UCLouvain

epl

École polytechnique de Louvain

Mining patterns in software - the FreqTals algorithm:

Maintenance of the python implementation

Author: **Arnaud Spits**
Supervisors: **Kim Mens, Seigfried Nijssen**
Reader: **Axel Legay**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

The FREQTALS algorithm, was first developed in 2019 for mining patterns in AST. This study shows that AST well captures the structure of the source code (it is a compiled version of). Patterns found in such data-structure could be analysed to enhanced our knowledge of software programming.

This algorithm was first implemented in Java. However, Python was then consider as this programming language is known to be easier to learn and implement. Accordingly, a new version of the algorithm written in Python was derived of the Java version.

This thesis shows how this Python implementation has been improved on different aspects. Its readability was enhanced and make the source code easier to understand. This research also bring a new structure to classes which contribute to the maintainability of this new implementation. Additionally, this new version of the FREQTALS algorithm is more efficient and allows convenient use for future researches.

Acknowledgements

I would like to express my gratitude to my supervisors, *Kim Mens* and *Seigfried Nijssen* for their availability. They offered me constructive feedback and advice which guided me through the realisation of this project.

I also would like to thanks my family who helped me in the redaction of this document.

Contents

1 Introduction **5**

2 Background definition **7**
2.1 Abstract Syntax Tree 7
2.2 Frequent Subtree Mining 8
2.3 FREQT 10

3 FREQTALS **13**
3.1 Constraints 14
3.2 Algorithm 17
3.3 Additional work 20

4 Maintenance of an existing Python implementation **21**
4.1 Improvement of the python style 21
4.1.1 Overall readability 22
4.1.2 Syntactic sugar 24
4.1.3 Class related implementation correction 26
4.2 Restructuring FreqTals algorithms 27
4.2.1 FreqT.py file 27
4.2.2 FreqT_ext.py file 28
4.2.3 New approach 28
4.3 Data structure 33
4.3.1 FTArray 33
4.3.2 Pattern extension 34
4.3.3 Projected and Location 35
4.4 Implementation correction 36
4.5 Additional change 42
4.5.1 Excessive use of *try except* 42
4.5.2 Support computed multiple times 42

5 Validation **43**
5.1 Readability . . . 43
5.2 Maintainability . . . 44
5.3 Reusability . . . 44
5.4 Efficiency . . . 45
5.5 Correctness . . . 49

6 Possible improvement **51**
6.1 Pre-processing step . . . 51
6.2 Simpler approach for *FreqTSubtree* . . . 52
6.3 Mandatory child constraint C5 and label of leaf nodes constraint C6 used for pruning . . . 53

7 Conclusion **55**

Chapter 1

Introduction

In the domain of software development, during the implementation process, software engineers usually applied a set of programming conventions (like design patterns) and other "good practices". Those practices are essential as they contribute to structuring and improving source codes. Studying or even discovering such practices could thus be very helpful. By enhancing our knowledge of them, we offer new tools for the development of new software.

As those programming practices and conventions are crucial to obtain a good program, they usually appear frequently through out its source code. This is the initial intuition behind the development of the FREQTALS [5] algorithm in 2019. The idea being that we could use *frequent mining algorithms* to find patterns in written code which reflect "good practices" for implementing software. One particular data-structure of interest to achieve such goal is Abstract Syntax Trees (ASTs). ASTs are tree representations of the syntactic structure of source codes. They are utilized by programming language to perform syntactic and semantic analyses on the code, but also to execute it. The advantage of such data-structures is that they well capture the effective structure of the code while not including punctuation. Frequent subtrees/patterns of ASTs could be analysed and interpreted, for example, as "good programming practices".

One frequent mining algorithm which was originally considered for that task is FREQT [1]. FREQT is a constraint-based frequent mining algorithm design for tree structures. While correct, the quantity of patterns in ASTs can grow exponentially large. Inspired from the FREQT algorithm and specificities of ASTs, the FREQTALS [5] algorithm proposes a new method for mining *labelled, ordered, and rooted trees*. This method enforces label-based constraints on patterns found by the algorithm in order to prune the search space, but also helps refining the "quality" of found patterns. In fact in our context, we are looking for patterns which well reflect structures from source code. In the same mind set, this method proposes to mine for *maximal frequent patterns*.

The general idea being to grow patterns as large as possible which usually are more expressive and easier to interpret.

This algorithm was originally implemented in Java [5]. While Java is a pretty efficient language, the Python language was subsequently considered for a new version of the FREQTALS algorithm implementation. In fact, while less efficient, the Python language is known to be easy to learn, implement and execute. This Python implementation would be more convenient and easier to adapt in future works related to the FREQTALS algorithm.

Loïc Quinet was offered in 2021 to produce this Python version of the algorithm [7]. This implementation was directly translated from the original Java implementation. However, during the translation process, many specificities of the Python language were lost or not taken into account. This thesis initially aims at improving the coding style of this version in order to make it more readable and easier to modify/adapt in future works. In addition, an other work on the Java version was concurrently done by *Quentin Hauspie* [9]. This work mentions some weaknesses of the Java version which were inherited by the Python version. Accordingly, an additional goal of this thesis is to correct those weaknesses.

In summary, this thesis focuses on the maintenance of the Python implementation of the FREQTALS algorithm [7]. We will attempt to improve the readability and maintainability of the code. But also correct its weaknesses while also trying to improve performance.

We will first present in chapter 2 background knowledge related to the implementation of the FREQTALS algorithm (algorithm explained in chapter 3). Then, we will cover in chapter 4 the set of modifications and improvements made on the code while discussing their advantages and impacts. Subsequently, we will assess in chapter 5 those modifications through 5 aspects: readability, maintenability, reusability, efficency and correctness. In chapter 6, we will suggest some paths of improvement. Finally, we will in chapter 7 conclude on the results of this thesis.

Chapter 2

Background definition

In this section, we will cover the background definitions which the FREQTALS algorithm is built on. We will first introduce the data structure type (AST) which the FREQTALS algorithm is designed for. We will then present the problem solved by this algorithm. And finally, we will explain the FREQT algorithm which is the foundation of the FREQTALS algorithm.

2.1 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a tree representation of the syntactic structure of a source code (written in a programming language). Typically, the programming language compiles source codes in AST form in order to perform syntactic and semantic analyses before its execution.

Nodes of the AST represent syntactic elements (from the programming language grammar) identified by a label. Nodes can represent "loop", "operation", "function definition", "variable assignation", ... Each node has an arbitrary number of children. Note that the ordering of the children of a node is meaningful.

The advantage of AST is that it doesn't include punctuation and directly represents the structure of the source code. Find an example of ASTs on Fig. 2.1.

ASTs are *labelled, ordered, and rooted trees.* We consider in this thesis the following definition for such a tree as it was originally defined for the FREQTALS algorithm:

Labelled, ordered, and rooted trees [5]: $T = (V, E, \lambda, \Sigma)$; $V = \{1, 2, ..., n\}$ is the set of node identifiers; $E \subseteq V \times V$ is the set of edges; $\lambda : V \mapsto \Sigma$ is the function that associate labels to nodes of V; Σ is the set of allowed labels. Since T is ordered, we assume that nodes are identified using contiguous integers listed in the order of *depth first* with *left-right* traversal. Node n is thus called the *rightmost node* of the tree.

Given 2 trees $T_1 = (V_1, E_1, \lambda_1, \Sigma)$ and $T_2 = (V_2, E_2, \lambda_2, \Sigma)$, T_2 is an induced subtree of T_1 ($T_1 \succ T_2$) if there is an injective function $f : V_2 \mapsto V_1$ such that: (1) edges 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) orders are preserved $\forall v_1, v_2 \in V_2 : v_1 < v_2 \Rightarrow f(v_1) < f(v_2)$.

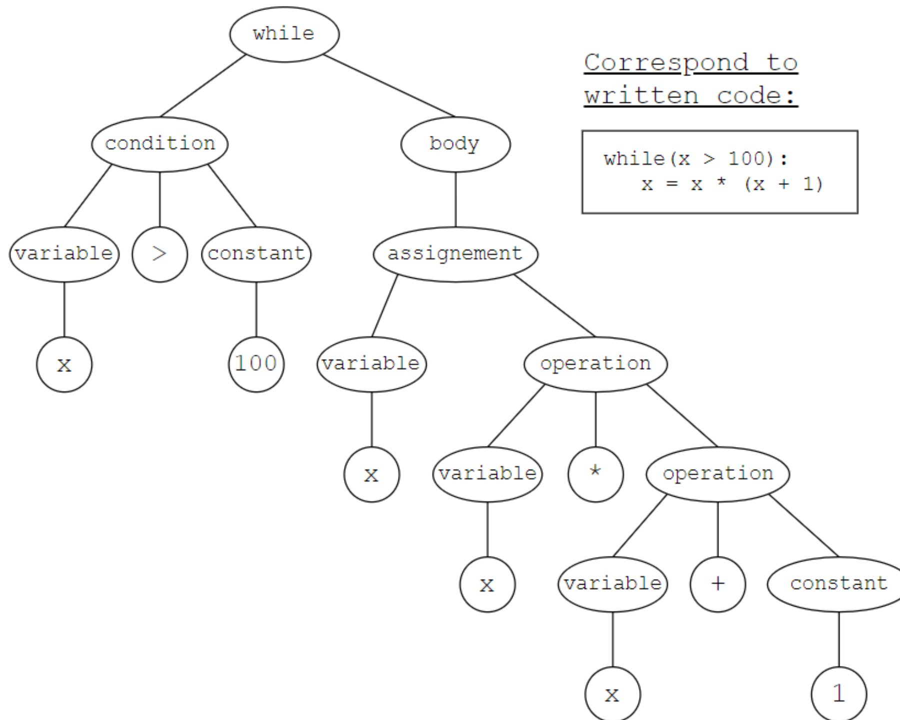

Figure 2.1: Example of Abstract Syntax Trees

2.2 Frequent Subtree Mining

Frequent subtree mining is a well-known problem in computer science [2, 3]. It consists of finding all patterns which have a higher *support* than a given threshold. In our context, patterns correspond to subtrees from a set of trees. We will refer to this set of trees as database. To understand *Frequent subtree mining*, we first have to define 2 key concepts:

Occurrence of a pattern: A subtree T_o from the database is an occurrence of a pattern T_p iff there exists a bijection between T_o and T_p.

Support of a pattern: the *support* of a pattern reflects how frequently this pattern appears in a database. Note that the *support* of a pattern varies depending on the database considered. There is many different ways of defining the *support*. For example, given a pattern, *support* could correspond to:

- The number of trees from the database in which this pattern occurs at least once.
- The number of occurrences of this pattern in the database.

They are the definitions utilized in this thesis, respectively denoted "support" and "weighted support".

These patterns (with a *support* over a given threshold) are called *frequent patterns* and satisfy the *minimum support constraint* denoted **C0**.

However, in practice, the number of *frequent patterns* can grow exponentially large. In fact, given a large *frequent pattern* from the database, every subtree of this pattern is also *frequent* by definition and thus has to be included. To address this issue, one usual agreement is to mine for *condensed representations* [2, 3] and *maximal frequent subtrees*.

Maximal frequent subtrees: Let $\mathcal{T}_c$ denote the set of pattern satisfying **C0**. A frequent subtree T is *maximal* iff T is not dominated by any other frequent patterns: $T \in \mathcal{T}_c \mid \nexists T' \in \mathcal{T}_c : T' \succ T$. This concept is illustrated on Fig. 2.2.*Maximal frequent subtree mining* is the problem of finding all *maximal frequent subtrees* $= \{T \in \mathcal{T}_c \mid \nexists T' \in \mathcal{T}_c : T' \succ T\}$ [5].

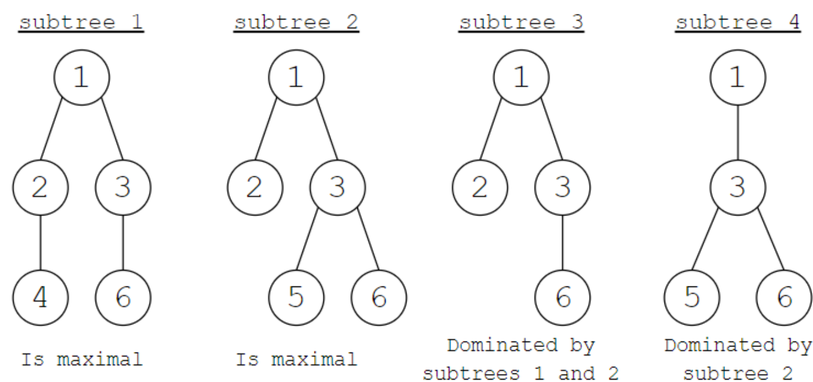

Figure 2.2: Illustration of the *maximal frequent subtrees*

2.3 FREQT

FREQT [1] algorithm is a *Frequent subtree mining* algorithm designed for labelled ordered rooted trees. Its overall structure is described in Algorithms 1 and 2 (adapted version of a previous study [5]).

Algorithm 1 FREQT

```
1: FP ← ∅
2: C ← findLabels()
3: prune(C)                                   ▷ C0 and C1
4: for c in C do
5:     FP.add(c)                              ▷ C2
6:     expand(c)
7: return FP
```

Algorithm 2 expand procedure

```
1: function EXPAND(f)
2:     C ← findCandidates(f)         ▷ grow patterns using rightmost path
3:     prune(C)                               ▷ C0 and C1
4:     for c in C do
5:         FP.add(c)                          ▷ C2
6:         expand(c)
```

The algorithm recursively grows new patterns from already encountered patterns, starting from a set of single node patterns denoted *root patterns*. Candidate patterns are created by adding one node to another patterns. But note that without any restrictions, this algorithm is very likely to repeat same patterns multiple times. Indeed, given a pattern T, there exists exactly n_{leaf} other patterns which this pattern is an extended version of (where n_{leaf} corresponds to the number of leaf nodes of T).

In order to ensure that each pattern is visited/created only once, they are expanded using *the rightmost path* extension. The idea is to add node only to the right of the *rightmost path* (see Fig. 2.3).

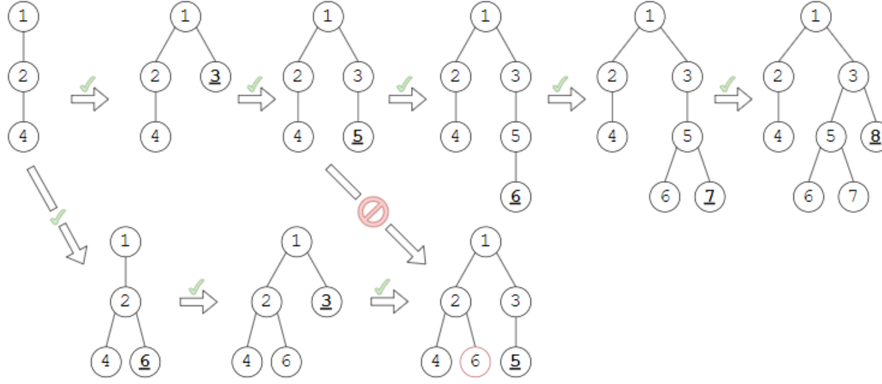

Figure 2.3: Illustration of extensions to the right of the *rightmost path*

During the execution, for a given pattern, the algorithm generates a set of candidate patterns. This set of candidates can be pruned before any further expansion. Pruning is used to effectively reduce the search space with anti-monotonic constraints.

Anti-monotonicity: A constraint C is anti-monotonic iff $\forall T_1, T_2$ with $T_1 \succeq T_2$, if T_2 does not satisfy $C \Rightarrow T_1$ does not satisfy C either [4].

By default FREQT [1] prunes candidates according to *minimum support constraint* (**C0**) and *maximum size constraint* (**C1**).

Maximum size constraint C1: This constraint defines a maximal size threshold on patterns. Patterns can be pruned if their size exceeds this threshold. The size of a pattern can either be defined by its number of nodes or its number of leaf nodes.

Subsequently, the FREQT algorithm continues expanding patterns (from the set of candidates) in depth-first. This operation is illustrated on Fig. 2.4. While extending patterns, *frequent patterns* found are stored in the output. In addition, we may want to avoid small patterns from being outputted as they can be undesirable depending on the context. Accordingly, patterns have to satisfy a *minimum size constraint* (**C2**) before being added to the output [1].

Minimum size constraint C2: This constraint defines a minimum size threshold on patterns. Patterns cannot be outputted if their size is below this threshold. The size of a pattern can either be defined by its number of nodes or its number of leaf nodes. Note that since **C2** is monotonic it cannot be used to prune patterns.

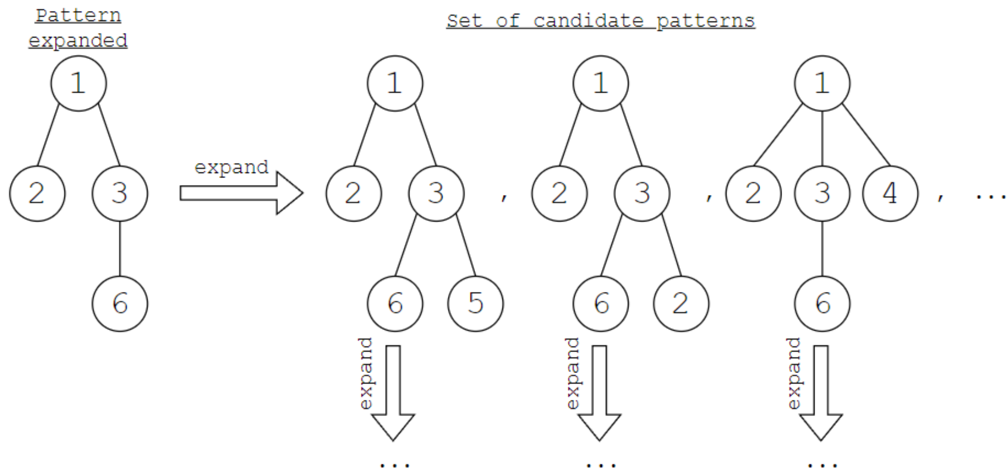

Figure 2.4: Example of the expansion of one pattern

Remark

When extending one pattern, it would be very inefficient to generate every possible extension since many of them are very likely to not even occur in the database. Given a pattern, the algorithm instead starts from each of its occurrences and considers every node attached to it (to the rightmost path) as potential extensions [12]. As a result of this operation, we have computed for each extension, their set of occurrences from which the *support* can easily be derived (which make the pruning **C0** convenient).

Chapter 3

FREQTALS

FREQTALS [5] is an extension of the FREQT [1] algorithm. It was developed and published in 2019 by *Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry and Vadim Zaytsev.* The initial trigger of this project was to perform analyses on ASTs. AST is an interesting representation of source code because it well captures the structure of the code. *Frequent subtree mining* algorithms, like FREQT, could be used on such data structures. However, those algorithms usually output a lot of patterns. This quantity of patterns makes them hard to interpret or even unusable. Accordingly, a set of new label-based constraints was implemented inside the FREQT algorithm in order to refine the output / filter patterns which correctly reflect interesting programming structures from less expressive patterns. FREQTALS implements a new approach for *constraint-based frequent subtree mining algorithm.* This approach has the advantage of providing easy to interpret constraints on patterns and guarantees to find all frequent patterns satisfying those constraints [5].

This section will be discussing the new ideas brought by this algorithm. We will first focus on each new constraint added to the FREQT algorithm, and then we will cover the overall execution of this FREQTALS algorithm. You can find a summary of its implementation at the end of this section: Algorithm 3, 4 and 5 (adapted version of a previous study [5]).

3.1 Constraints

The FREQTALS algorithm introduces a set of new label-based constraints **C3** - **C6**. We will discuss their purpose and how they are implemented. Their explanation are inspired from their original definition for the FREQTAL algorithm [5] (note that the constraint C5 and C7 from this original paper have been grouped into one constraint **C5** because their application is very similar).
Additionally, we will also discuss why mining for *maximal frequent subtree* is, in our context, convenient.

C3 - Constraint on the label of root nodes

This constraint limits the set of labels allowed to occur as root of patterns, as we may be interested to study those root nodes and the hierarchy of nodes below them. It also contributes to the reduction of the search space by reducing the initial number of *root patterns* to be expanded by the algorithm.

This constraint is implemented during the creation of the initial set of *root patterns*. One *root patterns* is created for each label allowed, and with a single traversal through the database, we can easily compute their occurrences. Note that this constraint is anti-monotonic.

As the initial set of *root patterns* satisfies C3 and new patterns are created by extension of those *root patterns*, every pattern generated by the algorithm also satisfies C3.

C4 - Blacklisted labels

We can also provide the FREQTALS algorithm with a list of labels forbidden to occur in patterns. In an effort to reduce the search space, it allows to divide trees from the database in smaller trees. Instead of checking for each new extension whether they are blacklisted or not (which would be computationally inefficient), the database is at first preprocessing in order to prevent those blacklisted labels from occurring in patterns.

During the initialisation of this database, the algorithm reads trees from a file given as input and skips branches when a black-label is encountered (see Fig. 3.1). As those blacklisted label are not present in the database, every pattern generated by this algorithm on this database satisfy **C4**. As a result, portions of trees from the database are ignored. However, this method could cause the support of some patterns to be miscalculated [9]. We will be discussing this aspect in section 6.1.

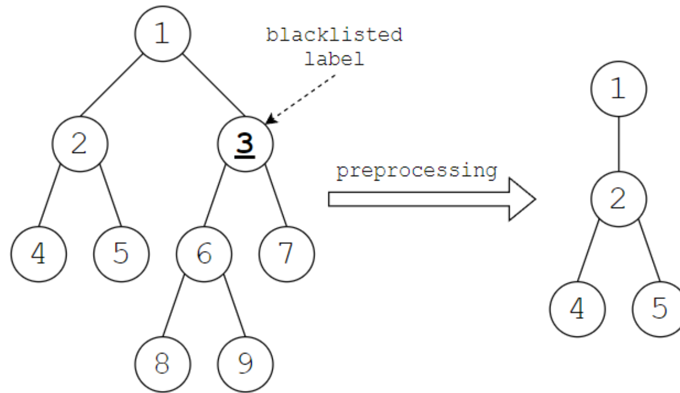

Figure 3.1: Illustration of the preprocessing of the database to satisfy **C4**

C5 - Obligatory child constraint

The grammar of a programming language is used to define the syntax of the language and thus the overall structure of ASTs (produced by this language). In this AST representation, it tells us about the children that parent node can have. For example, we can expect an *operation* labelled node to have *operand*, *operator* or other *operation* nodes as children. Accordingly, some children can be mandatory for the structure to be syntactically correct. For example, *while loop* nodes require one *condition* node which defines whether we should continue looping. In our mining context, we are interested to keep those mandatory links otherwise patterns would lose their meaning.

The FREQTALS algorithm allows to define a grammar as input. It is built using a dictionary which lists, for each node concerned, an ordered set of labels which corresponds to the rules for the constitution of their children (from left to right). Labels from this set can either be defined as mandatory (and thus must occur) or as optional. It can be utilized to make some children obligatory but also to limit the number of siblings in a pattern that can have the same label (for example, we may want to limit the number of *variable assignment* siblings as they usually are very frequent in the code *body*).

During the expansion process, we can detect whether a pattern violates this constraint (illustrated in Fig. 3.2). Note that we can here use the concept of anti-monotonicity.

In fact, if the pattern violates **C5**, so do its extended versions (remember that node can only be added to the right of the *rightmost path*). **C5** can thus be used to prune patterns and effectively reduce the search space. This operation is denoted as *left mandatory child* constraint.

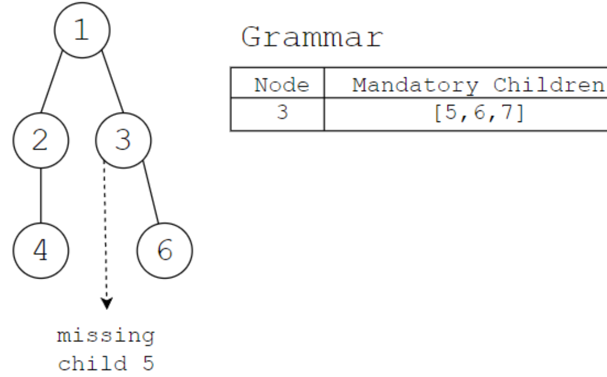

Figure 3.2: Illustration of the *left mandatory child* constraint

However, this constraint can only be fully determined when the pattern has stop expanding, since patterns can have an incomplete set of children (according to the provided grammar) during the expansion phase. In fact, on Fig. 3.2, we cannot tell yet whether the pattern is missing child 7 as future extensions could include this node.

When a pattern satisfying every other constraint is found, before any addition to the output, the algorithm has to verify whether every node of this pattern and their children agree with the provided grammar. This operation is denoted as the *right mandatory child* constraint. It requires some computation time, but the number of patterns on which this constraint have to be checked is significantly smaller than the total number of patterns generated.

C6 - Constraint on the label of leaf nodes

In an effort to find patterns that reflect interesting/meaningful *structures* to interpret, we would like that patterns include program-specific information [5]. Such information are defined on top of the syntactical structure of source code. They are detained by *leaf nodes* of ASTs.

Accordingly, we impose that all leaf nodes in a pattern must have a label included in Σ_{leaf}, where Σ_{leaf} is the set of labels that occurs in the leafs of the trees from the database. We'll denote such patterns as *leaf patterns*.

In order to implement this constraint, the algorithm stores encountered *leaf patterns*. Whenever a pattern cannot be extended anymore and so is considered for addition to the output, in order to satisfy **C6**, the algorithm should remove every extra extension which ends in *leaf nodes* with label $\notin \Sigma_{leaf}$. We thus obtain a *leaf pattern*.

This operation is equivalent to considering its closest parent *leaf pattern* (from the tree of generated pattern) for addition to the output. This is the approach that was implemented. The algorithm stores encountered *leaf pattern* and retrieves them when needed. However, this method has many drawbacks. We will discuss in section 4.4 how this method can be improved by instead notifying patterns whether they should be added to the output.

Mining maximal frequent subtree

In our context, we desire to find large patterns. Thanks to their size, they hold lots of information about the source code and are typically easier to interpret. At the contrary, we want to avoid subtrees of such patterns. Indeed, they repeat information and don't even contain as much information as large patterns. Mining for *maximal frequent sub-tree* is thus very convenient.

To do so, the algorithm scans the output and removes frequent patterns dominated by others. In order to save memory, every time a new *frequent pattern* (satisfying **C0-C6**) is added to the output, it is compared with every other *frequent pattern* already stored. Dominated patterns are removed accordingly.

But, given a pair of trees T_1 and T_2, the problem of determining whether T_1 dominates T_2 ($T_1 \succ T_2$) is non-trivial. The FREQTALS algorithm uses a naive and fast checking function which greedily matches nodes from T_2 with nodes from T_1 using a depth-first traversal of each tree. If this function fails, the algorithm switches to more complete method which however takes more computation time. This method is an variant of the FREQT algorithm. It consists of mining patterns in a database composed of 2 trees T_1,T_2 with a support threshold of 2. At the end of the mining process, if the largest *frequent pattern* found is equal to T_2 we can deduce that $T_1 \succ T_2$. In fact, it means that T_2 occurs in both T_1 and T_2.

3.2 Algorithm

In practice, even with the addition new constraints (**C3**,**C4**,**C5**) which help to reduce the search space, the tree of possible candidate patterns grows exponentially. We usually have to define a restrictive maximum size threshold **C1** on patterns for the algorithm to run in reasonable time. But, this constraint is undesirable. As already mentioned, we are interested in growing *frequent patterns* as large as possible, because large patterns provide more information about ASTs and are typically easier to interpret.

The FREQTALS algorithm addresses this issue by implementing a new way of finding large patterns [5] (Algorithm 3). The idea is to compute small sets of root occurrences to grow large patterns on. This method works in 3 steps:

- First, we mine frequent patterns using the FREQT algorithm under constraints **C0** - **C6**. Note that **C1** is the primary tool to reduce the search space. Fine tuning of **C1** is required to obtain good performance. As a result, we obtain a set of *frequent patterns* (each of which satisfies **C0** - **C6**) denoted $\mathcal{FP}$.

- Then, for each pattern we derive its set of *root occurrences* denoted $\mathcal{RO}$. *Root occurrences* of a pattern correspond to the root node of its occurrences in the database denoted $occ(T)$.

  $$\mathcal{RO} = \{occ(T)|T \in \mathcal{FP}\}[5]$$

  However, the FREQTALS algorithm introduces here a new intuition. Let r_{small},r_{big} be 2 sets of *root occurrences*, with r_{small} being a subset of r_{big} ($r_{small} \subset r_{big}$). We note that patterns occurring in r_{big} also occurs in r_{small}. Accordingly, r_{big} can be remove from RO with the guarantee that every *maximal frequent pattern* will still be found by the algorithm. Subsequently, we only keep sets that are *minimal*:

  $$\mathcal{ROM} = \{r \in \mathcal{RO}|\nexists r' \in \mathcal{RO} : r' \subset r\}[5]$$

  In practice, this step is processed during the execution of the first step (in a similarly fashion as the *maximal frequent subtree* constraint). To preserve memory space, whenever a frequent pattern is generated, its set of root occurrences is added to $\mathcal{ROM}$ and compared with already stored sets. *No-minimal* ones are removed accordingly.

- Finally, we call the FREQT algorithm under constraints **C0**,**C2** - **C6** for each set of root occurrences $\in \mathcal{ROM}$. Note that we have dropped the constraint **C1**. In fact, we are looking for large patterns. By reducing the initial set of root occurrences, we expect the search to be more efficient.

we will denote the implementation that takes this optimisation into account as *FreqTals 2 steps*. The naive implementation which only calls FREQT once will be denoted as *FreqTals 1 step*.

Algorithm 3 FREQTALS

```
1: FP = freqt()
2: ROM ← groupRootOccurence(FP)
3: MP ← ∅
4: for r in ROM do
5:     MP_r ← mineMaximalSubtree(r)      ▷ equivalent to expand() without C1
6:     MP ← MP ∪ MP_r
7: return MP
```

Algorithm 4 FREQT with additional constraints

```
1: function FREQT()
2:     preprocessing()                    ▷ preprocess C4 blacklisted labels
3:     FP ← ∅
4:     PF_1 ← findLabels()                ▷ initialized from C3 the set of allowed roots
5:     prune(PF_1)                        ▷ prune using C0 minimum support
6:     for c in PF_1 do
7:         expand(c)
8:     return FP
```

Algorithm 5 expand procedure

```
1: function EXPAND(f)
2:     C ← findCandidates(f)              ▷ grow patterns using rightmost path
3:     prune(C)                           ▷ prune using C0 and C5 left mandatory child
4:     for c in C do
5:         if c is leaf pattern: then
6:             storeLeafPattern(c)
7:         if C1 then                     ▷ terminate the extension of C1 too large pattern
8:             l ← getLeafPattern()       ▷ get the closest parent leaf pattern
9:             FP.add(l)                  ▷ check C2 and C5 right mandatory child
10:        else
11:            expand(c)
```

3.3 Additional work

The FREQTALS algorithm in the state described above, is designed to treat databases composed of trees belonging to the same class. We will refer to this implementation as *freqTals 1 class*.

However, what *freqTals 1 class* was not originally designed for is the comparison of different scripts with one another. Such comparison could be made by grouping trees in two distinct classes and mining for patterns that are *frequent* in both classes. It could for example be utilized to compare two versions of the same code. But, *FreqTals 1 step* don't support this type of databases. This was the premise of the work conducted by *Kim Mens, Siegfired Nijssen and Hoang-Son Pham* in 2021 [8] which have adapted the FREQTALS algorithm to implement this feature. We will refer to it as *freqTals 2 classes*.

Chapter 4

Maintenance of an existing Python implementation

The primary goal of this thesis is to obtain a code for the FREQTALS algorithm easier to read and adapt. To do so, the new implementation [12] was adapted from a previous Python implementation made by *Loïc Quinet* [7]. In addition to this work, and other study by *Quentin Hauspie* has identified a mistake in the original Java implementation of the FREQTALS algorithm [9]. One additional goal of this thesis is to correct wrong implementations of the Python version and to correct mistakes from the Java version (which were inherited).

In this chapter, we will first cover the improvements made on the coding style to enhance readability. Then, we will present a new approach for structuring the FREQTALS algorithm in smaller classes. Subsequently, we will discuss the data-structure choice for this implementation. And eventually, we will talk about some mistakes present in the code and how they were solved. Additionally, we will mention a set of smaller improvements which didn't fit in the categories described above.

4.1 Improvement of the python style

As mentioned previously, the Python implementation of the FREQTALS algorithm [7] was directly derived from an existing Java implementation [5]. This really helps coding the algorithm. But, the resulting code style was more similar to Java. Many specificities of the Python language were lost or not taken into account during the translation.

One of the main goals of this thesis is to obtain a more readable implementation that could be more convenient for future works. As better readability usually means better understanding of the code. In an effort to improve the clarity of written

code, the following sections discuss how we tried to improve the implementation style of this Python implementation.

4.1.1 Overall readability

The main changes brought to the implementation involve its readability.

Most variable and function names were formatted in a non Pythonistic way. To better match the Python standards, they have been put in lower case separated by underscore (so called snake case). Some of them have even been renamed to better explain their purpose.

Example

- The implementation encodes labels into integers as they are more convenient to store inside data structures. In order to still being able to translate those integers into their corresponding label, the implementation build a dictionary used to decode labels.

  This variable was originally named:
  ```
  self._labelIndex_dict = dict()
  ```

  It as been renamed as:
  ```
  self.label_decoder = dict()
  ```

In the same mindset, most functions have received a detailed description of their purpose/usage (or their description has been developed). Their arguments and return values have been clearly identified. In addition, some additional comments and spacing have been inserted throughout the implementation to further improve the clarity of individual functions.

Example

- Here are some examples of such descriptions:

```
def add_tree(self, pattern, projected):
"""
 * Is called every time a frequent pattern (satisfying every constraints) is found
 !! note: this function should copy pat
:param pat: FTArray, the frequent pattern
:param proj: Projected, projection of pat
"""


def check_subtree(pat1, pat2):
"""
 Check if a on pattern is a subtree of an other pattern
:param pat1: FTArray
:param pat2: FTArray
:return: int, 0 = no subtree
              1 = pat1 is a subtree of pat2
              2 = pat2 is a subtree of pat1
"""
(...)


def satisfy_post_expansion_constraint(self, pat):
    """
     * check minimum size constraints
     * check right obligatory children
    :param pat: FTArray, a pattern
    :return: whether pat satisfy every constraint
    """
(...)
```

Note that unused functions (and even class) have been deleted:

- *Error* class implements an error object that could be thrown. However, the implementation doesn't raise this errors but uses errors provided by the Python language.
- *filterFP* function allows filter *maximal pattern* from an input. But the implementation instead gradually scans the built output for *non maximal pattern* using an other function *add_maximal_pattern()*.
- *printTransaction*, *printCandidates*, *printFTArray*, *printFTArray2*, *printProjected* are printing functions which are not invoked.

4.1.2 Syntactic sugar

The Python language provides a wide selection of built-in functions and syntactic sugar that can be used to further improve readability and even performance (Python is an interpreted language which usually makes programs slower, whereas built-in functions are directly written in C and are thus more efficient). Shorthand if, tuple assignment, del keyword, set data-structure, ... are examples of such improvements that have been made on the implementation.

Example

- The support of patterns was previously computed by counting the occurrences stored in its projection *projected* with a for loop. The support was computed as followed:

```
def getRootSupport(self, projected):
    rootSup = 0
    oldLocationID = -1
    oldRootID = -1
    for i in range(projected.getProjectLocationSize()):
        locationID1 = projected.getProjectLocation(i).getLocationId()
        rootID1 = projected.getProjectLocation(i).getRoot()
        if locationID1 == oldLocationID and rootID1 != oldRootID or locationID1 != oldLocationID:
            rootSup += 1
        oldLocationID = locationID1
        oldRootID = rootID1
    return rootSup
```

  With the use of *len()* and set from the built-in Python library, this function can be rewritten:

```
return len({(loc.get_location_id(), loc.get_root()) for loc in projected.get_locations()})
```

  The weighted support has received a similar treatment:

```
return len({loc.get_location_id() for loc in projected.locations})
```

- *addRootIDs()* is used to add one set of occurrences to the output *rootIDs_dict* and remove super-sets of other sets. The previous implementation had to implement additional function (provided in an *Util* package):

```
def addRootIDs(self, pat, projected, _rootIDs_dict):
    # find root occurrences of current pattern
    util = Util()
    rootOccurrences = util.getStringRootOccurrence(projected)

    # check the current root occurrences existing in the rootID or not
    isAdded = True
    l1 = rootOccurrences.split(";")

    to_remove_list = list()
    for key in _rootIDs_dict:
        rootOccurrence1 = util.getStringRootOccurrence(key)
        l2 = rootOccurrence1.split(";")
        # if l1 is super set of l2 then we don't need to add l1 to rootIDs
        if util.containsAll(l1, l2):
            isAdded = False
            break
        else:
            # if l2 is a super set of l1 then remove l2 from rootIDs
            if util.containsAll(l2, l1):
                to_remove_list.append(key)
    for elem in to_remove_list:
        _rootIDs_dict.pop(elem, -1)
    if isAdded:
        # store root occurrences and root label
        rootLabel_int = pat.subList(0, 1)
        _rootIDs_dict[projected] = rootLabel_int
```

But this function can be re-implemented using Python set:

```
def add_root_ids(pat, proj, root_ids_list):
    """ ... """
    # set of root occurrences for current pattern
    root_occ1 = {(loc.get_location_id(), loc.get_root()) for loc in proj.get_locations()}

    index = 0
    # only keep sub-sets
    while index < len(root_ids_list):
        root_occ2 = root_ids_list[index][1]
        if len(root_occ1) <= len(root_occ2):
            if root_occ1.issubset(root_occ2):
                del root_ids_list[index]
                index -= 1
        else:
            if root_occ1.issuperset(root_occ2):
                return
        index += 1

    # store root occurrences and root label
    root_ids_list.append((pat.get(0), root_occ1))
```

4.1.3 Class related implementation correction

Unlike Java, objects in Python are initialized using *___init___()* (if it exists). But many classes (*FreqT*, *FreqTExt*, *FTArray*, *Location*, *Projected*, ...) missed this function. This means that when a new object had to be created, the object was initialized twice: first with the default inherited initialisation method, then using an additional initialization function.

For example, *addProjectLocation()* function is used inside *Projected.py* to add new locations to the projection of one pattern on the database:

```
def addProjectLocation(self, classID, id, pos, occurrences):
    loc = Location()
    loc.location2(occurrences, classID, id, pos)
    # ... proceed to add this location to the stored list of locations
```

But the initialisation of a *Location* object can be simplify as followed:

```
def addProjectLocation(self, classID, id, pos, occurrences):
    loc = Location(classID, id, pos, occurrences)
    # ... proceed to add this location to the stored list of locations
```

On the other hand, some other classes (*Constraint*, *PatternInt*, *Variable CheckSubtree*, ...) were only including static functions/variables. Whenever those functions had to be used, the corresponding class was initialized. Static functions, in Python, can be implemented outside classes.

For example, a *Constraint* object was created every time *expandPattern()* was called in order to *prune* the generated set of candidates.

```
def expandPattern(self, pattern, projected):

    candidates_dict = self.generateCandidates(projected, self._transaction_list)

    constrain = Constraint()
    constrain.prune(candidates_dict, self._config.getMinSupport(),
                    self._config.getWeighted())

    for key in candidates_dict:
        # ...  continue expanding candidates: expandPattern()
```

But this is not necessary:

```
def expandPattern(self, pattern, projected):

    candidates_dict = self.generateCandidates(projected, self._transaction_list)

    Constraint.prune(candidates_dict, self._config.getMinSupport(),
                    self._config.getWeighted())

    for key in candidates_dict:
        # ...  continue expanding candidates: expandPattern()
```

4.2 Restructuring FreqTals algorithms

In this section, we will be discussing how the overall class structure of the FREQ-TALS algorithm can be improved to make it more readable and flexible. We will cover the previous state of *FreqT.py* and *FreqT_ext.py* (which are the files related to the implementation of the algorithm) in order to highlight the explanation of the new approach.

4.2.1 FreqT.py file

Previously, the *FreqT* class combined different implementations of the FREQTALS algorithm. Depending on the initial configuration, the program either runs *FreqTals 1 class or 2classes, 1 step or 2 steps.* This implementation in one class has downsides :

- The class is unnecessary big, which makes it harder to read/understand and update. The combination of different implementations of the FREQTALS algorithm into one class further adds complexity to the code. Indeed, the addition of conditional clauses are required to implement their differences. This can be constraining if located in a frequently called functions.

  During the process of expanding patterns, whenever the algorithm finds a good pattern, it calls the function *add_tree()* which requests the addition of this newly found pattern to the output (note that *add_tree()* also ensures that patterns satisfy the post expansion constraints). However, each variant of the FREQTALS algorithm allows different types of patterns in their output. Accordingly, add_tree() has to check which configuration is currently used and delegates to the appropriate function.

  ```
  def addTree(self, pat, projected):
      # post expansion constraints
      if constraint.checkOutput(pat, ...) and not constraint.missingRightObligatoryChild(pat, ...):
          if self._config.get2Class():
              if constraint.satisfyChiSquare(projected, ...):
                  if self._config.getTwoStep():
                      self.addHighScorePattern(pat, projected, self.__HSP_dict)
                  else:
                      self.addMaximalPattern(pat, projected, self.MFP_dict)
          else:
              if self._config.getTwoStep():
                  self.addRootIDs(pat, projected, self.rootIDs_dict)
              else:
                  self.addMaximalPattern(pat, projected, self.MFP_dict)
  ```

  The combination of many algorithm variants in a single class is overall inflexible. In fact, this approach makes it harder to implement new variants of the original algorithm.

4.2.2 FreqT_ext.py file

As the second step of the FREQTALS algorithm is significantly different from the other algorithms, it was implemented inside his own class *FreqT_ext* [5], extended version of *FreqT*. But *FreqT_ext* is used at the end of the first step and thus inside its own parent. This causes major import issues.

In fact, in Python, whenever we import a class, all its import clauses are also executed. Importing *FreqT_ext* inside its parent would cause an infinite loop of imports. This issue was solved by importing this external class during the execution:

```
# run the second step
import freqt.src.be.intimals.freqt.core.FreqT_ext as freqtext
freqT_ext = freqtext.FreqT_ext()
freqT_ext.FreqT_ext(self._config, self._grammar_dict, self._grammarInt_dict, self._blackLabelsInt_dict,
                    self._whiteLabelsInt_dict, self._xmlCharacters_dict, self._labelIndex_dict,
                    self._transaction_list, self.sizeClass1, self.sizeClass2)
freqT_ext.run_ext(_rootIDs_dict, report)
```

This approach is unconventional and inefficient.

4.2.3 New approach

In order to solve the mentioned issues, *FreqT.py* and *FreqT_ext.py* has been split in different classes and files:

- *FreqTCore*: This class is used for the main portion of the FREQT algorithm (common to every variant). It includes the whole process of expanding patterns. The data initialization and the outputting of patterns are defined as abstract methods. Each variant of the FREQTALS algorithm should extend this class.
- *FreqT1Class*: This class implements the FREQT algorithm with constraints **C0-C6**.
- *FreqT1Class2Step*: This class implements the first step of the FREQTALS algorithm.
- *FreqT1ClassExt*: This class implements the second step of the FREQTALS algorithm. It's called at the end of *FreqT1Class2Step* implementation.
- *FreqT2Class*, *FreqT2Class2Step*, *FreqT2ClassExt*: are the equivalent of respectively *FreqT1Class*, *FreqT1Class2Step*, *FreqT1ClassExt*, designed for databases which define 2 classes of ASTs.

- *InitData*: This file groups functions used to initialise the data. *init_data_1class(config) / init_data_2class(config)* are respectively used to initialise databases with 1 class/2 classes
- *AddTree*: This file groups static methods used to compile patterns.
- *FreqTStrategy*: This class was created in an effort to make constraints more flexible. It follows the so called *Strategy* design pattern [6]. Each FREQTALS variants have an *FreqTStrategy* object which defines their specific constraints. *FreqTStrategy* defines 4 functions :
  - *allowed_label_as_root(self, label)*
    which defines whether *label* is allowed as root label.
  - *is_pruned_pattern(self, pattern, candidate_prefix)*
    which defines whether *pattern* can be pruned and thus ignored during the expansion of patterns (*candidate_prefix* is the extension that has been added to the pattern).
  - *stop_expand_pattern(self, pattern)*
    which defines whether we should stop the expansion of *pattern*.
  - *satisfy_post_expansion_constraint(self, pattern)*
    which defines whether *pattern* satisfies the post expansion constraints. It is called just before the addition of *pattern* to the output.

  Whenever a constraint has to be checked, FREQTALS variants should call those functions.
  In addition, new sets of constraint can be easily implemented by implementing the *FreqTStrategy* class.

Fig. 4.1 summarize the class hierarchy between the variants of the FREQTALS algorithm.

This approach avoids excessive use of conditional clauses and allows practical implementation of new variants of the original algorithm (by extension of FreqT classes). New set of constraint can be defined by extending *FreqTStrategy*.

To illustrate the gain of this approach, we will be presenting the implementation of *FreqT1ClassExt*. This implementation is very similar to *FreqT1Class* but differs by the way it is initialised and the way *expand_pattern()* is called (see section 3.2). Accordingly, *FreqT1ClassExt* extends *FreqT1Class*. Inside its definition, the functions *__init__()* and *run()* has been modified (which well reflect the differences between the 2 classes):

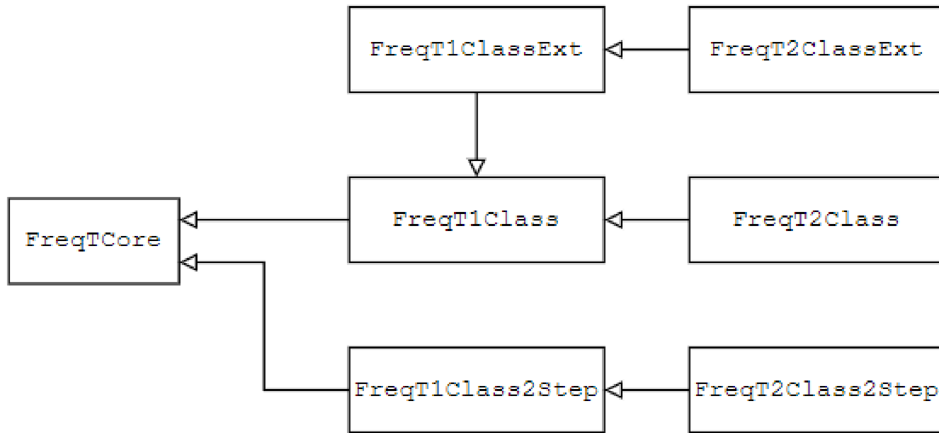

Figure 4.1: Hierarchy between FreqT classes

- **__init__:** During the second step of the FREQTALS algorithm, we are provided with sets of root occurrences on which patterns should be grown on. *FreqT1ClassExt* receives this already processed data during its initialisation.

  In addition, we expect *FreqT1ClassExt* to drop the *maximum size constraint* **C2**, so that we can grow large patterns. Accordingly, it uses a *FreqT1ExtStrategy* object (implementation of *FreqTStrategy*). This object allows to verify every constraint but ignores **C2**.

- **run():** During the second step of the FREQTALS algorithm, we grow large patterns for each set of root occurrences provided. To do so, we build their corresponding sets of *root patterns.* the expansion function *expand_pattern()* (provide by *FreqTCore*) is subsequently called for each set of *root patterns.*

  Note that the allocated time is shared between each call of *expand_pattern().* The timeout function thus needs revision.

```
class FreqT1ClassExt(FreqT1Class):

def __init__(self, _config, root_ids_list, _grammar_dict, _grammarInt_dict, _xmlCharacters_dict, label_decoder,
             _transaction_list):
    super().__init__(_config)

    self.root_ids_list = root_ids_list

    self.constraints = FreqT1ExtStrategy(_config, _grammarInt_dict)

    self._transaction_list = _transaction_list
    self._grammar_dict = _grammar_dict
    self._xmlCharacters_dict = _xmlCharacters_dict
```

```
        self.label_decoder = label_decoder

        # -- FreqTExt timeout variable --
        self.__interruptedRootIDs = None

    def run(self):
        """ ... """
        # set running time for the second steps
        self.set_starting_time()
        timeout_step2 = self.time_start + self._config.getTimeout() * 60

        while len(self.root_ids_list) != 0:
            # note : each group of rootID has a running time budget "timePerGroup"
            #        if a group cannot finish search in the given time
            #        this group will be stored in the "interruptedRootID"
            #        after passing over all groups in rootIDs, if still having time budget
            #        the algorithm will continue exploring patterns from groups stored in interruptedRootID

            # calculate running time for each group in the current round
            remaining_time = timeout_step2 - time.time()
            if remaining_time <= 0:
                break
            time_per_group = remaining_time / len(self.root_ids_list)

            self.__interruptedRootIDs = list()

            for elem in self.root_ids_list:
                # start expanding a group of rootID
                self.set_timeout(time_per_group)
                root_pat, occ = elem
                proj = self.getProjected(occ)

                # keep current pattern and location if this group cannot finish
                interrupted_pattern = root_pat
                # expand the current root occurrences to find maximal patterns
                self.expand_pattern(FTArray.make_root_pattern(root_pat), proj)

                if not self.finished:
                    self.__interruptedRootIDs.append((interrupted_pattern, occ))

            # update lists of root occurrences for next round
            self.root_ids_list = self.__interruptedRootIDs

        # print the largest patterns
        if len(self.mfp) != 0:
            self.output_patterns(self.mfp, self._config, self._grammar_dict, self.label_decoder,
                                 self._xmlCharacters_dict)

    def getProjected(self, root_occ):
        # create location for the current pattern
        proj = Projected()
        proj.set_depth(0)
        for loc, root in root_occ:
            proj.add(Location(root, root, loc, 1))
        return proj

    # --- TIMEOUT --- #

    def set_starting_time(self):
        """
         * set time to begin a run
```

```
        """
        self.time_start = time.time()

    def set_timeout(self, budget_time):
        self.finished = True
        self.timeout = time.time() + budget_time
```

4.3 Data structure

This section will cover the main data structures involved in the FREQTALS algorithm and will justify their choice.

4.3.1 FTArray

FTArray is primarily used in the algorithm to represent patterns. It is implemented as an *extensible array* of integers. Each entry ($\neq -1$) of this array corresponds to one node of the pattern. They are listed in the order of *depth first* with *left-right* traversal, additionally -1 tokens indicate backtracking inside the tree structure. As an example, the array $[1, 2, -4, -1, -1, 3, -5, -1, 6]$ corresponds to the pattern in Fig. 4.2. Node labels are identified using integers, negative integers indicate that their label corresponds to a leaf in the database, useful when checking leaf related constraints (note that -1 is a reserved value and doesn't correspond to a leaf label).

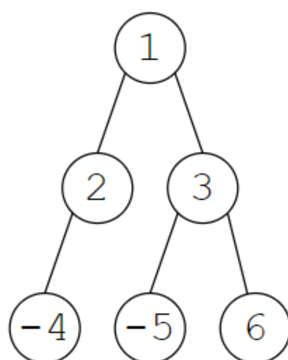

Figure 4.2: Pattern corresponding to the representation $[1, 2, -4, -1, -1, 3, -5, -1, 6]$

The *FTArray* class has been improved to better fit its purpose and improve performance:

- *FTArray* used to have 2 arrays: a primary one of size 512, and a secondary one not yet initialised. Whenever the capacity of the first array was exceeded, its size was increased by 512. The first array continues extending until element outside the range of short was encounter. The content was subsequently migrated to the second array. Note that this migration of array contents doesn't make sense, since integers in Python effectively has no bound. It was used in Java to store shorts if possible, otherwise we were using integers. This approach is overall complex and can be adapted using *lists* from the

built-in Python library, since *lists* already are implementation of *extensible arrays*. A major portion of the *FTArray* implementation has been simplified using a single *list*.

- In order to satisfy **C1** and **C2**, the number of nodes and leaf labelled nodes had to be recomputed for newly encountered patterns. And thus the whole pattern had to be traversed.

  To address this issue, *FTArray* now stores 2 additional fields n_node, n_leaf which respectively correspond to the number of nodes and leaf labelled nodes contained in the pattern. They are updated accordingly when the *FTArray* is altered. This operation is trivial: during the expansion of pattern, whenever we add a node to a pattern we increment n_node. And if this node has a *leaf label*, n_leaf is incremented.

- Since patterns are stored as keys in a dictionary, it is essential to implement functions *__hash__()* and *__eq__()*, which were missing.

- Previously, *FTArray* objects were copied using *deepcopy()*. But *deepcopy()* is usually inefficient since it allows to copy any object. Instead, in order to make a copy of an *FTArray*, the list stored inside is copied (using *list.copy()*) and used in the creation of a new *FTArray*.

- Pattern related functions like *find_children_position(self, parent_pos)* (used to return the children of some node in a pattern) which were defined as static functions are now included inside *FTArray* definition.

4.3.2 Pattern extension

When expanding one pattern, the algorithm first has to generate a set of extended versions of this pattern. But computing the whole pattern for every candidate found would be computationally inefficient. Since lots of them are likely to be pruned by **C0**. Only the extension added on top of the expanding pattern has to be computed.

Those extensions were previously implemented using *FTArrays* which are then appended to the expanding pattern to compute new candidate patterns. But, due to the way pattern are represented, some extension can contain many -1 (tokens indicating backtracking).

Pattern extensions data structure have been simplify using a *tuple* with 2 entries:

- The number of backtracking we have to made inside the pattern before adding a new node

- The label of the new added node

This is illustrated in Fig. 4.3.

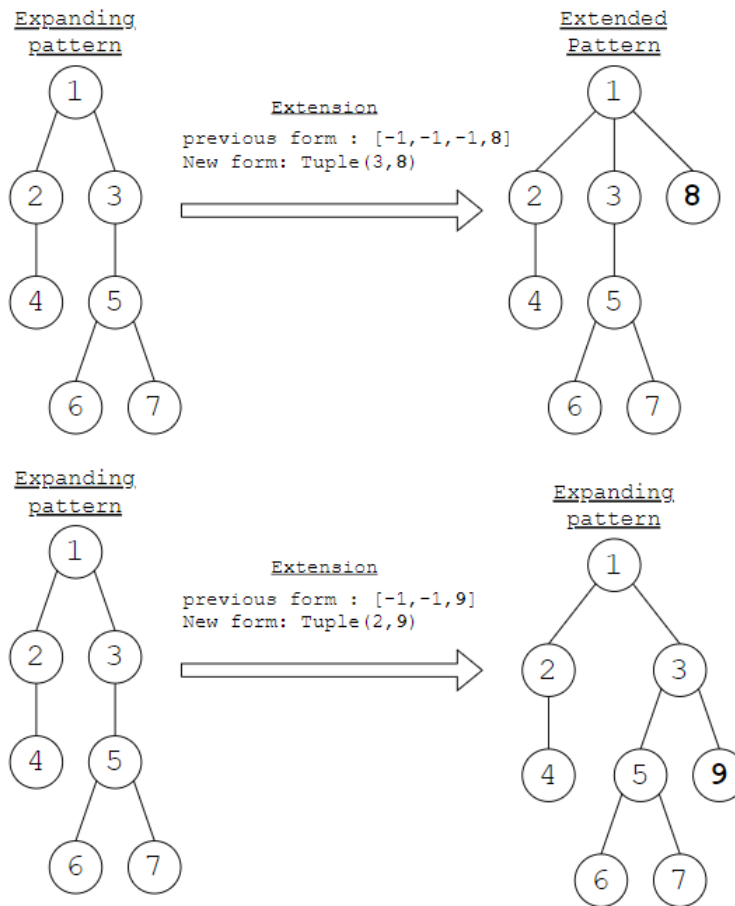

Figure 4.3: Examples of the addition of extension on patterns

4.3.3 Projected and Location

During the generation of *candidate patterns*, we compute extended versions of patterns and their projection on the database. Projections are implemented by the *Projected* class. This object stores every occurrence of an extended version of a pattern and computes its support. Those occurrences are implemented by *Location* class.

The *Location* class used to extend FTArray. It stores the *root pattern* occurrence it originates from, its rightmost node, the id and class of the tree from the database it is located at.

However, it is unnecessary to store an entire *root pattern* array as we know that it only contains one node. In section 4.3.1, we even saw that *FTArray* used to hold a array of size 512. This means that when a new location was created, an array of size 512 was copied. This is of course inefficient.

Instead, *Location* now implements its own class which consists of 4 integer entries for the root node, the rightmost node, the id and class of the tree (from the database) the location corresponds to.

4.4 Implementation correction

Expand_pattern() is the function used to browse and expand patterns, see Algorithm 5 (implemented inside *FreqTCore.py*). For a given pattern, it generates a set of candidates, each of which are pruned and then expanded. Patterns that met every constraint are added to the output.
One of those constraints **C6** ensures that every leaf of the pattern is also a leaf in the data.

Previously, when such pattern was encountered it was stored. Whenever a pattern couldn't be expanded anymore, the last stored parent pattern was added to the output, see Algorithm 6. However, this implementation has many downsides:

- "Leaf patterns" are very likely to be added multiple times to the output since whenever a pattern stops expanding, its closest leaf pattern parent is added to the output. We thus have to ensure that we are not adding duplicates, see Fig. 4.4.
  In addition, depending on the implementation, we either have to ensure that outputted patterns are not subtree of other patterns or ensure that outputted sets of root occurrences are not super-set of other sets. In both cases, this requires the output to be scanned every time a new pattern is added. To solve this issue, the previous implementation maintains a dictionary *not_maximal_set* containing already encountered patterns that should not be added anymore. Note that the post expansion constraints are checked after storing the *leaf pattern*. It means that those constraints are checked multiple times on the same pattern.

- In order to save memory space, instead of creating new FTArray for every candidate generated, the implementation uses a single FTArray which represents the current pattern treated by the algorithm. It can subsequently be copied in order to be added to the output. However, here we are storing leaf patterns before any addition to the output. We thus have to make a copy of each leaf pattern, even if they are eventually not added to the output.

Algorithm 6 expand_pattern() - Previous implementation

```
1: function EXPAND_PATTERN(pattern)
2:     candidates_set ← generate_candidates(pattern)
3:     min_support_prune(candidates_set)
4:     if len(candidates_set) == 0 then
5:         leaf_pattern ← get_stored_pattern()
6:         if satisfy_post_expansion_const(leaf_pattern) then
7:             add(leaf_pattern)

8:     for c in candidates_set do
9:         candidate ← pattern.extend(c)        ▷ add an extension to the pattern
10:        if candidate is leaf pattern then
11:            store(candidate)                  ▷ leaf pattern found and stored
12:        old_leaf_pattern ← get_stored_pattern().copy()
13:        if candidate cannot be pruned then
14:            if stop expansion criteria met then
15:                leaf_pattern ← get_stored_pattern()
16:                if satisfy_post_expansion_const(leaf_pattern) then
17:                    add(leaf_pattern)
18:            else
19:                expand_pattern(candidate)
20:        store(old_leaf_pattern)              ▷ need to restore the previous state
21:        pattern.undo(c)                      ▷ remove the extension
```

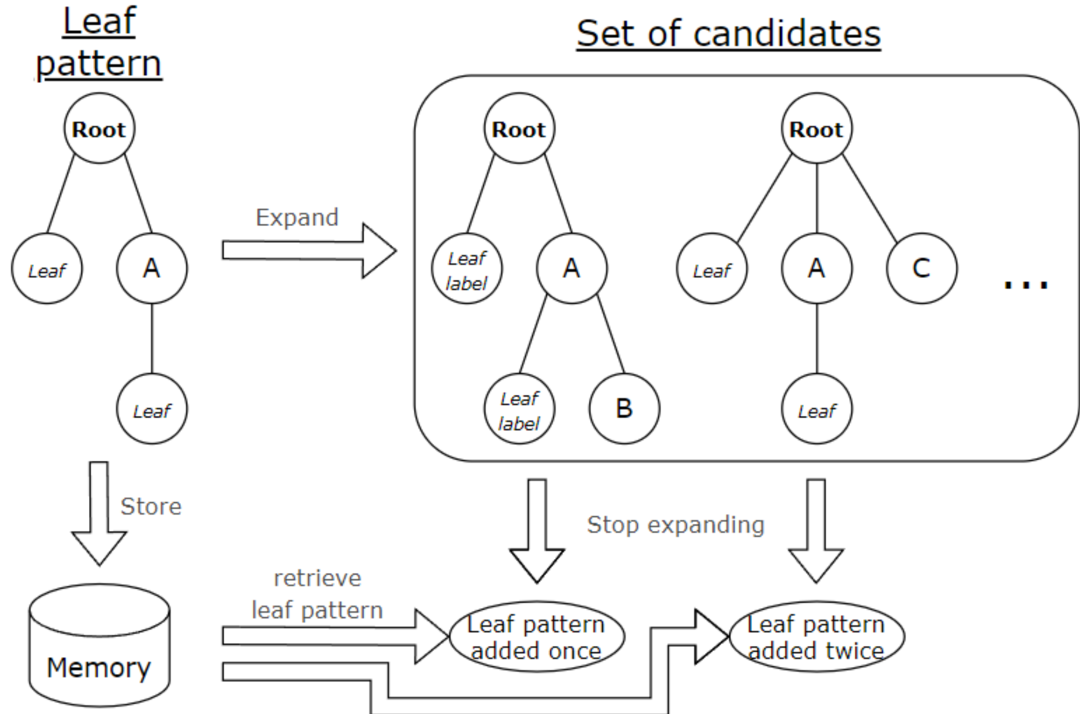

Figure 4.4: Previous approach adds duplicate to the output

> In addition, this approach requires to reset the state of the stored leaf pattern at the end of each *expand_pattern()* call, otherwise *get_stored_pattern*() may not return the correct pattern. We thus have to make additional copies of patterns.

The new implementation aims at reducing the number of times that the algorithm attempts to add patterns to the output.
The first idea is that we can notify parent patterns whenever one of its children has stopped expanding.

When a pattern stops expanding, it returns an *add request* (Boolean). This request is passed to parents until a "leaf pattern" receives it. This *leaf pattern* consumes the request and is added to the output, see Algorithm 7.

This implementation avoids storing *leaf patterns*, and requires a single copy for each pattern stored in the output. *Leaf pattern* encountered are now added only once to the output. We thus don't have to check for duplicates anymore.

The second idea come from the fact that we are mining for *maximal frequent subtree*. We thus want to avoid storing frequent patterns which are subtrees of other frequent patterns. And we note that, in the tree of generated patterns, a

Algorithm 7 expand_pattern() - First idea implementation

```
1: function EXPAND_PATTERN(pattern)
2:     candidates_set ← generate_candidates(pattern)
3:     min_support_prune(candidates_set)
4:     if len(candidates_set) == 0 then
5:         return true                                   ▷ generate add request
6:     current_request = false
7:     for c in candidates_set do
8:         candidate ← pattern.extend(c)
9:         if candidate cannot be pruned then
10:            if stop expansion criteria met then
11:                if candidate is leaf pattern then
12:                    try_add(candidate)
13:                else
14:                    current_request = true            ▷ generate add request
15:            else
16:                child_request = expand_pattern(candidate)
17:                if child_request then                 ▷ add request received
18:                    if candidate is leaf pattern then
19:                        try_add(candidate)
20:                    else
21:                        current_request = true        ▷ preserve add request
22:        pattern.undo(c)
23:    return current_request

24: function TRY_ADD(pattern)
25:    if satisfy_post_expansion_const(leaf_pattern) then
26:        add(pattern.copy())
```

parent is always subtree of its children.

So whenever a pattern is successfully added to the output, its parent patterns shouldn't be added. This can be easily implemented by notifying parent patterns whether one of its children has been added successfully to the output, see Algorithm 8.

Algorithm 8 expand_pattern() - New implementation

```
1:  function EXPAND_PATTERN(pattern)
2:      candidates_set ← generate_candidates(pattern)
3:      min_support_prune(candidates_set)
4:      child_pattern_added = false
5:      for c in candidates_set do
6:          candidate ← pattern.extend(c)
7:          if candidate cannot be pruned then
8:              if stop expansion criteria met then
9:                  if candidate is leaf pattern then
10:                     if try_add(candidate) then
11:                         child_pattern_added = true     ▷ notify parent: "pattern added"
12:             else
13:                 did_add_pattern = expand_pattern(candidate)
14:                 if did_add_pattern then                 ▷ notification received
15:                     child_pattern_added = true          ▷ preserve notification
16:                 else if candidate is leaf pattern then
17:                     if try_add(candidate) then
18:                         child_pattern_added = true     ▷ notify parent: "pattern added"
19:         pattern.undo(c)
20:     return child_pattern_added

21: function TRY_ADD(pattern)
22:     if satisfy_post_expansion_const(leaf_pattern) then
23:         add(pattern.copy())
24:         return whether add() has succeeded
25:     return false
```

4.5 Additional change

In this section, we will cover a set of smaller improvements. They contribute to the readability and efficiency of the implementation but require less explanation.

4.5.1 Excessive use of *try except*

Through the code, many function contents were defined inside *try except* clauses. Many of them were unnecessary as the function couldn't fail. In addition, since Python is an interpreted language, errors are detected at running time. Since most of them were caught, it makes the identification of implementation mistakes harder.

The deletion of most *try except* clauses helps the clarity of the code, but also is more convenient in case of implementation mistakes.

4.5.2 Support computed multiple times

For the *FreqTals 2 steps 2 classes* implementation, we are interested to compute the chi-square score of patterns found [8]. To perform this task we have to compute the support of patterns on each class of trees from the database. However this operation was done 3 times in the previous version of the implementation. Which is of course inefficient and has been fixed.

Chapter 5

Validation

The maintenance of programs is essential. Through this process we are improving the code on many different aspects. It is a good practice to make code more convenient. This is the main goal aimed by this thesis.

In total, **14685** lines of code were added and **14927** were removed. In order to validate these changes made, we will cover the different aspects of maintenance: readability, maintainability, reusability, correctness and efficiency. For each of them, we will recall how the implementation was updated in their regards and justify why those modifications are improvements.

5.1 Readability

Readability is the main goal of this thesis. This aspect aims at making the implementation easier to read/understand. Readability is essential for future works on this FREQTALS implementation. We can spend less time understanding the implementation if the code is easier to read and thus gain time on FREQTALS related projects. In addition, with a better understanding of the code we are less likely to make mistakes. But, this aspect is in practice difficult to validate as this topic is subjective. The time to understand the implementation varies depending of the actual person reading the implementation.

One of the main way of enhancing readability is via the use of comments. Comments can provide useful explanations about how functions work and help identifying their purpose. It can also provide context about the usage of certain functions. Overall comments are a great medium for improving readability. In section 4.1.1, we have seen that many descriptions for functions were revised. In addition, the arguments of functions have been clearly identified.

But readability is not bounded to the presence of comments. In fact, the naming of variables, the segmentation of code in smaller functions and other good implementation practices ... help to give structure to the code. In order to assess the implementation style, we will be analysing the source code with the *pylint* library [11] in a similar way as *Loïc Quinet* did on the previous version [7].

pylint library provides tools to analyse written code and helps enforcing standards. It can also be used to identify complex/bad implementation style. *pylint* grades source files on a scale of ten. The current implementation of the FREQTALS algorithm [12] obtains **6.82/10**.

5.2 Maintainability

Maintainability reflects how easy it is to make modifications and correct mistakes in an application. This aspect was the primary reason why the FREQTALS algorithm was originally adapted in Python [7]. The Python language is known to be a language easy to learn and easy to implement, however it is less efficient than most programming languages. The Python implementation was not meant to be more efficient but aims at being practical and easy to adapt. To achieve this goal, it requires a good readability. Indeed, readability hugely contributes to the maintainability of a program. Since a better understanding of the code usually means a better capability to make modifications and identify/fix mistakes.

But maintainability can also be enhanced thanks to the structure of the classes. In fact, in chapter 4.2, we have seen how the variants of FREQTALS algoritm have been separated in their own classes. This structure allows easy modification to them independently of each other, and thus increase the effectiveness of such modifications.

In addition, the constraints provided by the algorithm have received a similar treatment, inspired from the *Strategy* design pattern [6]. The *FreqTStrategy* class groups them by their usage. This allows to easily identify their impact on the implementation.

5.3 Reusability

The reusability is the ability of the code to be used/adapted for different projects/applications. Even if the FREQTALS [5] was originally designed for mining ASTs, it was implemented as a tools for *Labelled, ordered, and rooted trees.* Indeed, this *constraint-based* algorithm as the advantage of being easy to configure. The new structure of the implementation (cfr. chapter 4.2) further improves this aspect.

In fact, extended versions of the FREQTALS algorithm can be derived by implementing the *FreqTCore* class. This approach is convenient as it offers an interface for implementing:

- The way patterns are added to the output (*add_tree*).
- The initialisation and pre-processing of the database (*init_data*).

In addition, new set of constraints can easily be adapted by extension of the *FreqTStrategy* class. This class sorts constraints by their actual usage inside the algorithm. For example, new anti-monotonic constraints could be added to the *is_pruned_pattern()* which is used to effectively prune the search space.

5.4 Efficiency

During this thesis, we have mainly focused on making the code easier to read and to use. But we have seen that a set of changes could be made to improve the execution time of the FREQTALS algorithm. In order to assess the gain of those changes, we will be using the Python profiler *cProfile* [10]. This library provides statistics on how long and how frequent are individual functions called during the execution of programs. In addition to being a medium for assessment, it really helped identifying critical sections of the implementation.

Among the statistics provided by *cProfile*, we will focus on 3 of them:

- *ncalls:* It corresponds to the number of time a function is called. This number includes primitive and recursive calls.
- *tottime:* It corresponds to the total amount of time (in second) that the program has spent inside a function. It reflects how the execution time is shared between functions.
- *cumtime:* It corresponds to the *cumulative time* (in second) spent in a function and its sub-function. *cumulative time* is equal to the sum of *tottime* of the function and the *tottime* of every sub-function invoked.

In this section, using those statistics, we will compare functions from the previous implementation of the FREQTALS algorithm [7] against their equivalent (if there exist) from the improved implementation [12]. The performance of each implementation was recorded on the test provided in *Test_Main_one_step.py* file.

Previous implementation				Improved implementation			
name	ncalls	tottime	cumtime	name	ncalls	tottime	cumtime
run	1	0.000	8.770	run	1	0.000	0.314
expandPattern	981	0.030	8.547	expand_pattern	981	0.006	0.132
generateCandidates	981	0.051	6.496	generate_candidates	981	0.024	0.049
updateCandidates	10368	0.029	6.419	update_candidates	10394	0.006	0.017
location2	10368	0.009	6.043	Location.__init__	11016	0.003	0.003
keepLeafPattern	2903	0.005	1.699	*no equivalent*			
ftarray	13283	0.027	7.718	FTArray.copy	19	0.000	0.000
deepcopy	6814179	4.069	7.688	*no equivalent*			

Table 5.1: impact of *copy*

Impact of copying *FTArrays*

The Main performance loss comes from copying excessively FTArrays during the expansion of patterns. The copy function of FTArray (*ftarray*) has a time complexity of $\mathbf{O(n)}$ which is not particularly constraining. However, if called too many times, it can significantly slow the program. In fact, on Tab. 5.1, we can see that this function takes by itself around **88%** of the total running time (= cumtime of *run*). It was solved by significantly reducing the number of copies made during the expansion of patterns (*expandPattern*):

- In section 4.3.3, we saw that *Location* is not required to store an array of size 512. Creating such array for each new *Location* is very inefficient.

  For this example, the creation of those *Locations* (implemented by *location2*) used to represent **68.9%** of the total running time. This improvement prevents **10368** *FTArray* from being copied which represents **78.1%** of the previous total number of copies.

- In section 4.4, we saw how we could avoid storing *leaf patterns* (*keepLeafPattern*) during the expansion phase.

  For this example, this operation used to take **19.3%** of the total running time. This improvement prevents **2903** *FTArray* from being copied which represent **21.8%** of the previous total number of copies.

As a result, only **0.1%** of the original number of copies is used by the improved implementation. Those remaining copies are utilized when patterns have to be copied to the output.

Remark: As this performance gain is so significant, we will consider the *total running time without copies* = cumtime *run* - cumtime *ftarray*. This metric allows to better visualise the impact of the other improvements.

Computation of the number of nodes and leaf nodes

Previous implementation				Improved implementation			
name	ncalls	tottime	cumtime	name	ncalls	tottime	cumtime
satisfyMinNode	1140	0.000	0.011	satisfy_min_node	231	0.000	0.000
satisfyMinLeaf	1232	0.001	0.012	satisfy_min_leaf	233	0.000	0.000
satisfyMaxLeaf	2245	0.001	0.026	satisfy_max_leaf	2245	0.000	0.000
countLeafNode	3477	0.019	0.036	*no equivalent*			
countNode	1177	0.006	0.011	*no equivalent*			

Table 5.2: Improvement on the computation of the number of nodes/leaf nodes

We saw in section 4.3.1 that a smarter approach to check constraints related to the size of a pattern is to gradually compute its number of nodes and leaf nodes while extension are being added on top of this pattern. Previously, the implementation used to count the number of nodes (*countNode*) and the number of leaf nodes (*countLeafNode*) in pattern every time considered. While these operations require a single traversal through the whole pattern and thus have a complexity of $\mathbf{O(n)}$, the new approach can update and retrieve those statistics in $\mathbf{O(1)}$. This improvement saves **36.3%** of the *total running time without copies*.

Note that, on Tab. 5.2, the number of times *satisfy_min_node* and *satisfy_min_leaf* are called has decreased. This is due to the fact that less patterns are considered for addition to the output, see section 4.4.

Computation of ***supports***

Previous implementation				Improved implementation			
name	ncalls	tottime	cumtime	name	ncalls	tottime	cumtime
getSupport	3334	0.010	0.015	compute_support	3199	0.002	0.005
getRootSupport	3334	0.010	0.019	compute_root_support	3064	0.002	0.008

Table 5.3: Improvement on the computation of supports

We saw in section 4.1.2 that built-in functions in Python are usually faster than re-implementing them. One pertinent example of such fact is the computation of the support (which was already illustrated). On Tab. 5.3 we observe that, for this example, those functions *compute_support*, *compute_root_support* run respectively **66%**, **56%** faster.

This improvement saves **11.4%** of the *total running time without copies*.

Function ***expand_pattern*** assessment

In section 4.4, we have discuss 3 different implementations of *expand_pattern*:

- **impl. a:** the previous implementation of *expand_pattern()* which stores *leaf patterns*.
- **impl. b:** this implementation generates "addition to the output" request which could be consumed by other pattern (satisfying the constraint).
- **impl. c:** this implementation notifies patterns whether there are subtrees of other *frequent patterns* already encountered.

The primary intuition behind the improvement of *expand_pattern()* was to reduce the number of times we attempt to add pattern to the output. In fact when doing so, we additionally have to identify and remove *non-maximal* patterns. In order to compare those implementations, we introduce 4 statistics:

- tot_pat_visited: the total number of non-pruned pattern generated.
- n_post_exp_const: This corresponds to the number of times we consider a frequent pattern for addition to the output. It is equal to the number of times the post expansion constraints (**C2**, **C5**, **C6**) are checked.
- n_add_tree: This corresponds to the number of times we have attempted to add a new frequent pattern to the output. It is equal to the number of *add_tree()* call

- add_maximal_set_hit: We used to maintain a set of patterns which have been encountered and shouldn't be added to the output anymore (denoted *non_maximal_set*). This statistic corresponds to the number of times *add_tree()* was called but failed due to the pattern being already included in *non_maximal_set*.

In order to highlight the differences between implementations, the maximum number of leafs was raised from 4 to 9. The algorithm thus grows significantly more patterns.

	impl. a	impl. b	impl. c
Output size	25	25	25
cumtime	1.406	0.210	0.179
tot_pat_visited	86016	86016	86016
n_post_exp_const	36187	8724	8625
n_add_tree	322	54	32
non_maximal_set_hit	126	0	0

Table 5.4: Comparison of *expand_pattern* implementation

Let us first note that every implementation returns the same output. In addition, we observe (Tab. 5.4) that they visit the same number of patterns, and thus most likely visit the same patterns. In fact, the optimisations made don't alter the way patterns are visited but the way patterns are considered for addition to the output.

Impl. b and *impl. c* decrease significantly the number of patterns which are considered for addition, respectively by **76%** and **78%**. The algorithm thus requires less verifications whether patterns satisfy the *post expansion constraints*.

Note that *impl. b* and *impl. c* have also the advantage of not using the *non_maximal_set* (thus saving memory).

5.5 Correctness

One essential aspect to evaluate is the correctness of the implementation. It tells us whether the program performs the same task or returns an improved result (it for example could find more *frequent patterns* in our context). Since the changes made in chapter 4 didn't affect the output of the algorithm, we expect the previous version and the new one to find the same set of *frequent patterns*.

During this thesis, when a portion of the implementation was modify, 2 test files were executed to ensure that these changes didn't affect the end result. Those files have been implemented in a previous work [7] on FREQTALS:

- *Test_Main_one_step.py*: This test file ensures the correct working of the naive approach *FreqT 1 step.*
- *Test_Main_two_step.py*: This test file ensures the correct working of *FreqT 2 steps* approach.

Test file	Prev. running time(s)	Impr. running time(s)	Output size	Same output
Test_Main_one_step.py	8.770	0.314	13	yes
Test_Main_two_step.py	5.280	0.249	3	yes

Table 5.5: Results default test suite

We observe on Tab. 5.5 that the results are coherent with the previous version of the implementation. In addition, we note that the improved version is way faster as mentioned earlier.

However, those tests are relatively small, as the databases they are executed on are small as well. Accordingly, some additional tests have been defined a real case study (the grammar of *antlr* library) to further assess the correctness of the new version of the implementation. n_{pat_gene} denote the number of times we have generated sets of candidate patterns (*generate_candidates*). We observe on Tab.

Text	Database	min support	min node	min leaf	max leaf	algorithm
test1	antlr/grammar/v3	3	10	1	5	*FreqTals 1 step*
test2	antlr/grammar/v3	2	10	1	5	*FreqTals 1 step*
test3	antlr/grammar/v3	3	10	1	4	*FreqTals 2 step*
test4	antlr/grammar/v2 antlr/grammar/v3	3	10	2	5	*FreqTals 1 step*

	Previous implementation			Improved implementation		
Test	n_{pat_gene}	Running time(s)	Output size	n_{pat_gene}	Running time(s)	Output size
test1	493	165.866	4	493	14.840	4
test2	9598	*timeout*	3	79857	262.223	63
test3	685	220.728	4	685	14.927	4
test4	n/a			114791	*timeout*	1

Table 5.6: Result on real study cases

5.6, that both implementations return the same output if the execution didn't timeout. In fact, in test2, we note that the previous implementation runs in more than 10 minutes. the improved implementation is able to generate candidates faster and is thus can tackle bigger search space. During its execution, the previous implementation is able to find 3 patterns. The improved version also finds these 3 patterns in addition to 60 other (which satisfy all the constraints).

Chapter 6

Possible improvement

Some possible improvements were noticed but was not explored or discuss enough due to the time constraint. In this chapter, we will propose points of interest which could be analysed in more depth.

6.1 Pre-processing step

During a previous study [9], it was mentioned that the way **C4** was implemented could lead to the miscalculation of patterns *support*.

In fact, whenever the algorithm encounters a blacklisted label, the underlying branch inside the tree was ignored. This approach effectively reduces the size of the input and thus the search space. By ignoring portion of the database, we are ignoring some pattern occurrences which leads to the miscalculation of the *support*.

But, since it is *maximal frequent patterns* we are mining for, we are only interested to correctly compute the *support* of this patterns. Accordingly, we can prune underlying branches of blacklisted node if they cannot contain a least one pattern satisfying the constraints. For example, inside such branches, if there exist no patterns which satisfy the minimum size constraint *C2*, the branches can be accordingly ignored during the preprocessing step. This is illustrated on Fig. 6.1.

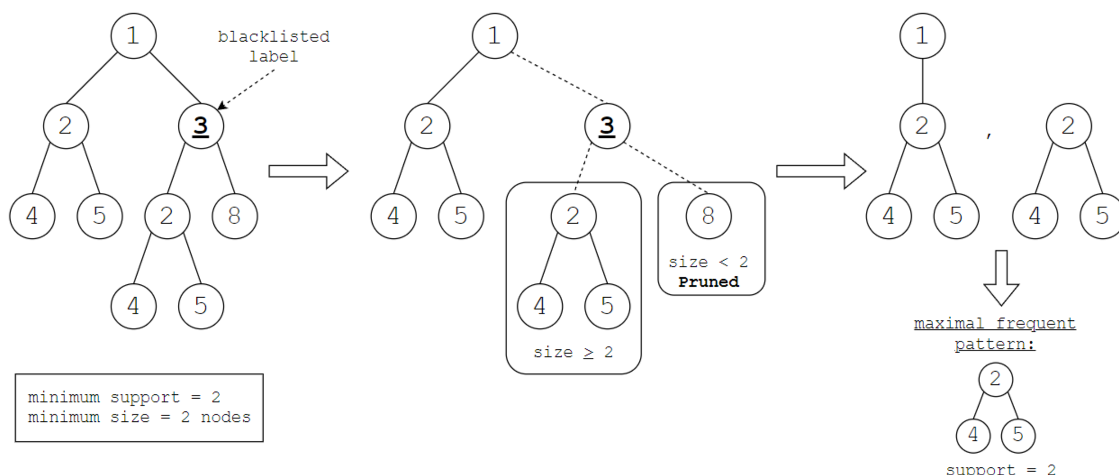

Figure 6.1: Illustration of the impact of preprocessing on support

6.2 Simpler approach for *FreqTSubtree*

The *FreqTSubtree* class is used to check for sub-tree relationship between patterns whenever the naive/quick method fails.

To verify whether T_1 is a subtree of T_2, *FreqTSubtree* generates patterns using the FREQT algorithm with support threshold of 2. The idea being that if we found T_2 as a frequent pattern, we can conclude that $T_1 \succ T_2$.

However, this approach may consider patterns which miss nodes of the small tree T_2 in the order of left to right depth-first traversal. Such patterns are not useful as they cannot be extended into T_2, since we are extending to the right of the rightmost path. In fact there exists only one way (ordered set of extensions) for a pattern to be generated by the algorithm (see section 2.3).

An other approach gradually builds T_2 by addition of extension, while computing its set of occurrences inside T_1. If one of the gradually built patterns has a *support* of 0, we can conclude that $T_1 \not\succ T_2$. This method doesn't require minimum support pruning.

6.3 Mandatory child constraint C5 and label of leaf nodes constraint C6 used for pruning

In section 3.1, we saw how the FREQTALS algorithm handles the *mandatory child constraint*. In particular, *left mandatory child constraint* can be used to prune the search space. It detects whether a node has been added to the pattern too early and thus at least one of the mandatory children of the parent node cannot be included anymore (see Fig 3.2). But, during the addition of an new extension to a pattern, if the number of time we backtrack (-1 tokens) in the pattern is non-zero, every node we have backtracked on won't be able to receive children anymore (due to the rightmost path generation method). We can thus check whether the set of children agrees with the provided grammar (see Fig. 6.2). This could be used to prune many patterns.

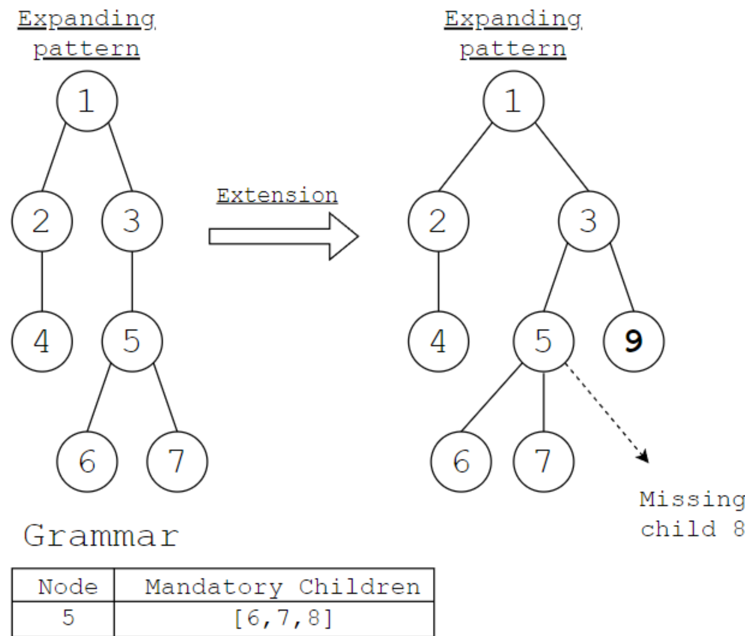

Figure 6.2: Potential additional pruning using **C5**

In the same mind set, if the number of time we backtrack (-1 tokens) in the pattern is non-zero, the previous rightmost node won't be able to receive children anymore. According to constraint **C6**, if this (rightmost) node has not a label corresponding to a leaf in the database, this pattern should be ignored (see Fig. 6.3).

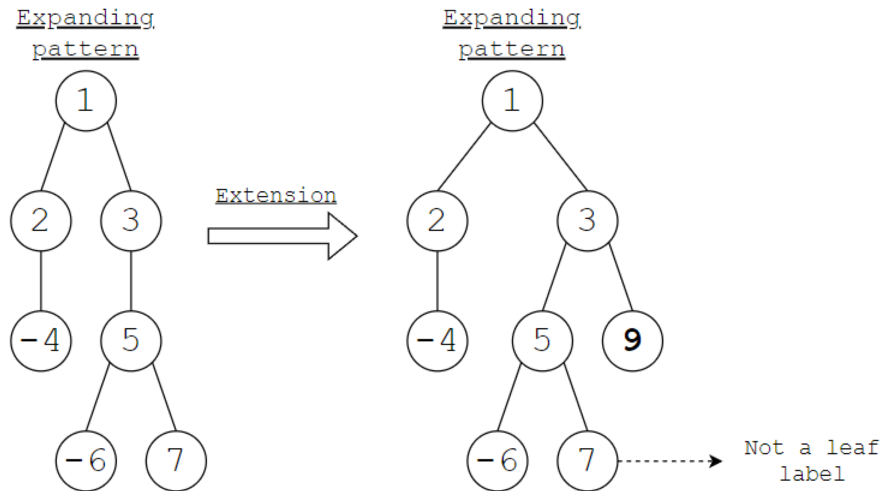

Figure 6.3: Potential additional pruning using **C6**

In theory, this helps reducing the search space and thus greatly improve performance. The impact of such modifications could be explored in more depth.

Chapter 7

Conclusion

In this thesis, we tried to improve the previous Python implementation of the FREQTALS algorithm [7]. The main goal being to obtain a code more convenient and easier to maintain.

We have discussed how the enhancement of comments and function descriptions makes individual functions easier to understand. However the readability of source code isn't limited to good comments. Indeed, many portions of the source code have been simplify using the Python framework like the way patterns are expanding, the internal working of *FTArray*, ... and we have build a new structure to the implementation which clearly separates the different variants of the FREQTALS algorithm but also allows practical implementation of new variants (with a new set of constraints for example).

Through the implementation of these modifications, we have identified/corrected mistakes which significantly were slowing the program. Mainly, the way new occurrences of patterns were computed was inefficient. The improved program now runs significantly faster.

Unfortunately, the wrong implementation of **C4** during the pre-processing of the database (mentioned in a previous study [9]) was not correctly implemented in time, but we have discussed the issue in detail as well as other potential improvements of the code (on the Python and Java version).

In conclusion, during this thesis we have improved the readability of the Python implementation. The performance gain allows more convenient testing. Even if some improvements could still be made, the performance of the FREQTALS algorithm was enhanced and allows efficient use in the Python framework.

Bibliography

[1] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa. “Efficient substructure discovery from large semi-structured data”. In: *IEICE TRANSACTIONS on Information and Systems* 87.12 (2004), pp. 2754–2763.

[2] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok. “Frequent subtree mining–an overview”. In: *Fundamenta Informaticae* 66.1-2 (2005), pp. 161–198.

[3] A. Jiménez, F. Berzal, and J. C. C. Talavera. “Frequent tree pattern mining: A survey”. In: *Intell. Data Anal.* 14.6 (2010), pp. 603–622.

[4] Clapham Christopher, Nicholson James. “Oxford Concise Dictionary of Mathematics (5th ed.)” In: *Oxford University Press* (2014).

[5] Hoang-Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, CoenDe Roover, Johan Fabry and Vadim Zaytsev. “Mining Patterns in Source Code using Tree Mining Algorithms”. In: *Proceedings of the 22nd International Conference on Discovery Science (DS2019).* (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 11828 LNAI). DS2019: 22nd International Conference on Discovery Science. (2019), pp. 471–480. DOI: https://doi.org/10.1007/978-3-030-33778-0_35.

[6] Alexander Shvets, Andrew Wetmore, Rhyan Solomon and Dmitry Zhart. *Dive into design patterns (v2021-2.28).* Refactoring.guru, 2021.

[7] Hauspie, Quentin. “Can Data Mining Discover Software Changes?” Ecole polytechnique de Louvain, Université catholique de Louvain. Prom. : Kim Mens, Siegfried Nijssen, 2021. URL: http://hdl.handle.net/2078.1/thesis:30664.

[8] Kim Mens, Siegfired Nijssen and Hoang-Son Pham. “The Good, the Bad, and the Ugly: Mining for Patterns in Student Source Code”. In: *In Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence (EASEAI’21), August 23, 2021, Athens, Greece. ACM, New York, NY, USA* (2021). DOI: https://doi.org/10.1145/3472673.3473958.

[9] Quinet, Loïc. “Using Python to Mine for Patterns in Software”. Ecole polytechnique de Louvain, Université catholique de Louvain. Prom. : Kim Mens, Siegfried Nijssen, 2021. URL: http://hdl.handle.net/2078.1/thesis:30540.

[10] Python. *cProfiler, deterministic profiler*. Version 3.10.4. URL: https://docs.python.org/3/library/profile.html.

[11] Python. *pylint, analysis tool for written source code*. Version 2.13.9. URL: https://pypi.org/project/pylint/.

[12] Quentin Hauspie, Arnaud Spits. *FREQTALS, improved Python implementation of the freqtals algorithm*. URL: https://github.com/Zmartic/FREQTALS---Thesis/.

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