the way of calculating the support of patterns instead of leaving the choice to the user, as was the case previously. Another solution could be to use other functions than chi-square.

Figure 5.2: Example of mismatching support values

## 5.2 Implementation of White List Label

The second problem is related to the way $FreqTals$ implements the $WhiteListLabel$[1] ($WLL$). Indeed, for some patterns in the results, the support computed by the Pattern Miner ($FreqTals$) is not the same as the support computed by the Pattern Matcher. For example, in Figure 5.2, we can see that pattern 30 has a support of 22 with the Pattern Miner and of 26 with the Pattern Matcher.

The error comes from the Pattern Miner side. To implement the $WLL$, the latter performs a preprocessing of Abstract Syntax Trees before representing them internally. During this preprocessing, $FreqTals$ keeps, for a node $B$, only the children that $WLL$ allows this node $B$ to have. Unfortunately, the way this is implemented deletes the unallowed children of a node and the subtrees under each of these children. As a result, the input data is not the same as the data used internally, and therefore, the support may differ since the internal data is incomplete. Figure 5.3 shows a visual representation of this problem.

[1] Figure 4.2

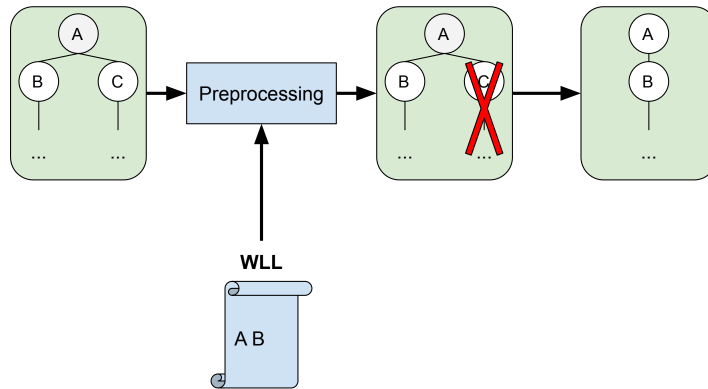



Figure 5.3: Preprocessing global overview

To solve this problem, we first removed this implementation during preprocessing so that the internal data is the same as the input data for consistency.

In a second step, to respect the $WLL$ rules, we decided to add a filter during candidate generation. Indeed, when we add a node to a frequent pattern (= generation of a new candidate), it is easy to check a property concerning this node and its parent. Figure 5.4 shows the global path to generate a candidate from a frequent pattern, while algorithm 4 shows the pseudo-code added to $FreqTals$.

**Algorithm 4:** WLL implementation

```
   Input: Node Parent p, Node Child c, White List Label wll
1  if p ∈ wll then
2      if c ∈ wll[p] then
3          acceptCandidate;
4      else
5          rejectCandidate;
6  else
7      acceptCandidate;
```

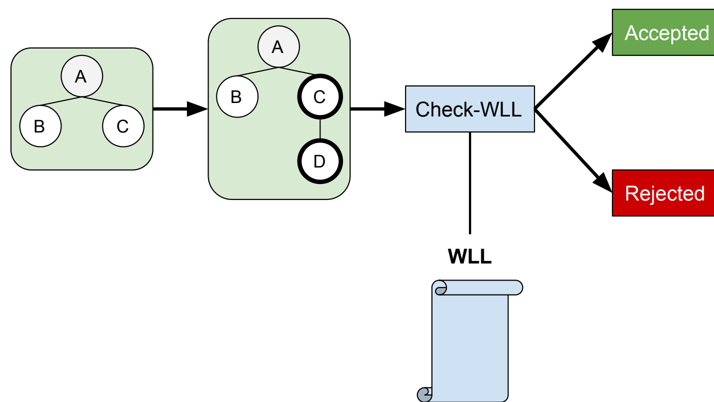



Figure 5.4: $WLL$ in candidate generation

## Summary

Now that we have solved the two problems that the $FreqTals$ algorithm had, we must again perform the experiments under the same conditions to compare results. Given the two changes made, we expect two consequences.

First, the execution time should increase since we do not delete data anymore, as was the case with the old $WLL$ implementation.

Second, regarding the number of patterns, it's difficult to say since, on the one hand, the way of calculating the support has a neutral or negative effect on the support values. But on the other hand, since we consider all the data now, we may find new patterns.

The next chapter will aim at confirming or not these two hypotheses.

# Chapter 6

# Validation of new implementation

This chapter aims to show the consequence of changes made in the previous chapter to correct the few errors of the *FreqTals* algorithm. Because of corrections made, we first ask ourselves whether results will still be as interesting as before. Secondly, we want to know if hypotheses about the execution time and the number of patterns found will be confirmed. If this is the case, and consequently the execution time grows, and the number of patterns grows, what will be the consequences from a quantitative point of view; will the execution time be a problem. If the number of patterns is too high, how will we manage the manual analyses?

The experiments with the new version of the algorithm were performed under the same experimental conditions as those presented in section 4.1. The only difference is that to see the impact on the execution time, we removed the timeout so that the experiments can end. We chose to compare the old algorithm results and the new one only on the interval that interested us, i.e., between version 3.0 and 4.0 of *jgraphx* software.

This chapter is structured as follows; we separate the qualitative analysis from the quantitative analysis. The latter is the subject of section 1. We compare the new algorithm results in terms of the number of patterns and execution time with the results of the old algorithm. In section 2, we focus more on the qualitative analysis, and particularly on what happened to the three cases that we presented in section 4.3, which showed the quality of results of the old algorithm. And finally, we conclude with a short critical summary of changes made and the new algorithm.

| | Old algorithm | New algorithm |
|---|---|---|
| Number of patterns | 53 | 45 |
| Execution time | 2 hr 14 min 5 sec | 2 hr 45 min 14 sec |

Table 6.1: Quantitative comparison

## 6.1 Quantitative analysis comparison

Table 6.1 represents the overall indices from a quantitative perspective for each of the algorithms; the old one and the new one.

As we can see, we go from 53 patterns found to 45, which is a slight decrease. We had doubted the impact of corrections on the number of patterns, and this slight decrease shows us that the intuition was correct; the difference is not that important. It is, in fact, the consequence of the way we calculate the support. The patterns obtained with the new algorithm have globally much lower support than before. Combined with the fact that we did not change the thresholds of the constraints, the patterns that were close to one of the thresholds are pruned with the new implementation.

Regarding execution time, our intuition was correct; the new algorithm is much slower than the old one, as shown in Table 6.1. We should have expected this since the new version of the algorithm considers all the input data. In contrast, the old version made a mistake by eliminating some significant parts of the input data. It is a logical consequence but not a negative point since it was a mistake to destroy some input data.

## 6.2 Qualitative analysis comparison

| | Old algorithm | | New algorithm | |
|---|---|---|---|---|
| | chi-square score | occurrences | chi-square score | occurrences |
| pattern 4 | 19.23 | 0-17 | 5.48 | 0-5 |
| pattern 30 | 25.22 | 0-22 | 18.05 | 0-16 |
| pattern 49 | 0.85 | 2-4 | 0.85 | 2-4 |
| pattern 1 | 4.67 | 5-0 | 0.92 | 1-0 |

Table 6.2: Evolution of interesting patterns

From a qualitative point of view, all patterns involved in the exciting analyses

done with the old algorithm in chapter 4 are still present in the results. Table 6.2 shows changes in occurrences and chi-square score for each of these patterns. As we predicted, the way support is calculated had a neutral (pattern 49) or negative (pattern 4, 30, 1) impact on the chi-square score.

There are new patterns found and others that are no longer seen, but no very interesting patterns are involved from a qualitative point of view. The most interesting patterns are those involved in the three cases analyzed in chapter 4. It is a good thing they are still around since changes have corrected the algorithm without having many consequences on the quality of results.

## Summary

Overall, the new implementation of *FreqTals* is a success. First, we fixed two subtle errors that we did not detect in the past. And secondly, these corrections did not have a significant impact on the quality of the patterns. The main consequence concerns the execution time of the new implementation. We can set a timeout value in the configuration file to overcome this problem, as was already the case with the old implementation of *FreqTals* since its execution time was already high.

# Chapter 7

# Discussion

This last chapter presents some possible improvements and concludes this work by noting the essential elements.

## 7.1 Future Work

Although the approach presented gives good quality results, there are nevertheless some areas for improvement.

- Think of other functions than chi-square to compute the interestingness of patterns. For example, it could be interesting to use the Information Gain function and compare the results.
- Experiment with much more complex, diversified and specific configuration files (e.g. *rootLabel* and *whiteListLabel* files). We have limited ourselves to very standard configurations. It could be useful to configure *FreqTals* to mine something very specific (for example, patterns rooted in *IfStatement*).
- Experiment with the algorithm on other cases, more varied, older, or even larger. The experimental analyses allow us to continuously improve the approach and make it fit the user's needs.
- Do not limit ourselves to maximal patterns. During the experiments, we noticed that non-maximal patterns could be interesting, although not maximal. It would be interesting to be less restrictive and accept non-maximal patterns, having a very good chi-square score.

## 7.2 Conclusion

In this work, we tried to find out if it was possible to discover regularities in the modifications made on software. In particular, how the source code of a system evolves.

Frequent Tree Pattern Mining is an approach able to analyze this kind of data and help us discover regularities.

In the first part of this work, we presented this field and the two algorithms that we use; *FreqT* and $FreqTals$. *FreqT* is an algorithm that mines frequent patterns in trees. We have extended this algorithm to create $FreqTals$, an algorithm capable of mining frequent and interesting patterns in two-class data. For ease of use, we use a framework in which $FreqTals$ is only a component.

In the second part of this work, we used the previously defined framework to validate and verify the $FreqTals$ algorithm.

First, we validate the $FreqTals$ algorithm. Indeed, using the framework, we show that excellent results can be obtained. Regularities in the code have been found, and these can be very useful for developers to understand how a code evolves, which programming practices are good to use, or which ones to avoid.

And secondly, we verify the $FreqTals$ algorithm based on the first experimental analyses. We found two subtle errors that we had not detected because their impact on the results was very limited. We corrected these two errors, and then we validated the new $FreqTals$ implementation again. As already mentioned, since the mistakes were minimal, the impact on the results was not felt, and therefore, this new version of the $FreqTals$ algorithm also gave significant results.

This work, given the results, allows us to say that it is possible to discover regularities in the modifications made to the software and that patterns in the evolution of a source code can be found and used for the future.

# Appendix A

# Grammar Building

The content of this section is based exclusively on [19].

**Algorithm 5:** Grammar extractor from ASTs

```
   Input  : Collection of Abstract Syntax Trees ASTs
   Output : Grammar
1  grammar = ∅
2  foreach ast ∈ ASTs do
3  |  foreach node ∈ ast do
4  |  |  if node ∉ grammar then
5  |  |  |  addNewNode(node,grammar)
6  |  |  else
7  |  |  |  updateNode(node,grammar)
8  |  |  end
9  |  end
10 end
11 return grammar
```

To create grammar based on Abstract Syntax Trees, we use the pseudo-code shown in algorithm 5. This one browses the ASTs, and for each node, adds it to the grammar if it was not there (function $addNewNode(node, grammar)$) or updates the information about this node in the grammar if it was already there ($updateNode(node, grammar)$). To be concrete, we will use as an example the collection of Abstract Syntax Trees of Figure A.1.

The first thing to analyze when passing over a node is the order between its children. We consider the two following cases:

- **Unordered node**

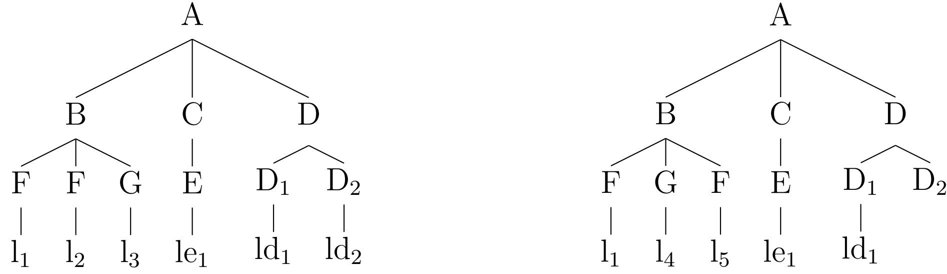



Figure A.1: Example of ASTs collection

The node has more than one child, and its children are repeated, or the children order is different depending on the occurrence in the ASTs.
In the considered example, the only unordered node is B since it has more than one child (3), and some of its children are repeated (2 times label F in each AST). Moreover, it does not matter here, but the order between its children is not the same from one occurrence to the other; in one AST, we have {F,F,G} and in the other AST, we have {F,G,F}.
B will be represented as $B[false, 1..*, F¥*, G¥*]$.
*false* means that this node is not ordered, *1..** means that the number of children is not determined and *F* and *G* followed by *¥** means that these are the children of B and that they will appear 0 or more times each.

Here are 2 examples of this type of nodes in Java:

- modifiers[false,1..*,Modifier¥*,MarkerAnnotation¥*,SingleMemberAnnotation¥*]
- statements[false,1..*,IfStatement¥*,ForStatement¥*,WhileStatement¥*,...]

- **Ordered node**
  The children of the node are ordered, and the number of children is identical no matter what the occurrence in the ASTs.
  For ordered nodes, there is then a difference that is specified between mandatory and optional children. By default, a child is considered mandatory, but it becomes optional in some cases. In the case where a child is a leaf node (it has no children), it will be considered optional (**condition 1**). Nodes that have exactly one child with a different label, their children are also regarded as optional (**condition 2**).

  In the example considered, node A is ordered because its number of children is always the same, and the order between them also. A does not fit in any of the specific cases for optional children, so its children are all mandatory,

and A is represented as $A[true, 3, B¥true, C¥true, D¥true]$.
*true* means that this node is ordered, *3* means that the number of children is determined. *B*, *C* and *D* means that these are the children of A. The boolean value after ¥ tells us if the child is mandatory or not, and for A, the three children are mandatory.

Here are some examples of nodes like A in Java:

- SimpleName[true,2,identifier¥true,var¥true]
- ImportDeclaration[true,3,static¥true,name¥true,onDemand¥true]

Node D, in our example, fulfills condition 1 for having an optional child. Indeed, one of its children, node $D_2$, is a leaf node in one of its occurrences. So $D_2$ is considered an optional child while $D_1$ is a mandatory child. D is represented as $D[true, 2, D_1¥true, D_2¥false]$.

Here are some examples of nodes like D in Java:

- IfStatement[true,3,expression¥true,thenStatement¥true,elseStatement¥false]
- ForStatement[true,4,initializer¥true,expression¥true,updaters¥false,body¥true]

And finally, in our example, node F fulfills condition 2 for having optional children because he has only one child, and this child has label $l_1$ or $l_2$. Its children are considered as optional children and F is represented as $F[true, 2, l_1¥false, l_2¥false]$

Here are some examples of nodes like F in Java:

- name[true,1,QualifiedName¥false,SimpleName¥false]
- identifier[true,1,org¥false,apache¥false,tools¥false,...]

The final grammar of our example is the following:
$A[true, 3, B¥true, C¥true, D¥true]$
$B[false, 1..*, F¥*, G¥*]$
$C[true, 1, E¥true]$
$D[true, 2, D_1¥true, D_2¥false]$
$D_1[true, 1, ld_1¥true]$
$D_2[true, 1, ld_2¥false, ¥false]$
$E[true, 1, le_1¥false, le_2¥false]$
$F[true, 2, l_1¥false, l_2¥false]$

$G[true, 1, lg_1¥false, lg_2¥false]$

In this section, we have presented in a rather detailed way and with many examples of how the grammar is built to apply some constraints.

# Appendix B

# Configuration file

```
# JAVA configurations
# data is one or two−class? true: 2−class data; false: 1−class
2Class = true
# input directory
inputPath = input
# output directory
outputPath = output
####################
#parameters used when 2class = true
####################
# path of old version
inputPath1 = commit1
# path of new version
inputPath2 = commit2
# minimum discriminative score
minDSScore = 0.4
# number of highest score patterns at the end
numPatterns = 1000
# timeout (minutes)
timeout = 10
# min number of leaf
minLeaf = 1
# max number of leaf
maxLeaf = 3
# min number of node
minNode = 10
```

```
# running two steps or not? true: running two step; false: running one step
twoStep = true
# count support: true: count support based on number of occurrences; false: count
    support based on number of files
weighted = true
# replace all variable names by a dummy node (*)
abstractLeafs = false
####################
# build grammar: true – build grammar from input data; false – read grammar from
    given file
buildGrammar = true
# file contains a list of root labels
rootLabelFile = conf/listRootLabel.txt
# file contains a list of label that only allow in patterns
whiteLabelFile = conf/listWhiteLabel.txt
# file contains a list of xml characters
xmlCharacterFile = conf/xmlCharacters.txt
# file contains a list of xml characters: used when abstractLeafs = true
reservedVariableNameFile = conf/reservedVariable.txt
# file contains configuration of clustering algorithm
clusterConfig = conf/cluster_conf.json
####################
#configurations for running parallel
####################
# list of minimum support thresholds
minSupportList=3,4,5
# list of folders: note – there is no space between folders
inFilesList=sub_dir1,sub_dir2
```

# Bibliography

[1] "Git," https://git-scm.com/, 2021.

[2] H. Pham, S. Nijssen, K. Mens, D. Di Nucci, T. Molderez, C. De Roover, J. Fabry, and V. Zaytsev, "Mining patterns in source code using tree mining algorithms," 10 2019, pp. 471–480.

[3] A. Jiménez, F. Berzal, and J.-C. Cubero, "Frequent tree pattern mining: A survey," *Intell. Data Anal.*, vol. 14, pp. 603–622, 01 2010.

[4] Y. Chi, R. Muntz, S. Nijssen, and J. Kok, "Frequent subtree mining - an overview," *Fundam. Inform.*, vol. 66, pp. 161–198, 01 2005.

[5] Y. Chi, Y. Yang, and R. Muntz, "Canonical forms for labelled trees and their applications in frequent subtree mining," *Knowl. Inf. Syst.*, vol. 8, pp. 203–234, 08 2005.

[6] A. Jiménez, F. Berzal, and J.-C. Cubero, "Potminer: Mining ordered, unordered, and partially-ordered trees," *Knowl. Inf. Syst.*, vol. 23, pp. 199–224, 05 2010.

[7] "Freqt: An implementation of freqt (frequent tree miner)," http://www.chasen.org/~taku/software/freqt/, 2021.

[8] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and A. Setsuo, "Efficient substructure discovery from large semi-structured data," 04 2002.

[9] Y. Chi, S. Member, Y. Xia, Y. Yang, and R. Muntz, "Mining closed and maximal frequent subtrees from databases of labeled rooted trees," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, 10 2004.

[10] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," *Proc. 20th Int. Conf. Very Large Data Bases VLDB*, vol. 1215, 08 2000.

[11] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," vol. 29, 06 2000, pp. 1–12.

[12] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi, “Efficient pattern-growth methods for frequent tree pattern mining,” 05 2004, pp. 441–451.

[13] K. Mens, S. Nijssen, H. Pham, C. De Roover, J. Fabry, and V. Zaytsev, “Intimals prototype : Adaptation of the frequent subgraph mining algorithm to support,” 05 2020.

[14] D. Di Nucci, H.-S. Pham, J. Fabry, C. De Roover, K. Mens, T. Molderez, S. Nijssen, and V. Zaytsev, “A language-parametric modular framework for mining idiomatic code patterns.” in *SATToSE*, 2019.

[15] “Intimals - java importer,” https://gitlab.soft.vub.ac.be/intimals/metamodel-tools/-/tree/master/java-importer, 2021.

[16] “Intimals - forestmatcher,” https://gitlab.soft.vub.ac.be/intimals/forest-matcher, 2021.

[17] “Github repository - jgraphx,” https://github.com/jgraph/jgraphx, 2021.

[18] “Freqtals result,” https://github.com/QuentinHauspie/FreqTals_result, 2021.

[19] “Intimals - how to build grammar,” https://gitlab.soft.vub.ac.be/intimals/metamodel/-/blob/master/extracted_java_grammar/extracted_grammar_instruction.pdf, 2021.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**