UCLouvain

epl

École polytechnique de Louvain

Mining Tree Patterns from Source Code

Author: **Mariusz Losicki**
Supervisors: **Siegfried Nijssen, Kim Mens**
Reader: **Axel Legay**
Academic year 2018–2019
Master [120] in Computer Science

Abstract

The increased availability of large, widely used, and open-source software systems has sparked the interest of researchers from different communities. One possible use of this data is to build new developments tools. These tools can help programmers analyse a previously unseen codebase, get familiar with a new language, or write idiomatic code. However, finding how to infer knowledge from source code is still an active topic of research.

Here, we focus on using pattern mining techniques and the kind of modifications they require to extract this knowledge in the form of code idioms. In particular, we explore whether closed tree patterns and tree patterns that compress the data can successfully be used when mining for idioms in source code.

This thesis was made in the context of an ongoing research project, whose goal is to develop a software tool which would assist in legacy software modernisation. We describe our contributions to their development effort of the mining process as well.

Acknowledgements

I would like to thank my supervisors, Siegfried Nijssen and Kim Mens, for their time and support throughout this semester.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**

2 Background **3**

2.1 Frequent pattern mining 3

2.1.1 Problem definition 3

2.1.2 FreqT 5

3 Related work **6**

3.1 Existing research 6

3.1.1 Code idioms 7

3.1.2 Code clones 7

3.2 INTiMALS 8

3.2.1 Source code importer 8

3.2.2 Preprocessor 9

3.2.3 Pattern miner 9

3.2.3.1 Grammars 10

3.2.3.2 Constraints 11

4 Mining closed idioms **13**

4.1 Background 13

4.2 Implementation 15

4.3 Limitations 17

5 Parsing the ASTs **22**

5.1 Background 23

5.2 Possible parser 25

5.2.1 Modelisation 26

5.2.1.1 Definition 26

5.2.1.2 Equivalences 29

5.2.2 Practical application 31
5.3 Limitations 33

6 Mining compressing patterns **35**
6.1 Background 36
6.2 Related 37
6.3 Inducing a TSG 38
6.4 Model description 40
6.5 Implementation 42
6.6 Validation 43
6.6.1 Datasets 43
6.6.2 Experiments 46

7 Conclusion and future work **50**

A Appendix A **52**

B Appendix B **53**

Bibliography **54**

List of Figures

3.1 Representation of the architecture. 9

4.1 Example of computing closed patterns 15
4.2 Experiments with closed pattern mining 18
4.3 Example of an undesirable interaction between closed and leaf constraint 19
4.4 Example of an undesirable interaction between closed and root whitelist constraint 20

5.1 Example of the grammar for `InfixExpression` 27
5.2 Example of parent annotation of a grammar 28
5.3 Example of non-LL(1) construct 30

6.1 Inserting patterns into a TSG 39
6.2 Characteristics of the datasets 46
6.3 Summary of results obtained for compressing patterns 47
6.4 Example of patterns found 48

Chapter 1

Introduction

In the past few years, data mining on large software repositories has become a research topic of increased interest among the machine learning, programming languages and software engineering communities. Conferences regarding this specific topic are also organized since many years [27]. This thesis takes part in this research process and investigates whether *frequent pattern mining* techniques can be used to gain insight into a previously unseen codebase. The representation of choice for source code is the abstract syntax tree. Many different frequent pattern mining algorithms can and have been applied on tree-shaped data but there is still little research on their usefulness when applied to source code, a very specific type of data. This thesis investigates whether closed tree patterns or tree patterns that compress data the most can be useful when looking for code idioms, syntactic fragments that occur frequently across software projects and have a single semantic purpose [4].

This thesis was made in the context of an ongoing research project, INTiMALS [22], whose ultimate goal is to build an intelligent modernisation assistant for legacy software systems. This assistant would provide recommendations of modernisation actions based on insight learned from existing large source code repositories. The framework aims to work with any kind of programming language and follows a pipeline architecture with five separate components: source code importer, mining preprocessor, pattern miner, pattern matcher and modernisation assistant. At the start of this thesis, all of the five components were already partially implemented but their integration was limited [29]. We will focus here on only one of this component: the pattern miner.

The INTiMALS research project is a university-industry collaboration between UCLouvain, the Vrije Universiteit Brussel [37] and Raincode [31], a company which specialises in the maintenance and modernisation of large and complex legacy codebases written in older languages like, for example, COBOL. The usefulness and advantage of having such an assistant is clear: it helps with the migration process, provides immediate insight when starting work with a

new repository for junior and experienced developers alike.

This thesis is structured as follows: in Chapter 2 we present the required knowledge of frequent pattern mining. We follow with a review of prior research related to this work and a description of the existing INTiMALS approach, Chapter 3. Chapter 4 describes mining closed pattern and the limitations of this approach. In Chapter 5 we discuss the need of parsing the ASTs to learn a probabilistic context-free grammar (PCFG) and its implications. Chapter 6 describes the approach to mine patterns to compress the data and its results. Chapter 7 concludes.

Chapter 2

Background

In order to have a better understanding of this thesis, some basic knowledge from different domains is required. This chapter will focus on concepts and notions related to frequent pattern mining. When necessary, the following chapters will start by introducing the additional background needed. Overall, besides frequent pattern mining, we will also discuss topics such as (statistical) parsing and the minimum description length principle.

2.1 Frequent pattern mining

A frequent pattern is defined as an itemset, subsequence or structure that occurs with a frequency above a given threshold in the transactions of a database [13]. In our case, a transaction is the abstract syntax tree (AST) of a single source code file and the database is a set of those files e.g. a repository. The patterns we are interested in are subtrees, i.e. snippets of code.

2.1.1 Problem definition

More formally, a database $\mathcal{D} = \{T_0, T_1, \ldots, T_N\}$ is a set of N transactions where each transaction is a tree. A transaction and a pattern are both *rooted*, *labelled trees* (V, E, L, r) where V is a set of vertices, E is a set of edges, L is a set of labels, and r is the root of the tree where $r \in V$. A labelling function $l : V \mapsto L$ exists and assigns a label to each vertex.

Rooted trees can be *ordered* or *unordered* depending on whether there is a predefined ordering between siblings, i.e. vertices that share the same parent. Unless specified otherwise, we are considering rooted labelled ordered trees with a left-to-right ordering. Following this definition, we can define the left and right siblings of a node as well as the *leftmost* and *rightmost* child. By repeatedly taking the rightmost child, starting from the root, we obtain the *rightmost* path.

We are interested in subtrees, but multiple definitions exist depending on how the matching between the subtree and the tree is done. The type of subtrees we are considering are *induced* subtrees. A tree $Y' = (V', E', L', r')$ is an induced subtree of $Y = (V, E, L, r)$ if and only if there is a mapping function φ. This function provides a one-to-one mapping between the vertices of V' and V that preserves the vertex labels, the adjacency of each vertex and the root [13]. We write that $\exists \varphi \colon V' \rightarrowtail V$ such that

$$\forall u, v \in V' \colon (u, v) \in E' \iff (\varphi(u), \varphi(v)) \in E \tag{2.1}$$

Intuitively, the induced subtree Y' can be obtained by repeatedly removing leaf nodes from Y. We say that Y is a *supertree* of Y'.

A pattern $\mathcal{P}$ is a (induced) subtree of T if $\mathcal{P} \subseteq T$ where $T \in \mathcal{D}$. This relation allows us to define an indicator function to decide whether a pattern $\mathcal{P}$ occurs in a tree T

$$covers(P, T) = \begin{cases} 1 & \text{if } \mathcal{P} \subseteq T, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

The number of transactions in which the pattern occurs is given by the size of the cover:

$$support_{\mathcal{D}}(\mathcal{P}) = |\{T \in \mathcal{D} \mid covers(P, T) = 1\}| \tag{2.3}$$

This way of defining the support makes it an anti-monotonic function. This property, which is often referred to as the Apriori principle [3], is a useful one as it allows to prune very large parts of the search space and mine patterns in large databases [2]. Mining algorithms typically need to generate candidate patterns by either extending or combining frequent patterns discovered previously. This property allows to reduce the number of candidates considered as any extension of an infrequent pattern is also infrequent.

The minimum support threshold *minsupp* is a parameter such that $1 \leq minsupp \leq N$ transactions and the goal of frequent ordered tree mining is to find $\{\mathcal{P} \subseteq T \mid support_{\mathcal{D}}(\mathcal{P}) \geq minsupp\}$, the set of all subtrees that are more frequent than given a threshold.

It is clear that the size of this set can potentially be huge and most of the subtrees in this set will be useless. This is a classic and long known problem in the data mining community working on pattern mining [2]. Many different techniques were researched and proposed to find only interesting patterns. The key issue is that interestingness is, by definition, subjective. There is not a single general measure of interestingness that we can formalize and that will satisfy everyone [2]. This thesis explores whether closed patterns and compressing

patterns can successfully be used when using source code as data.

2.1.2 FreqT

Depending on the type of input data, different types of frequent pattern mining algorithms were proposed. The most common types of data include itemsets, sequences, and graphs. Here, we are only interested in one specific algorithm which mines inside trees, a particular type of graph. Many different algorithms for mining subtrees exist, but an initial choice has already been made by the research group and we decided to continue with it.

The FreqT algorithm, proposed in [7], discovers all frequent tree patterns from a large collection of labeled ordered trees. It is an iterative pattern mining strategy based on the Apriori principle and a depth-first exploration. The two key components of this algorithm are its efficient candidate generation and occurrence storage.

FreqT relies on a rightmost path expansion [7] for the candidate generation phase. To produce candidates of size $k+1$ from frequent patterns of size k, only extensions of nodes on the rightmost path of the tree are considered. This way of generating candidates is different from the methods used before, i.e. merging patterns of size k which have $k-1$ nodes in common. It also allows collecting extensions from the data efficiently.

The second component, which makes this algorithm efficient memory-wise, is that it does not need to store the complete occurrence of a pattern. For each pattern, we only record to which nodes in the database the rightmost node can be mapped. The reason for this is related to the previous key component: only the vertices on the rightmost path of each occurrence of a pattern need to be extended. Those paths can be easily derived from the occurrence of the rightmost node that we have stored.

The notations and concepts presented in this section should be a sufficient introduction to frequent tree mining. A more in-depth overview of all the concepts related to frequent pattern mining in general can be found in [2] and related to only frequent subtree mining in [13].

Chapter 3

Related work

In this chapter, we review some of the existing research related to our problem. Even though researchers have studied many source code related tasks, only a couple of recent papers directly approaching the problem of mining code idioms. We will also review the current INTiMALS approach and, especially, their pattern miner.

3.1 Existing research

The increase of large, high-quality and open-source repositories has sparked the interest of researchers from different communities. One of the more popular research fields is based around using statistical models for source code. Their use is supported by two interesting observations.

The first observation is that source code corpora have statistical properties similar to natural language corpora [6]. Researchers applied techniques used in natural language processing (NLP) with promising results [21]. The success of statistical models used in NLP to solve difficult tasks like, for example, machine translation is also a reason why those models were considered and tested on software corpora.

The second observation is that software contains a lot of redundancies. This observation may seem intuitive and was actually confirmed empirically in [17] where 430 million lines of source code where analyzed. With a granularity level low enough (between one to seven lines), a general lack of uniqueness was observed.

There are many tasks for which those models can be useful ranging from auto-completion [5] to generating programs [26]. Other researched topics include API mining [1] and code clone detection [9, 8]. However, the topic we are interested in are code idioms.

3.1.1 Code idioms

Regarding mining code idioms from source code specifically, the paper "Mining Idioms from Source Code" by Allamanis and Sutton [4] is probably the most relevant one for our work and, as the title suggests, describes a technique to mine code idioms from a corpus of software projects.

In their work, the problem of finding code idioms consists in inferring commonly reoccurring fragments in ASTs. To do that, they apply a new technique related to statistical NLP introduced in [14] by Cohn, Blunsom, and Goldwater. This paper describes a new way of inducing a probabilistic tree substitution grammar (PTSG). A more formal definition can be found in Chapter 6 but, informally, a tree substitution grammar is simply an extension of a context-free grammar (CFG) where a production rule can potentially generate a tree structure at once. The advantage of those *tree rules* is that they can keep some local context, something that CFGs cannot do. A probabilistic version of a grammar also has a probability associated with each production rule.

This new technique proposed to learn a TSG is to use *non-parametric Bayesian methods* [4, 13]. The reason for using those methods is that they can automatically infer an appropriate model depending on the size and complexity of the data. To illustrate, let us take clustering as an example. Typically, one would have to define a prior on how many clusters there are, but it is not always easy to do as this number is usually unknown and can potentially be infinite. Non-parametric methods address this by removing the need of defining the number of parameters beforehand and by growing them depending on the complexity of the data. The same reasoning can be applied to the TSG inference problem. It is difficult to decide how many tree rules, which are parameters in the model, we require. Non-parametric Bayesian models address this problem by growing this number depending on the size and complexity of the input ASTs. Once the TSG is learned, the code idioms are the tree rules that can be produced by this grammar.

3.1.2 Code clones

One of the closest fields of research related to code idiom mining is code clone detection. Indeed, one could think that such techniques could also be used for code idiom mining. However, Allamanis and Sutton argue that code clones are not code idioms. The reason is that code detection tools attempt to find the largest fragment that is copied but code idioms require a trade-off between size and the frequency with which programmers would use it in order to appear

"natural" and be considered useful [4]. Their experiments in which they compare both mined code idioms and code clones seem to confirm this theory. It remains to be seen whether this is also the case for frequent pattern mining techniques used to mine code idioms. Indeed, syntactic approaches to find code clones are also based on ASTs and tree-matching. An overview of the techniques explored in this research field can be found in [33].

3.2 INTiMALS

Another related research project is that of the INTiMALS group, whose ultimate goal is to build an intelligent modernisation assistant for legacy software systems. This tool's knowledge would be based on code idioms mined from high-quality and idiomatic source code. Currently, the development effort is focused on the core of the project, mining code idioms. Later on, the miner should also be capable of mining for procedure calls. This change effectively requires to consider graphs as input data instead of trees. Another possible improvement is to consider the versioning history of a repository as additional input. The assistant tool also has many possible improvements such as providing recommendations on the fly, finding almost-matches or even missing idioms in a user's source code.

This thesis was made alongside this research project but the foundation is based on their existing work. This section describes the essential parts that impact our work and which understanding is necessary for later.

As mentioned in the introduction, the framework follows a pipeline architecture with five separate components: source code importer, mining preprocessor, pattern miner, pattern matcher and modernisation assistant. The first three components are the ones in which we are interested and require some description. Figure 3.1 gives a visual representation of the process and data flow.

3.2.1 Source code importer

The source code importer is the key component in making the framework language-independent. Given a set of source files, the role of the importer is to produce their respective ASTs in a specific predefined format. This format, called the *metamodel*, defines a standard representation of abstract trees (in XML) and allows the subsequent components to work on this single format independently of the input language. Note that each programming language

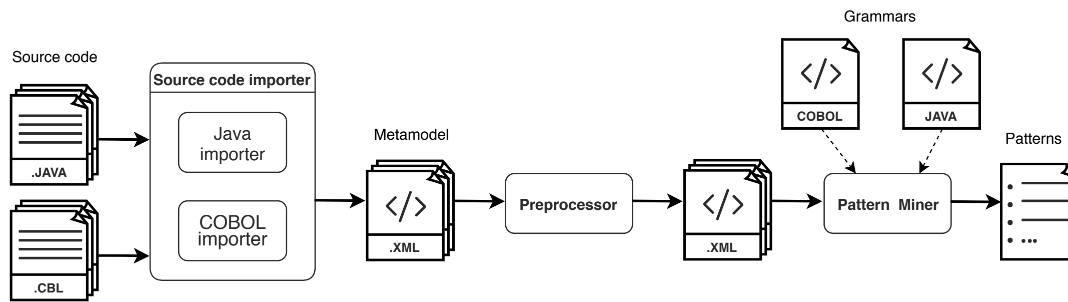


FIGURE 3.1: Representation of the architecture. Only the first three components related to mining code idioms are represented.

requires an importer. Currently, there is an importer for Java based on the Eclipse compiler [16] and an importer for COBOL based on a proprietary compiler developed by Raincode.

3.2.2 Preprocessor

The mining preprocessor takes as input the generated ASTs and applies different kinds of transformations depending on the specified settings. This is especially useful for the next component, the pattern miner, as depending on the transformations applied, we can mine different types of patterns without changing the mining algorithm. Currently, one of the possible transformations is splitting a single identifier node into multiple subnodes whenever they follow the camel case convention.

3.2.3 Pattern miner

The pattern miner is based on the FreqT algorithm, which, as explained in Section 2.1.2, mines for rooted ordered trees. This may seem surprising when we recall that our input data is source code. Depending on the programming language used, we can write some parts of code in any order without changing the semantics (e.g. method declarations in a Java class). Partially-ordered trees could be considered a more appropriate model for the transactions and some mining algorithms for those types of trees already exist [23]. However, our goal is to find techniques to discover interesting ones and simply using another type of frequent pattern mining algorithm would not work [4].

The unmodified algorithm outputs *all* the frequent patterns, even those that an expert would not consider interesting or even a code idiom whatsoever. Also, the runtime can be quite long depending on the number of files in the repository and the sizes of the ASTs. The current approach to tackle both phenomena is

to add some constraints which reduce the output size and the search space considered.

3.2.3.1 Grammars

Some of those constraints require the grammar of the language in order to be implemented. This is illustrated in Figure 3.1 as the grammar files next to the pattern miner component. These are classic XML files but, conceptually, they represent the context-free grammar of the associated programming language. The grammar respects a common representation in XML regardless of the programming language. The structure follows the structure of ASTs very closely: an XML node defines the possible children of an AST node. Some children can be optional and are marked as such in the XML with an `optional` attribute set to `true`. By default, if this attribute is not mentioned, the child is mandatory. Each child also has at least one of the following XML attributes: `simplevalue`, `node` or `nodelist`.

A `simplevalue` indicates a literal in the source code, e.g. an operator, a number, etc. This is a terminal if we follow the CFG terminology.

A `node` points to another XML node which defines the structure of this child. This is the next production rule that will be applied for this child.

A `nodelist` is similar to the `node` attribute but can generate an unbounded list of children. This list can be ordered or unordered e.g. in Java, statements are ordered but the methods in a class can be declared in any order.

A special attribute called `abstract` can be set on main XML nodes. This attribute is useful to express choices. For example, in Java, many things can be an expression (`InfixExpression`, `ThisExpression`, etc.). Whenever a rule requires an expression, we use the `Expression` which defines a production that can pick between concrete rules. Another way to interpret this is that `Expression` is a supertype of `InfixExpression`, `ThisExpression` and others. A concrete example extracted from the Java grammar can be found in Appendix A.

Even though, in theory, one may think that a grammar is relatively easy to obtain for programming languages, in practice, this task is much more complicated than anticipated. First of all, depending on the programming language, the grammar can be complex and hard to write down. Even for Java, the grammar required a lot of additional (often manual) work to have a working version. Another issue is that programming languages change over time and have different versions/dialects. To solve this issue, we would require different versions of a grammar and also their maintenance. Finally, the tool is aimed primarily at legacy code and older languages. Writing a grammar for them may require

expertise and may not be available as an open-source resource. This happens to be the case for the COBOL grammar which is not a trivial task to get right.

For this reason, also a different approach was tested. Instead of defining the grammar, we can learn it from input data. The downside is that some constraints may prune less than with a fully fledged-out grammar. We also lose some information about the order as we cannot infer whether a node should be considered as an ordered list. The upside is that, by dropping the requirement of having a predefined grammar, we also drop the issues explained above regarding the difficulties linked with this requirement. Currently, both approaches are evaluated. We specify whenever necessary which grammar, defined or learned, we consider.

3.2.3.2 Constraints

We discussed previously that the unmodified FreqT algorithm outputs *all* the frequent patterns. However, many of those patterns are not code idioms and thus not interesting. The FreqT was modified and a set of constraints was introduced as a possible solution to this problem.

The constraints that were added, in addition to the minimum support constraint which is present by default, are:

Minimum size of trees
: Patterns should be greater than a given size

Maximum size of trees
: Do not expand patterns larger than a given size

Expand to leaf nodes on the left side of the pattern
: Every leaf node of a reported pattern should also be a leaf in the AST (except on the rightmost path)

Maximum repetition of a label in a pattern
: Stop expanding if any label appears more than the given amount of times

Maximum repetition of a label in unordered nodelist
: Stop expanding if any label appears more than a given amount of times as a child of a node

Minimum number of leaves
: Patterns should have the given amount of leaf nodes

Maximum number of leaves

Patterns should not have more leaves than the given amount

Expand mandatory children

Patterns must have expanded all the children that are mandatory, i.e. children not marked as optional in the grammar

Note that there is a distinction between leaf nodes of patterns and ASTs. The constraints related to leaves enforce that leaf nodes of a pattern are also leaf nodes of ASTs, meaning that leaf nodes of the pattern are terminals of our grammar. Later on, even stronger constraints were added. They slightly differ from the previous ones as they reduce the search space more explicitly:

Root whitelist

Only start expanding patterns from a given list of root nodes

Children blacklist

Discard the expansion of specific children of a node

While all the constraints mentioned above are indeed reducing the output size and the search space, defining the optimal values for those parameters is a hard task. Compared to hyperparameter optimisation, there is not a single objective function on which we can optimise. Moreover, depending on which kind of code idioms we are interested in, a different configuration of parameters may be more appropriate. In the following chapters, we look if techniques, such as closed or compressing patterns which are more formal and have stronger foundations, are useful for mining code idioms.

Chapter 4

Mining closed idioms

In this chapter, we are going to describe our first tested approach. The goal is to reduce the number of patterns reported by the modified FreqT algorithm explained in Chapter 3. Indeed, since the algorithm reports *all* frequent patterns, their number is quite significant. However, we would prefer to reduce the size of the output to make it more interpretable. We also observed that many reported patterns are subtrees of some other, larger pattern. This larger pattern is usually the only one in which we are interested.

To alleviate those two issues, we are going to mine only *closed* patterns. We say that a pattern p is closed when no supertree of p has the same support as p. Mining closed patterns, in all kind of structures ranging from itemsets to trees, is not something new but, to the best of our knowledge, was not applied in the context of mining in source code. Also, the approach looks promising considering the results obtained by the initial algorithm, and we do not lose information when applying this technique. Indeed, only keeping the set of closed patterns along with their support can be seen as a form of lossless compression.

First, we introduce some notations and related work in mining closed frequent patterns. We continue by describing how this type of mining was integrated into our existing approach. Finally, we talk about the limitations of this approach which appeared quite quickly.

4.1 Background

The paper written by Chi et al. [12] describes their algorithm, $CMTreeMiner$, which mines only *closed* and *maximal* frequent subtrees in rooted unordered trees. The techniques described in their paper serve as a basis for our implementation as it works almost out-of-the-box with only minor modifications required.

More formally, we follow the same definitions as they defined in their paper [12]. We define the *blanket* of a frequent tree t as the set of trees:

$$B_t = \{t' \mid \text{removing a leaf or the root from } t' \text{ can result in } t\} \tag{4.1}$$

The blanket B_t of t is the set of all supertrees of t that have one more vertex than t. This additional vertex is written as $t' \setminus t$ and can be distinguished based on its position in the tree. Additionally, we define the *right* blanket B_t^R as the subset of B_t where $t' \setminus t$ is on the rightmost path of t'. The $left$ blanket B_t^L are all the other vertices left that are not on the rightmost path of t'.

$$B_t^R = \{t' \in B_t \mid t' \setminus t \text{ is on the rightmost path of } t'\} \tag{4.2}$$

$$B_t^L = B_t \setminus B_t^R \tag{4.3}$$

A frequent subtree t is *closed* if and only if, for every $t' \in B_t$, $support_D(t') < support_D(t)$.

We say $t' \in B_t$ and t are *occurrence-matched* if, for each occurrence of t, there is at least one corresponding occurrence of t'. We also say $t' \in B_t$ and t are *support-matched* if, for each transaction $s \in D$ such that $covers(t, s) = 1$, we have $covers(t', s) = 1$. The key difference is that, in the first case, we compare with each *occurrence* and, in the second, we compare for each $transactions$.

With these definitions, we can define two subsets of B_t:

$$B_t^{SM} = \{t' \in B_t \mid t' \text{ and } t \text{ are support-matched}\} \tag{4.4}$$

$$B_t^{OM} = \{t' \in B_t \mid t' \text{ and } t \text{ are occurrence-matched}\} \tag{4.5}$$

The set B_t^{SM} is useful to decide whether t is closed. Indeed, t is closed if and only if $B_t^{SM} = \emptyset$. The second set, B_t^{OM} is useful to prune some candidates for which we know that they never lead to a closed pattern. Intuitively, if, for every occurrence of a pattern t, we can find an extension $t' \setminus t$, then, by definition, this pattern t is not closed because of t'. If this extension is not on the rightmost path, it means that we can never generate t' as a candidate starting from t because we only generate candidates by adding vertices to the rightmost path. It also means that we can prune t and all of its candidate extensions as they are not closed. Using formal notations, a subtree t can be pruned if $\exists t' \colon t' \in B_t^{OM} \wedge t' \in B_t^L$, i.e. $B_t^{OM} \cap B_t^L \neq \emptyset$.

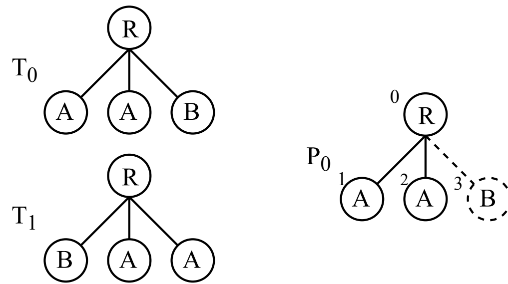


FIGURE 4.1: An example with two transactions where it is important to consider order as it changes the possible matching and blanket of the pattern

4.2 Implementation

In order to add this feature to the modified pattern mining algorithm in use, FreqT, some modifications are needed. Recall that FreqT only keeps the occurrences of the rightmost node of the pattern and not its full occurrence. To compute the blanket, i.e. the set of possible one-vertex extensions, the complete occurrence of the pattern must be known. FreqT can be easily modified to store all those occurrences. It does not do it because the rightmost path extension, as explained in Section 2.1.2, does not need this information. Even if our implementation allows enabling and disabling the closedness check, all the occurrences would be stored anyway. Also, some patterns are a significant overhead on memory, e.g. a pattern that tries to match ten method declarations in a factory class of thirty methods in total. However, an advantage of mining closed patterns is that we can prune some candidates that do not lead to a closed pattern and thus reduce the search space.

Another difference to take into consideration is that the algorithm for *CM-TreeMiner*, presented in Section 4.1, works on rooted *unordered* trees. When working with ordered trees, as in our case, one must be careful when computing the blanket and the support of each extension. Indeed, when matching in an ordered fashion instead of unordered, some transactions may not be in the support of the pattern anymore. This has a direct impact on whether a pattern can be considered closed. Figure 4.1 illustrates this situation. We find a database with two transactions, T_0 and T_1, along with a pattern, P_0. At start, the pattern P_0 is present in both transactions regardless of order. In other words, $D = \{T_0, T_1\}$ and $support_D(P_0) = 2$. Now we consider extending P_0 with the node B and thus producing a new candidate, P'_0. If we consider the transactions as unordered, the supports remains the same at two. The previous pattern, P_0, is thus not closed by definition as P'_0, a supertree of P_0, has the same support. However, in the ordered case, P'_0 does not cover T_1 anymore and

$support_D(P'_0) = 1$. In this small example, we know that both P_0 and P'_0 are both closed as there are no more possible candidates to extend the patterns.

Another relevant observation can be made using this Figure 4.1. Once again, let us consider the situation at start and with only P_0 under consideration. We are interested in finding the blanket of P_0, i.e. B_{P_0}. In practice, the blanket only needs to store *extensions* that lead to a supertree, P'_0, of P_0. An extension is a quadruple $(i_{root}, l, i_{prev}, b)$ where $i_{root,prev} \in I \cup \{-1\}$, the set of node identifiers given by the preorder traversal of a tree, $l \in L$, the set of labels and $b \in \{0, 1\}$. i_{root} is the position in the preorder traversal of the node we are extending. l is the label that we will use. i_{prev} gives the node identifier of the left sibling if the extension is added. b indicates whether the extension is in the left or right blanket with $b = 1 \iff P'_0 \in B^R_{P_0}$. We add -1 to indicate that there is no sibling.

To illustrate, we consider the example in Figure 4.1. The blanket of B_{P_0} is a set of two trees which (coincidentally) is the set of trees represented by T_0 and T_1. We can also write this set as a set of two extensions where the extension of P_0 given by the tree in T_0 is $(0, \mathrm{B}, 2, 1)$. Similarly, the extension given by the tree in T_1 is $(0, \mathrm{B}, -1, 0)$.

The two last values in our extensions require some explanation. Keeping the left sibling is needed because we consider ordered trees. We already illustrated the importance of this in the previous paragraph. In our algorithm, we visit all the occurrences of a pattern and try to match the extension, but we need to know precisely at which position among the children of the node located at i_{root}. Keeping track whether the extension is on the rightmost path is also important when deciding if we can prune the current candidate and all of its descendants. In this example, we can prune neither P_0 nor P'_0 because of the extension in T_0. If T_0 was removed from the database, we would satisfy the conditions to prune this branch of the search space.

The implementation can be decomposed into four key methods: compute the blanket of a given pattern, check whether the pattern can be pruned, verify if it is a closed pattern and prune root candidates of patterns that cannot lead to a closed pattern. The last method can be seen more as an optimisation because the output would be the same whether we use it or not but is nonetheless useful to reduce the search space by not considering some labels as roots of patterns. It is based around the observation that if, for every occurrence of a node with a label l_i, there is a node with a label l_j such that l_j is always the label of the parent of l_i, then a pattern starting with l_i as root label cannot be closed. This pruning technique, for which they coined the term "Search space reduction

using the backward scan", is explained in [25]. It consists in an initial scan of the database to check whether possible pattern root label always has another label as parent. If yes, we can remove this root from the initial set of roots to be expanded.

Data: P, the currently evaluated pattern
Data: Occ_P, the list of all occurrences of this pattern
Result: The set of frequent closed tree patterns
Procedure PROJECT(P, Occ_P)

```
    B_P ← blanket(Occ_P);
    canPrune ← occurrenceMatch(B_P^L);
    if canPrune ∨ other constraint violated then
        return;
    end
    isClosed ← B_P^SM = ∅;
    if isClosed ∧ other constraints respected then
        Add P to output;
    end
    C ← generateCandidates(Occ_P);
    C ← prune(C);
    foreach candidate (P_k, Occ_{P_k}) ∈ C do
        PROJECT((P_k, Occ_{P_k}))
    end
```

Algorithm 1: Modified FreqT algorithm to mine only closed patterns

The first three key methods of our closedness implementation can be seen in action in Algorithm 1. Based on all the occurrences of the pattern, we build the set of possible extensions. From those extensions, we consider only those in the left blanket and check whether there is one which matches all the occurrences of the pattern. In that case, we can stop generating candidates from this pattern and return. Before adding the pattern to the output, we have to check whether it is closed or not. We follow the definition and check whether there is a supertree with the same support. If there is, the pattern is not closed (but also not pruned) and we continue generating candidates from it. We prune this set of candidates based on our constraints and recursively call the procedure on those that are left. The constraints expressed in this algorithm are a subset of those defined in Section 3.2, depending on the configuration given.

4.3 Limitations

We have presented how we integrated mining closed patterns into the existing FreqT algorithm. However, during the evaluation phase, it appeared to be clear

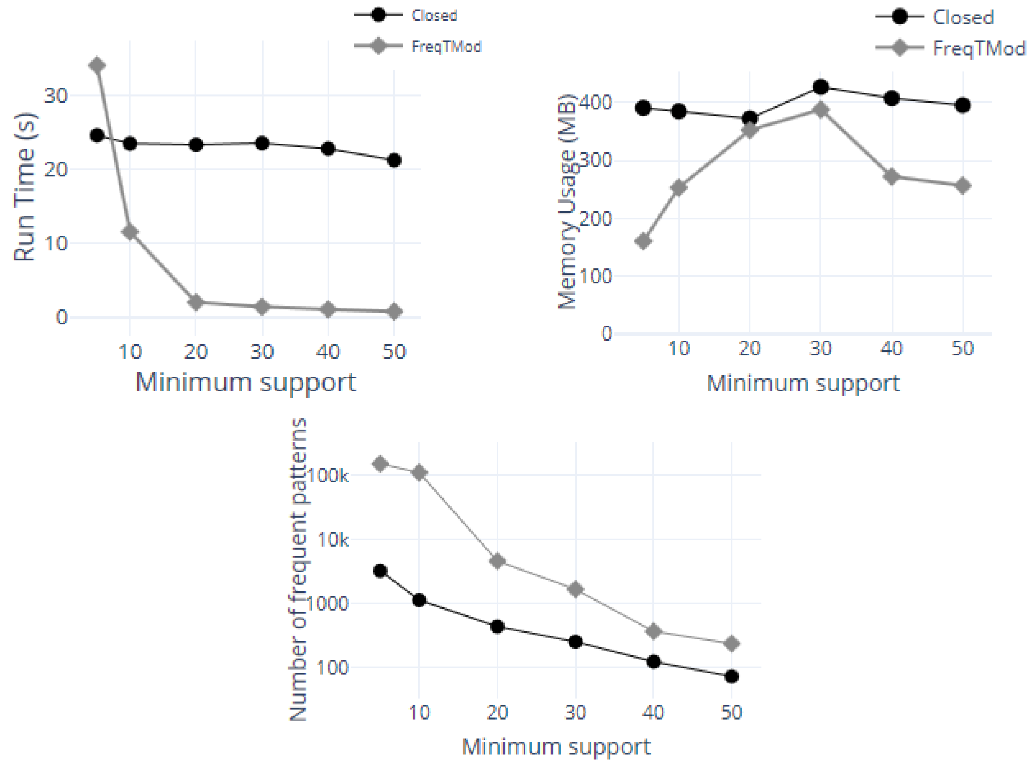


FIGURE 4.2: In this experiment, we compare *Closed* pattern mining with the initial implementation of FreqT in Java, *FreqT-Mod*, provided by the research group. We use the dataset of one hundred Java ASTs described here Section 6.6.1 with varying minimum support.

that this approach is not a promising one. Even though the output size tends to be smaller and there are fewer redundancies in the reported patterns, many issues with this solution held us from pursuing further evaluation.

The first drawback is that checking for closedness introduces a lot of overhead, both in time and memory consumption, as we need to keep the complete occurrences and compute the blanket for each pattern. We already discussed this in theory, but the experiments also confirm this. Moreover, the gain due to the possibility of pruning some of the search space proved to be insufficient to compensate this increase. We illustrate this in Figure 4.2.

In this experiment, we compare our *Closed* pattern mining algorithm with the initial implementation of FreqT in Java, *FreqTMod*, provided by the research group. We use the dataset of one hundred Java ASTs described here Section 6.6.1 with varying minimum support. The constraints used are the same in both cases: we use the mandatory children expansion constraint and the minimum pattern size of two. Note that we also disabled the leaf constraint on the left side of the pattern for both. The biggest difference comes thus

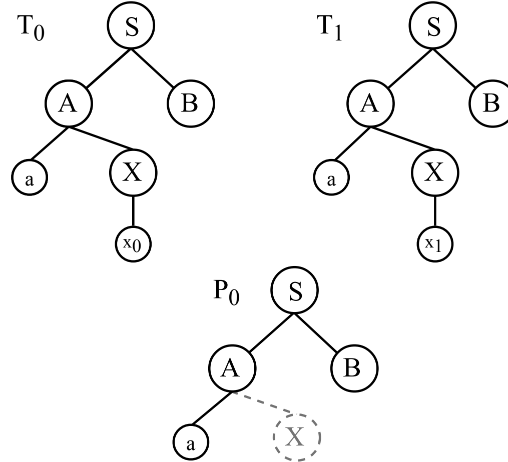


FIGURE 4.3: An example illustrating an issue which can arise when using the closed and the leaf constraint, i.e. expand to leaf nodes on the left side of the pattern. In this example, with a minimum support of two, the pattern P_0 would not be reported.

from the need of keeping all the occurrences and computing the blanket B_t, something not necessary in *FreqtMod*. We observe that, indeed, we find fewer patterns (a subset, to be precise) when using the *Closed* algorithm. Also, the memory requirement is higher due to the storage of the complete occurrences.

Finally, note that the FreqT algorithm used here does not follow the original implementation to the letter. The original one reports every pattern provided it is frequent [7]. Here, the given implementation in Java only reports in specific cases depending on the constraints. In this example, the pattern is in the output if there are no frequent candidates anymore.

Another drawback of this approach is that this closedness constraint does not combine well with some of the other existing constraints listed in Section 3.2. Indeed, depending on the configuration of the constraints, it happens that there are few or even no patterns reported in the output. One may think that this is an issue with the implementation, but the actual explanation lies in the fact that some of the existing constraints may prune a branch of the search space which leads to a closed pattern. Also, most of the currently tested configurations prune quite a lot of the search space which amplifies this problem. Those issues come from the fact that the algorithm we use, FreqT, works well with constraints that are anti-monotonic. One such constraint is the minimum support. Here, however, we use some constraints that are not anti-monotonic when considering the complete subtree and not only the rightmost path.

A concrete example of this is depicted in Figure 4.3. In this example, we

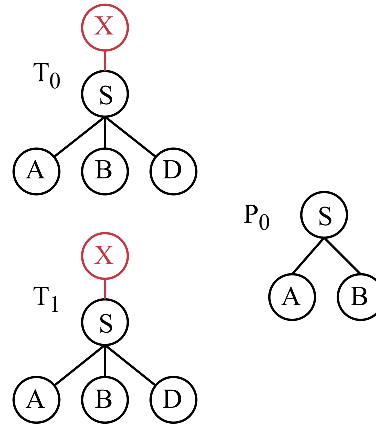


FIGURE 4.4: Another example illustrating an issue between constraints. Here, we consider closedness and root whitelist constraint. Assume the only root in the whitelist is S. A pattern such as P_0 would not reported as it is not closed due to X.

have two transactions and one pattern. Nodes in lowercase represent terminals in ASTs (e.g. operators, strings, `null`, etc.). They are also leaf nodes (or terminals) in the pattern. We assume the closedness constraint and leaf constraint are used. With a minimum support of two, the pattern P_0 would not be reported. First, we consider the pattern without the X node. In this case, the leaf constraint is respected because a is a terminal in both the pattern and the AST. The closedness constraint is not respected because we can still add X to the pattern and keep the same cover, T_0 and T_1. If we add X, we cannot respect the leaf constraint anymore: if we add either x_0 or x_1, the pattern becomes infrequent.

Note that this issue is not only related to this single constraint. Otherwise, we could just disable the one problematic constraint and call it a day. We also cannot keep only the closedness constraint as, without the other constraints, the search space becomes too big to be tractable. The runtime becomes too significant even when using a small subset of a repository.

Another example of a constraint which requires changes in the implementation to combine properly with closedness is given in Figure 4.4. In this figure, we have two transactions and a possible pattern. We assume that the root whitelist contains only S, i.e. only S can be the root of a pattern and also that the closedness constraint is applied. A pattern such as P_0 would not reported as it is not closed due to X. Indeed, with a minimum support of two, no pattern will be found as we do not allow to start one from X, the only possible start for a closed pattern. We show here a trivial example for the root whitelist, but

it is the same principle for the children blacklist constraint. Some modifications were added to handle those issues. For example, in this case, we know we should not reject the pattern because the node which violated the closedness constraint is not a possible root. The problem is that this constraint requires a lot of additional computation, as described previously. If we have to reduce its pruning capabilities, it may just not be worth to include it.

The last and probably the biggest drawback is that this solution does not address the problem of finding *interesting* patterns. Indeed, the output size is reduced, but we still find way too many patterns that are not considered as code idioms by an expert. The constraints are supposed to help with this problem, but finding the appropriate combination of constraints and their values is hard. An overly constrained miner tends to be biased towards very (and often too) specific patterns. In the following chapter, we take inspiration from techniques used in statistical NLP such as PCFGs. The goal is to reduce the number of parameters in our model that are currently set manually and try to infer them based on our input data.

Chapter 5

Parsing the ASTs

In the previous chapter, we have seen that the closedness constraint does not merge well with the existing approach used to mine code idioms, i.e. FreqT with a set of specific constraints. However, we have also seen that, even without this new constraint, the approach leaves a lot to be desired. In this and following chapter, we are interested in finding a new way to mine code idioms. This solution would be based on two steps: first, building a probabilistic model that encodes syntactic knowledge inferred from ASTs and second, using this model to mine interesting patterns. In Chapter 6, we define *interesting* as patterns that compress the database. In this chapter, however, we describe the first step: building our probabilistic model.

We chose to use a simple probabilistic context-free grammar as our model. We call it "simple" because it is not the best model for source code. Indeed, if one were to generate programs using a *probabilistic context-free grammar* (PCFG), he would observe that they hardly look like written programs. The context-free nature of the grammar does not capture important aspects of source code such as the scope of variables. Other models, such as a generalisation of PCFGs which allow conditioning of a production rule beyond the parent non-terminal [10], have been investigated in the literature specifically as models for source code, but we restrict ourselves to a simpler one in this initial attempt.

Before going any further, there is a crucial element that needs to be clear. Given a corpus of trees, it is easy to extract a CFG and estimate its parameters to obtain a PCFG [24]. However, this is not what we are interested in (or at least not in this chapter). Recall that we base ourselves on an existing research project which assumes that the grammar of the programming language is given and follows a specific format described in Section 3.2. The problem thus becomes, given C a corpus of ASTs, G the grammar of the used programming language and both C and G respecting the metamodel format, *parse* the ASTs using this grammar in order to estimate the parameters of the PCFG. Indeed, once the trees are parsed according to the grammar, we can use the relative

frequency estimation to get the probability associated with each rule. Note that we are *not* parsing the source code once again. The ASTs represent this parsing step and are given by a compiler. However, we can consider our grammar as a *metagrammar*, a grammar of the ASTs themselves. If it is correct, one should be able to parse the ASTs by considering their nodes as terminals in our metagrammar.

As we will see, if the metamodel format is respected, the grammar has some useful properties which make that the required parser is relatively simple. This parsing step is also useful to detect mismatches between the grammar and the ASTs. Indeed, writing the grammar and producing the ASTs is done in two steps and this situation leads to inconsistencies between them. By writing *two steps*, we mean that the ASTs are not produced using a CFG, but extracted from their respective compilers. The CFG is then built in another step. In practice, mismatches occur between the two. This leads to a paradox were a task such as parsing the ASTs by using the grammar that supposedly generated them can lead to errors. We also illustrate later on with some concrete examples of issues that were detected. Here, we also study approaches to work around such issues.

5.1 Background

The most commonly used probabilistic grammar is the probabilistic context-free grammar (PCFG). It is an extension of context-free grammars in which each rule is associated with a probability. Formally, a context-free grammar G is a quadruple (Σ, T, R, S), where Σ is a set of non-terminal symbols, T a set of terminal symbols, R a set of rules/productions in the format $A \longrightarrow \beta$ with $A \in \Sigma$ and $\beta \in \Omega$ with $\Omega = (\Sigma \cup T \cup \{\varepsilon\})^*$. Note that we suppose a grammar with ε-rules, i.e. rules consuming an empty string.

A PCFG augments the definition of a context-free grammar by changing R into P, the same set of rules but each one of them now associated with a probability p which expresses $P(\beta \mid A)$. To estimate those parameters, we define $c(A)$, the number of times the non-terminal A appears and $c(A \longrightarrow \beta)$, the number of times the rule $A \longrightarrow \beta$ was used [24]. Formally, we write:

$$c(A) = \sum_{\beta} c(A \longrightarrow \beta) \tag{5.1}$$

$$\hat{P}(A \longrightarrow \beta \mid A) = \frac{c(A \longrightarrow \beta)}{c(A)} \tag{5.2}$$

An $LL(1)$ grammar is a specific type of context-free grammar with properties useful for parsing input strings. LL(1) means that we scan the input from **L**eft to right, using a **L**eftmost derivation and looking only **one** input ahead to decide which production to apply.

A *leftmost derivation* consists in always replacing the first non-terminal on the left. There is a notation specifically for this. Let α, β, γ be an arbitrary sequence of any symbol, i.e. $\alpha, \beta, \gamma \in \mathbf{\Omega}$. If we can derive β from α in zero or more steps, we write $\alpha \Rightarrow^* \beta$. If we can derive γ from β in a single step, we write $\beta \Rightarrow \gamma$. Consequently, we write $\alpha \Rightarrow^* \gamma$

This kind of grammar is interesting because it can build, by definition, a *parsing table* that has no multiple entries. A parsing table takes as input a non-terminal α and a symbol i from the input and returns the unique set of non-terminals which should rewrite α. More formally, let G a context-free grammar, $A \in \mathbf{\Sigma}$ and $\alpha, \beta \in \mathbf{\Omega}$ with $A \longrightarrow \alpha \mid \beta$ and $\alpha \neq \beta$, we write that G is LL(1) if and only if:

$$\mathit{FIRST}(\alpha) \cap \mathit{FIRST}(\beta) = \emptyset \tag{5.3}$$

$$\alpha \Rightarrow^* \varepsilon \implies \mathit{FIRST}(\beta) \cap \mathit{FOLLOW}(A) = \emptyset \tag{5.4}$$

FIRST(γ) is the set of terminal symbols which can start a string produced by γ. If γ is a terminal, FIRST(γ) is simply the set that contains only this terminal. If γ is a non-terminal, it is the union of the FIRST sets of symbols (possibly also non-terminals) on its right-hand side. Formally, the set is defined as $\text{FIRST}(A) = \{t \mid A \longrightarrow t\omega\}$ where ω is an arbitrary derivation, A is the non-terminal under consideration and t is a terminal.

The FOLLOW set is the set of terminals that may come after a given non-terminal. Formally, the set is defined as $\text{FOLLOW}(A) = \{t \mid S \Rightarrow^* \alpha A t \omega\}$ where S is the start symbol, α and ω are arbitrary derivations, A is the non-terminal and t is a terminal.

Note that we do not discuss here how to compute those sets. An intuition of what those sets represent is sufficient for understanding the concepts explained in this work.

An efficient *top-down* parser can be built for a LL(1) grammar. A top-down parser starts with the start symbol and tries to produce the given input. It uses a stack, initialised with the start symbol on top along with a sentinel value, `#`, under it, and rewrites the productions based on the table, the current symbol on top of the stack and the current input value.

A specific type of LL(1) grammar exists and is called *simple LL(1)* or *SLL(1)*.

It is a grammar which satisfies the following requirement: all right-hand sides of a non-terminal start with a different terminal symbol. With this restriction, the parsing is guaranteed to be deterministic. The difference with LL(1) is that, in general, LL(1) also allows non-terminals at the start of the right-hand side and allows ε-transitions.

One more definition is needed and comes from a technique used in NLP. It is called *parent annotation* or also *vertical markovization* and consists in annotating the non-terminals with their parent. If we have rules such as $A \longrightarrow$ a X, $B \longrightarrow$ b X and $X \longrightarrow$ a | b and we know that X always produces the terminal "a" when starting from A, then we can rewrite it as $A \longrightarrow$ a X_A and similarly for X_B. X_A is a new non-terminal symbol in our grammar and we say that X is annotated with its parent A. This is a trivial example but it showcases that parent annotation allows to keep some context, the parent, to make a decision before rewriting the non-terminal.

The definitions presented in this section should be sufficient to follow the rest of this chapter. A more in-depth explanation of other concepts and techniques related to parsing can be found in [19].

5.2 Possible parser

We are interested in parsing our ASTs using the defined grammar. In this section, we check what kind of parser is required to do this. For Java, all the mentioned input files (grammar and ASTs) respect a specific format, the metamodel, which has some interesting properties.

The key observation here is that the Java grammar defined in this metamodel format respects the first condition of being LL(1) which means that there is no overlap between the terminals at the start of any right-hand side. Note, however, that we specifically did not write that it *is* LL(1). Indeed, we show that the given Java grammar is not LL(1) due to some problematic grammar constructions. Those problems are due to the fact that we allow ε-rules, something that is not possible in a traditional SLL(1) grammar. However, a top-down stack-based parser can still be made with some small adjustments.

For the given COBOL grammar, this first LL(1) condition does not hold and either the grammar must change to conform to this requirement or a more complex parser must be designed.

Before going further, we stress that we discuss the specific Java and COBOL grammars respecting the metamodel format as defined by the research group and not Java or COBOL grammars used by the programming languages. The

difference is that we parse the ASTs and not the source code tokens so the grammars are different.

5.2.1 Modelisation

5.2.1.1 Definition

A grammar respecting the metamodel format allows a limited set of possible constructions for its non-terminals. A non-terminal A allows the following formats:

The first one is defined as $A \longrightarrow \mathrm{t}\alpha_1, \alpha_2, \ldots, \alpha_N$ where α_i is a non-terminal and t is a terminal. In our grammar, rules in this format define the "structure" of a node, i.e. its possible children.

The second one is defined as $A \longrightarrow \mathrm{t}\alpha \mid \varepsilon$. The main difference here is that we allow the ε-transition. This format is useful to express the fact that some children are optional and might not be present in the parsed AST. Also, α can be a terminal. If that is the case, we have reached a leaf node in our AST, i.e. a leaf in the XML. These are terminals *in the source code* and they correspond to values such as, for example, `true/false`, all the operators, method and variable names etc. This special case of the format parses the `simplevalue` attribute as defined in Section 3.2.

The third one is defined as $A \longrightarrow \gamma \mid \ldots \mid \beta$. The non-terminals γ, β and all the other non-terminals in-between are in the first format. This format allows us to pick between productions. This is the case for nodes that are marked as `abstract`: we can pick the appropriate subtype to use based on the input.

The last format is defined as $A \longrightarrow \gamma A \mid \ldots \mid \beta A \mid \varepsilon$ where γ, β are also in the first format. Notice that this is recursive rule as the non-terminal A appears on the right-hand side. This format is typically used when one needs to generate an unbounded list of symbols with ε marking the end.

To illustrate this, we consider the example in Figure 5.1. We present a grammar which can be used to generate an `InfixExpression`, given in Appendix A and rewritten following our formalism. For clarity, the operator is simplified and only produces a $+$. The same goes for an `Expression` which can only produce an `InfixExpression` or a `NumberLiteral`. The rule 5.5 is the starting production and follows the third format. The rule 5.6 defines the children needed to produce an `InfixExpression` and thus follows the first format. The three rules that follow describe the children of an `InfixExpression` and follow the second format. The non-terminals O and T illustrate the specific case of the second format where α is a terminal in the source code. Note that this is

$$\Sigma = \{S,\ E,\ I,\ L,\ O,\ R,\ N,\ T\}$$
$$T = \{\texttt{infixexpression},\ \texttt{leftoperand},\ \texttt{operator},\ \texttt{rightoperand},$$
$$\texttt{numberliteral},\ \texttt{token},\ \texttt{+},\ \texttt{string}\}$$
$$P = \{S \rightarrow I \tag{5.5}$$
$$E \rightarrow N \ \mid \ I \tag{5.6}$$
$$I \rightarrow \texttt{infixexpression}\ L\ O\ R \tag{5.7}$$
$$L \rightarrow \texttt{leftoperand}\ E \ \mid \ \varepsilon$$
$$O \rightarrow \texttt{operator +}$$
$$R \rightarrow \texttt{rightoperand}\ E \ \mid \ \varepsilon$$
$$N \rightarrow \texttt{numberliteral}\ T$$
$$T \rightarrow \texttt{token string}$$
$$\}$$

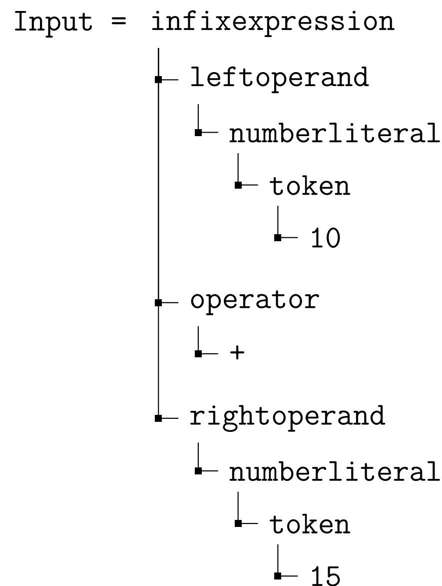


FIGURE 5.1: Simplified `InfixExpression` defined in the original grammar in Appendix A and rewritten to follow the formalism defined in Section 5.2.1
The input below can be parsed using this grammar.

$\Sigma = \{S,\ C,\ D,\ BODY,$
$STMT,\ BLOCK$
$EXC,\ E\}$
$T = \{$`catchclause`, `body`,
`dostatement`$\}$
$P = \{$
$S \rightarrow C \mid D$
$C \rightarrow$ `catchclause` $EXC\ BODY$
$\boldsymbol{BODY} \rightarrow$ `body` $STMT$
$\mid$ `body` $BLOCK \mid \varepsilon$
$D \rightarrow$ `dostatement` $BODY\ E$
$\}$

(A) Not annotated

$\Sigma = \{S,\ C,\ D,$
$BODY_C,\ BODY_D,$
$STMT_BODY,\ BLOCK_BODY,$
$EXC_C,\ E_D\}$
$T = \{$`catchclause`, `body`,
`dostatement`$\}$
$P = \{$
$S \rightarrow C \mid D$
$C \rightarrow$ `catchclause` $EXC_C\ BODY_C$
$\boldsymbol{BODY_C} \rightarrow$ `body` $BLOCK_BODY \mid \varepsilon$
$\boldsymbol{BODY_D} \rightarrow$ `body` $STMT_BODY$
$D \rightarrow$ `dostatement` $BODY_D\ E_D$
$\}$

(B) Parent annotated

FIGURE 5.2: The two grammars both define rules for the Java `catch` and `do` statements. The left version is the base version and the right one is parent annotated. The productions in bold are making the crucial difference.

not the final grammar used for parsing as there is still an issue that needs to be addressed, but it does not change the format of the rules.

This remaining issue is illustrated in Figure 5.2. We take as example the rules which generate the nodes `CatchClause` and `DoStatement` in our ASTs. The right-hand sides of the non-terminals *STMT*, *BLOCK*, *E*, *EXC* are omitted but they generate, respectively, a statement, a list of statements, an expression and the variable declaration for the exception. If we use the grammar on the left (Figure 5.2a) and start with the token `catchclause` as our input, the production *C* will be used and the token consumed. Then we parse, via the *EXC* production, the type of exception we want to catch. At some point, when we reach an input token `body` and we have the production *BODY* at the end of our current derivation, we have an issue: how to pick the correct right-hand side. A `CatchClause` expects a `Block` while a `DoStatement` expects a `Statement` but we cannot look ahead more than one token which is `body`.

We solve this in the right grammar (Figure 5.2b) by *parent-annotating* the grammar. We can immediately see that the set of non-terminals Σ is larger than before. We also observe that the non-terminal *BODY* was split into *BODY_C* and *BODY_D* which define, respectively, what is inside the body

of a `CatchClause` and `DoStatement`. This time we would have *BODY_C* at the end of our derivation and predict `body` *BLOCK_BODY*. The non-terminal *BODY_C* now also respects the second form which was not the case for *BODY* in the left grammar.

5.2.1.2 Equivalences

5.2.1.2.1 First condition

Using this formally defined model, we can reason whether our given Java grammar for ASTs, which respects the metamodel format, is SLL(1), LL(1) or neither. First, we check whether our grammar respects the first condition defined in the Equation (5.3). This condition says that, for all possible right-hand sides of a non-terminal, their *FIRST* sets cannot intersect. To prove this, we would have to take a look at the complete grammar, but we can avoid this and define the conditions based on our model instead.

First, we can remove from consideration the rules in the second format because, for every rule in this format, there are only two possible right-hand sides: $\mathtt{t}\alpha$ or ε. We always have that $\text{FIRST}(\mathtt{t}\alpha) \cap \text{FIRST}(\varepsilon) = \emptyset$. We also made sure that those rules have only two possible right-hand sides by using the parent annotation as described in Figure 5.2. The condition is thus always respected for rules in the second format. The rules in the third and fourth format, by their definition, define a choice between rules in the first format. For those rules in this first format, the FIRST set is always $\{\text{t}\}$. In other words, to respect the first condition of LL(1), we cannot find two non-terminals $A \longrightarrow \text{t}_A\alpha_1, \alpha_2, \ldots, \alpha_N$ and $B \longrightarrow \text{t}_B\beta_1, \beta_2, \ldots, \beta_M$ such that $t_A = t_B$ as it would mean that $\text{FIRST}(A) \cap \text{FIRST}(B) \neq \emptyset$.

It so happens that this is true for the given Java grammar, but not for the COBOL one. Another problem identified with the COBOL grammar is that the second format is sometimes not respected and results in rules such as $A \longrightarrow \alpha \mid \varepsilon$. Those rules are problematic as we do not have immediately the FIRST set of A anymore as it depends now on α. We skip examples for COBOL as the grammar is proprietary and not many readers might be familiar with the language.

How does this grammar compare to existing definitions ? A SLL(1) grammar must have all right-hand sides of a non-terminal start with a different terminal, i.e. respect the condition in Equation (5.3), and not allow ε-transitions. Those transitions are possible in our formalism so it is trivially not SLL(1). The more interesting question is whether it is LL(1). If that's the case, we know that a

$$
\begin{aligned}
&\Sigma = \{JD,\ TL,\ T,\ N,\ F\}\\
&T = \{\texttt{tags},\ \texttt{tagelement},\ \texttt{tagname},\ \texttt{string},\ \texttt{fragments}\}\\
&P = \{S \rightarrow JD\\
&\quad JD \rightarrow \texttt{tags}\ TL && (5.8)\\
&\quad TL \rightarrow T\ TL\ \mid\ \varepsilon && (5.9)\\
&\quad T \rightarrow \texttt{tagelement}\ N\ F && (5.10)\\
&\quad N \rightarrow \texttt{tagname}\ \texttt{string}\\
&\quad F \rightarrow \texttt{fragments}\ TL\\
&\}
\end{aligned}
$$

FIGURE 5.3: The (simplified) rules presented in this grammar generate a `Javadoc` tag in our ASTs. This grammar does not respect the second condition of being LL(1).

simple LL(1) parse can be used. The first condition is satisfied, but what about the second?

5.2.1.2.2 Second condition

Unfortunately, some possible constructions defined in the Java grammar do not respect this second condition. We illustrate the problem in Figure 5.3. The simplified grammar in this example can generate a `Javadoc` tag in our ASTs. We simplify, for the sake of readability, by not parent-annotating the rules and considering only a subset of possible constructions.

A `Javadoc` tag is a list of `TagElement` tags. This is expressed by the rules Rule 5.8 and Rule 5.9 where this last rule follows the fourth format in our formalism and thus generates a list. A `TagElement` has a name (e.g. `@return`) and a list of *fragments*. Fragments usually are the strings of documentation, but, more interestingly for us, also can be other `TagElement` tags. This is the only possible construction we consider in rule F. An example of nested tag elements would be: `@return {@code <Object>}`. In an AST, the XML tag `TagElement` named `@code` would be nested inside the fragments of the tag `TagElement` named `@return`.

The problem with this grammar is that it does not respect the second condition of LL(1). To show this, we compute the *FIRST* and *FOLLOW* set of the production TL in Rule 5.9. We find that $FIRST(TL) = \{\texttt{tagelement}, \varepsilon\}$ and $FOLLOW(TL) = \{\#, \texttt{tagelement}\}$ where `#` marks the end of the input.

This means that $FIRST(TL) \cap FOLLOW(TL) = \{\texttt{tagelement}\}$ and the second condition in Equation (5.4) is not respected.

More intuitively, let us consider *parsing the AST* generated by the nested tags `@return {@code <Object>}` as a *sequence of string tags.* We can do this without problems up until we encounter the input token `tagelement` which starts the nested tag. At this point, however, we do not know if the `TagElement` named `@code` should be considered as a nested tag or as a second tag in our original javadoc. Indeed, both are acceptable by the grammar in Figure 5.3. The original javadoc grammar (the one used to parse the source code and not ASTs) requires braces around nested tags to avoid this issue. A naive parser could parse this as a nested tag by default, but this does not need to be the case in the original source code. Instead of being nested, it could be the second tag of our javadoc. This error affects the PCFG estimation. It would mean that the *parent-annotated* production for generating the javadoc tags, TL_JD, produced only one `TagElement` when it actually may have been two.

As a final remark, note that our ASTs do not contain `Javadoc` tags as we do not mine for patterns in comments. One could imagine that we can ignore this issue, but there are other rules with the same problem (e.g. `InfixExpression` in Appendix A). We explain the `Javadoc` rule because it is small enough to be used as a concrete example.

Also, note that some tools such as [20] can do the verification of the first and second condition of being LL(1) automatically, provided the grammar is in the required format and is small enough.

5.2.2 Practical application

The research group has at its disposal the Java grammar in the metamodel format and the ASTs generated from the *QUAATLAS* corpus [15], a refined subset of the *Qualitas Corpus* which is “a curated collection of software systems intended to be used for empirical studies of code artefacts” [30, 35]. Currently, we focus on only one repository, the *jhotdraw* project, which is a 2D drawing library.

The goal was to parse all its files using the predefined grammar. The implementation of the parser is similar to a top-down stack-based LL(1) parser, but with some changes considering our particular setup. Indeed, we keep a stack with the non-terminals yet to be processed. We read the XML nodes, one-by-one, in a depth-first fashion as our input. We can either match a terminal, and consume the input token, or a non-terminal, and replace it with the appropriate right-hand side. If no match is found, we try to produce an epsilon, and, if it

is not possible, then the derivation is not possible. To disambiguate between nested lists, we use the information about the depth of the node to decide if we continue the list or produce an epsilon. When the parser could not produce a valid derivation, we would analyse what is the cause of the issue. By doing this repeatedly until all files were parsed correctly, we discovered the following types of issues:

Mandatory or optional children
: The most common issue was that children marked as mandatory in the grammar were often optional in the ASTs. This leads to an obvious error during the parsing step since we expect a specific non-terminal that isn't there at this position in the AST.

Missing definition for abstract node
: One of the possible abstract nodes, `IExtendedModifier`, can be either a `Modifier` or an `Annotation` following its definition in the JDT Core compiler [16]. In the given grammar, the `Annotation` option was missing which led to an error when such a rule was needed in the ASTs.

Inconsistencies
: In some cases, the grammar defines a different XML structure than the one observed in the ASTs. For example, the node `NullLiteral` must have a `token` child. This issue is also present for the `Operator` abstract nodes which do not follow the grammar closely and require workarounds to parse correctly.

Nodes pruning
: One of the post-processing of the ASTs consists in pruning empty XML nodes. We observed that it can have a significant impact on the grammar. For example, the node `ThisExpression` has an optional child `qualifier`. This child will appear in the AST if we write, for example, `MyClass.this`. If we write only `this`, the AST node `ThisExpression` which represents this token would be pruned as it is empty. This impacts children which produce `Expression`, its supertype.

  For example, in Appendix A, the `InfixExpression` node has optional left/right operands which produce an `Expression`. However, in the JDT Core definition, those two are mandatory. We could not do the same because `ThisExpression` nodes would be pruned when they did not have a child thus removing a mandatory child from the operands.

Most of those issues were detected thanks to this parsing procedure and analysis. Required fixes were also proposed and merged by the repository owners. This situation showcases that producing a grammar to use it in the mining process is not a trivial task, even if we are only focusing on programming languages.

5.3 Limitations

We have seen that the implemented parser can parse Java ASTs that respect the metamodel grammar. This parsing allows us to know which productions were used, how many times, and to estimate the parameters for a PCFG based on this information. As a bonus, we were able to identify some mismatches between the grammar and ASTs and fix them.

There is one final but important observation that needs to be discussed. In this chapter, we modelled the problem as parsing a *sequence of tokens* as input. The tokens are string representations of XML tags used in the ASTs (e.g. the token `tagelement` for the XML tag `<TagElement>` defined in Appendix B).

We have seen that we can use an efficient top-down stack-based parser to do it *if* the grammar is LL(1). This appeared not to be the case as there are some constructions that violate the second condition of being LL(1). An example of this was given in Figure 5.3. To solve this, the astute reader will notice that we used some information that is not present in a sequence of tokens: the depth of the token inside the AST tree. This suggests another way to approach this problem. We can directly consider the input as a *sequence of tree nodes*. In this case, the grammar respecting the metamodel format can be used in combination with a breadth-first traversal in order to compute the counts of the productions used. If the grammar expects a specific child but this child is not present under a given AST node, it means it follows the second format and the ε right-hand side was used. It also means that, if the metamodel grammar is correct, this child is marked as optional using the attribute `optional="true"` in the XML. However, this still does not solve the issue with the COBOL grammar as it does not follow the metamodel nor the formalism defined. The reason not to immediately model the problem as a sequence of tree nodes is that parsing a sequence of tokens is a well-studied problem. Using this model first allows us to reuse the same notations and definitions to have a formal description of the problem and be able to reason on it as we did it in this chapter.

Recall that the original goal was to produce a probabilistic model for our source code. This model can then be used to define a formal criterion of interestingness for patterns. The task of building this model with the assumption that a grammar is given has proven to be quite time-consuming. In the next chapter, we drop this assumption and, instead, we infer the model and its grammar directly from the input ASTs.

Chapter 6

Mining compressing patterns

In this chapter, we describe a new approach for mining code idioms. The current approach proposed by the research group relies heavily on choosing and setting an appropriate combination of constraints to find code idioms instead of all frequent patterns. While doable, it is also hard to define what is the optimal configuration of the constraints and the appropriate values of the parameters for those constraints.

Another approach is to define some sort of interestingness score for patterns. With a formal definition for this score, we would be able to apply optimisation techniques to find the optimal set of patterns or, at least, an approximation. We would also rely less on manually-defined constraints. The ones with a high score would be considered to be code idioms. We start with the assumption that code idioms are patterns that can compress the database the most. More precisely, the goal is to find the *probabilistic tree substitution grammar* (PTSG) which provides the best compression based on the *Minimum Description Length* (MDL) principle [32, 18].

In the previous chapter, we used our predefined grammar to parse the ASTs to produce a PCFG. This approach, while possible, was time-consuming for Java due to some inconsistencies between the ASTs and the given grammar. For COBOL, this approach was not possible altogether due to the grammar not respecting the predefined format. In this chapter, we drop this requirement of having a predefined grammar and infer it from the input ASTs instead. The advantage is that we can skip the parsing step and also apply it on COBOL ASTs. The disadvantage is that it remains an approximation and is dependent on the number of input ASTs at our disposal. For older languages, obtaining them in large quantities may be harder than for more modern languages.

Note that inducing a grammar from a corpus of strings is one of the central challenges of computational linguistics [14]. Here, however, we have the luxury of having a treebank corpus, our ASTs, making it a supervised learning problem. Also, we work with programming languages, which are more restrictive than

natural language but are also repetitive and predictable [21]. Finally, we are not using our grammar to parse new ASTs, but as a probabilistic model for source code. This model allows us to compute the number of bits to encode our data and model.

We start by introducing some definitions related to information theory and tree substitution grammars. We continue with an explanation of our model to compute the value of Equation (6.1). We follow with the details of the implementation and an evaluation of the results obtained.

6.1 Background

MDL was introduced by Rissanen in [32] as a noise-robust model selection technique [36]. The main idea is to evaluate the models based on their compression capabilities. This means that, given a set of models $\mathcal{M}$ and a database D, we find the model M which compresses D the most. Formally, the best model $M \in \mathcal{M}$ is the one which minimises:

$$L(M) + L(D \mid M) \tag{6.1}$$

where

- $L(M)$ is the number of bits required to encode the model
- $L(D \mid M)$ is the number of bits required to encore the data using this model

A *tree substitution grammar* (TSG) is a generalisation of CFGs. The key difference is that CFGs allow only a *sequence* of terminals/non-terminals on the right-hand side of their productions. On the other hand, TSGs, as their name suggests, allow inserting tree structures, which we call *elementary trees*. The internal nodes of those elementary trees are non-terminals from our grammar, and the leaf nodes can be either a terminal or a non-terminal. Elementary trees of height one correspond to productions in a context-free grammar [14].

The advantage of having this kind of productions is that they allow keeping some local context, something not possible in the basic context-free grammar. This is especially relevant in our context of mining source code ASTs for code idioms. Ideally, our learned TSG would contain only code idioms as elementary trees.

Formally, a tree substitution grammar G is a quadruple (Σ, T, R, S), where Σ is a set of non-terminal symbols, T a set of terminal symbols, R a set of rules/productions in the format $A \longrightarrow \tau$ with $A \in \Sigma$ and $\tau = (V, E, A)$.

The set of vertices V is divided into internal nodes $V_I \subseteq \Sigma$ and leaf nodes $V_L \subseteq (\Sigma \cup T)$ such that $V_I \cup V_L = V$. A is also the root of the elementary tree and $A \in V_I$.

A probabilistic tree substitution grammar augments the definition of a tree substitution grammar by changing R into P, the same set of rules but each one of them now associated with a probability p which expresses $P(\tau \mid A)$.

6.2 Related

The use of PTSGs has been heavily studied in the NLP community. It has its roots in the Data-Oriented Parsing (DOP) approach. The first innovation introduced by this approach was to use fragments of corpus directly as a grammar instead of training a predefined one. The previous models started with this predefined grammar and used a corpus only for estimating the rule probabilities [11]. The use of a predefined grammar is similar to our approach in Chapter 5.

The other innovation was to use *all* corpus fragments, of any size, instead of a small subset [11]. The original approach had some issues which researchers tried to address later on [14]. This is different from our approach and the approach proposed in “Mining Idioms from Source Code” [4, 14]. In their work, they use a non-parametric prior which favours compact grammars [14]. In ours, favouring compact grammars is built-in the MDL principle. Finally, note that the DOP model was developed in the context of parsing new sentences using this learned grammar, something we are not interested in. We are only interested in the induced TSG.

In the data mining community, the MDL principle also has been used to find only a subset of all the frequent patterns. One paper from which we draw inspiration to test the MDL principle is “Krimp: Mining Itemsets That Compress” [36]. Their *KRIMP* algorithm defines an approach to find the set of itemsets that compresses the database the most. The first difference is that we are interested in tree structures, not itemsets. The other difference is that, in their approach, they start with a large set of frequent patterns as input from which they pick their candidates to add to their output set. This is harder to do in our context as selecting from the set of *all* frequent subtrees would not be manageable. In our approach, we build the candidates greedily based on their compression capability. However, we keep a simplifying assumption that they also used regarding overlaps: we do not allow overlap between patterns inserted in our TSG. Their reason to do it is that there is no fast heuristic they can apply if they allow overlap. Similarly, in our case, we reduce the search space

thanks to this simplification because we do not have to consider nodes already covered. The implementation is also more straightforward.

Although all the concepts presented here have already been studied extensively, our contribution is in combining those techniques in the specific context of mining code idioms, by inducing a TSG using the MDL principle. In the next section, we describe our approach to inducing a TSG.

6.3 Inducing a TSG

In Chapter 3, we presented the approach used in “Mining Idioms from Source Code” which uses techniques presented in [14], i.e. inducing a TSG using non-parametric Bayesian methods. Here, we look at inducing a TSG using the MDL principle. The main idea is to build a TSG by iteratively adding patterns that compress the database the most.

The simplest TSG one can build is a TSG with only elementary trees of height 1. This is our initial TSG, and we denote it with TSG_0. To build it, we traverse the nodes in our ASTs and collect, for each label, the sequence of children it has into a set. Those are the leaf nodes of elementary trees of height 1. Indeed, each element of this set defines a possible right-hand side. This TSG_0 is equivalent to a CFG. An example of this is given in Figure 6.1. At the top, we find some productions of TSG_0 that were learned on two transactions, T_0 and T_1. The transactions and their derivations are initially identical and are depicted under TSG_0. The elementary trees of height 1 are indicated with *arrowed* lines. An inserted pattern is indicated by *coloured lines without arrows.*

The first inserted pattern is drawn in blue. We write it, using the bracket notation, as: $[_\text{S}\ [_\text{A}\ [\ \text{a}\ ]\ ]\ [\ \text{B}\ ]\ ]$. This insertion has an impact on the existing derivations which we need to update. This update is done in two steps. First, every occurrence of the pattern is now explained by the newly added TSG production. We can see that, in T_0, the production was replaced by a new production, in blue, based on the pattern.

The second step comes from the fact that the pattern does not explain everything in T_0, i.e. we cannot find a valid derivation for T_0 anymore if we add the pattern to the TSG as-is. To solve this, we add new non-terminals to the pattern *in-between* its existing nodes. For the blue pattern, the new non-terminal is S'_0. Those new non-terminals can be uniquely identified by the combination of the pattern which created it, the position of their parent in a preorder traversal, and their position among the siblings.

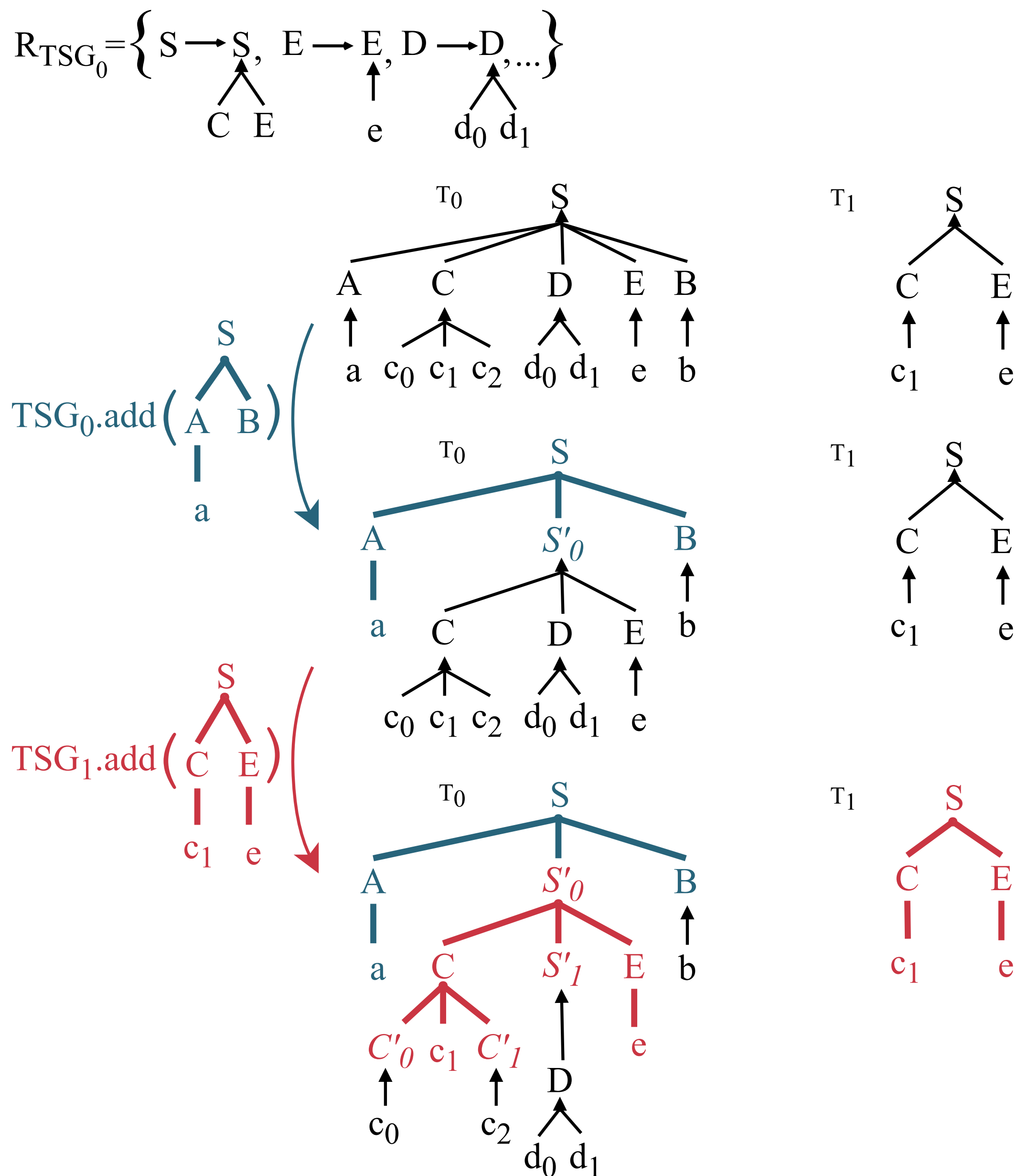


FIGURE 6.1: Example of the impact on the derivation by successively inserting two patterns into an initial TSG with elementary trees only of height 1 (TSG_0). R_{TSG_0} shows some productions available in TSG_0. The derivations which use those rules are marked with solid black arrows. Larger elementary trees that are inserted into the TSG afterwards are denoted by coloured lines. Additional non-terminals are added to a pattern to keep a valid derivation.

Note that we also need to be able to remove an added pattern from the TSG. We will require this ability for our search procedure later on. However, we make a simplifying assumption and allow only the removal of the last inserted pattern. This is easier to implement as it amounts to “reloading” the elementary trees of height one for the nodes in the added pattern.

The benefit of doing this transformation with new non-terminal nodes in-between is that we go from a valid derivation to another valid derivation and avoid any (re-)parsing step. Thus, we are always able to estimate the probabilities for our PTSG. The drawback is that, in the worst case, we have to add, for each node of the pattern with k children, $k+1$ additional children to explain nodes in-between that were left over (such as C, D, E in our example). However, we do not add useless nodes such as, for example, additional children before A or after B as those rules would produce ε with probability 1. They would not participate in the coding length of the data but would increase the size of the model for no reason. Another issue is that, by adding new non-terminals, the coding length of the model may penalise adding new patterns too much and no improvement will be found.

The patterns inserted in this example are not necessarily the best ones for compression, although the second one is a good candidate. We focus in this section only on the impact of adding new elementary trees into our TSG, not on finding which patterns we should add. We describe this process later on. If we run our algorithm on this example with a minimum support of one, the most compressing grammar we report is the one with three productions and two non-terminals: $\mathrm{E} \longrightarrow \mathrm{e}$ and $\mathrm{S} \longrightarrow \mathrm{T}_0 \setminus \{\mathrm{e}\} \mid \mathrm{T}_1 \setminus \{\mathrm{e}\}$, i.e. the same trees as T_0 and T_1, but without the terminal e. We go from a total coding length of 84.37 bits to 59.69. In the next section, we describe how do we compute those two values.

6.4 Model description

To minimise the number of bits required to encode the model and the data using this model, we still need to define how to compute this value, i.e. the function $L(\cdot)$.

Based on Shannon's source coding theorem [34], the number of bits needed to encode an event with probability p is given by $-\log_2(p)$. In the rest of this work, the logarithm is always in base 2.

First, let us consider the coding length of the data. We want to encode the data, our ASTs, using a model, a tree substitution grammar. If we know the

derivation which produced the ASTs using this TSG, we can estimate a PTSG with the maximum likelihood estimator as described in Section 5.1. The length function for the data D given a model M, a PTSG (Σ, T, P, S) estimated on the data, can be computed with:

$$L(D \mid M) = \sum_{(A \longrightarrow \tau) \in P} -\log(\hat{P}(A \longrightarrow \tau))$$

Computing the coding length of the model is a bit more involved. We know that we have to encode the productions in P, but we have a choice to make on how to encode the elementary trees, i.e. the right-hand sides of those productions.

The value we want to compute is the number of bits needed to encode the productions in P:

$$L(M) = \sum_{(A \longrightarrow \tau) \in P} L(\tau)$$

To encode the elementary tree τ, we start by encoding the root of the tree, A. We use a uniform distribution over the non-terminals to avoid a bias towards nodes which occur very often. We also observe that those are usually not interesting as code idioms.

Next, we encode the children of each node as a list of symbols (terminals/non-terminals). We assume that each child has an independent probability of appearing. We also encode the length of the list by encoding if we stop or continue the list after each symbol. We can see this as a geometric distribution where p_e is the probability of ending the list of children of a non-terminal. We estimate each of those probabilities based on the current state of the derivation.

The last thing we encode is whether a node is internal or a leaf. We use a simple code of one bit to do it. This is equivalent to the number of nodes in the tree.

More formally, let $\tau = (V, E, A)$ be an elementary tree. The list of children of a non-terminal node $\mathrm{v} \in V$ is given by $children(\mathrm{v}) = c_0, c_1, \ldots, c_N$ and their probability of appearing by $p(c_i)$. We can write the following:

$$L(\tau) = -\log(\frac{1}{|\Sigma|}) + \sum_{\mathrm{v} \in V} L(\mathrm{v}) + |V| \tag{6.2}$$

where

$$L(\text{v}) = -\sum_{i=0}^{n-1}(\log(p(c_i)) + \log(1-p_e)) - \log(p(c_n)) - \log(p_e) \quad c_i \in \mathit{children}(\text{v}) \tag{6.3}$$

With those definitions, we are now able to compute the total coding length as defined in Equation (6.1).

6.5 Implementation

Now that we have seen how to compute the coding length, we describe how to actually build this compressing model. Recall that we know how to build an initial TSG_0, compute its coding length and build TSG_{i+1} from TSG_i, but we still need to define how to choose the pattern which we will add. This is where frequent pattern mining comes in.

Indeed, we reuse the FreqT implementation in Java provided by the research group to build the *candidate patterns* which we can add to our TSG. Be sure to notice the difference between our candidate patterns, trees that we add to our TSG, and candidates in FreqT, single node extensions to the current pattern. The algorithm follows the same candidate generation phase as described in Section 2.1.2, but we add a greedy evaluation of the compressing capability of a candidate pattern. After producing candidate patterns of size $k+1$ from candidates of size k, we add one into the current TSG_i, compute the new coding length, and remove it before doing the same for the remaining candidate patterns. The difference between the two coding lengths is its estimated compression capability. Note that the pattern should have a height of at least two to avoid inserting elementary trees of size one which are the building blocks of our initial TSG.

Ideally, the coding length with the pattern included is smaller, but this is not always the case. Indeed, if you recall our model, the cost of adding a new pattern and additional production rules may be too high due to the additional non-terminals required. In this situation, we continue expanding the pattern to hopefully find a better one which reduces the total coding length. However, there is one exception. If we are currently expanding a pattern which reduces the coding length, and all of the extensions of this pattern only increase the coding length, then we add this pattern into the TSG_i to produce TSG_{i+1}. We then start a new search with the new TSG_{i+1} and all the single node frequent patterns as candidates once again. The single node which starts a new pattern

is chosen randomly. This procedure allows us to consider different orders of insertion of compressing patterns, something especially important since we do not allow overlap.

We also added some additional heuristics to our search process. First, we use a tabu list which restricts the set of single node patterns when we restart our search with a new TSG_{i+1}. After adding the best candidate pattern to our TSG_i to create TSG_{i+1}, we add its root to the tabu list. Considering that we do not have a specific value-ordering heuristic for our set of single frequent nodes and we pick them randomly, we want to prevent expanding patterns with the same roots too many times. The second heuristic stops expanding the root of a pattern and picks a new root when no improvement was observed during a given amount of iterations. This heuristic tries to avoid getting stuck by forcing an expansion a new pattern from a new root. The tabu tenure and the limit of not-improving iterations can be chosen similarly to the minimum support.

6.6 Validation

6.6.1 Datasets

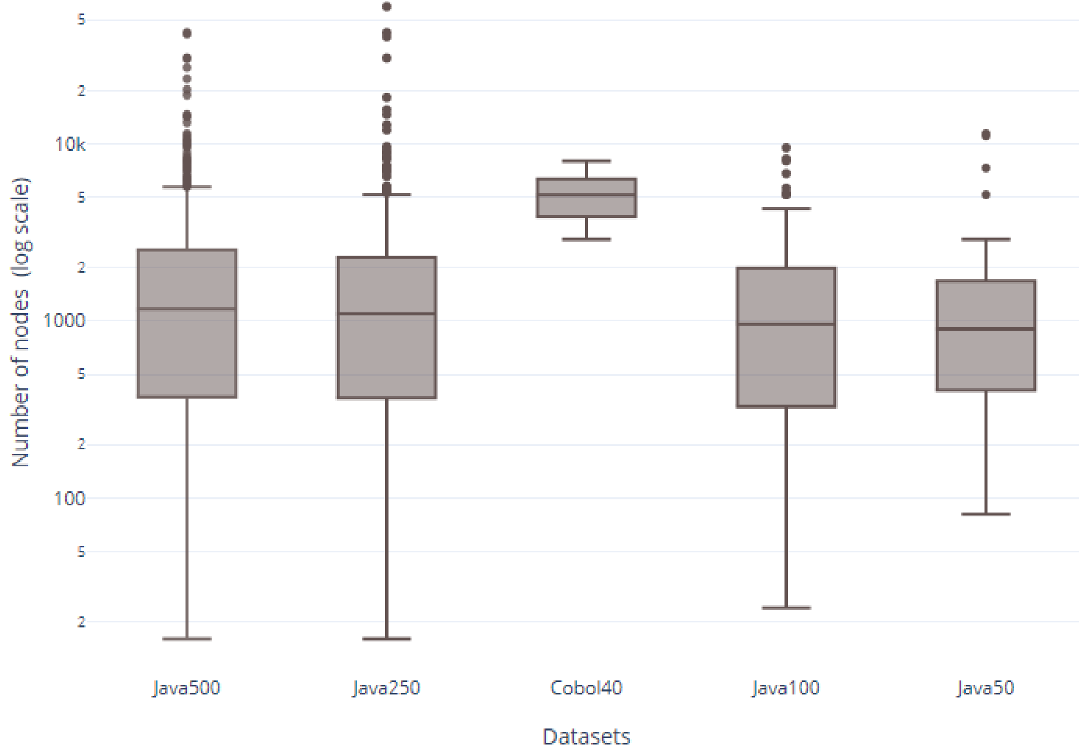
To test our algorithm, we use custom datasets extracted from the *QUAATLAS* corpus [15]. The research group made available the ASTs of each one of the seventy-nine repositories available in this corpus. Currently, mining in multiple repositories is too time-consuming, so we limit ourselves to smaller datasets extracted from this large corpus.

To build our four Java datasets, we started by randomly picking four software systems from the main corpus in addition to our original dataset of choice, `jhotdraw`. Then, we sampled some ASTs from them proportionally to their size. Overall, the selected systems are `quartz`, `webmail`, `cobertura`, `velocity`, and `jhotdraw` [30]. The four datasets contain 50, 100, 250, and 500 files respectively. Note that we preprocessed the ASTs and removed the nodes containing the data about the package and imports. Those two nodes usually contain very large subtrees and, following our observations, are not interesting as code idioms.

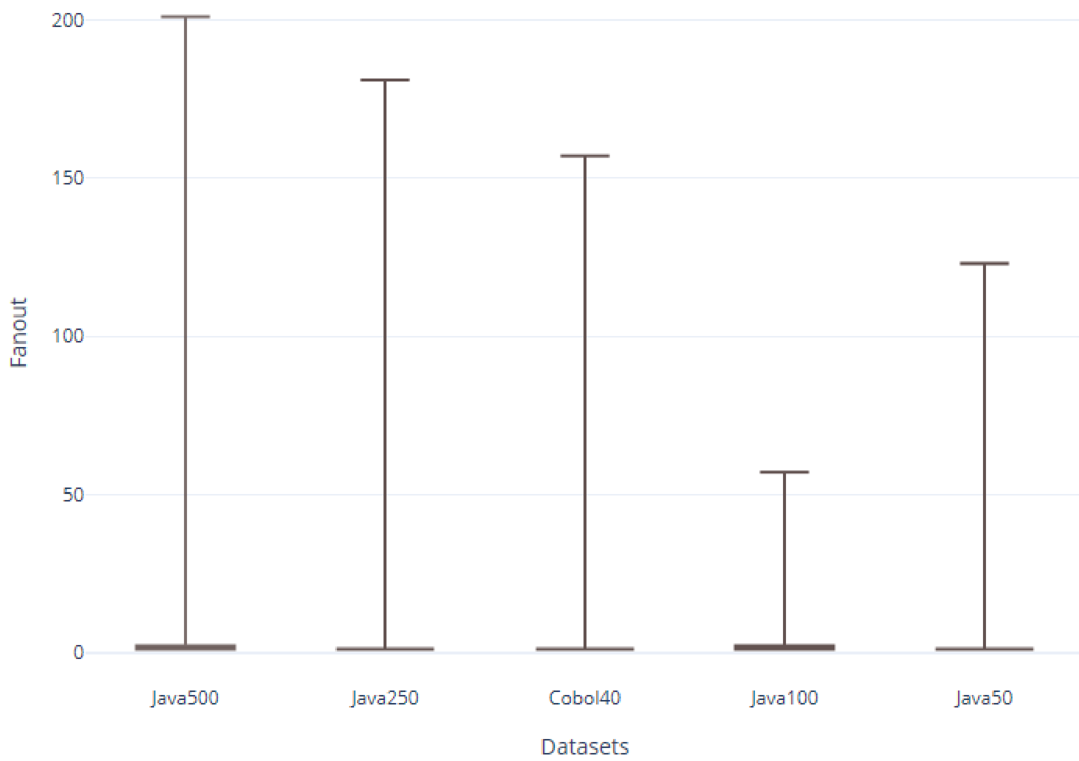
The last dataset is composed of ASTs of COBOL programs. The source code comes from the NIST COBOL 85 compliance test suite [28]. Those files and the dataset of 40 files that we use were made available by the research group.

Some characteristics of the datasets are represented in Figure 6.2. We observe that some files have a substantial number of nodes. The maximum is observed in the dataset of 250 files with an AST with 59634 nodes. We do not

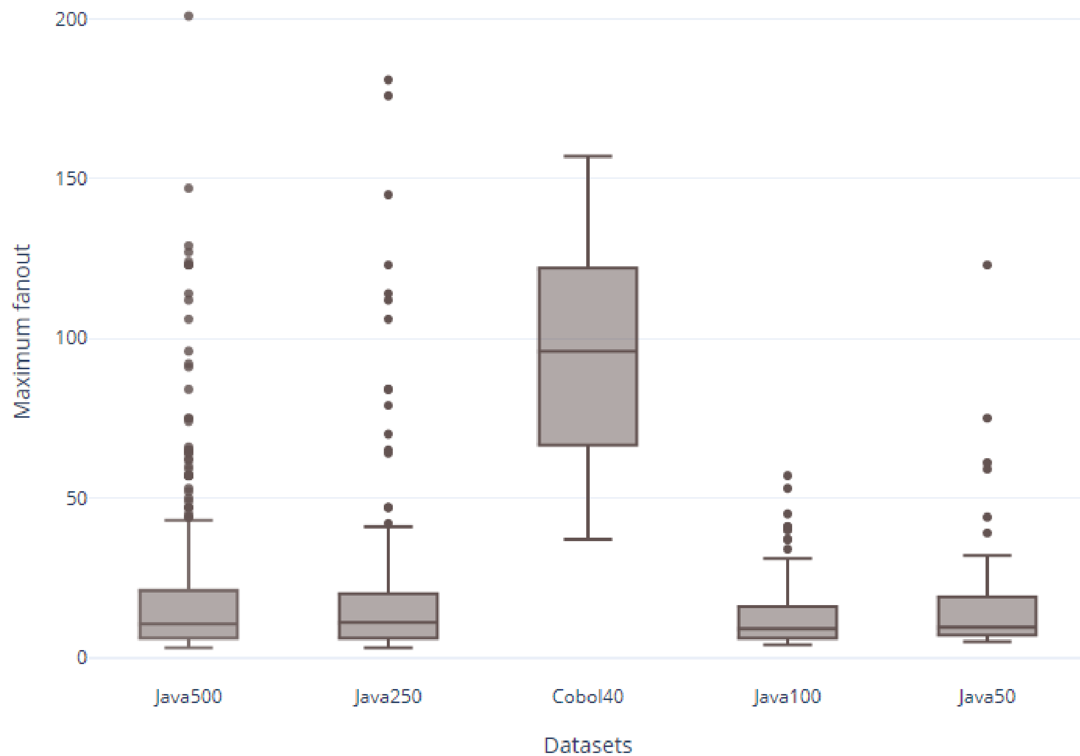
observe this phenomenon in COBOL dataset, but this dataset may not be representative since we picked ASTs from one single project. The fanout per node is small across all the datasets with an average between one and two nodes. Some nodes, however, have a large fanout of up to 200 nodes. The maximum fanout per file is, on average, between 17 and 19 nodes for the Java datasets. On the other hand, all files in the COBOL dataset contain at least one node with many children.



(A) Number of nodes per file



(B) Fanout per node



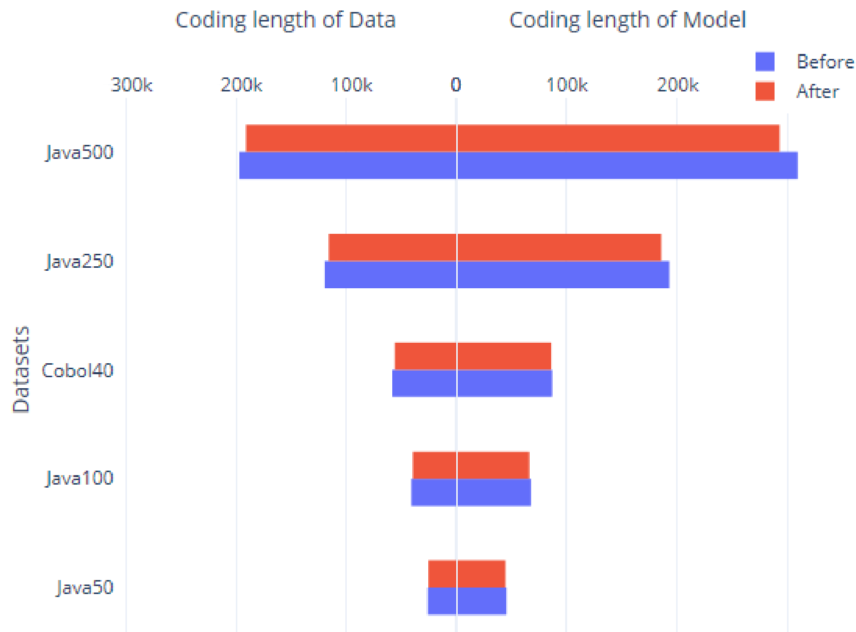
(c) Maximum fanout per file

Figure 6.2: Characteristics of the datasets

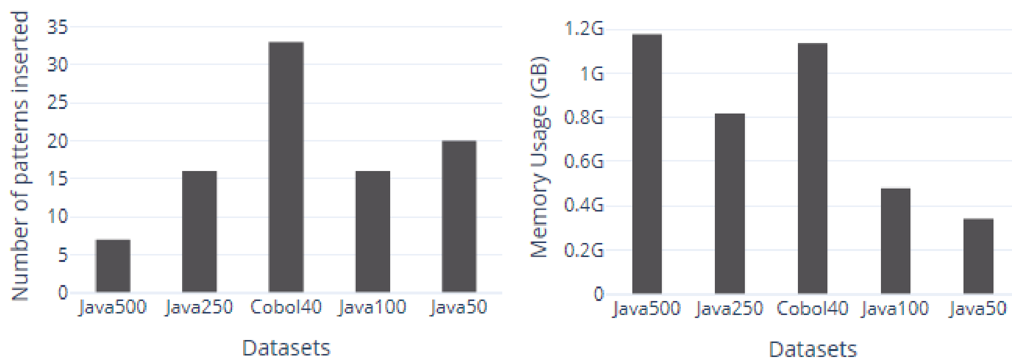
6.6.2 Experiments

To evaluate our compressed pattern miner, we chose the following setup. We ran our miner for five minutes on each dataset. We set the minimum support to 20%, the tabu tenure to 100 iterations and the not-improving iterations limit to 50. The only constraints that are used are the non-overlapping patterns constraint, as explained in Section 6.5 and not expanding patterns with duplicate children labels. The first constraint is forced by our implementation and cannot be toggled on and off. The second one is required for performance issues. Recall that some of the nodes can have a very high fanout as seen in Figure 6.2. Usually, the high fanout nodes have children with many duplicate labels. Without a specific way of addressing this issue, all of the tested pattern algorithms struggled, including the modified FreqT version with constraints.

The final results of our compressed pattern miner are represented in Figure 6.3. The first observation is that our patterns do not compress by much considering the total coding length of the dataset. The best reduction of 4.46% is observed on the Java500 dataset and the worst of 2.15% on the Java50 dataset. However, compressing is not the goal by itself. We are interested in whether the



(A) Coding length in number of bits



(B) Number of patterns in best result

(C) Memory usage

Figure 6.3: Summary of results obtained for compressing patterns

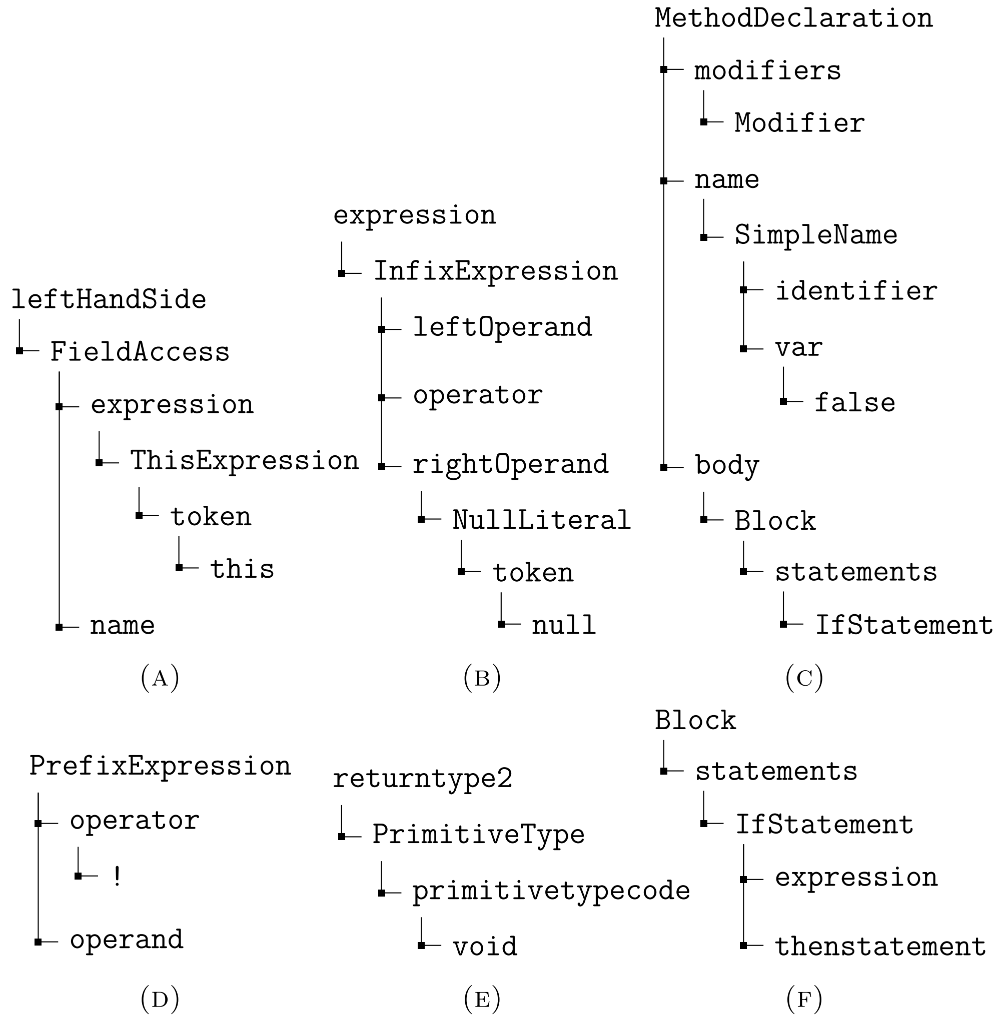


FIGURE 6.4: Examples of patterns found in Java

patterns reducing the coding length can be useful as code idioms or not. We also observe that the coding lengths for the models are typically larger than the coding lengths for the data. The Java500 dataset has the largest coding length for a model. Its coding length is equal to 308296.7 bits.

For Java, out of the 59 patterns found in total, 48 are unique. However, they typically have a large common subtree and a couple of different nodes attached to it. Also, we observe that the number of patterns for COBOL is higher than any Java dataset. This is probably due to the nature of the language which contains much more redundancies than Java. The memory usage is also higher for the COBOL dataset. We suspect that this is due to the high number of nodes on average in this dataset (Section 6.6.1).

Examples of Java patterns found can be seen in Figure 6.4. We observe that most of the patterns represent typical Java language constructs that are not necessarily interesting as code idioms. For example, the pattern Figure 6.4d tells us

that the operator in a prefix expression is often a negation, Figure 6.4e indicates that methods often return the type `void`, Figure 6.4f shows that there is often an `if` in the list of statements. Larger patterns tell us a bit, but not much more. Figure 6.4a tells us that the left-hand side of an assignment is often the `this` keyword, Figure 6.4b shows that the right-hand side of an `InfixExpression` is often a `null` token, Figure 6.4c describes a typical definition of a method in Java. The same analysis can be made for the COBOL patterns. We can see some trivial constructs, such as the fact that a COBOL program has a name. This construct is not a surprise because the language enforces it. Other patterns are a bit more involved, but still only describe some typical constructs that are not surprising even to a beginner COBOL developer.

Examples of possible constructs in the Java grammar may be helpful to interpret the results and can be found in Appendix A and Appendix B.

Chapter 7

Conclusion and future work

The goal of this work was to investigate whether frequent pattern mining techniques can be used to mine code idioms from the ASTs of source code. We worked in the context of an ongoing research project, INTiMALS [22], whose goal is to build an intelligent modernisation assistant for legacy code. Their current approach to mine code idioms is based on the *FreqT* algorithm with additional constraints on the patterns. Some of these constraints also take advantage of a predefined grammar of the language.

First, we added the closedness constraint to the existing pattern miner. We were motivated by the observation that too many patterns are reported and many of them are redundant. Reporting only closed patterns allows us to keep only a subset of those. We found that the closedness constraint does not combine well with other constraints already in use and requires to take a hit performance-wise to be implemented in the existing algorithm.

Next, we decided to use probabilistic models of source code to guide our pattern miner. We planned to build one using the grammar provided by the research group, but we have seen that constructing a grammar, even for a programming language, is not a trivial task. By parsing the ASTs using this grammar, we found discrepancies between the two. Our proposed fixes were accepted, but we decided to move away from a predefined grammar and infer it from the data instead. This simplification also allowed us to work on COBOL source code immediately.

Finally, we investigate whether compressing patterns can be seen as code idioms. The goal is to find the probabilistic tree substitution grammar which provides the best compression based on the Minimum Description Length principle. We achieve this by greedily mining for patterns that provide the best compression and adding them to our TSG. Our evaluation on four Java and one COBOL datasets shows that the compression provided is limited, but the patterns found do reflect some typical constructions in the language. However, those constructions would be considered too typical and too small to be

considered as code idioms.

This leads us to possible improvements to our compressing pattern miner.

Many existing heuristics for local search can be used to help us guide the search process. We can even reintroduce some of the constraints already proposed by the research group. The goal in our pattern miner was to reduce the constraints required as much as possible, but it appears that they cannot be ignored.

The performances of the pattern miner can be improved. The constraint that prevents us from expanding children with duplicate labels helps a lot but may make us miss some interesting patterns. We cannot relax this constraint completely either due to nodes with very large fanouts.

The way we defined the coding length of the model and the data given the model can also be changed. Our model may penalise too much adding new patterns. Our miner is thus unable to find suitable candidates to add. We can also use this second model to make comparisons and get a better understanding of what would be a good model when one wants to find code idioms.

Appendix A

Appendix A

```
<.../>
<MethodDeclaration>
    <javadoc optional="true" node="Javadoc"/>
    <modifiers optional="true" ordered-nodelist="IExtendedModifier"/>
    <constructor simplevalue="boolean"/>
    <typeParameters optional="true" unordered-nodelist="TypeParameter"/>
    <returnType2 optional="true" node="Type"/>
    <name node="SimpleName"/>
    <receiverType optional="true" node="Type"/>
    <receiverQualifier optional="true" node="SimpleName"/>
    <parameters optional="true" ordered-nodelist="SingleVarDeclaration"/>
    <extraDimensions2 optional="true" unordered-nodelist="Dimension"/>
    <thrownExceptionTypes optional="true" unordered-nodelist="Type"/>
    <body optional="true" node="Block"/>
</MethodDeclaration>
<InfixExpression>
    <leftOperand optional="true" node="Expression"/>
    <operator simplevalue="Operator"/>
    <rightOperand optional="true" node="Expression"/>
    <extendedOperands optional="true" unordered-nodelist="Expression"/>
</InfixExpression>
<ThisExpression>
    <qualifier optional="true" node="Name"/>
</ThisExpression>
<NumberLiteral>
    <token simplevalue="String"/>
</NumberLiteral>
<Expression abstract="true">
    <ThisExpression/>
</Expression>
<Expression abstract="true">
    <InfixExpression/>
</Expression>
<Expression abstract="true">
    <NumberLiteral/>
</Expression>
<.../>
```

LISTING 1: Example of the Java grammar in the metamodel format. The first rule defines the AST of a method declaration in a class. The following rules define an infix and `this` expression. Notice that both of them are inside a `Expression` tag declared as abstract.

Appendix B

Appendix B

```
<.../>
<Javadoc>
    <tags unordered-nodelist="TagElement"/>
</Javadoc>
<TagElement>
    <tagName simplevalue="String"/>
    <fragments unordered-nodelist="IDocElement"/>
</TagElement>
<IDocElement abstract="true">
    <TagElement/>
</IDocElement>
<IDocElement abstract="true">
    <.../>
</IDocElement>
<.../>
```

LISTING 2: Another example of the Java grammar in the meta-model format. The rules given here define the representation of `javadoc` in an AST. The `Javadoc` node contains a list of `TagElement`. A `TagElement` can be, for example, `@return - value`. Other possible values for `IDocElement` are omitted for clarity.

Bibliography

[1] Mithun Acharya et al. “Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications”. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE ’07. Dubrovnik, Croatia: ACM, 2007, pp. 25–34. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287630.

[2] Charu C. Aggarwal and Jiawei Han, eds. *Frequent Pattern Mining*. Springer, 2014. ISBN: 978-3-319-07820-5. DOI: 10.1007/978-3-319-07821-2.

[3] Rakesh Agrawal and Ramakrishnan Srikant. “Fast Algorithms for Mining Association Rules in Large Databases”. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. ISBN: 1-55860-153-8.

[4] Miltiadis Allamanis and Charles Sutton. “Mining Idioms from Source Code”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. The 22nd ACM SIGSOFT International Symposium. Hong Kong, China: ACM Press, 2014, pp. 472–483. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635901.

[5] Miltiadis Allamanis and Charles Sutton. “Mining Source Code Repositories at Massive Scale Using Language Modeling”. In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013 10th IEEE Working Conference on Mining Software Repositories (MSR 2013). San Francisco, CA, USA: IEEE, May 2013, pp. 207–216. ISBN: 978-1-4799-0345-0. DOI: 10.1109/MSR.2013.6624029.

[6] Miltiadis Allamanis et al. “A Survey of Machine Learning for Big Code and Naturalness”. In: *ACM Computing Surveys* 51.4 (July 2018), pp. 1–37. ISSN: 0360-0300. DOI: 10.1145/3212695.

[7] Tatsuya Asai et al. “Efficient Substructure Discovery from Large Semi-Structured Data”. In: *Proceedings of the 2002 SIAM International Conference on Data Mining*. Ed. by Robert Grossman et al. Philadelphia, PA: Society for Industrial and Applied Mathematics, Apr. 11, 2002, pp. 158–174. DOI: 10.1137/1.9781611972726.10.

[8] H.A. Basit and S. Jarzabek. “A Data Mining Approach for Detecting Higher-Level Clones in Software”. In: *IEEE Transactions on Software Engineering* 35.4 (July 2009), pp. 497–514. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.16.

[9] I.D. Baxter et al. “Clone Detection Using Abstract Syntax Trees”. In: *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. Proceedings. International Conference on Software Maintenance. Bethesda, MD, USA: IEEE Comput. Soc, 1998, pp. 368–377. ISBN: 978-0-8186-8779-2. DOI: 10.1109/ICSM.1998.738528.

[10] Pavol Bielik, Veselin Raychev, and Martin Vechev. “PHOG: Probabilistic Model for Code”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, June 20–22, 2016, pp. 2933–2942.

[11] Rens Bod. “The Data-Oriented Parsing Approach: Theory and Application”. In: *Computational Intelligence: A Compendium*. Ed. by John Fulcher and L. C. Jain. Red. by Janusz Kacprzyk. Vol. 115. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 307–348. ISBN: 978-3-540-78293-3. DOI: 10.1007/978-3-540-78293-3_7. URL: http://link.springer.com/10.1007/978-3-540-78293-3_7.

[12] Yun Chi et al. “CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees”. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang. Red. by Takeo Kanade et al. Vol. 3056. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 63–73. ISBN: 978-3-540-24775-3. DOI: 10.1007/978-3-540-24775-3_9.

[13] Yun Chi et al. “Frequent Subtree Mining - an Overview”. In: *Fundamenta Informaticae - Advances in Mining Graphs, Trees and Sequences* 66 (2005), p. 161. ISSN: 0169-2968.

[14] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. “Inducing Tree-Substitution Grammars”. In: *J. Mach. Learn. Res.* 11 (Dec. 2010), pp. 3053–3096. ISSN: 1532-4435.

[15] Coen De Roover, Ralf Lammel, and Ekaterina Pek. "Multi-Dimensional Exploration of API Usage". In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013 IEEE 21st International Conference on Program Comprehension (ICPC). San Francisco, CA, USA: IEEE, May 2013, pp. 152–161. ISBN: 978-1-4673-3092-3. DOI: 10.1109/ICPC.2013.6613843.

[16] The Eclipse Foundation. *JDT Core Component*. URL: https://www.eclipse.org/jdt/core/ (visited on 12/24/2018).

[17] Mark Gabel and Zhendong Su. "A Study of the Uniqueness of Source Code". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE '10*. The Eighteenth ACM SIGSOFT International Symposium. Santa Fe, New Mexico, USA: ACM Press, 2010, p. 147. ISBN: 978-1-60558-791-2. DOI: 10.1145/1882291.1882315.

[18] Peter D. Grnwald, In Jae Myung, and Mark A. Pitt. *Advances in Minimum Description Length: Theory and Applications (Neural Information Processing)*. The MIT Press, 2005. ISBN: 0-262-07262-9.

[19] Dick Grune. *Parsing Techniques: A Practical Guide*. 2nd. Springer Publishing Company, Incorporated, 2010. ISBN: 978-1-4419-1901-4.

[20] Michael Hielscher. *AtoCC*. URL: http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=en&site=main (visited on 10/29/2018).

[21] Abram Hindle et al. "On the Naturalness of Software". In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 837–847. ISBN: 978-1-4673-1067-3.

[22] *INTiMALS - Intelligent Modernisation Assistance for Legacy Software*. URL: http://soft.vub.ac.be/intimals/index.html (visited on 12/17/2018).

[23] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. "POTMiner: Mining Ordered, Unordered, and Partially-Ordered Trees". In: *Knowledge and Information Systems* 23.2 (May 2010), pp. 199–224. ISSN: 0219-1377, 0219-3116. DOI: 10.1007/s10115-009-0213-3.

[24] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009. ISBN: 0-13-187321-0.

[25] Sangeetha Kutty, Richi Nayak, and Yuefeng Li. “PCITMiner – Prefix-Based Closed Induced Tree Miner for Finding Closed Induced Frequent Subtrees”. In: (2007), p. 10.

[26] Chris J. Maddison and Daniel Tarlow. “Structured Generative Models of Natural Source Code”. In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, pp. 649–657.

[27] *Mining Software Repositories*. URL: http://www.msrconf.org/ (visited on 12/29/2018).

[28] Carmelo Montanez-Rivera. *Cobol Available Test Suite*. URL: https://www.itl.nist.gov/div897/ctg/cobol_form.htm (visited on 12/30/2018).

[29] Dario Di Nucci et al. “Language-Parametric Modular Framework for Mining Idiomatic Code Patterns”. Unpublished. 2018.

[30] *Qualitas Corpus*. URL: http://qualitascorpus.com/ (visited on 12/20/2018).

[31] *Raincode*. URL: https://www.raincode.com/ (visited on 07/29/2018).

[32] J. Rissanen. “Modeling by Shortest Data Description”. In: *Automatica* 14.5 (Sept. 1978), pp. 465–471. ISSN: 00051098. DOI: 10.1016/0005-1098(78)90005-5.

[33] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach”. In: *Science of Computer Programming* 74.7 (May 2009), pp. 470–495. ISSN: 01676423. DOI: 10.1016/j.scico.2009.02.007.

[34] C. E. Shannon. “A Mathematical Theory of Communication”. In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[35] Ewan Tempero et al. “Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies”. In: *2010 Asia Pacific Software Engineering Conference (APSEC2010)*. Dec. 2010, pp. 336–345. DOI: http://dx.doi.org/10.1109/APSEC.2010.46.

[36] Jilles Vreeken, Matthijs van Leeuwen, and Arno Siebes. “Krimp: Mining Itemsets That Compress”. In: *Data Mining and Knowledge Discovery* 23.1 (July 2011), pp. 169–214. ISSN: 1384-5810, 1573-756X. DOI: 10.1007/s10618-010-0202-x.

[37] *Vrije Universiteit Brussel*. URL: https://www.vub.ac.be/en/ (visited on 12/20/2018).

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