Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Graph Pattern Mining In Code

Bachelor's Thesis

Jakob Flunger

jflunger@student-ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Peter Belcák
Prof. Dr. Roger Wattenhofer

May 24, 2022

Abstract

Our goal is to find common patterns in abstract syntax trees with added semantic information. We use popular python projects on GitHub as our dataset.

We apply classical methods for graph pattern mining to this task. The key insight is that most traditional methods run out of either time or memory when encountering wide fanning graphs in the dataset. As a solution to this problem, we propose two algorithms, BeSick and BeSwole , to circumvent the issues such graphs create.

While the patterns we find aren't significantly more interesting than previous work, our approach remains expandable and can still be useful for statistical analysis.

Contents

Chapter 1

Problem Description

Given a set of graphs we want to find subgraphs, also called patterns, that occur in many of those graphs.

1.1 Frequent Subgraph Mining

The frequent subgraph mining problem in its general case consists of a set of graphs known as the graph database, and a $min\_support$ value.

Definition 1.1 (Support)**.** For some pattern graph P and graph database $\mathcal{D} = G_1, ..., G_n$: we define

$$support(P) = \frac{1}{n}\sum_{i=1}^{n} I_i \quad \text{where} \quad I_i = \begin{cases} 1 & \text{if } P \preccurlyeq G_i \\ 0 & \text{else} \end{cases} \tag{1.1}$$

Definition 1.2 (Frequent Pattern)**.** A pattern graph P is considered frequent iff

$$support(P) \geq min\_support \tag{1.2}$$

Lemma 1.3 (Anti-monotonicity of Support)**.** *Given two pattern graphs P and P' such that $P \preccurlyeq P'$ then*

$$support(P) \geq support(P') \;\Rightarrow\; P \textit{ is not frequent} \;\rightarrow\; P' \textit{ is not frequent} \tag{1.3}$$

and in particular

Lemma 1.4 (Apriori Property)**.**

$$P' \textit{ is frequent} \iff \forall P \preccurlyeq P' : P \textit{ is frequent} \tag{1.4}$$

This Apriori property allows for exploration of the frequent pattern space without exploring the total pattern space. This can be done in two ways:
Apriori algorithms explore the space like a breadth first search. They compute

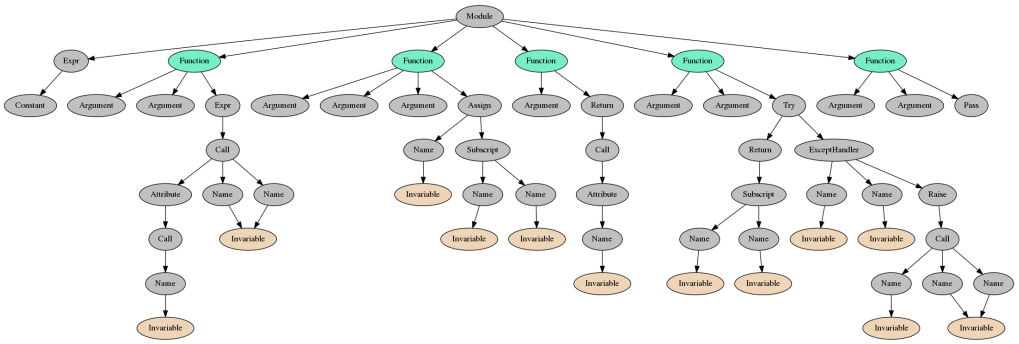

Figure 1.1: One of the smallest SASTs in our dataset.

all patterns of size n first, then compute patterns of size $n+1$ by trying to merge smaller, already mined patterns. Since this comes at a considerable memory overhead it is rarely used.
Pattern growth algorithms explore the space like a depth first search. Given a set of starting patterns they attempt to add a node or edge to the pattern and check if the extended pattern is still frequent. If it is, then the algorithm starts a recursion on the new extended pattern.
To the best of my knowledge, all classical graph pattern mining algorithms fall into one of these categories. Each algorithm has its own enumeration strategy, used to avoid mining duplicate patterns, and its own method of frequency computation.

1.2 Semantized Abstract Syntax Trees

Python source code can be represented by its abstract syntax tree. While this representation captures program structure well it looses a lot of semantic information, such as the same variable being used in multiple places. We aim to combat this by adding back some of the lost semantic information in the form of additional nodes and edges. This semantized AST is called a SAST.
Each node in a SAST has a label as can be seen in Figure 1.1.

The particular details of SAST construction are not important to the matters of this thesis, and may leave room for future experimentation. However the algorithms presented here assume that no directed cycles are added to the graph. More formally, we assume that all SASTs are directed acyclic graphs.

1.3 Subtrees and Subgraphs

Given tree $S = (V_S, E_S)$ and tree $T = (V_T, E_T)$ In this thesis we will use $S \preccurlyeq T$ to denote that S is an induced subtree of T, according to the definitions given by [1].

Definition 1.5 (Label Function)**.** For a graph $G = (V, E)$ the label of each node is given as a label function $l : V \to L$ where L is the set of labels.

Definition 1.6 (Isomorphic Subgraph)**.** Given graph $S = (V_S, E_S)$ and graph $T = (V_T, E_T)$, we say S is an isomorphic subgraph of T iff there exists a one-to-one (a.k.a. injective) mapping $\phi : V_S \to V_T$ such that $(x, y) \in E_S \iff (\phi(x), \phi(y)) \in E_T$

Definition 1.7 (Induced Subgraph)**.** Given graph $S = (V_S, E_S)$ and graph $T = (V_T, E_T)$, we say S is an induced subgraph of T iff S is an isomorphic subgraph of T and ϕ preservers labels, i.e., $l(x) = l(\phi(x)) \ \ \forall x \in V_S$

1.3.1 Induced Rooted Subtrees

ASTs and SASTs have a fixed root node. We can use this to simplify the problem further. In the rest of this thesis when talking about subtrees we will refer to the case of induced rooted subtrees. The same definition as for induced subgraphs, but the graphs in question are rooted trees.

1.3.2 Embedded Rooted Subtrees

Embedded subtrees are more general than induced subtrees. Every induced subtree is an embedded subtree, but the reverse is not guaranteed. The isomorphic condition is relaxed to $(x, y) \in E_S \Rightarrow$ there exists a path from $\phi(x)$ to $\phi(y)$ in T This thesis originally intended to mine embedded subgraphs, but due to vastly increased computational effort, the project was scaled back to induced subgraphs.

1.4 Complexities

1.4.1 Subgraph Isomorphism

Subgraph isomorphism is the decision problem of $T \preccurlyeq T'$.
There is neither a proof showing that it is in P, nor any proof that it is in NP, making it a candidate for the class NP-intermediate. Most importantly, there are no known efficient (polynomial) algorithms to solve this problem for general graphs.

1.4.2 Subtree Isomorphism

For normal trees the subtree isomorphism problem was solved by [2] in $\mathbf{O}(n^{\frac{5}{2}})$. We will show a different algorithm, compatible with labelled trees and designed to work for finding trees in DAGs. The algorithm proposed will be similar to [3] and run in polynomial time with the caveat of producing some false positives.

1.4.3 DAG Subgraph Reduction

We will show that the subgraph isomorphism problem, restricted to just DAGs, is still just as difficult as general undirected graph isomorphism. We will formalize the reduction proposed by [4].

Theorem 1.8 (DAG generality)**.** *Undirected subgraph isomorphism can be reduced to DAG subgraph isomorphism.*

Proof. Given undirected graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ we want to compute whether $G \preccurlyeq H$.
We construct $G' = (V_G \cup E_G, E'_G)$ where $E'_G = \{(n, e) \ \ \forall (n, e) : e \in E_G$ and n is an endpoint of edge $e\}$. We construct H' analogously. Note that both G' and H' are DAGs.

$$
\begin{aligned}
& G' \text{ is a subtree of } H' && (1.5)\\
\iff & \exists \text{ injection } \phi : V'_G \to V'_H \text{ that preserves edges} && (1.6)\\
\iff & \forall (x, y) : (x, y) \in E'_G \leftrightarrow (\phi(x), \phi(y)) \in E'_H && (1.7)\\
\iff & \forall (u, v, e) : u, v, e \in V'_G \ \wedge \ (u, e), (v, e) \in E'_G && (1.8)\\
& \quad \leftrightarrow \phi(u), \phi(v), \phi(e) \in V'_H \ \wedge \ (\phi(u), \phi(e)), (\phi(v), \phi(e)) \in E'_H && (1.9)\\
\iff & \forall e : e = (x, y) \in E_G && (1.10)\\
& \quad \leftrightarrow \phi(u), \phi(v), \phi(e) \in V'_H \ \wedge \ (\phi(u), \phi(e)), (\phi(v), \phi(e)) \in E'_H && (1.11)\\
\iff & \forall (x, y) : (x, y) \in E_G \leftrightarrow (\phi(x), \phi(y)) \in E_H && (1.12)\\
\iff & \exists \text{ injection } \phi : V_G \to V_H \text{ that preserves edges} && (1.13)\\
\iff & G \text{ is a subtree of } H && (1.14)
\end{aligned}
$$

□

With this knowledge we are reminded that the problem we are trying to solve is very complex and computationally expensive to solve in the general case. We have to use the close relation of SASTs to trees to make mining on a large scale feasible.

1.5 The Dataset

For authentic mining "in the wild" data we will use python source code freely available on GitHub. The final dataset contained 9943 python files, which were split into 96335 function-level graphs.

1.5.1 Crawler

A crawler was used to download the 100 most starred python projects on GitHub. Due to rate limiting this process requires an API key to scrape the entire dataset in one operation. Any file with a .py extension was then extracted.

1.5.2 Preprocessing

The python files are converted into SASTs by the sast module. All these SASTs consist of a root node with type "Module" and typically multiple nodes of type "Function" below that. To decrease the computational cost we decided to split these SASTs up into function level graphs. For each node n of type "Function" we copy the graph rooted at n.
All labels are then converted to integers to speed up computation.

Chapter 2

Enumeration Strategy

The enumeration strategy used by our algorithm will be similar to Gaston [5] by mining for rooted induced subtrees first, then mining for cycle closing edges. For the first step we use an enumeration strategy similar to SLEUTH [1], for the second step we use a variation of Gastons cycle closing edge enumeration strategy.

2.1 Initial Approach

As mentioned above, the initial approach of this thesis was to be very simple.

1. Mine embedded frequent subtrees in the AST using SLEUTH [1].
2. Use Gaston [5] to mine in the SAST, with the previously found embedded subtrees as the starting patterns.

However even when limiting the search to induced subtrees for step 1, the computational complexity is far too great for the search to ever terminate on an arbitrary dataset.
This is because both of the algorithms mentioned above suffer from the combinatorial explosion of occurrence lists.

2.1.1 Sleuth

SLEUTH was designed to efficiently mine embedded subtrees. For its enumeration strategy it uses equivalence class based extensions:
An equivalence class is given by a pattern graph P and a list of frequent extensions (x, i), each indicating that the graph P_x^i, which is P but with an extra node with label x attached to node i in P, is frequent. (Nodes are identified by their DFS number)

SLEUTH will then construct the equivalence class for each P_x^i by checking every other extension (y, j) in the equivalence class of P if:

1. Node j must be on the rightmost path of P_x^i
2. The resulting pattern graph must be canonical

If both conditions are met, then SLEUTH will attempts to add both the cousin extension (y, j), as well as a child extension $(y, k{+}1)$ (where k is the last node in P_x^i), to the equivalence class of P_x^i.
Before adding an extension to an equivalence class, SLEUTH checks if the resulting pattern is frequent. It does this with an operation called the scope list join.

Given valid initial equivalence classes, this strategy efficiently and non-redundantly enumerates all frequent embedded subtrees.

In SLEUTH each extension (x, i) comes with a scope list. Scope lists are a fancy form of occurrence lists, optimized for frequency computation with embedded subtrees. What is important for our purposes is that there is one entry for every occurrence of P_x^i in the graph database.
In particular for given graph database $\mathcal{D}$ and pattern $P_x^i = (V_P, E_P)$ the scope list for (x, i) must track every single edge- and label-preserving injection $\phi : V_P \rightarrow V_G \quad \forall G{=}(V_G, E_G) \in \mathcal{D}$.

2.1.2 Gaston

Gaston is a relatively straightforward pattern growth algorithm designed for general graphs.
Patterns in Gaston can be either paths, free trees or general graphs. It utilizes the fact that there is a definitive hierarchy of these classes.

- Paths are first extended into paths, then free trees, then general graphs
- Free trees are first extended into free trees, then general graphs
- General graphs are extended into general graphs

When extending general graphs, Gaston only adds cycle closing edges. This ensures that any general graph pattern must grow from a spanning tree, and free trees are only allowed to grow from their longest backbone path.
However this enumeration strategy may still find patterns more than once. So Gaston must check against a database of previously found patterns to avoid duplicate computations. The originally published version of Gaston uses the external Nauty [6] algorithm to check if a new pattern is isomorphic to any previously

found pattern.

A pattern in Gaston consists of the pattern graph $P = (V_P, E_P)$, an indicator of the pattern type (path/free tree/general graph), and an occurrence list. This occurrence list contains every injection $\phi : V_P \to V_G \ \ \forall G \in \mathcal{D}$ that preserves edges and labels, where $\mathcal{D}$ is the graph database.
preserving edges: $(x, y) \in E_P \iff (\phi(x), \phi(y)) \in E_G$
preserving labels: $l(x) = l(\phi(x)) \ \ \forall x \in V_P$
This is why Gaston too suffers from the combinatorial explosion of occurrence lists.

2.2 Canonical Rightmost Path Extension

For our enumeration strategy we will use the rightmost path extensions as defined by [1].

Definition 2.1 (Tree String Encoding)**.** To obtain the string encoding $\mathcal{T}(T)$ of tree T, traverse the nodes in T by depth-first preorder. For each node v append the label $l(v)$ to the string encoding, and add a unique up character \$ whenever the DFS backtracks. When sorting, \$ is sorted higher than any other label.

Definition 2.2 (Graph Isomorphism)**.** Graphs $G = (V_G, E_G)$ and $H = (V_H, E_H)$ are isomorphic, denoted as $G \equiv H$ iff there exists a bijection $\phi : V_G \to V_H$ such that $(x, y) \in E_G \iff (\phi(x), \phi(y)) \in E_H$ and ϕ preservers labels:
$l(x) = l(\phi(x)) \ \ \forall x$

Definition 2.3 (Canonical Tree)**.** An ordered tree T is canonical if its string encoding is the lexicographically smallest:

$$\mathcal{T}(T) \leq \mathcal{T}(T') \ \ \forall T' \equiv T \tag{2.1}$$

Note that $\mathcal{T}(T) = \mathcal{T}(T')$ implies $T = T'$, thus the canonical tree is uniquely defined among its isomorphism group.

To check if a tree is canonical we need to make sure that for all vertices $v \in T$, $\mathcal{T}(T_{c_i}) \leq \mathcal{T}(T_{c_{i+1}}) \ \ \forall i \in [1, k-1]$, where $c_1, c_2, ..., c_k$ are the ordered children of v and T_{c_i} is the subtree rooted at node c_i. For the proof of this property see [1].

Definition 2.4 (Rightmost Path Extension)**.** Adding a node of label x to tree T, attached to node i, is a rightmost path extension iff the attachment point i is on the rightmost path of in T.

It is sometimes called a prefix extension because if we look at trees only by their string encodings, it is equivalent to adding zero or more up symbols \$, followed by x to the prefix string $\mathcal{T}(T)$. Note that tailing \$ symbols are omitted in $\mathcal{T}(T)$.

As shown in [1], using this enumeration strategy on the starting set of single node patterns with unique labels, enumerates all unordered trees non-redundantly. While this is less sophisticated than the equivalence class extensions used by SLEUTH, we use this as the tree enumeration strategy for our algorithm.

2.3 Cycle Closing Edge Mining

2.3.1 With Occurrence List

Gaston finds potential cycle closing edges of a pattern graph $P = (V_P, E_P)$ by enumerating all occurrences of P in all graphs $G \in \mathcal{D}$ and counting the frequency for each potential edge. Since there are less than $|V_P|^2$ candidates for cycle closing edges, and new occurrence lists can be constructed for all of them in a single pass we achieve a runtime of $\mathbf{O}(|V_P|^2|L|)$ to compute all derivative patterns of the input pattern P.

The expansion of a pattern, given its occurrence list L, is polynomial in the size of the pattern, and the size of the occurrence list.
For each expanded pattern, the occurrence list monotonically decreases in size: $|L_{expanded}| \leq |L|$
However this does not mean that this algorithm is efficient or even feasible, since the size of occurrence lists can, and often does in practice, grow exponentially when adding nodes to a pattern.

2.3.2 Without Occurrence List

We will later demonstrate how to compute the frequency of tree patterns $P = (V_P, E_P)$ via subtree isomorphism tests.
The proposed algorithm can be modified to check (somewhat brute force-ish) for a cycle closing edge e if $P' = (V_P, E_P \cup \{e\})$ is still frequent.
By performing this check for all $|V_P|^2 - |V_P| + 1$ possible cycle closing edges. We can find the initial set of frequent cycle closing edges $E_{cc} = \{e_1, e_2, ..., e_m\}$

We proceed now by running a pattern growth algorithm on the edges. Frequency computation for this process is quite complicated and will be discussed in the next chapter.
We define a refinement pattern as a tuple of the pattern tree and the set of cycle closing edges. The initial patterns are $P_i = (P, \{e_i\})$ for i=1...m.
To expand a pattern $R = (P, \{e_{i_1}, e_{i_2}, ..., e_{i_k}\})$ with k cycle closing edges, we first find $i_{max} = max\{i_1, i_2, ..., i_k\}$. We attempt to expand R by all edges in $\{e_{i_{max}+1}, e_{i_{max}+2}, ..., e_m\}$
This enumeration strategy ensures that each combination of edges will be gen-

erated non-redundantly, while making use of the Apriori property to save on computation branches.
Eventually every frequent combination of cycle closing edges is found, without ever enumerating all occurrences of the initial tree pattern.

2.4 Constraints

Beyond occurrence lists, the space of frequent subtrees itself can be incredibly large, especially as patterns fan out. To reduce the space of frequent subtrees while keeping as many interesting patterns as possible, Pham et al.[7] came up with a list of constraints. Their paper aims to find frequent subtrees in Java ASTs. Since their problem is similar to ours, it makes sense to copy some of their constraints as follows.

Constraint	Variable	Value
C1	Maximum # of Leaves	4
C2	Minimum # of Leaves	2
C3	Legal Root Labels	Function
C5	Maximum # of Similar Siblings	10
C10	Minimum # of Nodes	8
C11.1	Interesting labels	Function, If, While, For, Match
C11.2	Minimum Interest	0.5

The naming convention is consistent with both Pham et al. [7] and the code of this project. Note that only constraints C1 - C5 were taken from Pham et al. [7]. C10 is self explanatory.
C11 works as follows:

Definition 2.5 (Interest)**.** Given a labelled pattern graph $P = (V_P, E_P)$ and a set of interesting labels $\mathcal{L}_{interesting}$, then P has interest $\mathcal{I}(P)$
Let $V_{interesting} = \{v \in V_P \mid l(v) \in \mathcal{L}_{interesting}\}$

$$\mathcal{I}(P) = \frac{|V_{interesting}|}{|V_P|} \tag{2.2}$$

Constraint C11 dictates that any pattern P must have $\mathcal{I}(P) \geq min\_interest$ as given by C11.2

Constraints C1, C3 and C10 adhere to anti-monotonicity. Once they are violated by pattern P, there is no extension P' of P that satisfies the constraint again.

This means that C1, C3, and C10 vastly improve computation speed, as they disqualify entire computation branches at once.
C11 is not completely compatible with the enumeration strategies presented so far. It is used as if it was anti-monotonic, as a tool to vastly decrease the size of the result space. But it is not anti-monotonic. To ensure that we loose as few patterns as possible, we use the heuristic of making all interesting labels lexicographically smaller than all other labels.
This flaw to minimum interest will become apparent in the results chapter.

Chapter 3

Frequency Computation

To compute the frequency of a pattern P in a graph database $\mathcal{D} = \{G_1, G_2, ..., G_n\}$ we perform a subgraph isomorphism check on each $G_i \in \mathcal{D}$. This may seem like a brute force approach at first, but is actually asymptotically faster than tracking occurrences. This is because of the following problem:

3.1 Combinatorial Explosion of Occurrence Lists

3.1.1 The Advantages of Occurrence Lists

Occurrence lists make it very easy and elegant to expand patterns. Let $P = (V_P, E_P)$ be a pattern and L be the occurrence list of P.
When extending with a cycle closing edge, the benefits were already discussed in the previous chapter. To reiterate, we can compute all frequent extensions and their respective occurrence lists in $\mathbf{O}(|V_P|^2|L|)$. Furthermore for each resulting pattern $(P_{extended}, L_{extended})$ it holds that $|L_{extended}| \leq |L|$.

When extending with a node the results look polynomial as well:
We can compute all frequent extensions and their respective occurrence lists in $\mathbf{O}(|V_P||F_1||L|)$ where F_1 is the set of frequent labels. For each resulting pattern $(P_{extended}, L_{extended})$ it holds that $|L_{extended}| \leq |L| \cdot max\_deg$.
This multiplicative factor comes from the fact that when extending with label x attached to node i and given occurrence ϕ, there may be max_deg neighbors of $\phi(i)$ with label x, where max_deg is largest out degree in the database. Each such neighbor constitutes a new occurrence ϕ' spawning from ϕ.
While all of the bounds above are polynomial, in practice, we are expanding patterns over and over.
With each expansion possibly adding a multiplicative term to the size of the occurrence list, the number of occurrence we have to track quickly balloons beyond a feasible size.

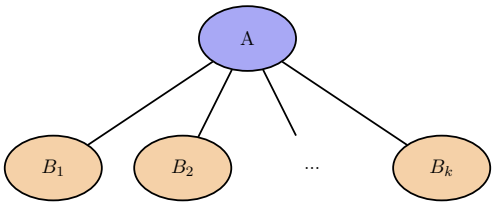

Figure 3.1: Example Graph G

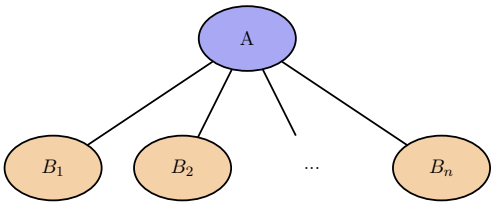

Figure 3.2: Example Graph H

3.1.2 Example

Assume we have two wide fanning trees:
$G = (V_G, E_G) = (\{v_{root}\} \cup \{v_1, v_2, ..., v_k\}, \{(v_{root}, v_i) \ \ \forall i \in [1...k]\})$
$H = (V_H, E_H) = (\{v_{root}\} \cup \{v_1, v_2, ..., v_n\}, \{(v_{root}, v_i) \ \ \forall i \in [1...n]\})$

further assume that $k \leq n$, $l(v_{root}) = \text{A}$ and $l(v_i) = \text{B} \ \ \forall i \in [1...m]$.
To track all occurrences of G in H is to find all edge- and label-preserving injections $\phi : V_G \to V_H$. It is easy to see that the number of ways to fit G in to H is $\binom{n}{k}$

Assume now that we don't have G predefined and are instead enumerating all frequent subtrees in the graph database $\mathcal{D} = \{H\}$ with $min\_support = 1$.
Every graph G with $1 \leq k \leq n$ will be frequent in this problem instance.
Thus the total number of occurrences encountered during the mining process is

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1 \tag{3.1}$$

With this worst case scenario established, we can say that the number of occurrences enumerated when mining with occurrences lists is $\mathbf{o}(2^n)$ where n is the number of nodes in the graph database.
This lower bound is enough to demonstrate the flaws of occurrence lists for the purposes of this thesis. But we conjecture without proof that this bound is in fact tight, i.e. $\Theta(2^n)$.

3.1.3 Pruning as Solution

If we limit the width of candidate patterns as described in the Constraints section to 4, then the example above would only produce at most $\binom{n}{4}$ occurrences. In this case the explosion of the occurrence list only depends on the database graph. One might be inclined to prune any database graph that produces too many occurrences.

This does in fact work in practice, but even with high $min\_support$ (15%),

and allowing up to 10^6 occurrences per graph, it still forces us to prune over 25% of the dataset.
We decided against this approach, instead working on an algorithm that could handle the full spectrum of SASTs.

3.1.4 Frequency Computation Without Occurrence Lists

The true solution to this problem is to compute the frequency of a pattern P in dataset $\mathcal{D}$ without computing all of its occurrences.
By the definition of frequency we do not care how often P appears in each graph $G_i \in \mathcal{D}$. It suffices to perform a subgraph isomorphism check and count the number of graphs for which $P \preccurlyeq G_i$.
We track for each pattern P which graphs it was not contained in. Once P is confirmed not to be contained in graph G_i, any extension $P' \succcurlyeq P$ will also not be contained in G_i, due to the Apriori property: $P \not\preccurlyeq G_i \implies P' \not\preccurlyeq G_i \ \forall P' : P \preccurlyeq P'$
This saves on expensive subgraph isomorphism checks.

3.2 Subgraph Isomorphism in Rooted Trees

3.2.1 Subtree Isomorphism Algorithm

For the following section let tree $G = (V_G, E_G)$ be the pattern tree as we want to decide if G is contained in tree $H = (V_H, E_H)$. To check for subtree isomorphism we propose the **B**ottom-up **E**xlcusive-activations **S**ubtree **I**somorphism **C**hec**K**, or BeSick for short. It first divides the graph into levels, then starts from the bottom (the level furthest from the root) and proceeds upwards, level by level.

Definition 3.1 (Activation)**.** In BeSick a $node_H \in V_H$ is said to activate as $node_G \in V_G$ iff the subtree rooted at $node_H$ contains the subtree rooted at $node_G$.

With bottom-up execution we can assume that activations have been correctly computed for all children $\mathcal{C}_H = \{c_{1_H}, c_{2_H}, ..., c_{m_H}\}$ of $node_H$. Let $\mathcal{C}_G = \{c_{1_G}, c_{2_G}, ..., c_{n_G}\}$ be the children of $node_G$. Each child might have multiple activations.

$$
\begin{aligned}
&\text{The subtree rooted at } node_H \text{ contains the subtree rooted at } node_G \\
\iff &\exists \text{ injection } \psi : \mathcal{C}_G \to \mathcal{C}_H \text{ that respects activations:} \\
&c_G \in activations[c_H] \ \ \forall (c_G, c_H) \in \psi
\end{aligned}
$$

In other words we need to find a way to choose a single activation for each child such that $node_H$ has at least one child activated as c_{i_G} for each child c_{i_G} of $node_G$.

Algorithm 1 BeSick Algorithm for solving $G \preccurlyeq H$ with exclusive activation

```
function IsSubtree(G, H, exclusive_activations)
    divide graph into levels 0 ... L
    for l in [L, L − 1, ..., 0] do
        for node_H in levels_H[l] do
            for node_G in G do
                if CanActivate(node_H, node_G, exclusive_activations) then
                    activations[node_H].append(node_G)
                end if
            end for
            if activations[node_H] contains root_G then
                return true
            end if
        end for
    end for
    return false
end function

function CanActivate(node_H, node_G, exclusive_activations)
    if label(node_H) ≠ label(node_G) then
        return false
    end if
    if node_G in exclusive_activations then
        if exclusive_activations[node_G] == node_H then
            return true
        end if
        return false
    end if
    if node_G is a leaf in G then
        return true
    end if
    requirements = {child_G for all direct children of node_G in G}
    activation_providers = {child_H for all direct children of node_H in H}
    provided_activations = ∅
    for child_H of node_H in H do
        for child_G in activations[child_H] do
                                ▷ indicate that child_H can fulfill requirement child_G
            provided_activations = provided_activations ∪ {(child_H, child_G)}
        end for
    end for

    V_b ← activation_providers ∪ requirements
    E_b ← provided_activations
    bipartite_graph ← (V_b, E_b)
    M ← FindMaximalMatching(bipartite_graph)
    if |M| == |requirements| then
        return true
    end if
    return false
end function
```

This assignment can be found with a maximal matching algorithm. If the maximal matching M covers all children of $node_G$, then M forms an injection and the above condition is met.

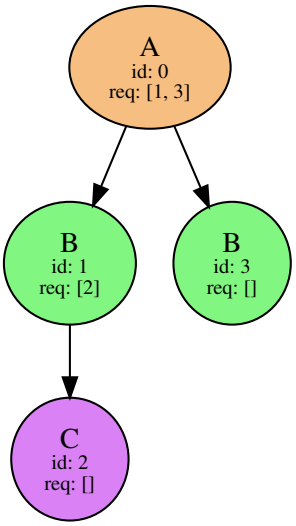

Figure 3.3: Example graph G with ids and requirements

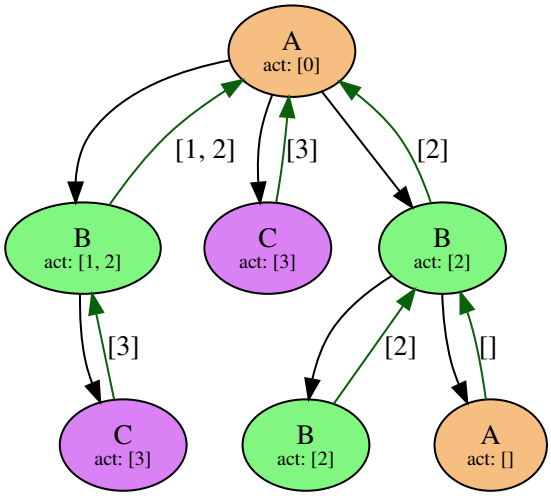

Figure 3.4: Example graph H with activations

3.2.2 Exclusive Activations

The exclusive activations parameter is a partial mapping $\chi : V_G \to V_H$ that must be a part of any injection (occurrence) used to confirm subtree isomorphism. This is easy to enforce by only allowing n_H to activate as n_G. The ability to provide partial mappings will be crucial for mining cycle closing edges later.

3.2.3 Runtime

We examine the pseudocode for BeSick :
Since each node in H can only be in one level of $levels_H$, the first two for loops in total iterate once over all nodes in H. Therefore CanActivate is called $\mathbf{O}(|V_H||V_G|)$ times. Ford-Fulkerson algorithm [8] can find a maximal bipartite matching in $\mathbf{O}(|V_b||E_b|)$ for a bipartite graph $G_b = (V_b, E_b)$.

Theorem 3.2 (Subgraph Isomorphism Runtime)**.** BeSick *terminates in* $\boldsymbol{O}(|V_H|^2|V_G|)$

Proof. Given the analysis above we can say that for two nodes $n_G \in G$ and $n_H \in H$ the bipartite graph $B = (V_B, E_B)$ used to find a maximal matching will

have $|V_B| = deg(n_G) + deg(n_H)$. Due to its bipartite nature we also know that $|E_B| \leq deg(n_G)deg(n_H)$. We can then determine the runtime:

$$\mathbf{O}\Big(\sum_{L_i \in levels_H} \sum_{n_H \in L} \sum_{n_G \in G} |V_B||E_B|\Big) \tag{3.2}$$

$$\leq \mathbf{O}\Big(\sum_{L_i \in levels_H} \sum_{n_H \in L} \sum_{n_G \in G} (deg(n_G) + deg(n_H))(deg(n_G)deg(n_H))\Big) \tag{3.3}$$

$$= \mathbf{O}\Big(\sum_{L_i \in levels_H} \Big(\sum_{n_H \in L} \sum_{n_G \in G} deg(n_G)deg(n_H)^2 + \sum_{n_H \in L} \sum_{n_G \in G} deg(n_G)^2 deg(n_H)\Big)\Big)$$

$$\leq \mathbf{O}\Big(\sum_{L_i \in levels_H} \Big(|V_G| \sum_{n_H \in L} deg(n_H)^2 + |L_{i+1}| \sum_{n_G \in G} deg(n_G)^2\Big)\Big) \tag{3.4}$$

$$\leq \mathbf{O}\Big(\sum_{L_i \in levels_H} \Big(|V_G||L_{i+1}|^2 + |L_{i+1}||V_G|^2\Big)\Big) \tag{3.5}$$

$$\leq \mathbf{O}\Big(|V_G||V_H|^2 + |V_G|^2|V_H|\Big) \tag{3.6}$$

$$\leq \mathbf{O}\Big(|V_G||V_H|^2\Big) \tag{3.7}$$

Note that 3.7 holds because $0 \leq |V_G| \leq |V_H|$

□

3.3 Recovering Occurrence List

Occurrence list mining algorithms know all occurrences of a pattern by updating the list as the pattern grows step by step. When given just graphs G and H it is considerably more difficult to find all occurrences of G in H. The algorithm above can be modified to enumerate all occurrences. We propose BeSwole , the **B**ottom-up **E**xlcusive-activations **S**ubtree isomorphism **W**ith **O**ccurrence **L**azy-**E**numeration.
Analyzing the exact runtime is of little use when the algorithm can produce an output exponential in the size of the input. But BeSwole can compute the next occurrence in polynomial time, and will enumerate all occurrences eventually.

3.3.1 Enumerating All Possible Solutions

While BeSick is content with finding a single way for $node_H$ to activate as $node_G$, BeSwole must find all possible ways. This means finding all maximal matchings of the bipartite graph constructed by BeSick .
To do this we use the algorithm proposed by Uno [9], which can enumerate all maximal matchings in $\mathbf{O}(mn^{\frac{1}{2}} + nN_m)$ where N_m is the number of maximal

matchings.
For the implementation we used the Python implementation by Jason [10].

3.3.2 Enumerating All Matching Combinations

To obtain all occurrences we must proceed recursively. Assume n_H activates as n_G. Without loss of generality, we will treat occurrences as sets of tuples. The goal is to compute all occurrences of the subtree rooted at n_G in the subtree rooted at n_H, denoted as $\Phi_{n_H,n_G} = \{\phi^{(1)}_{n_H,n_G}, \phi^{(2)}_{n_H,n_G}, ..., \phi^{(N_{occs})}_{n_H,n_G}\}$

If n_G is a leaf then there is only one mapping: $\Phi_{n_H,n_G} = \{(n_H, n_G)\}$

Else we need to compute all maximal matchings of size $|children(n_G)|$:
$\mathcal{M}_{max} = \{M_1, M_2, ..., M_m\}$. Each matching
$M_j = \{(c_H, c_G) \ : c_H \in children(n_H), c_G \in children(n_G)\}$ is itself an injection $children(n_H) \to children(n_G)$

$$\Phi_{n_H,n_G} = \bigcup_{M_j \in \mathcal{M}_{max}} \Big(\{(n_H, n_G)\} \cup \phi' \\ \forall \phi' \in \textsc{AllCombinations}(\{\Phi_{c_H^{(1)},c_G^{(1)}}, \Phi_{c_H^{(2)},c_G^{(2)}}, ..., \Phi_{c_H^{(k)},c_G^{(k)}}\}) \Big)$$

Where $M_j = \{(c_H^{(1)}, c_G^{(1)}), (c_H^{(2)}, c_G^{(2)}), ..., (c_H^{(k)}, c_G^{(k)})\}$.
As you can see the AllCombinations output alone can contain an exorbitant amount of elements. Explained intuitively, for every way to match n_H to its children, we output every combination of ways those children can match to their own subtrees.
This is a recursive definition of all possible edge- and label-preserving mappings in Φ_{n_H,n_G}. In the case of trees, all these mappings will be injections. Since the algorithm is implemented according to this recursive definition we will not argue the correctness of BeSwole any further.

3.3.3 Lazy evaluation

Amazingly, all of the recursive elements of BeSwole can be implemented using lazy evaluation.

- Uno [9] can lazily compute the next maximal matching in $\mathbf{O}(mn^{\frac{1}{2}} + n)$
- To produce the next occurrence we need at most $\mathbf{O}(|V_H|)$ new maximal matchings
- We can lazily enumerate all combinations of a list of lazy evaluated lists $\mathbb{L}$. Computing the next element in the worst case scenario takes $\mathbf{O}(|\mathbb{L}|)$

The upper bounds given above are very generous and could likely be tighter. But they show that BeSwole is able to not just enumerate all occurrences, but always produce the next occurrence in polynomial time. In other words, BeSwole is an enumeration algorithm with polynomial delay, as defined by [11].

3.4 Subgraph Isomorphism in Directed Acyclic Graphs

The algorithms presented above can also be used to check if a tree G is contained in a DAG H. However both algorithms treat merging branches as if they duplicated the merge point and the subtree rooted there. In practice this means that means that BeSick will sometimes produce false positives.

3.4.1 Phantom Occurrences

Definition 3.3 (Phantom Occurrence)**.** A phantom occurrence of $G = (V_G, E_G)$ in $H = (V_H, E_H)$ is a mapping $\phi : V_G \to V_H$ that preserves labels and edges, but is not injective.

If such a mapping exists, then BeSick will return **true**, even if $G \not\preccurlyeq H$.
To catch this one-sided error we can use BeSwole to enumerate all occurrences, and check if they are injective. Once the first injective occurrence is found we can determine that $G \preccurlyeq H$ and we can stop the enumeration.
The downside of this approach is that in the worst case we must enumerate the entire occurrence list, which may be very large.

3.4.2 Working With Cycle Closing Edges

We use exclusive activations to simulate cycle closing edges. As previously stated, exclusive enumerations allow us to force certain nodes in H to activate as certain nodes in G.

When mining for cycle closing edges, a pattern $P = (T, E)$ is given by its tree pattern $T = (V_T, E_T)$, and a list of cycle closing edges E.

Definition 3.4 (Edge Occurrence)**.** Given pattern $P = (T, E)$ and graph $G = (V_G, E_G)$ We define an edge occurrence as an injection $\psi : E \to E_G$ such that there exists an occurrence of T in G that ψ is compatible with. More precisely, for any occurrence $\phi : V_T \to V_G$, it holds that $\psi((u_T, v_T)) = (u_G, v_G) \Rightarrow \phi(u_T) = u_G \wedge \phi(v_T) = v_G \;\; \forall (u_T, v_T) \in E$.

As you can see by the definition above, such an edge occurrence can be expressed as a list of exclusive activations.

With this system we can track all ways the cycle closing edges can map to edges in the graph. Sadly this once again means we are back to tracking occurrences, however this form of occurrence list is significantly smaller than the complete occurrence list of the pattern graph.

Chapter 4

Mining Strategy

4.1 Optimistic Mining

4.1.1 Large Pattern Heuristic

Let us revisit the example from 3.1.2. In this worst case scenario, the number of occurrences is $\binom{n}{k}$ where k is the width of the pattern and n is the width of the graph. $\binom{n}{k}$ reaches its maximum at $k = \frac{n}{2}$ and decreases back to 1 as you get further away from that point.
Based on this we conjecture that as patterns are grown, their occurrence count tends to increase quickly, but eventually decreases again. If we somehow manage to mine only large patterns, then checking frequency with occurrence lists could be feasible.

4.1.2 Double Checking

We optimistically mine for frequent subtrees in SASTs, using BeSick for frequency computation. This can produce phantom patterns that are wrongfully considered frequent. Constraints are used to output only patterns with 8 or more nodes. BeSwole is then used to double check the frequency of these patterns.

4.1.3 Performance

In theory optimistic mining could produce an exorbitant amount of phantom patterns, that are not actually frequent in the dataset. However in practice our dataset is so similar to a tree-only dataset that BeSick is rarely wrong and there are very few phantom patterns.
This second step however still comes with a large computational cost.

4.2 Refinements

Borrowing the term from the terminology of Gaston [5], refining refers to the process of adding cycle closing edges to patterns.
Once we have mined frequent tree patterns for some minimum support s_{min}, we can then mine for frequent refinements with minimum support $s'_{min} \leq s_{min}$.
Lowering the support value is useful since the resulting DAG patterns tend to be far less frequent than the trees patterns they are based on.

Chapter 5

Results

5.1 Termination

The biggest accomplishment of our method is that it terminates on a very large and uncontrolled dataset of directed acyclic graphs. Any algorithm that relies on occurrence tracking would encounter a wide fanning graph very soon, and get hung up enumerating its occurrences until the memory or time limit is exceeded.

5.2 Statistical

We ran our method with minimum support values of 10%, 15% and 20%. The setup with 5% did not terminate within the given computation time (7 days), but we can extract information from the other three runs.

We also analyze the frequencies of each label in the patterns found. From this we conclude that interesting labels are generally very rare in patterns of such high support values.

5.3 Minimum Interest Constraint

Setups with a minimum interest constraint set to 50% were run using $min\_support$ values $5\%, 4\%, 3\%, 2\%, 1\%$. Curiously the 4% run terminated in time, finding only 900 patterns, while all other runs found over 4700 patterns and exceeded the maximum computation time (7 days).
This indicates that the minimum interest constraint interferes with the enumeration strategy far more than expected. In essence this constraint removes completeness guarantees from the canonical rightmost path extension strategy. It may disqualify a pattern from being extended even if that pattern is the only one that can be extended into several interesting patterns later down the line.

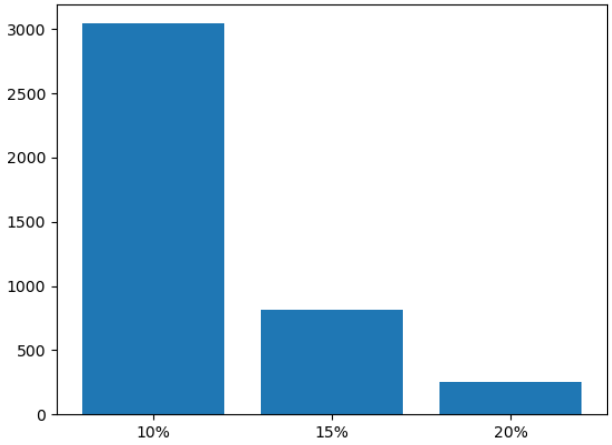

Figure 5.1: Number of patterns found

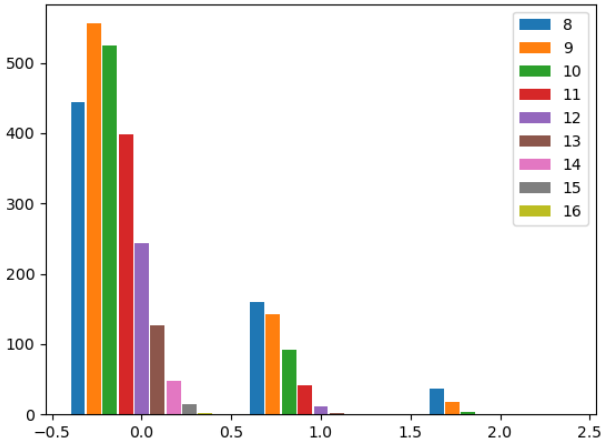

Figure 5.2: Sizes of patterns found

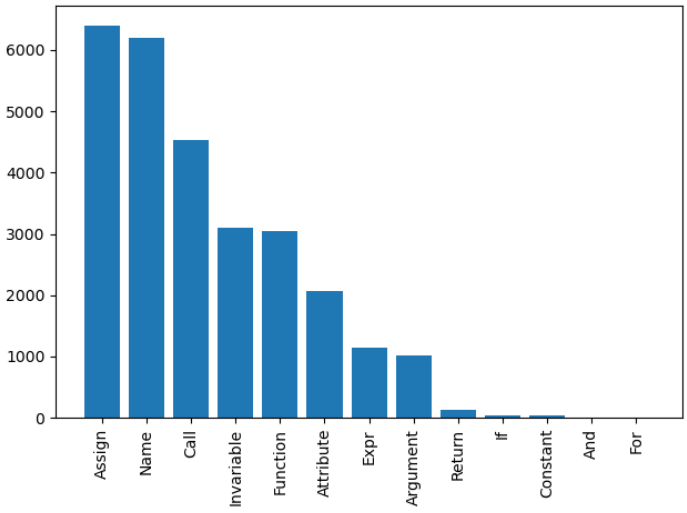

Figure 5.3: Frequency of labels in the found patterns

As mentioned earlier we took steps to mitigate this problem but it seems to have helped very little.

5.4 Refined Patterns

While we made strides in mining patterns from this dataset, in practice the final step of adding cycle closing edges is still computationally expensive. It takes roughly 2 hours on all 20 cores of a Dual Deca-Core Intel Xeon E5-2690 v2, as well as 200 Gigabytes of memory to mine refinements for a single pattern.
Because of limited resources we cannot perform this operation for all the mined patterns, but here are a few hand-picked examples. The support values used were 10% for the initial mining step, and 0.5% for the refinement mining step.

5.5 Comparison to Other Methods

5.5.1 Finding More Interesting Patterns

We have proposed a traditional graph mining approach to mining ASTs with added semantic information. Allamanis et al. [12] use a process they call coiling, which is similar to the implementation of SASTs used in this thesis. They use probabilistic tree grammars instead of full graph pattern mining and manage to find more interesting patterns while bypassing the issues of graph pattern mining.

The same can be said about the work of Sivaraman et al. [13], who used a more complex process to add semantic information. They call this process dataflow augmentation and their mining approach once again foregoes graph mining in favor of non-parametric Bayesian methods.

It must be noted that these methods restrict themselves only to loop-level graphs, a further simplification to function-level graphs used by our method.

5.5.2 Advantages

Our algorithms are not dependent on the exact implementation of SASTs. As long as the representation is a connected DAG, our methods will be compatible. The results, and specifically the amount of phantom patterns found, will come down to the specific implementation of SASTs.

With the two-step mining approach it is possible to avoid some of the workload of the second step by filtering for interesting patterns as an intermediate step.

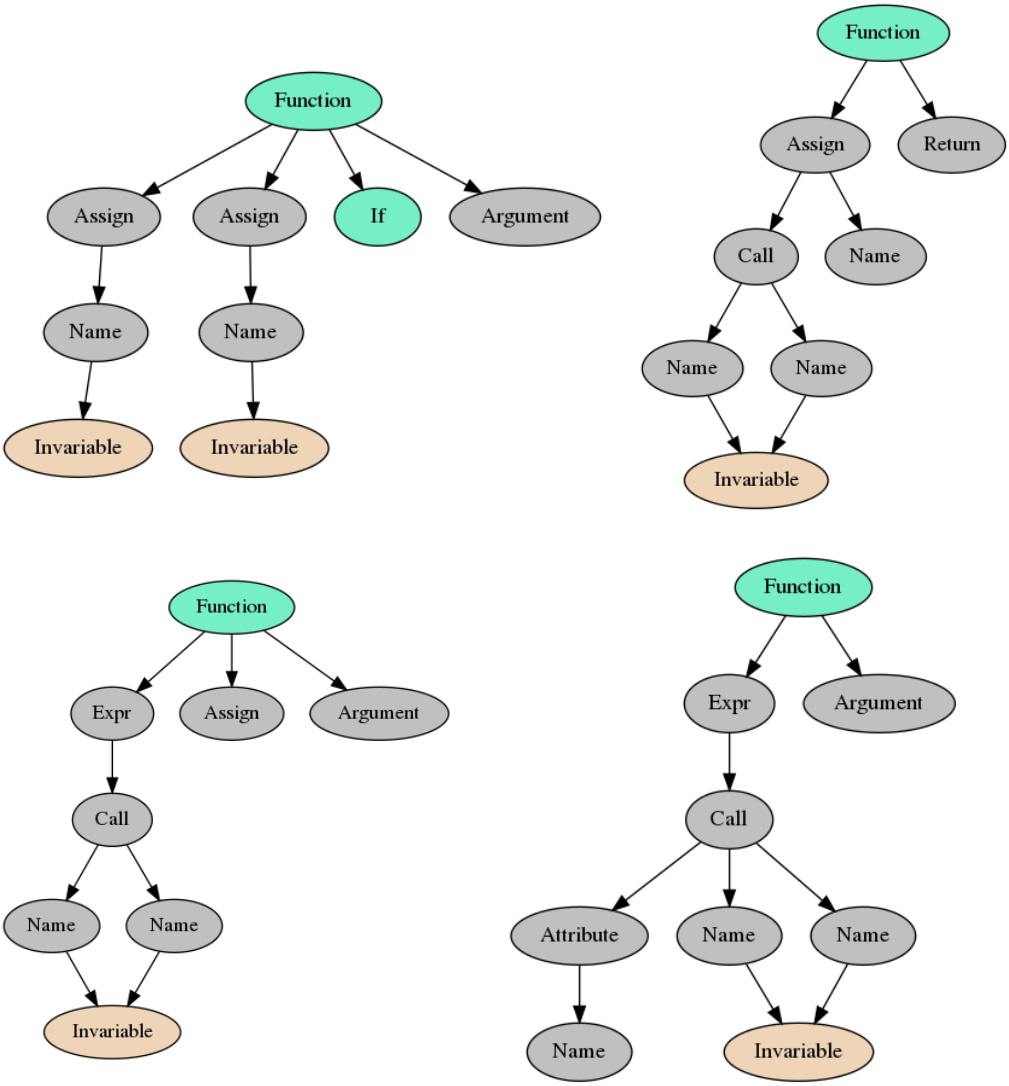

Figure 5.4: Some hand picked interesting patterns.

5.5.3 Future work

To improve upon our method, one could implement a more advanced enumeration strategy for the first mining step.
Also given a specific implementation of SASTs, one might come up with a specialized version of our method that handles that implementation more efficiently.

However there will always be one bottleneck to traditional graph pattern mining approaches: How well are we able to define what makes a pattern "interesting" by constraints and parameters.
For this reason alone it may be beneficial to use graph neural networks for such applications in the future.

Bibliography

[1] M. J. Zaki, "Efficiently mining frequent embedded unordered trees," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 33–52, 2005.

[2] D. W. Matula, "Subtree isomorphism in o (n5/2)," in *Annals of Discrete Mathematics*. Elsevier, 1978, vol. 2, pp. 91–106.

[3] P. Olivares, R. Pagli, F. Luccio, A. Enriquez, P. Rieumont, and L. Pagli, "Bottom-up subtree isomorphism for unordered labeled trees," 07 2004.

[4] D. E. (https://cstheory.stackexchange.com/users/95/david eppstein), "Is dag isomorphism np-c," Theoretical Computer Science Stack Exchange, uRL:https://cstheory.stackexchange.com/q/25976 (version: 2017-12-24). [Online]. Available: https://cstheory.stackexchange.com/q/25976

[5] S. Nijssen and J. N. Kok, "A quickstart in frequent structure mining can make a difference," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 647–652.

[6] B. D. McKay *et al.*, "Practical graph isomorphism," 1981.

[7] H. S. Pham, S. Nijssen, K. Mens, D. D. Nucci, T. Molderez, C. D. Roover, J. Fabry, and V. Zaytsev, "Mining patterns in source code using tree mining algorithms," in *International Conference on Discovery Science*. Springer, 2019, pp. 471–480.

[8] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian journal of Mathematics*, vol. 8, pp. 399–404, 1956.

[9] T. Uno, "Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs," in *International Symposium on Algorithms and Computation*. Springer, 1997, pp. 92–101.

[10] J. (https://stackoverflow.com/users/2005415/jason), "All possible maximum matchings of a bipartite graph," Stack Overflow, uRL:https://stackoverflow.com/questions/37144423/ (version: 2017-5-23). [Online]. Available: https://stackoverflow.com/questions/37144423/

[11] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, "On generating all maximal independent sets," *Information Processing Letters*, vol. 27, no. 3, pp. 119–123, 1988.

[12] M. Allamanis, E. T. Barr, C. Bird, P. Devanbu, M. Marron, and C. Sutton, "Mining semantic loop idioms," *IEEE Transactions on Software Engineering*, vol. 44, no. 7, pp. 651–668, 2018.

[13] A. Sivaraman, R. Abreu, A. Scott, T. Akomolede, and S. Chandra, "Mining idioms in the wild," *arXiv preprint arXiv:2107.06402*, 2021.

Appendix A

Combinations Algorithm

Given a set of sets of sets of sets AllCombinations computes the union of all combinations of the first order sets.
Example: AllCombinations($\Big\{\big\{\{1\},\{2,3\}\big\},\big\{\{A\},\{B,C\}\big\}\Big\}$) =
$\{\{1,A\},\{1,B,C\},\{2,3,A\},\{2,3,B,C\}\}$
It treats each of the given set in the input like a wheel on a combination lock.

Algorithm 2 AllCombinations pseudocode

```
function AllCombinations(wheels)
    all_combinations = ∅
    if wheels = ∅ then
        return {∅}
    else
        (w, remaining_wheels) ← wheels
        for elem in w do
            for suffix in AllCombinations(remaining_wheels) do
                all_combinations = all_combinations ∪ {{elem} ∪ suffix}
            end for
        end for
    end if
    return all_combinations
end function
```

Appendix B

Occurrence Enumeration Algorithm

Algorithm 3 BeSwole Algorithm for enumerating occurrences of G in H with exclusive activations

```
function EnumerateOccurrences(G, H, exclusive_activations)
    run BeSick(G, H, exclusive_activations) but store every matching
    occurrences = ∅
    for node_H in H do
        if node_root in activations[node_H] then
            occs = EnumerateOccurrencesFromNode(node_H, node_root)
            occurrences = occurrences ∪ occs
        end if
    end for
    return occurrences
end function

function EnumerateOccurrencesFromNode(node_H, node_G)
    if node_G is leaf then
        return {(node_H, node_G)}
    end if
    all_occurrences = ∅
    for matching M in activations[node_H][node_G] do
        child_combo_elements = ∅
        for (c_H, c_G) in M do
            child_occs = EnumerateOccurrencesFromNode(c_H, c_G)
            child_combo_elements = child_combo_elements ∪ {child_occs}
        end for
        child_occurrences = AllCombinations(child_combo_elements)
        for child_occ in child_occurrences do
            occurrence = child_occ ∪ {(node_H, node_G)}
            all_occurrences = all_occurrences ∪ occurrence
        end for
    end for
    return all_occurrences
end function
```