The Core of Change

Frequent Subgraph Discovery in Software Evolution Graphs

Bachelor Thesis

Faculty of Science, University of Bern

submitted by

Damian Rhyn

from Waldkirch, Switzerland

Supervision:

PD Dr. Kaspar Riesen

Institute of Computer Science (INF)

University of Bern, Switzerland

Abstract

Software undergoes constant evolution in the form of code changes. These changes can be modeled as graphs, so-called variation diffs, that capture the structural evolution of software. This thesis explores the application of frequent subgraph mining (FSM) on variation diffs. The goal is to find interesting patterns in these software evolution graphs. The GraMi algorithm [1] is selected for mining the frequent subgraphs, based on its performance and compatibility with the data. To enhance the quality of the output, several pre- and post-processing strategies are proposed. These include filtering the input graphs to remove unchanging structures, extracting previously found subgraphs from the input graphs, and randomly relabeling certain nodes and edges to influence the mining process toward more significant patterns.

Experimental results show that running GraMi on the unmodified data predominantly reveals unchanged code structures. The proposed pre- and post-processing methods improve the quality of the output. The findings suggest that FSM, supported by some processing steps, is a promising approach for finding interesting patterns in software evolution, though challenges related to performance and pattern relevance remain.

Contents

Chapter 1

Introduction

In this section, the topic of this thesis is embedded in a broader context, and the research question is articulated. Then, an overview of the subsequent structure of the thesis is provided.

1.1 Frequent Subgraph Mining in Context

The field of computer science is characterized by a broad spectrum of disciplines focused on the representation, processing, and analysis of data. This encompasses areas such as algorithms and data structures, security and cryptography, and computer graphics and visualization [2].

Among its many subfields, pattern recognition plays a central role in enabling computers to detect regularities in data [3]. Whether the task involves recognizing handwritten digits [4], classifying images [5], or analyzing code change patterns [6], pattern recognition provides a foundation for identifying structures in complex datasets.

Within pattern recognition, a distinction exists between the type of data, ranging from sequences and time series [7] to trees [8] and graphs [9], each requiring specialized analytical techniques. Graph-based pattern recognition focuses on data represented as nodes and edges, a natural format for modeling phenomena as diverse as social network interactions [10], biochemical pathways [11], transportation networks [12], or software and its evolution [13].

Methods for analyzing graph-structured data include graph clustering and community detection (to uncover tightly knit groups of nodes) [14], graph classification (to assign whole-graph labels, for instance, identifying malicious versus benign network topologies) [15], and, more recently, graph neural networks (GNNs) for end-to-end learning on graph inputs [16].

Frequent subgraph mining (FSM) remains a core technique in this landscape, allowing the discovery of subgraphs that recur across a collection of graphs and revealing common structures or shared building blocks [17].

1.2 Software Evolution as Graphs

Graphs are a versatile data structure that can represent a wide variety of real-world and abstract systems. In the context of code analysis, source code can be mapped onto graphs in many ways. Abstract syntax trees capture the hierarchical structure of language constructs [18], control-flow graphs model possible execution paths [19], call graphs describe inter-procedural invocation relationships [20], and program-dependence graphs (PDGs) combine control and data dependencies into a single representation [21]. Each of these graph views highlights different aspects of a program's behavior and structure, making them invaluable for tasks ranging from optimization [22] and bug detection [23] to change analysis and refactoring support [24].

This thesis is focused on the analysis of changes between versions of code. As software undergoes development, programmers implement modifications to introduce new functionality, fix bugs, or refactor code. When these modifications are represented in graphs, nodes represent elements of code, and edges represent the relationship between these elements. The nodes and edges are labeled to indicate the type of code element and how the elements and their relationships have changed. In order to encompass the intricacies of these relationships, the edges are directed. Every specific code change is therefore encoded as a labeled, directed graph [25].

In this context, a frequent subgraph represents a recurring change pattern. Identifying these change patterns is valuable for several reasons. For instance, it informs tools used for code auto-completion [26], automatic refactorings [27], or quick fixes [28]. By mining frequent subgraphs from variation graphs, we gain insight into the evolutionary patterns of software [13].

1.3 Frequent Subgraph Mining Algorithms

A multitude of different FSM algorithms have been developed over the years, each with distinct capabilities and optimizations depending on the requirements of the input data. Among these, some stand out for their impact and innovations.

Among the earliest is gSpan, which was presented in 2002 [29]. gSpan introduced a new lexicographic ordering of graphs and a DFS-code representation. This innovation made it significantly more efficient than earlier approaches and suitable for mining large datasets of undirected, labeled graphs.

Algorithm	Node labels	Edge labels	Directed
gSpan	✓	✓	✗
GraMi	✓	✓	✓
FS^3	✓	✓	✗
T-FSM	✓	✓	✗
SPMiner	✓	✗	✓

Table 1.1: FSM algorithms and their capabilities.

GraMi was proposed in 2014 [1] to tackle the scalability bottlenecks of FSM on large, labeled, directed graphs. GraMi adds pruning strategies to reduce the search space as well as other optimizations, including a heuristic. These techniques allow it to avoid a large fraction of expensive matching operations and maintain a compact in-memory data footprint, leading to significant speed-ups over earlier methods on real-world graphs.

FS^3 [30] employs a probabilistic approach by sampling subgraphs and estimating their frequencies. It was proposed in 2015 and operates on undirected labeled graphs. FS^3 is particularly useful when the dataset is of considerable size and exhaustive enumeration is not feasible.

More recently, T-FSM was proposed in 2023 [31] and is designed to leverage hardware capable of parallel processing, such as multicore processors. T-FSM operates on labeled, undirected graphs and boasts high scalability for graph mining on large data platforms.

SPMiner, introduced in 2023 [32], employs a novel approach by integrating concepts from representation learning and subgraph mining. It learns vector embeddings of subgraph structures to guide the mining process and supports directed graphs with node labels. The utilization of representation learning enables it to scale more effectively in comparison to exhaustive pattern enumeration methods.

The key capabilities of these algorithms are summarized in Table 1.1.

Given that the data consists of directed graphs with both node and edge labels, GraMi is an ideal choice for the FSM tasks in this thesis. By applying GraMi to these graphs, the thesis aims to uncover interesting patterns in variation diffs.

1.4 Thesis Structure

The remainder of this thesis is organized as follows.

After this introduction, there will be an explanation of FSM in general and the GraMi algorithm in particular. Chapter 3 will provide a more thorough exploration of the data and how it was generated as well as the application of GraMi on this data. This will be followed by three possible pre- and post-processing methods to improve the output of GraMi for this specific data. Chapter 5 will conclude the thesis with a summary and suggestions for future work.

Chapter 2

Foundation

GraMi, proposed by Elseidy et al. in 2014 [1], is an FSM algorithm designed to operate on a single large, directed graph with both node and edge labels. Rather than storing every embedding of a candidate subgraph, GraMi stores only compact templates and reformulates the FSM problem as a constraint satisfaction problem (CSP). This CSP formulation enables GraMi to discover the minimal set of subgraph appearances required to verify the frequency, thereby avoiding redundant searches.

To further improve performance, GraMi employs a heuristic and a suite of optimizations that prune entire branches of the search tree, prioritize fast searches and defer more expensive ones, and exploit special graph structures when present.

This chapter begins with a review of the fundamentals of frequent subgraph mining and then moves on to explain the workings of GraMi. Finally, the optimization techniques are explained in greater detail.

2.1 Frequent Subgraph Mining and GraMi

2.1.1 Foundations of Frequent Subgraph Mining

To help with the comprehension of the following detailed description of FSM, some intuition is provided first. The idea behind FSM is to identify subgraphs in a graph that appear frequently. A threshold is defined, and a subgraph is considered frequent if it occurs at least as many times in the graph as the threshold defines [1].

For example, if we consider the graph in Figure 2.1(a) and set the frequency threshold τ to 3, the only subgraph that is considered frequent is the one shown in Figure 2.1(b). If the threshold is lowered to $\tau = 2$, however, both the subgraphs illustrated in Figure 2.1(c) also qualify as frequent.

Before delving into an in-depth description of GraMi, the groundwork of a formal foundation for FSM is established.

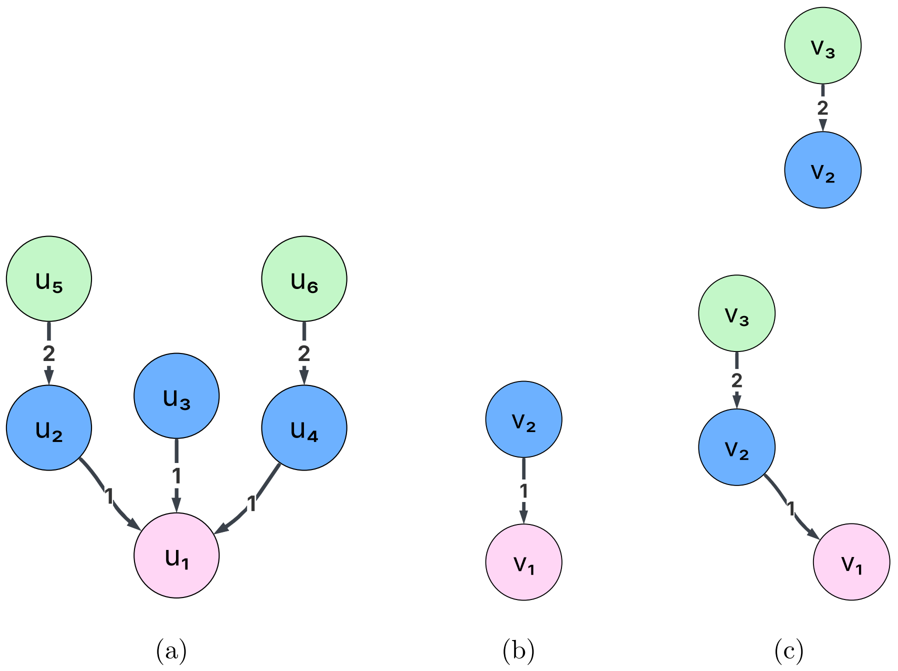


Figure 2.1: Frequent subgraphs found using different thresholds.

A graph $G = (V, E, L)$ consists of a set of nodes V, a set of edges E, and a labeling function L that assigns labels to nodes and edges [1].

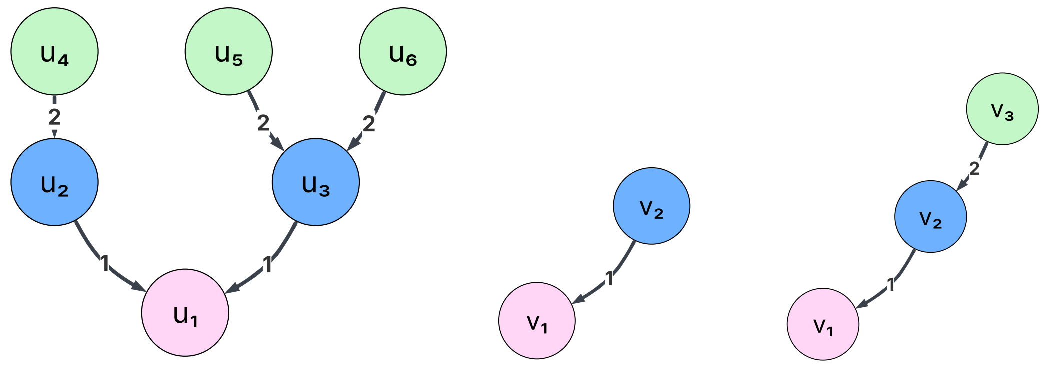
A graph $S = (V_S, E_S, L_S)$ is a subgraph of a graph $G = (V, E, L)$ if and only if $V_S \subseteq V$, $E_S \subseteq E$, and $L_S(v) = L(v)$ for all $v \in V_S \cup E_S$ [1].

The data used in this thesis contains nodes and edges with a single label each, and all edges are directed. For illustrative purposes, edges are represented by arrows, and nodes are colored according to their labels.

Definition 1 (Subgraph Isomorphism, from [1])**.** *Let $S = (V_S, E_S, L_S)$ be a subgraph of a graph $G = (V, E, L)$. A* subgraph isomorphism *of S to G is an injective function $f : V_S \to V$ such that $L_S(v) = L(f(v))$ for all nodes $v \in V_S$, and $(f(u), f(v)) \in E$ with $L_S(u, v) = L(f(u), f(v))$ for all edges $(u, v) \in E_S$.*

A subgraph isomorphism finds a smaller graph inside a larger one by establishing a one-to-one correspondence between their nodes and edges, ensuring that both connections and labels match. For example, the subgraph in Figure 2.1(b) can be mapped to the larger graph in 2.1(a) in three distinct ways: in every mapping, v_1 corresponds to u_1, while v_2 corresponds to u_2, u_3, or u_4.

To measure the support of a subgraph, meaning the number of times it can be found in a graph, one can count the number of subgraph isomorphisms. However, when determining the support in this way, it is not anti-monotone. A measure is anti-monotone when its support cannot increase when extending a subgraph [1].



(a) (b) (c)

Figure 2.2: Anti-monotonicity example.

This is not the case for counting subgraph isomorphisms, which is illustrated in Figure 2.2.

The subgraph in Figure 2.2(b) has two subgraph isomorphisms in the graph in Figure 2.2(a), namely mapping v_1 to u_1 and v_2 to u_2 or u_3. When extending the subgraph to the graph seen in Figure 2.2(c), the support increases to three, because v_3 can map to u_4, u_5, or u_6. Having an anti-monotone metric is crucial for pruning, as a subgraph whose support falls below the threshold can only be discarded if one knows its support cannot rise again in any extension [1].

There are various ways to measure support, many of which are anti-monotone by design, such as homomorphism-based support [29] or maximum edge coverage [33]. GraMi employs the minimum-image-based support (MNI) metric [1], which counts the number of distinct nodes in each domain and takes the minimum of these counts as the pattern's support.

Definition 2 (Minimum Image-based Support (MNI), from [1])**.** *Let $f_1, \dots, f_m$ be the set of subgraph isomorphisms of a subgraph $S = (V_S, E_S, L_S)$ in a graph G. Also let $F(v) = \{f_1(v), \dots, f_m(v)\}$ be the set that contains the (distinct) nodes in G whose functions $f_1, \dots, f_m$ map a node $v \in V_s$. The* minimum image-based support *(MNI) of S in G, denoted by $s_G(S)$, is defined as $s_G(S) = \min\{t \,|\, t = |F(v)| \, for \, all \, v \in V_S\}$.*

Using this metric on the example in Figure 2.2 and looking at the subgraph in Figure 2.2(b) as S_1, one can deduce that $F(v_1) = \{u_1\}$ and $F(v_2) = \{u_2, u_3\}$, therefore $s_G(S_1) = 1$. Extending this to the subgraph in Figure 2.2(c) as S_2 with $F(v_3) = \{u_4, u_5, u_6\}$ does not change the MNI, so $s_G(S_2) = s_G(S_1) = 1$.

With these definitions, the FSM problem can be formally stated as follows:

Problem 1 (Frequent Subgraph Isomorphism Mining, from [1])**.** *Given a graph G and a minimum support threshold τ, the* frequent subgraph isomorphism mining problem *is defined as finding all subgraphs S in G such that $s_G(S) \geq \tau$.*

Note that Problem 1 does not require an exact calculation of the MNI; it only requires that the MNI be greater than τ. This fact allows the problem to be modeled as a constraint satisfaction problem (CSP) [1].

Before defining a CSP formally, it helps to build some intuition. At its core, a CSP is a mathematical framework used for finding values that satisfy a collection of rules. The CSP consists of variables, each of which must be assigned a value; a domain of possible values for each variable; and a set of constraints [1]. This concept is common in many everyday scenarios, for example, solving a Sudoku puzzle, where each empty cell is a variable, the numbers 1–9 form the domain, and the rules of Sudoku serve as the constraints.

In the context of subgraph isomorphisms, CSPs provide a structured way to describe how nodes of a smaller graph can be mapped to nodes of a larger graph while preserving connectivity and labels. For each node in a subgraph, a corresponding node in the larger graph must be found that has the same label and edges. If a matching node can be found for each node in the subgraph, and if the connectivity and labels are correct, then this is considered a solution to the CSP. In this case, the CSP's variables match the subgraph's nodes, its domains are all nodes from the large graph that could potentially match, and its constraints are the connectivity and labels. Formally, this can be described as follows:

Definition 3 (Subgraph S to Graph G CSP, from [1])**.** *Let $S = (V_S, E_S, L_S)$ be a subgraph of a graph $G = (V, E, L)$. The* subgraph S to graph G CSP *is defined as a constraint satisfaction problem (X, D, C), where:*

- *X contains a variable x_v for every node $v \in V_S$.*
- *D is a set of domains, where each variable $x_v \in X$ has a domain $D_v \subseteq V$.*
- *Set C contains the following constraints:*
    - *$x_v \neq x_{v'}$ for all distinct $v, v' \in V_S$.*
    - *$L(x_v) = L_S(v)$ for all $v \in V_S$.*
    - *$L(x_v, x_{v'}) = L_S(v, v')$ for all $(v, v') \in E_S$.*

Whenever it is clear from the context, v may be used to refer to the corresponding variable x_v to simplify notation.

An assignment of a node u to a variable v is considered valid if and only if there exists a solution to the CSP that assigns u to v. Each valid assignment corresponds to an isomorphism [1].

Proposition 1 (from [1]). *Let (X, D, C) be the subgraph S to graph G CSP. The MNI support of S in G satisfies τ, i.e., $s_G(S) \geq \tau$, if and only if every variable in X has at least τ distinct valid assignments (i.e., isomorphisms of S in G).*

The GraMi algorithm builds on this proposition, which allows it to consider the number of valid assignments for each variable to determine if a subgraph is frequent.

Proposition 1 shows that the MNI support of a subgraph S in G is simply the smallest number of distinct CSP assignments (i.e., node mappings) any node from S can take. This gives us a direct, variable-centric way to test frequency. Once subgraph matching has been modeled as a CSP, the MNI support can be computed by examining each variable's set of valid assignments. If every variable can reach at least τ different host nodes, then S occurs with a support larger than τ; otherwise it does not [1].

Proposition 1 thus provides the formal justification for GraMi's core strategy. Rather than explicitly enumerating every embedding of S, GraMi simply verifies that the MNI support of S meets the threshold τ and uses pruning and other optimization techniques to quickly determine whether a subgraph is frequent or can be discarded.

2.1.2 The GraMi Algorithm

Algorithm 1 FrequentSubgraphMining(G, τ) (from [1])

```
1: $result \leftarrow \emptyset$
2: Let $fEdges$ be the set of all frequent edges of $G$
3: for each $e \in fEdges$ do
4:     $result \leftarrow result \cup \textsc{SubgraphExtension}(e, G, \tau, fEdges)$
5:     Remove $e$ from $G$ and $fEdges$
6: end for
7: return $result$
```

The main loop of the GraMi algorithm is shown in Algorithm 1 [1]. Due to the anti-monotone property of the MNI measure, only frequent edges—those that occur more than τ times—need to be considered for frequent subgraphs. Based on this, the subgraph candidates begin as a single edge from the set of frequent edges $fEdges$ (line 3) and are iteratively extended by an edge from $fEdges$ by Algorithm 2 (line 4) [1].

Algorithm 2 SubgraphExtension($S, G, \tau, fEdges$) (from [1])

```
1: $result \leftarrow S, candidateSet \leftarrow \emptyset$
2: for each edge $e$ in $fEdges$ and node $u$ of $S$ do
3:     if $e$ can be used to extend $u$ then
4:         Let $ext$ be the extension of $S$ with $e$
5:         if $ext$ is not already generated then
6:             $candidateSet \leftarrow candidateSet \cup ext$
7:         end if
8:     end if
9: end for
10: for each $c \in candidateSet$ do
11:     if $s_G(c) \geq \tau$ then
12:         $result \leftarrow result \cup \textsc{SubgraphExtension}(e, G, \tau, fEdges)$
13:     end if
14: end for
15: return $result$
```

In Algorithm 2 [1], the candidate subgraph S is, if possible, extended by a frequent edge (line 4). If an extension has already been considered, it is skipped (line 5). Then, based on the anti-monotone property, only frequent subgraphs are considered for further extension (line 11). The algorithm recursively calls itself on these subgraphs, extending each one until no further frequent extension exists (line 12) [1].

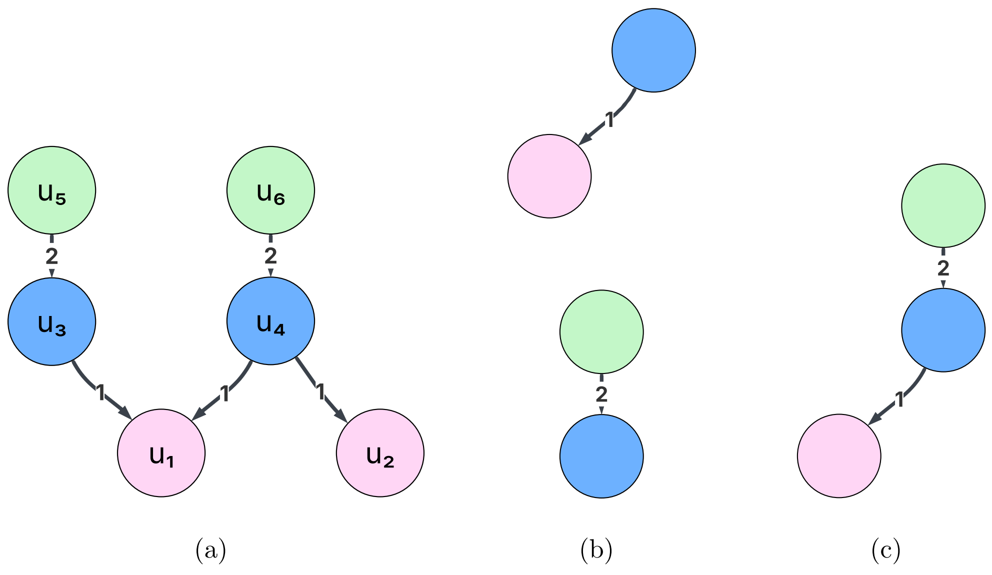


Figure 2.3: Subgraph extension process example.

For clarity, an example of this process can be seen in Figure 2.3. Assuming a support threshold of two, graph G, as seen in Figure 2.3(a), has two frequent edges, which can be seen in Figure 2.3(b). These will be called e_1 and e_2 for the top and bottom, respectively. Node u_1 cannot be extended by edge e_2, because e_2 does not contain any pink nodes. However, it can be extended by e_1. If this extension does not already exist—for example, by extending u_3 to u_1—it will be added to the candidate set. From there, the candidate is further extended by edge e_2. Again, if this further extension, depicted in Figure 2.3(c), is new, it is added to the candidate set as well. This process continues with u_1 as the origin, and then it is repeated with each other node in G as the origin. After generating the entire candidate set, the frequency of each candidate is checked. If a candidate is frequent, it is added to the result set. In this example, the subgraphs shown in Figures 2.3(b) and 2.3(c) make up the result set, which is returned as the output of the algorithm.

Algorithm 3 IsFrequentCSP(S, G, τ) (from [1])

```
 1: Consider the subgraph $S$ to graph $G$ CSP
 2: Apply node and arc consistency
 3: if the size of any domain is less than $\tau$ then
 4:     return $false$
 5: end if
 6: for each solution $Sol$ of the $S$ to graph $G$ CSP do
 7:     Mark all nodes of $Sol$ in the corresponding domains
 8:     if all domains have at least $\tau$ marked nodes then
 9:         return $true$
10:     end if
11: end for
12: return $false$
```

An integral part of Algorithm 2 is the frequency check on line 11. To perform this check, consider Algorithm 3 [1], which is based on Proposition 1. This algorithm determines whether a subgraph S is frequent in a graph G according to a threshold τ. To achieve this, it first asserts node and arc consistency (line 2) [1].

Node consistency involves excluding nodes from G that cannot be matched, such as nodes with labels that are not present in S or nodes with a lower degree than those in S. Arc consistency ensures the consistency between the assignments of two variables. Specifically, if nodes v and v' in S are connected by an edge, arc consistency ensures that each candidate in the domain of v has a compatible candidate in the domain of v' that is connected by an edge with the correct label. If, after enforcing these consistencies, the size of any domain is smaller than τ, the algorithm can return $false$ based on Proposition 1 (line 4) [1].

Following this pruning, the algorithm considers all solutions to the subgraph S to graph G CSP and marks the corresponding nodes in the domains (line 7). According to Proposition 1, S is frequent if all domains have at least τ marked nodes [1].

Algorithm 3 provides a basic idea for determining the frequency of a subgraph using Proposition 1. Further optimizations are presented in the following chapter.

2.2 Optimization Techniques

GraMi implements multiple optimizations, primarily when checking the frequency of a subgraph in a graph. This section will first explain the heuristic used to speed up the search process. Then Subsections 2.2.2, 2.2.3, and 2.2.4 cover optimizations using a lookup table, the properties of special graphs, and symmetric candidate subgraphs, respectively. Finally, Subsections 2.2.5 and 2.2.6 address inefficient searches. The pseudocode for the frequency checking algorithm that incorporates all of these optimizations can be found in Appendix A.

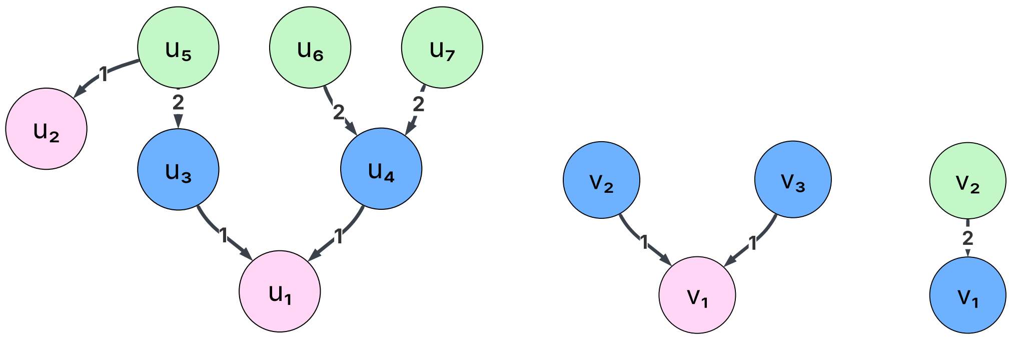
2.2.1 Heuristic

Algorithm 4 IsFrequentHeuristic(S, G, τ) (from [1])

```
1: Consider the subgraph $S$ to graph $G$ CSP
2: Apply node and arc consistency
3: for each variable $v$ with domain $D$ do
4:     $count \leftarrow 0$
5:     Apply arc consistency
6:     if the size of any domain is less than $\tau$ then
7:         return $false$
8:     end if
9:     for each element $u$ of $D$ do
10:        if $u$ is already marked then
11:            $count++$
12:        else if a solution $Sol$ that assigns $u$ to $v$ exists then
13:            Mark all values of $Sol$ in the corresponding domains
14:            $count++$
15:        else
16:            Remove $u$ from the domain $D$
17:        end if
18:        if $count = \tau$ then
19:            Move to the next $v$ variable (Line 3)
20:        end if
21:    end for
22:    return $false$
23: end for
24: return $true$
```

GraMi uses a heuristic to speed up frequency checking. An improved version of Algorithm 3, which employs this heuristic, is shown in Algorithm 4 [1]. It considers each variable and searches for τ valid assignments. If they are found, it continues to the next variable. More specifically, after applying node and arc consistency, Algorithm 4 iterates over all elements of the domain of each variable and looks for a solution that assigns this element to the corresponding variable (line 12). If a solution is found, the *count* variable is incremented (line 14); otherwise, the element is removed from the domain (line 16). Removing an element from a domain may cause inconsistencies in other domains, so arc consistency must be enforced again (line 5). If for any domain *count* fails to reach the threshold τ, the subgraph is not frequent (line 22); otherwise, it is considered frequent (line 24) [1].

Algorithm 4 also marks all elements of all domains that are part of a solution (line 13). When evaluating an element that is already marked, we know that a valid assignment for this element has already been found and can simply increment *count* without searching for another solution (line 10). This allows the algorithm to avoid redundant searches [1].



(a) (b) (c)

Figure 2.4: Heuristic example.

For clarity, an example is given using the graphs in Figure 2.4. When considering a support threshold of two, the graph in Figure 2.4(a) as G and the subgraph in Figure 2.4(b) as S_1, the domains are as follows: $D_{v_1} = \{u_1, u_2\}, D_{v_2} = \{u_3, u_4\}, D_{v_3} = \{u_3, u_4\}$. When enforcing arc consistency, u_2 is dropped from v_1's domain because it has no connection to any of the other domains. With the new domain of v_1 being $\{u_1\}$, $false$ is returned on line 7 because this domain is smaller than the threshold τ.

Now consider the second subgraph S_2 from Figure 2.4(c), with initial domains $D_{v_1} = \{u_3, u_4\}$ and $D_{v_2} = \{u_5, u_6, u_7\}$. Enforcing node and arc consistency does not remove any values, so the domains remain the same. Then, each CSP variable is processed in turn. For v_1, take the first domain element u_3. A supporting assignment $(v_1 \mapsto u_3, v_2 \mapsto u_5)$ is found, so u_3 and u_5 are marked, and *count* is incremented to 1. Next, consider u_4. A second valid assignment $(v_1 \mapsto u_4, v_2 \mapsto u_6)$

is found, u_4 and u_6 are marked, and *count* is incremented to 2. Once *count* reaches the threshold $\tau = 2$, the algorithm moves on to v_2.

Considering the next variable v_2, the element u_5 is already marked, and therefore *count* is incremented to 1 immediately without looking for any solutions. The same happens for element u_6, bringing *count* to 2. Since v_2 has now reached the threshold $\tau = 2$ and there are no more variables left, the algorithm terminates and returns *true*. Note that no further matching checks were required for variable v_2, and u_7 was never examined, thanks to the markings of previously found matches and the early exit once each variable's count reached τ.

2.2.2 Push-Down Pruning

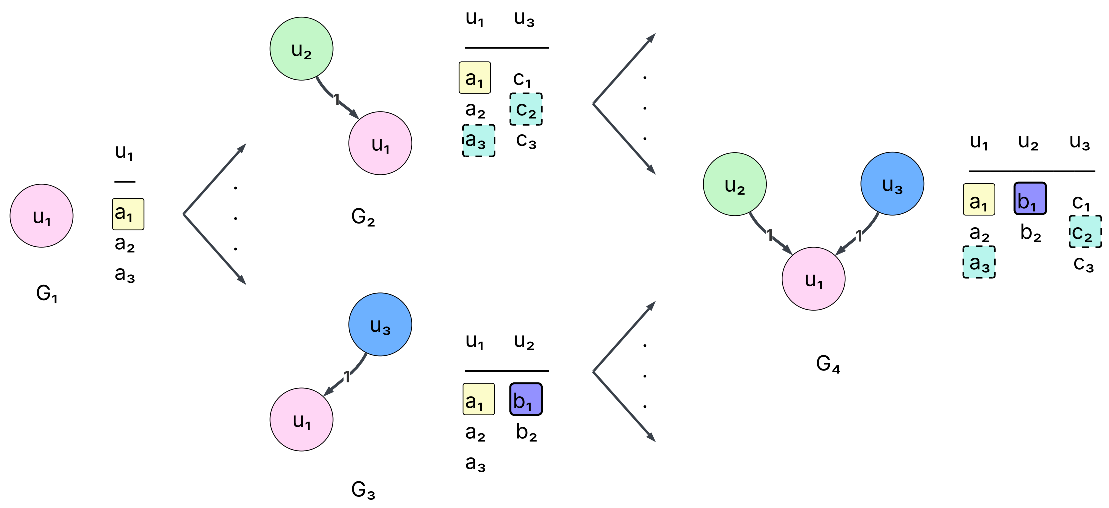
```
1: for all edge $e$ of $S$ do
2:     Let $S/^e$ be the graph after removing $e$ from $S$
3:     Remove values in the domains of $S$ corresponding to invalid assignments in $S/^e$
4: end for
```

Figure 2.5: Sketch of the push-down pruning procedure (from [1]).

Figure 2.5 [1] outlines the push-down pruning routine, which exploits the fact that any node mapping ruled out for a subgraph remains invalid for an extension of that subgraph. Before iterating over all elements of the domains and looking for solutions to the CSP for a subgraph S, GraMi iterates over each edge e in S, forms the smaller graph $S/^e$ by removing e from S, and looks up any assignment already known to be invalid for that graph. These assignments are then removed from the domain of S.

In practice, these invalid assignments are stored in a global lookup table during the search step. By pushing down these prunings from each possible parent graph into S, the domain sizes are reduced upfront, cutting down the elements that need to be tested in the subsequent support checking phase. Some subgraphs are even disqualified without the need for any searches [1].

Figure 2.6 shows an example of this process. Graph G_1 is extended to G_2 and G_3, among others. Graph G_4 is an extension of both G_2 and G_3. Next to each graph, the hypothetical domains of its variables are displayed. The invalid assignments of each step are marked. For example, the element a_1 is found to be invalid in G_1 and is marked as such. This marking is pushed down to all extensions, even over multiple iterations. Thus, graph G_4 inherits the invalid assignments of all previous graphs, which significantly reduces its domains.



(a)

Figure 2.6: Push-down pruning example.

2.2.3 Unique Labels

In the cases where a graph G has a single label per node and a graph S is a subgraph of G with unique labels and a tree-like structure, the frequency calculation can be simplified. Specifically, if all of S's directed edges are treated as undirected and S then represents a tree where each label only exists once, enforcing node and arc consistency is sufficient to calculate the frequency $s_G(S)$, without requiring any explicit subgraph isomorphism searches [1]. This simplification is possible because the uniqueness of labels in S ensures that each node in G can match at most one node in S, making the variable domains in the CSP disjoint. As a result, after enforcing consistency, the frequency $s_G(s)$ can be computed directly from the domain size, and no further search is required.

```
1: if $S$ and $G$ satisfy the unique labels optimization conditions then
2:     if any domain size is less than $\tau$ then
3:         return false
4:     end if
5:     return true
6: end if
```

Figure 2.7: Sketch of the unique labels procedure (from [1]).

This procedure, depicted in Figure 2.7 [1], is executed before iterating over the subgraph and searching for subgraph isomorphisms. It intersects all subgraphs that satisfy the criteria, eliminating the need for exhaustive search in these cases.

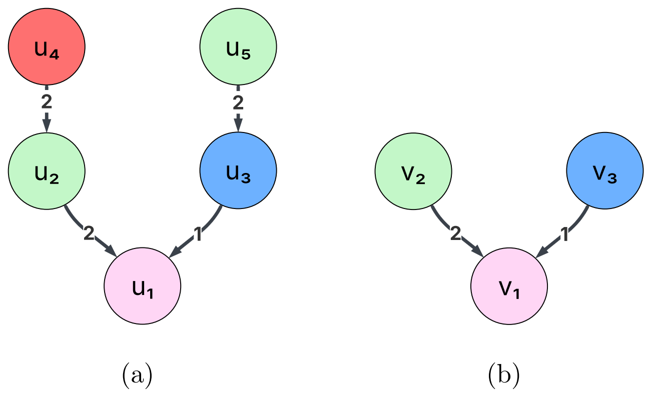


Figure 2.8: Unique labels example.

Refer to Figure 2.8 for an example of this process. The graph in Figure 2.8(a) is G, and the subgraph in Figure 2.8(b) is S. S completes all objectives of the unique labels optimization, because each label occurs only once, and replacing the directed edges with undirected ones would result in a tree. After enforcing node consistency, u_4 does not need to be considered anymore because its label does not appear in S. u_5 cannot survive arc consistency enforcement because no edge connects it to any element in the domain of v_1. The remaining elements in the domains now participate in a subgraph isomorphism between G and S.

2.2.4 Automorphism

An automorphism is an isomorphism from a graph to itself. In the context of subgraph mining, automorphisms arise when a subgraph contains symmetries, so if two nodes or substructures are indistinguishable. In this case, the valid assignments found for one node or substructure can be reused for the other, which avoids redundant searches [1].

```
1: **if** there exists an automorphism with a computed domain $D'$ **then**
2:     $D \leftarrow D'$ and continue to the next variable
3: **end if**
```

Figure 2.9: Sketch of the automorphism procedure (from [1]).

In practice, the procedure shown in Figure 2.9 [1] can be applied to each variable prior to the computation of the domain. If an automorphism maps a variable's domain D to that of another variable, the corresponding domain D' can be directly reused, eliminating the need for any further domain calculations.

Consider the graph in Figure 2.10(a) as G and the subgraph in Figure 2.10(b)

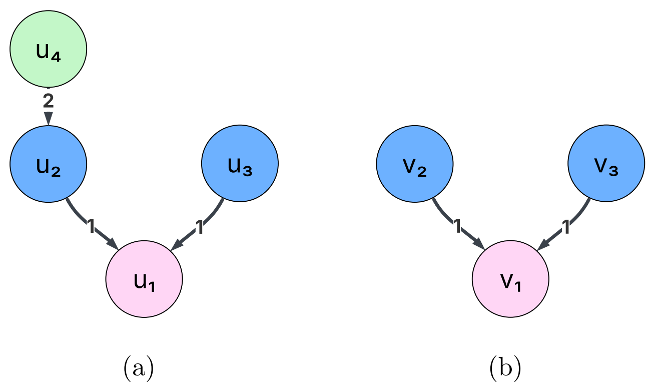


Figure 2.10: Automorphism example.

as S, with domains $D_{v_1} = \{u_1\}, D_{v_2} = \{u_2, u_3\}$, and $D_{v_3} = \{u_2, u_3\}$. Intuitively, due to the symmetries in S, the nodes v_2 and v_3 are interchangeable. As a result, any node in G that is a valid match for v_2 must also be a valid match for v_3, and vice versa. Therefore, after computing the domain of v_2, the same domain can be assigned to v_3 without recomputing it.

2.2.5 Lazy Search

To determine whether a subgraph S is frequent in a graph G, the frequency checking algorithm explores all possible partial assignments of nodes in S to nodes in G, searching for enough valid assignments to meet the frequency threshold τ. However, not all partial assignments are equally efficient to evaluate. Some are computationally expensive and may never lead to a valid embedding. Since only τ valid assignments have to be found for a subgraph to be considered frequent, it is not necessary to explore every possible assignment, especially those that are expensive to compute [1].

To optimize this process, GraMi imposes a time limit, and if a search exceeds this limit, the algorithm proceeds to the next partial assignment. This allows the algorithm to possibly reach the required number of valid embeddings through more efficient assignments. In many cases, a subgraph can be confirmed frequent without ever needing to resume the slow searches [1].

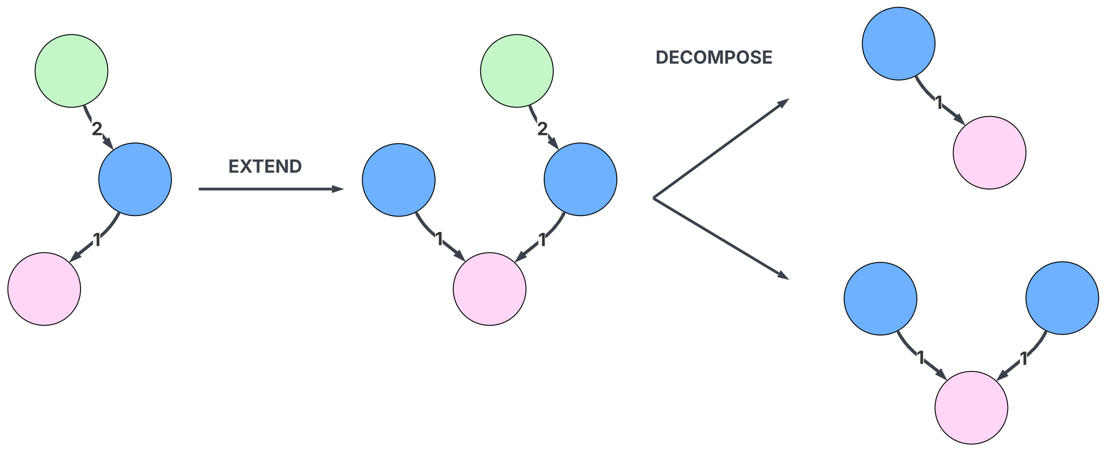
However, if the number of embeddings found without timeouts is not sufficient, i.e., fewer than τ, but could possibly reach τ with the timed-out searches, GraMi revisits these searches. Each suspended search resumes from its last saved state, this time without a time limit. In this way, GraMi avoids unnecessary computation when possible but guarantees a complete search of the entire search space if required [1].

2.2.6 Decomposition Pruning

```
1: Decompose $S$ into a set of graphs $Set$ containing the newly added edge
2: for all $s \in Set$ do
3:     Remove invalid assignments of $s$ from domains of $S$
4: end for
```

Figure 2.11: Sketch of the decomposition pruning procedure (from [1]).

Decomposition pruning, as seen in Figure 2.11 [1], is applied exclusively when resuming timed-out searches, with the goal of reducing the problem size by focusing on smaller components of the subgraph. Algorithm 2 extends a subgraph S with a new edge e. Decomposition pruning generates all connected components of S that include the edge e by iteratively removing one edge at a time from S and recording the connected components containing e. Identifying and eliminating invalid assignments on these smaller components instead of the full subgraph S can be done more efficiently. These invalid assignments can then be handed down to S and reduce its search space. Components of S that do not contain the edge e are excluded from decomposition pruning, as their invalid assignments are already handled by push-down pruning [1].



(a)

Figure 2.12: Decomposition pruning example.

This approach is illustrated in Figure 2.12. The graph on the left is extended by a new edge. Following this extension, the graph is decomposed into all connected components that include the newly added edge. Before checking the frequency of the extended graph, the invalid assignments for the connected components are identified and transferred to the extended graph. Finding invalid assignments for the components is faster because of their smaller size, and, based on the anti-monotone property, the invalid assignments can be transferred to the full subgraph.

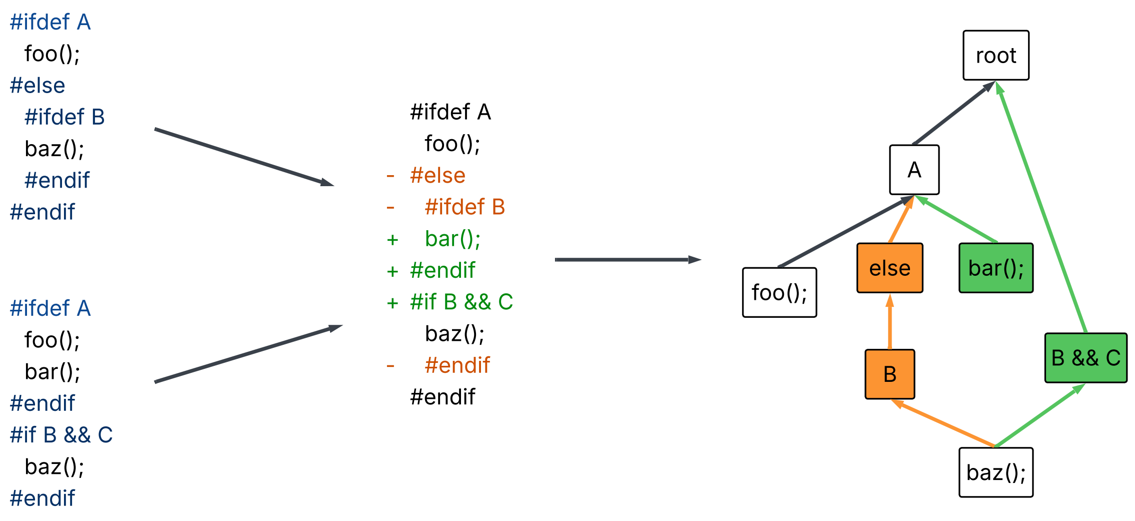
Chapter 3

Methodology

In this thesis, the underlying data consists of graphs that represent changes in software. These graphs were created using DiffDetectve [25]. This chapter briefly explains how the graphs are created and then presents some statistics and results from applying GraMi to the dataset.

3.1 Dataset Construction

DiffDetective is designed to operate on any generic differencing technique [25]. In the context of this thesis, the focus is on analyzing software code changes. To that end, DiffDetective is applied to the version history of a GitHub repository. For the purpose of this thesis, a demo repository was chosen; however, the entire process is fully reproducible and can be applied to any GitHub repository.



```
#ifdef A
  foo();
#else
  #ifdef B
  baz();
  #endif
#endif
```

```
#ifdef A
  foo();
  bar();
#endif
#if B && C
  baz();
#endif
```

```
   #ifdef A
     foo();
-  #else
-    #ifdef B
+    bar();
+  #endif
+  #if B && C
     baz();
-    #endif
   #endif
```

(a)

Figure 3.1: Data generation using DiffDetective (adapted from [25]).

In Figure 3.1, two versions of an example code are shown. A change file can be created from these two versions of the code, like the one seen in the center of Figure 3.1. The green, plus-marked code has been added, and the orange, minus-marked code has been deleted. This change file can be converted into a graph. Each node represents a piece of code, and each edge represents the relationship between them. The nodes and edges are color-coded in the same way as the code: green if added and orange if deleted during the code change. Note that most nodes have one edge originating from them. However, there are exceptions, such as the node containing $baz()$; at the bottom of the graph. The node itself is uncolored because $baz()$; existed before and after the change. Its relation to node B was removed, and a relation to node $B\&\&C$ was added. This is based on the fact that the position of the function $baz()$; changed inside the code [25].

For the purpose of FSM, having the full content of a line of code as node labels is not ideal. In FSM, it is important to generalize beyond exact character-by-character equality, allowing similar nodes to be considered equivalent. Therefore, nodes and edges are assigned more generalized labels that better support pattern comparison.

Node Labels	Meaning
ADD_*	Added
REM_*	Removed
NON_*	Unchanged
*_IF	If Statement
*_ELSE	Else Statement
*_ELIF	Elif Statement
*_ARTIFACT	Other Code

Table 3.1: Meaning of node labels.

Edge Labels	Meaning
b	before
a	after
ba	before & after

Table 3.2: Meaning of edge labels.

The labelling scheme used in this thesis is shown in Table 3.1 for nodes and Table 3.2 for edges. Edge labels encode whether the edge existed before, after, or in both versions of the code. This corresponds to the edge colors in Figure 3.1. Node labels, on the other hand, consist of two components. The first component indicates whether the node was added, removed, or stayed unchanged; the second categorizes the type of code the node represents.

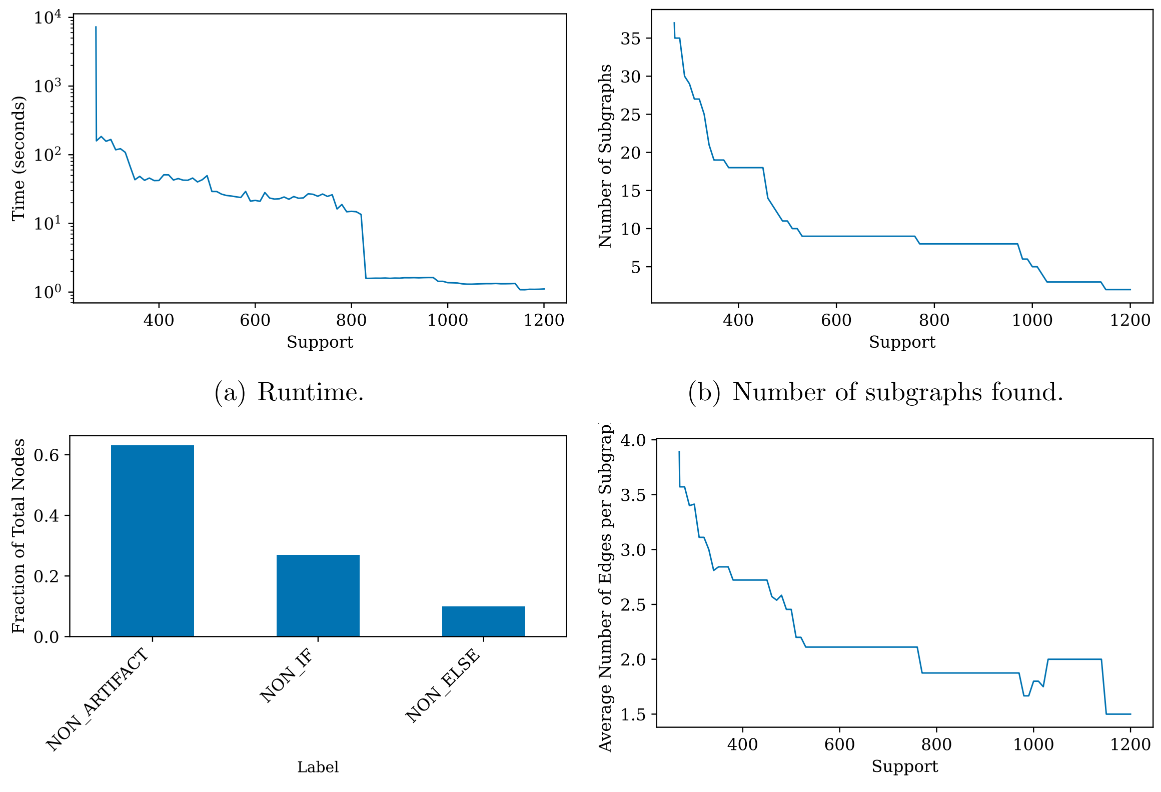
$\|G\|$	109
$\|V\|$	554.8
$\|E\|$	554.3

Table 3.3: Dataset Statistics

For this thesis, a dataset consisting of 109 graphs has been constructed. Key characteristics of the dataset are summarized in Table 3.3. Note that the average number of edges is only slightly smaller than the average number of nodes. This distribution is consistent with the graph construction process described earlier.

3.2 Baseline Frequent Subgraph Mining

GraMi was executed on the dataset described in Section 3.1 using a machine equipped with an Intel i7-1260P processor, 32 GB of RAM, and running Windows 11 Pro. GraMi, as described in Chapter 2, is designed to operate on a single graph as input. To accommodate this requirement, the 109 individual graphs in the dataset were merged into a single graph composed of 109 disjoint connected components, each corresponding to one of the original graphs.



(a) Runtime.

(b) Number of subgraphs found.

(c) Node label distribution (support 270).

(d) Average number of edges per subgraph.

Figure 3.2: Statistics of GraMi executions.

With a decrease of the frequency threshold, the number of subgraphs found by GraMi increases exponentially, as seen in Figure 3.2(b), and so does the computation time, shown in Figure 3.2(a). Due to this rapid rise in time consumption, a frequency threshold below 269 could not be processed within a reasonable timeframe of eight hours or less.

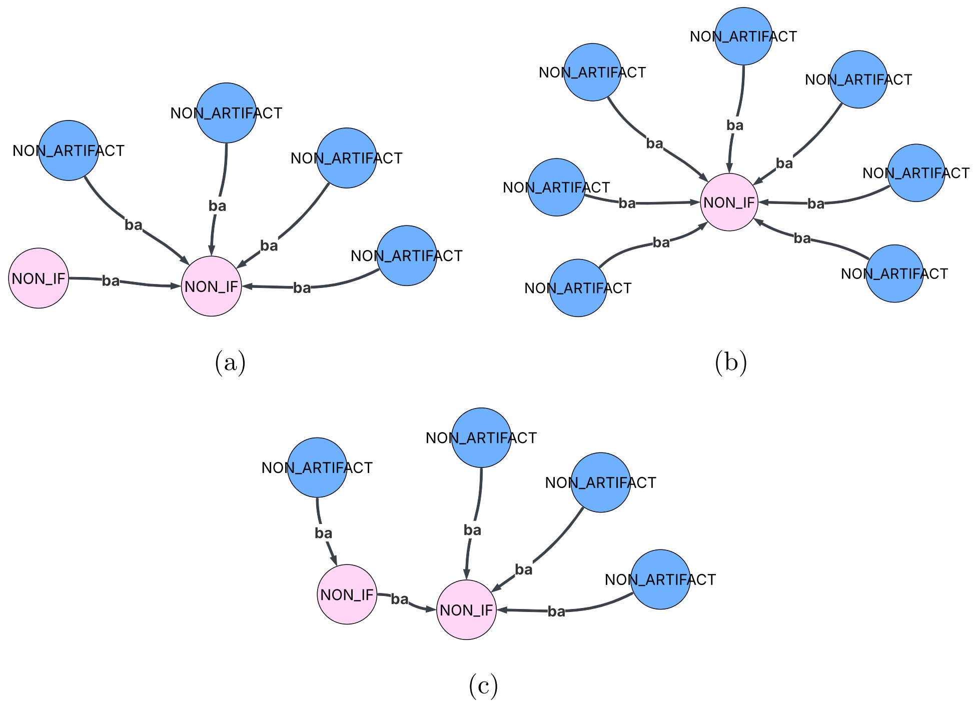


Figure 3.3: Frequent subgraphs found by GraMi.

As the frequency threshold decreases, the average number of edges per subgraph generally increases (Figure 3.2(d)). Intuitively, this makes sense because larger subgraphs likely appear less often and are therefore only found with a lower frequency threshold.

In Figure 3.3, three representative frequent subgraphs identified by GraMi have been hand-selected to represent the overall characteristics of the output. Most of the discovered subgraphs follow a structure similar to those shown in Figures 3.3(a) and 3.3(b), typically consisting of several $NON\_ARTIFACT$ nodes pointing to a NON_IF node. Occasionally, a NON_IF or NON_ELSE node appears alongside them. A few subgraphs exhibit slightly more structural depth, such as the subgraph shown in Figure 3.3(c), where a $NON\_ARTIFACT$ node is nested inside two layers of NON_IF nodes.

This pattern can also be seen in the distribution of labels shown in Figure 3.2(c). The vast majority of nodes in the output subgraphs are labeled $NON\_ARTIFACT$, with a smaller number of NON_IF and NON_ELSE nodes. Most NON_IF nodes appear at the base of the subgraphs.

Note that none of the present subgraphs contain labels without the NON_* prefix. Since the goal of this thesis is to uncover meaningful patterns in code changes, rather than unchanged structure, the current results do not fully meet this objective. This observation motivates the development of additional methods to better target subgraphs that reflect actual change.

Chapter 4

Refinements

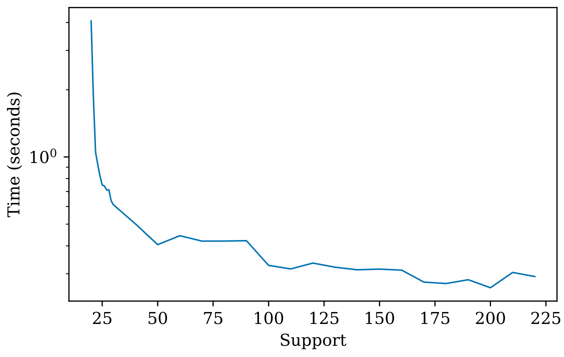
In Chapter 3, the output of GraMi on the constructed dataset was presented. However, the results were not fully satisfactory, as they mainly reflected unchanged code patterns rather than meaningful patterns of change. To address this, three improvement strategies are presented in this chapter.

The first improvement involves filtering the input graphs by removing all edges labeled *ba*, which represent unchanged relationships. As an alternative, the second improvement consists of a two-pass approach. After running GraMi once, the resulting frequent subgraphs are removed from the input dataset, and GraMi is executed again on the reduced graphs. The third approach proposes artificially decreasing the weight of unchanged nodes or edges by randomly relabeling them.

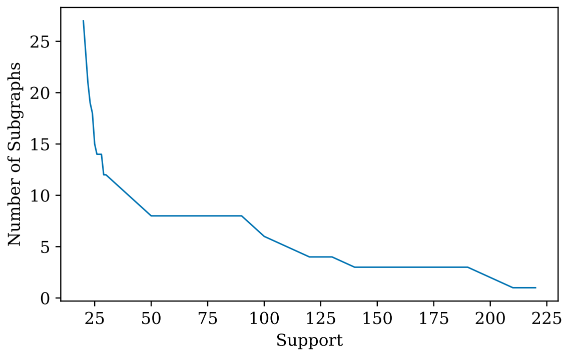
4.1 Filtering

As seen in Chapter 3, the results produced by GraMi contain exclusively node labels with the NON_* prefix. Additionally, they only contain edges holding the *ba* label. Since the objective of this thesis is to discover meaningful patterns in code changes, rather than unchanged structures, a filtering approach is proposed. Specifically, all edges holding the *ba* label are removed from the input graphs. This preprocessing of the data is intended to focus the FSM process on changes by eliminating unchanged relationships.

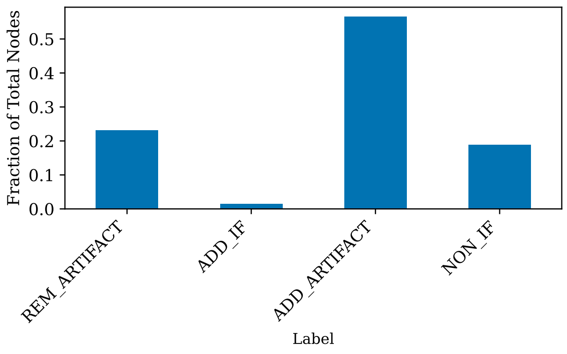
When running GraMi on the filtered dataset using the same hardware configuration as mentioned in Chapter 3, the frequency threshold can be set lower compared to the unfiltered case. This is due to improved performance at equivalent thresholds, as illustrated in Figure 4.1(a). The number of found subgraphs scales similarly, as shown in Figure 4.1(b). When filtering out edges labeled *ba*, the share of node labels carrying the NON_* prefix drops significantly, as illustrated in Figure 4.1(c). Nevertheless, some NON_IF labels remain, typically at the root of the subgraphs



(a) Runtime.



(b) Number of subgraphs found.

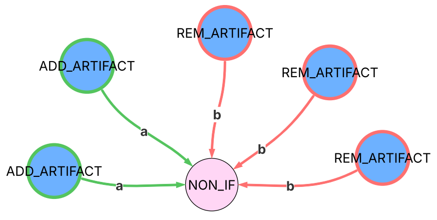


(c) Node label distribution (support 20).

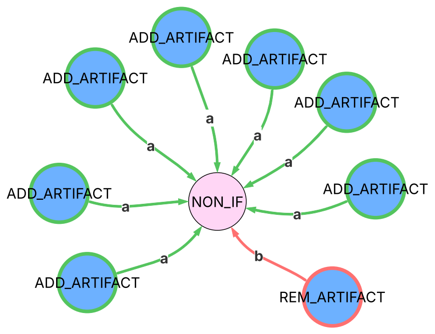
Figure 4.1: Statistics of filtered approach.

and without any outgoing edges.

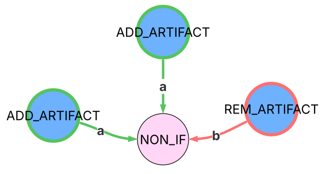
Three representative subgraphs from the output have been hand-selected and are shown in Figure 4.2. They illustrate the typical structure of the subgraphs produced after filtering. Each subgraph has a node with the NON_* prefix at the root and a varying number of $*\_ARTIFACT$ nodes.



(a)



(b)

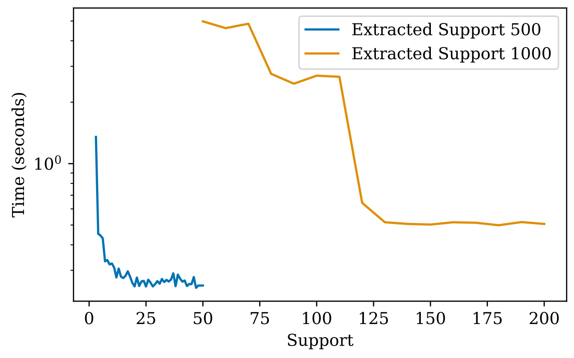


(c)

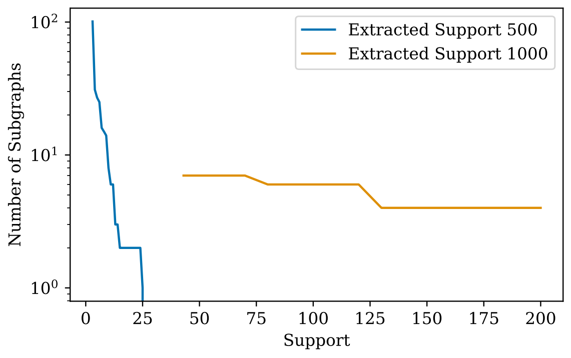
Figure 4.2: Frequent subgraphs found using filtering.

4.2 Extracting

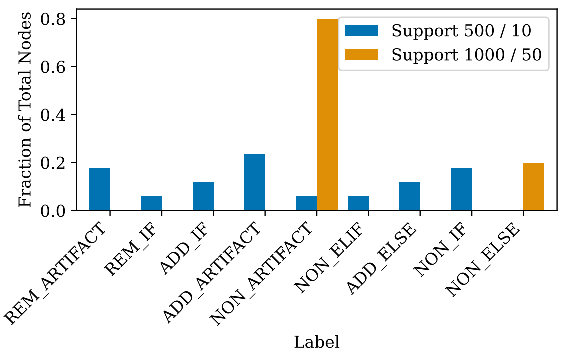
This section presents a strategy for increasing the share of nodes carrying the ADD_* or DEL_* prefix while retaining some structure given by *ba* edges. To achieve this, the output subgraphs that were generated by GraMi as presented in Chapter 3 are extracted from the input graphs. This forms a reduced set of input graphs. Then, GraMi is run on the reduced dataset in a second pass. This strategy aims to remove uninteresting subgraphs, such as those from the first pass, while still allowing *ba* edges to exist. These runs are characterized by two frequency thresholds, denoting the thresholds used for the first and second passes, respectively.



(a) Runtime.



(b) Number of subgraphs found.

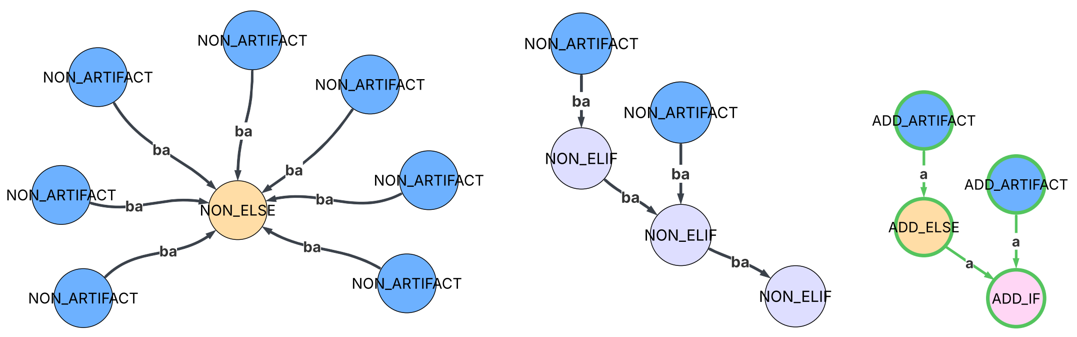


(c) Node label distribution.

Figure 4.3: Statistics of extracted approach.

With a first-pass frequency threshold of 500, eleven subgraphs were extracted from the source graphs, compared to only five subgraphs extracted when using a threshold of 1000. This greater reduction in the dataset allows for lower possible second-pass thresholds, as shown in Figure 4.3(a). The number of subgraphs found scales similarly, as seen in Figure 4.3(b).

Since the subgraphs from the first-pass contain many $NON\_ARTIFACT$ nodes, their removal results in a significant drop in the portion of such nodes after the second pass. This effect is especially pronounced when using a lower first-pass



(a) (b) (c)

Figure 4.4: Frequent subgraphs found using extraction.

threshold. In contrast, the higher threshold of 1000 removes fewer subgraphs and therefore leaves in a larger portion of $NON\_ARTIFACT$ nodes, as illustrated in Figure 4.3(c).

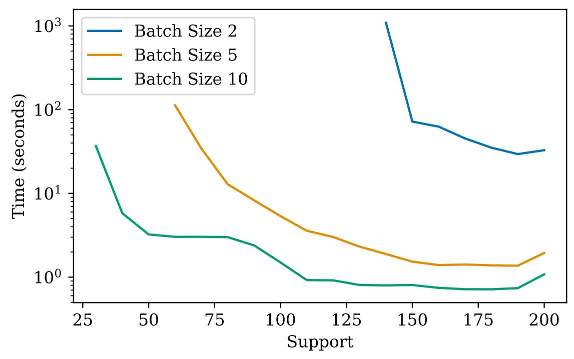
One of the subgraphs obtained using a first-pass threshold of 1000 and a second-pass threshold of 50 is shown in Figure 4.4(a). All other subgraphs produced with these settings are subgraphs of this graph, and therefore contain no edges with a label different from *ba*.

Subgraphs obtained with a first-pass threshold of 500 and a second-pass threshold of 5 are shown in Figures 4.4(b) and 4.4(c). These examples were hand-picked to represent the two dominant patterns observed in the output. The first pattern, exemplified by Subgraph 4.4(b), features a relatively complicated structure with multiple layers; however, all edges carry the label *ba*. The structure of the second pattern, as in Subgraph 4.4(c), exhibits a simpler structure but does contain edges with labels *b* and *a*. Some interesting patterns can be found, such as the added if-else block with two associated artifact nodes shown in Subgraph 4.4(c).

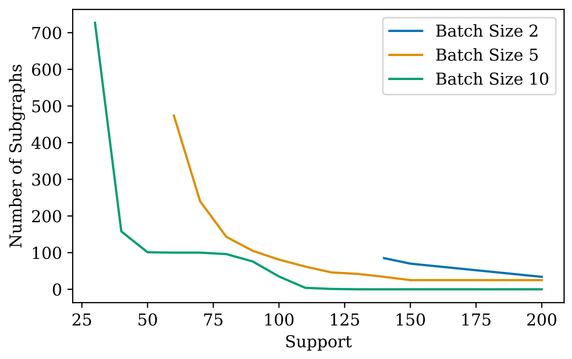
4.3 Random Relabeling

The third approach presented in this thesis does not involve the removal of any parts of the input graphs but aims to reduce the relative importance of edges labeled *ba* or nodes carrying the NON_* prefix. This is achieved by randomly relabeling these nodes or edges.

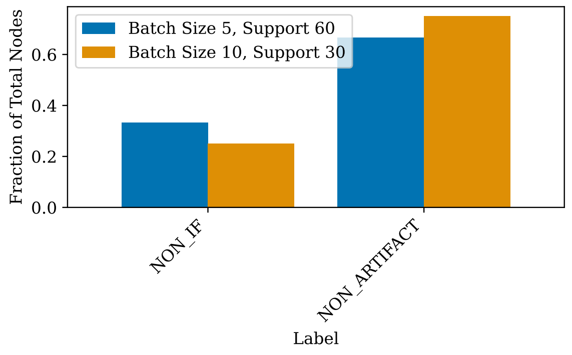
For edge relabeling and with an assumed batch size of two, each edge labeled *ba* is randomly renamed to either $ba1$ or $ba2$. As a result, two identical edges labeled *ba* in the source graph now only have a 50% chance of being considered equal during the FSM process. In contrast, edges with other labels remain unaffected and retain full comparability. Node relabeling follows a similar approach. Nodes labeled with the



(a) Runtime.



(b) Number of subgraphs found.



(c) Node label distribution.

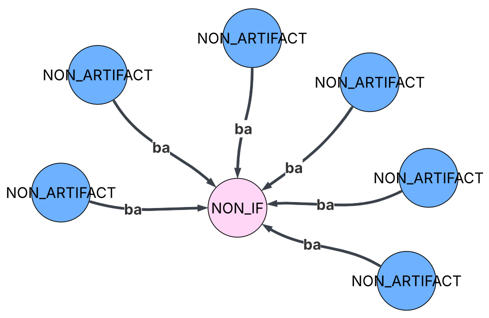
Figure 4.5: Statistics of random relabeling approach.

NON_* prefix are randomly relabeled into multiple variants, lowering the likelihood that such unchanged nodes align.

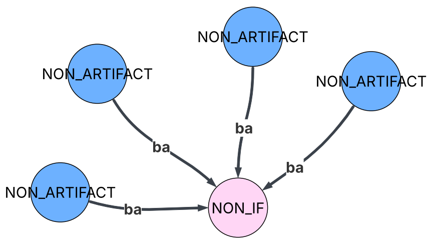
This process makes it less likely for ba edges or NON_* nodes to appear in the same way across graphs, which means they are less likely to form part of a frequent subgraph. As a result, their influence on the mining process is reduced.

As can be seen in Figure 4.5(a), the time consumption decreases with an increase in batch size. Similarly, the number of found subgraphs also decreases with an increase in batch size, as seen in Figure 4.5(b). Important to note is that for these figures, the nodes and edges have been relabeled to their original form without any appended numbers. The label distribution, illustrated in Figure 4.5(c), presents a domination of $NON\_ARTIFACT$ nodes and no nodes without the NON_* prefix.

The handpicked example subgraphs found using a batch size of eight and a frequency threshold of 30 are shown in Figure 4.6. They all exhibit similar patterns, typically consisting of a varying number of $NON\_ARTIFACT$ nodes pointing to a NON_* prefixed node at the root of the graph.



(a)



(b)

Figure 4.6: Frequent subgraphs found using random relabeling.

Chapter 5

Conclusions and Future Work

5.1 Conclusion

In this thesis, the goal was to find interesting patterns in software evolution graphs using an FSM algorithm, specifically GraMi, introduced in 2014 by Elseidy et al. [1]. GraMi approaches FSM by formulating it as a CSP and applies several optimization techniques, primarily while checking the frequency of a subgraph in a larger graph.

These techniques include the exploitation of special graph types and structures, which allows the avoidance of any searches in the case of a subgraph with unique labels and a tree-like structure. The detection of automorphisms in a subgraph candidate helps eliminate redundant searches by recognizing symmetries. Additionally, based on the anti-monotone property, invalid assignments found in a subgraph can be pushed down to all its extensions. To avoid inefficient computations, lazy search postpones expensive subgraph matches unless needed. If such searches become necessary, their complexity is reduced via decomposition pruning. Finally, a heuristic is used throughout to accelerate frequency checking.

The dataset used in this thesis consists of graphs generated by DiffDetective [25], based on the changes made within a GitHub repository. In total, 109 individual graphs were created and combined into a single graph with multiple connected components to meet GraMi's input requirements.

An initial attempt was made by running GraMi on the dataset. This revealed several challenges, namely the dominance of edges labelled *ba* and nodes carrying the NON_* prefix, as well as a lower bound on the frequency threshold of 269 due to time constraints. To address these issues, three pre- and post-processing methods were developed.

The first approach involved filtering out all edges labeled *ba*. The second contained two passes of the GraMi algorithm. Subgraphs found in the first pass were extracted from the input graphs before performing a second pass. The final method randomly renamed NON_* prefixed nodes or *ba* edges to lower the likelihood of such elements matching during the FSM process.

When running GraMi on the unmodified dataset, the frequency threshold could not be lowered below 269 within a reasonable timeframe. This yielded no subgraphs with an edge carrying a label other than *ba*.

When filtering the source graphs to exclude all edges labeled *ba*, a much lower frequency threshold could be set. The subgraphs that were found differ from those produced by running GraMi on the unfiltered dataset, but they do not exhibit significantly more variety, as they are now dominated by artifact nodes. This is suspected to be either due to a substantial loss of structural information caused by removing all *ba* edges or because the frequency threshold remains too high. While the original issue of the domination of *ba* edges was resolved, the approach did not lead to the discovery of more interesting patterns.

As a second approach, extracting the output subgraphs from the input graphs and performing a second pass allowed some *ba* edges to remain, preserving more of the original structure. This method also allowed for a lower frequency threshold compared to running GraMi on the unmodified dataset. Because of this lower threshold, some output subgraphs revealed more structure, such as if-else blocks, which could be considered interesting patterns. However, the reveal of these structures required setting the second pass threshold at five, which raises concerns about whether they can be considered frequent.

Finally, when relabeling the nodes or edges, lower thresholds could be achieved compared to the unmodified dataset, though not as low as those achieved by extracting subgraphs. This also depends on the batch size, so the number of distinct names a label can be assigned. Higher batch sizes result in lower possible thresholds. While the subgraphs discovered using this technique are potentially more interesting, a significant number of *ba* edges remain, even with a batch size as large as ten.

Overall, while this thesis did not uncover many highly interesting patterns, it demonstrates that FSM is a valid method for analyzing software evolution graphs. The experiments show that the quality of the results can be significantly improved by various pre- and post-processing steps. Some methods are constrained by performance, while others produced results dominated by artifact nodes, limiting their usefulness. A few interesting patterns did emerge; however, they required a very low frequency threshold of five, suggesting that the data may not contain many interesting patterns with a high frequency.

5.2 Future Work

Future research could subdivide the broad $*\_ARTIFACT$ node label category into more specific types, which might lead to more interesting patterns. Additionally, techniques that allow for low frequency thresholds, such as the extraction method, hold promise but require closer inspection of the extracted subgraphs to evaluate their practical relevance. Generally, applying these techniques to larger or differently structured repositories may yield more valuable insights. Finally, leveraging a more performant FSM algorithm with the ability to handle this type of data could reveal more interesting patterns.

Appendix A

Optimized Frequency Checking

Algorithm 5 IsFrequent(S, G, τ) (Part 1, from [1])

```
1: Consider the subgraph $S$ to graph $G$ CSP and apply node and arc consistency
2: for all edge $e$ of $S$ do
3:    Let $S/^e$ be the graph after removing $e$ from $S$
4:    Remove values in the domains of $S$ corresponding to invalid assignments in $S/^e$
5: end for
6: if $S$ and $G$ satisfy the unique labels optimization conditions then
7:    if any domain size is less than $\tau$ then
8:       return false
9:    end if
10:   return true
11: end if
12: Compute the automorphisms of $S$
13: for all variable $x$ and its domain $D$ do
14:    $count \leftarrow 0$, $timedoutSearch \leftarrow \emptyset$
15:    if an automorphism has a computed domain $D'$ then
16:       $D \leftarrow D'$ and continue to next $x$
17:    end if
18:    Apply arc consistency
19:    if any domain size is less than $\tau$ then
20:       return false
21:    end if
```

Algorithm 6 IsFrequent (Part 2, from [1])

```
22:    for all element $u \in D$ do
23:       if $u$ is marked then
24:          $count \leftarrow count + 1$
25:       else
26:          Search for a solution assigning $u$ to $x$ with a time threshold
27:          if search timeouts then
28:             Save search state to $timedoutSearch$
29:          else if solution $Sol$ is found then
30:             Mark all values of $Sol$ in their respective domains
31:             $count \leftarrow count + 1$
32:          else
33:             Remove $u$ from $D$ and add it to invalid assignments in $S$
34:          end if
35:          if $count = \tau$ then
36:             move to next variable
37:          end if
38:       end if
39:    end for
40:    if $|timedoutSearch| + count \geq \tau$ then
41:       Decompose $S$ into a set of graphs $Set$ containing the newly added edge
42:       for all $s \in Set$ do
43:          Remove invalid assignments of $s$ from domains of $S$
44:       end for
45:       for all saved state $t \in timedoutSearch$ do
46:          Resume search from $t$
47:          if solution $Sol$ is found then
48:             Mark values of $Sol$ in corresponding domains
49:             $count \leftarrow count + 1$
50:          else
51:             Remove $u$ from $D$ and add to invalid assignments in $S$
52:          end if
53:          if $count = \tau$ then
54:             Move to the next variable
55:          end if
56:       end for
57:    end if
58:    return false
59: end for
60: return true
```

Bibliography

[1] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. GRAMI: frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, 2014.

[2] Peter J. Denning, Douglas Comer, David Gries, Michael C. Mulder, Allen B. Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline: preliminary report of the ACM task force on the core of computer science. In Herbert L. Dershem, editor, *Proceedings of the 19th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 1988, Atlanta, Georgia, USA, February 25-26, 1988*, page 41. ACM, 1988.

[3] Lasse Holmström and Petri Koistinen. Pattern recognition. *WIREs Computational Statistics*, 2(4):404–413, 2010.

[4] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

[6] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[7] Tak-Chung Fu. A review on time series data mining. *Eng. Appl. Artif. Intell.*, 24(1):164–181, 2011.

[8] Kaizhong Zhang and Dennis E. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[9] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *Int. J. Pattern Recognit. Artif. Intell.*, 18(3):265–298, 2004.

[10] Mark E. J. Newman. The structure and function of complex networks. *SIAM Rev.*, 45(2):167–256, 2003.

[11] Young-Rae Cho and Aidong Zhang. Predicting protein function by frequent functional association pattern mining in protein interaction networks. *IEEE Trans. Inf. Technol. Biomed.*, 14(1):30–36, 2010.

[12] Ammar Haydari and Yasin Yilmaz. Deep reinforcement learning for intelligent transportation systems: A survey. *IEEE Trans. Intell. Transp. Syst.*, 23(1):11–32, 2022.

[13] Mario Janke and Patrick Mäder. Graph based mining of code change patterns from version control commits. *IEEE Trans. Software Eng.*, 48(3):848–863, 2022.

[14] Santo Fortunato. Community detection in graphs. *CoRR*, abs/0906.0612, 2009.

[15] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. A survey on graph kernels. *Appl. Netw. Sci.*, 5(1):6, 2020.

[16] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[17] Büsra Güvenoglu and Belgin Ergenç Bostanoglu. A qualitative survey on frequent subgraph mining. *Open Comput. Sci.*, 8(1):194–209, 2018.

[18] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 368–377. IEEE Computer Society, 1998.

[19] Frances E. Allen. Control flow analysis. In Robert S. Northcote, editor, *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, pages 1–19. ACM, 1970.

[20] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Trans. Software Eng.*, 5(3):216–226, 1979.

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[22] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973.

[23] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In Patrick D. McDaniel, editor, *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[24] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[25] Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Timo Kehrer, and Thomas Thüm. Variability-aware differencing with diffdetective. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 632–636. ACM, 2024.

[26] Rahul Amlekar, Andrés Felipe Rincón Gamboa, Keheliya Gallaba, and Shane McIntosh. Do software engineers use autocompletion features differently than other developers? In Andy Zaidman, Yasutaka Kamei, and Emily Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 86–89. ACM, 2018.

[27] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 233–243. IEEE Computer Society, 2012.

[28] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson R. Murphy-Hill. From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 211–221. IEEE Computer Society, 2016.

[29] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*, pages 721–724. IEEE Computer Society, 2002.

[30] Tanay Kumar Saha and Mohammad Al Hasan. Fs^3: A sampling based method for top-k frequent subgraph mining. *Stat. Anal. Data Min.*, 8(4):245–261, 2015.

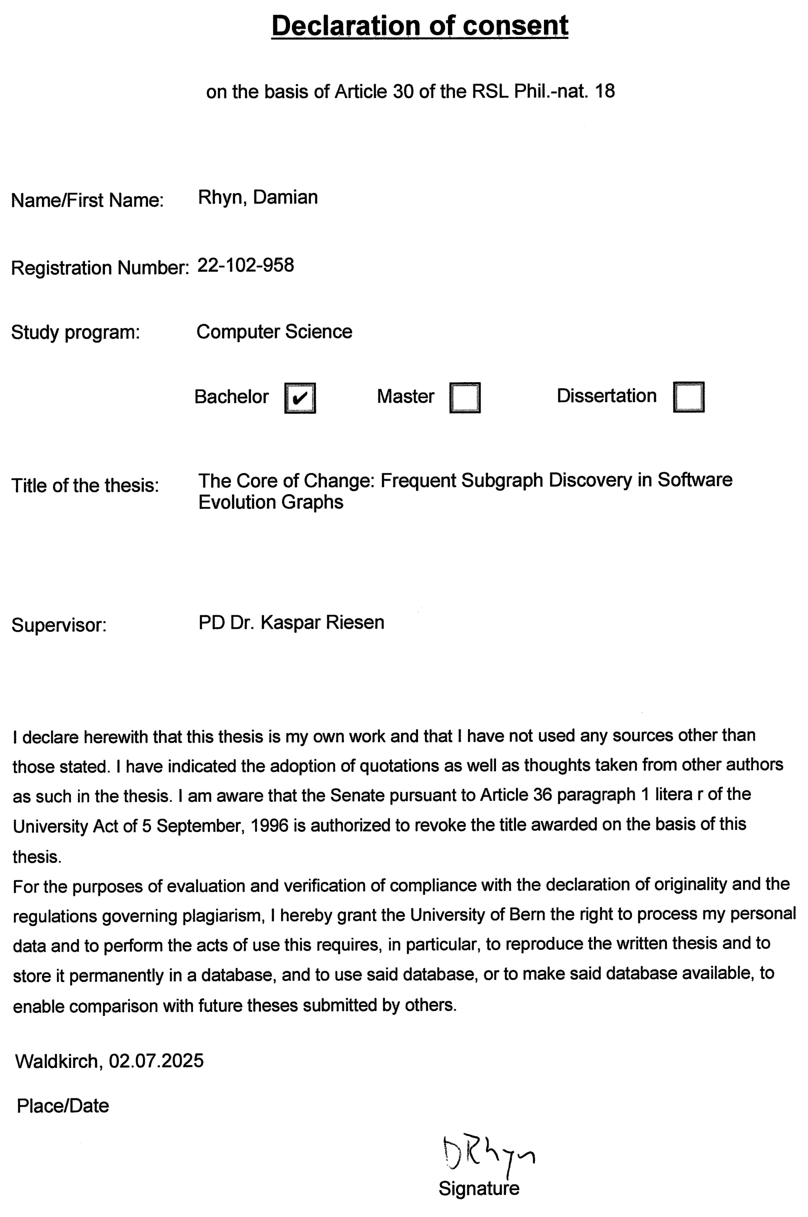
[31] Lyuheng Yuan, Da Yan, Wenwen Qu, Saugat Adhikari, Jalal Khalil, Cheng Long, and Xiaoling Wang. T-FSM: A task-based system for massively parallel frequent subgraph pattern mining from a big graph. *Proc. ACM Manag. Data*, 1(1):74:1–74:26, 2023.

[32] Rex Ying, Tianyu Fu, Andrew Wang, Jiaxuan You, Yu Wang, and Jure Leskovec. Representation learning for frequent subgraph mining. *CoRR*, abs/2402.14367, 2024.

[33] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In Takashi Washio, Einoshin Suzuki, Kai Ming Ting, and Akihiro Inokuchi, editors, *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference, PAKDD 2008, Osaka, Japan, May 20-23, 2008 Proceedings*, volume 5012 of *Lecture Notes in Computer Science*, pages 858–863. Springer, 2008.

Appendix B

Declaration of Consent



Declaration of consent

on the basis of Article 30 of the RSL Phil.-nat. 18

Name/First Name:	Rhyn, Damian
Registration Number:	22-102-958
Study program:	Computer Science
	Bachelor [✓] Master ☐ Dissertation ☐
Title of the thesis:	The Core of Change: Frequent Subgraph Discovery in Software Evolution Graphs
Supervisor:	PD Dr. Kaspar Riesen

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of 5 September, 1996 is authorized to revoke the title awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with future theses submitted by others.

Waldkirch, 02.07.2025

Place/Date

[illegible]

Signature