UCLouvain

École polytechnique de Louvain

Using Python to Mine for Patterns in Software

Auteur: **Loïc Quinet**
Promoteurs: **Kim Mens, Siegfried Nijssen**
Lecteurs: **Guillaume Derval, Kim Mens, Siegfried Nijssen**
Année académique 2020–2021
Master [120] : ingénieur civil en informatique

Résumé

L'analyse des répétitions dans un code informatique peut être très intéressante lors l'apprentissage d'un nouveau langage ou lors de la compréhension d'un code ou encore lorsqu'il s'agit de localiser des erreurs. L'extraction de ces répétitions est possible à l'aide de l'algorithme *FREQTALS* implémenté en Java. Cet algorithme permet d'extraire des patterns fréquents tout en limitant leur nombre mais aussi en ne conservant que les plus utiles. Pour se faire, une série de contraintes ont été ajoutées au fil du temps. Ce code fonctionnel continue d'être développé aujourd'hui. Force est de constater que le code actuel n'est pas des plus simples à modifier. C'est pourquoi il m'a été proposé de le traduire en un code Python claire et facile à comprendre. L'objectif final est d'utiliser cette traduction comme laboratoire. Durant cette année, j'ai donc traduit cet algorithme mais aussi créé des tests afin de m'assurer de sa justesse. L'objectif de ce mémoire est d'avoir une traduction Python complète, fonctionnelle et claire. Ce mémoire reprend une explication de l'algorithme *FREQTALS*, une explication de son implémentation en Python ainsi qu'une analyse de sa qualité et de ses résultats.

The study of repetitions in computer code can be very interesting while learning a new language, understanding a code, or locating errors. These patterns can be extracted using the *FREQTALS* algorithm implemented in Java. This algorithm makes it possible to extract frequent patterns while limiting their number and keeping only the most useful ones. To do this, a series of constraints have been added over time. This functional code continues to be developed nowadays. However, the current code is not easy to change. That is why I was offered to translate it into a clear and easy to understand Python code. The final goal is to use this translation as a laboratory. So this year, I translated this algorithm, but I also created some tests to make sure it was accurate. The objective of this thesis is to have a complete, functional and clear Python translation. This thesis includes an explanation of the *FREQTALS* algorithm, an explanation of its implementation in Python, and an analysis of its quality and results.

Remerciements

Lors de la réalisation de ce mémoire, j'ai reçu l'aide et les conseils de nombreuses personnes. Je voudrais donc les remercier. Elles m'ont permis d'atteindre les objectifs fixés.

Tout d'abord, je voudrais remercier mes promoteurs, Monsieur Kim Mens et Monsieur Siegfried Nijssen, pour le temps consacré lors de nos réunions hebdomadaires. Ces dernières ont toujours été sérieuses, constructives et ont eu lieu dans la bonne humeur. Je les remercie aussi pour leurs précieux conseils et leurs avis qui m'ont été très utiles afin de me permettre de finir dans les temps. Enfin, je souhaite aussi les remercier pour les nombreuses relectures.

Mes remerciements s'adressent également à Monsieur Hoang Son Pham pour son aide et ses réponses lors de la découverte du code Java ainsi que pour les réunions auxquelles il a participé durant le premier quadrimestre de l'année académique.

Ensuite, je souhaite remercier Quentin Hauspie pour les échanges partagés à propos du code Java. Ces échanges constructifs m'ont permis de comprendre un autre point de vue sur ce code.

Enfin, je souhaite remercier mes parents pour leur soutien moral et surtout pour leur aide très utile et fort appréciée lors des relectures de mon mémoire.

Table des Matières

Chapitre 1

Introduction

Habituellement, lorsque l'on veut analyser un code informatique, on se focalise principalement sur l'efficacité ou les résultats. Cependant, d'autres critères existent. Par exemple, l'étude des répétitions dans un code est très intéressante. Cette étude permet par exemple, d'aider à la compréhension d'un code mais aussi à la refactorisation de ce dernier ou encore d'y localiser des fautes. Plusieurs outils existent afin de localiser ces répétitions. Cependant, celui qui servira de base à ce mémoire sera l'algorithme *FREQTALS* expliqué dans le papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1]. Ce papier présente le résultat de recherche à propos de l'extraction de patterns fréquents d'un code informatique. L'objectif étant de poser les conditions nécessaires afin de n'extraire que des patterns utiles ainsi que d'en limiter leur nombre. Ce papier explique l'algorithme utilisé ainsi que les résultats obtenus. Il a été écrit suite au développement d'un code Java.

L'objectif de mon mémoire est de traduire ce code Java existant en un programme Python. Cette traduction est nécessaire pour plusieurs raisons. Tout d'abord, Java n'est pas le langage le plus populaire lorsqu'on réalise du pattern mining. Généralement, le langage Python lui est préféré. De plus, le langage Java est souvent plus difficile à prendre en main; ce qui rend toutes modifications du code plus difficiles. Or, des recherches sont toujours en cours actuellement afin d'améliorer les contraintes existantes. Il semble intéressant de pouvoir réaliser ces recherches dans les meilleures conditions et dans un environnement plus aisé tel que Python. Dans ces conditions, traduire le code Java existant en un code Python élégant et utilisant les bonnes pratiques d'implémentation Python est recommandé. L'objectif principal de cette traduction est avant tout d'avoir un code complet et facilement compréhensible afin de faciliter toute modification future. L'efficacité n'est quant à elle qu'un objectif secondaire. En effet, chaque nouvelle contrainte validée en Python sera ensuite implémentée dans le code Java, réputé actuellement comme efficace.

Ce mémoire résume l'avancé de mon travail durant cette année académique. Il se divise en plusieurs parties. Dans un premier temps, je résume l'algorithme *FREQTALS* et définis quelques termes utiles pour une bonne compréhension de ce mémoire. Ensuite, le travail de traduction réalisé durant cette année s'étend sur plusieurs chapitres. Un premier, le chapitre 4, explique l'implémentation créée. Le chapitre 5 reprend les difficultées rencontrées et les adaptations à faire lorsque l'on traduit un code du langage Java en Python. Afin de vérifier la justesse de la traduction, le chapitre 6 explique les tests effectués. Au chapitre 7, j'aborde brièvement les résultats obtenus. Enfin, je propose à l'occasion du chapitre 8 quelques pistes d'amélioration dans l'objectif d'une poursuite éventuelle du travail effectué cette année.

Chapitre 2

Quelques définitions

Dans ce chapitre, je vais définir différents termes fréquemment utilisés dans ce mémoire. Ensuite, dans les chapitres suivants, je vais aborder le fonctionnement de l'algorithme *FreqT* et ses extensions possibles ainsi que ma traduction en Python. Il est à noter que ce chapitre a été écrit sur base du papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1].

2.1 Abstract Syntax Tree

Un premier concept important est celui "d'arbre de syntaxe abstraite" (AST) ou *"Abstract Syntax Tree"* en anglais. Ces arbres sont des représentations de la structure syntaxique abstraite d'un code source écrit dans un langage de programmation. Les nœuds d'un arbre sont nommés à l'aide d'une étiquette ou *label* en anglais et sont ordonnés entre eux. Un tel arbre peut être obtenu à l'aide d'un analyseur ou *parser* en anglais et ce pour n'importe quel langage. L'arbre obtenu est ordonné et peut être défini comme $T = \{V, E, \lambda, \Sigma\}$. Dans cette définition, V correspond à l'ensemble des noeuds totalement ordonnés et est donné par $V = \{1, 2, ..., n\}$. L'ensemble des noeuds de l'arbre est numéroté selon l'ordre d'un *depth first search*, allant de gauche à droite transversalement. Donc pour résumer, le nœud numéro 1 correspond à la racine tandis que le nœud le plus à droite correspond à l'identifiant n. $E \subseteq V \times V$ comprend l'ensemble des arrêtes permettant de définir une relation parent-enfant dans l'arbre. Σ est l'ensemble des étiquettes possibles pour un nœud. Finalement, $\lambda : V \to \Sigma$ est une fonction associant une étiquette à chaque nœud de V. Une des caractéristiques d'un AST est que le chemin le plus court pour passer du nœud 1 à n, se fait en empruntant le chemin le plus à droite. Cette propriété sera utile par la suite.

Cependant, pour un alphabet donné, il existe plusieurs arbres T. Ces arbres

composent un ensemble défini comme $\mathbf{T}$. Deux arbres $T_1 = \{V_1, E_1, \lambda_1, \Sigma_1\}$ et $T_2 = \{V_2, E_2, \lambda_2, \Sigma_2\}$ peuvent être comparés entre eux tels qu'on puisse dire que T_2 est induit par T_1 ($T_2 \preceq T_1$) s'il existe une fonction injective f: $V_2 \to V_1$ tel que:

- les arrêtes sont préservées: $\forall (v, v') \in E_2 : (f(v), f(v')) \in E_1$
- les étiquettes sont préservées: $\forall v \in V_2 : \lambda_2(v) = \lambda_1(f(v))$
- l'ordre est préservé: si $v_1 < v_2$ pour toutes les pairs de nœuds de V_2, alors $f(v_1) < f(v_2)$

Ce qui revient à relier les nœuds de V_2 aux nœuds de V_1. C'est équivalent à retrouver une occurrence de l'arbre T_2 dans T_1 qui est définie par la fonction f.

2.2 Support

Le terme *support* d'un pattern sera aussi très utilisé dans la suite de ce mémoire. Il correspond au nombre de fois qu'un pattern se retrouve dans la base de données. Dans notre cas, ce pattern correspond à un sous-arbre, T_1. La base de données se compose elle aussi d'un ensemble d'arbres $T \in \mathbf{T}$. Le support est alors défini comme le nombre d'arbres de la base de données contenant l'arbre T_1. Ce qui revient à compter le nombre d'arbres de la base de données où T_1 est induit dans T, $T_1 \preceq T$.

2.3 Le nombre d'occurrences

Le nombre d'occurrences d'un pattern désigne le nombre d'apparitions du nœud racine dans un ensemble d'arbres. Il peut être plus grand que le support si le pattern apparaît plusieurs fois dans un même arbre.

2.4 Extraction de sous-arbres fréquents

L'algorithme tente de résoudre un problème appelé extraction de sous-arbres fréquents ou en anglais *Frequent subtree mining*. Ce type de problème tente de trouver tous les patterns tels que leur support soit supérieur à une valeur donnée appelée *minimum support threshold* ou *threshold* pour faire court. Comme on peut s'y attendre, le nombre de patterns trouvés va dépendre de la valeur de cette limite et peut devenir assez conséquent. C'est pourquoi, l'algorithme *FreqT*, qui sera introduit dans le chapitre 3, a été amélioré pour pouvoir y ajouter d'autres contraintes qui seront expliquées également plus tard. Cependant, un des concepts

ajoutés mérite d'être expliqué. En effet, pour gérer un grand nombre de patterns, il est possible d'exécuter une recherche extrayant uniquement les sous-arbres fréquents maximaux ou *maximal frequent subtrees* en anglais. Ce type de recherche a pour but de trouver tous les patterns fréquents qui sont dominés par aucun autre pattern. En notation mathématique, cela donne:

$$max(T_p) = \{T \in T_p \mid \nexists\, T' \in T_p : T' \succ T\} \tag{2.1}$$

où T_p correspond à l'ensemble des patterns fréquents selon les contraintes posées.

2.5 Contrainte anti-monotone

Une contrainte C est anti-monotone si pour tous les sous-arbres S et S', avec S' inclus dans S et S respecte la contrainte C, alors S' respecte aussi la contrainte C. En notation mathématique, cela donne:

$$\forall S,\, S', \text{ si } S' \preceq S \text{ et } S \text{ respecte } C \Rightarrow S' \text{ respecte } C \tag{2.2}$$

De manière équivalente, si un sous-arbre S viole la contrainte anti-monotone C, alors tout sous-arbre S' contenant le pattern S violera la contrainte C. Ce genre de contrainte est très intéressante afin d'éliminer une partie de l'espace de recherche. En effet, si on trouve un pattern ne respectant pas une contrainte anti-monotone, alors tous les supers patterns contenant ce dernier peuvent être rejetés sans calculer leur fréquence. Cette propriété est très utile pour accélérer le processus de recherche de patterns fréquents et limiter la quantité de patterns à analyser.

2.6 Chemin le plus à droite

Un autre terme fréquemment abordé lors de l'explication de l'algorithme *FreqT* est le terme "chemin le plus à droite" ou *rightmost path* en anglais. En effet, l'algorithme *FreqT* agrandit toujours les patterns trouvés en les étendant par le chemin le plus à droite soit selon un ordre de type *depth first search*. Afin d'être plus explicite, la figure 2.1 ci-dessous constitue un exemple d'une base de données avec deux ASTs.

De cette dernière, nous allons y extraire un pattern fréquent maximal selon la méthode d'extension par le chemin le plus à droite en ajoutant comme contrainte que le support du pattern obtenu soit égale à 2. La figure, 2.2, montre l'extension progressive du pattern par son chemin le plus à droite jusqu'à obtenir un pattern fréquent maximal.

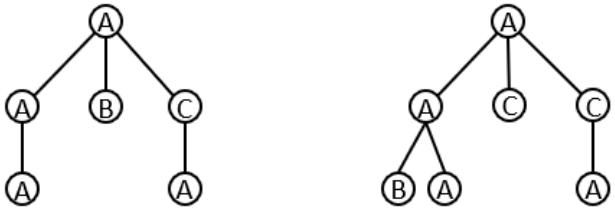

Figure 2.1: ASTs se trouvant dans la base de données

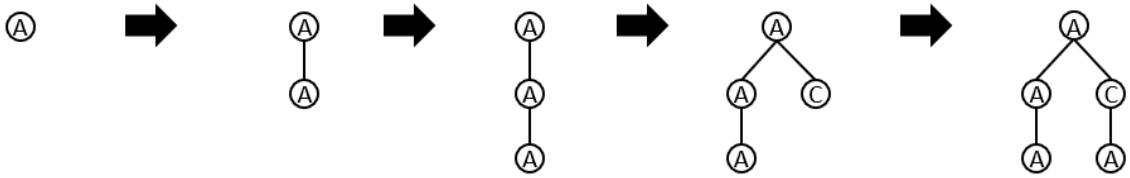

Figure 2.2: Extension du pattern par son chemin le plus à droite

Chapitre 3

L'algorithme

Ce chapitre est consacré à l'algorithme *FreqT*. Après une explication de ce dernier, j'évoque les extensions existantes qui améliorent son efficacité. Une implémentation possible de ces extensions est abordée par la suite. L'ensemble de ces démarches permet d'obtenir l'algorithme *FREQTALS*. A nouveau, le contenu de ce chapitre est basé sur le papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1].

3.1 FreqT

L'objectif de l'algorithme FreqT est de trouver des patterns fréquents dans un arbre ordonné et étiqueté. L'algorithme construit les patterns méthodiquement en étendant toujours le chemin le plus à droite. C'est à dire, que l'algorithme correspond à un *depth-first-search* et ne peut ajouter un nouveau nœud qu'à droite du dernier nœud le plus à droite. Chaque ajout se fait donc dans un ordre correspondant à un *depth-first-search*, allant de gauche à droite transversalement.

L'algorithme FreqT (cfr figure 3.1) correspond à un algorithme de pattern mining de base. En effet, initialement, la liste de patterns fréquents (FP) est vide. Le but va être de l'accroître progressivement élément par élément tout en respectant les contraintes. Pour commencer, il faut collecter tous les éléments possibles et créer des arbres se composant d'un seul nœud. Tout cela est réalisé lors de la fonction *findLabels*. Ensuite, afin de limiter le nombre d'arbres à étendre, il est déjà intéressant de vérifier s'ils respectent les conditions. La fonction *prune* permet de vérifier si un arbre respecte toutes les conditions et supprime les autres. Ensuite, pour chaque pattern fréquent trouvé, la fonction *add* permet de les garder en mémoire en les ajoutant aux patterns fréquents, *FP*. Enfin, nous allons étendre les patterns fréquents trouvés grâce à la fonction *expand*. Cette fonction va tenter

de trouver des éléments permettant d'étendre le pattern existant par son chemin le plus à droite grâce à la fonction *findCandidates*. Ensuite, ces candidats seront analysés afin de supprimer ceux ne respectant pas les conditions. Les candidats restants seront ajoutés au pattern fréquent. Cette extension va continuer jusqu'à ce que tous les patterns fréquents aient été trouvés et enregistrés dans la liste *FP*.

Algorithm 1: FREQT

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

Algorithm 2: expand procedure

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

Figure 3.1: Pseudo-code de l'algorithme FreqT extrait du papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1]

Par défaut, seule la contrainte de minimum support est utilisée par l'algorithme FreqT. L'avantage de cette contrainte est qu'elle est anti-monotone. Des contraintes telles que le minimum support sont assez faciles à implémenter et requièrent juste d'être ajoutées dans la fonction *prune*. L'implémentation de cette contrainte permet déjà de diminuer la taille de l'espace de recherche.

Cependant, il arrive que le nombre de patterns trouvés soit élevé. En effet, le nombre de patterns trouvés tend à augmenter de manière exponentielle. C'est pourquoi, il serait intéressant de pouvoir utiliser des critères/contraintes afin de limiter ce nombre, d'autant plus si ces contraintes nous permettent de garder uniquement des patterns intéressants à analyser. Pour ce faire, un certain nombre de contraintes ont été ajoutées à l'algorithme FreqT.

3.2 Les contraintes

Pour comprendre quelles contraintes sont imposées pour obtenir un nombre acceptable de patterns, et de surcroît des patterns intéressants, il faut se pencher sur deux cas intéressants à analyser.

D'une part, un programme informatique est structuré et est prévisible. Les patterns qui reflètent uniquement ce côté prévisible sont inutiles. Par exemple, une expression infixe se composera toujours de 3 enfants. Un opérant à gauche et un opérant à droite qui encadre un opérateur. Ce type de pattern est fréquent mais

n'est pas une conséquence du code que l'on analyse mais plutôt de la structure du langage qu'on utilise. De même, un pattern qui met de côté les enfants d'une expression infixe n'est pas non plus intéressant. Il est donc nécessaire d'imposer qu'une telle expression soit toujours suivie de ses enfants dans le pattern.

D'autre part, pour qu'un pattern soit intéressant à analyser, il faut qu'il ne soit pas trop petit. En effet, un pattern consistant en une expression infixe et ses 3 enfants n'apporte rien. Il est donc nécessaire que le pattern trouvé soit suffisamment large et nous permette de voir le contexte dans lequel il se trouve. Ajouter une condition sur la taille minimale des patterns trouvés ne semble donc pas une mauvaise idée.

Maintenant que ces observations ont été faites, il est temps de se pencher sur les contraintes qui peuvent être ajoutées à l'algorithme FreqT de base expliqué ci-dessus.

3.2.1 Contrainte de taille minimale

Le but de cette contrainte est d'éviter d'avoir des patterns de tailles trop petites et donc plus difficile à interpréter. Il y a deux moyens de satisfaire cette contrainte:

- Imposer que le nombre de feuilles d'un pattern soit suffisamment large,
- Imposer que le nombre de nœuds d'un arbre soit suffisamment large.

3.2.2 Contrainte sur les étiquettes des nœuds

Pour trouver certains patterns plus larges, il est nécessaire de rechercher des patterns avec une fréquence de support plus ou moins basse. Cependant, pour éviter que le nombre de patterns retournés soit trop élevé, il est intéressant de trier les patterns à rechercher à l'aide de leur étiquette. Pour ce faire, plusieurs contraintes sont possibles.

- Sélectionner et limiter les étiquettes qui peuvent être sélectionnées comme racine de notre pattern,
- Interdire que des étiquettes se retrouvent dans un pattern (blacklist),
- Limiter le nombre de frères et sœurs dans un même pattern qui peuvent avoir la même étiquette.

La dernière contrainte est importante afin de supprimer tous les patterns ayant des sous-arbres répétitifs. Un pattern correspondant à un ensemble d'assignations de variables n'est pas intéressant. Dès lors, forcer la diversité dans les étiquettes des

enfants d'un noeud semble une bonne solution pour garder une certaine diversité dans les patterns.

3.2.3 Contrainte sur les feuilles

Idéalement, les patterns trouvés ne doivent pas uniquement représenter la structure du langage utilisé mais aussi contenir des informations propres au code analysé. Ces dernières sont généralement contenues dans les feuilles des ASTs figurant dans la base de données. La contrainte suivante peut dès lors être ajoutée:

- Toutes les feuilles d'un pattern doivent avoir une étiquette qui se trouve être au moins une fois la feuille d'une des ASTs de la base de données.

Ajouter une telle contrainte est utile afin de s'assurer que les patterns trouvés terminent tous par des noeuds terminaux. Ainsi, il sera impossible de trouver un noeud avec comme étiquette "expression infixe" comme noeud terminal. En effet, ce type de noeud doit toujours être suivi d'opérants et d'un opérateur et ne peut pas par conséquent, être une feuille d'une AST de la base de données. De plus, cette information seule est incomplète et ne permet pas une compréhension parfaite du pattern trouvé.

3.2.4 Enfants obligatoires

La structure d'un langage de programmation peut imposer à un nœud parent d'avoir des enfants. Il est donc intéressant de pouvoir produire des patterns contenant l'ensemble des enfants des nœuds qu'ils contiennent. Il faut néanmoins définir les conditions pour lesquelles une étiquette est structurelle:

- Dans chaque occurrence, deux enfants ne peuvent pas avoir la même étiquette,
- Pour toutes les paires d'occurrences, l'ordre des étiquettes doit toujours être la même.

La première contrainte est importante pour s'assurer de retrouver l'ensemble des enfants d'un noeud évitant ainsi d'en oublier mais aussi afin de pouvoir décider d'un ordre entre les différents enfants en évitant toute indécision possible. Ces deux caractéristiques combinées donnent la contrainte suivante:

- Pour tous les nœuds appartenant aux étiquettes structurelles, il est exigé que ses enfants contiennent les nœuds obligatoires.

Cette contrainte ajoutée à celle sur les feuilles implique aussi que tous les nœuds structurels incluent des nœuds feuilles. Par exemple, cette condition est très

importante pour ce qui est des expressions infixes. En effet, pour être complètes, il faut qu'elles se composent d'opérants et d'un opérateur. L'oubli d'un de ces enfants nuit à la bonne compréhension du pattern trouvé et à son exactitude. Imposer un lien parent-enfants permet ainsi de s'assurer que tous les enfants soient présents dans le pattern trouvé.

3.2.5 Contrainte de sous-arbre maximal

Toutes les contraintes définies plus haut permettent de trouver des patterns intéressants. Cependant, elles ne limitent pas toujours assez la taille de l'espace de recherche pour exécuter ces recherches en un temps raisonnable ou alors le nombre de patterns trouvés reste beaucoup trop grand.

Pour résoudre ces problèmes, une contrainte de taille maximum du pattern trouvé a été créée. L'exécution de l'algorithme se fera alors en deux étapes. Dans un premier temps, l'ensemble des contraintes citées précédemment ainsi que la contrainte de taille maximale des patterns sont utilisés afin de trouver des patterns fréquents de bonne taille tout en limitant l'espace de recherche. Ensuite, les patterns fréquents maximaux trouvés à l'étape précédente seront à nouveau étendus afin d'extraire les patterns les plus grands possibles et ce sans la contrainte sur la taille maximale du pattern imposé lors de l'étape 1.

La première phase permet de s'assurer que le noyau dur du pattern est intéressant. Tandis que la deuxième s'assure que le pattern est interprétable et contient l'ensemble des éléments qui apparaissent simultanément avec ce noyau dur.

3.3 FREQTALS

L'algorithme FREQTALS [1] est une amélioration de l'algorithme FreqT en y ajoutant l'ensemble des contraintes citées ci-dessus. Pour la majorité des contraintes, les modifications à effectuer se limitent aux fonctions *prune* et *add*. Seule la contrainte de sous-arbre maximale requiert de plus grandes modifications.

3.3.1 Ajout des contraintes de taille minimale

Il suffit de modifier la fonction *add* afin de vérifier si les contraintes sont respectées et d'ajouter uniquement ces patterns à la liste des patterns fréquents.

3.3.2 Ajout des contraintes sur les étiquettes des nœuds

Toutes les contraintes prenant en compte l'étiquette des nœuds sont des contraintes anti-monotones c'est à dire que si un arbre ne respecte pas cette contrainte, alors tous les sous-arbres ne la respecteront pas non plus. Cette définition facilite l'ajout de cette contrainte dans l'algorithme. En effet, seule la fonction *prune* doit être modifiée afin de n'ajouter que les extensions respectant la contrainte au candidat possible.

3.3.3 Ajout des contraintes sur les feuilles

Vérifier cette contrainte n'est pas facile. En effet, cette contrainte n'est pas anti-monotone au sens strict. Dans les premiers pas de la recherche de patterns fréquents, le pattern ne contiendra certainement pas de feuilles et ces dernières seront ajoutées plus tard. Cependant, l'algorithme étend l'arbre toujours selon son chemin le plus à droite. Ce qui veut dire qu'une grande partie de l'arbre ne peut plus changer. C'est dans cette partie que nous allons vérifier que la contrainte est toujours respectée.

Toutes les feuilles se trouvant en dehors du chemin le plus à droite peuvent d'ores et déjà vérifier la contrainte. S'il s'avère qu'une de ces feuilles de l'arbre n'appartient pas à l'ensemble des feuilles permises, alors l'arbre viole la contrainte et sera éliminé des candidats possibles. Ainsi, il suffit de vérifier dans la fonction *prune* que toutes les feuilles de l'arbre exceptée celle sur le chemin le plus à droite appartiennent à l'ensemble des feuilles valables.

3.3.4 Ajout des contraintes sur les enfants obligatoires

A nouveau, cette contrainte n'est pas anti-monotone au sens strict. Cependant, l'ajout des enfants d'un nœud requiert un ordre particulier. En effet, comme seul le chemin le plus à droite est étendu, l'enfant deux ne peut pas être ajouté avant l'enfant un mais doit être ajouté avant l'enfant trois. Il est donc possible de vérifier si l'ensemble des enfants ont été ajoutés et si l'ordre est respecté.

Pour vérifier ces conditions, il suffit d'ajouter une vérification permettant de s'assurer que tous les enfants obligatoires se trouvent dans l'ordre dans l'arbre et ce jusqu'au chemin le plus à droite. Cette vérification doit se faire dans la fonction *prune* afin de supprimer tous les candidats ne respectant pas cette condition.

3.3.5 Ajout de la contrainte de sous-arbre maximal

Ajouter cette contrainte nécessite plus de modifications que les autres contraintes. En effet, comme expliqué précédemment et montré dans l'algorithme de la figure 3.2, l'algorithme se déroule en 3 phases:

1. rechercher des sous-arbres fréquents avec toutes les contraintes présentées ci-dessus ainsi que la contrainte sur le nombre maximum de feuilles de l'arbre,
2. regrouper les arbres fréquents par occurrence de la racine,
3. pour chaque ensemble d'occurrences des racines identifiées, lancer une recherche pour trouver le sous-arbre maximal ayant cette occurrence de racine.

Algorithm 3: FREQTALS algorithm

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

Figure 3.2: Pseudo-code de l'algorithme FREQTALS extrait du papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1]

Dans la figure 3.2, à la ligne 1, l'algorithme FreqT avec l'ensemble des contraintes présentées ci-dessus est lancé. Les fonctions *add* et *prune* ont donc été adaptées comme expliqué plus haut. De plus, la fonction *prune* est modifiée pour ajouter la contrainte du nombre maximum de feuilles d'un arbre ce qui permet de limiter la taille des arbres fréquents trouvés. Une optimisation afin de supprimer les arbres pour lesquels il existe une extension fréquente a aussi été ajoutée.
La ligne 2 groupe les patterns par l'occurrence de leurs racines. Le résultat obtenu correspond à un ensemble RO = {occ(T) | T $\in$ *FP*} où plusieurs patterns peuvent avoir la même valeur de occ(T). La taille de l'ensemble obtenu est donc plus petite que celle de l'ensemble original. De plus, occ(T) correspond à un ensemble de nœuds dans la base de données et seuls les ensembles de taille minimale seront

gardés ROM = { r ∈ RO | ∄ r' ∈ RO: r' ⊂ r}. Cette contrainte n'affecte pas nos résultats. En effet, les patterns apparaissant dans le plus grand ensemble peuvent aussi être trouvés dans le plus petit.
La ligne 6 lance une recherche pour trouver les patterns maximaux pour chaque occurrence r de RO. Pour ce faire, une version modifiée de FreqT est nécessaire:

- On démarre les recherches avec les labels des racines de RO,
- Les occurrences de chaque racine considérée sont seulement celles contenues dans r même si le label de la racine apparaît ailleurs dans les données originelles,
- Au lieu d'utiliser une contrainte de minimum support, on va imposer que le pattern extrait se trouve dans toutes les occurrences de la racine de r,
- De plus, la contrainte de taille maximale n'est plus appliquée durant cette étape pour vérifier si un pattern est fréquent,
- Pour chaque pattern généré, on vérifie le respect des contraintes en prenant en compte les modifications citées aux deux points précédent afin de vérifier s'il peut être ajouté à *MP*.

Il peut arriver que la fonction *mineMaximalSubtrees* trouve plus d'occurrences dans les données originales que dans celles trouvées durant la recherche. Cela n'impacte pas la qualité de la recherche car le pattern trouvé reste maximal et fréquent. Cette observation implique qu'utiliser FreqT avec une occurrence plus petite permet de ne manquer aucune occurrence et améliore l'efficacité de l'algorithme.

Il arrive cependant, que la recherche du pattern maximal par la fonction *mineMaximalSubtrees* prenne trop de temps. Pour ce faire, une stratégie de recherche incrémentale a été implémentée. Elle permet d'allouer le même temps de recherche pour chaque $r \subset ROM$ afin de s'assurer que chaque pattern ait une chance de grandir. Le temps restant, s'il en reste, servira à agrandir les patterns qui ont été interrompus plus tôt.

Chapitre 4

L'implémentation Python

Dans ce chapitre, j'introduis la manière dont l'implémentation Java est construite. Ensuite, j'explique la méthode de traduction choisie dans le cadre de ce mémoire et son influence sur la structure du code Python. L'étude de ce dernier permet de souligner enfin quelques points intéressants et de les comparer avec les pseudo-codes présentés dans le chapitre 3.

4.1 Structure de l'implémentation Java

L'implémentation Java se trouve dans un dossier nommé *freqt*. Ci-dessous figure la liste des principaux éléments contenus dans ce dossier:

- Un dossier nommé *ressources*,
- Un dossier dénommé *src*,
- Un exécutable Java nommé *forestmatcher.jar*,
- *freqtals.jar*, un exécutable Java,
- Un fichier *README.md*.

Le contenu du dossier *freqt* tant inventorié, il convient désormais de développer chacun des éléments précités plus en détail.

Commençons par le fichier *README.md*. Ce fichier contient une petite explication de ce qu'est l'algorithme *FREQTALS*, la configuration PC nécessaire à son exécution ainsi qu'une description des librairies externes nécessaires à son bon fonctionnement. La suite du document explique comment lancer le code et quels arguments lui fournir. Enfin, le reste du document contient des informations

nécessaires pour remplir le fichier *.config*, quant au type de fichiers à donner en *input* d'une part et au type de fichiers attendus en output d'autre part. En bref, ce fichier permet d'expliquer le code, tous les éléments nécessaires à son bon fonctionnement et donne aussi des informations sur la paramétrisation possible de ce dernier.

Le fichier *freqtals.jar* est l'exécutable Java de l'algorithme *FREQTALS*. C'est l'appel à ce fichier qui permet de faire fonctionner le code contenant l'ensemble des contraintes exposées dans le chapitre précédent.

Le fichier *forestmatcher.jar* est un exécutable Java utilisé lors de l'exécution de l'algorithme *FREQTALS*. Cet exécutable permet de localiser les patterns trouvés dans les codes sources reçus en *input*. Il est donc important qu'il reste en tout temps dans le même dossier que l'exécutable *freqtals.jar*.

Le dossier *ressources* contient quant à lui plusieurs dossiers. Ces dossiers sont au nombre de 6. En plus de ces 6 dossiers, un fichier *README.md* s'y trouve aussi. Ce dernier contient quelques informations sur le contenu de 3 des 6 dossiers ainsi qu'un exemple de commande à utiliser pour lancer l'exécutable *freqtals.jar* sur les fichiers contenus dans les 6 dossiers se trouvant dans *ressources*. En ce qui concerne les dossiers, ils se classent en deux groupes de 3. Chaque groupe contient un dossier contenant les fichiers de configuration. Ils sont reconnaissables à leur nom contenant le mot "conf". Le second type de dossiers stocke plusieurs codes Java et leur traduction en ASTs enregistrés dans des fichiers *.XML*. Ils serviront d'input à l'algorithme *FREQTALS*. Enfin, le dernier des trois dossiers sert à enregistrer les outputs créés par l'algorithme *FREQTALS*.

Le dernier dossier nommé *src* contient l'ensemble du code Java. Le code se compose de plusieurs fichiers séparés en plusieurs groupes. Chaque groupe décrit l'utilité du fichier et correspond à un dossier. Il existe 8 groupes auxquels viennent s'ajouter le fichier *main.java*. Ce dernier permet de lancer le code avec les configurations reçues en *input*. Ces 8 groupes/dossiers se nomment:

- *config*,
- *constraint*,
- *core*,
- *grammar*,
- *input*,
- *output*,

- *structure*,
- *util*.

Le dossier *config* contient le code Java permettant d'interagir avec les informations contenues dans le fichier *.config* passé en argument.

Le code implémentant les contraintes se retrouve dans le dossier *constraint*. Ce fichier définit l'ensemble des contraintes et un appel à ces fonctions suffit pour les utiliser.

Le dossier *core* recueille le code principal de l'algorithme *FREQTALS*. Par exemple, l'implémentation des pseudo-codes présentés dans le chapitre précédent peuvent y être retrouvés.

Les dossiers *input* et *output* contiennent respectivement le code permettant de lire les *inputs* et le code permettant de créer les *outputs*.

Le dossier *grammar* contient les codes permettant de créer la grammaire des ASTs donnés en *input*. Cette grammaire est utilisée à plusieurs endroits dont notamment dans les fichiers de codes se trouvant dans le dossier *core* et *constraint*.

Ensuite, le dossier nommé *structure* contient l'ensemble des structures de données utilisées pour stocker les informations utiles à l'exécution de l'algorithme *FREQTALS*.

Enfin, le dossier *util* contient plusieurs fichiers permettant entre autres de faciliter l'exécution de l'algorithme *FREQTALS* ou de faciliter le débuggage du code en proposant plusieurs fonctions permettant d'afficher le contenu des différentes structures utilisées.

4.2 Méthode de traduction

Maintenant que la structure du code Java est connue, il est intéressant de s'attarder sur la méthode de traduction choisie. En effet, le choix de celle-ci peut dans certains cas influencer la structure de l'implémentation Python.

Pour traduire le code Java en Python, j'ai choisi de suivre une méthode *bottom-up*. En d'autres termes, j'ai commencé par traduire de petits éléments tels que les structures avant de m'avancer plus loin dans le code et créer la base de l'algorithme ou encore la fonction *main*. Cette approche permet de ne traduire

que du code requérant uniquement des éléments déjà existants. J'ai choisi cette méthode de traduction car elle m'a permis de créer des tests en parallèle de la traduction. Cela n'aurait pas été possible si je commençais par traduire les fichiers correspondant au cœur de l'algorithme soit selon une méthode *top-down*. En effet, les fichiers au cœur de l'algorithme font appel à de nombreuses fonctions se trouvant dans d'autres fichiers. J'aurais dû alors traduire ces autres fichiers avant de pouvoir créer un premier test. Cette approche procure donc peu de certitudes sur la véracité des traductions faites à l'instant t et a donc été mise de côté.

Ensuite, une fois l'ordre de traduction des fichiers établis, seule le type de traduction reste à définir. Je suis parti sur une traduction 1 pour 1. C'est à dire que tout ce qui se trouve dans la version Java se trouve dans la version Python. Cependant, les langages Java et Python ne sont pas toujours pareils. Dans ce cas-là, la traduction 1 pour 1 était légèrement adaptée afin d'obtenir les mêmes résultats en adaptant le chemin utilisé. Quelques exemples de ces modifications seront abordés dans la suite de cette section. Cela étant, une grosse partie du code peut être facilement retrouvée et consiste en une traduction directe. L'avantage d'une telle traduction est que la recherche d'un bout de code Java dans l'implémentation Python reste assez facile. En effet, si ce dernier apparaît, il doit se trouver dans le fichier Python portant le même nom que le fichier Java d'où le code provient. Il se peut cependant que certains éléments aient dû être ajoutés pour un bon fonctionnement en Python ou encore que certaines fonctions Java n'aient pas été traduites car elles n'étaient plus utiles dans la dernière version de l'algorithme.

4.3 Structure de l'implémentation Python

Après une brève explication de la manière dont j'ai procédé pour créer ma traduction, il est temps d'expliquer la structure des fichiers de l'implémentation Python. Le code se trouve dans un dossier nommé *freqt_python*. Comme on peut s'y attendre, ce dossier contient les mêmes éléments que le dossier Java, c'est à dire:

- Un dossiers nommé *ressources*,
- Un dossier *src*,
- Un dossier *test*,
- Un exécutable Java nommé *forestmatcher.jar*,
- Et enfin, un fichier *README.md*.

Vous pouvez déjà remarquer que le fichier *freqtals.jar* a disparu et un nouveau dossier appelé *test* l'a remplacé.

La disparition du fichier *freqtals.jar* est facilement explicable. En effet, il s'agit de l'exécutable Java de l'algorithme traduit. Il ne nous sera plus d'aucune utilité grâce à l'implémentation Python. En revanche, aucun exécutable Python n'est venu le remplacer car Python est un langage interprété, c'est à dire qu'il ne crée pas de fichier exécutable mais exécute le code directement.

Le dossier *test* quant à lui fait son apparition. Ce dossier sert à contenir l'ensemble des fichiers utilisés lors de l'exécution des tests unitaires. Ces fichiers sont des copies de fichiers se trouvant dans le dossier *ressources*. Cependant, il était utile de créer ce nouveau dossier afin de s'assurer que les tests restent corrects. En effet, rien ne nous empêche de modifier un fichier contenu dans le dossier *ressources*. Une telle modification entraînerait la nécessité de modifier les tests unitaires afin qu'ils restent corrects. Devoir effectuer une telle modification n'est pas pratique. C'est pourquoi, j'ai créé ce dossier *test* afin de conserver une copie non modifiée gardant ainsi les tests unitaires corrects en tout temps. Il est donc possible de se fier entièrement à ces tests. Cette propriété est très intéressante en cas de modification du code afin de vérifier si la nouvelle version est correcte.

Enfin, l'exécutable Java nommé *forestmatcher.jar* reste utile. En effet, dans le code Java, c'est lui qui permet de localiser les patterns trouvés dans les codes sources reçus en *input*. J'ai donc garder le même principe afin que le code Python soit aussi complet en terme d'informations recueillies sur un pattern. Par conséquent, je fais appel au fichier *.jar* depuis la traduction Python. Ce fichier n'a pas été traduit car il n'est pas directement en lien avec l'algorithme *FREQTALS*.

Le contenu des autres éléments n'est pas abordé car leur contenu ressemble fortement à ce qui a été expliqué dans la section sur la structure de l'implémentation Java.

4.4 L'implémentation Python

Cette section se focalisera sur l'implémentation Python. Son but sera de souligner quelques points importants de l'implémentation Python. Dans certains cas, une comparaison avec les pseudo-codes présentés précédemment sera aussi établie. A l'aide de ces derniers, les fonctions les plus importantes de l'implémentation seront abordées. Dans la même optique, un choix d'implémentation fait lors de la création du code Java sera aussi expliqué et analysé.

4.4.1 La fonction *prune*

Une première fonction intéressante à analyser est la fonction *prune*. Cette fonction a déjà été évoquée dans le chapitre 3. Pour rappel, son but est d'éliminer de la liste des patterns fréquents trouvés tous les patterns non-fréquents ou ne respectant pas les conditions posées. Cette fonction est intéressante à analyser car un choix de conception a dû y être fait lors de son développement Java. En effet, nous avons pu voir dans le chapitre précédent que plusieurs exécutions différentes étaient possibles. La première trouve des patterns maximaux directement. Tandis que la seconde cherche initialement des patterns de petites tailles à l'aide d'une contrainte sur le nombre maximum de nœuds et va dans un second temps, étendre les patterns trouvés sans la contrainte sur le nombre de nœuds maximum. Pour satisfaire ces deux types d'exécutions, deux implémentations sont possibles.

La première est de faire une fonction *prune* prenant en compte les deux cas et de les différencier à l'aide de conditions. L'avantage serait que toutes les contraintes seraient vérifiées grâce à un seul appel de fonction. Cependant, une telle fonction nécessiterait beaucoup d'arguments.

La deuxième solution serait de faire une fonction *prune* générale ne vérifiant que le support d'un pattern et de vérifier les autres contraintes lors de la découverte de nouveaux candidats. C'est cette solution qui a été choisie lors du développement Java. Ce choix implique que la fonction *prune* serve juste à vérifier la fréquence d'un pattern. Les autres contraintes telles que celles sur la taille, les enfants, ... sont quant à elles vérifiées directement dans les fonctions recherchant les patterns telles que les fonctions *expand* ou *addTree*.

Afin d'illustrer cette explication, vous trouverez la fonction *prune* traduite en Python ci-dessous. Cette traduction est une traduction quasi-directe de la version Java. Le seul changement est qu'en Java, afin d'itérer à travers un dictionnaire, on utilise un Iterator. En Python, la création d'un Iterator n'est pas nécessaire. Cependant, cet avantage a aussi ses inconvénients. En effet, en Java, les patterns non-fréquents sont directement retirés du dictionnaire *candidates* alors qu'en Python, j'ai dû passer par une liste intermédiaire avant de pouvoir les retirer. Autrement, une erreur était retournée car je changeais la taille du dictionnaire *candidates* durant l'itération. Hormis cette légère adaptation, la fonction ressemble en tout point à son homologue Java.

```
1 """
2  * prune candidates based on minimal support
```

```
 * @param: candidates, dictionary with FTArray as keys and Projected
      as values
 * @param: minSup, int
 * @param: weighted, boolean
"""
def prune(self, candidates, minSup, weighted):
    to_remove_list = list()
    for elem in candidates:
        sup = self.getSupport(candidates[elem])
        wsup = self.getRootSupport(candidates[elem])
        if weighted:
            limit = wsup
        else:
            limit = sup
        if limit < minSup:
            to_remove_list.append(elem)
        else:
            candidates[elem].setProjectedSupport(sup)
            candidates[elem].setProjectedRootSupport(wsup)
    for elem in to_remove_list:
        candidates.pop(elem, -1)
```

4.4.2 La fonction *run*

La fonction *run* permet de démarrer l'algorithme, c'est à dire que c'est cette fonction qu'on appelle pour lancer l'exécution de l'algorithme *FREQT* et *FREQTALS*. Ce qui est intéressant dans cette fonction Python est qu'elle combine des parties des algorithmes *FREQT* et *FREQTALS*. Dans un premier temps, les similitudes avec le pseudo-code de l'algorithme *FREQT* seront abordées. Dans un second temps, je ferai un parallèle entre le code Python et le pseudo-code de l'algorithme *FREQTALS*. Afin d'améliorer la qualité de l'explication, les pseudo-codes des deux algorithmes auxquels je fais référence, seront repris distinctement ci-dessous.

Algorithm 4: FREQT

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

Figure 4.1: Pseudo-code de l'algorithme FreqT extrait du papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1]

Après le rappel du premier pseudo-code, il est temps de présenter sa traduction Python, la fonction *run*.

```
# run Freqt with file config.properties
def run(self):
    try:
        # read input data
        self.initData()
        # set starting time
        self.setStartingTime()
        # init report file
        report = self.initReport(self._config,
              len(self._transaction_list))

        # build FP1: all labels are frequent
        FP1_dict = self.buildFP1(self._transaction_list,
              self.rootLabels_set, self.__transactionClassID_list)
        # FP1_dict is a dictionary with FTArray as keys and Projected
              as values

        # remove node SourceFile because it is not AST node
        for elem in FP1_dict:
            if notASTNode.equals(elem):
                FP1_dict.pop(elem, -1)

        # prune FP1 on minimum support
        constrain = Constraint()
        constrain.prune(FP1_dict, self._config.getMinSupport(),
              self._config.getWeighted())

        # expand FP1 to find maximal patterns
        self.expandFP1(FP1_dict)
        if self._config.getTwoStep():
            self.notF_set = set()
            if self._config.get2Class():
                # group root occurrences from 1000-highest chi-square
                      score patterns
```

```
30                  self.rootIDs_dict = self.groupRootOcc(self.__HSP_dict)
31              # find pattern from root occurrences
32              self.expandPatternFromRootIDs(self.rootIDs_dict, report)
33          else:
34              self.outputPatternInTheFirstStep(self.MFP_dict,
                      self._config, self._grammar_dict,
                      self._labelIndex_dict, self._xmlCharacters_dict,
                      report)
35      except:
36          e = sys.exc_info()[0]
37          print("Error: running Freqt_Int " + str(e) + "\n")
38          trace = traceback.format_exc()
39          print(trace)
```

Il important de savoir que le fichier config.properties est celui qui contient l'ensemble des paramètres à utiliser lors de l'exécution. Ce fichier est lu et l'ensemble des informations qu'il contient est sauvegardé dans la structure nommée *config*. Un simple appel à cette dernière permet d'en retirer les informations voulues. Un exemple d'un tel appel se trouve à la ligne 22.

Le code Python démarre par un ensemble d'initiations telles que le *timer* ou la création de la grammaire utile pour certaines conditions. Cette initiation n'est pas présente dans le pseudo-code présenté plus haut car il ne prend pas en compte d'autres contraintes que la contrainte sur le support. Alors que le code Python a quant à lui été étendu pour prendre toutes les contraintes présentées plus haut en compte. L'initialisation de la grammaire a donc dû être ajoutée.

Par contre, le dictionnaire *FP1_dict* est bien présent et correspond à la variable *FP* du pseudo-code. Dans les deux cas, les deux variables sont initialement vides. Dans le code, on peut voir que la fonction *findLabels* permet de récolter quelques nœuds susceptibles d'être les racines de patterns. Ces derniers sont ensuite triés dans la fonction *prune* afin de garder uniquement les nœuds fréquents. Ces deux étapes peuvent être observées dans le code Python. En effet, la ligne 12 lance une recherche pour trouver des nœuds susceptibles d'être la racine d'un futur pattern. L'ensemble des nœuds trouvés sont alors ajoutés au dictionnaire *FP1_dict*. Ensuite, le code allant de la ligne 16 à 22, trie les nœuds trouvés selon leur fréquence. Les nœuds fréquents seront alors ajoutés comme racine des patterns et une procédure d'extension est alors lancée afin de compléter ces patterns. Tout cela se fait lors de l'appel à la fonction à la ligne 23. Ce que fait cette ligne est identique à ce qui est fait dans le pseudo-code entre la ligne 4 et 6. Enfin, le pseudo-code termine en lançant la création des fichiers de sortie contenant les patterns trouvés. Dans le code

python, cette ligne correspond à la ligne 34. Les lignes 26 à 33 restent cependant utiles et inexpliquées jusqu'à présent. En effet, le pseudo-code présenté plus haut correspond à l'algorithme de base recherchant uniquement des patterns maximaux. Mais cet algorithme a été étendu afin de permettre une deuxième exécution qui, quant à elle, se déroule en deux étapes. La première étape est identique à celle présentée dans le pseudo code mais une seconde étape servant à étendre les patterns maximaux trouvés, est réalisée à la suite de la première. La condition se trouvant à la ligne 26 permet de différencier ces deux cas. Le deuxième cas est représenté dans le pseudo-code ci-dessus 4.1. Dans un premier temps, l'algorithme *FREQT* est utilisé pour permettre le lancement de l'algorithme *FREQTALS*. La ligne 1 du deuxième pseudo-code 4.2 le montre clairement. Les lignes suivantes regroupent les patterns par occurrence et essayent de les étendre chacun à leur tour. C'est aussi ce qui se passe dans l'implémentation Python entre les lignes 27 et 32. Les appels aux fonctions se trouvant entre ces lignes permettent de grouper les patterns par occurrence et de lancer une nouvelle recherche afin d'étendre ces patterns. C'est identique à ce qui est fait aux lignes 2 à 6 du pseudo-code 4.2.

Algorithm 5: FREQTALS algorithm

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

Figure 4.2: Pseudo-code de l'algorithme FREQTALS extrait du papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1]

4.4.3 La fonction *expand*

Une autre fonction très importante est la fonction *expand*. C'est cette fonction qui va étendre récursivement les patterns fréquents. A nouveau, cette fonction

correspond à un pseudo-code présenté dans le chapitre 3. Afin de faciliter la comparaison avec le code Python, ce pseudo-code est remis ci-dessous.

Algorithm 6: expand procedure

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

Figure 4.3: Pseudo-code de la procédure d'extension extrait du papier "Mining Patterns in Source Code Using Tree Mining Algorithms", [1]

Ce pseudo code se déroule en plusieurs étapes:

1. Trouver des nœuds candidats pour étendre le patterns existants,
2. Eliminer tous les patterns ne répondant pas aux contraintes,
3. Pour chaque pattern respectant les contraintes, ajouter définitivement ces nœuds à la liste des patterns fréquents grâce à la fonction *add* uniquement s'ils ont atteint la taille maximale ou qu'il n'existe plus de candidat pour les étendre,
4. Dans les autres cas, on appelle récursivement la fonction *expand* afin d'étendre les nouveaux patterns.

Pour ce qui est du code Python présenté ci-dessous, les mêmes étapes peuvent être retrouvées. Seules quelques adaptations ont été faites. Premièrement, l'ajout d'un timer. Ce timer permet d'arrêter le code une fois le temps de recherche expiré. Ce timer peut être aperçu entre la ligne 8 et 10. Ensuite, à la ligne 13, l'appel à la fonction *generateCandidates* permet de trouver des nœuds susceptibles d'étendre les patterns déjà trouvés. Aux lignes 16 et 17, on peut voir un appel à la fonction *prune* qui va éliminer toutes les extensions pas assez fréquentes. Dans le cas où tous les candidats ont été éliminés, la recherche récursive se termine et les patterns trouvés sont enregistrés grâce à la fonction *addTree*. Enfin, s'il reste des candidats, les lignes 26 à 50 les ajoutent aux patterns et rappellent récursivement la fonction. Cependant, comme expliqué à la section 4.4.1, le choix de développement fait lors de la création de la fonction *prune*, nous oblige à vérifier les autres conditions. C'est pourquoi, ces lignes vérifient d'abord certaines contraintes avant d'effectuer l'appel récursif. Il est à noter que le reste des contraintes telles que le nombre minimum de nœuds composant le pattern est vérifié lors de l'appel à la fonction *addTree*. S'il ne respecte pas ces conditions, le pattern ne sera pas sauvegardé.

```
"""
 * expand pattern
 * @param: pattern, FTArray
 * @param: projected, Projected
"""
def expandPattern(self, pattern, projected):
    try:
        # if it is timeout then stop expand the pattern;
        if self.isTimeout():
            return

        # find candidates of the current pattern
        candidates_dict = self.generateCandidates(projected,
              self._transaction_list)  # dictionary with FTArray as
              keys and Projected as values

        # prune candidate based on minSup
        constrain = Constraint()
        constrain.prune(candidates_dict, self._config.getMinSupport(),
              self._config.getWeighted())

       # if there is no candidate then report the pattern and then stop
        if len(candidates_dict) == 0:
            if self.leafPattern.size() > 0:
                self.addTree(self.leafPattern, self.leafProjected)
            return

        # expand each candidate to the current pattern
        for key in candidates_dict:
            oldSize = pattern.size()
            # add candidate into pattern
            pattern.addAll(key)
           # if the right most node of the pattern is a leaf then keep
                 track this pattern
            if pattern.getLast() < -1:
                self.keepLeafPattern(pattern, candidates_dict[key])
            # store leaf pattern
            oldLeafPattern = self.leafPattern
            oldLeafProjected = self.leafProjected
            # check obligatory children constraint
```

```
37              if constrain.missingLeftObligatoryChild(pattern, key,
                      self._grammarInt_dict):
38                  # do nothing = don't store pattern
39                  pass
40              else:
41                  # check constraints on maximal number of leafs and real
                        leaf
42                  if constrain.satisfyMaxLeaf(pattern,
                        self._config.getMaxLeaf()) or
                        constrain.isNotFullLeaf(pattern):
43                      # store the pattern
44                      if self.leafPattern.size() > 0:
45                          self.addTree(self.leafPattern,
                                self.leafProjected)
46                  else:
47                      # continue expanding pattern
48                      self.expandPattern(pattern, candidates_dict[key])
49              pattern = pattern.subList(0, oldSize)
50              self.keepLeafPattern(oldLeafPattern, oldLeafProjected)
51      except:
52          e = sys.exc_info()[0]
53          print("Error: expandPattern " + str(e) + "\n")
54          trace = traceback.format_exc()
55          print(trace)
56          raise
```

4.4.4 La fonction *addTree*

La fonction *addTree* a déjà été légèrement abordée précédemment mais quelques détails vont être ajoutés. La fonction a pour but d'initier la procédure permettant de sauvegarder un pattern fréquent trouvé. Comme cité précédemment, certaines conditions sur le pattern doivent être vérifiées avant de le sauvegarder. Dans le cas où le résultat est négatif, le pattern est tout simplement oublié. Dans l'autre cas, la procédure d'enregistrement commence. Pour ce faire, la fonction dont le code Python se trouve ci-dessous, doit différencier les deux types d'exécution. L'exécution en une étape demande d'enregistrer uniquement les patterns maximaux et l'appel à la fonction *addMaximalPattern* permet de vérifier cette dernière condition. Dans le cas d'une exécution en deux étapes, la fonction *addRootIDs* est appelée et enregistre au mieux le pattern pour la suite des opérations. Cette fonction, à nouveau traduite directement depuis Java, termine la vérification des conditions exposées lors du chapitre précédent. Elle termine aussi de montrer les différences entre les pseudo-

codes considérant toutes ces conditions reprises dans la fonction *prune* et le choix d'implémentation fait lors de la création du code Java.

```
"""
 * add the tree to the root IDs or the MFP
 * @param: pat FTArray
 * @param: projected, Projected
"""
def addTree(self, pat, projected):
    # check minsize constraints and right mandatory children
    constraint = Constraint()
    if constraint.checkOutput(pat, self._config.getMinLeaf(),
          self._config.getMinNode()) and not
          constraint.missingRightObligatoryChild(pat,
          self._grammarInt_dict):
        if self._config.get2Class():
            # check chi-square score
            if constraint.satisfyChiSquare(projected, self.sizeClass1,
                  self.sizeClass2, self._config.getDSScore(),
                  self._config.getWeighted()):
                if self._config.getTwoStep():
                    # add pattern to the list of 1000-highest
                          chi-square score patterns
                    self.addHighScorePattern(pat, projected,
                          self.__HSP_dict)
                else:
                    self.addMaximalPattern(pat, projected,
                          self.MFP_dict)
        else:
            if self._config.getTwoStep():
               # add root occurrences of the current pattern to rootIDs
                self.addRootIDs(pat, projected, self.rootIDs_dict)
            else:
                # add pattern to maximal pattern list
                self.addMaximalPattern(pat, projected, self.MFP_dict)
```

4.5 Conclusion

Le code Python complet se trouve en annexe E. Il est une traduction quasi-directe voire directe du code Java *FREQTALS*. Cette caractéristique est soulignée par les similitudes entre les structures des codes Java et Python ainsi que par la méthode

de traduction utilisée. Les pseudo-codes abordés lors du chapitre 3 peuvent être retrouvés facilement dans les fonctions principales du code Python. La lecture de ces fonctions permet d'ores et déjà de comprendre le fonctionnement du code. Dans le chapitre suivant, une analyse plus précise de certains choix d'implémentation sera effectuée.

Chapitre 5

Traduction Java/Python adaptations et difficultés

Lors du chapitre 4, nous avons pu analyser une partie du code Python, et ainsi ses fonctions les plus importantes mais aussi mettre en évidence certaines difficultés rencontrées lors de la traduction. En effet, les langages Java et Python, bien qu'ayant beaucoup de similitudes, diffèrent sur certains points. Leurs différences peuvent parfois compliquer une traduction mais dans certains cas, la traduction peut être rendue plus aisée. Dans ce chapitre, nous allons aborder quelques-unes des adaptations mises en place.

5.1 Itérateur

Comme expliqué dans le chapitre 4, les langages Java et Python n'ont pas les mêmes fonctionnements lorsque l'on souhaite voyager à travers une structure telle qu'une liste ou un dictionnaire. En effet, en Python, ces types de structures sont directement considérées comme itérables, c'est à dire qu'une simple *loop* permet de passer en revue son contenu. En Java, par contre, de telles structures nécessitent la création d'un objet *Iterable* afin de pouvoir passer son contenu en revue. Le fait d'éviter la création de tels objets en Python permet souvent de diminuer la taille du code. Par exemple, la fonction *createTransactionFromMap* se trouvant dans le fichier nommé *ReadFile* montre bien la facilité du langage Python dans ce domaine. Cette fonction est respectivement reprise ci-dessous en Java et Python:

```
1 //create transaction from Map < pattern, supports>
2 public void createTransactionFromMap(Map<String, String > inPatterns,
3                                        ArrayList<ArrayList<NodeFreqT>>
                                               trans){
4
```

```
5       Iterator <Map.Entry<String,String> > iterMap =
              inPatterns.entrySet().iterator();
6       while(iterMap.hasNext()){
7           for(int i=0; i<inPatterns.size(); ++i){
8               Map.Entry<String,String> temp = iterMap.next();
9               String str_pattern = temp.getKey();
10              ArrayList <NodeFreqT> tran_tmp = new ArrayList<>();
11              str2node(str_pattern,tran_tmp);
12              trans.add(tran_tmp);
13          }
14      }
15  }
```

```
1   """
2    * create transaction from dictionary < pattern, supports>
3    * argument 1: a dictionary <String, String>
4    * argument 2: an list of list containing NodeFreqT elements
5   """
6   def createTransactionFromMap(self, inPatternsDict_dict,
          transListOfList_list):
7       for elem in inPatternsDict_dict.values():
8           tran_tmp_list = list()
9           self.str2node(elem, tran_tmp_list)
10          transListOfList_list.append(tran_tmp_list)
```

Cependant, cet avantage a aussi ses inconvénients. En Python, il n'est pas possible de retirer un élément de la structure alors qu'on itère à travers cette dernière. En Java, par contre, c'est possible même si c'est à réaliser avec une très grande prudence. En Python, cela est tous simplement impossible. Il est alors nécessaire de créer une structure permettant d'accueillir les éléments à supprimer et de les supprimer après la fin de l'itération. Un bon exemple est la fonction *prune* présentée dans la section 4.4.1. Néanmoins, ce genre de cas arrive peu fréquemment. Dans l'implémentation *FREQTALS*, ce type de construction est principalement utile lorsqu'il faut éliminer les patterns non-fréquents du dictionnaire de candidat. Son utilisation n'est donc que très ponctuelle.

5.2 L'absence de définition de types

Un autre problème d'un langage comme Python est l'absence de définition de types. Pour un petit programme, ce n'est pas un problème. Par contre, pour un code aussi grand que *FREQTALS* où de nombreuses structures différentes

sont utilisées, ce manque d'informations rend le code plus difficile à comprendre. En Java, chaque fonction reçoit ses arguments avec une définition du type qu'il doit avoir. Si l'argument reçu n'a pas le bon type, une erreur est retournée lors de la compilation. En Python, vu que les types ne sont pas précisés, à moins d'une erreur lors de l'exécution du corps de cette fonction, aucune erreur ne sera lancée. Afin, de remédier à ce problème, il est possible de vérifier les types de chaque argument passé à l'aide de conditions ou d'assertions. Cependant, ces vérifications sont très coûteuses au niveau du temps d'exécution, c'est pourquoi cette solution a été refusée. Bien que l'efficacité du code ne soit pas l'objectif principal de cette traduction, on ne peut pas se permettre de le ralentir démesurément. N'ayant pas de vérification automatique et afin d'aider d'éventuelles modifications du code, j'ai adopté d'autres techniques afin de préciser le type de structure présente dans les arguments et le corps des fonctions.

Tout d'abord, j'ai ajouté des commentaires au-dessus de la majorité des fonctions. Ces commentaires contiennent plusieurs informations :

- Une description de ce que fait la fonction,
- Les paramètres que la fonction requiert ainsi que leur type,
- Une description de ce que la fonction retourne précisant le type de l'élément si nécessaire.

Pour plus de clarté, tous les commentaires sont écrits de la même manière. Cependant, il est possible que seule une partie des informations données ci-dessus s'y retrouvent. En effet, pour certaines fonctions, un des éléments présentés ci-dessus peut se trouver inutile ou évident. Dans ce cas-là, ils ont été omis. Un exemple d'un tel commentaire peut être retrouver dans la section 5.1 au dessus de la fonction *createTransactionFromMap*.

Un autre moyen de clarifier le code est d'ajouter une petite terminaison au nom de variables telles que *_dict* ou *_list*. Cette technique est appelé *intention-revealing variable and method names*. Ces ajouts permettent de comprendre directement le type de structure que contient la variable. Cet ajout a surtout été fait sur les structures rassemblant en leur sein plusieurs éléments ou pour les éléments définis hors du corps de la fonction tels que les variables globales.

5.3 Les structures de données

Les structures de données utilisées sont un élément clé du code. Elles sont très utiles pour stocker les patterns fréquents trouvés ou encore pour contenir la grammaire

des codes Java reçue en *input*. Lors d'une traduction d'un code Java en Python, il est important de comprendre la raison de l'utilisation d'une structure de données plutôt qu'une autre. L'objectif est d'avoir un code Python réalisant les mêmes tâches que le code Java originel. Pour se faire, certains principes doivent selon les cas, rester les mêmes ou être adaptés tout en gardant leurs spécificités.

5.3.1 Les structures créées

Afin de contenir les nœuds composants un pattern ainsi que toutes les autres informations utiles lors de l'exécution du code, il est nécessaire de créer des structures de données pouvant rassembler plusieurs informations en même temps. Ces structures, dont le code peut être retrouvé dans le dossier *structure*, sont très importantes. La traduction de ces fichiers fut quasi-directe et donc peu de grandes modifications ont dû y être faites. L'important étant selon moi que la structure soit pratique à utiliser et la plus efficace possible. Le code Java respectant déjà ces conditions, il en va de soi que ma traduction y est similaire. Une importance a cependant été portée sur le choix des structures utilisées dans ces fichiers. Cette réflexion est développé dans la section 5.3.2.

5.3.2 Les autres structures de données

Pour ce qui est du reste du code, plusieurs structures de données différentes sont utilisées. Dans le code Java, la plupart d'entre elles correspondent à des *HashMap* ou des *ArrayList*. Ces structures de données assez typiques sont respectivement équivalentes aux *dict()* et aux *list()* en Python. En ce qui concerne les listes, leurs tailles sont rarement fixées définitivement dans le code. En effet, il arrive fréquemment qu'on ajoute un nouvel élément dans ces dernières. Une préoccupation quant à la complexité peut être soulevée. Cependant, il est à rappeler que la complexité amortie d'un tel ajout est en $O(1)$. L'utilisation d'une telle structure n'impacte donc quasiment pas le temps d'exécution.

Une autre structure présente dans l'implémentation Java sont les *LinkedList*. L'avantage d'une telle structure est que les insertions et les suppressions se font en $O(1)$. Cependant, pour accéder à l'élément de l'indice n, il est nécessaire de passer les n premiers éléments en revue. Cette caractéristique est assez impactante sur le temps d'exécution. Étant donné que ce type d'accès est assez fréquent, j'ai donc préféré utiliser une *list()* dans ma traduction Python. Ce choix implique que l'insertion a donc une complexité amortie de $O(1)$ au lien d'une complexité en $O(1)$. Par contre, l'avantage est que l'accès à un indice n se fait en $O(1)$ au lieu de $O(n)$.

Dans certains cas, l'ordre a de l'importance. Pour ce faire, en Java, des

LinkedHashMap sont utilisés. De manière similaire, en Python, il suffit d'appeler la librairie *collections* pour obtenir l'accès à des dictionnaires ordonnés appelés *OrderedDict()*.

Enfin, deux autres structures sont encore utilisées en Java mais en bien moindre quantité. Ces structures sont les *HashSet* et les *LinkedHashSet*. Ces structures sont intéressantes. Elles permettent de toujours s'assurer que les éléments contenus en leur sein soient toujours uniques. De plus, ces deux structures utilisent des fonctions de *hash* afin d'accéder aux éléments voulus plus rapidement. Pour traduire la première structures en Python, j'ai utilisé des *set()*. Cette structure Python a pour caractéristique d'être non-ordonnée, n'autorisant pas de modifications de son contenu et n'autorisant pas les doublons. Enfin, cette structure est elle aussi implémentée comme un tableau de *hash*. Pour ce qui est des *LinkedHashSet*, j'ai utilisé une *list()* dans ma traduction. Cependant, je ne suis plus d'accord avec mon choix. En effet, la complexité temporelle des fonctions les plus utiles de la structure *LinkedHashSet* sont en $O(1)$ y compris la fonction *contain*. L'utilisation d'une liste en Python est quant à elle pénalisante. En effet, la complexité de la fonction *contain* est en $O(n)$. Je recommande plutôt d'utiliser une des librairies Python permettant d'utiliser des *SortedSet* équivalents aux *Set()* présentés plus haut mais ordonnés.

5.3.3 Comparaison de structures

Un problème rencontré fut découvert lors de la première exécution de *updateCandidates*. Cette fonction sert à mettre à jour la location d'un candidat lorsqu'une extension possible a été trouvée. Pour se faire, il est nécessaire de passer les patterns existants en revue avec pour but de trouver celui à mettre à jour. En Java, cela a été réalisé en recréant la structure de données recherchée et en la remplissant adéquatement. Ensuite, il suffit d'aller rechercher cette structure dans une *Map* au moyen de la fonction *get*. Pour plus de clarté, un extrait de cette partie du code peut être retrouvé ci-dessous:

```
1 FTArray newTree = new FTArray();
2 newTree.addAll(prefixInt);
3 newTree.add(candidate);
4 Projected value = freq1.get(newTree);
```

Réaliser une traduction directe de ce bout de code n'a cependant pas eu le résultat escompté. En effet, bien que le patterns existent dans le dictionnaire, il n'était jamais trouvé. Le problème étant que ma fonction de *hash* calcule deux valeurs de *hash* proches mais différentes pour les deux structures malgré ces dernières soient identiques. Afin de résoudre ce problème plusieurs options sont possibles. Un

premier moyen est de modifier la fonction de *hash* utilisée par le dictionnaire. Tandis que la deuxième est d'utiliser la fonction *equals* présente dans la classe *FTArray*. N'ayant pas trouvé le moyen de réaliser la première solution, la deuxième fut appliquée. Cette dernière permet d'obtenir une fonction parfaitement opérationnelle dont la partie du code modifiée peut être retrouvée ci-dessous:

```
1 newTree = FTArray()
2 newTree.addAll(prefixInt)
3 newTree.add(candidate)
4 value = None
5 for key in freq1_dict:
6     if newTree.equals(key):
7         value = freq1_dict[key]
```

5.4 Librairies utilisées

En Java, de nombreuses importations peuvent être trouvées au début de chaque fichier. Cela peut être en partie expliqué par le fait que le langage Java impose d'importer la librairie d'une structure avant de pouvoir l'utiliser. En Python, nombre d'éléments sont déjà contenus dans la version de base. Cette solution permet alors de diminuer drastiquement le nombre d'*import*. Globalement, seules les librairies *collection*, *xml.dom*, *os*, *sys* et *time* ont dû être ajoutées au noyau de base de Python et ce pour l'ensemble des fichiers.

Les librairies *unittest* et *traceback* ont aussi été utiles respectivement pour les tests unitaires et pour obtenir plus d'informations sur les erreurs d'exécution.

Seule une librairie utilisée est moins connue. Elle s'appelle *pyjavaproperties*. Cette libraire sert à lire les fichiers *config.properties* dans le but de sauvegarder plus facilement ses données dans une structure appropriée. Cette librairie est en réalité une traduction Python de la librairie Java appelée *Properties*. L'existence de cette librairie Python m'a permis de gagner un temps précieux.

5.5 Conclusion

Pour conclure cette partie sur les différences entre Java et Python, je tiens à souligner que ces deux langages sont assez proches dans la manière de coder. Cependant, leur syntaxe est très différente. Celle de python est plus facile et plus compacte. Cependant, cette dernière caractéristique du langage Python peut engendrer des pertes d'informations, comme par exemple la disparition des types.

Il est donc très important d'y remédier à l'aide de commentaire ou par l'utilisation de nom de variable et de méthode plus révélateur. Un autre point très important d'une traduction Java vers Python est le choix des structures utilisées. Un mauvais choix peut très vite augmenter la complexité de votre code. Il est donc nécessaire de bien s'informer sur les caractéristiques de chacune des structures à traduire afin de choisir celle qui s'adaptera le mieux. Python étant déjà un langage plus lent que Java, il serait dommage d'agrandir cette différence dû à un mauvais choix de structure.

Chapitre 6

Vérification de l'implémentation

Les chapitres précédents abordent l'algorithme et la traduction du code. Une autre étape importante est de vérifier si le code fonctionne correctement. L'ensemble de ce chapitre aborde ce sujet et développe plusieurs techniques de tests utilisées. Chaque technique y est expliquée en y abordant l'objectif poursuivi et ses possibles limites.

6.1 Tests unitaires

Le système de tests le plus évident à utiliser dans ce genre de projet sont les tests unitaires. Ce type de tests permet de vérifier le bon fonctionnement d'une fonction voire même d'une classe. J'ai utilisé ce type de tests pour de nombreuse classes. Comme expliqué dans les chapitres précédents, il m'a été impossible d'exécuter le code tant que la partie minimale nécessaire pour le faire fonctionner n'était pas implémentée. Cette partie minimale est déjà d'une taille conséquente. Il faut au moins avoir implémenté toutes les structures de données, les fichiers capables de lire les *inputs*, la condition de fréquence minimale ainsi que l'algorithme de base permettant de trouver les candidats et d'étendre les patterns. Cette tâche est longue et réaliser cette dernière sans avoir la certitude que l'implémentation soit correcte n'est pas envisageable; c'est pourquoi j'ai implémenté des tests unitaires pour ces fichiers. Mon travail s'est alors organisé comme suit: dans un premier temps, j'ai traduit un fichier Java. Puis, j'ai créé un test unitaire permettant de s'assurer de son bon fonctionnement. Ces tests se composent de petits exemples réalisables de tête. Ils ne sont pas bien compliqués mais suffisants pour trouver des erreurs dans le code. Enfin, si nécessaire, la traduction a été corrigée jusqu'à obtenir une version correcte. Dans ce projet, deux types de tests unitaires différents ont été créés.

Le premier type utilisent des données inventées. Ils se retrouvent majoritairement

dans le dossier *structure*. Comme le nom de dossier l'indique, l'ensemble de ces tests permettent uniquement de vérifier que les structures servant à recueillir les patterns et autres informations soient correctes. Une bonne partie de ces fonctions servent donc à ajouter, obtenir ou retirer des informations à ces structures. Réaliser des tests avec des informations fictives n'empêche pas de vérifier le bon fonctionnement de ces fonctions, c'est pourquoi ces tests n'ont pas été retravaillés afin d'y utiliser seulement des données réelles. Néanmoins, ces tests restent efficaces pour vérifier si une modification du code n'entache pas le code d'erreurs. Ces tests unitaires utilisent la fonction *setUp* afin d'initialiser l'environnement nécessaire pour chaque test effectué. Enfin, chaque fichier de test a pour nom celui du fichier qu'il vérifie concaténé avec le mot *Test*.

Le deuxième type de tests unitaires sont quant à eux basé sur des données réelles. Ils peuvent notamment être retrouvés dans le dossier *input* ou *config*. Ce type de test se base de prêt ou de loin sur un ou des fichier(s) se trouvant dans le dossier *ressources*. Comme expliqué dans la section 4.3, ces fichiers ont été copiés dans le dossier *test* afin de s'assurer que les tests restent corrects en tout temps. A nouveau, la fonction *setUp* a été utilisée afin d'initialiser les variables nécessaires durant les tests. Enfin, les noms de fichiers des tests correspondent toujours à celui du fichier qu'ils testent. Un exemple de ces tests figure en annexe A. Il correspond au test du fichier *ReadXMLInt.py*.

Ces deux types de tests ont été très utiles. En effet, une fois le code minimal implémenté très peu d'erreurs ont dû être corrigées. Cela est dû en grande partie au nombre de tests unitaires créés. Ils couvrent **42.8%** de l'ensemble des fichiers traduits. Les tests présents dans l'implémentation Java ne couvrent quant à eux que **7,41%** des fichiers. Les tests unitaires créés m'ont permis d'anticiper des erreurs éventuelles et de les corriger directement. La couverture de tests existants peut être retrouvée dans le tableau 6.1. Certaines fonctions ne sont pas testées par les tests unitaires. Pour beaucoup d'entre elles, c'est uniquement parce qu'elle rassemblent en leur sein plusieurs appels à des fonctions pour lesquelles ils existent déjà un tests. Réaliser de tels tests semble dès lors répétitif. De bons exemples de ces fonctions se trouvent dans les fichier *ReadFile.py* et *ReadFileInt.py* où les fonctions *createTransactionFromMap* se constituent principalement d'un appel à la fonction *str2node*. Etant donné que ces deux fichiers ne contiennent que deux fonctions, ne pas effectuer de test pour l'une d'entre-elles diminue dramatiquement le pourcentage de couverture. Et ce, malgré que la fonction la plus importante soit testée.

Nom du dossier	Nom du test	Couverture
structure	FTArrayTest.py	96%
	LocationTest.py	100%
	NodeFreqT.py	100%
	ProjectedTest.py	100%
freqT	CheckSubtreeTest.py	60%
config	configTest.py	55%
input	ReadFileTest.py	50%
	ReadFileIntTest.py	50%
	ReadXMLIntTest.py	90.9%

Figure 6.1: Tableau donnant le pourcentage de couverture des tests unitaires

Certains fichiers tels que les *output*, ou les fichiers implémentant le corps de l'algorithme n'ont pas de tests unitaires qui leur sont dédiés. En effet, un autre type de test a été utilisé pour ces derniers. Ils sont développés dans la section 6.2.

6.2 Utilisation de la comparaison avec Java

Un autre moyen de vérifier le bon fonctionnement d'un code est de regarder si ces résultats sont identiques à la réponse attendue. Pour ce faire, une solution est d'utiliser des données à taille humaine afin de pouvoir calculer manuellement les résultats attendus. Il nous est alors permis d'exécuter le code et d'en comparer les résultats à ceux attendus. Cette technique permet par exemple, de regarder facilement si les fichiers traitants les *output* sont corrects. Il suffit dans ce cas-là de regarder si la structure de la réponse est la même. Pour ce qui est de l'algorithme, la comparaison des patterns trouvés est facilement analysable sur des cas de taille réduite.

De plus, dans le cadre de mon mémoire, un code Java fonctionnel existait déjà. Ce code m'a ainsi permis de vérifier ces patterns avec ceux que j'ai trouvés moi-même mais aussi avec les patterns trouvés par le code Java. Ce qui revient à faire une double vérification. En effet, nous ne sommes pas à l'abri d'une erreur éventuelle dans le code Java, d'où l'intérêt de cette double vérification.

Dans le cas où une erreur était trouvée, l'existence d'un code Java pouvait grandement aider à localiser cette dernière. En effet, à l'aide du fichier *util*, il est possible d'imprimer le contenu de toutes les structures de données présentes dans le code. Il est donc possible de créer quelques points de passage permettant de vérifier si le contenu de ces dernières sont les mêmes en Java et en Python. De plus, dans le cas d'un exemple à taille humaine, il est aussi possible d'exécuter

l'algorithme mentalement afin de vérifier l'exactitude de ses données ou de vérifier l'exactitude d'une contrainte.

Dans le cas de problèmes à taille humaine, utiliser un tel système de vérification n'est pas compliqué. Cependant, pour des problèmes plus larges, les résultats sont plus difficilement comparables. Il est donc nécessaire de pouvoir automatiser ce genre de vérification.

Une solution possible est de comparer les fichiers ligne par ligne. Si chacune des lignes sont identiques indépendamment des espaces alors le comparateur en conclura que le code est correct. Dans l'autre cas, il y a une erreur quelques part. Cette erreur peut correspondre à un pattern manquant en Java ou en Python ou encore à des patterns incomplets. Cependant, cette technique contient aussi de nombreux désavantages. Tout d'abord, elle ne permet pas de donner une quelconque information utile sur l'origine de la différence trouvée. Ensuite, un simple passage à la ligne dans un des deux fichiers peut entraîner une catégorisation différente. Enfin, l'ordre des patterns peut différer d'un code à l'autre. L'ensemble de ces raisons amène à la conclusion que cette méthode de vérification ne peut pas être fiable.

Une autre technique de vérification possible a pour but de tirer avantage type de fichier de sortie contenant les patterns. Ce fichier est une extension *.XML*. Il existe pour ce type de fichiers des librairies Python permettant de les lire facilement. Le but serait d'utiliser une de ces librairies afin de pouvoir librement voyager dans les données du fichier. Cela permettrait de retrouver les paires de patterns même si ces derniers se trouvent dans un ordre différent. De plus, une différence dans l'écriture du fichier comme par exemple, un retour à la ligne, n'aura plus d'impact sur le résultat de la comparaison. Ce système de vérification serait donc plus fiable. Et pourquoi ne pourrait-on pas suggérer de mettre en évidence les patterns incomplets, incorrects ou manquants en les imprimant dans un autre fichier? Cette solution semble réalisable et permettrait de vérifier automatiquement n'importe quel input quelle que soit sa taille. Il serait intéressant de mettre en place dans le futur ce deuxième moyen de vérification bien plus efficace que le système actuel qui n'est autre que la première solution présenté ci-dessus.

6.3 Test unitaire de la main

Le code Python créé est voué à évoluer et à être modifié lors de recherches futures. Il est par conséquent important de fournir un système de vérification plus facile,

plus efficace et moins fastidieux qu'une comparaison manuelle. Pour se faire, deux tests unitaires appelés *Test_Main_one_step.py* et *Test_Main_two_step.py* existent. Ces derniers vérifient la justesse des patterns obtenus pour les deux types de recherches possibles. Pour ce faire, les patterns obtenus sont comparés aux patterns de référence à l'aide du comparateur "ligne par ligne" expliqué plus haut. Ces patterns de références ont auparavant été vérifiés manuellement et attestés comme corrects. L'utilisation du comparateur "ligne par ligne" est possible dans ce cas-ci car le code Python est déterministe. Les patterns trouvés seront par conséquent toujours dans le même ordre et selon la même disposition. Enfin, pour toutes modifications du code Python n'impactant pas les résultats trouvés, ces tests continueront de fonctionner et vous assureront de la justesse des patterns trouvés. Dans le cas d'une modification impactant les résultats finaux tels que l'ajout d'une nouvelle contrainte, une adaptation du fichier contenant les patterns de référence sera nécessaire. Par contre, le code implémentant les tests ne devra subir aucune modification. Le test *Test_Main_one_step.py* peut être trouvé en annexe B.

6.4 Conclusion

Un effort a été fait afin d'avoir un maximum de moyen de vérification. Les tests unitaires développés sont largement plus nombreux que dans l'implémentation Java. Ces tests permettent d'assurer le bon fonctionnement des fonctions testés et seront très utiles en cas de modification de l'une d'entre elle. En plus de ces tests unitaires, des comparaisons de résultats entre le code Python, Java et de solutions résolues manuellement ont été faites. Afin de faciliter ce type de tests, deux systèmes de vérification ont été proposés. L'un d'entre eux est implémenté mais n'est pas aussi efficace que souhaité. Il serait donc intéressant d'implémenter le second dans le futur. Enfin, des tests unitaires permettant de vérifier le bon fonctionnement de l'entièreté de l'algorithme existent. En cas de modification du code n'impactant pas les patterns, l'exécution de ces tests permettront d'assurer que le code reste fonctionnel.

Chapitre 7

Résultats

L'objectif de mon mémoire était d'obtenir un code Python fonctionnel et traduit selon les bonnes pratiques d'implémentation de Python. Ce chapitre aborde les résultats selon plusieurs axes. Dans un premier temps, une analyse des patterns extraits est effectuée. Dans un second temps, je réalise une analyse basée sur la qualité du code Python. Enfin, je termine par comparer l'efficacité du code Java par rapport à sa traduction.

7.1 Résultat de l'extraction de patterns

Pour ce qui est des résultats en terme de patterns trouvés, on y retrouve peu de surprises. En effet, la traduction produite a pour but de recréer le code existant dans un autre langage. Dès lors, il semble logique que les patterns extraits par la traduction Python soit les mêmes que ceux extraits par le code Java. Étant donné que les résultats sont les mêmes, je ne vais pas m'attarder sur ces derniers. De fait, une analyse détaillée et complète des patterns trouvés a déjà été faite dans la section 5 du papier "MiningPatterns in Source Code Using Tree Mining Algorithms", [1]. Outre cette analyse, il est intéressant de relever d'autres aspects évoqués ci-après.

7.2 Qualité du code

L'objectif de cette traduction était d'être écrite dans un style Python correct mais surtout être facilement compréhensible pour un lecteur extérieur. Pour satisfaire ces critères, plusieurs éléments ont été mis en place.

Premièrement, les noms de variables choisis sont primordiaux. J'ai utilisé des noms de variables et de fonctions révélateurs de leur rôle dans le code et/ou de leur type.

Plusieurs exemples peuvent être trouvés dans les codes du chapitre 4.

Ensuite, le code est divisé en plusieurs petites fonctions. Les fonctions les plus imposantes regroupent en leur sein plusieurs appels à de plus petites fonctions. Cependant, leurs tailles restent respectables mais surtout facilement compréhensibles grâce aux noms assez explicites des fonctions qui les composent.

Enfin, j'ai ajouté de nombreux commentaires pour améliorer la compréhension du code. Afin de mieux illustrer mes propos, je vais évoquer quelques chiffres. Le code Python se compose de 3321 lignes de code et 922 lignes de commentaires. Le code Java quant à lui est constitué de 3978 lignes de code et de 650 lignes de commentaires. A première vue, on peut déjà observer que le nombre de commentaires dans ma traduction est nettement supérieur. Cependant, un meilleur critère de comparaison est donné par le pourcentage de commentaires présents dans chaque code. Les commentaires du code Python représentent **21.73%** de l'ensemble du code tandis qu'en Java, ce rapport est seulement de **14.04%**. Chaque fichier ne comprend pas toujours le même nombre de commentaires. En effet, les fichiers implémentant les structures sont majoritairement composés de fonction *get* et *set* qui ne nécessitent pas d'explications supplémentaires. Par contre, les fichiers représentant le cœur de l'algorithme sont quant à eux largement plus commentés que la moyenne. En effet, le rapport commentaires/nombre de lignes atteint généralement les **33%**. Ce dernier n'est que de **20%** dans le code Java.

Afin d'avoir un indice plus précis de la qualité de mon code, j'ai fait appel à la librairie *pylint*. Cette dernière permet d'analyser un code et lui donne une note de qualité sur 10. Cependant, il n'existe pas de *lower bound* sur cette dernière. Il n'est donc pas rare d'obtenir une note négative. L'analyse brut de mon code par cette librairie a donné une note de **2.56/10**. Cette note doit être relativisée. En effet, la librairie est assez stricte et n'autorise que très peu de modifications de ses standards. Par exemple, mes noms de variables ne correspondent pas à ceux demandés par la librairie. Dès lors, ces noms me font perdre des points malgré qu'ils soient uniformisés et clairs. Un simple retrait de l'analyse des noms de variables et de la longueur des lignes augmente ma note pour atteindre un score de **5.36/10**.

Pour conclure, le code Python est propre et bien commenté. Il est donc assez facile de le comprendre même sans l'accès à ce mémoire. Enfin, les chiffres donnés dans le paragraphe précédent montrent que le code Python est bien plus court que le code Java. Dans la réalité, la différence est moins flagrante. Le code Python est bel et bien plus compact grâce à ses simplifications d'écriture et aux facilités

qu'il offre lors de l'utilisation de structures de données telles que des listes ou des dictionnaires. Cette comparaison ne reste-t-elle pas biaisée par le fait que le code Java contient plusieurs fonctions non-utilisées, et par conséquent non traduites en Python. L'omission de ces dernières permet en effet de creuser l'écart de taille entre les deux codes.

7.3 Comparaison de l'efficacité des deux implémentations

Bien que l'efficacité du code Python ne soit pas l'objectif principal, cette qualité ne peut être mise de côté. Dans cette section se trouve une comparaison entre l'efficacité de l'implémentation Python et Java. Cette comparaison commence par quelques rappels des performances de ces deux langages. Ensuite, une analyse plus précise et chiffrée de l'efficacité des implémentations est réalisée.

Une grande différence entre le langage Java et Python est que ce premier est un langage compilé alors que Python est un langage interprété. Cette différence a de nombreuses conséquences. L'avantage de Python est sa meilleure gestion des erreurs, sa portabilité et sa simplicité de mise en place. Java, par contre, a pour avantage d'être rapide lors de l'exécution après la compilation et permet d'obtenir de meilleures performances. Il sera par conséquent, difficile pour mon code Python de rivaliser avec les performances du code Java. [2; 3]

Afin de vérifier la théorie, il est temps d'étudier la performance des deux implémentations. Pour se faire, des tests ont été réalisés sur quatre codes Java différents. Ces tests ont été effectués pour les deux types d'exécution possibles, soit celle en une étape et celle en deux étapes. Il est aussi à noter que les tests ont tous été réalisés avec un support égal à 2. Ce choix de support permet d'éviter que l'ensemble du code ne se trouve comme patterns fréquents tout en augmentant au maximum le nombre de patterns à trouver. Pour plus d'informations sur les options de configuration utilisées, elles ont été ajoutés dans l'annexe C.

Dans les tableaux 7.1 et 7.2 se trouvent les résultats obtenus. Seuls deux des quatre mesures effectuées sont présentées ci-dessous. Il est toutefois possible de retrouver les deux dernières dans l'annexe D. Afin de mieux interpréter les résultats par la suite, quelques explications sur les noms des mesures semblent nécessaires.

Tout d'abord, la mesure nommé *initiation* chronomètre le temps nécessaire à la lecture et la sauvegarde des informations données en *input*.

Le nom *extension* correspond à la mesure du temps nécessaire à l'extension des patterns lors de la première étape. Tandis que la mesure *extension_2_étapes* correspond quant à elle aux temps d'exécution nécessaire afin d'étendre les patterns maximaux de petites tailles trouvés lors de l'étape 1.
Enfin, la mesure *FREQTALS* donne, quant à elle, le temps d'exécutions nécessaires afin de réaliser l'ensemble des étapes de l'algorithme *FREQTALS*. Il comprend au minimum la somme des temps *initiation* et *extension*. Dans le cas d'une exécution, en deux étapes, le temps *extension_2_étape* vient s'ajouter à ce total. Quelques bouts de code se déroulant entre ces 3 blocs principaux viennent augmenter légèrement le temps total.

Ensuite, les temps nommés *forestMatcher* et *commonPattern* mesurent respectivement le temps nécessaire afin de localiser les patterns trouvés dans le code Java donné en *input* et le temps nécessaire pour trouver des patterns en commun dans chaque morceau de code localisé par l'étape *forestMatcher*.

Enfin, le temps nommé *Total* correspond comme son nom l'indique au temps nécessaire à l'exécution de l'ensemble du code.

		Implémentation Java		Implémentation Python	
		une étape	deux étapes	une étape	deux étapes
Visitor	initiation	0,225 s	0,238 s	0.156 s	0,234 s
	extension	0,051 s	0,073 s	2,094 s	2, 094 s
	extension_2_étapes	/	0,202 s	/	2,562 s
	FREQTALS	0,412 s	0,531 s	2,266 s	4,9 s
	forestMatcher	1,705 s	1,726 s	1,875 s	1,96 s
	commonPattern	1,733 s	1,697 s	1,947 s	1,97 s
	Total	3,982 s	4,072 s	6,089 s	9,017 s

Table 7.1: Tableau comparant le temps d'exécution des implémentations Java et Python sur les deux types d'exécution pour les ASTs du code Java *Visitor*.

		Implémentation Java		Implémentation Python	
		une étape	deux étapes	une étape	deux étapes
Prototype	initiation	0,245 s	0,214 s	0,249 s	0,129 s
	extension	0,035 s	0,048 s	0,798 s	0,797 s
	extension_2_étapes	/	0,160 s	/	0,426 s
	FREQTALS	0,421 s	0,443 s	1,064 s	1,366 s
	forestMatcher	1,638 s	1,693 s	1,846 s	1,815 s
	commonPattern	1,646 s	1,620 s	1,867 s	1,846 s
	Total	3,871 s	3,897 s	4,777 s	5,027 s

Table 7.2: Tableau comparant le temps d'exécution des implémentations Java et Python sur les deux types d'exécution pour les ASTs du code Java *Prototype*.

D'après ces résultats plusieurs éléments intéressants peuvent être soulignés. Premièrement, le code Python est plus lent que le code Java. Cependant, toutes les parties de l'algorithme ne sont pas logées à la même enseigne.

Tout d'abord, le temps nécessaire pour l'initialisation du code est relativement semblable. Par contre, le temps nécessaire à la recherche des patterns s'avère nettement plus élevé. Cette recherche permettant d'agrandir les patterns existants requiert la majorité du temps d'exécution quel que soit le type d'exécution choisi. Plusieurs facteurs pourraient expliquer de telles différences.

Un premier facteur concernerait une différence dans le nombre d'appels récursifs effectués par l'implémentation Java et Python. En effet, les mesures nommées *extension* et *extension_2_étapes* donnent le temps passé dans les fonctions récursives servant à chercher des extensions aux patterns. S'il s'avère qu'une des contraintes permettant d'éliminer les patterns est plus relaxée en Python qu'en Java, le nombre de patterns éliminés diminuerait. Par conséquent, le nombre d'appels récursifs se verrait augmenté. Dès lors, l'exécution de cette recherche prendrait davantage de temps. Cependant, après vérification, cette possibilité est à mettre de côté. Le nombre d'appels récursifs est le même dans les deux implémentations.

Un deuxième facteur est que les structures utilisées en Python sont moins efficaces que celles utilisées dans le code Java. Une discussion à propos des structures utilisées en Python a déjà été faite dans le chapitre 5.3.2. Globalement, le choix de ces dernières a été fait en prenant en compte l'efficacité. Seule une structure s'avère être largement moins performante que son homologue Java. Cependant, cette dernière n'entre pas en ligne de compte lors du calcul du temps d'exécution des mesures *extension* et *extension_2_étapes*.

Un troisième facteur résiderait probablement dans la perte d'optimisation de certaines traductions. Une première explication pourrait être la modification discutée dans la section 5.3.3. La solution choisie est moins efficace que l'implémentation Java. La modifier permettra sûrement d'améliorer l'efficacité du code Python.

La dernier facteur concernerait les différences de performances entre les langages Python et Java. Cependant, et avant tout il faudrait d'abord optimiser le code Python au maximum. Une fois celle-ci réalisée, il sera possible d'observer si cette différence a réellement un impact.

Il est a noté que le nombre d'appels récursifs démultiplie les différences d'efficacité.

Ces appels étant réalisés plus d'une centaine de fois, une petite modification peut très vite impacter grandement les performances.

Enfin, les temps d'exécution appelés *forestMatcher* et *commonPattern* sont eux aussi légèrement plus élevés dans l'implémentation Python. L'exécution du fichier *forestMatcher.jar* est le point central de ces mesures. La seule explication possible est que le code Python serait plus lent lors de l'appel de ce dernier.

7.4 Conclusion

Pour conclure, le code Python créé permet d'obtenir les mêmes résultats que le code Java. Il est cependant largement plus commenté. Ses commentaires ainsi que les noms de variables et de fonctions révélateurs aident grandement à la compréhension du code. De plus, la librairie *pylint* permettant d'évaluer la qualité de mon code Python lui donne un score correct.
En ce qui concerne l'efficacité du code, ce dernier est moins performant que le code Java. Ce résultat était prévisible pour plusieurs raisons. Tout d'abord, le langage Java est réputé comme plus performant que le langage Python. Ensuite, le code Java a été fortement optimisé afin de limiter au maximum les pertes de performance. Le code Python, même s'il a été traduit dans le même esprit, la qualité de sa traduction - but principal -, ne peut égaler un code dont la performance a été l'objectif principal. Enfin, un mauvais choix de structure et quelques traductions perdant une partie de leur optimisation peut aussi expliquer une moindre performance. A fortiori lorsque le code demandant le plus de temps est une fonction récursive, ses défauts seront forcément démultipliés.

Chapitre 8

Pistes d'améliorations

L'objectif de mon mémoire était de traduire l'algorithme *FREQTALS* d'un code Java en un code Python. Cet objectif a été rempli. En plus d'être fonctionnel, le code contient aussi des tests unitaires pour un certain nombre de fichiers. Cependant, un tel projet peut encore être perfectionné, c'est pourquoi je vais donner quelques pistes d'améliorations possibles dans ce chapitre. Ces pistes servent à situer le travail accompli mais aussi de projeter ce mémoire par rapport à des travaux futurs autour de ce sujet.

8.1 Création comparant les patterns trouvés en Python et en Java

Comme expliqué dans le chapitre 6, pour le moment, il n'existe pas encore de code fiable permettant de vérifier et de comparer les résultats du code Python et Java. La création d'un code comparant les fichiers *XMLs* produits par les deux implémentations, permettrait de s'assurer définitivement de son bon fonctionnement, évidement sous condition que le code Java soit correct. Pour plus de précisions sur la manière de réaliser un tel code, je vous renvoie à l'explication faite à la fin de la section 6.2.

8.2 Correction des erreurs trouvées dans le code Java

En parallèle au présent mémoire, un autre mémoire réalisé par Quentin Hauspie, a pour objet de poursuivre le développement du code Java existant, plus précisément de l'algorithme *FREQTALS*, en y ajoutant des contraintes. Lors de son travail, il a pu constater quelques erreurs dans le code Java. Ces erreurs n'impactent pas

trop les patterns trouvés, c'est pourquoi j'ai continuer ma traduction sans en tenir compte. L'objectif étant de les corriger une fois la traduction terminée. Afin de faciliter les corrections à effectuer dans le code Python, Quentin a pris note des modifications nécessaires. Il serait donc intéressant de réaliser ces modifications, de tester à nouveau le code et comparer les deux systèmes une nouvelle fois.

8.3 Implémentation des tests unitaires Python en Java

Une autre amélioration possible mais qui cette fois touche plus au code Java serait d'implémenter en Java les tests unitaires que j'ai créés pour le code Python. Étant donné que le code Python servirait de laboratoire pour créer de nouvelles contraintes ou réaliser des modifications avant de les transposer en Java, il serait intéressant d'avoir des systèmes de test pour les deux codes. Cela permettrait de s'assurer de la justesse des modifications de part et d'autre.

8.4 Modification de structures

Lors de la traduction, le type de structure à utiliser était une décision importante. En effet, ces dernières peuvent grandement influencer l'efficacité de l'implémentation. Lors de l'éléboration de celles-ci, j'ai dû faire plusieurs choix. Ces derniers sont expliqués dans la section 5.3.2. Cependant, un des choix ne fut pas des plus pertinents. En effet, les *LinkedHashList* en Java ont été traduits par des listes en Python. Ce choix impacte grandement l'efficacité du code dû à une complexité plus élevée en Python. Afin d'améliorer ce point, il serait utile de modifier les *list()* par des *SortedSet*.

8.5 Amélioration de l'efficacité des fonctions récursives d'expansion

Le chapitre 7.3 a montré que l'efficacité des fonctions récursives permettant d'étendre les patterns n'est pas aussi bonne que celle en Java. Il serait donc intéressant d'optimiser ces dernières. Une optimisation possible est d'améliorer le code présenté dans la section 5.3.3. En effet, ce dernier est appelé à chaque récursion. Le code Python a une complexité en $O(n)$ alors que le code Java effectue cette opération en $O(1)$. Après des centaines de récursions, cette modification de complexité peut avoir de grosses conséquences sur la performance. Une analyse approfondie mettra fort probablement en évidence d'autres améliorations possibles.

Chapitre 9

Conclusion

Ce mémoire s'inscrit dans le cadre d'un début de programme de recherche.

L'algorithme *FREQTALS* a pu être entièrement implémenté en Python. Complète et fonctionnelle, cette traduction offre l'avantage d'être largement commentée ; ce qui facilite grandement sa compréhension. D'autres techniques telles que des noms de variables clairs et uniformisés précisant le cas échéant le type de variables, des fonctions courtes et propres renforcent également cette compréhension. L'ensemble de ces techniques a été réfléchi et implémenté dans le but d'épouser au mieux le futur rôle de cette traduction. En effet, le code Python aura pour rôle de servir de laboratoire pour la création de nouvelles contraintes qui seront par la suite intégrées au code Java. Il était donc primordial que la traduction reste assez fidèle au code Java. Dès lors que les deux implémentations sont relativement proches, il est effectivement beaucoup plus simple de réimplémenter une contrainte en Java si le code Python utilisé pour la développer se structure de la même façon. Évidemment, des modifications/adaptations ont dû être faites lors de l'implémentation en Python. Cependant, elles ont été réalisées en respectant l'esprit du code Java.

Plusieurs systèmes de vérification ont été mis en place afin de tester et de vérifier la justesse du code et sa bonne fonctionnalité. En outre, des comparaisons de résultats obtenus lors de différents points de passage ont pu être effectuées avec l'implémentation Java.

Le code Python créé à l'occasion de ce mémoire constitue un bon point de départ pour continuer les recherches sur les contraintes déjà entamées. Des améliorations de ce code peuvent bien entendu être apportées en vue d'aider les futurs chercheurs. Quelques propositions ont d'ailleurs été faites dans le chapitre 8.

Annexe A

Exemple de test unitaire

Le test unitaire qui suit permet de tester le bon fonctionnement de l'ensemble des fonctions contenues dans le fichier *ReadXMLInt.py*. Il est à noter que ce test utilise des données réelles sauvegarder à l'abri de toutes modifications dans le dossier *Test*.

```
1  import unittest
2
3  from freqt.src.be.intimals.freqt.input.ReadXMLInt import *
4  from freqt.src.be.intimals.freqt.structure.NodeFreqT import *
5
6  from xml.dom import minidom
7  from xml.dom import Node
8
9
10 class MyTestCase(unittest.TestCase):
11
12     def setUp(self):
13         self.RXML = ReadXMLInt()
14         self.RXML.lineNrs = []
15         self.RXML._id = 0
16         self.RXML._top = 0
17         self.RXML._sr = []
18         self.RXML._sibling = []
19         self.RXML._labels = []
20         self.RXML.countSection = -1
21         self.RXML._abstractLeafs = False
22
23     def test_getlineNrs(self):
24         self.assertEqual(self.RXML.getlineNrs(), [])
```

```
25
26      def test_countNBNode(self):
27          doc = minidom.parse("../../../../../test/Basic/ast1.xml")
28          doc.documentElement.normalize()
29          self.assertEqual(self.RXML.countNBNodes(doc.documentElement) +
                1, 17)
30
31      def test_countNBChildren(self):
32          doc = minidom.parse("../../../../../test/Basic/ast1.xml")
33          doc.documentElement.normalize()
34        self.assertEqual(self.RXML.countNBChildren(doc.documentElement),
                3)
35          children = doc.documentElement.childNodes
36          i = 0
37          correct_output_child = [3, 1, 2]
38          for child in children:
39              if child.nodeType != Node.TEXT_NODE and child.nodeType ==
                    Node.ELEMENT_NODE:
40                  self.assertEqual(self.RXML.countNBChildren(child),
                        correct_output_child[i])
41                  i += 1
42
43      def test_readWhiteLabel(self):
44          correct_output = dict()
45          children = set()
46          children.add("B")
47          children.add("D")
48          correct_output["A"] = children
49          self.assertEqual(self.RXML.readWhiteLabel("../../../../../
                test/Basic/listWhiteLabel.txt"),
                correct_output)
50
51      def test_updateLabelIndex(self):
52          # cas 1
53          nodeLabel = "A"
54          trans = [NodeFreqT(), NodeFreqT(), NodeFreqT]
55          labelIndex = {}
56          output_labelIndex = dict()
57          output_labelIndex[0] = "A"
58          self.RXML.updateLabelIndex(nodeLabel, trans, labelIndex)
```

```
59          self.assertEqual(trans[0].getNode_label_int(), 0)
60          self.assertEqual(labelIndex, output_labelIndex)
61          # cas 2
62          nodeLabel = "B"
63          output_labelIndex = dict()
64          output_labelIndex[0] = "A"
65          output_labelIndex[1] = "B"
66          self.RXML.updateLabelIndex(nodeLabel, trans, labelIndex)
67          self.assertEqual(trans[0].getNode_label_int(), 1)
68          self.assertEqual(labelIndex, output_labelIndex)
69          # cas 3
70          nodeLabel = "A"
71          output_labelIndex = dict()
72          output_labelIndex[0] = "A"
73          output_labelIndex[1] = "B"
74          self.RXML.updateLabelIndex(nodeLabel, trans, labelIndex)
75          self.assertEqual(trans[0].getNode_label_int(), 0)
76          self.assertEqual(labelIndex, output_labelIndex)
77
78      def test_findLineNr(self):
79          doc = minidom.parse("../../../../../test/Basic/ast1.xml")
80          doc.documentElement.normalize()
81          children = doc.documentElement.childNodes
82          i = 0
83          correct_output = ["2", "6", "7"]
84          for child in children:
85              if child.nodeType != Node.TEXT_NODE and child.nodeType ==
                      Node.ELEMENT_NODE:
86                  self.assertEqual(self.RXML.findLineNr(child),
                            correct_output[i])
87                  i += 1
88
89      def test_countSectionStatementBlock(self):
90          doc = minidom.parse("../../../../../test/Basic/ast1.xml")
91          doc.documentElement.normalize()
92          self.RXML.countSectionStatementBlock(doc.documentElement, "0")
93          self.assertEqual(self.RXML.countSection, -1)
94          self.RXML.countSection = 2
95          self.RXML.countSectionStatementBlock(doc.documentElement, "1")
96          self.assertEqual(self.RXML.countSection, 3)
```

```
97          self.assertEqual(self.RXML.lineNrs, [1])
98
99      def test_populateFileListNew(self):
100         directory2 = '../../../../../resources/input-artificial-data/
                abstract-data'
101         directory = '../../../../../test/Harder/version1'
102         file_list = []
103         file2_list = []
104         correct_output_file_list = ['../../../../../test/Harder/
                version1/builder/Builder.xml',
105                                     '../../../../../test/Harder/
                                            version1/builder/
                                            GridBagLayout_Builder.xml',
106                                     '../../../../../test/Harder/
                                            version1/builder/
                                            GridLayout_Builder.xml',
107                                     '../../../../../test/Harder/
                                            version1/builder/
                                            JTable_Builder.xml',
108                                     '../../../../../test/Harder/
                                            version1/builder/
                                            TableBuilderDemo.xml',
109                                 '../../../../../test/Harder/version1/
                                            builder/TableDirector.xml',
110                                     '../../../../../test/Harder/
                                            version1/visitor/Book.xml',
111                                 '../../../../../test/Harder/version1/
                                            visitor/Fruit.xml',
112                                 '../../../../../test/Harder/version1/
                                            visitor/ItemElement.xml',
113                                     '../../../../../test/Harder/
                                            version1/visitor/
                                            ShoppingCartClient.xml',
114                                     '../../../../../test/Harder/
                                            version1/visitor/
                                            ShoppingCartVisitor.xml',
115                                     '../../../../../test/Harder/
                                            version1/visitor/
                                            ShoppingCartVisitorImpl.
                                            xml']
```

```
116         self.RXML.populateFileListNew(directory2, file2_list)
117         print(file2_list)
118         self.RXML.populateFileListNew(directory, file_list)
119         self.assertEqual(file_list, correct_output_file_list)
120 
121     def test_calculatePositions(self):
122         trans = []
123         self.RXML._sr = [0, 1, 2, 3, 4, 5, 6]
124         doc = minidom.parse("../../../../../test/Basic/ast1.xml")
125         doc.documentElement.normalize()
126         size = self.RXML.countNBNodes(doc.documentElement) + 1
127         correct_output_parent = [-1, -1, -1, -1, -1, -1, 5, -1, -1, -1,
                -1, -1, -1, -1, -1, -1, -1]
128         correct_output_child = [-1, -1, -1, -1, -1, 6, -1, -1, -1, -1,
                -1, -1, -1, -1, -1, -1, -1]
129         correct_output_sibling = [-1, -1, -1, -1, -1, 6, -1, -1, -1,
                -1, -1, -1, -1, -1, -1, -1, -1]
130         for i in range(size):
131             nodeTemp = NodeFreqT()
132             nodeTemp.nodeFreqtInit(-1, -1, -1, "0", True)
133             trans.append(nodeTemp)
134             self.RXML._sibling.append(-1)
135         self.RXML.calculatePositions(trans)
136         for i in range(len(trans)):
137             self.assertEqual(trans[i].getNodeParent(),
                      correct_output_parent[i])
138             self.assertEqual(trans[i].getNodeChild(),
                      correct_output_child[i])
139             self.assertEqual(self.RXML._sibling[i],
                      correct_output_sibling[i])
140 
141     def test_readTreeDepthFirst(self):
142         doc = minidom.parse("../../../../../test/Basic/ast1.xml")
143         doc.documentElement.normalize()
144         size = self.RXML.countNBNodes(doc.documentElement) + 1
145         node = doc.documentElement
146         trans = []
147         correct_output_parent = [-1, 0, 1, 2, 1, 4, 1, 6, 0, 8, 9, 8,
                11, -1, -1, -1, -1]
148         correct_output_label = ['A', 'B', 'F', '*f1', 'F', '*f2', 'F',
                '*f3', 'D', 'D1', '*d1', 'D2', '*d2', '', '', '', '']
```

```
149         correct_output_child = [1, 2, 3, -1, 5, -1, 7, -1, 9, 10, -1,
                12, -1, -1, -1, -1, -1]
150         correct_output_sibling = [-1, 8, 4, -1, 6, -1, -1, -1, -1, 11,
                -1, -1, -1, -1, -1, -1, -1]
151         for i in range(size):
152             nodeTemp = NodeFreqT()
153             nodeTemp.nodeFreqtInit(-1, -1, -1, "0", True)
154             trans.append(nodeTemp)
155             self.RXML._sibling.append(-1)
156         labelIndex = {}
157         whiteLabel = self.RXML.readWhiteLabel("../../../../../test/
                Basic/listWhiteLabel.txt")
158         self.RXML.readTreeDepthFirst(node, trans, labelIndex,
                whiteLabel)
159         for i in range(len(trans)):
160             self.assertEqual(trans[i].getNodeLabel(),
                    correct_output_label[i])
161             self.assertEqual(trans[i].getNodeParent(),
                    correct_output_parent[i])
162             self.assertEqual(trans[i].getNodeChild(),
                    correct_output_child[i])
163             self.assertEqual(trans[i].getNodeSibling(),
                    correct_output_sibling[i])
164
165
166 if __name__ == '__main__':
167     unittest.main()
```

Annexe B

Exemple de test unitaire de la main

Le test unitaire suivant permet de vérifier le bon fonctionnement du code lors d'une exécution en une seule étape en comparant les patterns obtenus avec les patterns de référence. Cette comparaison est faite à l'aide de la fonction contenue dans le fichier *Comparator.py* comparant les deux fichiers reçus en input ligne par ligne.

```
1  #!/usr/bin/env python3
2
3  import sys
4  import unittest
5
6  from freqt.src.be.intimals.freqt.Main import *
7  from freqt.src.be.intimals.freqt.Comparator import *
8
9
10 class MyTestCase(unittest.TestCase):
11
12     def test_main_one_step(self):
13         args_main_one_step =
                ["..\\..\\..\\..\\test\\TestMain\\config\\design-
                patterns\\builder\\config.properties", "2",
                "builder"]
14         # verify the correctness of one-step execution
```

```
15          args_comparator_pattern_one_step = ["comparator", "..\\..\\..」
                \\..\\test\\TestMain\\Correct_output\\design-」
                patterns\\builder\\builder_2_patterns.xml",
                "..\\..\\..\\..」
                \\test\\TestMain\\Current_output\\design-」
                patterns\\builder_2_patterns.xml"]
16          main(args_main_one_step)
17          sys.stdout.flush()
18          time.sleep(0.01)
19          value1 = comparator(args_comparator_pattern_one_step)
20          self.assertEqual(value1, "The files are identical")
21
22
23  if __name__ == '__main__':
24      unittest.main()
```

Annexe C

Configuration utilisée pour comparer l'efficacité des implémentations Java et Python

Les sections suivantes reprennent l'ensemble des configurations utilisées afin de réaliser les tests d'efficacité. Il est à rappeler que le support utilisé était égale à deux.

C.1 *config.properties*

Afin de réaliser ces tests pour les deux types d'exécution, il fut nécessaire de changer la valeur de l'argument *twoStep* à *true* pour le second type. Le reste ne nécessite aucune modification.

```
1  #Python configuration
2
3  inputPath =
       ../../../../resources/input-artificial-data/design-patterns
4  outputPath =
       ../../../../resources/output-artifical-data/design-patterns
5
6  #timeout (minutes)
7  timeout = 1
8
9  #leaf size constraints: using in the first step to limit the size of
       the search space
10 minLeaf = 2
11 maxLeaf = 4
```

```
12
13 #node size constraints: using to remove small patterns
14 minNode = 10
15
16 #mining patterns method :
17 twoStep = false
18 # true  - using 2 steps to mine maximal patterns:
19     # step 1: find frequent maximal patterns with size constraints
20     # step 2: grown frequent pattern to find maximal patterns
21 # false - find maximal patterns in 1 step
22
23 #filter maximal patterns method:
24 filter = true
25 # true  - directly filter maximal patterns in the mining process;
26 # false - filter maximality after having a list of frequent patterns
27
28 #abstract leaf label.
29 abstractLeafs = false
30 # true  - replace all leaf labels by **
31 # false - using leaf labels
32
33 #build grammar: true - build grammar from input data; false - read
       grammar from given file
34 buildGrammar = true
35 #file contains a list of root labels
36 rootLabelFile = ../../../../resources/conf-artifical-data/design-
       patterns/listRootLabel.txt
37 #file contains a list of label that only allow in patterns
38 whiteLabelFile = ../../../../resources/conf-artifical-data/design-
       patterns/listWhiteLabel.txt
39 #file contains a list of xml characters
40 xmlCharacterFile = ../../../../resources/conf-artifical-data/design-
       patterns/xmlCharacters.txt
41
42 #configurations for running parallel:
43 #list of minimum support thresholds
44 minSupportList = 2
45 #list of folders
46 inFilesList = builder,factorymethod,prototype,visitor
```

C.2 *listRootLabel.txt*

Ci-dessous se trouve le contenu du fichier *listRootLabel.txt* utilisé:

```
#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
Block
```

C.3 *listWhiteLabel.txt*

Le contenu du fichier *listWhiteLabel.txt* peut être retrouvé ci-dessous:

```
#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. Lines begin with # are comments
#i.e. SimpleName has 2 children, identifier and var.
#.    If we want to expand identifier and ignore var we can create a line as SimpleName identifier

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

#bodyDeclarations
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
```

C.4 *xmlCharacters.txt*

Le contenu du fichier *xmlCharacters.txt* utilisé se trouve ci-dessous:

```
#list of root labels (AST Nodes)
<       &lt;
>       &gt;
&       &amp;
```

Annexe D

Mesure des temps d'exécution

Dans cette annexe se trouve deux autres comparaisons du temps d'exécution des implémentations Java et Python. Ces mesures ont été prises en utilisant comme *input* les ASTs des codes Java *Builder* et *Factorymethod*.

		Implémentation Java		Implémentation Python	
		une étape	deux étapes	une étape	deux étapes
Builder	initiation	0,273 s	0,270 s	0,388 s	0,353 s
	extension	0,126 s	0,117 s	7,56 s	7,96 s
	extension_2_étapes	/	24,84 s	/	60,37 s
	FREQTALS	0,574 s	25,247 s	7,98 s	68,7 s
	forestMatcher	1,745 s	1,738 s	2,014 s	2,3 s
	commonPattern	1,776 s	1,774 s	2,008 s	2,3 s
	Total	4,245 s	28,89 s	12,018 s	73,32 s

Table D.1: Tableau comparant le temps d'exécution des implémentations Java et Python sur les deux types d'exécution pour les ASTs du code Java *Builder*.

		Implémentation Java		Implémentation Python	
		une étape	deux étapes	une étape	deux étapes
Factory-method	initiation	0,199 s	0,199 s	0,09 s	0,126 s
	extension	0,019 s	0,038s	0,296 s	0,306 s
	extension_2_étapes	/	0,124 s	/	0,09 s
	FREQTALS	0,375 s	0,381 s	0,410 s	0,536 s
	forestMatcher	1,616 s	1,615 s	1,85 s	1,92 s
	commonPattern	1,646 s	1,656 s	1,87 s	1,86 s
	Total	3,786 s	3,770 s	4,15 s	4,33 s

Table D.2: Tableau comparant le temps d'exécution des implémentations Java et Python sur les deux types d'exécution pour les ASTs du code Java *Factorymethod*.

Annexe E

Lien vers le GitHub de la traduction

Le code de la traduction Python peut être retrouvé dans son intégralité sur le GitHub: `https://github.com/loicq256/FREQTALS---Thesis`.

Bibliographie

[1] Pham H.S. et al. (2019) Mining Patterns in Source Code Using Tree Mining Algorithms. In: Kralj Novak P., Šmuc T., Džeroski S. (eds) Discovery Science. DS 2019. Lecture Notes in Computer Science, vol 11828. Springer, Cham. https://doi.org/10.1007/978-3-030-33778-0_35

[2] Necas F. (Avril 2020) Python vs Java en 2020. AXOPEN. https://blog.axopen.com/2020/04/python-vs-java-2020/

[3] Mouhtat B. (Novembre 2020) Java VS Python : Quel langage est le meilleur ?. Cours Gratuit. https://www.cours-gratuit.com/tutoriel-python/java-vs-python-quel-langage-de-programmation-est-le-meilleur

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