FREQUENT SUBGRAPH ANALYSIS AND ITS SOFTWARE ENGINEERING APPLICATIONS

by

TIM A. D. HENDERSON

Submitted in fulfillment of the requirements

for the degree of Doctor of Philosophy

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

August, 2017

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the dissertation of

Tim A. D. Henderson

candidate for the degree of **Doctor of Philosophy**.

Committee Chair
Dr. Andy Podgurski

Committee Member
Dr. Soumya Ray

Committee Member
Dr. Harold Connamacher

Committee Member
Dr. Gurkan Bebek

Date of Defense

April 26, 2017

Contents

Dedication

I dedicate this book to my wife Leigh. Thank your for your support, your critiques, your edits, and most of all your love for all these years.

Acknowledgments

I am grateful to everyone who helped and supported me through my years at Case Western Reserve University. Especially to my wife Leigh Henderson without whom this dissertation would not have been possible. I would also like to thank Stuart Saunders and Mobile Defense Inc. for their generous support through the years. I worked with wonderful people here at CWRU including: Stephen Johnson, Gary Doran, Gurkan Bebek, Gang Shu, Boya Sun, Arda Durmaz, Shaun Howard, Koby Picker, Junqi Ma, and Yigit Kucuk. Without their thoughtful collaboration and advice I could not have completed this work.

I would also like to acknowledge Tekin Özsoyoglu who first encouraged me and supported me with a GAANN Fellowship. Vincenzo Liberatore gave me my first teaching position by placing me in charge of the practical portion of the Software Craftsmanship course. Ken Loparo also believed in me allowing me the privilege of teaching Compiler Design and Implementation. Teaching this course lead to my renewed interest in my research and opened up new avenues of exploration.

I would like to acknowledge my committee members: Soumya Ray, Harold Connamacher, and Gurkan Bebek. Each of you have given me thoughtful advice and support through my years at CWRU. Finally, I would like to thank my research advisor Andy Podgurski. Through thick and sometimes thin you have continued to believe in me and my work. Your thoughtful guidance has kept me on the straight and narrow. Without your gentle nudges and suggestions my wandering mind would've whisked me toward the bogs of folly. This work was difficult for me and I could not have done it without your support.

Frequent Subgraph Analysis and its Software Engineering Applications

Abstract

by

TIM A. D. HENDERSON

Frequent subgraph analysis is a class of techniques and algorithms to find repeated sub-structures in graphs known as frequent subgraphs or graph patterns. In the field of Software Engineering, graph pattern discovery can help detect semantic code duplication, locate the root cause of bugs, infer program specifications, and even recommend intelligent auto-complete suggestions. Outside of Software Engineering, discovering graph patterns has enabled important applications in personalized medicine, computer aided drug design, computer vision, and multimedia.

As promising as much of the previous work in areas such as semantic code duplication detection has been, finding all of the patterns in graphs of a large program's code has previously proven intractable. Part of what makes discovering all graphs patterns in a graph of a large program difficult is the very large number of frequent subgraphs contained in graphs of large programs. Another impediment arises when graphs contain frequent patterns with many automorphisms and overlapping embeddings. Such patterns are pathologically difficult to mine and are found in real programs.

I present a family of algorithms and techniques for frequent subgraph analysis with two specific aims. One, address pathological structures. Two, enable important software engineering applications such as code clone detection and fault localization without analyzing all frequent subgraphs. The first aim is addressed by novel optimizations making the system faster and more scalable than previously published work on both program graphs and other difficult to mine graphs. The second aim is addressed by new algorithms for sampling, ranking, and grouping frequent patterns. Experiments and theoretical results show the tractability of these new techniques.

The power of frequent subgraph mining in Software Engineering is demonstrated with studies on duplicate code (code clone) identification and fault localization. Identifying code clones from program dependence graphs allows the identification of potential semantic clones. The proposed sampling techniques enable tractable dependence clone identification and analysis. Fault localization identifies potential locations for the root cause of bugs in programs. Frequent substructures in dynamic program behavior graphs to identify suspect behaviors which are further isolated with fully automatic test case minimization and generation.

Chapter 1

Introduction

Frequent subgraph analysis (FSA) is a family of techniques to discover recurring subgraphs in graph databases. The databases can either be composed of many individual graphs or a single large connected graph. This dissertation discusses my contributions to frequent subgraph analysis and applies the technique to address two pressing problems in software engineering: code clone detection and automatic fault localization.

The work on frequent subgraph analysis was motivated by the software engineering problems. Large programs are composed of repeated patterns arising organically through the process of program construction. Some regions of programs are duplicated (intentionally or unintentionally). The duplicated regions are referred to as *code clones* (or just *clones*). Other regions are similar to each other because they perform similar tasks or share development histories.

Code clones may arise from programmers copying and pasting code, from limitations of a programming language, from using certain APIs, from following coding conventions, or from a variety of other causes. Whatever their causes, the existing clones in a code base need to be managed. When a programmer modifies a region of code that is cloned in another location in the program they should make an active decision whether or not to modify the other location. Clearly, such decisions can only be made if the programmer is aware of the other location.

One type of duplication which is particularly difficult to detect is so called *Type-4 clones* or *semantic clones*. Semantic clones are semantically equivalent regions of code which may or may not be textually similar. Differences could be small changes such as different variable names or large changes such as a different algorithms which perform the same function. In general identifying semantically equivalent regions is undecidable as a reduction from the halting problem.

Frequent subgraph analysis (FSA) can be used to identify some *Type-4* clones (as well as easier to identify clone classes). I use FSA to analyze a graphical representation of the program called the *Program Dependence Graph* (PDG) [46]. Dependence graphs strip away syntactic information and focus on the semantic relationships between operations. Non-semantic re-orderings of operations in a program do not effect the structure of its dependence graph [64, 113, 114]. Since PDGs are not sensitive to unimportant syntactic changes some of the *Type-4* clones in a program may be identified with FSA.

The other motivating application I applied FSA to is automatic fault localization. When programs have faults, defects, or bugs it is often time consuming and sometimes difficult

to find the cause of the bug. To address this the software engineering community has been working on a variety of techniques for *automatic fault localization*. The family of statistical fault localization techniques analyzes the behavior of the program when the faults manifest and when they do not. These techniques then identify statistical associations between execution of particular program elements and the occurrence of program failures.

While statistical measures can identify suspicious elements of a program they are blind to the relationships between the elements. If program behavior is modeled through *Dynamic Control Flow Graphs*, then execution relationships between operations can be analyzed using FSA to identify suspicious interactions. These suspicious interactions represent larger *behaviors* of the program which are statistically associated with program failure. The behaviors serve as a context of interacting suspicious program elements which potentially makes it easier for programmers to comprehend localization results.

Much of the previous work in frequent subgraph analysis has focused on finding all of the frequent subgraphs in a graph database (called *frequent subgraph mining* [71]). I have shown that finding all of the frequent subgraphs in a database of graphs is not an efficient or effective way to either detect code clones or automatically localize faults. Program dependence graphs of large programs have huge numbers of recurring subgraphs. Experiments on a number of open source projects (see Chapter 4) showed that moderately sized Java programs ($\sim$70 KLOC) have more than a hundred of million subgraphs that recur five or more times. Mining all recurring subgraphs is an impractical way to either identify code clones or localize faults.

Furthermore, it turns out that program dependence graphs are particularly difficult to analyze for recurring subgraphs. These graphs often have certain structures which contain many *automorphisms*. A structure with an automorphism can be rotated upon itself. Each rotation appears to be a recurrence to traditional frequent subgraph mining algorithms. However, because it is merely a rotation, humans (e.g. programmers) do not perceive these rotations as instances of duplication.

To enable scalable frequent subgraph analysis of large programs new techniques were needed. I developed novel optimizations for mining frequent subgraphs and created a state of the art miner (REGRAX) for connected graphs (Chapter 3). To detect code clones from program dependence graphs, I developed an algorithm (GRAPLE) to collect a representative sample of recurring subgraphs (Chapter 4). Finally, a new algorithm (SWRW) was created for localizing faults from dynamic control flow graphs, which outperforms previous algorithms (Chapter 6).

REGRAX contains low level optimizations to the process of identifying frequent subgraphs. Chapter 2 provides the necessary background on frequent subgraph mining for understanding these optimizations. An extensive empirical study was conducted on REGRAX to quantify the effect of each of the new optimizations on databases from the SUBDUE corpus [27], on program dependence graphs, and on random graphs.

GRAPLE is a new algorithm to sample a representative set of frequent subgraphs and estimate statistics characterizing properties of the set of all frequent subgraphs. The sampling algorithm uses the theory of absorbing Markov chains to model the process of extracting recurring subgraphs from a large connected graph. By sampling a representative set of recurring subgraphs GRAPLE is able to conduct frequent subgraph analysis on large programs which normally would not be amenable to such analysis.

One of the questions in code clone detection is: "are code clones detected from program dependence graphs understandable to programmers?" GRAPLE was used to answer this question, as it not only collects a sample of frequent subgraphs but allows researchers to estimate the prevalence of features across the entire population of frequent subgraphs (including

those which were not sampled). Chapter 4 details a case study which was conducted at a software company to determine whether their programmers could make use of code clones detected from program dependence graphs. The study would not have been possible without the estimation framework in GRAPLE, as the software contained too many code clones to be reviewed in the allocated budget.

To apply frequent subgraph analysis to automatic fault localization, a new algorithm named Score Weighted Random Walks (SWRW) was developed. SWRW samples discriminative, suspicious, or significant subgraphs from a database of graphs. The database is split into multiple classes where some graphs are labeled "positive" and others "negative." In fault localization the "positive" graphs were those dynamic control flow graphs collected from program executions which exhibited a failure of some type. The "negative" graphs are from executions which did not fail.

SWRW, like GRAPLE, models the problem using the theory of absorbing Markov chains. Unlike GRAPLE, it uses an *objective function* (drawn from the statistical fault localization literature [99]) to guide the sampling process. In comparison to previous work in fault localization using graph mining, a much wider variety of objective functions can applied. This allows for functions better suited to statistical fault localization to be used as the objective function. SWRW outperforms previous approaches which used discriminative mining to localize faults in terms of fault localization accuracy.

Summary This dissertation makes important and novel contributions to frequent subgraph analysis which enable scalable semantic code clone detection and behavioral fault localization. These advances can help programmers maintain their software more efficiently leading to more stable and secure software for everyone. The software engineering advances are built on new frequent subgraph analysis algorithms. The new algorithms improve code clone detection time, fault localization latency and accuracy, and enable analysis of larger and more complex programs.

Chapter 2

An Introduction to Frequent Subgraph Analysis

Frequent subgraph mining is the process of identifying recurring sub-structures in graphical databases [25]. The database may either be a collection of small graphs or a single large graph. Figure 2.1 shows an example. On the right side of the figure is the "database of graphs" which contains three undirected unlabeled graphs. On the left side is a triangle graph which is a subgraph of all three graphs in the database. Thus, the triangle graph *recurs* three times in the database and is a *3-frequent* subgraph.

The goal of frequent subgraph mining is to discover all subgraphs of graphs in the database which recur at least k times. The parameter k is called the *minimum support*. A subgraph is considered "frequent" (and *supported*) if it recurs at least k times. When the database is a collection of small graphs the notion of recurrence (or *support*) is a straightforward one: a graph has support n if it is a subgraph of n graphs in the database. However, as we shall see, the concept of support is more subtle in when analyzing a single large graph.

2.1 Formal Definitions

A graph $G = (V, E)$ consists of a set of *vertices* V and *edges* $E \subseteq V \times V$. If $(u, v) \in E$ then there is an edge between the vertices (or *nodes*) u and v. If a graph is *undirected* then $(u, v) \in E$ implies that $(v, u) \in E$. However, if $(u, v) \in E$ does not imply $(v, u) \in E$ the graph is said to be *directed*. A *labeled graph* (either directed or undirected) has an additional component: a labeling function l which maps vertices and/or edges to a set of labels L. In this work, the focus will be on *labeled directed graphs* (called *labeled digraphs*) as the primary

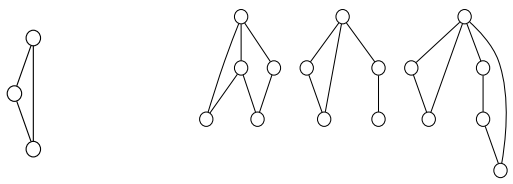


Figure 2.1: The triangle graph on the left recurs in each of the graphs shown on the right.

Param: Database of labeled digraphs $\mathcal{D}$
Param: Minimum support k
Result: Set of k-frequent subgraphs S

1 **for** *all candidate subgraphs H, H may be a subgraph of some $G \in \mathcal{D}$* **do**
2 **if** $|\{G : G \in \mathcal{D} \wedge H \sqsubseteq G\}| \geq k$ **then**
3 Add H to S
4 **end**
5 **end**
6 **return** S

Algorithm 1: High-level overview of k-Frequent Subgraph Mining.

software engineering applications involve labeled digraphs. Some of the examples (such as Figure 2.1) will utilize undirected graphs for simplicity.

Definition 2.1 (Subgraph Isomorphism)**.** *Given labeled digraphs H and G, an injective mapping $m : V_H \to V_G$ is a* subgraph isomorphism *when:*

1. *m maps each vertex in H to a vertex in G with the same label:* $\forall\ v \in V_H\ [l_H(v) = l_G(m(v))]$
2. *m preserves edges:*
   $\forall\ u, v \in V_H\ [(u, v) \in E_H \Leftrightarrow (m(u), m(v)) \in E_G]$
3. *m preserves edge labels:*
   $\forall\ (u, v) \in E_H\ [l_H(u, v) = l_G(m(u), m(v))]$

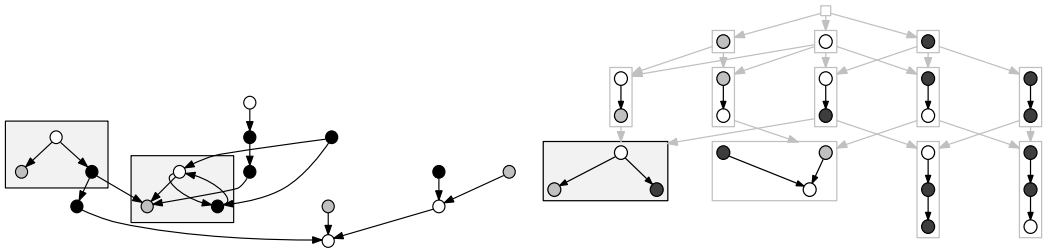
If there is a subgraph isomorphism m between H and G then H is a subgraph of G, denoted $H \sqsubseteq G$. A subgraph isomorphism from H into G is called an *embedding* of H in G. The set of all of subgraph isomorphisms from H into G is denoted by $[\![H]\!]_G$.

Definition 2.2 (k-Frequent Subgraph Mining a Database $\mathcal{D}$)**.**
Given a minimum frequency level $k > 1$ and a set of labeled digraphs $\mathcal{D}$ find all digraphs H such that $|\{G : G \in \mathcal{D} \wedge H \sqsubseteq G\}| \geq k$.

A graph H which is a subgraph of at least k graphs in $\mathcal{D}$ is called a k-frequent subgraph. Algorithm 1 gives a very high level overview of the process all frequent subgraph mining algorithms follow. Line 1 enumerates all potentially frequent subgraphs (called *candidate frequent subgraphs* of graphs in the database $\mathcal{D}$. Line 2 then counts the support of the current subgraph H in the database $\mathcal{D}$. If the graph H is frequent in the database it is added to the set of frequent subgraphs on Line 3. In order to efficiently mine frequent subgraphs careful implementations of Lines 1 and 2 will need to be constructed.

Line 1 in Algorithm 1 enumerates candidate frequent subgraphs. This enumeration can be viewed as a traversal of a conceptual structure called the *k-frequent connected subgraph lattice*. The lattice is a special graph where the nodes (vertices) of the lattice represent k-frequent subgraphs and the edges connect the nodes to their *direct* supergraphs or subgraphs. The k-frequent connected subgraph lattice is a subgraph of the connected subgraph lattice.

Definition 2.3 (Connected Subgraph Lattice of $\mathcal{D}$)**.** *The subgraph relation $\cdot \sqsubseteq \cdot$ induces the* Connected Subgraph Lattice $\mathcal{L}_\mathcal{D}$ *representing all the possible ways of constructing the graphs in $\mathcal{D}$ from the empty subgraph by adding one edge at a time. $\mathcal{L}_\mathcal{D}$ is a digraph where each vertex u represents a unique connected (ignoring edge direction) subgraph of some $G \in \mathcal{D}$.*



(a) A colored (labeled) digraph.

(b) A 2-frequent connected subgraph lattice.

Figure 2.2: Figure (b) is a connected subgraph lattice of the graph in Figure (a) including only the subgraphs with 2 or more embeddings in Figure (a). The boxed nodes in the graph show the embeddings of the boxed subgraph in the lattice. In the figure, the colors (black, gray, and white) are standing in for labels on the vertices.

There is an edge from u to v in $\mathcal{L}_\mathcal{D}$ if adding some edge ϵ to u creates a subgraph $u + \epsilon$ isomorphic to v, $v \cong u + \epsilon$.

Definition 2.4 (k-Frequent Connected Subgraph Lattice of $\mathcal{D}$)**.** *A k-Frequent Connected Subgraph Lattice k-$\mathcal{L}_\mathcal{D}$ is a connected subgraph lattice containing only those subgraphs which appear in least k graphs in $\mathcal{D}$.*

The goal of k-frequent subgraph mining is to discover all of the vertices of the k-$\mathcal{L}_\mathcal{D}$.

2.2 Key Problems in Frequent Subgraph Mining

At a high level all frequent subgraph mining algorithms are traversals of the k-frequent connected subgraph lattice of the database of graphs $\mathcal{D}$. All methods start from the empty subgraph and propose *candidate subgraphs* of G that might be frequent, check their support and propose further, larger candidate subgraphs until all vertices of the lattice have been found. We will consider four major sub-problems:

1. Subgraph enumeration order
2. Support computation
3. Generation of candidate subgraphs
4. Elimination of duplicate subgraphs

2.2.1 Subgraph Enumeration

Figure 2.2 shows an example of a k-frequent subgraph lattice k-$\mathcal{L}_G$ for a single connected graph. Frequent subgraph mining discovers the vertices of the lattice starting from an empty subgraph. Algorithms propose *candidate subgraphs* by enlarging known frequent subgraphs which are previously discovered lattice nodes. This amounts to an implicit or explicit traversal of the lattice.

The question then arises: what is the best order in which to discover nodes in the lattice. Broadly, there are two options breadth first search and depth first search. In the breadth first approach [71] lattice nodes are enumerated level by level. All subgraphs of one size are examined before examining subgraphs of a larger size. In the depth first approach [151] the enumeration follows the pattern of a depth first search over the lattice. Larger and larger

subgraphs are considered each time until there is no larger subgraph to consider. Then the algorithms backtrack to the next largest subgraph not yet examined.

When traversing the frequent subgraph lattice in a breadth first manner all nodes which represent subgraphs of a particular size must be held in memory at the same time. When lattices are much wider than they are deep (a common occurrence in many applications) this requires a large amount of memory. In contrast depth first traversal only needs to hold in memory the longest path in the lattice from root to edge. Even if the lattice is extremely deep a maximum number of edges could be specified making the depth first traversal practical. In contrast, there is no simple limitation or "fix" which allows breadth first miners to proceed when mining very broad lattices.

2.2.2 Support Computation

There are two approaches for computing the support of a candidate frequent subgraph. The first is to conduct *subgraph isomorphism tests* on each graph in the database until either the minimum support threshold k is reached or k cannot be reached [151]. The problem of finding a subgraph isomorphism mapping from a graph H into a graph G is NP-Complete [29]. If the graphs in question are labeled and not too many of the vertices share labels solving the subgraph isomorphism problem is not difficult. However, it does require building an index on the database which maps labels to vertices in each graph.

The second approach is to store all the *embeddings* (subgraph isomorphism mappings) for each frequent subgraph [18]. Then when a candidate subgraph is constructed, the list of embeddings of the candidate's subgraphs can be used to find all of the embeddings of the candidate without conducting expensive subgraph isomorphism tests. This approach works is much faster than the search approach when the total number of embeddings is small. However, if the total number is very large (and much larger than the minimum support threshold k) then conducting the explicit subgraph isomorphism tests is more advantageous.

2.2.3 Candidate Subgraph Generation

There are three approaches for creating candidate subgraphs. All of the approaches construct candidates by enlarging known frequent subgraphs. It is conceivable for an algorithm to blindly generate candidates by enumerating all possible subgraphs of the graphs in the database. However, the number of subgraphs of a graph is exponential to the size of the graph. Creating the candidates from previously discovered frequent subgraphs significantly prunes the search space (assuming most subgraphs are not frequent).

The first approach (inspired by the Apriori algorithm [5] for frequent itemset mining) is to join together two frequent subgraph with $n-1$ edges to form candidate subgraphs with n edges [66, 71]. The join operation may or may not produce graphs which are subgraphs of any graph in the database. These spurious candidates must be detected and filtered when counting the support. Any spurious candidate adds extra work so it is best to reduce the number of spurious candidates.

The second approach adds a single, frequent edge (and sometimes a vertex) to a known frequent subgraph [86]. The algorithms which take this approach pre-compute all of the frequent edges in the database to support this candidate generation strategy. Like the first approach, this approach may produce spurious candidates. However, it avoids the costly join operation which involves computing the overlap between two graphs.

The third approach adds a single, frequent edge (and sometimes a vertex) to a known frequent subgraph where the edge is found by consulting the embeddings of the known frequent subgraph [18]. This approach never computes spurious candidates as all candidates are computed by extending embeddings. This approach works well when the algorithm stores lists of embeddings instead of using subgraph isomorphism testing for support computation.

2.2.4 Elimination of Duplicate Subgraphs

As shown in Figure 2.2 most frequent subgraphs can be constructed in multiple ways. Thus, during a traversal of the frequent subgraph lattice it is important to detect when a particular node (subgraph) has been previously encountered. The most obvious approach to this problem is to conduct explicit *graph isomorphism* tests on all of the frequent subgraphs so far discovered.

Definition 2.5 (Graph Isomorphism)**.** *Given labeled digraphs G_1 and G_2, a* ***bijective*** *mapping $m : V_{G_1} \mapsto V_{G_2}$ is a* graph isomorphism *when:*

1. *m maps each vertex in G_1 to a vertex in G_2 with the same label:*
   $\forall\ v \in V_{G_1}\ [l_{G_1}(v) = l_{G_2}(m(v))]$
   $\forall\ v \in V_{G_2}\ [l_{G_2}(v) = l_{G_1}(m^{-1}(v))]$
2. *m preserves edges:*
   $\forall\ u, v \in V_{G_1}\ [(u, v) \in E_{G_1} \Leftrightarrow (m(u), m(v)) \in E_{G_2}]$
   $\forall\ u, v \in V_{G_2}\ [(u, v) \in E_{G_2} \Leftrightarrow (m^{-1}(u), m^{-1}(v)) \in E_{G_1}]$
3. *m preserves edge labels:*
   $\forall\ (u, v) \in E_{G_1}\ [l_{G_1}(u, v) = l_{G_2}(m(u), m(v))]$
   $\forall\ (u, v) \in E_{G_2}\ [l_{G_2}(u, v) = l_{G_1}(m^{-1}(u), m^{-1}(v))]$

The exact computational complexity class of the problem of finding the bijective graph isomorphism mappings is unknown. Graph isomorphism is in the class NP, as given a mapping m it obviously takes a polynomial number of steps to check (using the above definition) whether the mapping is a valid graph isomorphism mapping. However, it is currently unclear whether or not graph isomorphism is NP-Hard (and thus NP-Complete) [101]. On the whole, it is thought that graph isomorphism is *not* NP-Hard. László Babai has some important preliminary work showing graph isomorphism may be solvable in quasi-polynomial time [14]. However, Babai's work has not yet been fully vetted (one problem has been found and fixed so far) and it is too early to draw conclusions. Whatever the exact computational complexity class graph isomorphism belongs to, practical graph isomorphism algorithms such as `nauty`, `saucy`, `bliss`, and `traces` can all effectively solve the problem for a wide variety of real world graphs [101].

Conducting explicit graph isomorphism tests for every candidate subgraph against all graphs so far considered is prohibitively expensive. It is too expensive even if graph isomorphism was an "easy" problem. A graph has an exponential (in the size of the graph) number of subgraphs. Thus, as the mining process proceeds performing graph isomorphism tests will dominate the execution time of the algorithm.

A better way to check if a candidate subgraph has already been processed is to use *canonical ordering* [101]. The canonical order for a graph G is an ordering on the vertices V and the edges E such that all graphs which are isomorphic to G will have the same canonical ordering on their vertices and edges. Once two graphs have been placed in a canonical order the graph isomorphism problem can be solved in linear time by checking for a one-to-one correspondence between the ordered vertices and edges. All of the practical graph

```
1  Procedure ExtendFSGwithFreqEdges(𝒟, k, H)
       Param: Database of labeled digraphs 𝒟
       Param: Minimum support k
       Param: Known frequent subgraph H = (V_H, E_H, l_H)
       Result: Set of candidate frequent subgraphs C
2      for v ∈ V_H do
3          for ϵ = (u, w) ∈ E_G ∧ G ∈ 𝒟 ∧ |{g : G ∈ 𝒟 ∧ e ∈ V_g}| ≥ k ∧ l_H(v) = l_G(u) do
4              let c = H + ϵ;
5              Add c to 𝒞
6          end
7          for ϵ = (u, w) ∈ E_G ∧ G ∈ 𝒟 ∧ |{g : G ∈ 𝒟 ∧ e ∈ V_g}| ≥ k ∧ l_H(v) = l_G(w) do
8              let c = H + ϵ;
9              Add c to 𝒞
10         end
11     end
12     return 𝒞
```

Algorithm 2: Create all one edge extensions of a known frequent subgraph H.

isomorphism algorithms (`nauty`, `saucy`, `bliss`, and `traces` [101]) can compute the canonical orderings.

Once a graph has been placed in canonical order a *canonical labeling* for the graph can be computed [101]. The label is a string (sometimes called a certificate) which uniquely identifies the graph. The string is produced by encoding the canonically ordered vertex labels, the edges, and edge labels as a string. Once the canonical label has been produced it can be stored in a lookup table such as a hash table. Then checking to see if a candidate subgraph has been processed only requires computing the canonical label for the candidate and checking for membership in the lookup table.

2.2.5 Depth First Search Tree

The best approach for creating candidate subgraphs is by adding an edge at a time to a known frequent subgraph. However, in general a candidate frequent subgraph must be created for every edge which could be added to the known frequent subgraph. This is demonstrated by Algorithm 2 which extends a known frequent subgraph with frequent edges. The algorithm creates extensions for every vertex in the known frequent subgraph.

Creating candidate subgraphs from every vertex for every frequent subgraph ensures that all frequent subgraphs will eventually be discovered. However, it also results in many frequent subgraphs being "discovered" multiple times. Using canonical labeling to solve the graph isomorphism problem allows duplicates to be filtered out. However, it would be even better if the miner could avoid exploring lattice nodes more than once.

One way to avoid discovering lattice nodes over and over again is to pre-define a *depth first search tree*. A depth first search tree of the lattice identifies a unique path in the frequent subgraph lattice from the root lattice node to each lattice node. Normally in a depth first search the tree edge into a node is the first edge explored which reaches that node. When the tree edges are pre-defined the search will skip exploring certain edges and wait to discover certain nodes until they can be discovered from their predefined tree edges.

This predefinition of the depth first search tree is the *canonical depth first search tree* for the frequent subgraph lattice.

There are two methods for identifying the tree edges and both rely on properties of the canonical labeling. The first method identifies the *generating parent* for each lattice node [87]. The generating parent for a graph H with k edges is a subgraph $\bar{H} \sqsubset H$ with $k-1$ edges. The graph H has multiple subgraphs $H' \sqsubset H$ with $k-1$ edges but only one of them is the generating parent $\bar{H}$. To construct $\bar{H}$ arrange H in canonical order (using any canonical ordering algorithm). Then remove the last edge in H such that the subgraph is not disconnected. This subgraph is the generating parent. The generating parent can then be used to identify the tree edges of the canonical depth first search tree. Let A be a frequent subgraph and let a candidate subgraph be C constructed by adding an edge to A (e.g. $C = A + \epsilon$). To check if (A, C) is a tree edge construct the generating parent $\bar{C}$ for C. If and only if A is isomorphic to $\bar{C}$ ($A \cong C$) then (A, C) is a tree edge.

The second method defines a new canonical labeling algorithm called Minimum DFS Codes with is the key part of the gSpan miner [151]. gSpan defines a lexicographic ordering on the depth first search (DFS) tree of the **subgraphs** (not the lattice, an important and subtle point). The lexicographic ordering allows a unique minimal depth first search tree (of the subgraphs) to be identified. They encode this tree into a string: the *Minimum DFS Code* for the graph (which serves as the canonical label). In gSpan when a candidate graph C is constructed from a known frequent subgraph A the DFS Code for C is computed. If C's code is the minimal DFS code, then (A, C) is a tree edge.

Minimum DFS Codes computations are not competitive with other methods for solving the graph isomorphism problem [101]. However, gSpan's approach is generally faster than finding the canonical parent because it enables skipping some of the candidate subgraph generation. Recall, in Algorithm 2 extensions to a frequent subgraph H were created for every vertex $v \in V_H$. gSpan skips some of the vertices by extending only from vertices that fall on the *rightmost path* of H's minimum DFS tree (called the *rightmost extension*). Normally, it would not be sound to skip creating extensions from some of the vertices. However, the lexicographic ordering defined by gSpan ensures this optimization is sound.

2.3 Mining Connected Graphs

The previous section focused on the mining transactional datasets where the database $\mathcal{D}$ being mined consisted of many small graphs. However for important software engineering applications (such as code clone detection) the database is a single large graph G. This small change has big consequences in frequent subgraph mining as there are multiple definitions of "frequent." To deal with these definitions, we introduce a *support measure* σ which returns the frequency of a given subgraph H in a graph G.

Definition 2.6 (k-Frequent Subgraph Mining a Graph G)**.**
Given a support measure $\sigma : [\![\cdot]\!] \to \mathbb{N}^+$, a minimum frequency level $k \in \mathbb{N}$, and a labeled digraph G, find all subgraphs $H \sqsubseteq G$ s.t. $\sigma([\![H]\!]_G) \geq k$.

Definition 2.7 (Connected Subgraph Lattice of G)**.** *The subgraph relation $\cdot \sqsubseteq \cdot$ induces the* Connected Subgraph Lattice $\mathcal{L}_G$ *representing all the possible ways of constructing the graph G from the empty subgraph by adding one edge at a time. $\mathcal{L}_G$ is a digraph where each vertex $u \in V_{\mathcal{L}_G}$ represents a unique connected (ignoring edge direction) subgraph of G. There is an edge from u to v in $E_{\mathcal{L}_G}$ if adding some edge ϵ to u creates a subgraph $u + \epsilon$ isomorphic to v, $v \cong u + \epsilon$.*

Definition 2.8 (k-Frequent Connected Subgraph Lattice of G)**.** *A* k-Frequent Connected Subgraph Lattice k-$\mathcal{L}_G$ *is a connected subgraph lattice containing only those subgraphs having frequency at least* k *in* G *according to some support measure* σ*.*

2.3.1 Counting Support

When mining a large connected graph a support measure which computes the "frequency" of the graph is required. The simplest support measure is then number of unique embeddings in H's subgraph isomorphism class, $\sigma_\times(\llbracket H \rrbracket_G) = |\llbracket H \rrbracket_G|$. Unfortunately, $\sigma_\times$ does not satisfy a certain desirable *Downward Closure Property* (DCP) [19].

Definition 2.9 (Downward Closure Property)**.**
If A *is a subgraph of* B *then the support of* A *is greater than or equal to the support of* B*.*

$$A \sqsubseteq B \rightarrow \sigma(\llbracket A \rrbracket) \geq \sigma(\llbracket B \rrbracket)$$

DCP ensures all frequent subgraphs can be discovered by "extending" a smaller frequent subgraph by adding frequent edges. Furthermore, it ensures the order in which the edges are added doesn't make a difference. Thus, DCP ensures the k-$\mathcal{L}_G$ is connected.

Due to this problem with $\sigma_\times$, a number of support measures have been proposed [19, 73, 87]. An intuitively appealing measure, denoted $\sigma_\bullet$, is the number of *Maximal Independent (non-overlapping) Subgraphs* (MIS) [19].

Definition 2.10. *Given a graph* G *and subgraph* H*, the* Maximum Independent Subgraphs (MIS) *is the number of distinct subgraph isomorphism mappings of* H *into* G *which do not overlap. Let* $\llbracket H \rrbracket_G$ *be the set of all mappings, then MIS is*

$$\sigma_\bullet = \max |\{m_1, m_2, ..., m_k \ s.t. \ \forall \ i \leq k \, [m_i \in \llbracket H \rrbracket_G] \wedge \forall \ i, j \leq k \, [m_i(V_H) \cap m_j(V_H) = \emptyset]\}|$$

However, the problem of computing MIS is NP-Complete, and in practice using MIS is not a feasible support metric due to its computational overhead. We generally use *Minimum Image Support* [19], as it is both efficient to compute, satisfies DCP, and has been widely adopted [19, 25, 39, 139].

Definition 2.11. *Given a graph* G *and subgraph* H*, the* Minimum Image Support (MNI) *is the least number of distinct nodes in* G *that a node in* H *is mapped to.*

$$\sigma_\wedge(\llbracket H \rrbracket_G) = \min_{v \in V_H} |\{m(v) \ : \ m \in \llbracket H \rrbracket_G\}|$$

To compute MNI, find the set of all subgraph isomorphism mappings $\llbracket H \rrbracket_G$ of H into G. For each vertex $v \in V_H$ compute the unique locations mapped $m(v)$ by all mappings $m \in \llbracket H \rrbracket_G$. Some of the subgraph isomorphism mappings may overlap for a given vertex $v \in V_H$ such that $m_1(v) = m_2(v)$ with $m_1, m_2 \in \llbracket H \rrbracket_G$. MNI only counts one of the images for $m_1(v)$ and $m_2(v)$ not both. The final support for the subgraph H is the minimum number of the unique locations mapped for any vertex.

Chapter 3

REGRAX: Extracting Low-Frequency Recurring Subgraphs from Large Sparse Graphs with Subgraph Matching

3.1 Introduction

Mining recurring patterns in graph representations of program code [22, 24, 34–37, 59, 60, 72, 83, 85, 93, 94, 103, 106, 108, 111, 156] is a powerful approach to solving some challenging problems in software engineering. For instance, frequent subgraph mining (FSM) is used for finding duplicated code (code clones) [59, 83, 85], implicit programming rules [22], and bug patterns [135]; it is also used for localizing faults [24, 34–37, 94, 103, 111, 156], detecting plagiarism [93], and automatically completing fragments of program code [106]. An important example of the kind of graphs that are mined for such purposes is the *Program Dependence Graph* (PDG) [46]. A PDG is a sparse, labeled, directed graph in which the vertices represent program instructions, statements, or basic blocks. The edges are labeled and represent either data or control dependences between operations (see Figure 4.1).

Our experience revealed several properties of PDGs that make them difficult to analyze with FSM. Some PDGs contain subgraphs with many instances that overlap either fully (automorphism) or partially in the PDG. We have observed as many as 16 million instances of a single pattern. Since a software engineer is usually interested in at most one of the overlapping instances, finding all of them is not useful. However, in order to accurately count the *support* of a pattern [19] all of the overlapping instances must be found (Section 2.3.1).

To address the challenge of mining PDGs and other real world graphs, this paper presents REGRAX (Recurring GRAph eXtractor), a new framework for efficiently mining and sampling frequent subgraphs. REGRAX is very efficient: in an empirical evaluation described in Section 3.3, REGRAX outperformed GraMi [39], a recent high performance frequent

subgraph miner for large connected graphs. Each tool's performance was evaluated on multiple real world datasets, including PDGs of large programs, several of the SUBDUE [27] datasets, and synthetically generated graphs. REGRAX outperformed GraMi by an order of magnitude.

REGRAX's improved performance comes from several new optimizations: *Unsupported Extension Pruning*, *Overlap Pruning*, and *Pruning Subgraph Matches using the Greedy Independent Subgraphs Support Metric*.[1] As these optimizations have different effects on performance they are evaluated both individually and in combination. Finally, the implementation is *parallelized*. The parallelization provides an 8× speedup on the machine used for the empirical evaluation. All of the comparisons of REGRAX with GraMi in Section 3.3 are done with the parallelization disabled to provide a fair comparison as GraMi does not utilize the multiple processor cores as efficiently as REGRAX.

When mining PDGs or other program graph representations, software engineers are often interested in finding subgraphs that recur just a few times. For instance, in code clone studies [59, 60, 83, 85] it is not uncommon for a pattern to be considered a possible clone if it occurs in as few as two distinct locations in a program. We have found that with low frequency-thresholds (low support), large PDGs may have a huge number of distinct "frequent" subgraphs. For example, we found over 500 million 5-frequent subgraphs in jGit (see Table 5.1 on page 49) before the mining process was stopped after 7 days.

One approach to overcoming this so called "computational bottleneck" [25] is to *sample* frequent subgraphs [7, 8, 23, 59, 60] instead of searching for all of them. REGRAX has two modes – (1) complete mining, in which it finds all of the frequent subgraphs and (2) sampling, in which it randomly selects a representative subset of the frequent subgraphs. Both modes are evaluated in Section 3.3 along with the proposed optimizations and support metrics. The evaluation shows the that complete mining mode outperforms GraMi [39].

Evaluation of the sampling mode shows it is practical to extract frequent subgraphs from large programs with very low frequency thresholds (2 and 5). Some of the programs contain patterns with many overlapping instances, which required a new, unsound support measure, called *Greedy Independent Subgraphs* (GIS) (Section 3.2.5), to obtain results quickly (< 2 minutes). Previous work, such as GraMi, is not able to successfully mine such graphs at very low support levels.

Summary of Contributions

1. New optimizations: *Unsupported Extension Pruning*, *Overlap Pruning*, and an indexed *Subgraph Matching* algorithm integrating information from the pattern mining process (Sec. 3.2).
2. New support metrics: *Fully Independent Subgraphs* (FIS) and *Greedy Independent Subgraphs* (GIS) for dealing with difficult-to-mine patterns (Sec. 3.2.5).
3. An empirical study on the performance and scalability of the overall system (in both complete mining and sampling modes), the contributions of each optimization, and the effect of GIS and FIS (Sec. 3.3).

3.2 Mining Frequent Subgraphs

Recall from Chapter 2 that at a high level all frequent subgraph mining algorithms are traversals of the k-frequent connected subgraph lattice of the graph G being mined (denoted

[1] The Greedy Independent Subgraph Support Metric was briefly described as part of our workshop paper on software clones [60].

k-$\mathcal{L}_G$, see Def. 2.8). All methods start from the empty subgraph and propose *candidate subgraphs* of G that might be frequent, check their support (using a support measure) and propose further, larger candidate subgraphs until all vertices of k-$\mathcal{L}_G$ have been found.

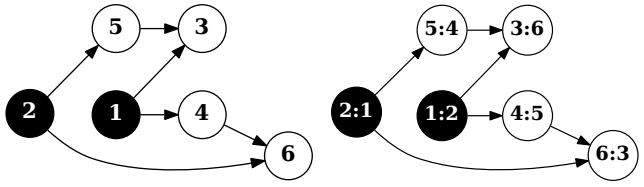
At a high level REGRAX conducts a depth first traversal of k-$\mathcal{L}_G$. Candidate subgraphs are proposed by extending embeddings (subgraph isomorphism mappings) of the subgraphs. However, the embeddings are not stored, only the proposed extensions are stored. To find the embeddings REGRAX uses a new fast subgraph matching algorithm with light weight indexing to measure the support of candidate subgraphs. The new subgraph matching algorithm collaborates with the traversal of k-$\mathcal{L}_G$ to prune the subgraph matching search space (called *Overlap Pruning*). The traversal has new optimization (*Unsupported Extension Pruning*) which allows it to avoid exploring structures which contain unsupported substructures. This is made possible by an explicit representation of the one-edge extensions which are added to known frequent subgraphs to generate candidate subgraphs.

REGRAX's approach is motivated by our desire to mine and analyze program dependence graphs to identify regions of duplicated code [59]. Program dependence graphs often contain substructures with overlapping instances (embeddings, subgraph isomorphism mappings) which contain automorphisms. A graph G has an automorphism when it has a non-trivial subgraph isomorphism mapping to itself. Figure 3.1 shows a simple example. In the figure, each vertex in 3.1b maps to a different vertex in 3.1a.

Automorphisms and overlapping embeddings cause commonly used support measures for connected frequent subgraph mining (such as MNI [19]) to double count subgraphs which refer to the same section of code. We have found, in real world programs, frequent substructures with over 15 million unique subgraph isomorphism mappings into the program dependence graph. Only a few (< 10) of those mappings refer to distinct locations in the code. REGRAX solves this problem by introducing a new support measure (*Greedy Independent Subgraphs*) which greedily prunes the subgraph matching search space. This aggressive pruning does not sacrifice sensitivity to duplications which occur in distinctly different areas of the graph but does (intentionally) reduce sensitive towards subgraphs whose embeddings overlap each other.

REGRAX's approach is outlined in Listing 3.1 which is in Python for brevity but the implementation is in Go. The function `mine_frequent_subgraphs` starts at the empty subgraph and enumerates the *canonical depth first search tree* of the k-$\mathcal{L}_G$ (see Section 2.2.5). Nodes of the k-$\mathcal{L}_G$ are modeled by the `LatticeNode` class (partially shown in Listing 3.2). Duplicate subgraphs are eliminated by employing a canonical ordering algorithm (BLISS [77]) to determine if a subgraph has been previously visited (see Section 2.2.4).

The heart of the algorithm is the function `supergraphs` in Listing 3.2 which produces the child nodes of the current k-$\mathcal{L}_G$ node. It does this by finding all of the *candidate subgraphs*



(a) Colored digraph (b) An Automorphism

Figure 3.1: The graph in 3.1b is a rotation (automorphism) of the graph in 3.1a. In 3.1a the numbers are the vertex identifiers. In 3.1b there are 2 numbers on each vertex read as: "subgraph vertex id n maps to graph vertex id m."

of G (lines 17, 23-27) which may be frequent and are 1-edge super-graphs of the graph represented by the current lattice node. It then determines if each candidate subgraph is frequent and if it generates a `LatticeNode` to represent it.

```
# param G: the graph G being mined
# param min_support: the minimum support for a graph to be frequent
# yields a sequence of frequent subgraphs
def mine_frequent_subgraphs(G, min_support):
  stack = list()
  stack.append(G.root_lattice_node())
  while len(stack) > 0:
    n = stack.pop()
    yield n
    for supergraph in n.supergraphs():
        stack.append(supergraph)
```

Listing 3.1: High level overview of REGRAX.

```
class LatticeNode(object):
    def __init__(self, G, sg, embs, exts):
        self.graph = G     ## The graph G = (V, E)
        self.subgraph = sg ## A subgraph of G which this lattice node represents
        self.supported_embeddings = embs  ## embeddings of sg into G
        self.extensions = exts ## extensions to sg to create candidate frequent
                               ## subgraphs an extension is a source vertex, a
                               ## target vertex, and an edge.

    # returns a sequence of children (direct supergraphs) on the canonical depth
    # first search tree
    def supergraphs(self, allow):
        exts = self.extend()
        for supergraph in exts:
            if not self.is_canonical(supergraph):
                continue
            support, exts, embs = exts_and_embs(self.graph, supergraph)
            if support >= min_support:
                yield LatticeNode(self.graph, supergraph, embs, exts)

    ## Uses the extensions stored in the lattice node to construct supergraphs
    ## of the current node.
    def extend(self):
        supergraphs = set()
        for ext in self.extensions:
            supergraphs.add(self.subgraph.extend(ext))
        return supergraphs

    ## returns True iff. the current subgraph (self) is the "generating
    ## parent" of the child. (see Section 2.2.5)
    def is_canonical(self, supergraph):
        e = len(supergraph.Edges) - 1
        h = supergraph.remove_edge(e)
        if not h.connected():
            if e <= 0:
                return False
            e -= 1
            h = supergraph.remove_edge(e)
        return bliss.canonical_label(h) == self.canonical_label()

    ## returns a canonical label for this graph based on the canonical ordering
    ## algorithm BLISS [77].
    def canonical_label(self):
        return  bliss.canonical_label(self.subgraph)
```

Listing 3.2: REGRAX's object oriented model of the k-$\mathcal{L}_G$.

3.2.1 Candidate Subgraph Generation

Candidate frequent subgraphs are graphs which are potentially frequent subgraphs in G. REGRAX implements two methods for candidate subgraph generation. In both methods, each candidate subgraph is obtained from a known frequent subgraph by adding one edge. Given a frequent subgraph A, all the graphs $A + \epsilon$ obtained by adding one edge ϵ to A are candidate subgraphs. Note that although A is a frequent subgraph of G, $A + \epsilon$ may not even be a subgraph of G. Section 2.2.3 described the two approaches adopted by REGRAX: *extending using frequent edges* and *extending using embeddings*. These two approaches are contrasted in Listings 3.3 and 3.4. As shown in the empirical evaluation (see Section 3.3), neither approach is better on all datasets. However, for program graphs (which are the graphs of interest for this paper) extending using the embeddings seems to be the more advantageous strategy.

3.2.2 Unsupported Extension Pruning

Every candidate generation strategy inevitably produces subgraphs that are not frequent in G. Such a non-frequent candidate $A + \epsilon$ is produced from a frequent subgraph A and an extending edge ϵ anchored at at least one vertex in A. Any graph B that is a supergraph of A can be extended with the edge ϵ creating a graph $B + \epsilon$. However, since $A + \epsilon$

```
# param G: the graph G being mined
# param sg: a subgraph of G.
# Computes the embeddings and the extensions for the subgraph sg. The extensions
# are computed from known frequent edges in G.
def exts_and_embs(G, sg):
    embs = list() # embeddings of sg into G
    for emb in find_embeddings(G, sg):
        embs.append(emb)
        if support(embs) >= min_support: break
    return support(embs), freq_edge_exts(G, sg), embs

# param G: the graph G being mined
# param sg: a subgraph of G.
# Computes the set of extensions from the subgraph sg which extend sg by 1 edge
# to create direct supergraphs. The extensions are represented by: (source
# vertex, target vertex, and an edge).
def freq_edge_exts(G, sg):
    exts = set() # a set of extensions
    ## create extensions for each vertex in sg.
    for u in sg.vertices:
        ## iterate over the frequent edges in G which have a source vertex with
        ## the same color as u.
        for edge in G.freq_edges_from_color(u.color):
            ## check for vertices in sg which match the target vertex of the edge.
            for v in sg.vertices:
                if G.color_of(edge.targ) = v.color:
                    exts.add(Extension(u, v, edge))
            exts.add(Extension(
                u, Vertex(len(sg), G.color_of(edge.targ)), edge))
        ## iterate over the frequent edges in G which have a target vertex with
        ## the same color as u.
        for edge in G.freq_edges_to_color(u.color):
            exts.add(Extension(
                Vertex(len(sg), G.color_of(edge.src)), v, edge))
    return exts
```

Listing 3.3: Compute the extensions from frequent edges.

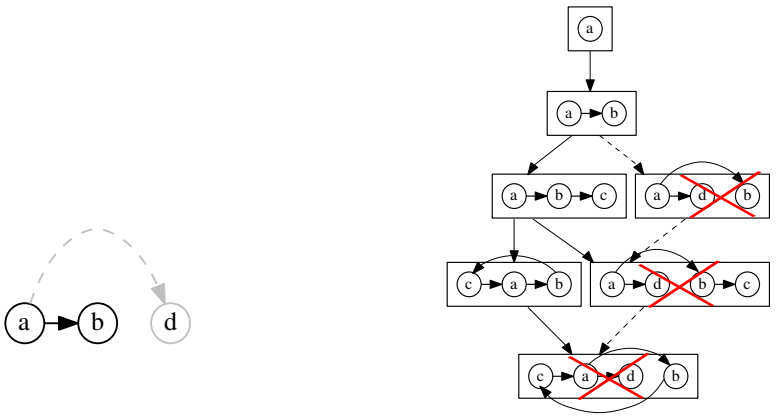
is a subgraph of $B + \epsilon$ and $A + \epsilon$ is not frequent then $B + \epsilon$ is also not frequent. Thus, if a particular extension ϵ leads to a non-frequent subgraph of G then that extension can be remembered and used to prune subsequent candidates. We call this novel optimization *Unsupported Extension Pruning* and while it is not shown in Listing 3.2, for brevity, it is straightforward to implement.

Figure 3.2 shows a visual example of how extension pruning works. On the left, a candidate frequent subgraph is shown which did not turn out to be frequent. REGRAX tracks the extension which created the unsupported candidate (shown in gray with a dashed

```
# param G: the graph G being mined
# param sg: the subgraph.
# Computes the embeddings and the extensions for the subgraph sg. The extensions
# are computed from the embeddings.
def exts_and_embs(G, sg):
    embs = list() # list of embeddings (subgraph isomorphism mappings) of sg in G
    exts = set() # set of one edge extensions of sg which create supergraphs of sg
                 # these supergraphs become candidate frequent subgraphs.
    ## Find all of the embeddings (subgraph isomorphism mappings) of sg in G
    for emb in find_embeddings(G, sg):
        ## Each embedding maps each vertex u in sg.vertices to a vertex in G.
        ## iterate over the mapped vertices (emb_idx)
        for emb_idx in embs.idxs:
            ## Using an index on G, look up edges that start from the vertex.
            for edge in G.children_of(emb_idx):
                add_ext(G, exts, emb, edge, emb_idx, -1)
            ## Using an index on G, look up edges that end at the vertex.
            for edge in G.parents_of(emb_idx):
                add_ext(G, exts, emb, edge, -1, emb_idx)
        embs.append(emb)
    return support(embs), exts, embs

# param G: the graph G being mined.
# param exts: The set of extensions.
# param emb: the current embedding.
# param edge: the edge from G which is going to become a new extension
# param src: the source vertex identifier in the subgraph
# param targ: the source vertex identifier in the subgraph
# Adds an extension (source vertex, target vertex, edge) an extisting set of
# extensions.
def add_ext(G, exts, emb, edge, src, targ):
    ## This algorithm needs to take the edge (which is from G) and map it onto
    ## the subgraph which is represented by the embedding (emb). It looks in the
    ## embedding to find matching vertices to the end points (src, targ) of the
    ## edge. It then creates an extension (ext) using that information.
    has_src = False; has_targ = False
    src_idx = len(emb); targ_idx = len(emb)
    if src >= 0:
        has_src = True; src_idx = src
    if targ >= 0:
        has_targ = True; targ_idx = src
    for sg_idx, emb_idx in emb:
        if has_src and has_targ: break
        if not has_src and edge.src == emb_idx:
            has_src = True; src_idx = sg_idx
        if not has_targ and edge.targ == emb_idx:
            has_targ = True; targ_idx = sg_idx
    ext = Extension(
            Vertex(src_idx, G.color_of(edge.src)),
            Vertex(targ_idx, G.color_of(edge.targ)),
            edge)
    if graph.frequency_of(ext) >= min_support:
        exts.add(ext)
```

Listing 3.4: Compute the extensions from the embeddings.



(a) A subgraph which is not frequent (unsupported) in the database.

(b) A portion of the depth first search tree looking for frequent subgraphs.

Figure 3.2: Unsupported extension pruning example.

edge). Using the "unsupported extension" REGRAX skips candidates which are created using the extension on the left. This is shown in the partial depth first search tree of the frequent subgraph lattice on the right as red cross-outs.

Lemma 3.1 (Unsupported Extension Pruning)**.**
Given a support counting function σ satisfying DCP, graphs $A \sqsubset B$, and an extending edge ϵ anchored to the same vertex v in both V_A and V_B such that: $A \sqsubset A + \epsilon$, $B \sqsubset B + \epsilon$, and $A + \epsilon \sqsubset B + \epsilon$. Then $\sigma(A + \epsilon) \geq \sigma(B + \epsilon)$.

Proof. Since $A + \epsilon$ is a subgraph of $B + \epsilon$ by the definition of DCP (Def. 2.9) $\sigma(A + \epsilon) \geq \sigma(B + \epsilon)$. □

3.2.3 Support Computation

There are two techniques to compute the support of a subgraph: (1) subgraph matching (as done by GraMi [39]) and (2) storing and growing the embeddings (as done by G-Miner [73]). It turns out that the store-and-grow approach is faster when no pattern has too many embeddings, because it avoids expensive subgraph matching computations. However, we have encountered pathological patterns, in dependence graphs of real programs, having more than 15 million embeddings. (The same patterns may have MNI support less than 10.) Such patterns often caused a frequent subgraph miner using store-and-grow to fail with an Out-of-Memory (OOM) error, even on a machine with 96GB of RAM. REGRAX implements both store-and-grow and subgraph matching, but due to space limitations only the subgraph matching mode is evaluated in Section 3.3.

We solve the subgraph matching problem with a new algorithm shown in Listing 3.5. Our new subgraph matching algoirthm proceeds as a backtracking tree-search procedure – in contrast to GraMi [39] which used a constraint solving approach. The tree nodes in the search correspond to partial subgraph isomorphism mappings (see Figure 3.3). They are represented by linked lists (called `IdNode` in the listing) allowing the reuse of partially correct mappings during the search without the need for new allocations.

```
# A linked list of vertex embeddings. Each node maps a subgraph vertex index
# (sg_idx) to a vertex in the graph G (emb_idx).
class IdNode(object):
    def __init__(self, sg_idx, emb_idx, prev=None):
        self.sg_idx = sg_idx
        self.emb_idx = emb_idx
        self.prev = prev
    ## Helper methods for iteration etc... omitted for brevity

# param G: the graph G being mined
# param sg: the subgraph to find embeddings of.
# param prune_fn : optional pruning function
def find_embedddings(G, sg, prune_fn=None):
    ## First, construct a spanning tree of the edges in sg.
    edges = spanning_tree(sg)
    for edge in sg.edges:
        ## Add the rest of the edges not in the spanning tree.
        if edge not in edges: edges.append(edge)
    stack = list()
    ## Embeddings for the source vertex of the first edge in the spanning tree.
    vembs = vertex_embeddings(G, sg, edges[0].src)
    ## Create a linked list of each of embeddings.
    for emb_idx in vembs:
        stack.append((IdNode(start_idx, emb_idx), 0))
    while len(stack) > 0:
        cur, eid = stack.pop() ## pop the next partial embedding and edge-id.
        if prune_fn is not None and prune_fn(cur):
            continue
        if eid >= len(edges):
            yield embedding_from_ids(cur)
        else:
            exts = extend_embedding(G,sg,cur,edges[eid])
            for next in exts:
                stack.append((next, eid+1))

# param cur: the current partial embedding of sg into G
# param edge: the edge to add to the current partial embedding
def extend_embedding(G, sg, cur, edge):
    src, targ = emb_idxs(cur, edge.src, edge.targ)
    if src > -1 and targ > -1:
        if G.has_edge(src, targ, edge.color):
            yield cur
    elif src > -1:
        targs = G.targs_from_src(src, edge.color, sg.color_of(edge.targ))
        for targ in targs:
            dok = degrees_ok(e.targ, targ)
            if dok and targ not in cur:
                yield IdNode(e.targ, targ, prev=cur)
    elif targ > -1:
        srcs = G.srcs_from_targ(targ, edge.color, sg.color_of(edge.src))
        for src in srcs:
            if degrees_ok(e.src, src) and src not in cur:
                yield IdNode(e.src, src, prev=cur)

# param embedding: a partial embedding (an IdNode list)
# param s_sg_idx: the source vertex in the subgraph
# param t_sg_idx: the target vertex in the subgraph
# Returns the indexes in the graph G for the source and target mapped by the
# current partial embedding. If no mapping exists then -1 is returned.
def emb_idxs(embedding, s_sg_idx, t_sg_idx):
    s_emb_idx = -1; t_emb_idx = -1
    for c in embedding:
        if c.sg_idx == s_sg_idx: s_emb_idx = c.emb_idx
        if c.sg_idx == t_sg_idx: t_emb_idx = c.emb_idx
    return s_emb_idx, t_emb_idx
```

Listing 3.5: The basic search algorithm for finding all subgraph isomorphisms.

The search starts by computing a spanning tree of edges for the subgraph. The choice of spanning tree is critical as some structures occur less frequently in the graph being searched than others. However, as previously noted [81] there is no known method for selecting the best spanning tree for all subgraph matching queries. The best spanning tree is the one which minimizes the runtime of the algorithm by minimizing the size of the subgraph matching search tree. Our algorithm for selecting the spanning tree chooses edges with higher *selectivity* first. The selectivity of an edge is the number of potential partial mappings created from the edge given the number of mappings created by the partial spanning tree. The fewer additional mappings an edge adds to the mappings for the partial spanning tree the higher the selectivity of the edge is. Empirically, this heuristic chooses a good (although not necessarily optimal) spanning tree most of the time.

The critical part of new algorithm is shown in function `extend_embedding`. Given a partial embedding (possibly containing a mapping for just one vertex in the subgraph) and a edge, the function `extend_embedding` in Listing 3.5 extends the embedding using the edge. It begins by finding the attachment points (lines 19, 37-42) of the edge onto the current embedding (`cur`). It then extends the embedding using the edge. There are three cases: (1) if both ends of the edge are in the current embedding (lines 21-23), (2) if the source (`src`) is in the current embedding (lines 24-29) and (3) if the target (`targ`) is in the current embedding (lines 30-35). To make this function fast, the data graph is indexed to allow a quick lookup of potential matching edges in the graph. If the added edge involves adding a new vertex to the partial mapping, the degrees of new vertices are checked (lines 28 and 34) to ensure they are greater than or equal to the expected degree in subgraph (as in NOVA [159]).

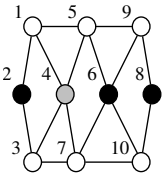
3.2.4 Overlap Pruning

Overlap Pruning is a new pruning strategy for subgraph matching which collaborates with the frequent subgraph mining algorithm. In frequent subgraph mining, all the subgraphs that are matched are built from smaller subgraphs whose embeddings were previously computed. Overlap pruning uses this observation to prune the search space for subgraph matching as shown in Figure 3.3.

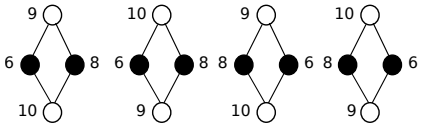
Given a graphs A and B which are both subgraphs of G. Let, B be a subgraph of A ($B \sqsubset A$) and be a frequent subgraph of G with known embeddings. Overlap pruning hinges on the observation that the only valid embeddings for a subgraph A all include embeddings for B. In overlap pruning, all the vertices shared between A and B are tracked. For each of the shared vertices the set of locations in the data graph G where they are embedded as part of B are recorded. These sets (called the *overlap*) prune potential embeddings of A that do not overlap with an embedding of B (see Figure 3.3).

Lemma 3.2 (Overlap Pruning)**.**
Let A and $A+\epsilon$ be subgraphs of G, where ϵ is an edge and $A \sqsubset A+\epsilon$. Let $[\![A]\!]_G$ be all of the subgraph isomorphism mappings from A into G. The set vertices in G mapped by the mappings $[\![A]\!]_G$ for a vertex $v \in V_A$ is denoted $[\![A]\!]_G(v) = \{m(v) : m \in [\![A]\!]_G\}$. Given the mappings $[\![A+\epsilon]\!]_G$ for $A+\epsilon$ into G, each vertex v in both V_A and $V_{A+\epsilon}$ satisfies $[\![A+\epsilon]\!]_G(v) \subseteq [\![A]\!]_G(v)$.

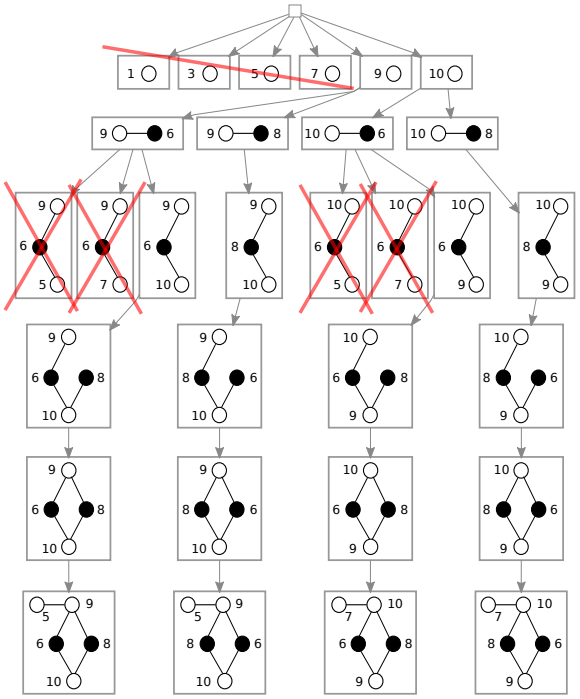
Proof. The statement holds since each mapping $m_{A+\epsilon \to G} \in [\![A+\epsilon]\!]_G$ includes a mapping $m_{A \to G}$ for A. If there was some mapping $m\prime_{A+\epsilon \to G}$ that did not including a mapping for A then there would be some edge in A not in $A+\epsilon$, but since $A+\epsilon$ is a super-graph of A all edges in A are in $A+\epsilon$ (a contradiction of the existence $m\prime_{A+\epsilon \to G}$). Thus, each vertex v



(a) The database being searched (colors are the labels, numbers are the vertex identifiers).



(b) Embeddings for the known frequent subgraph which is being extended (by adding one more edge to a new white node off the top of the diamond).



(c) The subgraph matching search tree with pruning shown as red cross outs.

Figure 3.3: Overlap pruning example.

in both V_A and in $V_{A+\epsilon}$ must satisfy $[\![A+\epsilon]\!]_G(v) \subseteq [\![A]\!]_G(v)$ because no mapping $m\prime_{A+\epsilon \to G}$ exists which could violate the statement. □

3.2.5 Dealing with Pathological Substructures

The primary application of REGRAX is for mining program dependence graphs (PDG)s. We have observed certain *pathological substructures* of the PDGs for real world Java programs which are very difficult to mine. The substructures contain many complete or partial *automorphisms* onto themselves. These automorphisms lead to millions of unique embeddings (subgraph isomorphism mappings) in the PDG. Although there are millions of embeddings only a few of them are of interest to software engineers. In applications such as Code Clone Detection [59] which finds instances of duplicated code, programmers only care to examine unique locations in the source code. A substructure with millions of embeddings which all map to the same location in the source code is uninteresting to the programmer.

Simply filtering out the automorphic embeddings as a post processing step is an attractive way to deal with this problem. However, finding or storing millions of embeddings for each candidate frequent subgraph is impractical and limits the application of frequent subgraph mining for software engineering applications. Unless frequent subgraph mining can scale up to large real world programs its impact on software engineering practice will be limited. To deal with this problem a new support measure *Greedy Independent Subgraphs* (GIS) is presented in this dissertation. GIS sacrifices satisfaction of the downward closure property (DCP) for subgraphs with automorphisms and overlapping embeddings in order to scale frequent subgraph mining to larger programs. The empirical study demonstrates both the necessity for GIS and quantifies its effect on the number of frequent subgraphs missed due to the violation of DCP.

To introduce GIS we will first develop a related measure *Fully Independent Subgraphs* (FIS). Recall the metrics introduced in Section 2.3.1. Those metrics were either slow to compute (such as MIS, $\sigma_\bullet$) or they allowed overlapping embeddings (such as MNI, $\sigma_\wedge$). What is needed for graphs that have frequent patterns with many automorphisms and overlapping embeddings is a support measure that is at least as restrictive as $\sigma_\bullet$ but much cheaper to compute. Therefore, we propose the *Fully Independent Subgraphs* (FIS) measure denoted by $\sigma_\triangle$.

Definition 3.1 (Embedding Graph)**.** *Given labeled digraphs H and G where $H \sqsubseteq G$, let each unique embedding (subgraph isomorphism mapping) $m \in [\![H]\!]_G$ be a vertex in an undirected, unlabeled embedding graph $\mathcal{E}_G^H$. Let there be an edge $(a,b) \in E_{\mathcal{E}_G^H}$ if the two embeddings $a, b \in V_{\mathcal{E}_G^H}$* overlap*, i.e., $\exists\ u, v \in V_H\ [a(u) = b(v)]$.*

Definition 3.2 (Fully Independent Subgraphs)**.**
The FIS support of a subgraph H of a labeled digraph G is the number of connected components in the embedding graph of H in G, $\mathcal{E}_G^H$.

$$\sigma_\triangle([\![H]\!]_G) = \left|\left\{c \mid c \text{ is a connected component of } \mathcal{E}_G^H\right\}\right|$$

FIS $\sigma_\triangle$ (in comparison with other proposed measures) is at least as restrictive as MIS $\sigma_\bullet$.

Lemma 3.3 (FIS $\leq$ MIS)**.** *For all graphs $H \sqsubseteq G$:*

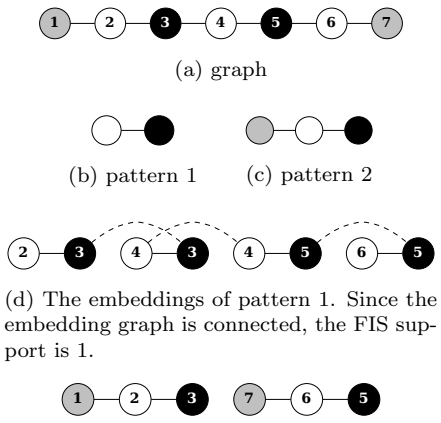
$$\sigma_\triangle([\![H]\!]_G) \leq \sigma_\bullet([\![H]\!]_G)$$

Proof. FIS $\sigma_{\triangle}$ is the number of connected components of the embedding graph $\mathcal{E}_G^H$. If every component has a single embedding, there are no overlapping embeddings and the MIS $\sigma_{\bullet}$ is equivalent to $\sigma_{\triangle}$. Consider the other case where there is one component $c \subseteq V_{\mathcal{E}_G^H}$ with multiple embeddings. Assume without loss of generality that all other components contain just one embedding. If c forms a clique in the embedding graph then the measures are equivalent since $\sigma_{\triangle}(c) = 1$ by Def. 3.2 and $\sigma_{\bullet}(c) = 1$ since no embedding in the clique is independent (non-overlapping) of any other embedding in the clique. However, if the embeddings in c do not form a clique there must be at least one pair of embeddings $u, v \in c$ such that there is no edge between them in the embedding graph $(u, v) \notin E_{\mathcal{E}_G^H}$. Then $\sigma_{\bullet}(c) \geq \sigma_{\triangle}(c)$ since $\sigma_{\bullet}$ may be ≥ 1 for a connected component but $\sigma_{\triangle}$ will always equal exactly 1 for a connected component. □

Lemma 3.4 (FIS does not satisfy DCP)**.** *There exists a labeled digraph G and subgraphs A and B of G such that $A \sqsubset B$ and $\sigma_{\triangle}([\![A]\!]_G) < \sigma_{\triangle}([\![B]\!]_G)$, violating DCP (Def. 2.9).*

Proof. This proof is by example. Consider the graph in Figure 3.4. The first pattern (a black vertex connected to a white vertex) has 4 embeddings. However, they are all connected together as part of the embedding graph (the dotted edges) so the support under FIS is 1. Pattern 1 is a super-graph of pattern 2. Pattern 1's support under FIS is 2 as its embedding graph is disconnected, proving the lemma. □

As a consequence of Lemma 3.3, FIS is more restrictive than MIS and MNI. Unlike MIS, computing FIS is tractable and can be done *online* as embeddings for a particular subgraph are found. As a consequence of Lemma 3.4 the FIS metric does not satisfy the downward closure property. Since the downward closure property is not satisfied, the frequent subgraph lattice for a graph may contain holes (missing vertices) or be disconnected (with part of the lattice unreachable from the root lattice node). These irregularities cause subgraph miners using FIS to miss potential subgraphs that are frequent under FIS. As illustrated in Figure 3.4, DCP violations occur when the subgraph A from the Lemma 3.4, which is a subgraph



(a) graph

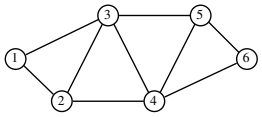
(b) pattern 1

(c) pattern 2

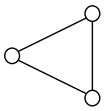
(d) The embeddings of pattern 1. Since the embedding graph is connected, the FIS support is 1.

(e) The embeddings of pattern 2. Since there is no overlap FIS support is 2.

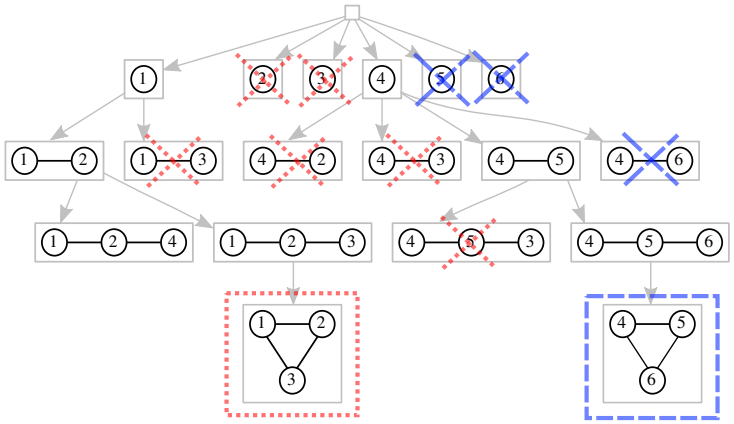
Figure 3.4: Proof by example for Lemma 3.4



(a) The database being searched (numbers are the vertex identifiers).



(b) Pattern to look for.



(c) The subgraph matching search tree with gis pruning cross outs.

Figure 3.5: GIS pruning example.

of B, has overlapping embeddings that are not included in the embeddings of B. If finding those subgraphs is critical for a particular application, FIS should not be used.

Listing 3.6 shows the Greedy Independent Subgraphs (GIS) pruning strategy. Figure 3.5 shows an example for a small undirected graph. GIS is an approximation of the FIS support measure. Like FIS it computes the number of connected components in the embedding graph of a subgraph H. However, it does so in a greedy fashion. During the subgraph matching process, if a subgraph vertex in a partial mapping maps to a vertex in the data graph that has been found in a complete embedding then the partial mapping is discarded. This is

```
1  ## This higher order function produces a pruning function for find_embeddings
2  ## (see Listing 3.5). Takes a set (seen) which should track vertices of G
3  ## which have appeared in some previously found embedding.
4  def gis_pruner(seen):
5      def gis_prune(cur):
6          ## for the current embedding (cur), check if any of the vertices
7          ## (emb_idx) have appeared in a previously discovered embedding
8          for n in cur:
9              if n.emb_idx in seen:
10                 ## If they have, add all vertices of G in cur to seen, and
11                 ## return true to indicate "cur" should be pruned
12                 for m in cur:
13                     seen.add(m.emb_idx)
14                 return True
15         return False
16     return gis_prune
17
18 ## A drop in replacement for the find_embeddings function (see Listing 3.5)
19 ## which implements GIS pruning.
20 def gis_pruning(G, sg):
21     seen = set()
22     embs_it = find_embeddings(G, sg, gis_pruner(seen))
23     for emb in embs_it:
24         for emb_idx in emb.idxs: seen.add(emb_idx)
25         yield emb
```

Listing 3.6: Pruning the embedding search space with GIS.

implemented by passing the `gis_prune` function into the `find_embeddings` function (from Listing 3.5) as the `prune_fn` parameter. By discarding a partial mapping, a portion of the subgraph matching search tree is pruned. GIS pruning can lead to significant reduction in effort, but at the cost of over-approximating FIS (and thus computing a higher support number than FIS). Like FIS, GIS does not satisfy DCP and should only be used when it is not feasible to mine a particular graph under MNI at the desired support setting.

3.2.6 Sampling Frequent Patterns

For large program graphs, finding all frequent subgraphs at low support is not feasible. Even if all of the frequent subgraphs are located, analyzing or storing them requires too much effort, because they can number in the hundreds of millions (or more) for large program dependence graphs. Sampling frequent subgraphs using an *unweighted forward random walk* (URW) (see Listing 3.7) has proven to be a fast effective method [23, 59, 60] for collecting a diverse set of maximal frequent subgraphs. In the empirical evaluation in Section 3.3, REGRAX's suitability for sampling low-frequency recurring subgraphs is demonstrated using the URW algorithm.

3.3 Empirical Evaluation

The new frequent subgraph miner, REGRAX, was evaluated in an empirical study. REGRAX[2] is implemented in the Go programming language and is parallelized. Four primary questions were examined:

1. How quickly can REGRAX mine program dependence graphs from large programs? (Table 5.1)
2. How does the performance of REGRAX compare to GraMi [39], a recent high performance frequent subgraph miner? (Table 3.2)
3. What is the effect of the novel optimizations and support measures on its performance? (Tables 3.3, 3.4, 3.5)
4. How quickly can REGRAX collect a sample of low frequency recurring subgraphs from the program dependence graphs of large programs? (Figure 5.2)

The evaluation was conducted on a dual socket server with two 2010 Intel Xeon CPUs (model # X5650). The server had 96 GB of RAM and a 1 TB RAID array. Despite the large amount of RAM available, REGRAX used at most 10 GB of RAM. GraMi failed to

[2]https://github.com/timtadh/regrax

```
1  # param G: the graph G being mined
2  # param min_support: the minimum support for a graph to be frequent
3  # yields a random frequent subgraph
4  def unweighted_random_walk(G, min_support):
5      prev = cur = G.root_lattice_node()
6      while cur is not None:
7          n = random_choice(cur.supergraphs())
8          prev = cur
9          cur = n
10     return prev
```

Listing 3.7: Unweighted Random Walk

Dataset	Type	Nodes	Edges	Labels	Index Time	k	Patterns	Time (sec)	Options
Credit	SUBDUE	14,750	14,000	79	0.72 ± 0.10	150	3,511	5.01 ± 0.11	*mxot*
						100	11,730	11.69 ± 0.15	*mxot*
						50	74,036	53.70 ± 1.04	*mxot*
Aviation	SUBDUE	101,185	135,000	6,225	5.45 ± 0.13	2200	309	9.20 ± 0.13	*met*
						2000	856	14.34 ± 0.21	*met*
						1500	5,247	50.70 ± 0.36	*met*
ExprCalc	PDG	1,110	2,162	369	0.17 ± 0.01	4	4,338	1.49 ± 0.12	*mxot*
						3	52,456	9.08 ± 0.90	*mxot*
Zookeeper	PDG	17,028	32,691	4,313	2.17 ± 0.04	12	26,383	12.73 ± 0.45	*mxot*
						10	74,980	27.37 ± 1.38	*mxot*
DDH	PDG	36,384	65,874	10,110	4.66 ± 0.06	25	1,388	6.24 ± 0.07	*mxot*
						20	2,666	7.48 ± 0.07	*mxot*
BCEL	PDG	52,731	108,542	9,781	6.73 ± 0.05	150	2,845	10.52 ± 0.11	*mxot*
						100	15,739	33.91 ± 0.22	*mxot*
jGit	PDG	136,716	300,550	25,074	17.56 ± 0.14	100	3,897	50.87 ± 0.80	*mxot*
Tomcat	PDG	388,098	829,801	68,506	50.38 ± 0.48	1000	141	53.54 ± 0.55	*mxot*
hBase	PDG	442,063	981,577	89,733	58.98 ± 0.58	200	27,305	735.52 ± 7.80	*mxot*
OrientDB	PDG	1,958,639	3,618,892	356,323	224.1 ± 19.5	1500	3,214	435.17 ± 30.73	*mxot*
Random-1	Random	150	224	11	0.04 ± 0.01	2	97	$0.073 \pm .006$	*mxot*
Random-2	Random	150	5,303	11	0.10 ± 0.01	21	33	19.33 ± 0.68	*met*

Table 3.1: REGRAX's performance using the MNI support measure and 24 threads. k is the minimum frequent (support) setting used. The times are reported as mean ± standard deviation and were computed from 50 independent runs of the program.
Options: m: MNI metric, f: FIS metric, g: GIS metric, x: extend from embeddings, e: extend from frequent edges, o: overlap pruning, t: extension pruning,

Dataset	k	R. # Patterns	R. Time (sec)	G. # Patterns	G. Time (sec)	R. Options
Credit	500	33	1.05 ± 00.02	24	6.90	*mxot*
	200	1,348	15.03 ± 00.13	1,325	305	*mxot*
Aviation	2200	309	31.16 ± 00.46	297	9.6	*met*
	2000	856	71.33 ± 01.31	843	59.2	*met*
	1500	5,247	379.33 ± 09.78	—	killed after 2 days	*met*
ExprCalc	4	4,338	6.29 ± 00.40	4,275	44	*mxot*
Zookeeper	25	842	6.16 ± 00.09	842	37	*mxot*
DDH	50	375	6.55 ± 00.07	302	17	*mxot*
BCEL	150	2,845	31.86 ± 00.33	1,687	244	*mxot*
jGit	150	2,335	90.41 ± 01.43	1,283	1,231	*mxot*
Tomcat	1000	141	71.64 ± 00.96	131	150	*mxot*
hBase	500	2,575	775.39 ± 014.8	2,672	4,897	*mxot*
OrientDB	1500	3,214	$1,930.49 \pm 144.5$	—	OOM crash	*mxot*
Random-1	2	97	0.13 ± 0.006	382	4.2	*mxot*
Random-2	21	33	125.84 ± 00.89	—	killed after 10 days	*met*

Table 3.2: Comparison against GraMi [39]. For this table, REGRAX was run in single threaded mode, putting REGRAX at a slight disadvantage as GraMi used 2 threads. GraMi was run in "undirected mode" for all graphs, as its directed mode is much much slower (10x for most datasets). Since GraMi mined in undirected mode the number of patterns it found was sometimes greater than REGRAX. In comparison to Table 5.1, some of the support values k are increased because GraMi was very slow with those settings.

Dataset	k	mx Time	me Time	mxo Time	mxt Time	met Time
Credit	150	6.72 ± 0.14	8.75 ± 0.12	5.60 ± 0.09	5.93 ± 0.11	6.87 ± 0.08
Aviation	2200	145.74 ± 4.57	17.92 ± 0.38	144.04 ± 4.67	30.19 ± 1.92	9.28 ± 0.39
ExprCalc	3	11.57 ± 1.84	63.82 ± 16.4	10.45 ± 1.92	6.91 ± 0.40	26.96 ± 6.27
Zookeeper	25	3.50 ± 0.06	3.49 ± 0.08	3.21 ± 0.06	3.14 ± 0.05	3.18 ± 0.07
DDH	25	7.24 ± 0.11	10.05 ± 0.09	6.68 ± 0.11	6.72 ± 0.09	8.52 ± 0.08
BCEL	100	55.93 ± 0.40	58.04 ± 0.31	44.20 ± 0.35	41.08 ± 0.36	40.45 ± 0.19
jGit	100	82.80 ± 1.00	77.22 ± 0.70	62.56 ± 0.77	60.15 ± 0.70	51.60 ± 0.36
Tomcat	1000	53.82 ± 0.53	53.38 ± 0.78	53.54 ± 0.49	53.81 ± 0.51	53.53 ± 0.47
hBase	200	$1,336.9 \pm 13.6$	$1,289.6 \pm 26.3$	904.82 ± 9.26	978.23 ± 10.4	858.00 ± 12.3
OrientDB	1500	732.8 ± 42.2	814.5 ± 42.7	523.18 ± 38.1	543.87 ± 35.5	569.44 ± 20.2
Random-1	2	$.0664 \pm 0.0056$	$0.12 \pm .011$	0.0688 ± 0.0071	0.0674 ± 0.0048	0.1118 ± 0.0142
Random-2	21	6076.8 ± 122.7	76.66 ± 1.15	6915.3 ± 130.5	1384.8 ± 51.7	19.12 ± 0.60

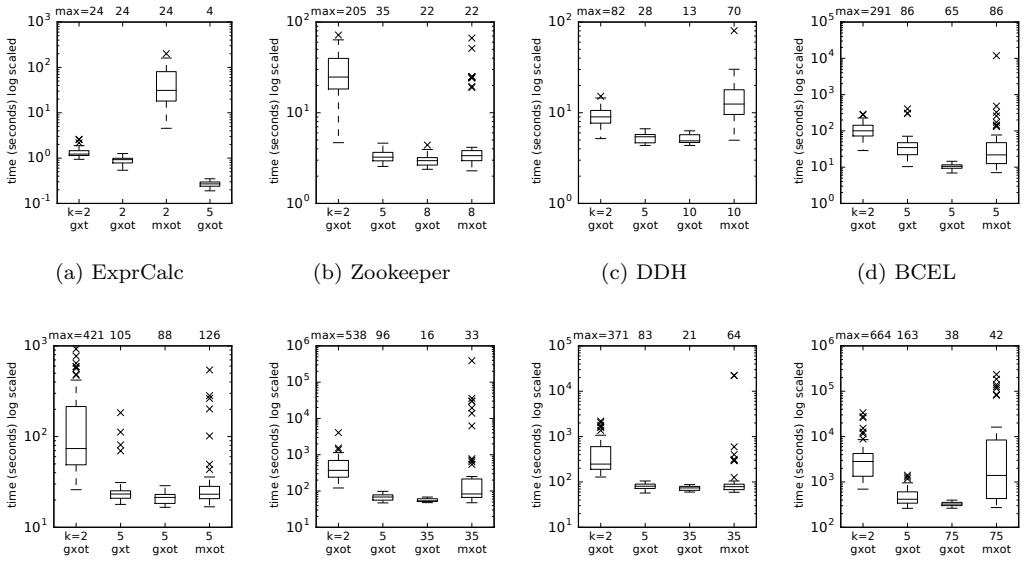
Table 3.3: Comparison of the optimizations and pruning strategies under MNI.

Dataset	k	Options	MNI Time	MNI Pat.	FIS Time	FIS Pat.	GIS Time	GIS Pat.
Credit	150	x	6.74 ± 0.16	3,511	6.28 ± 0.13	3,511	7.20 ± 0.16	3,511
Aviation	2200	e	17.91 ± 0.36	309	12.27 ± 0.19	309	11.01 ± 0.16	309
ExprCalc	3	x	11.77 ± 1.88	52,456	11.00 ± 2.03	52,442	7.45 ± 0.23	52,442
Zookeeper	12	x	20.14 ± 0.78	26,383	16.81 ± 0.47	24,687	16.59 ± 0.57	24,693
DDH	25	x	7.41 ± 0.10	1,388	6.41 ± 0.07	1,129	6.18 ± 0.09	1,133
BCEL	100	x	56.34 ± 0.52	15,739	52.99 ± 0.45	15,700	46.41 ± 0.39	15,701
jGit	100	x	82.68 ± 1.03	3,897	65.57 ± 0.33	3,615	60.11 ± 0.50	3,600
Tomcat	1000	x	53.95 ± 0.56	141	52.59 ± 0.46	121	52.27 ± 0.51	122
hBase	200	x	$1,338.01 \pm 14.6$	27,305	$1,091.25 \pm 10.9$	23,902	757.63 ± 8.72	23,742
OrientDB	1500	x	723.59 ± 42.1	3,214	537.35 ± 11.2	2,712	466.75 ± 4.58	2,722
Random-1	2	x	0.0672 ± 0.0077	97	0.0592 ± 0.0077	95	0.06 ± 0.008	95
Random-2	21	e	72.54 ± 1.52	33	0.11 ± 0.01	2	0.11 ± 0.01	2

Table 3.4: Comparison of support metrics. No pruning strategies were used.

Dataset	k	Options	o Time	o Pat.	t Time	t Pat.	ot Time	ot Pat.
Credit	150	x	5.94 ± 0.11	3,511	6.30 ± 0.12	3,511	5.32 ± 0.11	3,511
Aviation	2200	e	–	–	8.96 ± 0.14	309	–	–
ExprCalc	3	x	7.62 ± 1.77	52,425	7.11 ± 1.16	52,442	6.13 ± 1.24	52,425
Zookeeper	12	x	12.39 ± 0.30	23,509	13.54 ± 0.38	24,693	10.60 ± 0.37	23,509
DDH	25	x	5.71 ± 0.07	1,017	5.93 ± 0.07	1,133	5.59 ± 0.06	1,017
BCEL	100	x	33.01 ± 0.26	15,691	32.02 ± 0.27	15,701	24.12 ± 0.17	15,691
jGit	100	x	39.81 ± 0.28	3,358	40.57 ± 0.29	3,600	31.49 ± 0.25	3,358
Tomcat	1000	x	51.88 ± 0.52	119	51.94 ± 0.53	122	51.81 ± 0.47	119
hBase	200	x	299.92 ± 2.16	13,933	494.71 ± 3.14	23,742	221.34 ± 1.91	13,931
OrientDB	1500	x	378.83 ± 25.4	2,577	413.56 ± 25.6	2,722	332.77 ± 21.0	2,577
Random-1	2	x	0.066 ± 0.007	94	0.06 ± 0.007	95	0.068 ± 0.006	94
Random-2	21	e	–	–	0.11 ± 0.01	2	–	–

Table 3.5: Comparison of the pruning strategies under GIS.



(a) ExprCalc (b) Zookeeper (c) DDH (d) BCEL

(e) jGit (f) Tomcat (g) hBase (h) OrientDB

Figure 3.6: This figure shows the scalability of REGRAX for mining frequent patterns quickly from large search spaces. Each plot is a box-and-whisker diagram of execution time in seconds for 50 runs of the Unweighted Random Walk sampling algorithm on the PDG datasets. Each run collected samples containing 100 maximal frequent subgraphs. The number on the top of the axis is the maximum size (in edges) of a frequent subgraph found with the options specified below the axis.

mine the largest dataset, OrientDB (see Table 5.1), faulting with an Out-of-Memory Error (OOM) after consuming all available RAM.

Table 5.1 summarizes the performance of REGRAX on all datasets used in the evaluation. Credit and Aviation are undirected labeled graphs that have been used in several previous studies [27, 39, 72, 87] and are included here for comparison. These were mined as if they were directed labeled graphs. REGRAX outperforms the previously published numbers for these datasets (see Table 5.1). The Random-1 and Random-2 datasets are small directed random graphs generated from uniform label and edge distributions. Random-1 is a sparse graph with an edge-to-vertex ratio similar to those for the real world graphs in the study. Random-2 is a dense graph intended to be difficult to mine. Dense graphs with uniform edge and label distributions are a hard cases for frequent subgraph mining since there are many repeated subgraphs.

The rest of the datasets are Program Dependence Graphs (PDGs) produced from Java programs using `jpdg` [59, 60]. The OrientDB dataset includes not only the PDG produced from the source code of OrientDB but also includes the PDG of all the libraries it depends on (excluding the Java Standard Library). The DDH dataset is a portion of a proprietary application and has been anonymized. All of the graphs analyzed as part of this study have been made available as part of the release of REGRAX for the use in future research.

Note: all REGRAX executions were replicated 50 times in the evaluation. All timing data for REGRAX is reported as the mean number of seconds required $\pm$ the standard

deviation.

3.3.1 REGRAX versus GraMi

Table 3.2 compares the performance of GraMi [39] to REGRAX. GraMi was not observed to make good use of the 24 available hardware threads and REGRAX was therefore run with its support for parallel processing disabled. GraMi supports mining in both directed and undirected graphs. However, the directed mining mode is much slower, so we used the undirected mode. This led to a slight discrepancy in the number of reported frequent subgraphs. There were three datasets that GraMi could not mine: Aviation (at minimum support level 1500), OrientDB, and Random-2. GraMi failed with an Out-of-Memory error (OOM) on the OrientDB dataset. GraMi was left to run for 2 days on Aviation at support level 1500, but it did not complete and it used over 40 GB of memory. Finally, GraMi was left to run on the Random-2 dataset for 10 days, but it did not complete.

Overall, REGRAX significantly outperformed GraMi on all of the PDG datasets as well as on Credit, Random-1 and Random-2. GraMi performed better than REGRAX on the Aviation dataset at higher support levels but could not successfully mine Aviation at the lower support level. For this comparison REGRAX was handicapped by preventing it from using more than 1 thread – if more threads are allowed (see Table 5.1) REGRAX performs even better. Parallelization is an effective strategy for increasing the performance of a frequent subgraph miner.

3.3.2 Optimization Effects

Table 3.3 shows the effects of the optimizations introduced in Section 3.2 (consult Table 5.1 for the legend of pruning strategies and optimizations). The two candidate subgraph generation strategies (*Extend from Embeddings* x and *Extend from Frequent Edges* e) are compared. The effects of *Overlap Pruning* o and *Extension Pruning* t are also shown in combination with both candidate generation strategies. Note, *Overlap Pruning* cannot be used when generating candidates from frequent edges because (as shown in Listing 3.3) the enumeration of the embeddings exits early.

Neither of candidate generation strategies was clearly superior to the other in all instances. However, from our testing with larger graphs, with a large number of unique labels, and when using lower minimum support values, it is better to extend from the embeddings than from the frequent edges. When extending from embeddings fewer spurious candidates are generated, resulting in lower overall memory usage and higher performance. However, for some datasets (such as Aviation) extending from the frequent edges was better by an order of magnitude. Both *Overlap Pruning* and *Extension Pruning* proved to be effective. However, the magnitude of the effect was dependent on the dataset. For instance when extending from embeddings on the Aviation dataset, *Extension Pruning* reduced the mining time from 146 seconds to 30 seconds. However, for Tomcat it had a negligible effect.

Table 3.4 compares the effect of the three support measures MNI, FIS, and GIS. As a reminder, MNI [19] satisfies DCP while FIS and GIS do not. Unless mining with MNI is not feasible for your application we suggest using MNI. FIS and GIS reduce the number of patterns found (marginally) for most of the datasets. However, the payoff in improved performance can be dramatic as in the case of hBase, which took 1,338 seconds to mine under MNI but only 758 seconds under GIS.

Table 3.5 shows the effects of combining GIS with the *Overlap Pruning* and *Extension Pruning*. Combining GIS with *Extension Pruning* appears to have no empirical effect on the number of patterns found (although theoretically the combination could result in fewer patterns found). This combination always improves performance (sometimes dramatically, see hBase). Combining GIS with *Overlap Pruning* is less of a clear win as the number of patterns may be reduced. If losing a few patterns does not matter further combining all three optimizations together results in the best performance.

The subgraphs that are missed under the combination of GIS, *Extension Pruning*, and *Overlap Pruning* tend to have one of the following two properties. First, while a missed subgraph is frequent under MNI it is only frequent due to embeddings that overlap either fully or partially. The overlaps cause GIS (and also FIS) to return a lower support value than MNI. Second, a missed subgraph contains a subgraph with automorphisms. The automorphisms cause embeddings to be skipped under *Overlap Pruning*. If missing a few frequent subgraphs such as these is not a problem, this combination can enable mining graphs that are not easily mined otherwise.

3.3.3 Sampling Low-Frequency Recurring Subgraphs

Our experiments have shown that using the combination of GIS, *Extension Pruning*, and *Overlap Pruning* is required when sampling low-frequency recurring subgraphs from large program dependence graphs. This combination may result is some frequent subgraphs (which would have been findable under MNI) being missed. Figure 5.2 summarizes this result. It was possible to sample all of the PDGs at the lowest support levels. However, it was not possible to sample all of the graphs at such low support levels under MNI within our time budget of a maximum of 2 days per program execution.

Overall, REGRAX was able to successfully sample all of the datasets within our time budget using the GIS support measure. The longest amount of time it took to sample any of the datasets was 10 hours, for OrientDB at support level 2. All of the other datasets could be sampled much faster, with a worst case time of 1 hour. For hBase (the next largest dataset after OrientDB), the mean sampling time was just 9 minutes. However, under the MNI support measure the results were not as good. The Tomcat, hBase and OrientDB datasets could not be sampled within the 2 day time budget, and the minimum support level had to be increased in order to reliably get results. The underlying cause of the problem for these datasets was the presence of patterns with many millions of overlapping embeddings – all of which need to be computed to calculate the MNI support measure. However, using GIS the subgraph matching search tree is pruned making the performance much better.

3.4 Conclusion

Frequent subgraph mining large connected graphs is a useful and important data-mining technique with broad applications in many disciplines. Its use has been held back by scalability concerns and difficulties using it on large datasets with low minimum frequency settings. The novel system REGRAX presented in this paper contains new optimizations that help it outperform GraMi [39], a recent high performance miner, on both synthetic and real world datasets. REGRAX also implements new support measures. The new measures enable REGRAX to extract low-frequency recurring subgraphs from large graphs using sampling techniques. REGRAX is an advance in the state-of-the-art for mining frequent subgraphs from large connected graphs.

Chapter 4

Sampling Code Clones with GRAPLE

4.1 Introduction

Code clones are similar fragments of program code [125]. They can arise from copying and pasting, using certain design patterns or certain APIs, or adhering to coding conventions, among other causes. Code clones create maintenance hazards, because they often require subtle context-dependent adaptation and because other changes must be applied to each member of a clone class. To manage clone evolution the clones must first be found. Clones can be detected using any program representation: source code text, tokens, abstract syntax trees (ASTs), flow graphs, dependence graphs, etc. Each representation has advantages and disadvantages for clone detection.

PDG-based clone detection finds *dependence clones* corresponding to recurring subgraphs of a program dependence graph (PDG) [83,85]. Since PDGs are oblivious to semantics preserving statement reorderings they are well suited to detect *semantic* (functionally equivalent) clones. A number of algorithms find clones from PDGs [22,48,70,83,85,93,109,112,119]. However, as Bellon [16] notes, "PDG based techniques are computationally expensive and often report non-contiguous clones that may not be perceived as clones by a human evaluator." Most PDG-based clone detection tools are biased, detecting certain clones but not others.

The root cause of scalability problems with PDG-based clone detection is the number of dependence clones. Section 4.3 illustrates this with an example in which we used an unbiased frequent subgraph mining algorithm [87] to detect all dependence clones in Java programs. In programs with about 70 KLOC it detected around 10 million clones before disk space was exhausted. Processing all dependence clones is impractical even for modestly sized programs.

Instead of exhaustively enumerating all dependence clones, an unbiased random sample can be used to statistically estimate parameters of the whole "population" of clones, such as the prevalence of clones exhibiting properties of interest. For these reasons, we developed a statistically unbiased method for *sampling* dependence clones and for *estimating* parameters of the whole clone population.

[1]Note, portions of this chapter originally appeared as [59] https://dx.doi.org/10.1145/2989238.2989241.

We present GRAPLE (GRAph samPLE)[1], a method to generate a representative sample of recurring subgraphs of any directed labeled graph(s). It can be used to sample subgraphs from any kind of program graph representation. GRAPLE is not a general purpose clone detector but it can answer questions about dependence clones that other PDG-based clone detection tools cannot. We conducted a preliminary case study on a commercial application and had its developers evaluate whether the sampled subgraphs represented code duplication. To our knowledge, it is the first study to have professional programmers examine dependence clones.

Contributions

1. GRAPLE: a framework for unbiased sampling of frequent subgraphs of large graphs such as PDGs and for estimating statistics characterizing the whole population of frequent subgraphs. (Sec. 4.3.3, 4.3.4)
2. A case study in which GRAPLE was applied to a commercial Android application and in which its output was examined by developers. (Sec. 4.4)
3. `jpdg`: a new procedure dependence graph generator for JVM languages. (Sec. 4.2)

GRAPLE has applications in bug mining, test case selection, and bioinformatics. The sampling algorithm also applies to frequent item sets, subsequences, and subtrees allowing code clone sampling from tokens and ASTs.

4.2 A Review of Dependence Graphs

A *program dependence graph* (PDG) [46] represents possible dependence relationships between statements in a program, with vertices representing statements and directed edges representing control and data dependences. Informally, a statement s_1 is *control dependent* on a statement s_2 if s_2 is a branch predicate that controls the execution of s_1. A statement s_1 is *data dependent* on a statement s_2 if a value assigned to a variable x at s_2 can later be accessed from x at s_1. (This requires that all control flow paths from s_2 to s_1 do not assign a new value to x.)

PDGs approximate semantic dependencies between statements. They are not affected by reordering statements in ways which preserve the semantics. Horwitz *et al.* [64] showed that, under certain assumptions, if the PDGs of two programs are isomorphic then the programs are equivalent. Giv-en Horwitz's result and related results from Podgurski [113, 114], the PDG is a good representation to detect code clones with renamed variables, semantics-preserving statement re-orderings, and unrelated code insertions.

An important variant of the program dependence graph is the *system dependence graph* (SDG) [65], which consists of *procedure dependence graphs* (pDGs), each representing an individual subprogram, connected by *inter-procedural dependence edges* representing subprogram calls. The case study described in Section 4.4 involves mining pDGs. Figure 4.1 shows an example procedure dependence graph. The dotted lines indicate control dependencies and the solid lines indicate data dependencies.

To compute the pDGs used in this paper a prototype tool named `jpdg` was built. A successor to JavaPDG [132], `jpdg` was created to improve the PDG representation for code mining purposes. For instance, most dependence graphs place the arguments to operations in the vertices of the graph. In the graphs produced by `jpdg` these are associated with

[1] https://github.com/timtadh/graple

the edges (see Figure 4.1). `jpdg` is built on top of the Soot optimization framework [49][2], and non-constant vertices map to Jimple instructions. Control dependencies are computed using Cytron's method [30]. Data dependences are computed using Upward Exposed Uses analysis [104].

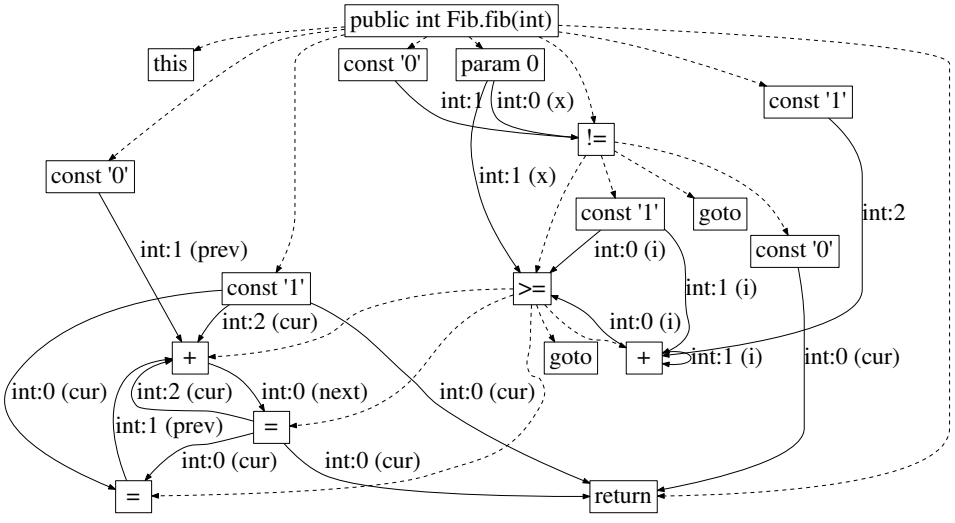
4.3 Sampling Dependence Clones

A *dependence clone* is duplicated code detected from the program dependence graph (PDG). To detect dependence clones, identify subgraphs appearing in multiple locations in the PDG.

[2]http://sable.github.io/soot/

```
1  public class Fib {
2    public int fib(int x) {
3      int prev = 0;
4      int cur = 1;
5      if (x == 0) {
6        cur = 0;
7      } else {
8        for (int i = 1; i < x; i++) {
9          int next = prev + cur;
10         prev = cur;
11         cur = next;
12       }
13     }
14     return cur;
15   }
16 }
```

(a) Program to compute terms from the Fibonacci sequence.



(b) Procedure dependence graph of Figure 4.1a.

Figure 4.1: Example procedure and procedure dependence graph. Solid lines are data dependencies. Dashed lines are control dependencies. The labels on the data dependencies indicate the type of the data and its usage context (ie. its parameter number). The data dependencies are also annotated with variable names to aid in readability. ©Tim Henderson

The problem of finding recurring subgraphs is called *Frequent Subgraph Mining* (FSGM) [25]. Frequent subgraph miners search for subgraphs which recur k times with $k > 1$.

Applying standard mining algorithms to program graphs is not straightforward. Software engineers are potentially interested in subgraphs with very few repetitions. Small frequency thresholds are uncommon in the applications typically considered in the data-mining literature. Our experiments on `jgit`[3] ($\sim$ 72 KLOC) have found that with a minimum frequency setting of 5, there are over 11.8 million frequent subgraphs. We were unable to completely mine `jgit` as we exhausted our disk space storing the patterns ($\sim$ 1 TB). This mining attempt, which used the vSiGraM algorithm [87], took over 12 days. We made another attempt where the patterns were simply logged to the console instead of stored. During this attempt, over 350 million patterns were discovered before the process was killed after 10 days. The application considered in Section 4.4 had similar results (¿ 10 million frequent patterns before disk space exhaustion and ¿400 million patterns after 10 days of mining).

So, not only is it impractical in time and space to use an algorithm like vSiGraM to do code clone detection it would not be possible for all patterns to be stored and individually examined.

4.3.1 How Frequent Subgraph Mining Works

Standard frequent subgraph miners search for subgraphs that recur at least a specified number of times in a graph database, which may contain one very large graph or many smaller graphs [25]. Conceptually, miners work by enumerating the subgraphs of the graphs in the database by traversing *frequent connected subgraph lattice* (see Fig. 4.2b). As each subgraph is found its *support* must be computed. Informally, the support of a subgraph is the number of *embeddings* that it has in the graph database.

Using subgraph-isomorphism checks to count support is expensive. A faster way to count support is to store the embeddings of each subgraph. The stored embeddings can also be used during the subgraph enumeration process to reduce the number of candidate patterns. As each subgraph is produced it is "canonicalized." The canonicalization process always puts isomorphic graphs into the same form and thus neatly solves the graph isomorphism problem. We use Bliss [77] for canonicalization.

4.3.2 From Mining to Sampling

Large PDGs may have a huge number of frequent subgraphs, but in applications of clone detection it may be unnecessary to consider them all. We focus on two use-cases: (1) developers want to manually examine mined clones, e.g., to propose refactorings, and (2) developers and researchers want to answer questions about the whole population of clones that *could* be mined given enough time and storage space.

In the first use-case, developers will have limited time to examine mined clones. Thus, they will generally prefer to consider a small, diverse set of clones. In the second use-case, the questions posed could be either objective (e.g., "What percentage of our code base is covered by one or more frequent subgraphs?") or subjective (e.g., "What proportion of potential dependence clones do our programmers want to refactor?"). Both kinds of questions can often by answered by examining a representative sample of frequent subgraphs. If care is taken in designing the method to select the sample, then *statistical estimation* [141] techniques can be used to estimate unbiased answers.

[3] https://github.com/eclipse/jgit

```
# param G : the graph being mined
# param bottom : lattice node for the empty subgraph
# param min_support : int, minimum number of embeddings
# returns : leaf node of the frequent connected subgraph
#           lattice which is a maximal frequent subgraph
def walk(G, bottom, min_support):
  v = u = bottom
  while v is not None:
    u = v
    v = rand_select(get_children(G, u, min_support))
  return u

# param u : a lattice node
# returns : a list of lattice nodes which are 1 edge
#           extensions of u
def get_children(G, u, min_support):
  exts = list()
  for emb in u.embeddings:
    for a in embedding.V:
      for e in G.edges_to_and_from(a):
        if not emb.has_edge(e):
          exts.append(emb.extend_with_edge(e))
  groups = group_isomorphs(exts)
  return [ LatticeNode(lbl, group)
           for lbl, group in groups.iteritems()
           if len(group) >= min_support ]

# param subgraphs : a list of subgraphs of G
# returns : map label -> list of isomorphic subgraphs.
def group_isomorphs(subgraphs):
  isomorphs = dict()
  for sg in subgraphs:
    label = bliss.canonical_label(sg)
    if label not in isomorphs:
      isomorphs[label] = list()
    isomorphs[label].append(sg)
  return { label: minimum_image_supported(group)
           for label, group in isomorphs.iteritems() }
```

Listing 4.1: GRAPLE's sampling procedure

We developed GRAPLE to address both use-cases. It samples randomly from the space of *maximal frequent subgraphs* (a frequent subgraph is maximal if no larger frequent subgraph can be constructed from it). The sampling procedure is well defined and allows us to compute *selection probabilities* for subgraphs, which can be used in statistical estimators such as the *Horvitz-Thompson (HT) unequal probability estimator* [141]. Furthermore, developers can use GRAPLE to collect small, diverse sets of potential dependence clones from an entire code base or from parts of interest. The same basic algorithm can also be applied to item-set, sub-sequence, and sub-tree mining.

4.3.3 Sampling Maximal Frequent Subgraphs

All frequent pattern miners traverse the frequent pattern lattice, which for subgraph miners means the frequent connected subgraph lattice (see Figure 4.2). Each node in the lattice represents a frequent subgraph, with the directed edges connecting A to B if adding one edge to A produces a graph isomorphic to B. Miners traverse the lattice in either a breadth first or depth first manner to find all of the frequent subgraphs.

Since we seek a random sample of the maximal frequent subgraphs, it is unnecessary to traverse the whole lattice. Instead, we make n partial traversals where n is the desired

sample size. Each traversal is an unweighted random walk over the lattice, which proceeds from smaller frequent subgraphs to larger ones. The walk terminates when it reaches a maximal frequent subgraph and that pattern and its embeddings are added to the sample. This procedure is outlined in Listing 4.1.

The most expensive part of frequent subgraph mining is extending a pattern with e edges to patterns with $e+1$ edges and computing support of each of those extensions. Extensions are computed on lines 17-22 by considering all possible extensions for each embedding. The lattice nodes hold a list of their embeddings, making this computation relatively cheap. After all possible extensions are computed they are grouped by their canonical labeling (lines 30-36) as computed by Bliss [77]. The groups are then minimized (lines 37-38) to remove many of the overlapping embeddings using minimum image support [19]. If a group contains enough embeddings to be considered frequent then a lattice node is created and it is returned as a child (lines 24-26).

Generalizing results from the sample of maximal frequent patterns to the population of all maximal frequent pattern requires taking into account *unequal sample inclusion probabilities*. In order to do this a correction factor or weight is applied to each sampled value. The weight for the value y_i of the i^{th} population unit is the inverse of the unit's probability π_i of inclusion in the sample. With these weights, the *Horvitz-Thompson (HT) estimator* [63], denoted $\hat{\tau}_\pi$, can be used to make an unbiased estimate of the population total τ for the study variable, and a simple variant $\hat{\mu}_\pi$ can be used to unbiasedly estimate the population mean μ. Let ν be the number of distinct units in the sample. Then

$$\hat{\tau}_\pi = \sum_{i=1}^{\nu} \frac{y_i}{\pi_i} \tag{4.1}$$

Observe that units that are rarely sampled will have their values boosted substantially by the weights $1/\pi_i$, while units which are commonly sampled have their values boosted less. Thompson [63] provides formulas for the HT estimator's mean and variance and for computing confidence intervals for estimates.

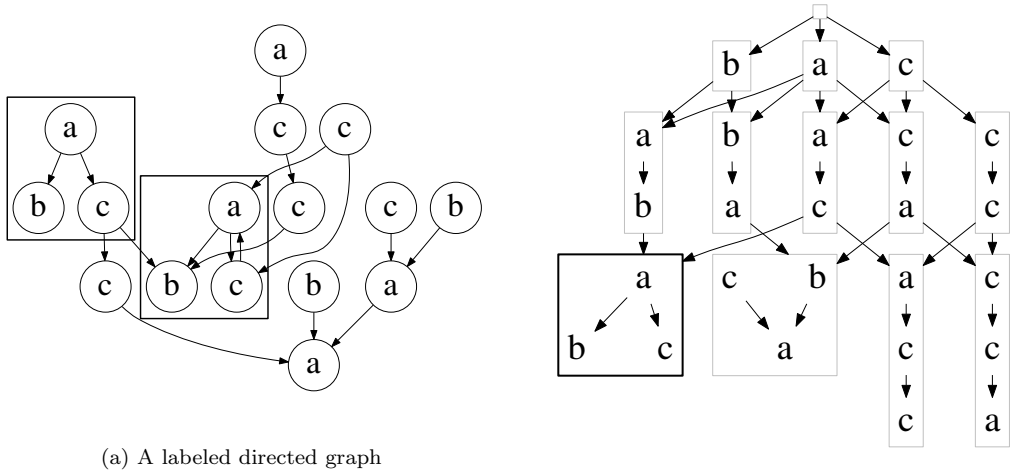
When sampling n units with replacement, the probability π_i that the i^{th} population unit is included in the sample can be computed from the probability p_i that unit i is selected on a particular random walk:

$$\pi_i = 1 - (1 - p_i)^n \tag{4.2}$$

4.3.4 Computing the Probability of Selecting a Maximal Subgraph

In order to use the HT estimator outlined in the previous section, it is necessary to determine the probability p_i that the i^{th} maximal frequent pattern $[\![H_i]\!]$ is selected on a random walk of the k-frequent connected subgraph lattice (k-$\mathcal{L}_G$). We compute these probabilities using the theory of Markov chains.

A *finite-state Markov chain* [54] consists of a finite set of states, $S = \{s_1, \ldots, s_n\}$, and a matrix $\mathbf{P}$, called the *transition matrix*, where $\mathbf{P}_{i,j}$ gives the probability of a state transition from s_i to s_j. A Markov chain moves from state to state according to the probabilities in the transition matrix. A random walk in a graph G can be viewed as a Markov chain whose set of states S corresponds to the vertex set V_G. An *absorbing Markov chain* [54] is a special type of Markov chain which always ends in a state that cannot be exited, called an *absorbing state* (see Appendix A.1).



(a) A labeled directed graph

(b) 2-frequent connected subgraph lattice

Figure 4.2: Figure 4.2b is a connected subgraph lattice of Figure 4.2a including only subgraphs with 2 or more embeddings in Figure 4.2a. The boxed nodes in the graph show the embeddings of the boxed subgraph in the lattice. ©Tim Henderson

To construct an absorbing Markov chain from the lattice k-$\mathcal{L}_G$, let the states of the chain be the vertices of the lattice (i.e., the frequent patterns $[\![H_i]\!]$). To model how the algorithm in Listing 4.1 transitions from one lattice node to the next by uniformly selecting a neighboring node, let the transition probability for an edge $v_i \to v_j$ be the reciprocal of the out-degree of v_i:

$$\mathbf{P}_{i,j} = \begin{cases} \frac{1}{\sum_k \mathbf{E}_{i,k}} & \text{if} \quad \mathbf{E}_{i,j} = 1 \\ 1 & \text{if} \quad i = j \wedge v_i \text{ is maximal} \\ 0 & \textit{otherwise} \end{cases} \tag{4.3}$$

The selection probability p_i of $[\![H_i]\!]$ is the probability that state s_i absorbs the Markov process starting at the bottom lattice node. To compute p_i, arrange the transition matrix $\mathbf{P}$ into *canonical form* such that the transient states come before the absorbing states:

$$\mathbf{P} = \begin{array}{c} \\ \text{TR.} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \end{array} \tag{4.4}$$

$\mathbf{Q}_{i,j}$ is the probability of transitioning from a transient state s_i to transient state s_j. $\mathbf{R}_{i,j}$ is the probability of transitioning from transient state s_i to absorbing state s_{t+j} where t is the number of transient states. $\mathbf{I}$ is the identity matrix and $\mathbf{0}$ is the zero matrix, as once a Markov process enters an absorbing state it never leaves. The probability of a process starting at the bottom of the lattice s_0 and being absorbed by state s_i with zero or more transitions ($\xrightarrow{\star}$) is [54]:

$$p_i = \Pr[s_0 \xrightarrow{\star} s_i] = (\mathbf{P}^\infty)_{0,i} = ((\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R})_{0,(i-t)} \tag{4.5}$$

Computing p_i with a submatrix of P

To compute p_i using Equation 4.5, the entire matrix $\mathbf{P}$ needs to be constructed. It turns out only a *submatrix* of $\mathbf{P}$ is needed to compute the probability p_i of selecting a frequent pattern $[\![H_i]\!]$ using the algorithm in Listing 4.1. The required rows and columns of $\mathbf{P}$ correspond to the vertices of k-$\mathcal{L}_G$ which are subgraphs of H_i.

Lemma 4.1. *Let s_i be an absorbing state in a Markov chain formed from a k-frequent connected subgraph lattice of a graph G. The selection probability $p_i = (\mathbf{NR})_{0,(i-t)}$ can be computed from a sub-matrix of the transition matrix $\mathbf{P}$ containing only those states from which s_i can be reached.*

Proof. If there does not exist a path $v_j \xrightarrow{\star} v_i$ in the k-frequent connected subgraph lattice k-$\mathcal{L}_G$ then the product of its adjacency matrix entries corresponding to any sequence of edges possibly connecting v_j to v_i must be zero. Therefore, summing over all such edge sequences, we have:

$$\left(\sum_{n=1}^{\infty}\sum_{k_1=1}^{n}\cdots\sum_{k_n=1}^{n}\mathbf{E}_{j,k_1}\left(\prod_{i=1}^{n-1}\mathbf{E}_{k_i,k_{i+1}}\right)\mathbf{E}_{k_n,i}\right)=0 \tag{4.6}$$

The probability of a Markov chain that starts in state s_j eventually reaching state s_i is

$$\Pr[s_j \xrightarrow{\star} s_i] = \sum_{n=1}^{\infty}\sum_{k_1=1}^{n}\cdots\sum_{k_n=1}^{n}\mathbf{P}_{j,k_1}\left(\prod_{i=1}^{n-1}\mathbf{P}_{k_i,k_{i+1}}\right)\mathbf{P}_{k_n,i} \tag{4.7}$$

If there does not exist a path in the lattice from v_j to v_i then this probability is zero. The selection probability formula $p_i = (\mathbf{NR})_{j,(i-t)}$ can be rewritten, with t indicating the number of transient nodes, as shown in Equation 4.8.

$$p_i = \sum_{k=1}^{t}\mathbf{N}_{j,k}\mathbf{R}_{k,(i-t)} \tag{4.8}$$

Using the definition of the fundamental matrix this equation can be rewritten as follows obtaining Equation 4.12.

$$p_i = \sum_{k=1}^{t}\left(\lim_{n=1}^{\infty}\left(\mathbf{I}+\sum_{e=1}^{n}\mathbf{Q}^e\right)\right)_{j,k}\mathbf{R}_{k,(i-t)} \tag{4.9}$$

$$p_i = \sum_{k=1}^{t}\left(\sum_{n=1}^{\infty}\sum_{k_1=1}^{t}\cdots\sum_{k_n=1}^{t}\mathbf{Q}_{j,k_1}\left(\prod_{x=1}^{n-1}\mathbf{Q}_{k_x,k_{x+1}}\right)\mathbf{Q}_{k_n,k}\mathbf{R}_{k,(i-t)}\right) \tag{4.10}$$

$$p_i = \sum_{k=1}^{t}\left(\sum_{n=1}^{\infty}\sum_{k_1=1}^{t}\cdots\sum_{k_n=1}^{t}\mathbf{P}_{j,k_1}\left(\prod_{x=1}^{n-1}\mathbf{P}_{k_x,k_{x+1}}\right)\mathbf{P}_{k_n,k}\mathbf{P}_{k,i}\right) \tag{4.11}$$

$$p_i = \sum_{n=1}^{\infty}\sum_{k_1=1}^{t}\cdots\sum_{k_n=1}^{t}\mathbf{P}_{j,k_1}\left(\prod_{i=1}^{n-1}\mathbf{P}_{k_i,k_{i+1}}\right)\mathbf{P}_{k_n,i} \tag{4.12}$$

Note, Equation 4.12 is equivalent to the right hand side of Equation 4.7. Since $\Pr[s_j \xrightarrow{\star} s_i] = 0$ if there is no path in the lattice from v_j to v_i, vertices from which s_i cannot be

reached have no effect on the computation of p_i and can be omitted. Omitting vertices v_j for all v_j where there does not exist a path in the lattice to v_i corresponds to removing the j^{th} row and column from $\mathbf{P}$. Therefore, only a sub-matrix of $\mathbf{P}$ containing those states from which s_i can be reached are needed. □

Lemma 4.2. *The states from which an absorbing state s_i can be reached in a Markov chain formed from a k-frequent connected subgraph lattice k-$\mathcal{L}_G$ correspond to the vertices of the connected subgraph lattice of the graph represented by state s_i.*

Proof. A vertex v of k-$\mathcal{L}_G$ represents a graph. Given two vertices u and v of k-$\mathcal{L}_G$, u reaches v if and only if a sequences of edges can be added to u such that u extended with those edges is isomorphic to v (e.g. $u + \epsilon_1 + \epsilon_2 + ... \cong v$). Thus, the statement u reaches v in k-$\mathcal{L}_G$ is equivalent to saying u is a subgraph of v. If u is a subgraph of v then it will be a vertex in v's connected subgraph lattice $\mathcal{L}_v$ by the definition of connected subgraph lattice. Therefore, all states in which can reach s_i must be correspond to subgraphs of s_i and are therefore in s_i's connected subgraph lattice. □

Theorem 4.1. *Let $[\![H_i]\!]$ be a maximal k-frequent pattern sampled from k-$\mathcal{L}_G$. Let s_i be the corresponding state in the Markov chain formed from k-$\mathcal{L}_G$. The selection probability of $[\![H_i]\!]$, $p_i = ((\mathbf{I}-\mathbf{Q})^{-1}\mathbf{R})_{0,(i-t)}$, can be computed from the submatrix of $\mathbf{P}$ that includes only the rows and columns that correspond to subgraphs of H_i.*

Proof. By Lemma 4.2 all states of the Markov chain which can reach s_i correspond to subgraphs of s_i. By Lemma 4.1 the only rows and columns of $\mathbf{P}$ which are necessary are the rows and columns for states which can reach s_i. Therefore, the sub-matrix of $\mathbf{P}$ that is needed only contains states which correspond to subgraphs of H_i and are in the connected subgraph lattice $\mathcal{L}_{H_i}$. □

As a consequence of Theorem 4.1, only a sub-matrix of $\mathbf{P}$ is needed to compute p_i for any given i, and that sub-matrix is exactly the one which corresponds to the connected subgraph lattice computed from the target subgraph.

Example 4.1. *The lattice in figure 4.2 corresponds to the following matrix $\mathbf{P}$ arranged in canonical form. Note: the empty spaces in the matrices represent the number 0.*

	0	1	2	3	4	5	6	7	8	9	10	11	12
0		.33	.33	.33									
1					.5	.5							
2					.25	.25	.25	.25					
3							.33	.33	.33				
4										1			
5											1		
6										.5		.5	
7											.5		.5
8												.5	.5
9										1			
10											1		
11												1	
12													1

If the subgraph $H_{11} = a \rightarrow c \rightarrow c$, corresponding to state s_{11}, is sampled the following would be the submatrix of $\mathbf{P}$ needed to compute p_{11}.

$$\mathbf{P}^{H_{11}} = \begin{array}{c} \\ 0 \\ 2 \\ 3 \\ 6 \\ 8 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 2 & 3 & 6 & 8 & 11 \end{array} \\ \begin{bmatrix} & .33 & .33 & & & \\ & & & .25 & & \\ & & & .33 & .33 & \\ & & & & & .5 \\ & & & & & .5 \\ & & & & & 1 \end{bmatrix} \end{array}$$

The fundamental submatrix would be:

$$\mathbf{N}^{H_{11}} = (\mathbf{I} - \mathbf{Q}^{H_{11}})^{-1} = \begin{array}{c} \\ 0 \\ 2 \\ 3 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 2 & 3 & 6 & 8 \end{array} \\ \begin{bmatrix} 1 & .33 & .33 & .1914 & .1089 \\ & 1 & & .25 & \\ & & 1 & .33 & .33 \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \end{array}$$

Computing $\mathbf{N}^{H_{11}}\mathbf{R}^{H_{11}}$ yields:

$$\mathbf{N}^{H_{11}}\mathbf{R}^{H_{11}} = \begin{array}{c} \\ 0 \\ 2 \\ 3 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 2 & 3 & 6 & 8 \end{array} \\ \begin{bmatrix} 1 & .33 & .33 & .1914 & .1089 \\ & 1 & & .25 & \\ & & 1 & .33 & .33 \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ 0 \\ 2 \\ 3 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 11 \end{array} \\ \begin{bmatrix} \\ \\ \\ .5 \\ .5 \end{bmatrix} \end{array} = \begin{array}{c} \\ 0 \\ 2 \\ 3 \\ 6 \\ 8 \end{array} \begin{array}{c} \begin{array}{c} 11 \end{array} \\ \begin{bmatrix} .150 \\ .125 \\ .333 \\ .5 \\ .5 \end{bmatrix} \end{array}$$

The probability for starting at the root node of the lattice (the empty subgraph H_0) and ending at H_{11} is:

$$\begin{aligned} p_{11} &= Pr[s_0 \xrightarrow{\star} s_{11}] \\ &= (\mathbf{P}^{\infty})_{0,11} \\ &= ((\mathbf{I} - \mathbf{Q})^{-1}\mathbf{R})_{0,11-9} \\ &= ((\mathbf{I} - \mathbf{Q}^{H_{11}})^{-1}\mathbf{R}^{H_{11}})_{0,0} \\ &= .150 \end{aligned}$$

Computing the submatrix of $\mathbf{P}$ for H_i requires finding every connected subgraph of H_i which is much less work than mining all frequent subgraphs of G. Unfortunately, computing the submatrix is only tractable for subgraphs with fewer than 20 edges (which can induce submatrices as large as $2^{20} \times 2^{20}$). In future work, we intend to estimate p_i rather than compute it exactly, in order to handle much larger subgraphs. Note that our sampling procedure does not have a size limitation (it has found frequent subgraphs with over 100 edges); only the probability computation has this limitation.

4.4 Case Study: Assessment of Clone Relevance

We conducted a preliminary case study to see if GRAPLE could help us assess the usefulness of PDG-based code clone detection to developers. Our study assessed the relevance of

dependence clones in a commercial Android application with $\sim$ 73 KLOC that had been under continuous development for 5 years. Nine developers participated in the study. The study was conducted as a survey which asked a set of 10 questions about each group of mined clones (two questions are in Figure 4.3). The clones were displayed as both graphs and highlighted regions of source code. The goal was to estimate the proportions of frequent subgraphs that represent duplicate code and that developers would act upon.

The survey involved 104 dependence clone groups sampled using GRAPLE, which was configured to require frequent subgraphs to have minimum support 5 and to contain at least 8 vertices. As sampling was done *with replacement*, 415 patterns were sampled, with 122 unique patterns. Of the 122, 104 with fewer than 20 edges were retained as discussed in Section 4.3.4. The sampling was done on an computer with an 8 core Intel Xeon processor, 64 GB of main memory and a 250 GB hard drive. It took 138 seconds to collect the samples and 23.2 hours to compute the selection probabilities used in the HT estimator. Computing the selection probabilities was made possible by using SuiteSparse[4] for large sparse matrix inversion.

At the beginning of the study, the participants were given a presentation on code clones. Code clones were somewhat familiar to these developers, as they utilize a commercial static analysis tool, SonarQube, which makes use of a clone detector based on Hummel *et al.*'s algorithm [69]. SonarQube detected none of the clones the developers reviewed in the study. In an ideal study each clone group would have been reviewed by each participant, but in order to maximize the number of clones reviewed, each clone group was only reviewed by a single participant.

4.4.1 Study Results

All of the 104 clone groups were reviewed by the participating developers. Despite the approximately unbiased nature of the modified Horvitz-Thompson (HT) estimator we used [129], its results can still be skewed if there is high variance in the inclusion probabilities. The best way to address such skew is to collect more data. As this was not possible the next best option of removing the outliers was taken. Thus two of the clone groups with outlying selection probabilities were discarded.

[4] http://faculty.cse.tamu.edu/davis/suitesparse.html.
See Davis' 2004 paper for details [33].

1. Do the highlighted portions of the code fragments, in conjunction with the associated graph, represent duplicated, similar or cloned code?
   (a) Yes
   (b) No
2. If you answered **Yes** to question 1, would you:
   (a) Create a story card to refactor this code?
   (b) Add a comment to consider refactoring on next change?
   (c) Add a note about duplicate code even if it cannot be refactored?
   (d) Ignore it?
   (e) Take some other action?

Figure 4.3: The two critical survey questions

For survey question 1, the estimate of the proportion of all mineable dependence clones for which the answer would be **Yes** if they were all examined was 61%, with a 95% confidence interval of $42\% - 78\%$. For survey question 2, the estimate of the proportion of all mineable clones for which one of the "action answers" (2.a, 2.b, 2.c, and 2.e) would be given if all of the clones were examined was 14% (which is smaller than the sample proportion of 33%), with a 95% confidence interval of $0\% - 33\%$.

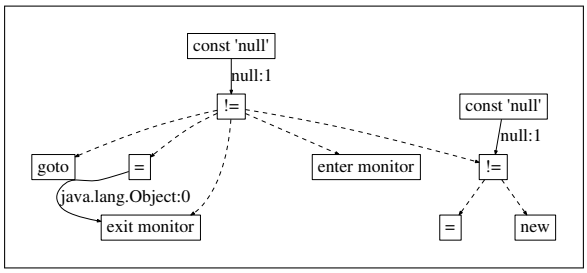
It should be emphasized that the results for survey question 1 do not actually mean that our frequent subgraph miner produced erroneous results 39% of time. It did in fact identify only 5-frequent PDG subgraphs. However, the developers had their own subjective criteria for deciding whether the corresponding code was duplicated. For example, Figure 4.4 shows two simplified clones sampled from the application. Figure 4.4a was identified by the developer who reviewed it as the *double-checked locking* pattern [130]. Although widely used throughout the code base, the reviewer indicated that it was too general to be a code clone, because each instance exhibited context-specific specialization. Figure 4.4d represents a class of different clones involving user-interface state modifications. In this clone, the color of a button is changed based on the current theme. As all elements of the application are themed to support multiple brands of the program, this code was duplicated in many locations (often non-contiguously). The reviewers recommended it for refactoring to centralize the theming decision.

Thus, the results for survey questions indicate that (a) about 61% of mineable dependence clones in the project would be judged by developers to be duplicate code and (b) the developers would want to take action for only about 14% of mineable clones. Assuming that the developers' judgments about the clones were justified, the difference between these two estimates suggests that for this project, additional filtering is needed to eliminate clones that are not of interest to developers. Further study of the sampled dependence clones and discussions with the developers about them might suggest what filtering criteria are needed. Note that in developers' responses to a followup questionnaire in our study, they indicated that they felt the exercise was useful and that they would like to periodically review new findings from a PDG-based tool.

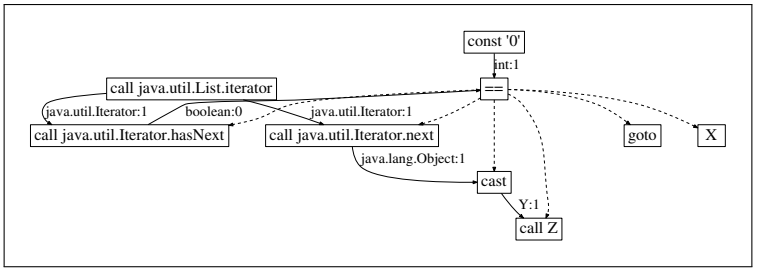
Further investigations are needed to fully understand these results in the context of other clone detection techniques. Unlike many detectors ours did not employ any filtering or normalization heuristics, making direct comparisons to previous results difficult. In future studies, GRAPLE could be applied to AST and token representations, allowing a direct comparison. The effect of filtering and normalization can also be estimated using GRAPLE.

4.5 Conclusion

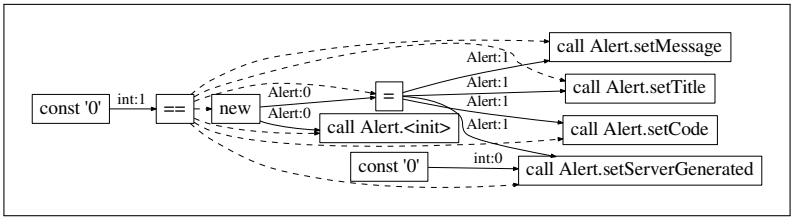
We have presented GRAPLE, a framework for randomly sampling unique frequent subgraph from directed labeled graphs. Our sampling method enables unbiased estimation of statistics characterizing the whole population of frequent subgraphs (without enumerating it). The results of our case study suggest that GRAPLE will prove useful to software engineering researchers and to developers who apply advanced analytical methods to better understand large code bases. In future work, we plan to estimate the sample inclusion probabilities needed for Horvitz-Thompson estimation rather than computing them exactly to enable studies involving larger patterns.



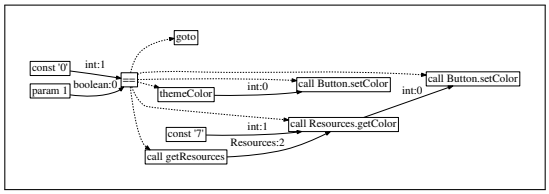
(a) Double Checked Locking. Used to create singletons. Not considered a code clone by the reviewer.



(b) An iterator protocol generated from a for-each loop. Not considered a code clone by the reviewer.



(c) Creating an application specific `Alert` object. Considered a code clone by the reviewer.



(d) Setting a button color depending on theme choice. Considered a code clone by the reviewer.

Figure 4.4: Four clones discovered in the study. Package names have been removed and application details have been obfuscated.

Chapter 5

Rethinking Dependence Clones

5.1 Introduction

Fragments of similar code are typically scattered throughout large code bases [125]. These repeated fragments or *code clones* often result from programmers copying and pasting code. Code clones (or just *clones*) may also result from limitations of a programming language, use of certain APIs or design patterns, following coding conventions, or a variety of other causes. Whatever their causes, existing clones need to be managed. When a programmer modifies a region of code that is cloned in another location in the program, they should make an active decision whether or not to modify the other location. Clearly, such decisions can only be made if the programmer is aware of the other location.

In general, there are 4 types of code clones [125]:

Type-1 Clones – Identical regions of code (excepting whitespace and comments).

Type-2 Clones – Syntactically equivalent regions (excepting names, literals, types, and comments).

Type-3 Clones – Syntactically similar regions (as in Type-2) but with minor differences such as statement additions or deletions.

Type-4 Clones – Regions of code with functionally equivalent behavior but possibly with different syntactic structures.

Much of the research on code clone detection and maintenance has been geared toward Type-1 and Type-2 clones [16, 119, 124, 125], as they are easier to detect and validate than Type-3 and Type-4 clones. The two most popular detection methods involve searching for clones in *token streams* [79, 128] and *abstract syntax trees* (ASTs) [72], respectively.

An alternative approach to clone detection is to search for them in a *Program Dependence Graph* (PDG) [46], which represents the control and data dependences between statements or operations in a program. Recurring subgraphs in PDGs represent potential *dependence clones* (see Figure 5.1 on page 47, which is examined in Section 5.2). Some of the previous work [61, 83, 85] on PDG-based clone detection used forward and backward path-slicing to find clones. This method can detect matching slices, but it cannot detect all recurring

[1] Note, portions of this chapter originally appeared as [60] https://doi.org/10.1109/IWSC.2017.7880512.

subgraphs. The latter can be identified using *frequent subgraph mining* (FSM) [87]. However, for low frequency thresholds, the number of PDG subgraphs discovered by FSM may be enormous. For example, we found that for a Java program with 70,000 lines of code (LOC), over 700 million PDG subgraphs with 5 or more instances were discovered by FSM.

Since it is infeasible for developers to examine so many subgraphs, we previously developed GRAPLE [59], an algorithm to select representative samples of maximal frequent subgraphs. In this paper, the core sampling process remains the same as in GRAPLE but we present a new algorithm for traversing the k-frequent subgraph lattice (see Section 5.3). One tricky aspect of FSM is how to define exactly what "frequency" means in a large connected graph [19]. In order to handle pathological cases that occur in real programs, we introduce a new metric to measure subgraph frequency (or "support"), called the *Greedy Independent Subgraphs* (GIS) measure. Section 5.4 details the first empirical examination of the scalability and speed of sampling dependence clones from large programs. The study showed that our new system can quickly sample from programs with 500 KLOC of code and successfully sample from programs with perhaps 2 MLOC. Finally, since at times the sampling algorithm may return several potential clones, which are quite similar to each other, we evaluate the performance of a density-based clustering algorithm on the samples collected.

5.2 A Motivating Example

Figure 5.1 shows an example *dependence clone* extracted from jGit, a Java implementation of the Git version control system. The clone was identified from the PDG of jGit produced by our PDG-generator `jpdg` [59]. PDG-based clone detection techniques, unlike techniques based on syntactic representations, do not distinguish between code fragments that differ only because of dependence-preserving statement re-orderings, which also preserve semantics [114]. Hence, using only static information, they detect semantically-equivalent clones that are not detected by syntactically-based techniques. (Of course, they cannot detect all semantically-equivalent code fragments.)

Three functions are shown in Figure 5.1 (two of them are partial) that parse PATCH files. The PATCH file format is a plain text format describing the differences (line by line) between two versions of a piece of software. There are several varieties of PATCH files including: "Traditional", "Combined", and "GIT", all of which jGit can parse. Both the Combined and GIT formats are supersets of the Traditional format, which leads to some amount of duplication, especially involving headers. Figure 5.1(d) shows a graph pattern "explaining" the highlighted duplication in the three functions in terms of a subgraph of jGit's PDG, which represents code detecting the start of a Traditional PATCH header.

However, the duplication in Figure 5.1 isn't due to a simple copy-paste. Each function contains unique context-specific code interleaved with portions that detect the start of a Traditional header. Figure 5.1(a) contains the function `parseTraditionalHeaders`, which uses the pattern to drive the parser to extract the changed filenames. Function `parseFile` in Figure 5.1(b) drives the parsing process for all file types and uses the pattern to detect the start of each "hunk." In contrast to the function in 5.1(a), `parseFile` reorders some of the statements and interleaves significant new functionality between statements. Finally, function `parseHunks` in Figure 5.1(c) exhibits a third statement ordering distinct from the first two. It uses the pattern to detect when it should stop parsing the current hunk.

Figure 5.1 illustrates that dependence-based clone detection can discover subtle pro-

gramming patterns that are difficult to detect using other program representations due to differences in the ordering of statements within pattern instances and to the interleaving of statements from within the pattern with other statements.

5.3 Sampling Dependence Clones

Dependence clones can be found by searching for "frequent" subgraphs of a PDG. An advantage of using FSM, rather than some other data mining techniques, is that lower frequency thresholds (e.g., 2-5) can be used without increasing the number of irrelevant patterns that are found [22]. This is because FSM helps ensure that the statements in a pattern are semantically related – not just co-occurring.

The process of finding frequent subgraphs of G can be viewed as a traversal of a *lattice* of subgraphs. The subgraph relation $\cdot \sqsubseteq \cdot$ induces a *Connected Subgraph Lattice* $\mathcal{L}_G$ representing all the possible ways of constructing G from the empty subgraph by adding one edge at a time. $\mathcal{L}_G$ is a digraph where each vertex u represents a unique connected (ignoring edge direction) subgraph of G. There is an edge from u to v in $\mathcal{L}_G$ if adding some edge ϵ to u creates a subgraph $u + \epsilon$ of G that is isomorphic to v, $v \cong u + \epsilon$. The *k-Frequent Connected Subgraph Lattice* k-$\mathcal{L}_G$ is a Connected Subgraph Lattice containing only those subgraphs that are at least k frequent in G according to some support measure σ.

The most natural definition of the support measure is $\sigma(\llbracket H \rrbracket_G) = |\llbracket H \rrbracket_G|$, i.e., the number of unique embeddings in H's isomorphism class. Unfortunately, this definition does not satisfy an important property of suitable support measures, called the *Downward Closure Property* [19]. A measure that is commonly used instead is *Minimum Image Support* (MNI) [19]. However, some pathological cases involving patterns with *automorphisms* (non-trivial isomorphisms from a subgraph to itself) and with overlapping embeddings can cause exponential-time computations when MNI is used. We have found that these actually occur in real programs. To circumvent this problem we use *an unsound support measure* called *Greedy Independent Subgraphs* (GIS).

An embedding n is *directly connected* to another embedding m if any vertex in n is also used in m. The embedding n is *connected* (possibly indirectly) to another embedding x if there is some sequence of embeddings $y_1 ... y_n$ such that: the embedding n is directly connected to the embedding y_1, y_1 is directly connected to y_2, and so on, until y_{n-1} is directly connected to y_n and y_n is directly connected to x. An embedding n is said to be *independent* of an embedding x if they are not connected by any sequence of embeddings. Recall that automorphic patterns with overlapping embeddings cause problems for MNI since there may be millions of unique embeddings. However, one can short-circuit this computation by computing the number of independent groups of embeddings in a greedy fashion. This yields the metric GIS, which is implemented by the function `gis_pruner` in Listing 5.1.

5.3.1 Sampling Frequent Subgraphs

Sampling N frequent subgraphs can be accomplished by an unweighted forward random walk on the connected frequent subgraph lattice [59]. The process is outlined in Listing 5.1 and is based on our previous work on GRAPLE [59]. Note the listing is in Python for brevity but the actual implementation is in the Go programming language. The core sampling process (lines 1-9) remains the same as in GRAPLE but there is a new algorithm for traversing the

```
1  int parseTraditionalHeaders(int ptr, final int end) {
2    while (ptr < end) {
3      final int eol = nextLF(buf, ptr);
4      if (isHunkHdr(buf, ptr, eol) >= 1) {
5        // First hunk header;
6        break; // break out and parse them later.
7      } else if (match(buf, ptr, OLD_NAME) >= 0) {
8        parseOldName(ptr, eol);
9      } else if (match(buf, ptr, NEW_NAME) >= 0) {
10       parseNewName(ptr, eol);}
11     } else {
12       // Possibly an empty patch.
13       break;
14     }
15     ptr = eol;
16   }
17   return ptr;
18 }
```

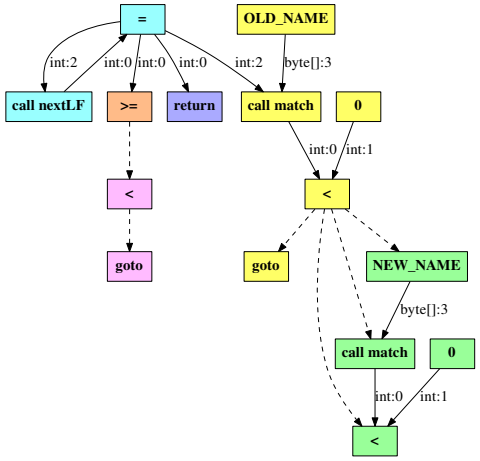
(a) org.eclipse.jgit.patch.FileHeader (line 496)

```
1  int parseHunks(final FileHeader fh, int c, final int end) {
2    final byte[] buf = fh.buf;
3    while (c < end) {
4      // If we see a file header at this point, we have
5      // all of the hunks for our current file. We should
6      // stop and report back with this position so it
7      // can be parsed again later.
8      if (match(buf, c, DIFF_GIT) >= 0)
9        break;
10     if (match(buf, c, DIFF_CC) >= 0)
11       break;
12     if (match(buf, c, DIFF_COMBINED) >= 0)
13       break;
14     if (match(buf, c, OLD_NAME) >= 0)
15       break;
16     if (match(buf, c, NEW_NAME) >= 0)
17       break;
18
19     if (isHunkHdr(buf, c, end) == fh.getParentCount()) {
20       // OMITTED: Parses the hunk
21       continue;
22     }
23     final int eol = nextLF(buf, c);
24     if (fh.getHunks().isEmpty()
25         && match(buf, c, GIT_BINARY) >= 0) {
26       fh.patchType = FileHeader.PatchType.GIT_BINARY;
27       return parseGitBinary(fh, eol, end);
28     }
29     if (fh.getHunks().isEmpty()
30         && BIN_TRAILER.length < eol - c
31         && match(buf, eol-BIN_TRAILER.length, BIN_TRAILER)
32                                                 >= 0
33         && matchAny(buf, c, BIN_HEADERS)) {
34       // The patch is a binary file diff, with no deltas.
35       fh.patchType = FileHeader.PatchType.BINARY;
36       return eol;
37     }
38     c = eol;  // Skip this line and move to the next.
39   }
40   // OMITTED: Check for empty patch which might be binary.
41 }
```

(c) org.eclipse.jgit.patch.Patch (line 272)

```
1  int parseFile(byte[] buf, int c, final int end) {
2    while (c < end) {
3      if (isHunkHdr(buf, c, end) >= 1) {
4        //Needs header
5        error(buf, c,
6            JGitText.get().hunkDisconnectedFromFile);
7        c = nextLF(buf, c);
8        continue;
9      }
10     // OMITTED: Valid git style patch?
11     final int n = nextLF(buf, c);
12     if (n >= end) {
13       // Patches cannot be one line long
14       return end;
15     }
16     if (n - c < 6) {
17       // A valid header is >= 6 bytes
18       c = n;
19       continue;
20     }
21     if (match(buf, c, OLD_NAME) >= 0
22         && match(buf, n, NEW_NAME) >= 0) {
23       // Probably a traditional patch. check "@@ -"
24       final int f = nextLF(buf, n);
25       if (f >= end)
26         return end;
27       if (isHunkHdr(buf, f, end) == 1)
28         return parseTraditionalPatch(buf, c, end);
29     }
30     c = n;
31   }
32   return c;
33 }
```

(b) org.eclipse.jgit.patch.Patch (line 172)



(d) Graph pattern explaining the code duplication

Figure 5.1: The highlighted regions above illustrate semantic code duplication in jGit (commit `efd91ef8a`), which was not found by the clone detector CCFinderX [79]. jGit is a Java implementation of the Git version control system. The three functions shown parse portions of PATCH files. The function in 5.1a parses the header of traditional PATCH files. The function in 5.1b (which has portions removed) parses all types of PATCH files used by Git (and "drives" the rest of the parsing functions). The function in 5.1c (which has portions removed) parses the hunks from the PATCH file looking for headers to signal the start of the next hunk. The graph in 5.1d is a frequent subgraph found in the PDG of jGit (produced by `jpdg` from JVM bytecode) and is a subgraph of the pDGs of these functions. In the graph, dotted lines represent control dependences. Solid lines represent data dependences and are annotated with their types and usage context.

```
def sample(N, graph, min_support):
    for _ in xrange(N):
        yield unweighted_random_walk(G, min_support)

def unweighted_random_walk(G, min_support):
    prev = cur = G.root_lattice_node(min_support)
    while cur is not None:
        n = randchoose(cur.children()); prev = cur; cur = n
    return prev

class LatticeNode(object):
    def __init__(self, G, sg, embs, exts):
        self.graph = G
        self.subgraph = sg
        self.supported_embeddings = embs
        self.extensions = exts
    def children(self):
        G = self.graph
        for sg_ext in self.extend():
            support, exts, embs = exts_and_embs(G, sg_ext)
            if support >= min_support:
                yield LatticeNode(G, sg, embs, exts)
    def extend(self):
        exts = set()
        for ext in self.extensions:
            exts.add(self.subgraph.extend(ext))
        return exts

def exts_and_embs(G, sg):
    embs = list(); exts = set(); seen = set()
    for emb in find_embeddings(G, sg, gis_pruner(seen)):
        for emb_idx in embs.idxs:
            for edge in G.children_of(emb_idx):
                add_ext(G, exts, emb, edge, emb_idx, -1)
            for edge in G.parents_of(emb_idx):
                add_ext(G, exts, emb, edge, -1, emb_idx)
        embs.append(emb)
    return gis_support(embs), exts, embs

def find_embedddings(G, sg, prune_fn=None):
    edges = spanning_tree(G, sg)
    for edge in sg.edges:
        if edge not in edges: edges.append(edge)
    stack = list()
    vembs = vertex_embeddings(G, sg, edges[0].src)
    for emb_idx in vembs:
        stack.append((ListNode(start_idx, emb_idx), 0))
    while len(stack) > 0:
        cur, eid = stack.pop()
        if prune_fn is not None and prune_fn(cur):
            continue
        if eid >= len(spanning_edges):
            yield embedding_from_ids(cur)
        else:
            for n in extend_embedding(G,sg,cur,edges[eid])
                stack.append((n, eid+1))

def gis_pruner(seen):
    def gis_prune(cur):
        for n in cur:
            if n.emb_idx in seen:
                for m in cur: seen.add(m.emb_idx)
                return True
        return False
    return gis_prune
```

Listing 5.1: Sample N k-frequent subgraphs.

k-frequent subgraph lattice. The new algorithm incorporates the GIS support metric and pruning strategy into its subgraph matching algorithm `find_embeddings`. The algorithm also merges the support and candidate extension computations in `exts_and_embs`.

In Listing 5.1, the function `sample` does N walks over the frequent subgraph lattice by calling `unweighted_random_walk` repeatedly. Each walk starts at a `LatticeNode` representing the empty subgraph. In every step of the walk a call is made (line 8) to `LatticeNode.children()`, which computes frequent super-graphs of the graph represented by the current `LatticeNode`. One super-graph is selected at random to use for the next step. The walk terminates when there are no frequent super-graphs. The `children` method computes candidate frequent super-graphs using the `extend` method. Each candidate is computed from an "extension", which is obtained by adding a single edge to the graph represented by the `LatticeNode`. After the candidates are computed, each one must be checked to see if it is frequent. This is done by the `exts_and_embs` function, which computes the frequency (support), the candidate one-edge extensions, and the supported embeddings for the candidate frequent subgraph.

To compute the support for a subgraph, all of its embeddings need to be found. This is implemented by the `find_embeddings` method, which solves the subgraph matching problem. This method implements a back-tracking tree search procedure. The nodes in the search are partial subgraph isomorphism mappings. The search tree for a subgraph H has height $|E_H|$, where $|E_H|$ is the number edges in H. The search starts at a node mapping a single vertex in H to some vertex with the same label in the graph G. It proceeds edge by edge, building up a mapping until a full mapping is obtained or back-tracking is performed by discarding the current mapping and considering another partial mapping. If the `gis_pruner` is supplied as the pruning function, `prune_fn`, to `find_embeddings` then on line 50 there is a chance to discard the current mapping. GIS discards the mapping if any of the mapped vertices in it have already appeared in a completed mapping.

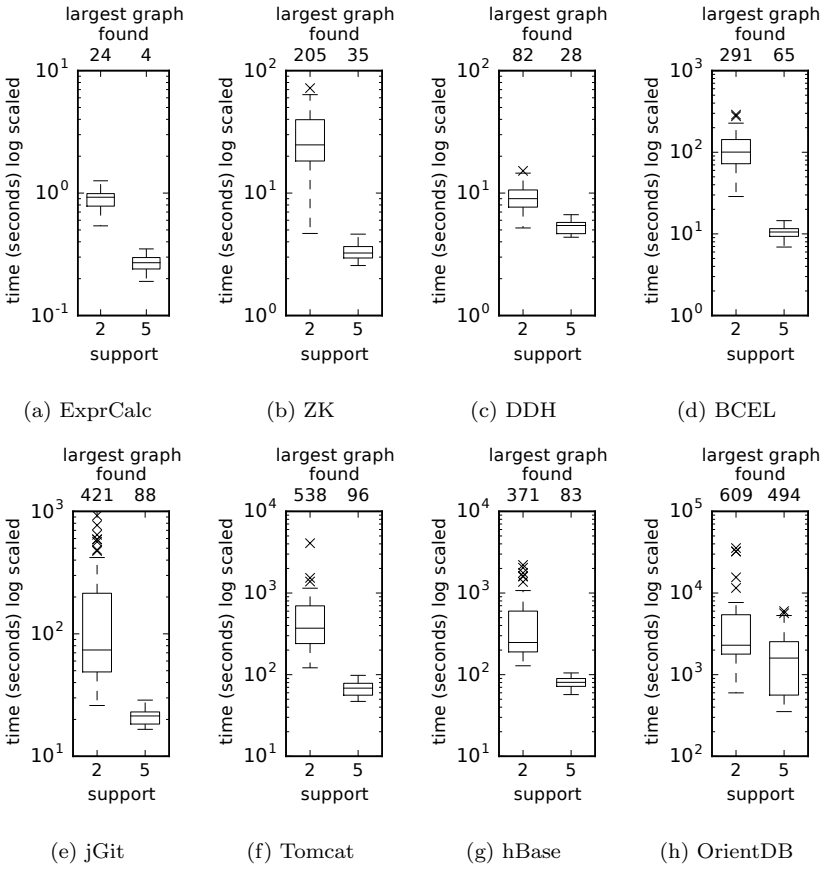
While not shown in the listing, the implementation contains several other optimizations to the embedding search. For instance, it uses a lightweight index to ensure that candidate vertices used to extend the current embedding have at least the degree of the matching subgraph vertex.

5.4 Evaluation

A new dependence-clone sampling system was implemented using the ideas described in Section 5.3.1, and it was evaluated in an empirical study. The study examined samples of 100 potential code clones from the eight subject programs described in Table 5.1. The

Table 5.1: The datasets used in the study. KLOC – Kilo Lines of Code.

Dataset	KLOC	Nodes	Edges	Description
ExprCalc	0.8	1,110	2,162	Arithmetic calculator
Zookeeper	32.4	17,028	32,691	Distributed KV store
DDH	19.3	36,384	65,874	Anonymized prorietary application
BCEL	28.6	52,731	108,542	JVM bytecode lib.
jGit	72.1	136,716	300,550	GIT in Java
Tomcat	220.7	377,657	806,824	Web server
hBase	561.4	442,063	981,577	Database
OrientDB	N/A	2,022,640	3,476,158	Database & all dependencies.



(a) ExprCalc (b) ZK (c) DDH (d) BCEL

(e) jGit (f) Tomcat (g) hBase (h) OrientDB

Figure 5.2: Execution time to sample 100 dependence clones from the subject programs. Box and whisker diagrams of execution time in seconds of 50 runs of the Unweighted Random Walk sampling algorithm; each run collected a sample containing 100 frequent subgraphs. The number on the top of the axis is the size (in # of edges) of the largest dependence clone found. The number on the bottom axis is the minimum support (frequency) used.

sample size was chosen to represent the maximum number of clones a single developer would likely be able to review in a day. The study considered code cloned in at least five locations as well as code cloned in at least two locations, because code that is duplicated in more locations is potentially more relevant to the programmer and can be found more quickly with pattern mining methods. Since every sample collection run collects a different selection of 100 potential clones, there is variance in the amount of time it takes to collect the sample. The variance was measured by collecting 50 independent samples with 100 frequent subgraphs in each sample (see Figure 5.2).

To look at the scalability of the new implementation on larger programs, OrientDB was examined with the addition of *all* of its library dependencies – including generated code. Thus, the OrientDB PDG includes code from both the application and all of the non-application libraries. Since the libraries were delivered as JVM bytecode no line count could be determined. OrientDB generated around four times the number of vertices as the next largest program – so the total line count of OrientDB and its dependencies may be as large as 2 MLOC.

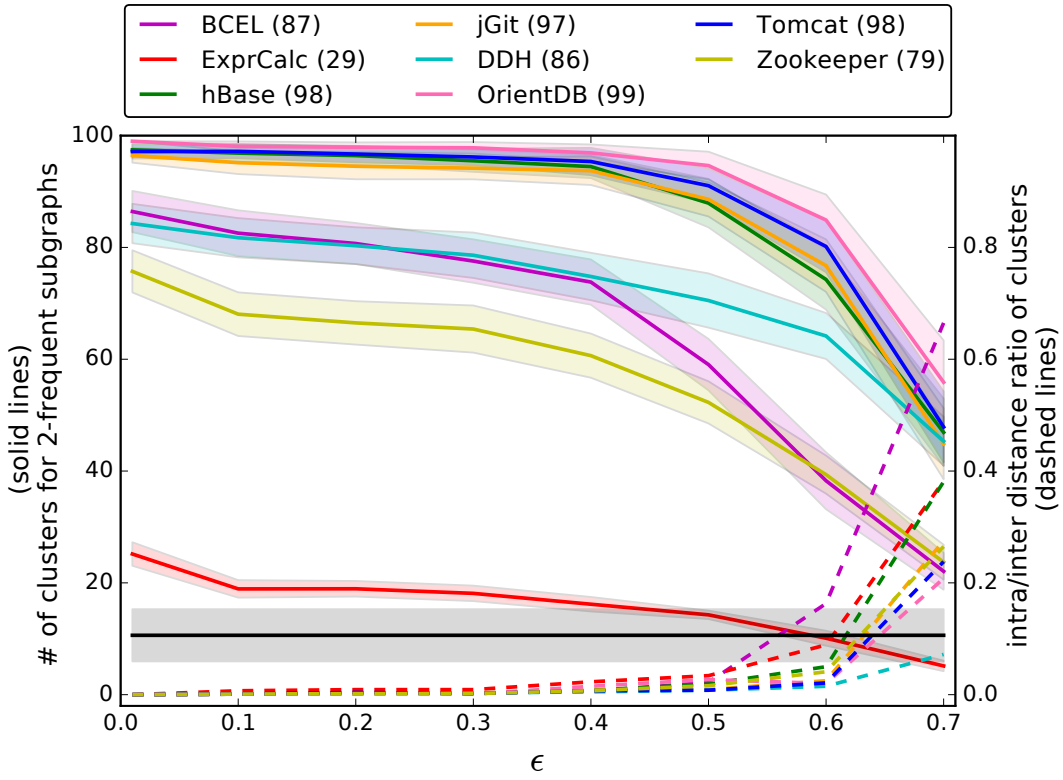


Figure 5.3: The mean number of clusters for samples of 2-frequent subgraphs (solid lines; variances are shown as shaded regions), for various choices of ϵ. Items α and β cluster together if $\delta(\alpha, \beta) < \epsilon$ where δ is a distance metric between the items. The number next to each dataset name is the average number of unique items sampled when sampling (with replacement) 100 items total. The dashed lines show the intra/inter cluster distance ratio – the closer it is to 0 the better the clustering. The solid black line shows this ratio for random clustering (the shaded area indicates the variance). If a dashed line is below the black line it is better than random.

The PDGs used in the study were generated by our tool `jpdg` [59]. The evaluation was conducted on a dual-socket server with 2010 Intel Xeon X5650 CPUs and 96 GB of memory. Despite the large amount of memory on the server used to collect the timing data, all computations ran well on a laptop with 16 GB of memory. All used under 4 GB of RAM except for those involving the OrientDB dataset, which required around 10 GB of RAM. The memory usage was dominated by storage for the graph index and varied only slightly during the actual sampling procedure.

It took under 2 minutes to collect a sample of 100 five-frequent subgraphs for all of the programs except OrientDB (Figure 5.2). For the smaller programs, collecting a sample required less than 10 seconds – making our new implementation suitable for desktop usage. For the smallest program, ExprCalc, we were able to find all of the frequent subgraphs ($\sim$2.5 million at minimum support 2). The rest of the programs had too many frequent subgraphs (>100 million) for us to mine them all at minimum support 2 or 5. At support level 5 it yields only frequent subgraphs containing at most 4 edges. Larger 5-frequent subgraphs were found for rest of the subject programs, ranging in size from 35 edges for Zookeeper to 96 edges for Tomcat to 494 edges for OrientDB (see the top axis in Figure 5.2).

Programmers are potentially interested in regions of code duplicated in just one other location. As shown in Figure 5.2, the execution time required to collect a sample of 100 2-frequent subgraphs was greater than the time for 5-frequent subgraphs. The longest amount of time spent collecting a sample for any program except OrientDB was 67 minutes, while the highest mean time was $8\frac{3}{4}$ minutes. OrientDB took much longer — as long as 10 hours. Recall that OrientDB is four times larger than the next largest program, hBase, which has 500 KLOC. Clearly, there is work left to be done on the scalability of PDG-based clone detection. In the future, when detecting clones for code review only the ones related to changed code need to be found. Performance may also be improved by using a better indexed subgraph matching algorithm [159] and incrementally updating the PDGs [62, 120].

Another issue with code clones in general, but especially with clones found from graphical representations such as PDGs, is reporting multiple clones that are very similar to each other. These "clone families" arise naturally since many frequent subgraphs share common subgraphs with other frequent subgraphs. When reviewing code clones, programmers do not want to see very similar clones over and over again. Future PDG-based clone detection systems should address this problem. Towards that end, Figure 5.3 evaluates a simple density based clustering algorithm, DBSCAN [43] (with no minimum number of items allowed in a cluster). In density based clustering, items α and β cluster together if the distance between them, according a metric δ, is less than ϵ: $\delta(\alpha, \beta) < \epsilon$. In Figure 5.3 the δ metric is the Jaccard set similarity coefficient applied to the sets of vertex labels of two subgraphs. Thus, subgraphs that contain the same combination of operators, method calls, and constants are placed together.

DBSCAN identified sizable "tight" clusters (as measured by the intra/inter cluster distance ratio) of clones for 4 of the subject programs (ExprCal, Zookeepr, BCEL, and DDH in Figure 5.3). These clusters indicate the presence of clone families, which we confirmed with visual inspection. Identifying these clusters reduces the programmer effort needed to review the set of potential clones, since only a representative from the cluster needs to be reviewed by the programmer. For the other 4 programs (jGit, Tomcat, hBase, and OrientDB) most of the sampled frequent subgraphs were distinct from each other, and cluster quality was poor when they were grouped together (as can be seen in the figure, for the higher settings for ϵ). Future work could integrate an online clustering technique into the sampling procedure to ensure adequate diversity in the output. In addition, there are many other similarity measures for graphs [122] and some of them such as graph kernels can take into account the structure of graphs. Such measures may perform better for this application and their utility for PDG-based clone-detection should be evaluated.

5.5 Conclusion

We have presented a new algorithm for sampling potential code clones from program dependence graphs, using unweighted random walks over the frequent connected subgraph lattice. The algorithm uses the greedy independent subgraphs measure to prune the subgraph-matching search space, which reduces the computation costs for difficult-to-mine program graphs. Empirical results were presented that demonstrated that the algorithm is capable of mining large programs. For programs with at least 500,000 LOC it can sample clones fast enough to be used either on the desktop or in a continuous integration system for use during code review. The algorithm presented does not use any heuristics or limit the size of frequent subgraph found. Results were also presented for the effectiveness of using density

based clustering on the returned clones. For half of the programs significant clusters were found. The time has come to reconsider PDG-based clone detection as part of a holistic strategy of clone management and to develop clone management systems that integrate multiple detection strategies.

Chapter 6

Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs

6.1 Introduction

Automated fault localization techniques have been developed to help programmers locate software faults (bugs) responsible for observed software failures. Many of these techniques are statistical in nature (e.g., [75, 92, 94]). They employ statistical measures of the *association*, if any, between the occurrence of failures and the execution of particular program elements like statements or conditional branches. The program elements that are most strongly associated with failures are identified as "suspicious", so that developers can examine them to see if they are faulty. The association measures that are used are often called *suspiciousness metrics* [74]. Such *statistical fault localization* (SFL) techniques typically require *execution profiles* (or *spectra*) and PASS/FAIL labels for a set of both passing and failing program runs. Each profile entry characterizes the execution of a particular program element during a run. For example, a statement-coverage profile for a run indicates which statements were executed at least once. The profiles are collected with program instrumentation, while the labels are typically supplied by software testers or end users.

Kochar *et al.* [82] recently surveyed 386 software engineering practitioners about their expectations for automated fault localization. They found both strong demand for fault localization solutions and high barriers to adoption. Foremost among the barriers was a requirement for high localization accuracy. More than 80% of respondents considered a fault localization session successful only if a fault is found among the top five suggested locations. Seventy-five percent of respondents felt that fault localization should be successful at least 75% of the time. Scalability was also important to respondents. Seventy-five percent of respondents felt that a fault localization technique should scale to programs of at least 100,000 SLOC. Finally, over 85% of respondents felt it was important for a fault localization technique to help programmers to comprehend its results.

To address these requirements for automated fault localization, we present a new algorithm, called *Score-Weighted Random Walks* (SWRW), for "behavioral" fault-localization. Behavioral fault localization techniques (e.g. [24]) extend basic SFL techniques so as to

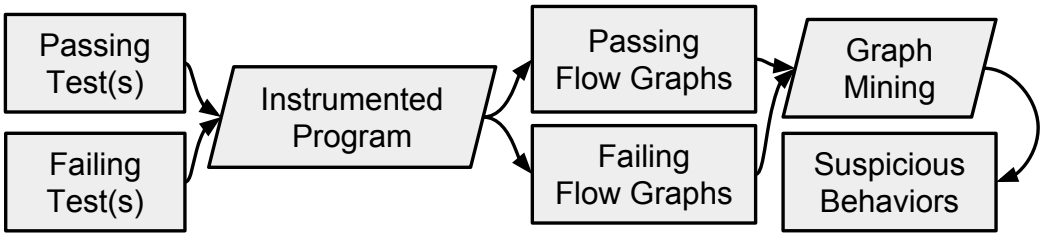


Figure 6.1: Process for localizing faults with discriminative graph mining.

localize faults within a behavior, which is a context formed of interacting program elements such as basic blocks or functions. Behavioral fault localization combines evidence from simple program elements and potentially makes it easier for programmers to comprehend localization results. Our new algorithm SWRW belongs to a family of *discriminative graph-mining algorithms* that have previously been used for behavioral fault localization [24, 34–37, 94, 100, 103, 111, 156]. Figure 6.6 illustrates the process of fault localization with discriminative graph mining.

Discriminative graph mining algorithms are very powerful in principle but they must make tradeoffs to address the challenging combinatorics of the graph mining problem. Our new algorithm SWRW does this by randomly sampling "suspicious" subgraphs from basic-block level dynamic control flow graphs collected during the execution of passing and failing tests. During the sampling process, the most suspicious subgraphs are favored for selection. SWRW provides more accurate fault localization than similar algorithms, as we demonstrate in an empirical study. Unlike those algorithms, SWRW can be used with a variety of suspiciousness metrics.

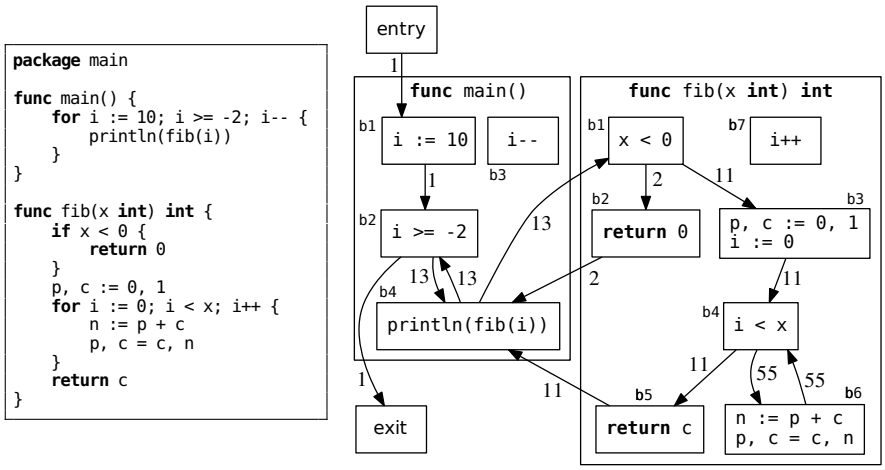
Summary of Contributions

1. A new behavioral fault localization algorithm, SWRW, that samples suspicious subgraphs from dynamic control flow graphs collected from passing and failing executions. Unlike similar algorithms, SWRW can be used with a variety of suspiciousness metrics.
2. New generalizations of existing suspiciousness metrics that allow them to be applied to behaviors represented by subgraphs of dynamic control flow graphs.
3. Dynagrok, a new instrumentation, mutation, and analysis tool for the Go programming language.
4. An empirical study whose results suggest that SWRW is more accurate than similar algorithms.

6.2 Dynagrok: A New Profiling Tool

All coverage based statistical fault localization (CBSFL) techniques use *coverage* profiles to gather information on how software behaved when executed on a set of test inputs. A coverage profile typically contains an entry for each program element of a given kind (e.g., statement, basic block, branch, or function), which records whether (and possibly how many times) the element was executed during the corresponding program run. The profiles and PASS/FAIL labels for all tests are then used to compute a statistical *suspiciousness score* for each program element.

The process of gathering the coverage information from running programs is called *profiling* and there are many different varieties of profilers and profiling techniques available.




```
package main

func main() {
    for i := 10; i >= -2; i-- {
        println(fib(i))
    }
}

func fib(x int) int {
    if x < 0 {
        return 0
    }
    p, c := 0, 1
    for i := 0; i < x; i++ {
        n := p + c
        p, c = c, n
    }
    return c
}
```

```
package main

import "dgruntime"

func main() {
    defer func() { dgruntime.Shutdown() }()
    dgruntime.EnterFunc("main")
    defer func() { dgruntime.ExitFunc("main") }()
    for i := 10; dgruntime.EnterBlkFromCond(2) && i >= -2; i-- {
        dgruntime.EnterBlk(4)
        println(fib(i))
    }
}

func fib(x int) int {
    dgruntime.EnterFunc("fib")
    defer func() { dgruntime.ExitFunc("fib") }()
    if x < 0 {
        dgruntime.EnterBlk(2)
        return 0
    }
    dgruntime.EnterBlk(3)
    p, c := 0, 1
    for i := 0; dgruntime.EnterBlkFromCond(4) && i < x; i++ {
        dgruntime.EnterBlk(6)
        n := p + c
        p, c = c, n
    }
    dgruntime.EnterBlk(5)
    return c
}
```

Figure 6.2: An example dynamic control flow graph (DCFG) for a Go program (listing on top right) that computes elements from the Fibonacci sequence. Each vertex is a basic block with a basic block identifer (e.g. `b1`) that, in conjunction with the name of the containing function, serves as the label for the block (e.g. `main:b1`). Each edge shows the number of traversals taken during the execution of the program. Note that the loop update blocks (`main:b3` and `fib:b7`) will be not in the profiles because Dynagrok instruments the Go source code and profiling instructions cannot be syntactically inserted in those locations. The instrumented program is shown in the bottom listing.

Coverage profiling is a simple and widely implemented technique, which is why it has been widely used by the fault localization community. Another technique is *tracing*, which logs the sequence of program locations as they are executed. The traces provide detailed information on the behavior of the program but could grow to be very large for long running programs. This paper uses *execution flow profiling* which computes the dynamic interprocedural control flow graph of a program's execution. This provides some of the benefits of tracing without recording an excessive amount of data.

To capture execution flow profiles we developed *Dynagrok*, a new analysis, instrumentation and mutation platform for the Go programming language. Go is a newer language (2009) from Google that has been seeing increasing adoption in industry. It has been adopted for web programming, systems programming, "DevOps," network programing, and databases.[1] Dynagrok builds upon the *abstract syntax tree* (AST) representation provided by the Go standard library.

Dynagrok collects profiles by inserting instrumentation into the AST of the subject program. The profiles currently collected are *dynamic control flow graphs* (DCFGs) whose vertices represent basic blocks. A *basic block* is a sequence of program operations that can only be entered at the start of the sequence and can only be exited after the last operation in the sequence [6]. A basic-block level *control flow graph* (CFG) is a directed labeled graph $g = (V, E, l)$ comprised of a finite set of vertices V, a set of edges $E \subseteq V \times V$, and a labeling function l mapping vertices and edges to labels. Each vertex $v \in V$ represents a basic block of the program. Each edge $(u, v) \in E$ represents a transition in program execution from block u to block v. The labeling function l labels the basic blocks with a unique identifier (e.g. `function-name:block-id`), which is consistently applied across multiple executions but is never repeated in the same execution.

Figure 6.2 shows an example DCFG collected by Dynagrok for a simple program that computes terms of the Fibonacci sequence. To collect such graphs Dynagrok parses the program into an AST using Go's standard library. Dynagrok then uses a custom control flow analysis to build static control flow graphs. Each basic block holds pointers to the statements inside of the AST. The blocks also have a pointer to the enclosing *lexical block* in the AST. Using this information, Dynagrok inserts profiling instructions into the AST at the beginning of each basic block. The instructions inserted by Dynagrok use its `dgruntime` library to track the control flow of each thread (which is called a *goroutine* in Go). When the program shuts down (either normally or abnormally) the `dgruntime` library merges the flow graphs from all the threads together and writes out the result.

6.3 From Suspicious Locations to Suspicious Behaviors

Before explaining our notion of a suspicious behavior we will review the concept of suspicious statements, blocks, or other locations in a program. There is a large body of work on coverage based statistical (also called spectrum-based) fault localization (CBSFL) (e.g., [1, 12, 13, 26, 75, 76, 88, 99, 133, 138, 155]), which identifies suspicious program locations from code coverage profiles collected from passing and failing program runs. All of this work tries to assess the suspiciousness of a particular program element based on a statistical measure of the association between coverage of the element and the occurrence of program failure.

A simple measure of the suspiciousness of a program location l is the probability $\Pr[F|l]$ that the program will fail given execution of the location l [138]. Let n be the total number

[1] tiobe.com/tiobe-index/, blog.golang.org/survey2016-results

of executions, f be the number of executions that failed, p be the number of executions that did not fail, n_l be the total number of times l was executed, f_l be the total number of times l was executed and the program failed, and p_l be the total number of times l was executed and the program did not fail. Then we can define an estimator for $\Pr[F|l]$ in terms of these counts by:

$$\Pr[F|l] = \frac{\Pr[F \cap l]}{\Pr[l]} \approx \frac{\frac{f_l}{n}}{\frac{n_l}{n}} = \frac{f_l}{n_l} \tag{6.1}$$

Many more measures have been developed to assess the suspiciousness of program locations but many can be expressed using combinations of simple estimators of just four probabilities: the probability of the program failing $\Pr[F] \approx \frac{f}{n}$, the probability of the program not failing (test passing) $\Pr[P] \approx \frac{p}{n}$, the probability of the execution of a location l when the program fails $\Pr[F \cap l] \approx \frac{f_l}{n}$, and the probability of the execution of a location l when the program does not fail $\Pr[P \cap l] \approx \frac{p_l}{n}$. For example, using these estimators the popular *Ochiai* metric [1] can be expressed [12] as:

$$\begin{aligned} Och &\approx \sqrt{\Pr[F|l] \times \Pr[l|F]} \\ &= \sqrt{\frac{\Pr[F \cap l]}{\Pr[F \cap l] + \Pr[P \cap l]} \times \frac{\Pr[F \cap l]}{\Pr[F]}} \end{aligned}$$

Each simple program statement or instruction is contained within a basic block (see Figure 6.2). Since the execution of one operation of a basic block implies the execution of the whole block (under most circumstances) all operations in a block are equally suspicious under any coverage-based statistical suspiciousness measure. Thus, it suffices to compute the suspiciousness for a block as a whole rather than doing so separately for each statement or instruction in the block.

One method of measuring program behavior is through flow graph profiling (e.g., as performed by Dynagrok). In this paper a "suspicious behavior" is a subgraph h of a dynamic control flow graph (DCFG) g such that execution of h is statistically associated with program failure. The framework outlined above will be extended from particular basic blocks to subgraphs of flow graphs. This allows nearly any CBSFL suspiciousness measure to be re-used as a suspiciousness measure for flow graph fragments.

As before, the probability of program failure is $\Pr[F] \approx \frac{f}{n}$ and the probability of a program not failing is $\Pr[P] \approx \frac{p}{n}$. However, our other two "building block" estimators will need to be modified for use with subgraphs. Let $\mathcal{F}$ be the set of DCFGs collected from failing executions and let $\mathcal{P}$ be the set from non-failing (passing) executions. Let g be a dynamic control flow graph, and let h be a subgraph of g, denoted $h \sqsubseteq g$. We say that the subgraph h is "covered" by any program execution with DCFG g.

In the previous section the Ochiai metric was defined in Equation 6.2 in terms of the probabilities $\Pr[F \cap l]$ (the probability that the program fails and the location l is executed) and $\Pr[P \cap l]$ (the probability that the program does not fail and the location l is executed). Ochiai can be adapted for use with subgraphs by replacing these probabilities with analogous ones: $\Pr[F \cap h]$ (the probability that the program fails and subgraph h is covered) and $\Pr[P \cap h]$ (the probability that the program does not fail and subgraph h is covered).

Estimators for these probabilities can be defined as:

$$\Pr[F \cap h] \approx \frac{|\{g : g \in \mathcal{F} \wedge h \sqsubseteq g\}|}{n} \tag{6.2}$$

$$\Pr[P \cap h] \approx \frac{|\{g : g \in \mathcal{P} \wedge h \sqsubseteq g\}|}{n} \tag{6.3}$$

Ochiai can then be redefined for suspicious subgraphs (behaviors) as:

$$Och_{\mathcal{F},\mathcal{P}}(h) \approx \sqrt{\frac{\Pr[F \cap h]}{\Pr[F \cap h] + \Pr[P \cap h]} \times \frac{\Pr[F \cap h]}{\Pr[F]}} \tag{6.4}$$

The other suspiciousness measures discussed by Sun and Podgurski [138] can be adapted in a similar fashion.

Many suspicious behaviors never appear in full among the dynamic flow graphs of the non-failing (passing) executions. However, portions of these behaviors do appear. The estimator above for $\Pr[P \cap h]$ will always estimate the probability of such subgraphs as 0. This seems to be an underestimate for subgraphs for which a majority of their vertices and edges are covered by non-failing executions. An alternative (and efficiently computable) estimator $\widetilde{\Pr}[P \cap h]$ averages the probability estimates for each edge and vertex:

$$\begin{aligned}
\widehat{\Pr}[P \cap \epsilon] &= \frac{|\{g : g \in \mathcal{P} \wedge \epsilon \in E_g\}|}{n} \\
\widehat{\Pr}[P \cap v] &= \frac{|\{g : g \in \mathcal{P} \wedge v \in V_g\}|}{n} \\
\widetilde{\Pr}[P \cap h] &= \frac{\sum_{\epsilon \in E_h} \widehat{\Pr}[P \cap \epsilon] + \sum_{v \in V_h} \widehat{\Pr}[P \cap v]}{|E_h| + |V_h|}
\end{aligned} \tag{6.5}$$

This new estimator gives "partial credit" to a graph h which has substructures which are covered by passing executions.

Table 6.1 summarizes the definitions of the probability estimators used throughout the rest of this chapter. Table 6.2 provides the formulas for a representative set of suspiciousness metrics adapted for use with DCFG subgraphs. Note, if a subgraph h contains a single vertex v and no edges, the estimators are equivalent to the estimators described in the previous section.

6.4 Mining Suspicious Behaviors

This section reviews previous work on discriminative graph mining [86,127,140,150] and how it can be applied to statistical fault localization [24,111]. The next section will introduce our new algorithm Score-Weighted Random Walks using the background reviewed in this section. The basics of extracting the most suspicious subgraphs will be introduced here by reviewing three related discriminative subgraph mining algorithms: Branch-And-Bound [86, 127, 140], sLeap [150], and LEAP Search [150]. As introduced in Section 6.3, the subgraphs will be extracted by analyzing two sets of graphs. The first set $\mathcal{F}$ contains the dynamic control flow graphs collected from executions that failed. In our empirical study the DCFGs were collected with our new Dynagrok platform (see Section 6.2). The second set $\mathcal{P}$ contains the DCFGs from passing (or non-failing) executions of the program. The three aforementioned

Table 6.1: Probability estimators adapted to subgraphs of basic block flow graphs.

Probability Estimator	Formula
$\widehat{\Pr}[F]$	$\frac{f}{n}$
$\widehat{\Pr}[P]$	$\frac{p}{n}$
$\widehat{\Pr}[F \cap h]$	$\frac{\lvert\{g : g \in \mathcal{F} \wedge h \sqsubseteq g\}\rvert}{n}$
$\widetilde{\Pr}[P \cap h]$	$\frac{\sum_{\epsilon \in E_h} \widehat{\Pr}[P \cap \epsilon] + \sum_{v \in V_h} \widehat{\Pr}[P \cap v]}{\lvert E_h\rvert + \lvert V_h\rvert}$
$\widehat{\Pr}[h]$	$\widehat{\Pr}[F \cap h] + \widetilde{\Pr}[P \cap h]$

algorithms will be described with reference to the problem of finding the most suspicious subgraphs of the graphs in $\mathcal{F}$, in contrast to the subgraphs of the graphs in $\mathcal{P}$, although the algorithms are applicable to discriminative subgraph mining problems in general.

Finding suspicious subgraphs from $\mathcal{F}$, the set of dynamic control flow graphs of failed program executions, is an application of *discriminative (or significant) subgraph mining* [150]. Significant subgraph mining is a variation of *frequent subgraph mining* (FSM) [71,151]. FSM finds all subgraphs of a graph (or graphs) that recur k or more times for a chosen k. In significant subgraph mining, instead of finding all subgraphs that are frequent, the goal is to find the most significant subgraph(s) [25] according to some measure of significance. If there are multiple classes of graphs in the database (e.g. "positive" and "negative" graphs), significance measures such as Information Gain [24, 150] are used to guide the algorithm to find subgraphs that *discriminate* between the positive and negative graphs. In fault localization, the positive graphs are the DCFGs collected from failing executions and the negative graphs are the DCFGs collected from passing executions.

Previous studies in Fault Localization [24,111] used algorithms in the Branch-And-Bound family [86] (such as LEAP Search [150]) to mine the top-k suspicious subgraphs.

Definition 6.1 (Top-k Suspicious Subgraph Mining)**.** *Given $\mathcal{F}$, the set of dynamic control flow graphs collected from failing program executions, $\mathcal{P}$, the set of DCFGs from passing executions, and a suspiciousness measure ς, find a set of subgraphs H such that $|H| = k$ and $\sum_{h \in H} \varsigma(h)$ is maximized.*

In principle the suspiciousness measure ς in the definition above could use arbitrary information from a subgraph h and the sets $\mathcal{F}$ and $\mathcal{P}$. However, in symmetry to the discussion in Section 6.3 and with previous work [150] we will only consider measures defined in terms of the probability estimators in Table 6.1.

Listing 6.1 shows a modified version of the Branch-And-Bound algorithm [86, 127, 140, 150]. It was modified to find the Top-k discriminative subgraphs (instead of the single most discriminative subgraph). Branch-And-Bound enumerates the subgraphs of $\mathcal{F}$ in a depth-first manner. At each step of the algorithm Branch-And-Bound considers a subgraph h of a graph $g \in \mathcal{F}$. If h's score $\varsigma(h)$ is greater than that of the best graph $\bar{h}$ found so far, then h becomes the new $\bar{h}$. Branch-And-Bound then checks to see if any supergraph $h' \sqsupseteq h$ (where

Table 6.2: A representative set of suspiciousness metrics for statistical fault localization [99] defined in terms of probability estimators [138].

Suspiciousness Metric	Formula
Precision	$\frac{\Pr[F \cap h]}{\Pr[h]}$
F1	$2\frac{\Pr[h]}{\Pr[F]+\Pr[h]}\frac{\Pr[F \cap h]}{\Pr[h]}$
Ochiai	$\sqrt{\frac{\Pr[F \cap h]}{\Pr[F]}\frac{\Pr[F \cap h]}{\Pr[h]}}$
Jaccard	$\frac{\Pr[F \cap h]}{\Pr[F]+\Pr[P \cap h]}$
Information Gain	$\Big[\frac{\Pr[F \cap h]}{\Pr[h]}\log_2\Big(\frac{\Pr[F \cap h]}{\Pr[h]}\Big) + \frac{\Pr[P \cap h]}{\Pr[h]}\log_2\Big(\frac{\Pr[P \cap h]}{\Pr[h]}\Big)\Big] - [\Pr[F]\log_2(\Pr[F]) + \Pr[P]\log_2(\Pr[P])]$
Associational Risk	$\frac{\Pr[F \cap h]-\Pr[F]\Pr[h]}{\epsilon+\Pr[h]-(\Pr[h])^2}$
Contrast	$\Pr[F \cap h]-\Pr[P \cap h]$
Relative-Precision	$\frac{\Pr[F \cap h]}{\Pr[h]}-\Pr[F]$
Relative-F1	$2\frac{\Pr[h]}{\Pr[F]+\Pr[h]}\Big(\frac{\Pr[F \cap h]}{\Pr[h]}-\Pr[F]\Big)$
Relative-Ochiai	$\sqrt{\frac{\Pr[h]}{\Pr[F]}}\Big(\frac{\Pr[F \cap h]}{\Pr[h]}-\Pr[F]\Big)$
Relative-Jaccard	$\frac{\Pr[F \cap h]}{\Pr[F]+\Pr[P \cap h]}-\Pr[F]$

$h' \sqsubseteq g$ for a graph $g \in \mathcal{F}$) could have a score at least as large as $\varsigma(\bar{h})$, by computing an upper bound $\hat{\varsigma}$ on ς. If none of the supergraphs of h have a score at least as large as the current maximum $\bar{h}$ then all of the supergraphs of h are pruned. Finally, all graphs which can be constructed from h by adding one edge are added to the queue of graphs to consider.

sLeap [150] improves on Branch-And-Bound by integrating a heuristic pruning condition into the Branch-And-Bound algorithm. The amount of heuristic pruning is controlled by a parameter γ (called σ in the original paper). The condition prunes the current subgraph h if it has a supergraph h' that has already been processed and the difference in support for h and h' is with-in γ.

$$\frac{2 * (\Pr[F \cap h] - \Pr[F \cap h'])}{\Pr[F \cap h] + \Pr[F \cap h']} < \gamma \tag{6.6}$$

$$\frac{2 * (\Pr[P \cap h] - \Pr[P \cap h'])}{\Pr[P \cap h] + \Pr[P \cap h']} < \gamma \tag{6.7}$$

In Yan's paper the heuristic was expressed in set notation. However, it is easier to understand the condition when expressed in terms of probabilities. The pruning condition is heuristic, meaning sLeap only approximates the behavior of Branch-And-Bound. The sLeap algorithm is shown in Listing 6.2.

LEAP Search uses sLeap as a subroutine of *Frequency Descending Mining*. As this name suggests, it works by running the sLeap algorithm repeatedly, each time with a lower setting for the minimum frequency (the `min_F_sup` parameter in Listing 6.1). At each step, the minimum frequency is halved until either it reaches 1 or the output of sLeap does not change between runs. Note that each iteration of LEAP feeds the output of sLeap back into itself to seed the set of maximally scored subgraphs. This pre-seeding of sLeap allows later iterations to prune the search space much faster. An (optional) last step of LEAP then runs sLeap one last time with the heuristic pruning parameter $\gamma < 0$ so that no heuristic pruning is performed.

The key to the algorithms in the Branch-And-Bound family is the determination of an *upper bound* on the suspiciousness (or significance) score for all potential supergraphs of a graph h. One bound (considered by Yan *et al.* [150]) could be determined by considering two cases:

1. The case when the set $\mathcal{F}$ does not contain any of the super graphs of h but the set $\mathcal{P}$ does contain them. This case can be expressed as: $\varsigma_c(0, y)$ where $y = \widehat{\Pr}[P \cap h]$.
2. The case when the set $\mathcal{P}$ does not contain any of the super graphs of h but the set $\mathcal{F}$ does contain them. This case can be expressed as: $\varsigma_c(x, 0)$ where $x = \widehat{\Pr}[F \cap h]$.

Using these two cases an upper bound for ς is

$$\hat{\varsigma}_c(x, y) = \max\{\varsigma_c(x, 0), \varsigma_c(0, y)\} \tag{6.8}$$

Since the measure ς may not be well defined at 0 typically a small (but non-zero) probability will be used in place of 0.

$$\hat{\varsigma}_c(x, y) = \max\{\varsigma_c(x, \epsilon), \varsigma_c(\epsilon, y)\} \tag{6.9}$$

However, the definition Equation 6.9 is too conservative on its estimate of the minimum probability of h appearing on the passing or failing side. An upper bound (which improves

```
# param F: The DCFGs collected from failing executions
# param P: The DCFGs collected from passing executions
# param score: The significance (suspiciousness) score
# param k: The number of graphs to mine
# param min_F_sup: The minimum number of graphs in the set
#                  F that a suspicious graph must appear in
# returns the k most suspicious maximal subgraphs in F
def branch_and_bound(F, P, score, k, min_F_sup=1):
    queue = max_priority_queue()
    # Initialize the queue with subgraphs containing
    # single vertices (one for each unique vertex label).
    for vertex in F.unique_vertices():
        queue.push(score(vertex), vertex)
    # best tracks the top-k most significant subgraphs
    best = min_priority_queue()
    # visited tracks which subgraphs has been processed
    visited = set()
    while queue:
        # Get the highest scoring element in the work list
        h = queue.pop_max()
        # Check to see if it has already been processed
        if h in visited:
            continue
        visited.add(h)
        # See if it is one of the best subgraphs found
        check_best(score, best, h)
        # Add supergraphs of h to the queue
        for supergraph in extend_with_one_edge(F, P, h):
            # Check the minimum support
            if F.count_support(supergraph) < min_F_sup:
                continue
            # Check upper bound pruning criteria
            if best.size() < k or \
               (upper_bound(score, supergraph) >= score(best.peek_min())):
                queue.push(score(h), h)
    return best

# Checks to see if the subgraph h should be added to the
# queue of the highest scoring subgraphs
# param score: The significance (suspicousness) measure
# param best: a min priority queue of subgraphs
# param h: a subgraph
def check_best(score, best, h):
    if best.size() < k:
        best.push(score(h), h)
    elif score(h) > score(queue.peek_min()):
        best.pop_min(); best.push(score(h), h)
    elif score(h) == score(queue.peek_min()) and \
            random.random() > .5:
        # flip a coin to decide which graph to keep
        best.pop_min(); best.push(score(h), h)
```

Listing 6.1: Python psuedocode for the Branch-And-Bound algorithm modified to search for the top-k most suspicious maximal subgraphs. The function `extend_with_one_edge` takes the subgraph `h` and produces every one-edge supergraph of `h` that exists in $\mathcal{F}$ and $\mathcal{P}$. Finally, the function `upper_bound` computes the upper bound on the function `score` for all supergraphs of `h`.

```
1  # param F: The flow graphs collected from failing executions
2  # param P: The flow graphs collected from passing executions
3  # param score: The suspicousness measure
4  # param k: The number of graphs to mine
5  # param gamma: sLeap's the heuristic pruning parameter γ
6  # param min_F_sup: The minimum number of graphs in the set F that a suspicious
7  #                  graph must appear in
8  # param best: a starting set of known suspicious subgraphs
9  # returns the most suspicious maximal subgraphs in F
10 def sLeap(F, P, score, k, gamma, min_F_sup=1, best=None):
11     queue = max_priority_queue()
12     for vertex in F.unique_vertices():
13         queue.push(score(vertex), vertex)
14     if best is None:
15         best = min_priority_queue()
16     visited = set()
17     while queue:
18         ## get the highest scoring element in the work list
19         h = queue.pop_max()
20         ## check to see if it has already been processed
21         if h.canonical_label() in visited:
22             continue
23         ## add it to the set of visited items
24         visited.add(h.canonical_label())
25         supergraphs = extend_with_one_edge(F, P, h)
26         skip = False
27         for sg in supergraphs:
28             if sg.canonical_label() not in seen:
29                 continue
30             Df = (2 * (Pr(F, cur) - Pr(F, sg))) / (Pr(F, cur) + Pr(F, sg))
31             Dp = (2 * (Pr(P, cur) - Pr(P, sg))) / (Pr(P, cur) + Pr(P, sg))
32             if Df < gamma and Dp < gamma:
33                 skip = True
34                 break
35         if skip:
36             continue
37         # See if it is one of the best subgraphs found
38         check_best(score, best, h)
39         for supergraph in supergraphs:
40             ## check the to see if it falls below the minimum support
41             if F.count_support(extension) < min_F_sup:
42                 continue
43             ## check to see if it fails the upper bound pruning criteria
44             if best.size() < k or
45                upper_bound(score, supergraph) >= score(best.peek_min()):
46                 queue.push(score(h), h)
47     return best
```

Listing 6.2: Python psuedo code for the sLeap algorithm modified to search for the top-k most suspicious maximal subgraphs. The `check_best` function is defined in Listing 6.1. This algorithm is nearly the same as the Branch-And-Bound algorithm the only difference is the addition of a new heuristic pruning strategy (lines $25-33$). The heuristic is not sound and may prune one or more of the top-k subgraphs but in practice usually does not.

upon the upper bound in [150]) for ς can be constructed using the following two facts. First, a user may ensure a subgraph h appears among the graphs in $\mathcal{F}$ at least $f_{\min}$ times. $\frac{1}{f_{\min}}$ is a lower bound on $\Pr[F \cap h]$. Second, the user may specify a maximum number of edges $|E|_{\max}$ allowed in a subgraph. $|E|_{\max}$ can be used to derive a lower bound on $\widetilde{\Pr}[P \cap h]$:

$$\widetilde{\Pr}[P \cap h]_{\min} = \frac{\sum_{\epsilon \in E_H} \widehat{\Pr}[P \cap \epsilon] + \sum_{v \in V_H} \widehat{\Pr}[P \cap v]}{2|E|_{\max} + 1} \tag{6.10}$$

Combining the two lower bounds yields an upper bound $\hat{\varsigma}$ on ς for the supergraphs of h:

$$\hat{\varsigma} = \max \left\{ \begin{array}{l} \varsigma(\Pr[F \cap h], \widetilde{\Pr}[P \cap h]_{\min}), \\ \varsigma(\frac{1}{f_{\min}}, \widetilde{\Pr}[P \cap h]) \end{array} \right\} \tag{6.11}$$

Branch-And-Bound algorithms depend on the suspiciousness (or significance) metrics satisfying a technical property (Equation 5 in the LEAP Search paper [150]) which we will call the Discriminative Velocity Property (DVP). Table 6.3 shows that, of the metrics in Table 6.2, only Information Gain satisfies this technical detail. Because satisifaction of DVP is a requirement for using Branch-And-Bound algorithms, of the metrics shown in Table 6.2, only Information Gain can be used. This is a major limitation of algorithms in this family.

Definition 6.2 (Discriminative Velocity Property (DVP))**.** *Let ς be a suspiciousness score, $x = Pr[F \cap h]$, and $y = Pr[P \cap h]$. DVP is satisfied if:*

$$(x \in (0,1] \cap y \in [0,1] \cap x > y) \implies \left(\frac{\partial \varsigma}{\partial x} > 0 \cap \frac{\partial \varsigma}{\partial y} < 0 \right)$$
$$(x \in (0,1] \cap y \in [0,1] \cap x < y) \implies \left(\frac{\partial \varsigma}{\partial x} > 0 \cap \frac{\partial \varsigma}{\partial y} < 0 \right)$$

DVP formalizes the intuition behind the upper bounds in Equations 6.8, 6.9, and 6.11 and ensures they are indeed upper bounds on ς [150]. DVP states that ς must be increasing if when $x > y$ either x increases or y decreases. In the same way, ς must also be increasing if when $x < y$ either x decreases or y increases. If ς does not satisfy DVP then equations do not define upper bounds as ς may not increase (or decrease) in the expected manner. If ς movements are not smoothly related to changes in the frequency of the subgraphs then the Branch-And-Bound algorithm will spuriously prune subgraphs which could have had scores greater than the current maximal score.

6.5 Sampling Suspicious Behaviors

The Branch-And-Bound framework from Section 6.4 has four drawbacks when applied to automatic fault localization. The first is the requirement that the suspiciousness metrics satisfy the Discriminative Velocity Property (DVP), restricting these algorithms to metrics such as Information Gain (see Table 6.3). The second drawback is the requirement (which stems from Equation 6.11) for the user to specify the maximum number of edges allowed in a discriminative subgraph. The third drawback is the requirement for the user to specify the maximum number of subgraphs to mine. The fourth drawback to the Branch-And-Bound framework is its enumeration of the subgraphs of graphs in $\mathcal{F}$ — a scalability bottleneck [25].

Table 6.3: Discriminative Velocity Property (DVP) (Eq. 5 in [150]) and Inverse Velocity Property (IVP) (Def. 6.3) satisifaction for each suspciousness metric in Table 6.2.

Suspiciousness Metric	Satisfies DVP	Satisfies IVP
Precision		✓
F1		✓
Ochiai		✓
Jaccard		✓
Information Gain	✓	✓
Associational Risk		
Contrast		✓
Relative-Precision		✓
Relative-F1		✓
Relative-Ochiai		
Relative-Jaccard		✓

Note: Satisfaction was checked using Mathematica for most measures but had to be checked by hand for Relative-Ochiai and Information Gain. Information Gain was previously shown to satisfy DVP [24,150] and our analysis confirmed this result. None of the other metrics satisfy DVP.

Recall, Branch-And-Bound enumerates all of the potentially suspicious subgraphs of the set dynamic flow graphs $\mathcal{F}$ collected from failing executions. The enumeration is done in a similar manner to the enumerations conducted in the mining algorithms presented in Chapter 3. As was found in Chapter 3 as graphs grow larger it becomes more difficult to quickly find frequent subgraphs and by extension suspicious subgraphs. In the same way, as the minimum required support for the subgraph decreases the mining difficulty increases.

Table 6.4 shows an example of this phenomena using the sLeap algorithm on a bug in an AVL Tree (see Table 6.11). Random tests were generated for the tree by generating a mixture of `Put`, `Get`, `Has`, and `Remove` operations. The keys used as arguments were generated randomly. Each time a new operation is generated there is a 50% chance that a previous key used in a `Put` operation is re-used. In Table 6.4 as the minimum support decreases the amount of time sLeap spends mining the top-20 graphs increases.

The LEAP Search algorithm tries to take advantage of this property by using Frequency Descending Mining. As indicated in Table 6.4, like sLeap, LEAP Search takes a long time to mine graphs at "Min Support 5." Although, LEAP Search is generally faster than Branch-And-Bound it isn't usually faster than sLeap. The advantage of LEAP Search over sLeap is LEAP Search returns the same answer as Branch-And-Bound while sLeap only approximates the output of Branch-And-Bound when $\gamma > 0$.

The problem shown in Table 6.4 can be partially ameliorated by setting stricter limits on the maximum number of edges. The performance of this "solution" is shown in Table 6.5. Note, the effect of the maximum edges parameter depends on the dataset. For some datasets generated from different bugs in the AVL tree the sLeap algorithm does not complete in 1 hour when the maximum number of edges is set to 25. While in Table 6.5, sLeap completes in 6 seconds for that setting.

Another fundamental problem with Branch-And-Bound, sLeap, and LEAP Search is they all find the top-k most suspicious subgraphs. This may be appropriate for other applications of discriminative or significant subgraph mining but it is not necessarily the right formulation for fault localization a statistical fault localization metrics do not always correctly predict the location of the fault. This can result in the location of the fault not appearing among the top-k most suspicious locations or behaviors. This may lead one to

pick a very large value for k and hope that the programmer either discovers the fault or moves onto another method before reaching the bottom of the list. However, the mining difficulty also increases as k is increased (see Table 6.6).

In order solve the aforementioned problems with the Branch-And-Bound framework, we have developed a new algorithm, called *Score Weighted Random Walks* (SWRW), for finding suspicious, significant, or discriminative subgraphs. The new method approximates Branch-And-Bound's output like sLeap does. The sampling is weighted by the suspiciousness scores, which (heuristically) minimizes the error in Branch-And-Bound's output. A sampling approach makes it easy to trade-off computation time with accuracy by adjusting the sample size.

The new method, unlike the Branch-And-Bound algorithms, does not require metrics to satisfy the restrictive Discriminative Velocity Property mentioned in the previous section (Equation 5 in [150]). Instead, metrics must satisfy a new property we call the *Inverse Velocity Property* (IVP). As shown in Table 6.3, all but two of the suspiciousness metrics shown in Table 6.2 satisfy IVP.

Definition 6.3 (Inverse Velocity Property)**.** *Let ς be a suspiciousness score, $x = Pr[F \cap h]$, and $y = Pr[P \cap h]$. IVP is satisfied if:*

$$(x \in (0,1] \wedge y \in [0,1]) \implies \left(\frac{\partial \varsigma}{\partial x} > 0 \wedge \frac{\partial \varsigma}{\partial y} < 0\right)$$

IVP requires a suspiciousness metric to increase as the support of a subgraph h either increases in the set $\mathcal{F}$ of graphs from failing executions or decreases in the set $\mathcal{P}$ of graph from passing executions. It also requires the metric to decrease when support for h falls in $\mathcal{F}$ or increases in $\mathcal{P}$. This follows the anti-monotonic structure of subgraph mining [25]:

$$h \sqsubseteq h' \implies \varsigma(h) \leq \varsigma(h')$$

Suspiciousness metrics which satisfy IVP induce a (newly defined) *suspicious subgraph lattice* on the subgraphs of the graphs in $\mathcal{F}$. Figure 6.3 shows an example suspicious subgraph lattice for a small undirected graph dataset. SWRW samples suspicious subgraphs from the lattice via a weighted forward random walk. The suspicious subgraph lattice is a graph where the nodes represent subgraphs of the dataset. The suspicious subgraph lattice is a subgraph of the *connected subgraph lattice* [59].

Definition 6.4 (Connected Subgraph Lattice of $\mathcal{G}$)**.** *The subgraph relation $\cdot \sqsubseteq \cdot$ induces the* Connected Subgraph Lattice $\mathcal{L}_\mathcal{G}$ *representing all the possible ways of constructing a graph $G \in \mathcal{G}$ from the empty subgraph by adding one edge at a time. $\mathcal{L}_\mathcal{G}$ is a digraph where each vertex u represents a unique connected (ignoring edge direction) subgraph of some $G \in \mathcal{G}$. There is an edge from u to v in $\mathcal{L}_\mathcal{G}$ if adding some edge ϵ to u creates a subgraph $u + \epsilon$ isomorphic to v, $v \cong u + \epsilon$.*

Definition 6.5 (Suspicious Subgraph Lattice)**.** *The subgraph relation $\cdot \sqsubseteq \cdot$ and a suspiciousness measure ς satisfying IVP induce a* Suspicious Subgraph Lattice ς-$\mathcal{L}_G$*. The lattice ς-$\mathcal{L}_G$ is a connected subgraph of the connected subgraph lattice $\mathcal{L}_\mathcal{G}$ rooted at the root of $\mathcal{L}_\mathcal{G}$. Let the empty subgraph $h_\varnothing$ be in $V_{\varsigma\text{-}\mathcal{L}_G}$ as the root node of ς-$\mathcal{L}_G$ (it is also the root of $\mathcal{L}_\mathcal{G}$). If an edge $(u, v) \in E_{\varsigma\text{-}\mathcal{L}_G}$ then v is in $V_{\varsigma\text{-}\mathcal{L}_G}$. An edge $(u, v) \in E_{\mathcal{L}_\mathcal{G}}$ is an edge in $E_{\varsigma\text{-}\mathcal{L}_G}$ if and only if:*

$$u \in V_{\varsigma\text{-}\mathcal{L}_G} \wedge \left[\varsigma(u) < \varsigma(v) \vee \varsigma(u) = \varsigma(v) \wedge \frac{Pr[F \cap v]}{Pr[v]} = 1\right]$$

Min Support	Mining Time (seconds)
10	1.22
9	1.95
8	2.31
7	3.21
6	3.42
5	> 120

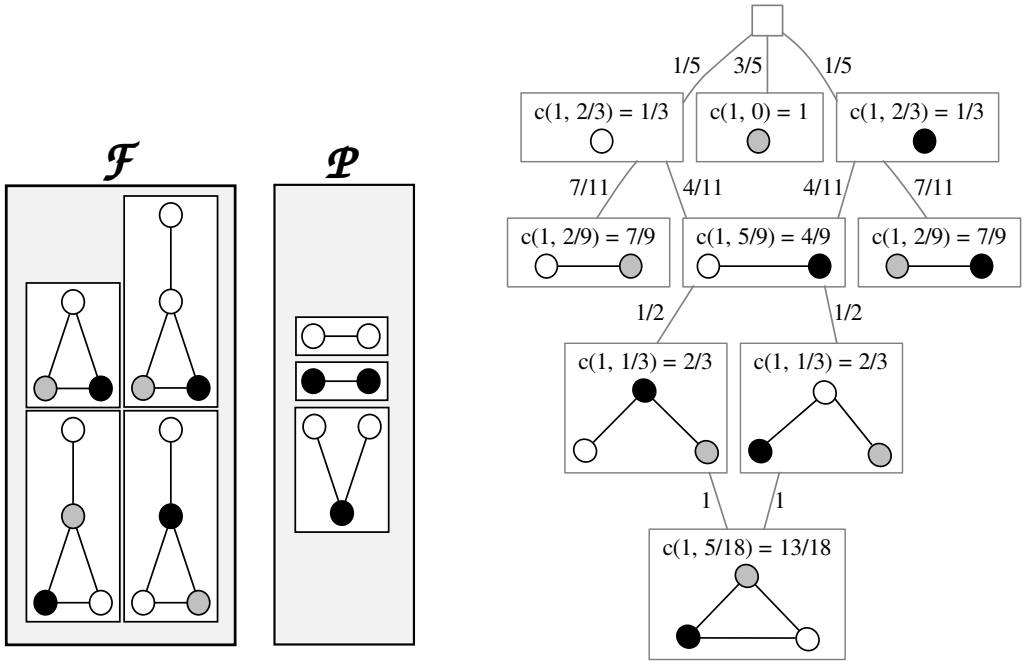
Table 6.4: Mining speed of sLeap as function of minimum support. sLeap was configured to find the top 20 most suspicious subgraphs of the flow graphs from the executions of 2nd buggy version of the AVL tree used in the empirical evaluation (Section 6.6). There were 10 flow graphs in the set $\mathcal{F}$ and two flow graphs in the set $\mathcal{P}$. The maximum edges parameter was set to 100. The "leap" heuristic parameter γ was set to 1 (maximum leaping). sLeap requires the suspicious metric to satisfy DVP and therefore the Information Gain metric was used (see Table 6.3).

Min Support	Max Edges	Mining Time (seconds)
5	5	0.069
5	10	0.22
5	15	2.85
5	20	5.43
5	25	6.13
5	50	7.55
5	100	> 120

Table 6.5: Mining speed of sLeap as function of the maximum number allowed edges for AVL Bug 2. Other parameters were set as in Table 6.4.

Min Support	Max Edges	k	Mining Time (seconds)
8	25	1	0.024
8	25	2	0.26
8	25	3	1.06
8	25	5	7.04
8	25	10	19.2
8	25	25	27.5

Table 6.6: Mining speed of sLeap as function of k, the number of graphs to mine, for AVL Bug 1.The γ parameter was set at .1. Other parameters were set as in Table 6.4.



(a) Dataset of graphs

(b) Suspicious subgraph lattice

Figure 6.3: Example suspicious subgraph lattice constructed using the Contrast suspiciousness metric (see Table 6.2). The colors stand in for labels in this simple example. In each lattice node (the boxes) the suspiciousness scores are shown ("c" stands for Contrast). On the lattice edges (edges between the boxes) the *forward* transition probabilities are shown.

```
# param F: The flow graphs collected from failing executions
# param P: The flow graphs collected from passing executions
# param score: The suspicousness measure
# param min_F_sup: The minimum number of graphs in the set F
#                  that a suspicious graph must appear in
# param start: the graph in the suspicious subgraph lattice
#              to start the walk from.
# returns a subgraph of F randomly sampled from the
# suspicious subgraph lattice induced by P and score.
def swrw(F, P, score, min_F_sup, start):
    cur = start
    prev = cur
    while cur is not None:
        # At each step in the walk, a random restart occurs
        # with probability proportional to the maximum length
        # of the walk.
        if random.random() < 1.0/MAX_EDGES:
            cur = start; prev = cur
            continue
        # Compute the direct supergraphs of the current
        # graph
        supers = extend_with_one_edge(F, P, cur)
        # Filter out graphs not in the suspicious subgraph
        # lattice
        supers = filter_supergraphs(F, P, min_F_sup, score,
                cur, supers)
        # Randomly select a supergraph to be the next graph
        # in the walk, favoring those with higher
        # suspiciousness.
        prev = cur
        cur = weighted_sample(score, supers)
    return prev

# Filters out graphs in supers that are not in the
# suspicious subgraph lattice
# param h: the current subgraph (which is in the SSL)
# param supers: supergraphs of h
# returns a subset of supers
def filter_supergraphs(F, P, min_F_sup, score, h, supers):
    allowed = list()
    for sg in supers:
        precision = Pr(F, sg)/(Pr(F, sg) + Pr(P, sg))
        if score(sg) == score(h) and precision == 1:
            allowed.append(sg)
        elif score(sg) > score(h):
            allowed.append(sg)
    return allowed

# Using the score to weight the graphs sample one graph
# from graphs
# param score: the weighting score
# param graphs: a list of graphs
# returns one graph from graphs or None if graphs was empty
def weighted_sample(score, graphs):
    if len(graphs) <= 0:
        return None
    if len(graphs) == 1:
        return graphs[0]
    min_weight = min(score(g) for g in graphs)
    shift = max(0, -min_weight)
    weights = [ score(g) + shift + 1e-8 ## ensure > 0
                for g in graphs ]
    total = sum(weights)
    i = 0
    r = total * random.random()
    while i < len(weights) - 1 and r > weights[i]:
        r -= weights[i]
        i += 1
    return i
```

Listing 6.3: Python psuedocode for the new SWRW algorithm.

```
# param F: The flow graphs collected from failing executions
# param P: The flow graphs collected from passing executions
# param score: The suspicousness measure
# param min_F_sup: The minimum number of graphs in the set F
#                  that a suspicious graph must appear in
# param k: the number of walks to take
# returns a set of at most k subgraphs
def k_walks(F, P, score, min_F_sup, k):
    return {
        swrw(F, P, score, min_F_sup, F.empty_subgraph())
        for _ in xrange(k)
    }

# param F: The flow graphs collected from failing executions
# param P: The flow graphs collected from passing executions
# param score: The suspicousness measure
# param min_F_sup: The minimum number of graphs in the set F
#                  that a suspicious graph must appear in
# param p: the percentage of the top scoring vertices to
#          start walks from
# param w: the number of walks to take from each vertex
# returns a set of suspicious subgraphs
def walk_top_vertices(F, P, score, min_F_sup, p, w):
    # Sort the unique vertices in F by their scores.
    vertices = F.unique_vertices()
    vertices.sort(key=lambda v: score(v))
    # Take walks starting from a percentage p of the
    # vertices, starting with the most suspicious vertices.
    subgraphs = set()
    for i, v in enumerate(vertices):
        if i >= p * len(vertices):
            break
        # Take w walks from each vertex
        for _ in xrange(w):
            subgraphs.add(swrw(F, P, score, min_F_sup, v))
    return subgraphs
```

Listing 6.4: Python psuedocode for collecting multiple subgraphs using SWRW.

Listings 6.3 and 6.4 show the new algorithm, Score Weighted Random Walk, for finding suspicious behaviors by sampling maximal suspicious subgraphs from the suspicious subgraph lattice ς-$\mathcal{L}_G$. The algorithm is built around a core function (`swrw` in Listing 6.3). The algorithm takes a random walk on an absorbing Markov chain [54, 80] built from the suspicious subgraph lattice ς-$\mathcal{L}_G$. A Markov chain is *absorbing* if an absorbing state is reachable from every state, where a state is absorbing if it cannot be left once it is entered. The random walk taken by SWRW is weighted by the suspiciousness scores ς of the subgraphs in the lattice, which causes it to visit the more suspicious subgraphs more frequently. When the walk reaches an absorbing state the graph represented by the state is sampled.

The function `swrw` in Listing 6.3 simulates a Markov process starting at some state in the chain (`start`) and transitioning until it is absorbed at an absorbing state. The absorbing state, which is a maximally sized suspicious subgraph, is then returned as the sample. At each step in the simulation there are four sub-steps. The first is to check for a random restart of the walk. The second sub-step computes the supergraphs of the graph `cur` that represents the current state of the Markov chain. The next sub-step filters the supergraphs such that only those in $V_{\varsigma\text{-}\mathcal{L}_G}$ are kept. Finally, the function `weighted_sample` is used to select the next state for the Markov process to transition to.

To collect multiple samples from the suspicious subgraph lattice, k random walks could be taken. This approach is shown in function `k_walks` in Listing 6.4. Each walk starts from the bottom of the lattice and, using the supplied walk function (e.g. `swrw`), samples a suspicious subgraph from the lattice. This approach begs the question of how to choose an appropriate value for k.

The function `walk_top_vertices` in Listing 6.4 provides an answer. Instead of taking k walks starting from the root node, it takes walks starting from a percentage p of subgraphs representing single vertices — suspicious locations — in the graphs in $\mathcal{F}$. The function orders the subgraphs in decreasing order from from most suspicious to least suspicious. This ensures, by the Inverse Velocity Property, that SWRW starts from vertices of ς-$\mathcal{L}_G$ that are likely to lead to the most suspicious subgraphs.

6.5.1 SWRW Versus the Branch-And-Bound Framework

At the beginning of this section several scalability issues with the Branch-And-Bound framework were demonstrated in Tables 6.4 and 6.5. Returning to those examples, Tables 6.7 and 6.8 show SWRW scaling much better than sLeap as the minimum support decreases.

Two tables are presented to show two scenarios for SWRW. In the first scenario in Table 6.7 SWRW is configured to take 10 walks starting from each basic block in the program. This ensures that SWRW has a chance to find all of the suspicious behaviors and thus better approximate Branch-And-Bound. In the second table (6.8), SWRW is configured to take only 2 walks starting from the top 10% most suspicious basic blocks in the program. This is very fast and results in a good approximation of sLeap's output (as can be seen in the error term).

The Tables 6.9 and 6.10 show effect of increasing the maximum number of edges allowed in a subgraph for the same two scenarios as used in the previous two tables. SWRW scales much better than sLeap as the number of maximum edges increase. As both tables show, increase the maximum number of edges has no effect on the runtime of SWRW past 15 edges but has a big impact on sLeap. The impact on sLeap makes since because the pruning condition in the Branch-And-Bound framework depends on the upper bound ($\hat{\varsigma}$) defined in Equation 6.11. The upper bound depends on the number of maximum edges allowed in a

Min Support	sLeap Time (sec)	SWRW Time (sec)	Error
10	1.22	1.62	0.009
9	1.95	1.63	0.004
8	2.31	1.81	0.008
7	3.21	1.92	0.010
6	3.42	2.22	0.012
5	–	4.28	–
4	–	8.37	–
3	–	10.74	–
2	–	13.03	–
1	–	30.46	–

Table 6.7: Mining speed of SWRW vs. sLeap as function of minimum support for AVL bug 2. For details see on sLeap's configuration and the AVL bug see Table 6.4. SWRW was configured to use the "Top Vertices" walk method from Listing 6.4. The parameter p was set to 1 and w was set to 10. The column Error is the standard error from the scores of top-k graphs of sLeap compared with the top-k graphs from SWRW. An entry 0 indicates perfect approximation. The entries for sLeap and Error of "–" indicate sLeap did not finish in the time budget of 2 minutes.

Min Support	sLeap Time (sec)	SWRW Time (sec)	Error
10	1.22	0.038	0.071
9	1.95	0.038	0.046
8	2.31	0.058	0.040
7	3.21	0.059	0.039
6	3.42	0.096	0.036
5	–	0.370	–
4	–	1.26	–
3	–	1.68	–
2	–	2.19	–
1	–	4.72	–

Table 6.8: Mining speed of SWRW vs. sLeap as function of minimum support for AVL bug 2. SWRW was configured with $p = .1$ and $w = 2$ to demonstrate it scalability. Everything else is as it was in Table 6.7.

subgraph so as the maximum number of edges allowed increases the upper bound becomes less and less tight.

SWRW scales better than sLeap as both the maximum number of edges allowed increase and the minimum support decreases. This solves the major scalability problems inherent to the Branch-And-Bound framework. Not shown was SWRW scalability as the number of subgraphs requested increase as SWRW reports far more subgraphs than required of sLeap or Branch-And-Bound in all of the above experiments. In general the scalability of SWRW with respect to k (the number of graphs requested) is linear.

As discussed above SWRW can actually approximate the output of Branch-And-Bound better and faster than sLeap can. In the future, a faster version of the exact algorithm LEAP (which uses sLeap internally) could be bootstrapped from SWRW. Instead of using the frequency descending mining strategy, SWRW could simply be run once and the top-k subgraphs it finds would then be used to bootstrap Branch-And-Bound. SWRW could also be used inside of Branch-And-Bound to provide a more accurate prediction of the maximum score of lattice branch by sampling maximal graphs from that branch.

Max Edges	sLeap Time (sec)	SWRW Time (sec)	Error
5	0.069	1.79	0
10	0.22	3.62	0
15	2.85	4.70	0.026
20	5.43	4.67	0.016
25	6.13	4.87	0.004
50	7.55	4.34	0.007
100	–	4.40	–

Table 6.9: Mining speed of SWRW vs. sLeap as function of the maximum number allowed edges for AVL Bug 2. Minimum support was set to 5 for this table. For details on sLeap's configuration see Table 6.5. SWRW was configured as in Table 6.7.

Max Edges	sLeap Time (sec)	SWRW Time (sec)	Error
5	0.069	0.16	0
10	0.22	0.35	0
15	2.85	0.50	0.068
20	5.43	0.45	0.060
25	6.13	0.42	0.047
50	7.55	0.49	0.052
100	–	0.44	–

Table 6.10: Mining speed of SWRW vs. sLeap as function of the maximum number allowed edges for AVL Bug 2. SWRW was configured with $p = .1$ and $w = 2$ to demonstrate it scalability. Everything else is as it was in Table 6.9.

6.6 Empirical Evaluation of SWRW

We empirically evaluated SWRW's fault localization performance in terms of accuracy and cost. SWRW was compared to a previously proposed behavioral fault localization approach [24] that used LEAP Search [150]. The following questions were considered:

1. Which suspiciousness metric (of those in Table 6.2) provides the most accurate fault localization with SWRW?
2. Do the alternative suspiciousness metrics provide better fault localization performance than Information Gain? Variants of Information Gain predominate in past work on behavioral fault localization [24, 35–37, 111].
3. Comparing SWRW to Branch-And-Bound and sLeap (both of which require the Information Gain metric) which algorithm provides the best fault localization performance?

A new dataset of five real world programs with injected mutation faults was created. The programs are all written in the Go programming language. Dynagrok (see Section 6.2) was used to inject the mutations and instrumentation into them. The injected mutations were all branch condition mutations which flip the condition (ex. `if true` → `if false`). The faults created by these mutations are favorable to localization by coverage based fault localization techniques in general, including all of the techniques considered in this paper. The mutations

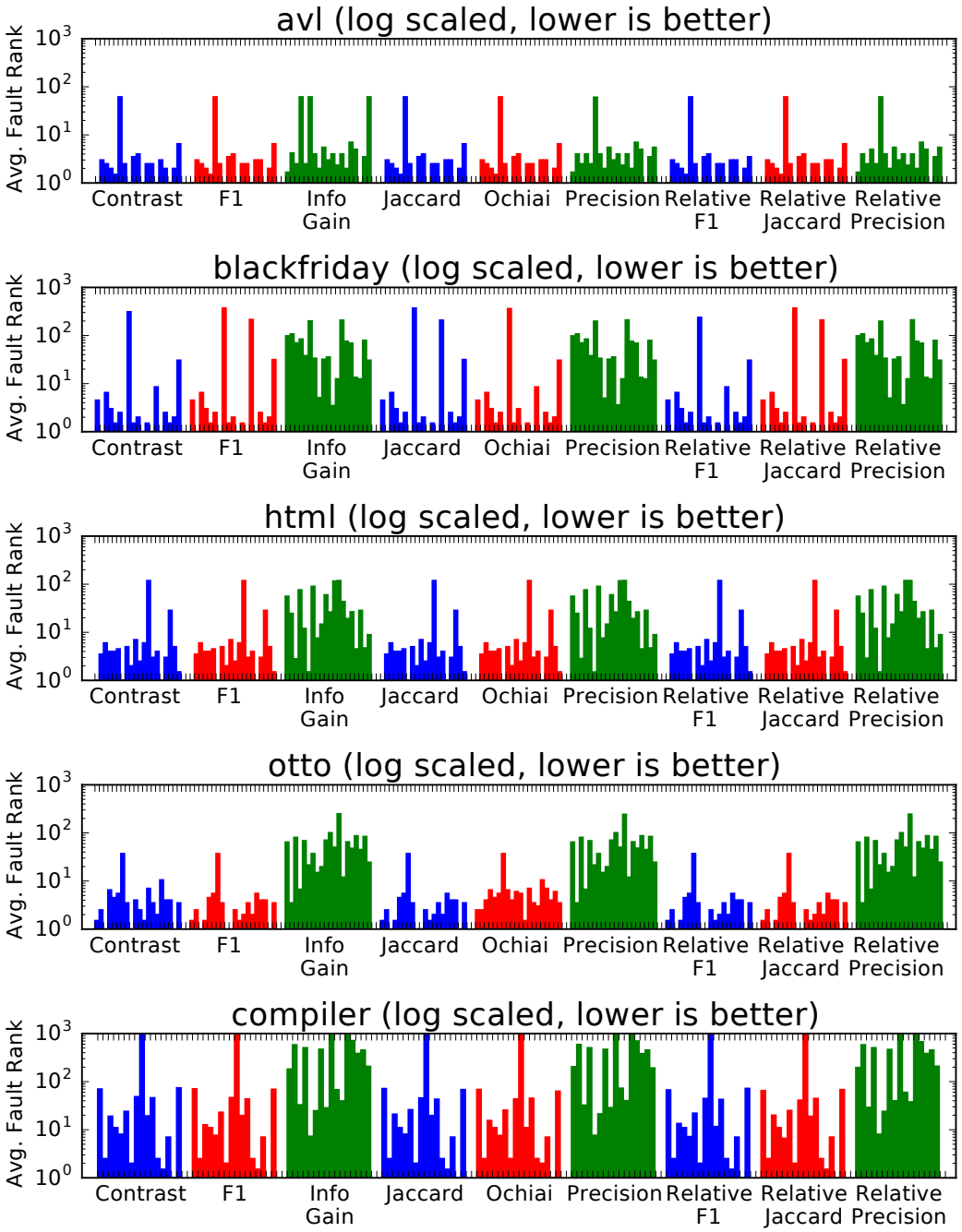


Figure 6.4: Fault localization performance of SWRW (log scaled, lower is better). Each bar gives the average fault rank for a particular buggy program version and suspiciousness metric from Table 6.2. SWRW was configured to use `walk_top_colors` from Listing 6.4 with $p = .2$ and $w = 2$. Subgraphs with at most 100 edges were collected. The experiment was repeated 20 times and the mean results are shown.

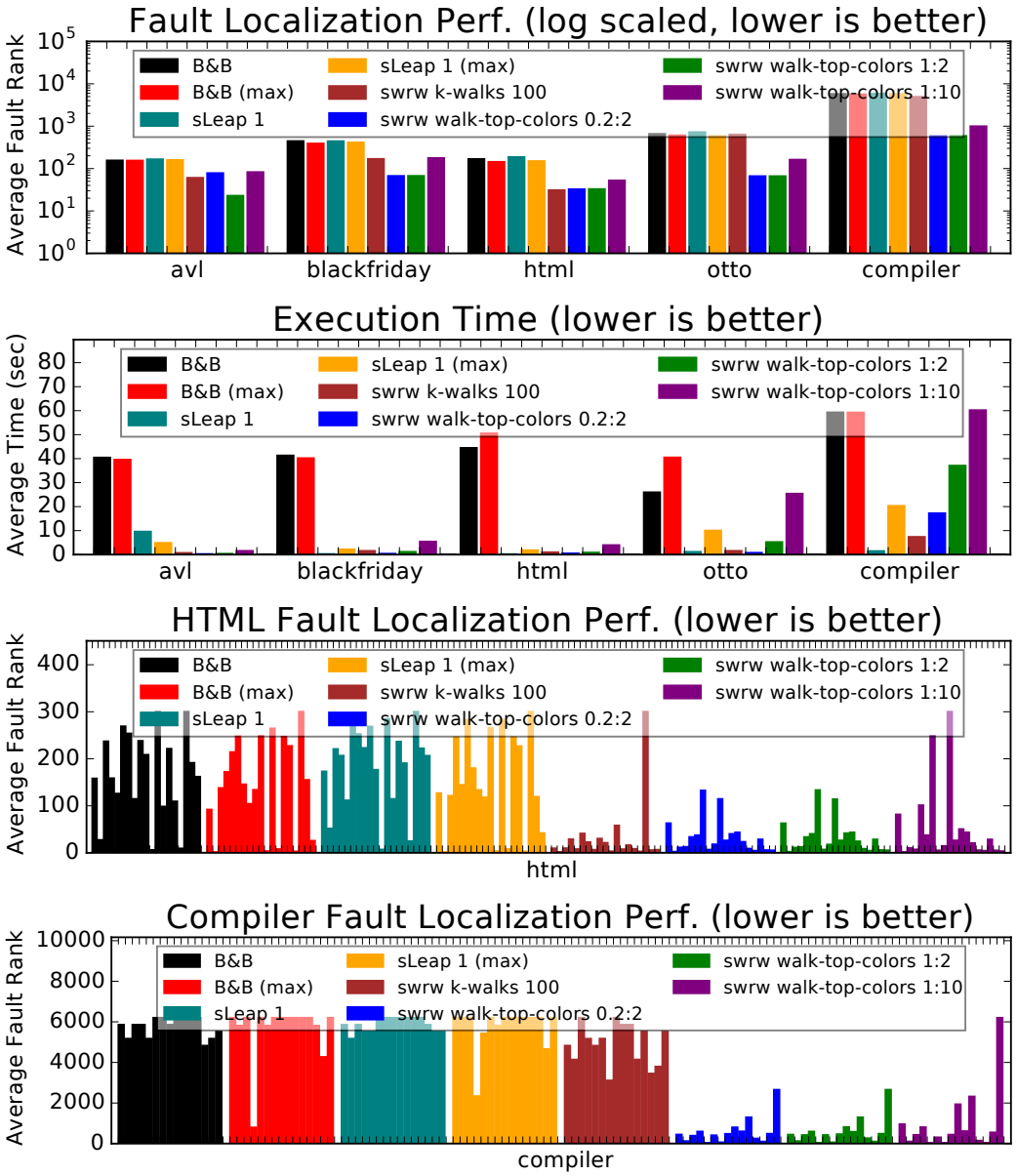


Figure 6.5: Comparision of Branch-And-Bound (B&B), sLeap, and SWRW using Information Gain. In the first and second graphs each bar represents the average performance for all program versions. The third and fourth graphs "zoom in" on the fault localization performance for the HTML Parser and the Go Compiler. Branch-And-Bound and sLeap collected 50 subgraphs each. All algorithms collected subgraphs with at most 15 edges. A 60 second timeout was set of all executions. The experiment was repeated 20 times and the mean results are shown.

created do not favor any techniques being compared over any of the other techniques. (In the future, we would like to support a wider variety of mutations in Dynagrok.)

Table 6.11 shows the programs used in the study. For each program 20 mutants were created. Mutants for which there were either no passing tests or no failing tests were removed. For the AVL tree, tests cases were automatically generated by constructing a random sequence of Put, Has, Get, and Remove operations. For the other programs, test cases were either collected from the Internet (for the HTML parser and the Markdown processor Blackfriday) or the project-supplied test cases were used (for the Javascript interpreter and the Go Compiler).

A fault rank cost measure was developed to assess the fault localization performance of Suspicious-Behavior Based Fault Localization (SBBFL). The behaviors (subgraphs) are scored using a suspiciousness metric and presented to the programmer in ranked order with the most suspicious subgraph first. The fault rank gives the expected number of behaviors a programmer would examine before examining a behavior containing the faulty location(s). This cost measure is similar to the way SBBFL techniques have been evaluated in the past [24,34,111]. The fault rank gives an objective score enabling comparison between different suspiciousness metrics for the same program version and between different programs and versions. However, it is not appropriate to directly compare standard CBSFL techniques to SBBFL techniques using fault ranks.

All of the algorithms compared in the study have an element of randomness to them. For instance, Branch-And-Bound algorithms make random choices about which subgraph to keep when some have equal scores. SWRW is a sampling algorithm and is explicitly randomized. Thus, all experiments were replicated and their results were averaged. Sometimes the algorithms return no behaviors which contain a faulty location — a localization failure. To incorporate localization failures into the overall average performance, the maximum fault rank for the relevant program (across all algorithms) is used as the fault rank when none of the returned behaviors contain the fault. Using the maximum fault rank as the cost of a localization failure means that when evaluating the performance of the Branch-And-Bound algorithms the average fault rank may be higher than the number of subgraphs mined.

Table 6.11: Datasets used in the evaluation

Program	L.O.C.	Mutants	Description
AVL (in https://github.com/timtadh/dynagrok)	483	19	An AVL tree
Blackfriday (github.com/russross/blackfriday)	8,887	19	Markdown processor
HTML (golang.org/x/net/html)	9,540	20	An HTML parser
Otto (github.com/robertkrimen/otto)	39,426	20	Javascript interpreter
`gc` (go.googlesource.com/go)	51,873	16	The Go compiler

Note: The AVL tree URL was ommitted to preserve the blinding of the review.

Table 6.12: The mean fault rank each metric in Figure 6.4.

Ochiai	40.6	Contrast	40.1	Precision	102.1
Jaccard	42.6	Relative Jaccard	42.5	Relative Precision	101.8
F1	42.6	Relative F1	**38.8**	Information Gain	131.3

6.6.1 Which Suspiciousness Measure Works the Best?

Figure 6.4 shows the fault localization performance of SWRW for all applicable suspiciousness metrics across all program versions. Table 6.12 gives the average ranks for each metric across all programs. SWRW was chosen because the other algorithms only support Information Gain. For all programs (except the AVL tree), Information Gain, Precision, and Relative Precision all performed markedly worse than the other metrics. Contrast, F1, Jaccard, Ochiai, Relative F1, and Relative Jaccard all performed about the same (with some small differences). This answers RQ1: the best metrics to use are Contrast, F1, Jaccard, Ochiai, Relative F1, or Relative Jaccard. It also answers RQ2: Information Gain, despite being the metric of choice in previous studies, was not competitive with five of the other suspiciousness metrics considered in this empirical study. As shown in Table 6.12, Information Gain had the worst average performance of all of the metrics considered.

6.6.2 Which Algorithm Performs the Best?

Figure 6.5 compares Branch-And-Bound, sLeap, and SWRW in various configurations. Branch-And-Bound (B&B) and sLeap both have two versions. The first version is the one discussed in Section 6.4. The second version (denoted "B&B (max)" and "sLeap (max)") imitates the behavior of SWRW by finding only maximal suspicious subgraphs. SWRW was run in 4 configurations: `k-walks 1` (demonstrating `k_walks` with $k = 100$ from Listing 6.4) and 3 configurations of `walk-top-colors` (demonstrating `walk_top_colors` from Listing 6.4 with: $p = .2$, $w = 2$; $p = 1$, $w = 2$; and $p = 1$, $w = 10$).

The top graph in Figure 6.5 shows the average fault localization performance (log scaled) for each algorithm on each dataset. SWRW in all configurations performed significantly better than Branch-And-Bound and sLeap. This can be seen by examining the third and fourth graphs, which "zoom in" on details for the HTML parser and the Go compiler, respectively. Since Branch-And-Bound and sLeap were limited to finding 50 subgraphs, they often failed to mine any subgraph containing the fault. In comparison, SWRW usually sampled a subgraph containing the fault. This answers RQ3: SWRW provided the best fault localization performance using Information Gain. As noted above, it provided even better performance when alternative suspiciousness metrics are used.

The second graph in Figure 6.5 shows the average execution time for each algorithm on each dataset. The default configuration of SWRW (corresponding to the blue bar) was able to extract the suspicious behaviors in less than 2 seconds for all of the datasets except that for the Go Compiler (for which it took 20 seconds). In comparison, Branch-And-Bound often timed out and had an average execution time of 30 seconds or more. sLeap performed much better than Branch-And-Bound (thanks to its pruning heuristic) and was competitive with SWRW. However, as was shown in the first, third, and fourth graphs, SWRW had much better fault localization performance, which is our ultimate goal.

6.6.3 Summary of Results and Threats to Validity

SWRW outperformed the other discriminative subgraph mining algorithms at behavioral fault localization. Information Gain, Precision, and Relative Precision all performed markedly worse than the other suspiciousness metrics. The previous mining algorithms use the Information Gain metric, and a technical restriction (see Section 6.4) prevents them from using any of the other metrics considered in Table 6.2. Since SWRW is not bound by the same

restrictions as algorithms in the Branch-And-Bound family, SWRW appears to be a better algorithm to use for behavioral fault localization.

This study only considered suspicious-behavior based fault localization (SBBFL) algorithms. The results are intended to indicate which of the algorithms considered performed the best at the fault localization task. Further work is needed to fully evaluate SBBFL algorithms against other approaches to fault localization (statistical or otherwise). Also, the study considered only injected mutation faults. The results may not fully generalize to other types of faults. While our study tried to consider a representative set of suspiciousness metrics, many other metrics are available [99] and the results shown here may not fully generalize to those metrics. Similarly, although the programs used in the study were chosen to represent programs with varying degrees of complexity, it was also necessary to choose programs with readily available inputs, which limits the generalizability of the results.

Another important threat to validity (for both our study and previous studies [24, 34, 111]) is the use of the fault rank cost measure. Recall that this cost measure is the expected number of behaviors a programmer would consider before considering a behavior containing the faulty location(s). This measure is simple to compute and explain but it does not consider the variable effort a programmer might need to expend on behaviors of different sizes. Furthermore, while the expectation takes into account the faulty location appearing in multiple behaviors it doesn't take into account a programmer skipping behaviors which contain locations the programmer has already examined.

6.7 DISCFLO: Integrating SBBFL and CBSFL

This section introduces a prototype tool DISCFLO which integrates coverage based statistical fault localization (CBSFL) [99], suspicious behavior based fault localization (SBBFL) [24], and test case minimization (also called delta debugging) [157]. The basic idea is to use SBBFL results to *re-weight* the results of CBSFL. The suspicious behaviors are then used to help *explain* to the programmer why each location is marked as suspicious. The explanation comes in two forms. First, the behaviors are visualized and presented to the programmer. This helps the programmer get a sense of suspicious flows in the program. Second, the behaviors are used as *test case minimization targets*. In delta debugging [157] a test case is made successive smaller until the program no longer fails. The failure is detected by a *failure oracle* which automatically detects the subject programs failure. Instead of detecting failure, in DISCFLO detects the presence of a specific suspicious behavior: the minimization target. This allows for the creation of a minimal test cases which cause the program to exhibit the suspicious behavior when executed.

One problem not yet discussed in this chapter with SBBFL is *linkage* between suspicious behaviors. Many behaviors reported by SBBFL may be very similar to each other. Even though SBBFL was formalized to only extract maximally sized behaviors many of those behaviors can share common substructures. Reviewing the same common structure over and over again holds no debugging value for a programmer. To address this problem DISCFLO uses density based clustering to group suspicious behaviors together. The clusters are then used to re-weight the CBSFL results.

One final element of DISCFLO addresses a fundamental concern with automatic fault localization: false positives. A recent study by Kochhar *et al.* [82] used a survey to explore practicing programmers' expectations for fault localization. The majority of programmers who responded felt the idea of fault localization was worthwhile one. However, they had

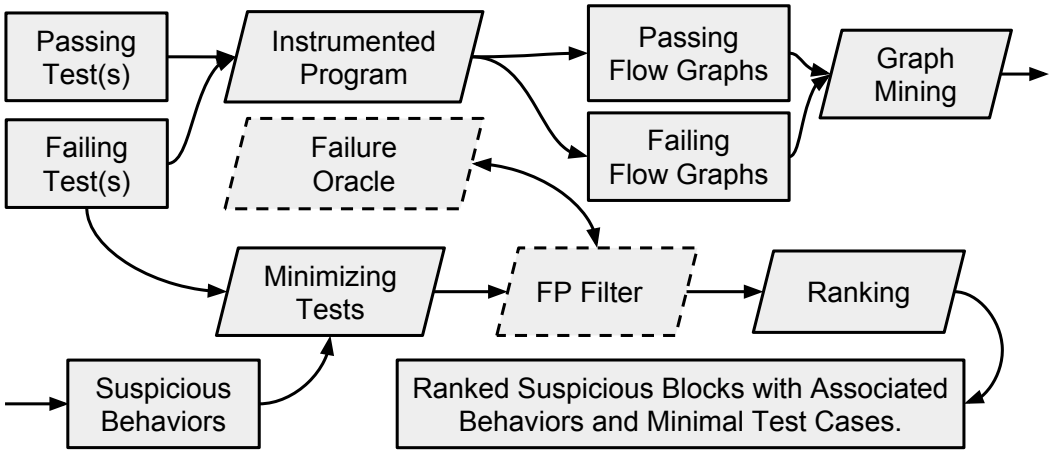


Figure 6.6: An overview of the DISCFLO fault localization system

extremely high expectations for the accuracy of a localizers demanding the faulty statement appear in the top 5 to 10 statements in the list.

DISCFLO can use its test case minimization system to exclude potential false positive suspicious behaviors. The exclusion of all suspicious behaviors which contain a particular program location (basic block) results in the exclusion of that location. This process is fully automatic if the programmer can supply a program (called an *oracle*) to determine the subject program fails when a running a test input. Even without an oracle the filtering process can still be conducted manually by a programmer.

The manual false positive filtering process has the advantage of integrating DISCFLO into a programmer's debugging work flow. In the future, this integration can be made tighter. For instance, when a minimal test case is run through the subject program there is no reason why DISCFLO could not automatically insert debugging breakpoints at the basic blocks involved in the suspicious behavior the minimal test case was produced from. This would allow the programmer to quickly zero in on whether or not that behavior is involved in program failure.

6.7.1 Clustering Suspicious Behaviors

Behaviors returned by SBBFL are often quite similar too each other. This is analogous to the situation for code clones detected from program dependence graphs described in Section 5.4. The reasons for both problems are the same: frequent and suspicious subgraphs both form lattice structures (see Figure 2.2). Maximal frequent or suspicious subgraphs share common sub-structures with each other. These common sub-structures can comprise a majority of the entire structure. Reviewing more than one of the structures that share the common sub-structure is often wasted effort both in fault localization and in code clones.

Thus, as in Section 5.4 DBSCAN [43] (a density based clustering algorithm) is used to cluster the suspicious behaviors. In density based clustering, items α and β cluster together if the distance between them, according a metric δ, is less than ϵ: $\delta(\alpha, \beta) < \epsilon$. Using density based clustering has two advantages over using an algorithm such as k-means. First, k-means is parameterized by the number of desired clusters. This makes the k-means (and

related algorithms) most suitable for applications where the number of clusters is either known or can be easily guessed. However, there is no such foreknowledge in the graph mining application. Second, because density based clusters is uses an ϵ to control when the two items α and β cluster together the rate at which items are inappropriately paired can be directly controlled by selecting smaller values for ϵ and choosing a more appropriate similarity metric δ.

In fault localization (as in code clones) which basic blocks appear in the behavior is more important than how they are connected. Motivated, by this observation the similarity metric used is the Jaccard set similarity coefficient (not to be confused with the Jaccard suspiciousness metric) applied to the sets of vertex labels of the two subgraphs being compared. The sets of vertex labels in suspicious behaviors is the set of basic blocks involved in the behaviors.

Using this Jaccard set similarity, DBSCAN places behaviors together which involve the same (or highly similar) sets of program locations. A programmer can then review just a representative (or two) from the cluster rather than reviewing every behavior individually. This saves the programmer time and effort. The real question in the fault localization context is how to rank and score clusters of behaviors. As shown in Tables 6.1 and 6.2 the suspiciousness metrics for subgraphs are defined in terms of the likelihoods of those subgraphs appearing in the dynamic flow graphs from passing and failing program executions, respectively. Since the likelihood of appearance obeys downward closure (see Definition 2.9, the likelihood of each substructure appearing is at least as great as the likelihood of the whole structure appearing. Therefore, subgraphs which share large portions in common with each other will have similar suspiciousness scores.

Since, everything in a cluster has similar scores the sensible way to assign the score to a cluster is to take the average. This cluster score neglects on aspect of the cluster which is the number of items in the cluster. The more items in a cluster the more likely items from that cluster are sampled by the SWRW algorithm. SWRW samples more suspicious subgraphs more frequently than less suspicious subgraphs. Therefore, if there are many subgraphs in one cluster the whole cluster is more suspicious. This leads to a size weighted average. To prevent the size weight from dominating the score a square root is taken.

$$C_{\text{score}}(c) = \sqrt{\frac{|c|}{|c|+1}} \left(\frac{\sum_{h \in c} \varsigma(h)}{|c|} \right) \tag{6.12}$$

6.7.2 Minimizing Test Cases with Suspicious Behaviors

Suspicious behaviors are program elements which are linked together by control flow. Behaviors may have multiple locations where the program many enter the structure during execution. Furthermore, the behaviors discussed are mined from flow graphs rather than execution traces. An execution trace is the sequence of locations the program executed. Thus, if the program executes a location multiple times (as in a loop) there will be an entry in the trace for each execution. In contrast, a flow graph summarizes this information by having only one entry for each location (see Figure 6.2). Thus, any given behavior may not represent a linear execution of the locations involved.

Since the behaviors may not represent a linear execution of the locations involved reproducing the behavior may not be trivial for a programmer. To solve this problem, we developed a new test case minimization technique (see Listing 6.5). Test case minimization (also known as Delta Debugging [157]) takes a test case and repeatedly removes portions of

it. Each time a smaller test case is created the program is run. If the program fails when it is run with the new test case then the test case is kept otherwise it is discarded. Using the smallest test case found so far as the input the process is repeated until no smaller test case can be found.

The key to Delta Debugging is deciding whether or not a program exhibited failure. For some bugs which cause the program to crash or obviously misbehave it is easy to decide if the program failed or not. However, there are other bugs which cause subtle changes to the output which are difficult to detect. For these, bugs constructing a program (called an oracle) to automatically determine if the subject program failed may be difficult.

The suspicious behaviors found with SBBFL techniques are considered suspicious because the metrics indicate their presence is strongly correlated with failure. If a behavior only occurs when the program fails and never when it does not fail it can be used as a *surrogate oracle* to detect failure. This relieves the programmer from the job of writing a potentially complex program to determine if the program failed or not.

However, there are two problems with this idea. The first is the metrics detailed Table 6.2 do not guarantee that a behavior always occurs when the program fails and never occurs when it doesn't. (Although, if such a behavior exists it will almost certainly have the top score!) The second problem is the program may have multiple faults. A behavior that occurs when the program fails may simply be a symptom of the failure and the fault. Using a symptom as a surrogate oracle in a multi-fault program may not isolate the faults from each other.

When using a suspicious behavior as a surrogate oracle for test case minimization what really happens isn't the isolation of the failure but the isolation of the *behavior*. Listing 6.5 shows the pseudo code for test case minimization in DISCFLO. The function `minimize` takes a test case (which is a string of bytes) and a subgraph which represents a suspicious behavior. It returns a smaller test case which when executed by the program the subgraph is contained in the execution's dynamic flow graph. The smaller test case is minimal as no smaller test case can be constructed from it by removing a sequence of bytes such that its execution's flow graph would still contain the subgraph.

The algorithm uses a helper function `minimizing_mutants` to generate smaller test cases. The smaller test cases are constructed by removing sequences of bytes from the supplied test cases. For example, all suffixes and prefixes of the supplied test are created. Also created are test cases which remove substrings from the middle of the test case. The function in the listing is an example function. More complex functions can be created which are aware of the syntax of the subject program's input.

6.7.3 Filtering False Positives with Minimal Tests

A false positive in fault localization is any location in the program that does not contain the root cause of the fault. A perfect fault localizer would only report the faulty locations and no other locations at all. In general, programmers have indicated [82] they have little patience for reviewing suggested locations which do not directly contain the fault. Unfortunately, past CBSFL techniques have not performed up the standard required by their potential users [82].

In the Section 6.7.2 a new method for generate behavior-specific tests was introduced. However, while the minimized tests produced by the algorithm in Listing 6.5 reproduces the suspicious behavior but it does not necessarily reproduce the failure of the program. What can be concluded if the minimized test no longer causes the program to fail? The behavior

```
# param test: string, the test case to minimize
# param subgraph: graph, the subgraph the minimal test case's dynamic flow graph
#                        should contain
# returns a smaller test case (if one exists) whose execution's dynamic flow
# graphs contiain the given subgraph
def minimize(test, subgraph):
    cur = test
    while True:
        if len(cur) <= 1:
            return cur
        found = False
        for mutant in minimizing_mutants(cur):
            try:
                profile = execute(mutant)
            except ExecutionException:
                continue
            if profile is None:
                continue
            if subgraph not in profile:
                continue
            found = True
            break
        if found:
            cur = mutant
        else:
            return cur

# param test: string, the test case
# returns a list of mutants of the test case all made by creating substrings of
# the test.
def minimizing_mutants(test):
    suffix = [
        test[:i]
        for i in xrange(1, len(test)-1)
    ]
    prefix = [
        test[i:]
        for i in xrange(1, len(test))
    ]
    blocks = [
        test[:i] + test[j:]
        for i in xrange(1, len(test))
        for j in xrange(i+1, min(i + min( max(15, int(.1*len(test))), 100), len(test)+1))
    ]
    return prefix + suffix + blocks
```

Listing 6.5: Python psuedo code for minimizing a test case.

does not completely describe the fault. For instance, if particular location of the program was buggy in such a way as to always cause the program to fail whenever executed then a behavior containing just that location would completely describe the fault. A minimal test case for that location would result in a test which always resulted in program failure. If a behavior-minimized test case no longer makes the program fail then that is strong evidence that the behavior is a false positive and can be filtered out of the results.

In Section 6.7.2 the suspicious behaviors served as *surrogate oracles* with which to conduct the minimization. However, in order to filter false positives an actual failure oracle is needed. A failure oracle is program which runs a test case in the subject program and determines if the subject program failed or not. For bugs which cause obvious failures such as program crashes failure oracles are easy to obtain. For other bugs they are difficult and labor intensive to write.

A failure oracle may be so labor intensive to create that it is not worth creating one to apply a techniques like Delta Debugging. Delta Debugging requires an automatic oracle because it will run the program hundreds or thousands of times to produce a single minimized test case. To check to see if a suspicious behavior is a false positive the output of the program only needs to be checked for failure once. Thus, even if a programmer does not want to invest the time to create failure oracles they can still apply false positive filtering by manually checking the output of the program.

6.7.4 Re-weighting CBSFL Results with SBBFL

Once DISCFLO has extracted suspicious behaviors, clustered them, and optionally applied false positive filtering the results can be displayed to the user. Rather than displaying a list of clusters of subgraphs of dynamic basic block flow graphs, DISCFLO displays a ranked list of locations. The list is displayed as part of a web application. In the application each location is click-able. Clicking on a location gives displays associated suspicious behaviors and lets the user take a variety of actions (such as triggering an additional test case minimization, manually filtering out a false positive, etc...)

To produce the list DISCFLO re-weights the results of a CBSFL metric. In theory a separate metric could be used in the SBBFL portion and in the CBSFL portion but currently the system always uses the same metric. To construct the list, DISCFLO collects all of the locations that appear in any suspicious behavior found by SBBFL. This may not include all of the locations in the program (especially if automatic false positive filtering was used). Each of those locations l is scored using a standard CBSFL suspiciousness metric $s(l)$ (see Section 6.3). Then each location is weighted by the average score of the scores of the clusters it appears in.

Definition 6.6 (DISCFLO Location Score)**.** *Let C be a list of clusters of suspicious subgraphs of dynamic basic block flow graphs. Let L be the set of locations that appear in any subgraph in any clusters in C. Let $m : L \to C^*$ be a function that maps a location l to the set of clusters it appears in. The score for a cluster $C_{score}(c)$ is as defined in Equation 6.12. The score for a location is $s(l)$. The DISCFLO score for a location l is*

$$D_{score}(l, m) = s(l) \frac{\sum_{c \in m(l)} C_{score}(c)}{|m(l)|} \tag{6.13}$$

DISCFLO computes the score D_{score} for each location and then orders the locations in the list by their scores. The highest scoring locations are the most suspicious and appear at the

AVL Relative-F1			
Bug	CBSFL	DFLO	DFLO + FP
1	2.0	2.0	2.0
2	4.0	2.5	3.0
3	3.5	2.5	2.5
4	8.0	43.5	43.5
5	5.0	1.5	1.5
6	93.0	47.5	47.5
7	5.5	4.0	4.0
8	2.0	2.0	2.0
9	7.0	30.5	25.5
10	7.0	1.0	0.5
11	0.5	0.5	0.5
12	4.0	3.0	2.5
13	4.0	5.5	0.5
14	1.5	1.5	1.5
16	6.0	3.5	3.5
17	6.0	3.5	3.5
18	2.5	2.5	2.5
19	5.0	4.0	4.0
20	8.0	22.0	22.0

Table 6.13: Comparison of the performance of CBSFL to DISCFLO with an without false positive filtering using the RF1 metric for the AVL Tree.

beginning of the list. The lowest scoring locations are the least suspicious and appear last in the list. Since DISCFLO produces a ranked list just like CBSFL its performance can be compared directly to CBSFL's performance.

6.7.5 Empirical Evaluation of DISCFLO

DISCFLO's fault localization performance was evaluated on the three of programs (AVL, Blackfriday, and HTML) shown in Table 6.11 which were previously used in Section 6.6. Tables 6.13, 6.14, and 6.15 show the results of the evaluation. Each table compares the fault localization performance of DISCFLO (with and without false positive filtering) to coverage based statistical fault localization (CBSFL) using the Relative F1 suspiciousness metric. DISCFLO was configured to use SWRW as the SBBFL algorithm with the Relative F1 objective function. SWRW was run using `walk-top-colors` (demonstrating `walk_top_colors` from Listing 6.4 with: $p = 1$ and $w = 10$. To assess the impact of the false positive filtering 10 tests were minimized for each buggy program version. This provides DISCFLO the opportunity to discard at most 10 suspicious behaviors which are not related to the fault.

A fault rank cost measure was developed to assess the fault localization performance of DISCFLO and CBSFL. The locations reported by both algorithms are scored and presented to the programmer in ranked order with the most suspicious location first. The fault rank gives the expected number of locations a programmer would examine before examining a behavior containing the faulty location(s). The fault rank gives an objective score enabling comparison between CBSFL and DISCFLO.

As can be seen in the datatables, DISCFLO often (but not always) outperforms CBSFL. DISCFLO's superior performance at fault localization supports the idea of using information from suspicious behaviors to re-weight the localization scores provided by standard CBSFL suspicious metrics. DISCFLO's false positive filtering mechanism often improves

HTML Parser Relative-F1			
Bug	CBSFL	DFLO	DFLO + FP
1	1.0	1.0	1.0
2	22.5	173.5	177.5
3	41.5	9.0	12.5
4	13.5	13.5	8.5
5	28.5	281.5	281.5
6	32.0	48.0	46.5
7	1.5	1.5	1.5
8	13.5	10.0	3.0
9	13.5	1.0	5.5
10	12.0	9.0	12.0
11	4.5	213.0	211.0
12	45.0	106.5	118.5
13	∞	∞	∞
14	5.0	5.0	4.0
15	12.5	18.0	3.0
16	3.5	3.5	3.5
17	5.5	10.0	9.0
18	28.0	268.0	263.0
19	13.0	1.0	5.0
20	6.5	4.0	4.0

Table 6.14: Comparison of the performance of CBSFL to DISCFLO with an without false positive filtering using the RF1 metric for the HTML Parser.

Blackfriday Relative-F1			
Bug	CBSFL	DFLO	DFLO + FP
1	7.0	5.0	5.0
2	55.5	57.5	57.5
3	1409.5	1444.5	1444.5
4	1363.0	1457.0	1457.0
5	2.0	2.0	2.0
6	5.0	8.5	8.5
7	7.0	7.0	7.0
8	152.5	120.5	92.5
9	12.0	11.0	10.5
10	28.0	9.5	9.5
11	348.0	474.0	475.0
12	668.0	378.0	378.0
13	0.5	0.5	0.5
14	120.0	1611.0	1607.0
15	2.5	2.5	2.5
16	1399.0	1416.0	1416.0
17	3.0	1244.0	1246.0
18	573.0	632.0	631.0
20	30.0	1243.0	1238.0

Table 6.15: Comparison of the performance of CBSFL to DISCFLO with an without false positive filtering using the RF1 metric for the `blackfriday` Markdown library.

DISCFLO's fault localization accuracy. For instance for the HTML Parser's Bug-15, DISCFLO's fault rank improved from 18 to 3.

6.8 Conclusions

We presented Score Weighted Random Walks (SWRW), a new algorithm for suspicious-behavior based fault localization (SBBFL). SWRW randomly samples suspicious subgraphs of dynamic control flow graphs of passing and failing executions, favoring selection of the most suspicious subgraphs. Unlike previous algorithms for SBBFL, SWRW may be used with a wide variety of suspiciousness metrics. Nine metrics were adapted from coverage based statistical fault We also presented DISCFLO which extends SWRW with test case minimization. An empirical study was conducted on five real world programs written in the Go programming language. To support the study a new profiling tool for Go, Dynagrok, was developed. The results indicate that SWRW is more accurate and scalable than similar behavioral fault localization algorithms.

Chapter 7

Related Work

Finding graph patterns in program code [22, 59, 72, 83, 85, 93, 106, 108] and behavior [24, 34–36, 94, 97, 100, 103, 111, 137, 156] is a powerful tool for solving pressing challenges in software engineering. Frequent subgraph mining is used to find duplicated code (code clones) [59, 83, 85], identify implicit specifications [22], isolate bug patterns [135, 137], localize faults [24, 34–36, 94, 97, 103, 111, 156], detect plagiarism [93], and construct auto-complete systems [106]. Despite all of these studies showing the great promise of frequent subgraph analysis it is still regarded [16, 119, 123–125], as a curiosity for analyzing program code.

Many of the studies analyzing program code mined variants of the *Program Dependence Graph* (PDG) [46], a labeled directed graph. In PDG's, the vertices represent computational operations such as arithmetic or branch instructions. The edges are labeled and represent either data or control dependence between operations (see Figure 4.1 on page 33). The studies analyzing program behavior largely examined the dynamic flow graphs [24, 94] potentially augmented with extra data. Dynamic flow graphs summarize the control flow paths executed by a program at run time (see Figure 6.2 on page 56). Like the PDG, a dynamic flow graph is a labeled directed graph.

7.1 Frequent Subgraph Mining

Frequent pattern mining was first proposed by Agrawal et al. [4] for the purpose of finding association rules by mining frequent itemsets. Agrawal and Srikant developed the *Apriori* [5] method for frequent itemset mining. Each step Apriori proceeds by finding all patterns of size k using the results of the previous step which found all patterns of size $k-1$. Apriori can be viewed as a level wise exploration of the frequent itemset lattice. Han et al. developed FP-Growth method for finding frequent patterns in [55]. FP-Growth, in contrast to Apriori, grows a single pattern at a time and can be viewed as a depth first exploration of the frequent itemset lattice. Aggarwal provides a recent survey of frequent pattern mining [3].

Many algorithms have been developed for frequent subgraph mining specifically. Classic algorithms such as AGM [71], gSpan [151], MoFa [18], FFSM [66], or GASTON [110] find all the frequent subgraphs in a database. Wörlein *et al.* [148] provides an empirical analysis of these approaches. Other algorithms, for instance SPIN [67], find just the *maximal frequent subgraphs* or the *closed frequent subgraphs* [152]. Finally, Cheng *et al.* [25] provides a recent overview of current directions for mining subgraphs from a database of graphs.

The most important algorithm developed for the transactional setting is gSpan [151]. gSpan presents a number of important innovations. First, it used a depth first search of the frequent subgraph lattice (see Figure 2.2). Using a depth first search greatly reduces the amount of memory required during mining in comparison to conducting a breadth first search. When conducting a breadth first search each "level" of the lattice must be loaded into memory. Because frequent subgraph lattices tends to be much wider than they are deep this causes excessive memory usage.

Second, gSpan introduced the concept of the Minimum DFS Codes for solving the Graph Isomorphism Problem [101]. In Minimum DFS Codes a ordering is established between Depth First Search (DFS) trees. Then using a search algorithm the minimal DFS tree is found according to the ordering. This is used to create a *canonical label* called the Minimum DFS Code to represent the graph and all of its automorphisms. Using the canonical label gSpan can easily check to see if a graph has been previously considered.

Canonical labels are a well established method for solving the graph isomorphism problem [77, 101] and Minimum DFS Codes are not competitive in terms of efficiency compared to modern approaches. gSpan uses Minimum DFS Codes to enable a more important optimization called the *right most extension*. The right most extension allows gSpan to skip the generation of most candidate subgraphs (see Section 3.2.1). Right most extension uses the structure implied by the ordering established between DFS trees to choose only some of the vertices in the graph as potential extension points.

The effective combination of a depth first exploration of the lattice using the right most extension gives gSpan a strong advantage over other methods. However, despite the fundamental importance of the technique there are exposition problems with the paper which confuse rather than illuminate the overall approach. Wörlein's explanation and empirical analysis [148] ameliorates the problems with original explanation and his implementation in ParSeMis[1] is the standard implementation used today.

7.1.1 Mining Connected Graphs

Mining connected graphs is a generalization of the original transactional case. Inokuchi *et al.* [71] first formulated frequent subgraph mining where the database $\mathcal{D}$ is a set of transactions such that each "transaction" is a small graph. Kuramochi and Karypis [87] recognized the limitation of Inokuchi's formulation and re-formulated the problem as mining recurring subgraphs in a single graph. In this setting, the transactional case can be viewed as a special case where the graph being mined is disconnected. When mining for subgraphs of a single graph a key question is how the miner defines the frequency or support of the graph. This question does not even arise in the transactional case but is critical in the single graph case. The most commonly used support metrics are: *Maximum Independent Subgraphs* (MIS) [87], *Minimum Image Support* (MNI) [19], and *G-Measure* [73].

The first study involving mining large graphs introduced the SUBDUE system [27]. SUBDUE predates the work by Inokuchi and does not use the same theoretical framework. Instead, the SUBDUE system attempts to compress the dataset by identifying recurring substructures. Kurakmochi and Karypis [87] created two algorithms to mine frequent subgraphs under MIS. One algorithm, hSiGram, mined using a breadth first search while the other, vSiGram, used a depth first search.

Bringmann and Nijssen [19] did not define a new mining algorithm but they studied the problem of frequency in connected graphs, defining MNI. Jiang *et al.* [73] created G-

[1] https://www2.cs.fau.de/EN/research/zold/ParSeMiS/index.html

Miner and defined the G-Measure support metric, which finds approximately minimally overlapping embeddings. Jiang *et al.* did not know of the work of Bringmann and Nijssen nor that of Kuramochi and Karypis, and they formulated their support measure using a different theory.

Hellal and Romdhane [58] presented NODAR for mining patterns using the SMNOES support metric. It turns out that SMNOES is equivalent to the MIS metric [87], which the authors appear to have been unaware of. SMNOES is formulated using the theory presented in Jiang *et al.*'s work. Unfortunately, NODAR is not competitive with current systems (such as GraMi [39]) according to their published performance numbers.

Elseidy *et al.* [39] presented GraMi[2] which solves support computation through a novel subgraph matching algorithm. Their work is based on the ParSeMiS[3] Framework [148]. GraMi has several innovative pruning strategies and views the subgraph matching problem in terms of constraint solving. One of its pruning strategies, Push Down Pruning, could be viewed as "dual" of Overlap Pruning in Section 3.2.3 (see Lemma 3.2). Push Down Pruning prunes potential vertices of G which have been proven not to be part of an embedding of a previous subgraph. In contrast, Overlap Pruning restricts the matching process to vertices of G which have been part of an embedding of a previous subgraph.

Talukder and Zaki [139] created Dist-Graph, a distributed frequent subgraph miner for large connected undirected graphs that is based on the C++ gSpan [151] implementation in gBoost [86]. Dist-Graph demonstrates an efficient way of distributing MNI computations across a network of compute nodes. Each compute node holds only a partition of the whole graph being mined. The effectiveness of the system was demonstrated on an IBM Blue Gene system mining a billion node graph with high support.

Dist-Graph represents an orthogonal improvement to frequent subgraph mining. The contributions from REGRAX are new pruning techniques, support measures, and a focus on extraction of very low frequency recurring subgraphs. Dist-Graph's sequential mode is gBoost's C++ implementation of gSpan modified to count support using MNI. Because Dist-Graph uses the "store-and-grow" (see Section 3.2.3) approach to support counting it cannot extract low frequent subgraphs from large graphs, as its memory usage increases dramatically. In the future, the techniques in Chapter 3 could be combined with the techniques in Dist-Graph to make an even faster system.

7.1.2 Sampling Techniques

Zou and Holder's 2010 paper [160] developed a sampling technique for finding frequent subgraphs of large graphs. Their method creates a representative sample of the large graph. Frequent subgraph mining is then performed on the sample. Hübler *et al.* devised an approach for selecting subgraphs with properties representative of a large graph in [68]. The sampled subgraphs are not guaranteed to be frequent. Leskovic and Faloutsos concluded in [89] that a number of random walk strategies for sampling subgraphs of large graphs can preserve important properties of the original graph. For frequent itemset mining, a variety of sampling algorithms have also been produced. An early example is Toivonen's algorithm [144].

The Unweighted Random Walk approach to sampling frequent subgraphs was first proposed in ORIGAMI [23]. Henderson and Podgurski [59] provided a more efficient technique to compute sampling probabilities and used them to estimate the overall relevance of code

[2]https://github.com/ehab-abdelhamid/GraMi
[3]https://www2.cs.fau.de/EN/research/zold/ParSeMiS/

clones found with frequent subgraph mining to professional programmers. Musk [7] constructs a Markov process that samples the maximal subgraphs *uniformly* given enough time. Al Hasan *et al.* proposed a Metropolis-Hastings approach [8] to uniformly sample all frequent subgraphs (as opposed to just the maximal subgraphs). FS^3 from Saha and Al Hasan [126] extends the Al Hasan's past work with "sideways" jumps in the frequent subgraph lattice to extract the top-k most frequent subgraphs.

Like the previous work FS^3 allows both forward and backward motion in the lattice which makes this approach less efficient over all at sampling then the forward approaches like GRAPLE and ORIGAMI. However, because both forward and backward motions are allowed the chains Al Hasan defines are ergodic which means they converge to a stationary distribution and if the chains are allowed to mix then sampling from the chains samples from the stationary distribution. Al Hasan defines the chains such that the stationary distribution is uniform over the states of interest. The important point (overlooked by Al Hasan) is the chains must *mix* before sampling is conducted [90]. It is difficult to say in general what the minimum number of transitions is required for arbitrary chains to be mixed. Upper bounds on the minimum tend to be conservative [90] and if taken would make these algorithms impractical due to the cost of each transition in the chain. More theoretical work and empirical work is needed on ergodic chains for sampling frequent patterns to establish appropriate bounds on the required mixing times.

Zhu *et al.* [158] created SpiderMine which mines smaller graph patterns (called spiders) and then joins them together to get much larger patterns. In contrast to other work, SpiderMine only returns the k largest patterns with probability $1 - \epsilon$ where ϵ is a user specified accuracy parameter. SpiderMine is evaluated against ORIGAMI [23] for finding large patterns. Zhu *et al.* conclude ORIGAMI cannot find large patterns. However, the results presented in the paper contradict results found during the GRAPLE study [59] which found many large patterns. The SpiderMine evaluation did not discuss the number of walks ORIGAMI was allowed nor running time making the evaluation suspect.

7.1.3 Significant and Discriminative Techniques

Section 6.4 introduced *significant subgraph mining* [11,17,56,57,86,96,118,127,134,140,149, 150] an alternative to the usual frequent subgraph mining problem. Instead of looking for subgraphs which are at least k frequent in the database in significant subgraph mining the problem is to find the most important subgraphs as judged by some objective function. The objective function (usually denoted $F(g)$ where g is a graph) gives higher scores to more important graphs and lower scores to less important graphs.

There are two settings for significant subgraph mining. In the first, the problem is to find all graphs g such that $F(g) > \delta$ for some importance threshold δ. In this setting F should behave in a generally anti-monotonic fashion ($a \sqsubseteq b \implies F(a) \geq F(b)$) to avoid the enumeration of every subgraph of the dataset $\mathcal{D}$. The second setting searches for the most important graph(s). It is often formulated as finding a graph g^* such that $g^* = \text{argmax}_g F(g)$.

Kudo *et al.* introduced gBoost [86] (which coincidentally DistGraph's [139] code is based off of) which integrates discriminative subgraph mining into gSpan [151]. gBoost was the first study to examine the significant subgraph mining problem and it sought to minimize the error rate of a decision stump classifier for graphs. The objective was flipped and turned into the problem of finding the subgraphs which maximized a certain gain function which is exactly the significant subgraph mining problem. gBoost introduced the first Branch-

And-Bound algorithm (see Section 6.4 for a description) to solve the gain maximization problem.

Following Kudo's work, He and Singh conducted two studies on extracting significant subgraphs [56, 57]. Their first study [56] used CloseGraph [152] to extract *closed frequent subgraphs* and then used *p-values* to identify and rank the significant graphs. CloseGraph is an extension of gSpan [151] which only returns graphs which are on the border of a *frequency boundary* in the frequent subgraph lattice. To compute the *p-values* the frequent subgraph are transformed into feature vectors. The *p-value* of a graph is then computed as the probability of its feature vector occurs in a random distribution of feature vectors. He and Singh's second study [57] looked at providing proven quality guarantees when extracting significant graphs. They formulate their problem as an instance of the k-MST problem. k-MST looks to find the minimum weight spanning tree with at most k vertices. To solve the problem they use dynamic programming to solve k-MST in an un-rooted undirected tree. They then generalize their solution to the context of rooted undirected graphs.

Saigo *et al.* [127] applied *partial least squares regression* to graph data with gPLS. gPLS uses the Branch-And-Bound framework developed in gBoost [86] to extract the patterns which are used in the regression. The regression uses a graph G as the independent variable which determines the dependent real valued variable y. The model is trained on a set of graphs $G_1, G_2, ..., G_n$ and values $y_1, y_2, ..., y_n$ to predict the value y' of a new graph G'. To train the PLS model features are extracted from each graph. The features (significant subgraphs) found using a modified version of the Branch-And-Bound framework which extracts all graphs with significance greater than a certain ϵ.

The next major development in significant graph mining was LEAP Search by Yan *et al.* [150]. Their algorithm is discussed in detail in Section 6.4. Briefly, it LEAP Search proposed to major improvements to the Branch-And-Bound framework. The first was a heuristic algorithm, sLeap, which skipped portions of the search space if they appeared very similar to portions already considered. These "leaps" might skip the most significant graph but most of the time would not. The heuristic algorithm was embedded in a new *frequency descending* algorithm which produced the exact solution.

GraphSig [118] returned to the *p-value* formulation first proposed in He's work [56]. In comparison to the previous work GraphSig improves the extraction of feature vectors. Each graph in the dataset is viewed as an ergodic Markov chain. The states of the chain are the vertices in the graph. The transition probabilities from one state to any of its neighbors in the graph is uniform. Finally, with probability $\frac{1}{\alpha}$ the Markov process returns to its starting node. To build a feature vector a Markov process is started from each vertex. At each transition of the process the feature vector of the starting vertex is updated with the label of the current vertex. The final algorithm in GraphSig then combines mining closed sub-vectors from the feature vector space with frequent subgraph mining. The new SWRW Algorithm from Section 6.5 also uses Markov chains. However, SWRW chain is defined over the suspicious subgraph lattice not over the graphs from the dataset.

Proposing another Branch-And-Bound technique for discriminative mining Thoma *et al.* [140] presented CORK which integrates a different pruning operator into gSpan. The pruning operator is depending on a *submodular* quality function q. Instead of operating on a single graph (like the normal objective function F) q operates on the set of discriminative graphs. The submodular condition ensures that the improvement of adding one more graph to the set of discriminative graphs is marginal. The marginality means that adding the same graph to a smaller set of discriminative graphs always has a greater effect on the quality score q than adding it to a larger one. Much like in gBoost or LEAP Search a upper bound

is found for their chosen quality score. However, unlike in LEAP Search the bound does not generalize to a class of scores and is specific to the score defined in the paper. CORK uses its new bound to help prune the graphs considered by gSpan. However, CORK runs gSpan in a loop each time expanding the set of discriminative graphs by one graph until no improvement is made.

Arora *et al.* [11, 17] considered a new take on the problem. They were interested in graphs which were created from spatial or geographical datasets. The vertices in the graphs considered each has one label drawn from drawn from a known or assumed distribution of labels. They considered both discrete labeling distributions and continuous. Using this assumption the *chi-square* (X^2) statistic was used to quantify the statistical significance of a subgraph. The problem they considered was extracting the top-k most significant subgraphs according to the X^2 statistic. Their unique approach identified the strongly connected components (their graphs are undirected) in the graphs. The connected components were then contracted into a single vertex in a new "super-graph" of the original graph. As long as the new "super-graph" is "dense-enough" a naive enumeration of all possible significant subgraphs of the "super-graph" is conducted to identify the top$k$ most significant subgraphs.

7.2 Code Clones

Code clones have been a rich area of research for many years and there a several recent surveys available [16, 119, 124, 125] which cover the full range of detection and management techniques. Recent work by Sajnani *et al.* on SourcererCC [128] showed that clone detection can scale to 100 MLOC when programs are represented as bags-of-tokens. It took SourcererCC only $1\frac{1}{2}$ days to find all of the clones from an artificially constructed code base consisting of 100 MLOC. In the SourcererCC study, the latest version of CCFinder [79] (CCFinderX) was competitive with SourcererCC on most benchmarks in terms of time, precision, and recall.

Krinke's Duplix algorithm [85] and Komondoor and Horwitz's algorithm [83] were the first attempts at detecting code clones from PDGs. Komondoor's algorithm found pairs of clones by slicing backwards and then forwards from matched starting vertices. However, the forward slicing operation is only applied when matching control vertices are discovered. Komondoor also applies a variety of heuristics to filter out certain types of clones. After pairs are identified, ones that include the same locations are grouped together and subgraphs are discarded in favor of their super-graphs. Higo and Kusomoto [61] created Scorpio, which extends Komondoor's algorithm to detect contiguous clones by adding links into the PDG. Krinke's algorithm Duplix is similar to Komondoor's but restricts itself to forward slicing up to a limit of k edges. Both of these algorithms, unlike the one presented in this paper, tend to find long paths through the PDG instead of general subgraphs (containing any edge structure) such as the one in Figure 5.1.

Gabel et al. [48] introduced another formulation by mapping PDG's to abstract syntax trees. They used the well known AST based clone detection tool DECKARD [72]. Higo and Kusomoto [61] extended Komondoor's algorithm to detect contiguous clones by adding execution dependencies to PDGs. They also merge similar nodes in a PDG. ModelCD from Pham et al. [112] detects code clones in Matlab/Simulink models by converting them to labeled directed graphs. They present an exact and an approximate algorithm both of which traverse the connected subgraph lattice. The exact algorithm uses a depth first search in the manner of vSiGraM [87]. The approximate algorithm is a breadth first search with a pruning

step. Higo et al. [62] presented an incremental approach to PDG based code clone detection, to improve its scalability. Part of their solution is to make the mining process interactive by having the user request clone information for specific files or methods. Hummel et al. [70] also present an incremental approach, but one intended for Matlab/Simulink models. They index the model graphs using small frequent graph motifs, allowing for incremental updates to the index and a speed up for detection.

Nguyen et al. [105, 107] created a two code completion tools based on the code mining tool in [109]. Both tools work by first building a graph database of *groums* which are object/object interaction graphs. The tool described in paper [107] builds a query groum based on the surrounding context. It then uses a weighted graph matching algorithm to find suggestions using a relevance score. The second tool [105] uses a Bayesian statistical model to suggest likely completions based on the surrounding context and the graph database.

7.3 Fault Localization from Behavior Graphs

There have been a number of studies [24, 26, 34–37, 94, 97, 100, 103, 111, 156] combining the statistic fault localization approach with graph mining. In general, the framework is similar to the statement coverage approach [99]. The program is instrumented to collect a record of runtime behavior. However instead of collecting statement executions, the instrumentation records a *dynamic flow graph* which provides a context insensitive approximation of the relative ordering of statement execution. Such a graph can be fine grained and collected at the *basic block* level or coarse grained and collected at the procedure level. Figure 6.2 shows an example dynamic basic block flow graph.

C. Liu, X. Yan, H. Yu, J. Han, and P. Yu [94] first introduced the idea of mining program behaviors to localize faults. Xifeng Yan and Jiawei Han had previously created gSpan [151], CloseGraph [152] but had yet to create LEAP Search [150] at the time of this paper. The big idea was to find "backtraces" for program failures which do not cause the program to crash. The backtraces found are portions of the behavior graphs which are associated with failing runs. The paper trains *support vector machine* classifiers with linear kernels for each function entrance and exit. The classifiers predict whether the program failed or not but are only trained on data up to the associated entrance or exit (collected via crafted unit tests). The classifiers uses frequent subgraphs mined with a variant of CloseGraph [152] called CloseMine as the features of each execution in the training set. The final step is to use the accuracy of the classifiers on the training set to identify which methods are relevant to the bug and assemble them into a "backtrace."

Di Fatta *et al.* [34] proposed an alternative application of discriminative pattern mining using *function call trees*. The call trees are collected from executions of the program in a similar way to flow graphs but distinguish between different calling contexts (unlike flow graphs which coalesce the contexts). Di Fatta uses a frequent subtree mining algorithm (FREQT) to extract frequently occurring subtrees from the collected call trees. The frequent subtrees collected are size limited and are collected independently from both the failing and succeeding executions. They then use the Precision metric (see Table 6.2) to compute how suspicious each frequent subtree is. The evaluation compared coverage based statistical fault localization (CBSFL) [99] using the Precision metric to the proposed method. In the evaluation a bug was considered localized once the subtree which contained the faulty location was considered. This evaluation standard was unfair to CBSFL as CBSFL ranks each location individually. Many of the later studies repeat this oversight.

One problem Di Fatta *et al.* encountered when mining call trees was scalability limitations. Di Fatta addressed theses issues by having extremely stringent limits on the number of edges allowed in a mined subtree; allowing at most 4 edges. One reason for Di Fatta *et al.* may have had difficulty mining larger subtree is due to the structure of the call trees themselves. Most call trees contain repeated calls and redundant information. Eichinger *et al.* [35] recognized this limitation and applied a reduction algorithm on the call trees. The algorithm transformed the call trees produced by the profiler they used into edge weighted function flow graphs. These are similar to the graph shown in Figure 6.2 but are more coarse grained only containing the vertices which represent functions.

The edge weighted function flow graphs (also called weighted dynamic call graphs) were mined using the CloseGraph [152] algorithm that was implemented by Wörlein [148] in ParSeMiS. Eichinger then used the Information Gain metric (see Table 6.2) to rank both the behaviors mined with CloseGraph and the methods in those behaviors. The locations in the method are also ranked using finer grained coverage information. The version of InformationGain used is modified from the version presented in Table 6.2. It takes into account not only the number executions the behavior appears in but also the edge weights which indicate the number times each edge in the graph was traversed.

D. Lo, H. Cheng, J. Han, S. Khoo, and C. Sun [97] developed an approach for using identifying discriminating iterative sequence patterns to classify whether or not an execution trace was recorded from a failing execution. This work is not strictly fault localization work but it is closely related. The mined patterns are *iterative patterns* which extend sequence patterns by including a repetition operator. The paper defines a mining algorithm to detect the patterns and uses the detected patterns as features in a classifier.

H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan [24] used LEAP Search [150] by X. Yan to localize faults. The faults were localized to suspicious behaviors which are subgraphs of either dynamic basic block flow graphs or dynamic function flow graphs (call graphs). Their approach is covered in detail in Section 6.4.

HOLMES [26] recognizes the potentially high amount of overhead introduced by detailed profiling techniques needed to collect the data to enable techniques like Cheng's [24]. HOLMES consists not only of a localizer which finds suspicious execution paths but also an adaptive profiler. The system automatically increases the granularity of suspicious areas of the program based on profiles, bug reports and other information from the field. At the finest granularity HOLMES collects path profiles [15]. The localizer component finally computes a statistical association metric to identify the most suspicious paths in the program.

In 2010 Eichinger followed up on the work from 2008 [35] with a new mining approach which included dataflow information in the call graphs [36]. The call edges were annotated with both the number of times the edge was traversed and the parameter values which were passed between the caller and the callee. Primitive values were recorded as, but strings, collections, and array values were reduced to their size. To produce the graphs the values were discretized by forming intervals. The intervals then became categorical labels on the edges. Eichinger once again used CloseGraph [152] from ParSeMiS [148] to mine frequent subgraphs from the set of dataflow enabled call graphs. Then following closely with the previous study [35] a new suspiciousness score was derived from Information Gain that made use of the extra information on the edges of the graph.

Then in 2011 Eichinger [37] applied an approach to solve the same problem HOLMES addressed. Recall, HOLMES was sensitive to the large amount of overhead potentially required to collect execution traces and call trees. Eichinger provide a general method for hierarchical localization. The hierarchy can in principle include any number of levels

but they consider three: package, class, and method level localization. By introducing the hierarchy Eichingers approach can potentailly save profiling overhead but it also decreases the difficulty of the graph mining step [152].

Parsa *et al.* [111] present an alternative algorithm to LEAP Search [150] for finding the top-k discriminative subgraphs for fault localization. Their algorithm fits into the Branch-And-Bound framework and they claim similar performance to LEAP Search. They define their own suspicious metric which they call the $\mathcal{F}_{\text{Score}}$. Since their algorithm is a Branch-And-Bound algorithm they also provide an upper bound for their score. However, unlike the LEAP Search paper their algorithm description is lacking in important details and imprecisely specified.

Mariani *et al.* [100] combine Dynamic Specification Mining proposed by Ammons [10] and fault localization. They learn a finite state automata model of the relationships between methods associated with particular objects in the program. The models are built from execution traces and I/O models produced by Daikon [41]. The behavior models are scored and ranked much like in other approaches to identify anomalous behaviors.

Call tree based defect localization is revisited by Yousefi and Wassyng [156]. Like Eichinger [35–37], Yousefi and Wassyng start with tracing but instead of producing weighted call graph they produce reduced call trees. They then apply a frequent subtree mining algorithm to identify frequent closed subtrees from the reduced call trees of successful executions. They then define a number of metrics on the subtrees which are used for scoring the subtrees. The subtrees are then matched against failing reduced call trees to find missing method calls or deviations from the mined patterns. In this way the mined subtrees act as dynamic specifications [10].

7.4 Test Case Generation and Minimization

There has been some previous work on test case generation that is driven by behavioral properties. For instance, the American Fuzzy Lop (AFL) fuzz tester uses feedback from the behavior of the program to select the next best mutation to explore.[4] Like Dynagrok (my proposed instrumentation system), AFL instrumentation collects a fine grained dynamic flow graph of the program. Unlike Dynagrok which uses source level instrumentation, AFL does so by instrumenting the target binary.

Patrice Godefroid [50–53] has created a number of systems for generating test cases using feedback from both the behavior the program under test and from symbolic execution of the program. Andreas Zeller created *Delta Debugging* [157] which is a strategy to minimize test cases with respect to a failure oracle (usually the program crashing). AFL includes a test case minimizer based on delta debugging which uses program crashes as the oracle. In contrast to this work, my proposed system would minimize the test cases with respect to a frequently occurring behavioral graph fragment which is correlated or even potentially *causally* related to the failure [12, 13]. This minimization strategy can be combined with a failure oracle (if one exists) to prevent false positives.

[4] AFL is not academic work but is arguably the most effective general purpose fuzzing system that is widely available today. It has found many critical bugs in many different types of programs. For more information see: http://lcamtuf.coredump.cx/afl/ and http://lcamtuf.coredump.cx/afl/technical_details.txt

7.5 Specification Mining

Specification mining is a broad field spanning static [2, 9, 21, 22, 40, 47, 78, 84, 91, 95, 102, 109, 116, 117, 131, 135, 136, 145, 147] and dynamic [10, 28, 31, 32, 38, 41, 42, 98, 115, 121, 153, 154] approaches. The modern study of Specification Inference or Mining Specifications can be traced back to Ernst et al.'s paper "Dynamically Discovering Likely Program Invariants to Support Program Evolution" [41]. The Ernst approach is a dynamic analysis and it works by instrumenting the target binary. The binary is instrumented to capture the values of variables at various program points. The algorithm then took those values and tried to inference invariants for the program variables. For example the algorithm might infer[5]

```
1  // pre-condition: N >= 0, N = length(B)
2  // post-condition: S = sum(B)
3  func f(B []int, N int) int {
4      i, s := 0, 0;
5      while (i != N) { // invariant: i > 0, i <= N
6          s = s + B[i];
7          i = i + 1;
8      }
9      return s;
10 }
```

In order for this to work a reasonable test suite needs to be provided. This has become one of the hallmarks of specification mining – dynamic analysis coupled with a test suite. The work is some what unusual however in that it tries to infer program invariants, later works will focus heavily on temporal orderings. There are however some problems with their approach. The first is the Ernst system is not fully automated requiring human operation at various points. Automation is key for adoption and later work attempts to address this issue with varying degrees of success.

7.5.1 Graphical Specification Mining

An important technique for mining specifications was first described in 2008 by Chang *et al.* [21,22,135,136]. The Chang et al. system works by mining Program Dependence Graphs (PDGs) [30,46,64,65,104,113,114,143]. It begins the mining process by using a commercial tool, Code Surfer, to construct the *System Dependence Graph* (SDG) [65]. The SDG is further annotated with new edges called Shared Data Dependence Edges (SDDEs) which connect nodes which are related by control flow and share a direct data dependency. These edges have turned out to be useful in practice for mining rules where such relations are semantically important. The SDE with the addition of the SDDEs is referred to as the *Enhanced System Dependence Graph* (ESDG).

After the ESDG is constructed candidate centers for mining are identified. In principle one could start with every node in the ESDG however in this work the authors choose only function call sites. Once the candidate nodes are identified a dependence sphere is grown around the candidates. This starts by identifying the *Call Site Graph* graph which contains all nodes related to formal parameters of of the function call. The sphere is then grown by adding nodes which have a direct data dependence relation with some node in the sphere until a threshold size is reached. Finally, the sphere is pruned by a process called reduction to remove nodes likely to contribute noise to the mining process. During reduction certain transitive relations are also made more explicit by introducing new edges. The final

[5] This example is adapted from the example provided in the paper. See figures: 1-3

dependence sphere is referred to as the *Fully Reduced Dependence Sphere* or FDRS in the paper.

The mining algorithm presented in the Chang et al. work uses a two phase process to first grow a frequent item set of connected nodes in a graph minor and second to add frequent edges between the nodes in the minor. The algorithm starts with the nodes in the Call Site Graph surrounding the candidate node. It then employs an iterative extension process where at each step a new node is added to the set. It only adds nodes which are directly connected by some edge to some node already in the set. It only adds such a node if the edge the nodes is connected by is found to be frequent via Frequent Item Set mining. Care is taken during the extension process that isomorphic instances will have identical labellings.

Once the nodes which constitute the frequent minor have been identified the edges must be selected. As with node selection, frequent item set mining is used as a subroutine to find frequent edges between nodes in the minor. Finally, some filtering takes place as a post-processing step to remove trivial frequent minors which are likely un-interesting.

The Chang system has been extended in a variety of ways. The above description describes the latest version as described in [21]. The follow up work by Sun et al. in [136] focusses on translating the mined rules into checkers for a commercial static analysis tool. While her work in [135] looks to integrate a supervised learning into the rule and violation review process.

Nguyen et al. describe a related approach to the Chang system in [109]. Instead of mining detailed dependency patterns however their system looks to mine temporal object interactions. The goal is an extension of Ammons et al. where instead of looking a single objects the authors attempt to mine multi-object interactions in Java. The authors begin by constructing a graphical representation with two types of nodes, method calls and control nodes. Edges are added to represent data dependencies and control flow. They then use a frequent subgraph mining algorithm to identify candidate rules. In comparison too the Change approach there is less potential power since the representation is simpler. However, less detail also means less noise which potentially means fewer false positives.

7.5.2 Mining Error Handling Specifications

In comparison to the broader field of specification inference there has been little work on error and exception handling specifications. Many papers specifically ignore error handling routines in an attempt to reduce noise. However, two groups, Necula's and Xie's, have investigated several different approachs.

In 2004 Weimer and Necula investigated the prevelance of semantic violations in exception handling code. The chose objects which had specifications readily available and created a tool to find violations of the known semantics. The tool used context sensitive intra-procedural data flow analysis to check for errors and looked specifically at aquire/release APIs in the Java Platform API. The authors found many programs contained errors around these resource management APIs and concluded that new language tools are needed for resource management.

Weimer and Necula followed up their 2004 work in 2005 with a system for mining temporal specifications based on Engler's 2001 work [40, 146]. The author's main conlusion from the 2004 work was, "Client code frequently violates API specifications in exceptional situations." Furthermore, it violates the specifications becase it is hard to write correct exception handlers with current languages idioms, especially in the case of nested resources.

The system tries to mine resource management specifications by looking for sequencing rules of the form, $(a \rightarrow b)*$. Where a and b must be methods operating on the same objects and $(a \rightsquigarrow b)$ must travel through a exception at least once. Additionally, the authors add the constraint that there must be one exceptional path from a where there is no b. The system then uses the Engler dataflow framework to compute the "belief" score whether an object obeys the constraints. Finally, it uses a statistical ranking procedure to select the objects with a highest likelihood of obeying the resource management specification.

Thummalapenta and Xie presented an alternative approach in [142] which addresses some of the short comings of the Weimer system. The authors note that while the Weimer system fails to mine rules correctly when the exception handling code must be different than the nominal code. They present the case of a database query, consider the following Java code

```
1  try {
2      conn = getConnection();
3      statement = conn.createStatement();
4      statement.executeUpdate("DELETE␣FROM␣table1␣where␣id␣=␣?", id);
5      conn.commit();
6  } catch (ConnectionError ce} {
7      log.Error("no␣connection␣bailing")
8  } catch (SQLException se} {
9      if (conn != null) {
10         conn.rollback();
11     }
12 } finally {
13     if (statement != null) { statement.close(); }
14     if (conn != null) { returnConnection(conn); }
15 }
```

In this complex block there are several cases which need to be handled correctly. The most important one is when a 'SQLException' occurs a rollback must be issued while in the nominal case a commit should be issued. To fix this problem the authors mine sequence association rules of the form:

$$c_1 \rightsquigarrow c_2 \ldots c_n \rightsquigarrow a \rightsquigarrow e_1 \rightsquigarrow e_2 \ldots e_m$$

where a is the of method call of interest, the c_is are proceeding method calls, and the e_js are method calls in the exceptional path.

To mine such complex rules the system uses frequent sub-sequence mining on generated static traces. It first collects all the call sites of interest. If there is not enough data for a particular method or object it uses, the now defunct, Google Code Search to find more code which uses the library of interest. It then constructs the Control Flow Graph and the Exception Flow Graph and generates static traces for both normal and exceptional paths. The traces are filtered to remove unrelated noise to reduce noise. Finally closed frequent sub-sequence mining is applied to find the patterns. This system has the advantage that the general approach could be used with dynamic traces as well as the synthetic ones it generates.

Finally, Buse and Weimer proposed a system to mine documentation instead of specifications in [20]. Instead of mining how to handle exceptions they mine what exceptions can occur and what conditions can lead to there occurrence. For example the system can generate documentation like

```
Exception NotFound raised when id is null or id.equals("")
```

The system works by computing a call graph where each method is annotated with which exceptions it can throw (transitively). The authors then employ a data-flow work list algorithm to compute a predicate for each method exception pair. The predicate gives the condition for when the exception will be thrown. The predicate is post processed and algebraically simplified. Finally, they generate English like sentences using the predicate and the exception type.

Appendices

Appendix A

Markov Chains

A.1 Absorbing Chains

This section follows Grinstead and Snell's book [54] to explain the computation of absorption times. The theory absorbing Markov chains is not new and can be found (in several presentations) in volumes 1 and 2 of Feller's classic text on probability theory [44, 45]. Kemeny and Snell first presented the formulation used here in their 1960 book *Finite Markov Chains* [80] (which was republished in 1976 by Springer). The Grinstead book follows the Kemeny book with improved explanations.

A state s_i in a Markov chain $(S, \mathbf{P})$ is called absorbing if it is impossible to leave it. That is, $\mathbf{P}_{i,i} = 1$ and for all $j \neq i$ the $\mathbf{P}_{i,j} = 0$. A Markov chain is absorbing if it has at least 1 absorbing state and it is possible to go from every state to an absorbing state in 1 or more transitions. In an absorbing Markov chain states which are not absorbing are called transient. Let the set of absorbing states be A and the set of transient states be T. Absorbing Markov chains have a *canonical form* where the transition matrix $\mathbf{P}$ is arranged such that the transient states come before the absorbing states:

$$\mathbf{P} = \begin{array}{c} \\ \text{TR.} \\ \text{ABS.} \end{array} \begin{array}{c} \begin{array}{cc} \text{TR.} & \text{ABS.} \end{array} \\ \begin{bmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \end{array} \tag{A.1}$$

The probability (for any Markov chain not just absorbing) of moving from state s_i to state s_j in t steps is $\Pr\left[s_i \xrightarrow{t} s_j\right] = \mathbf{P}^t_{i,j}$. The notation $\mathbf{P}^t$ means the t^{th} power of the matrix $\mathbf{P}$. This equation makes sense because if there is a row vector $\mathbf{u}$ which represents a probability distribution over the states S it can be multiplied by $\mathbf{P}$ to produce a new distribution after 1 step of the Markov process: $\mathbf{u} \cdot \mathbf{P} = \mathbf{u}'$. Multiplying $\mathbf{u}'$ by $\mathbf{P}$ again gives another step in the process: $\mathbf{u}' \cdot \mathbf{P} = \mathbf{u}''$. Which can be written as: $\mathbf{u} \cdot \mathbf{P} \cdot \mathbf{P}$ or $\mathbf{u} \cdot \mathbf{P}^2$. Thus taking t steps from an initial distribution $\mathbf{u}$ is: $\mathbf{u} \cdot \mathbf{P}^t$.

In the canonical form (Equation A.1) the sub-matrix $\mathbf{Q}$ contains the probability from transitioning from a transient state to a transient state. Given transient states s_i and s_j the probability of going from state s_i to state s_j in t steps is: $\mathbf{P}^t_{i,j} = \mathbf{Q}^t_{i,j}$. For any transient state s_i the total probability of arriving in a transient state after 1 step is: $\mathbf{Q}_{i,\cdot} = \sum_{s_j \in T} \mathbf{Q}_{i,j}$. There must be at least one transient state s_i which reaches an absorbing state in 1 step.

For such a state $\mathbf{Q}_{i,\cdot} < 0$. Since there is one row of $\mathbf{Q}$ which does not sum to 1 as $t \to \infty$ $\mathbf{Q}^t \to \mathbf{0}$.

Using the fact that $\mathbf{Q}^t \to \mathbf{0}$ as $t \to \infty$ a special matrix, *the fundamental matrix of absorbing Markov chains*, can be constructed. The fundamental matrix $\mathbf{N}$ is:

$$\mathbf{N} = \mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \mathbf{Q}^3 + ... \tag{A.2}$$

The matrix $\mathbf{N}$ exists if the series $\mathbf{I} + \mathbf{Q}^1 + \mathbf{Q}^2 + ...$ converges. The series does converge because $\mathbf{Q}^t \to \mathbf{0}$.

Theorem A.1. *Each entry $\mathbf{N}_{i,j}$ corresponds to the expected number of times the chain visits transient state s_j given that it started from transient state s_i.*

Proof. Recall, a Markov process can be viewed as a sequence of random variables $X^{(0)}, X^{(1)}, ...$ where

$$\begin{aligned} & \Pr\left[X^{(t+1)} = y \mid X^{(0)} = x_0, X^{(1)} = x_1, ..., X^{(t)} = x\right] \\ = \ & \Pr\left[X^{(t+1)} = y \mid X^{(t)} = x\right] \\ = \ & \mathbf{P}_{x,y} \end{aligned}$$

The probability of starting in s_i and ending in s_j after t steps is:

$$\begin{aligned} \Pr\left[s_i \xrightarrow{t} s_j\right] &= \mathbf{P}^t_{i,j} \\ &= \Pr\left[X^{(0)} = s_i \cap X^{(t)} = s_j\right] \end{aligned}$$

Now following [54], the event $(X^{(0)} = s_i \cap X^{(t)} = s_j)$ can be viewed as a binary random variable $Y^{(t)}_{i,j}$ which equals 1 when the event $(X^{(0)} = s_i \cap X^{(t)} = s_j)$ occurs and 0 otherwise. The expected value of $Y^{(t)}_{i,j}$ is:

$$\begin{aligned} \mathrm{E}\left[Y^{(t)}_{i,j}\right] &= \sum_{x \in \{0,1\}} x \cdot \Pr\left[Y^{(t)}_{i,j} = x\right] \\ &= \Pr\left[Y^{(t)}_{i,j} = 1\right] \\ &= \mathbf{P}^t_{i,j} \end{aligned}$$

The expected number of times the process is in s_j on the when $t = 1$ step is therefore:

$$\mathrm{E}\left[Y^{(1)}_{i,j}\right] = \mathbf{P}^1_{i,j}$$

This expectation makes sense because the Markov process can only be in one state at a time and number of times the process can visit a state at a particular t is at most 1. Thus, the expected number of times the process visits s_j for $t \leq n$ is:

$$\begin{aligned} & \mathrm{E}\left[Y^{(0)}_{i,j} + Y^{(1)}_{i,j} + ... + Y^{(n)}_{i,j}\right] \\ = \ & \mathrm{E}\left[Y^{(0)}_{i,j}\right] + \mathrm{E}\left[Y^{(1)}_{i,j}\right] + ... + \mathrm{E}\left[Y^{(n)}_{i,j}\right] \\ = \ & \mathbf{P}^0_{i,j} + \mathbf{P}^1_{i,j} + ... + \mathbf{P}^n_{i,j} \\ = \ & \mathbf{I}_{i,j} + \mathbf{P}^1_{i,j} + ... + \mathbf{P}^n_{i,j} \end{aligned}$$

Since both states s_i and s_j are transient

$$\mathbf{I}_{i,j} + \mathbf{P}^1_{i,j} + ... + \mathbf{P}^n_{i,j} = \mathbf{I}_{i,j} + \mathbf{Q}^1_{i,j} + ... + \mathbf{Q}^n_{i,j}$$

Letting n tend towards ∞ gives

$$\begin{aligned} \mathrm{E}\left[Y^{(0)}_{i,j} + Y^{(1)}_{i,j} + Y^{(2)}_{i,j} + ...\right] &= \mathbf{I}_{i,j} + \mathbf{Q}^1_{i,j} + \mathbf{Q}^2_{i,j} + ... \\ &= \mathbf{N}_{i,j} \end{aligned}$$

Thus, $\mathbf{N}_{i,j}$ is the expected number of visits to transient state s_j for a process starting in transient state s_i. □

Theorem A.2. *The matrix* $\mathbf{N}$ *can be computed by taking the inverse of* $(\mathbf{I} - \mathbf{Q})$*:*

$$\mathbf{N} = \mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \mathbf{Q}^3 + ... = (\mathbf{I} - \mathbf{Q})^{-1} \tag{A.3}$$

Proof. Following [54], the inverse of $\mathbf{I} - \mathbf{Q}$ exists:

$$\begin{aligned} \text{Let } (\mathbf{I} - \mathbf{Q})\mathbf{x} &= \mathbf{0} \\ \mathbf{I}\mathbf{x} - \mathbf{Q}\mathbf{x} &= \mathbf{0} \\ \mathbf{x} &= \mathbf{Q}\mathbf{x} \end{aligned}$$

Iterating on the last form gives:

$$\mathbf{x}^n = \mathbf{Q}^n\mathbf{x}^n$$

Since

$$\begin{aligned} \lim_{n\to\infty} \mathbf{Q}^n &= \mathbf{0} \\ \mathbf{x}^n &= \mathbf{0} \\ \mathbf{x} &= \mathbf{0} \end{aligned}$$

By the invertible matrix theorem $(\mathbf{I} - \mathbf{Q})^{-1}$ exists since the equation $(\mathbf{I} - \mathbf{Q})\mathbf{x} = \mathbf{0}$ only has the trivial solution $\mathbf{x} = \mathbf{0}$. Now that we know $(\mathbf{I} - \mathbf{Q})^{-1}$ exists we must prove it converges to $\mathbf{N}$ as defined Equation A.2. Begin by multiplying $(\mathbf{I} - \mathbf{Q})$ by $\mathbf{N}$:

$$(\mathbf{I} - \mathbf{Q})\mathbf{N} = (\mathbf{I} - \mathbf{Q})(\mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \mathbf{Q}^3 + ...)$$

For the moment, bound the sequence by some finite n

$$(\mathbf{I} - \mathbf{Q})(\mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \mathbf{Q}^3 + ... + \mathbf{Q}^n)$$

Expanding gives

$$\mathbf{I}^2 - \mathbf{Q} + \mathbf{Q} - \mathbf{Q}^2 + \mathbf{Q}^2 - \mathbf{Q}^3 + \mathbf{Q}^3 \ ... - \mathbf{Q}^n + \mathbf{Q}^n - \mathbf{Q}^{n+1}$$

Thus

$$(\mathbf{I} - \mathbf{Q})(\mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \mathbf{Q}^3 + ... + \mathbf{Q}^n) = \mathbf{I} - \mathbf{Q}^{n+1}$$

Multiply both sides by $\mathbf{N}$.

$$\begin{aligned}\mathbf{N}(\mathbf{I}-\mathbf{Q})(\mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+...+\mathbf{Q}^n) &= \mathbf{N}(\mathbf{I}-\mathbf{Q}^{n+1})\\ (\mathbf{I}-\mathbf{Q})^{-1}(\mathbf{I}-\mathbf{Q})(\mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+...+\mathbf{Q}^n) &= \mathbf{N}(\mathbf{I}-\mathbf{Q}^{n+1})\\ \mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+...+\mathbf{Q}^n &= \mathbf{N}(\mathbf{I}-\mathbf{Q}^{n+1})\end{aligned}$$

Now let n approach ∞

$$\begin{aligned}\mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+... &= \mathbf{N}(\mathbf{I}-\mathbf{Q}^{n+1})\\ \mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+... &= \mathbf{N}(\mathbf{I}-\mathbf{0})\\ \mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+... &= \mathbf{N}\mathbf{I}\\ \mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+... &= \mathbf{N}\\ \mathbf{N} &= \mathbf{I}+\mathbf{Q}+\mathbf{Q}^2+\mathbf{Q}^3+...\end{aligned}$$

Thus, $\mathbf{N}$ exists and is equal to the converged power series. □

Theorem A.3. *Let τ_i be the expected number of steps starting at state s_i before the Markov process reaches an absorbing state*

$$\tau_i = \sum_{s_j \in T} \mathbf{N}_{i,j}$$

Proof. Using the definitions from Theorem A.1:

$$\begin{aligned}\tau_i &= \sum_{s_j \in T} \mathbf{N}_{i,j}\\ &= \sum_{s_j \in T} \mathrm{E}\left[Y_{i,j}^{(0)} + Y_{i,j}^{(1)} + Y_{i,j}^{(2)} + ...\right]\\ &= \sum_{s_j \in T} \text{expected visits to } s_j \text{ for a process starting in } s_i\end{aligned}$$

□

Appendix B

Debugging

B.1 Traditional Debugging

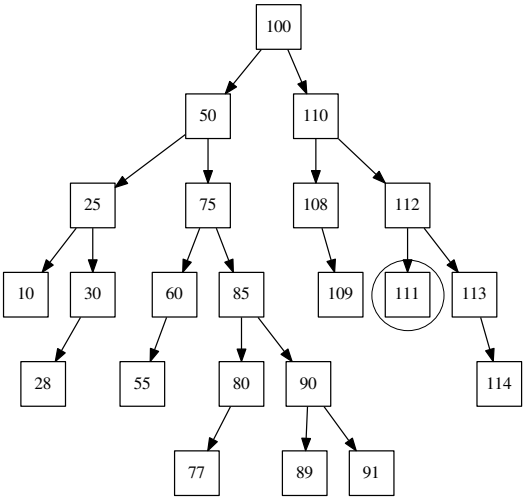
In Automatic Fault Localization the imagined scenario involves a programmer using a ranked list of locations in a program to find the fault. The programmer checks each location and then moves down to the next location. As noted this process bears little resemblance to a debugging session a programmer would actually conduct. The traditional debugging process which is taught and widely practiced is the *Scientific Debugging Process*.

Definition B.1 (Scientific Debugging Process)**.**

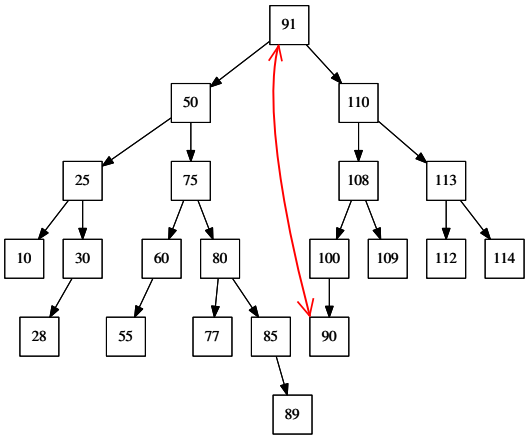
1. *Collect information about program failure (eg. from user reports or from running the program).*
2. *Reproduce the failure.*
   (a) *Form a hypothesis for what is causing failure.*
   (b) *Create a test or program input from the hypothesis.*
   (c) *Run the test and observe the program's behavior.*
   (d) *If the program did not fail, go back to step 2.(a).*
3. *Form a hypothesis for why the program failed. Hypothesis formation may involve running the program, inspecting the code, or other activities.*
4. *(optional) Construct an experiment using a debugging tool (`printf`, a debugger such as `gdb`, or a graphical tool such as Visual Studio or IntelliJ) to test your failure hypothesis.*
5. *(optional) If the hypothesis was rejected go back to step 3.*
6. *Construct a fix based on the hypothesis.*
7. *Run the fixed program and attempt to cause it fail.*
8. *If the fixed program failed go back to step 1 (potentially reverting the fix).*
9. *Congratulations the fault has been located and fixed.*

```
1     func (n *Node) balance() *Node {
2         if n == nil {
3             return nil
4         }
5         if abs(n.left.Height() - n.right.Height()) < maxSep {
6             return n
7         } else if n.left.Height() < n.right.Height() {
8             return n.rotateLeft()
9         } else {
10            return n.rotateRight()
11        }
12    }
13
14    func (n *Node) rotateRight() *Node {
15        if n == nil {
16            return nil
17        } else if n.left == nil {
18            return n
19        }
20        r := n.left.rightmostDescendent()
21++      n.left = n.left.Remove(r.Key)
22        return n.doRotate(r)
23    }
24
25    func (n *Node) rotateLeft() *Node {
26        if n == nil {
27            return nil
28        } else if n.right == nil {
29            return n
30        }
31        r := n.right.leftmostDescendent()
32++      n.right = n.right.Remove(r.Key)
33        return n.doRotate(r)
34    }
35
36    func (n *Node) doRotate(r *Node) *Node {
37--      n = n.Remove(r.Key)
38        r.left = n.left
39        r.right = n.right
40        return r.Put(n.Key, n.Value)
41    }
42
43    func (n *Node) rightmostDescendent() *Node {
44        if n == nil {
45            return nil
46        } else if n.right == nil {
47            return n
48        } else {
49            return n.right.rightmostDescendent()
50        }
51    }
52
53    func (n *Node) leftmostDescendent() *Node {
54        if n == nil {
55            return nil
56        } else if n.left == nil {
57            return n
58        } else {
59            return n.left.leftmostDescendent()
60        }
61    }
```

Listing B.1: This `diff` shows the change required to fix a buggy AVL tree implementation. The buggy statement is was on line 37. Calling the `Remove` function on the root of the subtree could (under the right circumstances) cause another rebalancing operation to occur during a rebalance operation. This (under the right circumstances) would cause the wrong key to be rotated putting violating the ordering property of a binary search tree.



(a) An AVL Tree before node 111 is removed.



(b) An AVL Tree after node 111 is removed. Nodes 90 and 91 are out of position violating the ordering property of a binary search tree.

Figure B.1: AVL Trees demonstrating the bug in Listing B.1.

In scientific debugging the fault is not truly "localized" until a fix can be constructed for the fault. The fix explains the fault and may not be in just one location. Automatic Fault Localization could help the programmer by assisting in "Hypothesis Formation" in Step 3. A good localizer would help the programmer form a good hypothesis quickly.

Listing B.1 shows a bug in an AVL tree which is very difficult to find with known CBSFL techniques. All CBSFL metrics tried on this bug produced equal rankings of all of statements in the AVL tree. Why would all of the statements be ranked the same? Because, the bug in the tree only manifests when the tree is rebalanced with a subtree in a certain (and unusual configuration). This means, there are plenty of both passing and failing tests which cover of all statements in the program.

The hypothesis which needs to be formed in order to debug the program is the fault gets triggered during a rebalance of the tree. The exact situation is illustrated in Figure B.1. In the figure, when the node 111 gets removed a local rebalance at node 112 occurs. This causes the right subtree of the root node (100) to become out of balance with the left subtree. A rebalance occurs to move node 100 into the right subtree and replace it with node 91. However, to remove node 91 a call to node 100's `Remove` method is made on line 37 in Listing B.1. This causes another rebalance to occur which moves node 100 and replaces it with node 90. Thus, when the original rebalance finishes node 90 is now the root node. Node 90 is moved into the right subtree and node 91 becomes the root. The rebalance is now finished but nodes 90 and 91 are out of binary search tree order.

CBSFL techniques cannot localize the bug in Listing B.1. The information needed to localize the fault is not in the coverage profile. For a statistical technique to detect a difference between executions of the faulty AVL tree which exhibit failure and those which do not more information is needed. The information collected would need to include some information about the structure of the memory when the `balance` method is invoked. For instance, if a call tree was collected the tree would need to include the identity of the receiver for the `balance` method. Without collecting the information that `balance` is invoked on a node while a balance was already occurring for the node a statistical fault localizer

would be unable to detect any difference in behavior between failing and passing tests.

Automatic localization of bugs like the one in Listing B.1 is a aspirational goal. The bug demonstrates that the most advanced techniques available today (including the ones outlined in this chapter) can be tripped up by "simple" 1 line bugs. Meanwhile, the scientific debugging process is general and can (with time) help a programmer find and fix any bug. The ambition for all debugging tools should be to augment the programmer during their own personal debugging process not to replace the process with an artificial one.

Bibliography

[1] Abreu, R., Zoeteweij, P., and Van Gemund, A. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)* (2006), IEEE, pp. 39–46.

[2] Acharya, M., Xie, T., Pei, J., and Xu, J. Mining API patterns as partial orders from source code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07* (New York, New York, USA, 2007), ACM Press, p. 25.

[3] Aggarwal, C. C., Bhuiyan, M. A., and Hasan, M. A. Frequent Pattern Mining Algorithms: A Survey. In *Frequent Pattern Mining*. Springer International Publishing, Cham, 2014, pp. 19–64.

[4] Agrawal, R., Imieliński, T., and Swami, A. Mining association rules between sets of items in large databases. *ACM SIGMOD Record 22*, 2 (jun 1993), 207–216.

[5] Agrawal, R., and Srikant, R. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1994), VLDB '94, Morgan Kaufmann Publishers Inc., pp. 487–499.

[6] Aho, A., Sethi, R., Lam, M. S., and Ullman, J. D. *Compilers: principles, techniques, and tools*. 2007.

[7] Al Hasan, M., and Zaki, M. *Musk: Uniform Sampling of k-Maximal Patterns*. Society for Industrial and Applied Mathematics, Philadelphia, PA, apr 2009, ch. 55, pp. 650–661.

[8] Al Hasan, M., and Zaki, M. J. Output Space Sampling for Graph Patterns. *Proc. VLDB Endow. 2*, 1 (aug 2009), 730–741.

[9] Alur, R., Cerny, P., Madhusudan, P., and Nam, W. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages - POPL '05* (New York, New York, USA, 2005), ACM Press, pp. 98–109.

[10] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. *ACM SIGPLAN Notices 37*, 1 (jan 2002), 4–16.

[11] Arora, A., Sachan, M., and Bhattacharya, A. Mining statistically significant connected subgraphs in vertex labeled graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14* (New York, New York, USA, 2014), SIGMOD '14, ACM Press, pp. 1003–1014.

[12] Baah, G. G. K., Podgurski, A., and Harrold, M. J. M. Causal inference for statistical fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis* (New York, NY, USA, 2010), ISSTA '10, ACM, pp. 73–84.

[13] Baah, G. K., Podgurski, A., and Harrold, M. J. Mitigating the Confounding Effects of Program Dependences for Effective Fault Localization. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (New York, NY, USA, 2011), ESEC/FSE '11, ACM, pp. 146–156.

[14] Babai, L. Graph Isomorphism in Quasipolynomial Time. Tech. rep., http://arxiv.org/abs/1512.03547, dec 2015.

[15] Ball, T., and Larus, J. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29* (1996), IEEE Comput. Soc. Press, pp. 46–57.

[16] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering 33*, 9 (sep 2007), 577–591.

[17] Bhadange, S., Arora, A., and Bhattacharya, A. GARUDA: A System for Large-scale Mining of Statistically Significant Connected Subgraphs. *Proc. VLDB Endow. 9*, 13 (sep 2016), 1449–1452.

[18] Borgelt, C., and Berthold, M. R. Mining molecular fragments: finding relevant substructures of molecules. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on* (2002), pp. 51–58.

[19] Bringmann, B., and Nijssen, S. What is frequent in a single graph? In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2008), vol. 5012 LNAI, pp. 858–863.

[20] Buse, R. P., and Weimer, W. R. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis - ISSTA '08* (New York, New York, USA, 2008), ACM Press, p. 273.

[21] Chang, R.-y., and Podgurski, A. Discovering programming rules and violations by mining interprocedural dependences. *Journal of Software: Evolution and Process 24*, 1 (jan 2012), 51–66.

[22] Chang, R.-Y., Podgurski, A., and Yang, J. Discovering Neglected Conditions in Software by Mining Dependence Graphs. *IEEE Transactions on Software Engineering 34*, 5 (sep 2008), 579–596.

[23] Chaoji, V., Al Hasan, M., Salem, S., Besson, J., and J. Zaki, M. ORIGAMI: A Novel and Effective Approach for Mining Representative Orthogonal Graph Patterns. *Stat. Anal. Data Min. 1*, 2 (jun 2008), 67–84.

[24] Cheng, H., Lo, D., Zhou, Y., Wang, X., and Yan, X. Identifying Bug Signatures Using Discriminative Graph Mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (New York, NY, USA, 2009), ISSTA '09, ACM, pp. 141–152.

[25] Cheng, H., Yan, X., and Han, J. Mining Graph Patterns. In *Frequent Pattern Mining*. Springer International Publishing, Cham, 2014, pp. 307–338.

[26] Chilimbi, T. M., Liblit, B., Mehra, K., Nori, A. V., and Vaswani, K. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering* (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 34–44.

[27] Cook, D. J., and Holder, L. B. Substructure Discovery Using Minimum Description Length and Background Knowledge. *J. Artif. Int. Res. 1*, 1 (feb 1994), 231–255.

[28] Cook, J. E., and Wolf, A. L. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology 7*, 3 (jul 1998), 215–249.

[29] Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71* (New York, New York, USA, 1971), ACM Press, pp. 151–158.

[30] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (oct 1991), 451–490.

[31] Dallmeier, V., Knopp, N., Mallon, C., Hack, S., and Zeller, A. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10* (New York, New York, USA, 2010), no. Section 4, ACM Press, p. 85.

[32] Dallmeier, V., Lindig, C., Wasylkowski, A., and Zeller, A. Mining Object Behavior with ADABU. 17–23.

[33] Davis, T. A. Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *ACM Trans. Math. Softw. 30*, 2 (jun 2004), 196–199.

[34] Di Fatta, G., Leue, S., and Stegantova, E. Discriminative Pattern Mining in Software Fault Detection. In *Proceedings of the 3rd International Workshop on Software Quality Assurance* (New York, NY, USA, 2006), SOQUA '06, ACM, pp. 62–69.

[35] Eichinger, F., Böhm, K., and Huber, M. *Mining Edge-Weighted Call Graphs to Localise Software Bugs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 333–348.

[36] Eichinger, F., Krogmann, K., Klug, R., and Böhm, K. Software-defect Localisation by Mining Dataflow-enabled Call Graphs. In *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part I* (Berlin, Heidelberg, 2010), ECML PKDD'10, Springer-Verlag, pp. 425–441.

[37] Eichinger, F., Ossner, C., and Böhm, K. Scalable software-defect localisation by hierarchical mining of dynamic call graphs. *Proceedings of the 11th SIAM International Conference on Data Mining, SDM 2011*, c (2011), 723–734.

[38] El-Ramly, M., Stroulia, E., and Sorenson, P. Interaction Pattern Mining: From run-time behavior to usage scenarios. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02* (New York, New York, USA, 2002), ACM Press, p. 315.

[39] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. In *Proceeding of the VLDB Endowment* (2014), vol. 7, pp. 517–528.

[40] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *ACM SIGOPS Operating Systems Review 35*, 5 (dec 2001), 57.

[41] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering - ICSE '99* (New York, New York, USA, 1999), ACM Press, pp. 213–224.

[42] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1-3 (dec 2007), 35–45.

[43] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. A Density-based Algorithm for Discovering Clusters a Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (1996), KDD'96, AAAI Press, pp. 226–231.

[44] Feller, W. *An Introduction to Probability Theory and Its Applications*, second ed., vol. 2. John Wiley & Sons, New York, 1965.

[45] Feller, W. *An Introduction to Probability Theory and Its Applications*, third ed., vol. 1. John Wiley & Sons, New York, 1968.

[46] Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization, jul 1987.

[47] Flanagan, C., and Leino, K. R. M. Houdini, an annotation assistant for ESC/-Java. In *World Congress on Formal Methods* (2001), pp. 500–517.

[48] Gabel, M., Jiang, L., and Su, Z. Scalable Detection of Semantic Clones. In *Proceedings of the 30th International Conference on Software Engineering* (New York, NY, USA, 2008), ICSE '08, ACM, pp. 321–330.

[49] Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. Optimizing Java Bytecode Using the Soot Framework : Is It Feasible ? *Lecture Notes in Computer Science 1781* (2000), 18–34.

[50] Godefroid, P. DART : Directed Automated Random Testing. *Science* (2005).

[51] Godefroid, P. Higher-order test generation. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (2011), ACM, pp. 258–269.

[52] Godefroid, P., Kiezun, A., and Levin, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08* (New York, New York, USA, 2008), ACM Press, p. 206.

[53] Godefroid, P., Levin, M., Molnar, D., and Others. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium* (2008), Citeseer.

[54] Grinstead, C. M., and Snell, J. L. *Introduction to Probability*, 2 ed. American Mathematical Society, Providence, RI, 1997.

[55] Han, J., Pei, J., and Yin, Y. Mining Frequent Patterns Without Candidate Generation. *SIGMOD Rec. 29*, 2 (may 2000), 1–12.

[56] He, H., and Singh, A. GraphRank: Statistical Modeling and Mining of Significant Subgraphs in the Feature Space. In *Sixth International Conference on Data Mining (ICDM'06)* (dec 2006), IEEE, pp. 885–890.

[57] He, H., and Singh, A. K. Efficient Algorithms for Mining Significant Substructures in Graphs with Quality Guarantees. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)* (oct 2007), IEEE, pp. 163–172.

[58] Hellal, A., and Romdhane, L. B. NODAR: Mining Globally Distributed Substructures from a Single Labeled Graph. *J. Intell. Inf. Syst. 40*, 1 (feb 2013), 1–15.

[59] Henderson, T. A. D., and Podgurski, A. Sampling Code Clones from Program Dependence Graphs with GRAPLE. In *International Workshop on Software Analytics* (2016), ACM.

[60] Henderson, T. A. D., and Podgurski, A. Rethinking Dependence Clones. In *International Workshop on Software Clones* (2017), IEEE.

[61] Higo, Y., and Kusumoto, S. Code Clone Detection on Specialized PDGs with Heuristics. In *2011 15th European Conference on Software Maintenance and Reengineering* (mar 2011), IEEE, pp. 75–84.

[62] Higo, Y., Yasushi, U., Nishino, M., and Kusumoto, S. Incremental Code Clone Detection: A PDG-based Approach. In *2011 18th Working Conference on Reverse Engineering* (oct 2011), IEEE, pp. 3–12.

[63] Horvitz, D. G., and Thompson, D. J. A Generalization of Sampling Without Replacement From a Finite Universe. *Journal of the American Statistical Association 47*, 260 (1952), pp. 663–685.

[64] Horwitz, S. Identifying the Semantic and Textual Differences Between Two Versions of a Program. *SIGPLAN Not. 25*, 6 (jun 1990), 234–245.

[65] Horwitz, S., Prins, J., and Reps, T. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 146–157.

[66] Huan, J., Wang, W., and Prins, J. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. In *Proceedings of the Third IEEE International Conference on Data Mining* (Washington, DC, USA, 2003), ICDM '03, IEEE Computer Society, pp. 549—-.

[67] Huan, J., Wang, W., Prins, J., and Yang, J. SPIN: Mining Maximal Frequent Subgraphs from Graph Databases. In *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '04* (New York, New York, USA, 2004), no. 1, ACM Press, p. 581.

[68] Hübler, C., Kriegel, H. P., Borgwardt, K., and Ghahramani, Z. Metropolis algorithms for representative subgraph sampling. In *Proceedings - IEEE International Conference on Data Mining, ICDM* (2008), no. 1, pp. 283–292.

[69] Hummel, B., Juergens, E., Heinemann, L., and Conradt, M. Index-based code clone detection: Incremental, distributed, scalable. In *IEEE International Conference on Software Maintenance, ICSM* (2010).

[70] Hummel, B., Juergens, E., and Steidl, D. Index-based Model Clone Detection. In *Proceedings of the 5th International Workshop on Software Clones* (New York, NY, USA, 2011), IWSC '11, ACM, pp. 21–27.

[71] Inokuchi, A., Washio, T., and Motoda, H. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *Principles of Data Mining and Knowledge Discovery - PKDD*, D. A. Zighed, J. Komorowski, and J. Żytkow, Eds., vol. 1910 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, jul 2000, pp. 13–23.

[72] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings - International Conference on Software Engineering* (2007), no. 0520320, pp. 96–105.

[73] Jiang, X., Xiong, H., Wang, C., and Tan, A.-H. Mining Globally Distributed Frequent Subgraphs in a Single Labeled Graph. *Data Knowl. Eng. 68*, 10 (oct 2009), 1034–1058.

[74] Jones, J. Fault localization using visualization of test information. In *Proceedings. 26th International Conference on Software Engineering* (2004), vol. 1, IEEE Comput. Soc, pp. 54–56.

[75] Jones, J., Harrold, M., and Stasko, J. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002* (2002).

[76] Jones, J. A., and Harrold, M. J. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2005), ASE '05, ACM, pp. 273–282.

[77] Junttila, T., and Kaski, P. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *In Ninth Workshop on Algorithm Engineering and Experiments* (Philadelphia, PA, jan 2007), D. Applegate and G. Stølting Brodal, Eds., Society for Industrial and Applied Mathematics, pp. 135–149.

[78] Kagdi, H., Collard, M. L., and Maletic, J. I. Comparing Approaches to Mining Source Code for Call-Usage Patterns. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)* (may 2007), IEEE, pp. 20–20.

[79] Kamiya, T., Kusumoto, S., and Inoue, K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering 28*, 7 (jul 2002), 654–670.

[80] Kemeny, J. G., and Snell, J. L. *Finite Markov Chains*, first ed. Van Nostrand, Princeton, NJ, 1960.

[81] Kim, S., Song, I., and Lee, Y. J. An Edge-based Framework for Fast Subgraph Matching in a Large Graph. In *Proceedings of the 16th International Conference on Database Systems for Advanced Applications - Volume Part I* (Berlin, Heidelberg, 2011), DASFAA'11, Springer-Verlag, pp. 404–417.

[82] Kochhar, P. S., Xia, X., Lo, D., and Li, S. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016* (New York, New York, USA, 2016), ACM Press, pp. 165–176.

[83] Komondoor, R., and Horwitz, S. Using Slicing to Identify Duplication in Source Code. *Lecture Notes In Computer Science 2126* (2001), 40–56.

[84] Kremenek, T., Twohey, P., Back, G., Ng, A., and Engler, D. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Seattle, Washington, 2006), pp. 161–176.

[85] Krinke, J. Identifying similar code with program dependence graphs. *Proceedings Eighth Working Conference on Reverse Engineering* (2001), 301–309.

[86] Kudo, T., Maeda, E., and Matsumoto, Y. An Application of Boosting to Graph Classification. In *Proceedings of the 17th International Conference on Neural Information Processing Systems* (Cambridge, MA, USA, 2004), NIPS'04, MIT Press, pp. 729–736.

[87] Kuramochi, M., and Karypis, G. Finding Frequent Patterns in a Large Sparse Graph*. *Data Mining and Knowledge Discovery 11*, 3 (nov 2005), 243–271.

[88] Le, T.-D. B., Oentaryo, R. J., and Lo, D. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting*

on Foundations of Software Engineering - ESEC/FSE 2015 (New York, New York, USA, 2015), no. 65, ACM Press, pp. 579–590.

[89] Leskovec, J., and Faloutsos, C. Sampling from Large Graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 631–636.

[90] Levin, D. A., Peres, Y., and Wilmer, E. L. *Markov Chains and Mixing Times*. American Mathematical Society, Providence, RI, 2009.

[91] Li, Z., and Zhou, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13* (New York, New York, USA, 2005), ACM Press, p. 306.

[92] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I. Scalable statistical bug isolation. *ACM SIGPLAN Notices 40*, 6 (jun 2005), 15.

[93] Liu, C., Chen, C., Han, J., and Yu, P. S. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 872–881.

[94] Liu, C., Yu, H., Yu, P. S., Yan, X., Yu, H., Han, J., and Yu, P. S. Mining Behavior Graphs for Backtrace of Noncrashing Bugs. In *Proceedings of the 2005 SIAM International Conference on Data Mining* (2005), Society for Industrial and Applied Mathematics, pp. 286–297.

[95] Livshits, B., and Zimmermann, T. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13* (New York, New York, USA, 2005), ACM Press, p. 296.

[96] Llinares-López, F., Sugiyama, M., Papaxanthos, L., and Borgwardt, K. Fast and Memory-Efficient Significant Pattern Mining via Permutation Testing. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15* (New York, New York, USA, 2015), ACM Press, pp. 725–734.

[97] Lo, D., Cheng, H., Han, J., Khoo, S.-C., and Sun, C. Classification of software behaviors for failure detection. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09* (New York, New York, USA, 2009), ACM Press, p. 557.

[98] Lo, D., and Khoo, S.-c. SMArTIC. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14* (New York, New York, USA, 2006), ACM Press, p. 265.

[99] Lucia, Lo, D., Jiang, L., Thung, F., and Budi, A. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process 26*, 2 (feb 2014), 172–219.

[100] Mariani, L., Pastore, F., and Pezze, M. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Trans. Softw. Eng. 37*, 4 (jul 2011), 486–508.

[101] McKay, B. D., and Piperno, A. Practical graph isomorphism, II. *Journal of Symbolic Computation 60* (jan 2014), 94–112.

[102] Monperrus, M., and Mezini, M. Detecting missing method calls as violations of the majority rule. *ACM Transactions on Software Engineering and Methodology 22*, 1 (feb 2013), 1–25.

[103] Mousavian, Z., Vahidi-Asl, M., and Parsa, S. Scalable Graph Analyzing Approach for Software Fault-localization. In *Proceedings of the 6th International Workshop on Automation of Software Test* (New York, NY, USA, 2011), AST '11, ACM, pp. 15–21.

[104] Muchnick, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[105] Nguyen, A. T., and Nguyen, T. N. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 858–868.

[106] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., and Nguyen, T. N. Graph-based pattern-oriented, context-sensitive source code completion. In *2012 34th International Conference on Software Engineering (ICSE)* (jun 2012), IEEE, pp. 69–79.

[107] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., Tamrawi, A., Nguyen, H. V., Al-Kofahi, J., and Nguyen, T. N. Graph-based Pattern-oriented, Context-sensitive Source Code Completion. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 69–79.

[108] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E* (New York, New York, USA, 2009), ACM Press, p. 383.

[109] Nguyen, T. T., Nguyen, H. A., Pham, N. H., Al-Kofahi, J. M., and Nguyen, T. N. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E* (New York, New York, USA, 2009), ACM Press, p. 383.

[110] Nijssen, S., and Kok, J. N. 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* (New York, NY, USA, 2004), KDD '04, ACM, pp. 647–652.

[111] Parsa, S., Naree, S. A., and Koopaei, N. E. Software Fault Localization via Mining Execution Graphs. In *Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part II* (Berlin, Heidelberg, 2011), ICCSA'11, Springer-Verlag, pp. 610–623.

[112] Pham, N. H., Nguyen, H. A., Nguyen, T. T., Al-Kofahi, J. M., and Nguyen, T. N. Complete and accurate clone detection in graph-based models. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 276–286.

[113] Podgurski, A., and Clarke, L. The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification* (New York, NY, USA, 1989), TAV3, ACM, pp. 168–178.

[114] Podgurski, A., and Clarke, L. A. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Trans. Softw. Eng. 16*, 9 (sep 1990), 965–979.

[115] Pradel, M. Dynamically inferring, refining, and checking API usage protocols. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09* (New York, New York, USA, 2009), ACM Press, p. 773.

[116] Ramanathan, M. K., Grama, A., and Jagannathan, S. Static specification inference using predicate mining. *ACM SIGPLAN Notices 42*, 6 (jun 2007), 123.

[117] Ramanathan, M. K., Sen, K., Grama, A., and Jagannathan, S. Protocol Inference Using Static Path Profiles. In *Proceedings of the 15th international symposium on Static Analysis* (2008), pp. 78–92.

[118] Ranu, S., and Singh, A. K. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *Proceedings of the 2009 IEEE International Conference on Data Engineering* (Washington, DC, USA, 2009), ICDE '09, IEEE Computer Society, pp. 844–855.

[119] Rattan, D., Bhatia, R., and Singh, M. Software clone detection: A systematic review. *Information and Software Technology 55*, 7 (2013), 1165–1199.

[120] Ray, A., Holder, L., and Choudhury, S. Frequent Subgraph Discovery in Large Attributed Streaming Graphs. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014* (2014), W. Fan, A. Bifet, Q. Yang, and P. S. Yu, Eds., vol. 36 of {*JMLR*} *Proceedings*, JMLR.org, pp. 166–181.

[121] Reiss, S., and Renieris, M. Encoding program executions. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001* (2001), 221–230.

[122] Riesen, K., Jiang, X., and Bunke, H. *Exact and Inexact Graph Matching: Methodology and Applications*. Springer US, Boston, MA, 2010, pp. 217–247.

[123] Roy, C. K., and Cordy, J. R. A Survey on Software Clone Detection Research. Tech. rep., Queens University, 2007.

[124] Roy, C. K., Cordy, J. R., and Koschke, R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming 74*, 7 (may 2009), 470–495.

[125] Roy, C. K., Zibran, M. F., and Koschke, R. The vision of software clone management: Past, present, and future. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on* (2014), pp. 18–33.

[126] Saha, T. K., and Hasan, M. A. FS³: A sampling based method for top-k frequent subgraph mining. In *2014 IEEE International Conference on Big Data (Big Data)* (oct 2014), IEEE, pp. 72–79.

[127] Saigo, H., Krämer, N., and Tsuda, K. Partial least squares regression for graph mining. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 08* (New York, New York, USA, 2008), ACM Press, p. 578.

[128] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering* (New York, NY, USA, 2016), ICSE '16, ACM, pp. 1157–1168.

[129] Särndal, C.-E., Swensson, B., and Wretman, J. *Model Assisted Survey Sampling*. Springer-Verlag, 1992.

[130] Schmidt, D. C., and Harrison, T. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. In *Pattern Languages of Program Design 3*, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley, Reading, Mass., 1998, ch. 7, pp. 363–376.

[131] Shoham, S., Yahav, E., Fink, S., and Pistoia, M. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07* (New York, New York, USA, 2007), ACM Press, p. 174.

[132] Shu, G., Sun, B., Henderson, T. A. D., and Podgurski, A. JavaPDG: A New Platform for Program Dependence Analysis. In *IEEE International Conference on Software Testing, Verification and Validation* (mar 2013), pp. 408–415.

[133] Steimann, F., Frenkel, M., and Abreu, R. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013* (2013), 314.

[134] Sugiyama, M., López, F. L., Kasenburg, N., and Borgwardt, K. M. Significant Subgraph Mining with Multiple Testing Correction. In *Proceedings of the 2015 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, Philadelphia, PA, jun 2015, pp. 37–45.

[135] Sun, B., Podgurski, A., and Ray, S. Improving the precision of dependence-based defect mining by supervised learning of rule and violation graphs. In *Software*

Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on (nov 2010), IEEE, pp. 1–10.

[136] Sun, B., Shu, G., Podgurski, A., and Robinson, B. Extending Static Analysis by Mining Project- Specific Rules. *ICSE* (2012).

[137] Sun, C., and Khoo, S.-C. Mining succinct predicated bug signatures. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* (New York, New York, USA, 2013), ACM Press, p. 576.

[138] Sun, S.-F., and Podgurski, A. Properties of Effective Metrics for Coverage-Based Statistical Fault Localization. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)* (apr 2016), IEEE, pp. 124–134.

[139] Talukder, N., and Zaki, M. J. A Distributed Approach for Graph Mining in Massive Networks. *Data Min. Knowl. Discov. 30*, 5 (sep 2016), 1024–1052.

[140] Thoma, M., Cheng, H., Gretton, A., Han, J., Kriegel, H.-P., Smola, A., Song, L., Yu, P. S., Yan, X., and Borgwardt, K. M. Discriminative frequent subgraph mining with optimality guarantees. *Statistical Analysis and Data Mining 3*, 5 (aug 2010), 302–318.

[141] Thompson, S. K. *Sampling*, 2 ed. John Wiley & Sons, New York, 2002.

[142] Thummalapenta, S., and Xie, T. Mining exception-handling rules as sequence association rules. In *2009 IEEE 31st International Conference on Software Engineering* (2009), IEEE, pp. 496–506.

[143] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[144] Toivonen, H. Sampling Large Databases for Association Rules. In *Proceedings of the 22th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1996), VLDB '96, Morgan Kaufmann Publishers Inc., pp. 134–145.

[145] Wasylkowski, A., Zeller, A., and Lindig, C. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07* (New York, New York, USA, 2007), ACM Press, p. 35.

[146] Weimer, W., and Necula, G. C. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for Construction and Analysis of Systems* (2005), Springer Berlin Heidelberg, pp. 461–476.

[147] Whaley, J., Martin, M. C., and Lam, M. S. Automatic extraction of object-oriented component interfaces. *ACM SIGSOFT Software Engineering Notes 27*, 4 (jul 2002), 218.

[148] Wörlein, M., Meinl, T., Fischer, I., and Philippsen, M. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In *9th European Conference on Principles and Practice of Knowledge Discovery in Databases* (Porto, Portugal, 2005), Springer Berlin Heidelberg, pp. 392–403.

[149] Wu, J., Hong, Z., Pan, S., Zhu, X., Cai, Z., and Zhang, C. Multi-graph-view subgraph mining for graph classification. *Knowledge and Information Systems 48*, 1 (jul 2016), 29–54.

[150] Yan, X., Cheng, H., Han, J., and Yu, P. S. Mining Significant Graph Patterns by Leap Search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 433–444.

[151] Yan, X., and Han, J. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* (2002), IEEE Comput. Soc, pp. 721–724.

[152] Yan, X., and Han, J. CloseGraph: Mining Closed Frequent Graph Patterns. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2003), KDD '03, ACM, pp. 286–295.

[153] Yang, J., and Evans, D. Dynamically inferring temporal properties. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '04* (New York, New York, USA, 2004), ACM Press, p. 23.

[154] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. Perracotta: mining temporal API rules from imperfect traces. In *Proceeding of the 28th international conference on Software engineering - ICSE '06* (New York, New York, USA, 2006), ACM Press, p. 282.

[155] Yoo, S., Harman, M., and Clark, D. Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches. *ACM Trans. Softw. Eng. Methodol. 22*, 3 (jul 2013), 19:1—-19:29.

[156] Yousefi, A., and Wassyng, A. A Call Graph Mining and Matching Based Defect Localization Technique. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops* (mar 2013), IEEE, pp. 86–95.

[157] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.

[158] Zhu, F., Qu, Q., Lo, D., Yan, X., Han, J., and Yu, P. S. Mining Top-K Large Structural Patterns in a Massive Network. In *International Conference on Very Large Data Bases* (2011), pp. Vol. 4, No. 11.

[159] Zhu, K., Zhang, Y., Lin, X., Zhu, G., and Wang, W. NOVA: A Novel and Efficient Framework for Finding Subgraph Isomorphism Mappings in Large Graphs. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications - Volume Part I* (Berlin, Heidelberg, 2010), DASFAA'10, Springer-Verlag, pp. 140–154.

[160] Zou, R., and Holder, L. B. Frequent subgraph mining on a single large graph using sampling techniques. *Proceedings of the Eighth Workshop on Mining and Learning with Graphs - MLG '10* (2010), 171–178.