APPLYING GRAPH MINING TECHNIQUES TO SOLVE COMPLEX SOFTWARE ENGINEERING PROBLEMS

A dissertation submitted

to Kent State University in partial

fulfillment of the requirements for the

degree of Doctor of Philosophy

By

Abrar Fawwaz AlAbed-AlHaq

December 2015

Dissertation written by

Abrar Fawwaz AlAbed-AlHaq

B.S., Yarmouk University, Jordan, 2007

M.S., Yarmouk University, Jordan, 2009

Ph.D., Kent State University, 2015

Approved by

_____
Jonathan Maletic, Professor, Ph.D., Computer Science, Doctoral Advisor

_____
Feodor Dragan, Professor, Ph.D., Computer Science

_____
Ruoming Jin, Associate Professor, Ph.D., Computer Science

_____
Michael Collard, Assistant Professor, Ph.D., Computer Science

_____
Antal Jakli, Professor, Ph.D., Chemical Physics

_____
Jay Lee, Professor, Ph.D., Geography

Accepted by

_____
Javed I. Khan, Professor, Ph.D., Chair, Department of Computer Science

_____
James L. Blank, Professor, Ph.D., Dean, College of Arts and Sciences

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Professor *Jonathan Maletic*, for his support and help, always being a great mentor, and allowing me the time and independence in finding my way of reaching the dissertation goal. He taught me how to develop and present ideas, encouraged me when I was stuck, and was always there when I needed his advice and assistance.

My deepest gratitude and sincere love go to my great Father *Professor Fawwaz* and my great Mother *Feryal* for their inspiration, encouragement and advice throughout my early life, whose prayers and wishes have always been the cause of every success I have ever had. My dissertation would not have been possible without great support and patience from my beloved husband, Eng. *Alaa*, and my lovely Son *Omar*. My sincere appreciation goes to my brothers and sisters, Eng. *Mohammed, Dr. AlBaraa, Dr. Thanaa, Suhaib, Sundus and my sister in law Ghiadaa, to them, I offer* my special thanks and love. Special thanks and appreciation are extended to my brother Eng. Mohammed and my sister in law Ruba and the little angle Taleen for supporting me and taking care of me and of my son Omar during my study.

I am deeply grateful for my sister Fatema Nafa and her husband Salem Othman for their support and help. I would like to extend my grateful thanks to my friends *Nahla Abid*, *Reem AlSuhaibani*, *Michael Decker, and Brian Bartman*, who helped and gave me the support to fulfill this dissertation. I am grateful to all my colleagues and friends in

software engineering development laboratory (SDML), Computer Science Department, and Kent State University.

Last but not least, I will never forget Marcy Curtiss, who plays the role of being a mother, a sister, and a mentor, for her my heartfelt appreciation. Finally, I also greatly thank my dissertation committee for their inspiration, insightful efforts, invaluable feedback, contribution, and fruitful recommendations.

Abrar AlAbed-AlHaq

December 2015, Kent, Ohio

CHAPTER 1

INTRODUCTION

There are a large number of research investigations that apply data mining techniques to software repositories [Kagdi, Collard, Maletic 2007]. The huge increase in the number of open source projects in the past 10 years has provided an invaluable wealth of data on real projects [Miltiadis, Charles 2013]. Mining Software Repositories (MSR) is an active research area in software engineering. The objective is to mine existing code repositories for to uncover emergent properties; such as discovering common behaviors. This in turn can then be used to extract specifications such as interfaces usage patterns or discover anomalies; which can then be used to find bugs or problematic behaviors.

Standard data mining techniques, such as sequence mining techniques, have been widely applied to software engineering problems such as association role mining [Michai 2000] to discover library reuse pattern in existing applications by discovering library classes and member functions that are typically reused in combination by application classes. Sequential pattern mining [Xie, Jian 2006] for mining API usages from open source repositories by leveraging existing code search engines and a frequent sequence miner the mining leads to a short list of frequent API usages for developers to inspect. Also, string-matching technique used to discover code fragments that are equal or very similar to solve the clone detection problems, which is effected by increasing source code size and duplication of errors. Using standard data mining techniques allow for the mining of transactional data only and they cannot be used for structural information. We

need new techniques to extract patterns in massive data sets representing complex interactions between entities. Graph pattern mining techniques can be used to capture more interesting relationships than frequent, and sequence mining techniques. As such we feel this is a very good opportunity to conduct novel research to solve software engineering problems using graph pattern mining.

There is very little research related to graph pattern mining in software engineering, we believe this is due to two main issues. First, is that it is difficult to extract large amounts of graph data from software. Second, is to identify a practical software engineering problem can be solved using graph-mining techniques. There is a variety of techniques for graph mining such as frequent subgraph mining, graph/subgraph matching, vertex similarity, dense component discovery, frequent connected subgraph mining, reachability/shortest distance etc. One of the goals of this research is to identify which of these techniques can be practically applied to actual software engineering problems.

1.1. Research Directions

The goal of the research is to apply graph-mining technique to the domain of software engineering in order to solve relevant software engineering problems. In particular the work focuses on using the graph mining technique of *Frequent Subgraph Mining*. Frequent subgraph mining identifies which subgraphs in a given dataset occur most often. The basic idea behind frequent subgraph mining is to "grow" candidate subgraphs, in either a breadth first or depth first manner, to generate candidate patterns and then determine if the identified candidate subgraphs occur frequently enough in the graph data set for them to be considered interesting based on the support count [Chuntao,

Michele 2013]. The work proposed here is to apply frequent subgraph mining to the source code of large-scale software systems. Specifically, the abstract syntax tree of the source code will be mined. This will allow for the discovery of patterns that have both textual and syntactic similarities.

This is a major departure from existing techniques in pattern mining of source code. Current methods and research examines only textual similarities and frequency. Syntactic similarities are not accounted for or considered. One of the software problems that can be addressed here is code completion, which is one of the major design objectives of source code editors to save time, and typing (keystroke) for code writing [Han et al. 2009]. As shown in Figure 1 the research steps.

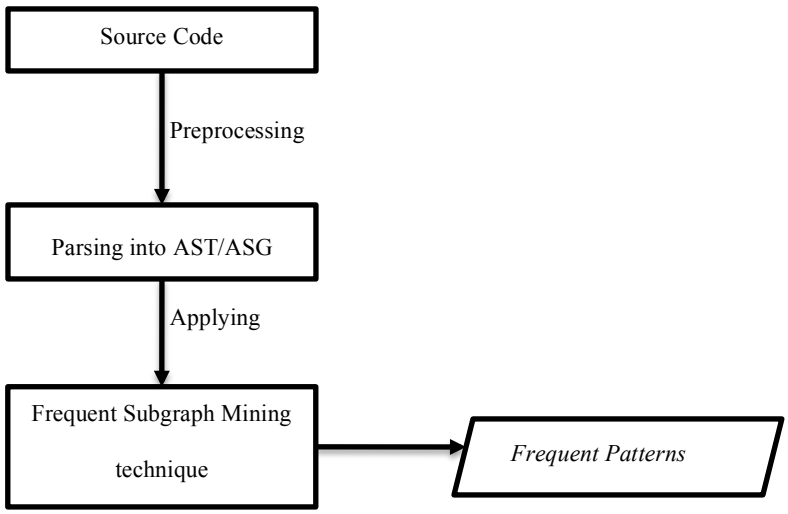


Figure 1. The Research Steps

1.2. Contributions

The main contributions of the research are to examine the applicability of using graph-mining techniques into software engineering domain to solve software engineering problems by discovering useful pattern in source code.

The work investigates two main questions:

- Can graph-mining techniques be successfully applied to software engineering data?
- What types of relevant, practical, software engineering problems can be addressed with these techniques?

To date, there is little or no research that applies graph-mining techniques to a practical software engineering problem. This work strives to identify how the different types of graph mining methods can be applied to different software engineering domains. The results will identify a variety of research problems along with the specifics of the work undertaken here.

The ultimate goal of the work is to uncover frequent patterns in the usage of specific APIs to support the task of automated code completion. This has the potential to greatly improve programmer productivity in the context of correctly using an API or similar library.

1.3. Organization

The dissertation is logically organized into three components: background information on graph mining technique, software engineering problems, and code

completion technique. The dissertation is organized in the following manner. Chapter 2 gives an overview of Graph mining techniques and software engineering problems. Chapter 3 presents TreeMiner algorithm. Chapter 4 describes research approach. Chapter 5 describes the discovered patterns. Following that is the evaluation in Chapter 6. . Conclusions and future work are given in Chapter 7.

CHAPTER 2

RESEARCH BACKGROUND AND RELATED WORK

There have been a number of efforts to mine software repository to solve different spheres in software engineering using standard data mining techniques, the proposed research is based on graph mining techniques. The first to be reviewed is graph-mining techniques. Next, an overview of software engineering problems, and software engineering data types that can be used on graph-mining techniques. Finally, an overview related work on software repository mining.

2.1. Overview of Graph Mining Techniques

Graph mining is a relatively new discipline in data mining, and innovative algorithms have been developed in recent years. It is one of the novel approaches for mining the dataset represented as graph structure to discover the repetitive subgraphs occurring in the input graphs, by finding subgraphs capable of compressing the data by abstracting instances of the substructures, and Identifying conceptually interesting patterns. Graph pattern mining techniques can be categorized into ten groups. Frequent subgraph mining (FSM)/ frequent subtree mining (FTM) is one of the most important techniques in graph pattern mining to extract all the frequent subgraphs/subtree in the dataset that occurrence counts are more than the specified frequency threshold. There are a number of algorithms for FTM such as: TreeMiner (Zaki, 2002) for discovering frequent embedded, ordered, and labeled trees, SLEUTH (Zaki, 2005) like TreeMiner but

for unordered trees. Table 1 shows a list of the popular FTM algorithms [Chuntao, Michele 2013].

Table 1. The popular FTM algorithms

Algorithm	Candidate Generation	Support Computation
TreeMiner	Equivalence class extension	Scope list join
SLEUTH	Equivalence class extension	Scope list
TreeFinder	Apriori itemset generation	Clustering techniques
uFreqT	Rightmost path expansion	Maximum bipartite matching
RootedTreeMiner	Enumeration tree	Occurrence list
FREQT	Rightmost path expansion	Occurrence list
Chopper XSpanner	n/a	n/a
AMIOT	Right-and-left tree join	Occurrence list
IMB3-Miner	TMG	Occurrence list
TRIPS	Leftmost path extension	Hash table
TIDES	Rightmost path extension	Hash table
FreeTreeMiner	Self-join	Subtree isomorphism
FTMiner	Extension tables	Support sets
CMTreeMiner	Enumeration tree	n/a
HybridTreeMiner	Extension + join	Occurrence list

Graph / Subgraph Matching; given tow graphs G and D, the problem is to find a set of Subgraph of G that matches to in a graph D. There are two types of graph / Subgraph

matching algorithms exact and inexact graph matching as shown in figure 1. These are defined as follow:

- Exact graph matching: Given two graphs $G_i = (V_i, E_i)$ and $G_j = (V_j, E_j)$, with $|V_i| = |V_j|$, the problem is to find a one-to-one mapping: $V_i \rightarrow V_j$ such that $(u, v) \in E_j$ iff $(f(u), f(v)) \in E_i$, this is called an isomorphism and G_j is isomorphic to G_i [Kollias, Sathe, Grama 2014].
- Inexact graph matching: is considered when both graphs do not contain the same number of vertices and edges which means that it is not possible to find an isomorphism between the two graphs to be matched, in this case the graph matching tries to find the best matching between them [Kollias, Sathe, Grama 2014].

Vertex Similarity; Given two graphs G_i, and G_j to measure how similar each vertex in the graph G_i to each vertex in the graph G_j. The similarity of two vertices is determined by the similarity of their neighbors. There are two groups of methods to compute graph similarity:

- The first group called similarity score, which indicates how G_i, and G_j are similar in their entirety; based on the principle that G_i, and G_j are similar if they share many vertices and/or edges.

- The second group is a set of numbers X_{ij}, representing the similarity of each vertex v_i in the graph G_i to every vertex v_j in the graph G_j (Local or global method) [Schaeffern 2007].

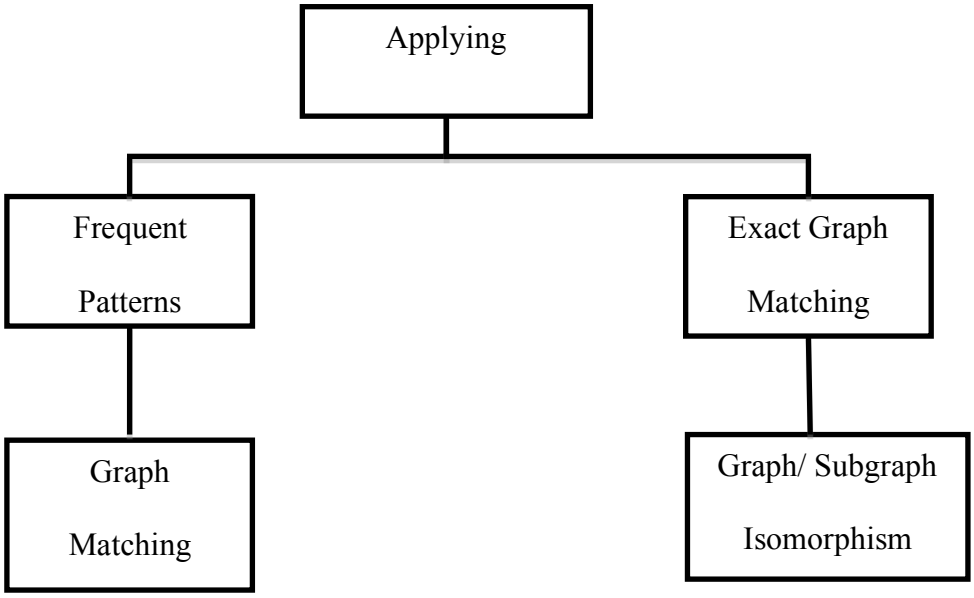


Figure 2. Graph/subgraph matching types

Clustering is the task of grouping objects into clusters based on similarity between pairs of objects. There are two types of clustering in the context of graph data [Lee, Ruan, Jin, Aggarwa 2010]:

- Node clustering algorithm (Vertex clustering): in this case there is a single lager graph which needs to cluster the nodes/ vertices of the graph into clusters by considering the edge structure of the graph in such a way that there should be many edges within cluster and few between the clusters.

- Graph clustering algorithms: in this case there is a set of graphs, which need to be clustered, based on structural similarity.

Dense component discovery is the task of detecting dense component (dense subgraphs) in a graph. It is a potential application to measure the cohesiveness and coupling. There are two types of dense component:

- Absolute density: This is concerned with fully connected subgraph of maximum density [Yu, Cheng 2010].
- Relative density: it compares the density of one subgraph to another subgraph to find the boundaries of components [Yu, Cheng 2010].

Frequent connected subgraph mining is the task of finding all connected graph that are subgraph isomorphic that occurrence count are more than the specified frequency threshold. It is related to dense component discovery, which focuses on making graph smaller, and finding really well connected subgraphs. Reachability/shortest distance is the task of testing if there is a path from a node *v* to another node *u* in a large directed graph, in more formally:

Given a directed graph $G = (V, E)$ with n node and m edges, then the reachability is denoted as $v \rightsquigarrow u$ where u and v are two nodes in *G*, it will return true if and only if there is a directed path in the directed graph *G from u to v* [Al Hasan, Zaki 2011].

- Reachability/ shortest distance concerns about path (narrow), and describing characteristics of the path.

Influence Maximization is the task of finding a small subset of nodes in a graph that could maximize the spread of influence. Link Prediction is considered as a supervised classification task that predicts the likelihood of a future association between two nodes knowing that there is no association between the nodes in the current state of the graph [Xie, Jian 2006]. Link prediction can be used as branch prediction for pre-caching functions before they are called for optimization. Graph Evolution / Temporal Graph; is considered about how graph changes and becomes more (or less) connected over time, potential application with Frequent Subgraph mining to find out both the process and pattern of connection between graphs.

2.2. Overview of Software Engineering Data Types

There are different types of data in software engineering which can represent as graph such as: Call graphs, which is a directed graph that represents calling relationships between subroutines in a program which captures the control flow of the program, given G = (V, E) each node ($v \in V$) represents a procedure method, and function and each edge (u, e) $\in$ E indicates that procedure u calls e. There are two types of call graph; dynamic call graphs, which are created during program execution and they represent the calling structure, and Static call graphs, which are created from the source code of a program, to represent every possible, run of the program.

Program Dependence Graph which is a directed graph that represents a program as G = (V, E) where the nodes such as ($v \in V$) are statements, and predicate expressions

(operators, and operands) and the edges such as (*u, e*) ∈ E indicate the control dependences (control flow).

- Program Dependence graph = Control dependence + data dependence

Co-Change Graph, which is an artificial graph, constructed from version control repository that abstracts the information in the repositories. Given graph G = (V, E) where the nodes such as (v ∈ V) represent the software artifacts, and the change transactions, and the edges such as (u, e) ∈ E connect the change transactions with their participating artifacts [Beyer, Noack 2005].

Abstract Syntactic Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language where each node in the tree represents a construct occurring in the source code.

2.3. Software Engineering Problems

There are a number of software engineering problems that can be possibly addressed using graph pattern mining techniques, such as; clone detection or code clones are sections of source code that are duplicated in multiple locations in a program. Clones are generated mostly due to the copy-and-paste activity of programmers where one section of code is copied and pasted into another location, in some cases with changes and in other cases with no changes between the original and cloned code [Tairas, Gray 2009]. So clone detection process is the discovery of code fragments that compute the same result.

Tairas and Gray [Tairas, Gray 2009] used an information retrieval method Latent Semantic Indexing (LSI) to cluster a large number of clone classes in the Windows NT kernel source code in which are by to help determine relationships among the clone classes. The results of the process yield connections between clones in the clone classes that would not be detected by a clone detection tool alone. These connections range from variations in the syntax of the clones to the use of the clones in different contexts based on the code surrounding the clones. This information could assist a programmer to both understand how the clones are used and to assist when maintenance of the clones is required.

Grant and Cordy [Grant, Cordy 2010] presented a method for estimating the optimal number of latent topics needed to optimize the topic distribution over a set of source code methods, by using Latent Dirichlet Allocation (LDA); which is a generative statistical model that postulates a latent set of topics threaded through a set of documents. It assumes these documents have been generated due to the probability distribution over these topics, and that the words in the documents themselves are generated probabilistically in a similar manner. Moreover, Marcus and Maletic [Marcus et al. 2004b] use LSI to derive similarity measures between source code elements. These measures are used to cluster the source code for the identification of abstract data types in procedural code and for the identification of clones.

Software bug localization is the process of detecting the exact locations of program bug, which is a very expensive and time consuming process. The effectiveness of bug localization depends on developers' understanding of the program being debugged, their

ability of logical judgment, their experience in program debugging, and how suspicious code, in terms of its likelihood of containing bugs, is identified and prioritized for an examination of possible bug locations. Bug localization process divided into two major parts. The first part is to use a technique to identify suspicious code that may contain program bugs. The second part is for programmers to actually examine the identified code to decide whether it indeed contains bugs [W. Eric Wong, and Vidroha Debroy]. The bug can be determined in the program by mining the call graphs, it can be used in order to provide software testing experts with possible bugs, thus they can make corrections, this can be done by using frequent subgraph pattern mining techniques to determine the frequent pattern that occur in the faulty executions.

In the recent past, a number of investigated researches have been addressed the bug localization problem. Eichinger et al. [Eichinger et al. 2010], they addressed the bug localization problem by mining weighted call graphs of program executions Also, they presented an analysis technique for such weighted call graphs based on graph mining and on traditional feature selection schemes. Their approach keeps the size of the resulting graphs relatively small while keeping more important information. In particular, it introduces edge weights representing call frequencies. As none of the recently developed graph mining algorithms analyses weighted graphs, also they developed a combined approach of structural and numerical mining techniques is key for precise localizations. It consists of conventional frequent subgraph mining and subsequently scoring of numerical edge weights using entropy based algorithm.

Cheng et al. [Cheng et al. 2009] proposed new technique called bug signature identification based on top-k discriminative graph mining. They extend RAPID bug signature identification by Hsu et al. in the following dimensions: they proposed a graph-based representation which is more compact and scalable in representing long traces; then they mine for graph patterns which are able to express contextual information incorporating both partial and total ordering of events; they compared and contrasted faulty and correct traces at both event and pattern levels for producing a set of multi-dimensional discriminative features; and finally they allow and account for imperfections in traces and slight variations of bug patterns. Their work produced two sets of graphs corresponding to the bug and correct traces. These graphs are then preprocessed to filter off non-suspicious edges. A top-k discriminative graph-mining algorithm is then run to produce a list of candidate discriminative graphs that serve as bug signatures identifying both the location and the context of a bug. They performed a set of experiments based on the Siemens benchmark dataset. The Experimental results indicated that their technique achieved up to 18.1% higher precision and 32.6% higher recall than RAPID.

In [Di et al. 2006] frequent pattern mining algorithms have been used to enhance fault localization for software systems. Their approach was based on a large set of test cases for a given set of programs in which faults can be detected. It uses a frequent pattern-mining algorithm on the function call trees that are used to represent test executions. They evaluated their approach experimentally using the Siemens programs test suit as benchmark. In addition, Liu et al. [Liu, et al. 2005] proposed new statistical

model-based approach, called SOBER that locates software bugs without any prior knowledge of program semantics. Moreover SOBER models evaluation patterns of predicates in both correct and incorrect runs respectively and regards a predicate as bug-relevant if its evaluation pattern in incorrect runs differs significantly from that in correct ones. SOBER features a principled quantification of the pattern difference that measures the bug-relevance of program predicates. On the other hand some information retrieval (IR) models such as latent semantic indexing technique has been used to build automated techniques for bug localization [Marcus et al. 2004b].

In [Kagdi, Maletic 2006] a sequential-pattern mining approach have been used to uncover frequently co-changed documents in the context of internationalization and localization. They evaluated their approach on a large open-source system. The results indicated that using the historical information in versions archive is a promising source for supporting internationalization and localization of web sites.

Library usage pattern, which is the discovery of library reuse pattern in existing applications. For example discovering library classes and member functions that are typically reused in combination by application classes, it can save a valuable time for the developers. Michail [Miltiadis, Charles 2013] presented an approach based on association rule mining to discover patterns such as components, classes, and functions that occur frequently together in library usages. In addition, Zhong et al. [Zhong et al. 2009] developed a tool called MAPO that mines API usage patterns from open source repositories automatically and recommends the mined patterns and their associated snippets on a programmer's requests. MAPO implements a mechanism that combines

frequent subsequence mining with clustering to mine API usage patterns from code snippets. In addition, MAPO provides a recommender that integrated with the existing Eclipse IDE.

Malicious software is the discovery of specifications of malicious behavior. Malicious code is "any code added, changed, or removed from a software system to intentionally cause harm or subvert the system's intended function" (McGraw and Morisett, 2000, p. 33). Kolter and Maloof [Kolter, Marcus 2006] proposed an approach based on machine learning, data mining, and text classification techniques to address the problem of detecting and classifying unknown malicious executable in the wild. They also evaluated how well the methods classified executable based on the function of their payload, such as opening a backdoor and mass mailing. Mihai et al. [Mihai, et al. 2008] presented an automatic approach to overcome the manual process of investigating known malware. Their approach gains such a specification by comparing the execution behavior of a known malware against the execution behaviors of a set of benign programs. In addition their prototype, MiniMal, infers malspecs by differencing the dependence graphs of a malware sample and of multiple benign programs. Experimental results showed that the malspecs mined by their algorithm compare favor ably with behavioral specifications manually constructed by human experts. Malspecs can also be used to detect multiple malware variants.

Code completion is one of the software engineering problems, which Improve programming productivity by recommending relevant code, and automatically filling in code. Han et al. [Han et al. 2009] presented s technique to complete multiple keywords at

a time based on non-predefined abbreviated input. Also, they presented an algorithm based on an HMM to find the most likely code completions. They presented a method to learn parameters of the HMM from a corpus of existing code and examples of abbreviations. A new user interface for multiple keyword code completion has been implemented on a demonstrational code editor.

In addition, Little and Miller [Little, Miller 2009] proposed new technique to reducing the need to remember details of programming language syntax and APIs, by translating a small number of unordered keywords provided by the user into a valid expression. Also, they present an algorithm for translating keywords into Java method call expressions. When tested on keywords extracted from existing method calls in Java code, the algorithm can accurately reconstruct over 90% of the original expressions.

Nguyen et al. [Nguyen et al. 2012] introduced GraPacc, a graph-based pattern-oriented, context-sensitive code completion approach that is based on a database of API usage patterns. GraPacc manages and represents the API usage patterns of multiple variables, methods, and control structures via graph-based models. In addition, it extracts the context-sensitive features from the code, and their relations to other elements. The features are used to search and rank the patterns that are most fitted with the current code. Moreover, Nguyen et al. [Nguyen et al. 2009] proposed a new approach for mining the usage patterns of objects using graph-based algorithm that takes into account both, temporal usage orders, and data dependencies called GrouMiner. They consider usage pattern as a subgraph that frequently appears in the object usage graphs extracted from all methods in the code base.

2.4. Discussion

After we have overview the graph mining techniques, software engineering problems, and the software engineering data type, we can conclude that there is a number of software engineering problems that can be possibly addressed using graph pattern mining techniques, such as:

- Clone detection. It is the discovery of code fragments that compute the same result. The source code can be represented by using AST to solve this problem by discovering the most frequent subtrees in the source code.
- Code completion. This can be done as follow; firstly the source code is being represented as AST. Secondly the AST is being mined to find the most frequent subtree pattern. Thirdly the link prediction technique is being used to predicate the future pattern.
- Software bug localization. The bug can be determined in the program by mining the call graphs. So it can be used in order to provide software-testing experts with possible bugs. Thus they can make corrections, this can be done by using frequent subgraph pattern mining techniques to determine the frequent pattern that occur in the faulty executions.
- Library usage pattern. It is the discovery of library reuse pattern in existing applications. For example discovering library classes and member functions that are typically reused in combination by application classes. This can be done as follow; firstly the source code is being represented as call graph. Secondly the call

graph is being mined to find the most frequent subgraph pattern. Thirdly the link prediction technique is being used to predicate the future pattern.

- Malicious software. It is the discovery of specifications of malicious behavior. This can be done as follow; firstly collect the execution traces from malware and benign programs. Secondly, construct the program dependence graph. Finally, the program dependence graph is being mined to find the most frequent subgraph pattern.

As shown in Table 2 the applicability of graph mining techniques to solve software engineering problems.

Table 2. Software Engineering Problems, Types of Input, and Graph Pattern Mining Techniques

Techniques / Types of input data	Frequent Subgraph Mining	Graph/ Subgraphs Matching	Vertex Similarity	Clustering	Dense Component Discovery	Reachability /Shortest Distance	Influence Maximization	Link Prediction
Call Graph	Testing the architecture if it is good		Similar functionality (stereo type)	Identify strongly connected subgraph	Use the Frequent call graph to measure the Cohesiveness	Testing fault (if there is a path from a to b through c, e, d) indicating of error (Uncovering fault)		Optimization and (binary organization)
	Library Usage							
	Software Bug Localization		Redundancy	Measuring Coupling, and Cohesion		Configuration Management		
	Mining Call Graph for Optimizing Control Flow							
Program Dependence graph	Identify Common Sequence that should be optimized							
	Specifications of Malicious Behavior							
Co-Change Graph	Unit refactoring, Clone Detection							
AST	Clone Detection Code Completion	Reusable Component						Code Completion
	Code Completion	Applying transformation						What is the next Model that may be evolved (Helpful for planning)

CHAPTER 3

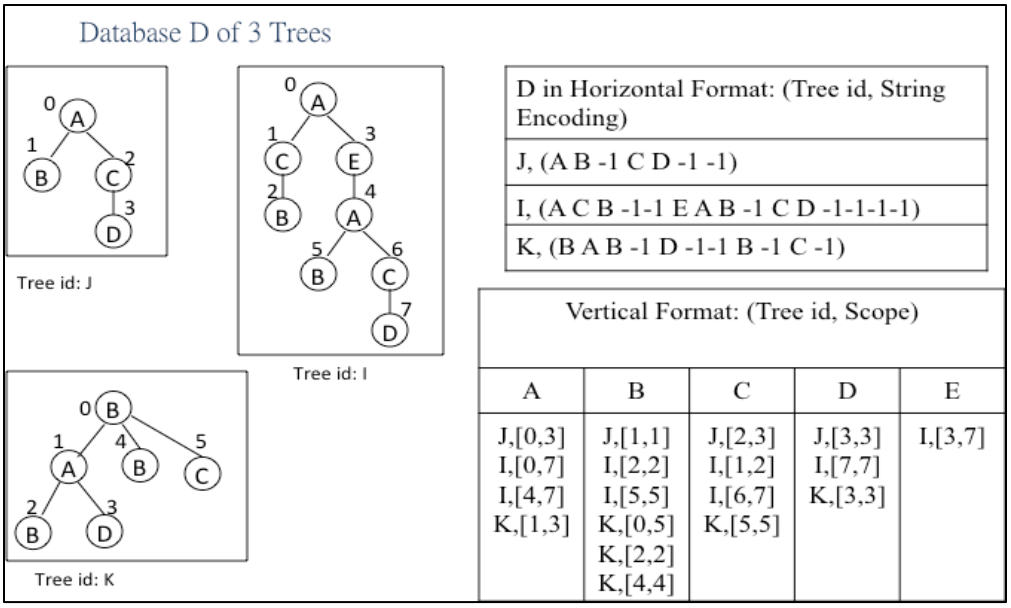
TREEMINER ALGORITHM

In this chapter, we present TreeMiner algorithm, which is one of frequent subtree mining techniques that deal with extracting patterns (association, sequence, frequent tree, graph, and etc.) in massive databases proposed by Zaki, 2005 [Mohammed Zaki 2005] to efficiently enumerate all frequent subtrees in a forest (database of trees) according to a given minimum support (*minsup*). The support of a subtree S is the number of trees in D that contains one occurrence of S. A subtree S is frequent if its support is more than or equal to a user specified *minsup* value.

The intent here is to mine the open source code using the TreeMiner algorithm. The data set that will be mined is the abstract syntax information of the source code. We will extract this information using the srcML infrastructure [].

TreeMiner is an algorithm for mining, rooted tree is a tree that one of the nodes is distinguished from others trees, ordered tree that the children of each node are ordered, then can be designate them as first child, second child, and so on up to k^{th} child. A labeled tree that each node of the tree is associated with label, and embedded tree if the nodes have some common ancestor. TreeMiner algorithm deals with extracting patterns in massive databases that represent *complex interaction* between entities. A preorder traversal is a visitation of nodes starting at the root by using depth-first search from left subtree to the right subtree [Mohammed Zaki 2005].

TreeMiner algorithm has two representation format, horizontal format and vertical format as shown in Figure 3. The horizontal format follows preorder traversal (tree id, string encoding) that used as input to the TreeMiner, the vertical format, which is represented as list of pairs of (tree id and the scope of the node) for each node on a tree. There are two main steps to generating candidate trees; first we have to representing trees as string encoding to construct the horizontal format, then the candidate generation process takes place.



D in Horizontal Format: (Tree id, String Encoding)
J, (A B -1 C D -1 -1)
I, (A C B -1-1 E A B -1 C D -1-1-1-1)
K, (B A B -1 D -1-1 B -1 C -1)

Vertical Format: (Tree id, Scope)

A	B	C	D	E
J,[0,3] I,[0,7] I,[4,7] K,[1,3]	J,[1,1] I,[2,2] I,[5,5] K,[0,5] K,[2,2] K,[4,4]	J,[2,3] I,[1,2] I,[6,7] K,[5,5]	J,[3,3] I,[7,7] K,[3,3]	I,[3,7]

Figure 3. TreeMiner representation format horizontal, and vertical formats

3.1. Frequent Subtree Enumeration

Figure 4, shows the high level structure of the TreeMiner algorithm. The main steps include the computation of the frequent item set and 2 subtrees, and the enumeration of all other frequent subtrees using depth first search within each class of $[P]_1 \in F_2$.

1. *TreeMiner (D (database of tree, Forest), minsup)*
    1. *F_1 = {frequent 1-subtrees};*
    2. *F_2 = {classes $[P]_1$ of frequent 2-subtrees};*
    3. *For all [P], do Enumerate-Frequent-Subtree;*
2. *Enumerate-Frequent-Subtree F_k*
    1. *For each element (x, i) € [P] do*
        i. *For each element (y, j) € [P] do*
            1. *(y, j) join (x, i) => at most two new candidate subtrees*
            2. *For each subtree, do scope-list joins*
            3. *If it is frequent, then we add the subtree to the list of frequent-subtree.*
    2. *Repeated until all frequent subtrees have been enumerated.*

P: prefix class. $[P]_1$ means the prefix size = 1, i.e., only one node in the prefix class. P_x refers to the new prefix tree formed by adding (x, i) to P.

F_k: the set of all frequent subtrees of size k.

Figure 4. TreeMiner Algorithm

To compute F1, for each item $i \in T$, the string encoding of tree *T, i's* count will incremented in one-dimensional vector. This step also computes; the number of trees, maximum number of labels, and the frequent items in the forest. All labels in F1 belong to the class with empty prefix. The position -1 indicates that *i* is not attached to any item. For computing F2 we compute; the support for each candidate using tow dimensional

vector, the scope list for each item in the forest, and the prefix class for each item in the forest. Finally, to compute Fk ($k \geq 3$) the input will be the prefix class with their scope list. To generate frequent subtree we have to do three steps; first is the pruning step to ensure that the subtrees of the resulting tree are frequent, then the scope list join step take place to join the scope list for each items otherwise the scope list join step will be avoided. Figure 5, 6, and 7, shown the main steps in TreeMiner algorithm

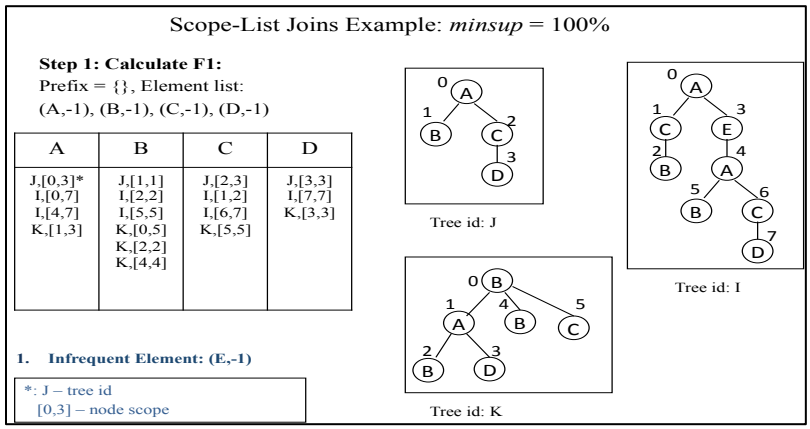


Figure 5. First step in TreeMiner calculate F1

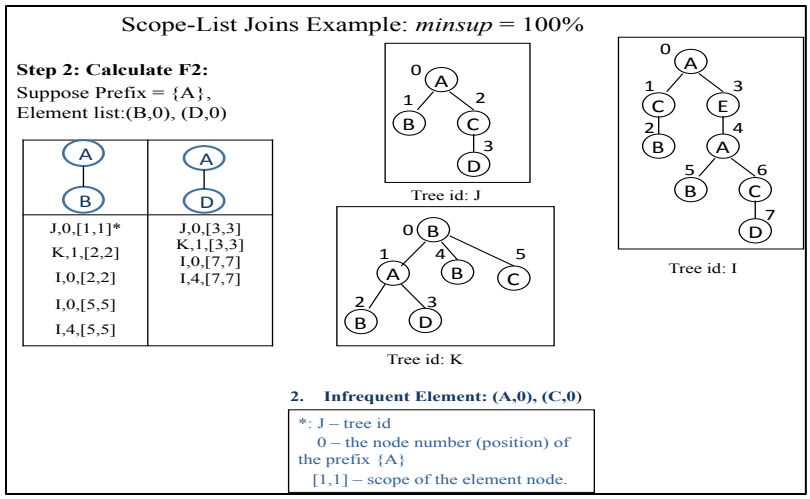


Figure 6. The second step in TreeMiner calculate F2

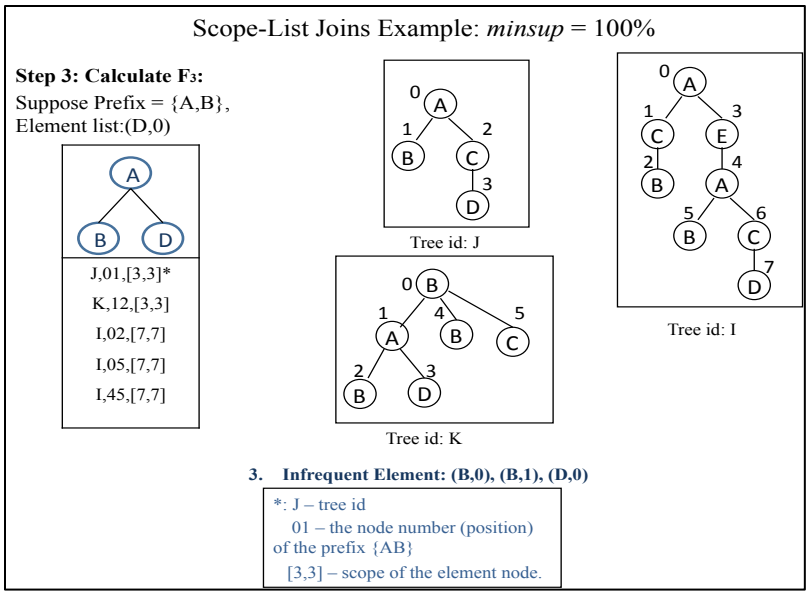


Figure 7. The third step in TreeMiner calculate F3

3.2. TreeMiner Limitations

TreeMiner is open source software that is available to use but not for a larger forest, which is a collection of trees. Unfortunately, we found that the TreeMiner implementation is not scalable to handle large datasets. It only works with small forest that consist of small number of trees and small number of nodes. In the implementation of TreeMiner technique they used a database structure to read transactions (forest) from file into fixed array size to process the dataset, this work with small dataset size only because it is will keep the array, and the scope list for the prefix classes in main memory during the running time of TreeMiner in the case of larger forest this does not work because the memory footprint is too large and a segmentation fault is occurring. They defined in their implementation that the database buffer size is fixed and equal to 2048

and the number of transactions equal to three. As shown on Table 3 the maximum number of trees and nodes that can be handled in the original TreeMiner implementation.

Table 3. The Maximum Forest Size for the original TreeMiner implementation

Maximum Number of Nodes in Each Tree	Maximum Number of Trees	Maximum Number of Node in Forest	Maximum Forest Size
25	3	75	2048 KB

To handle large dataset and to make the TreeMiner more scalable we improve the implementation, using different data structure such as files and vectors instead of using fixed array size and list to hold the data to be processed and to save the result on it. Moreover we have done some changes on the implementation to make it fast and scalable such as in the computing F1 method we replace the vector size value to be equal to the maximum node id instead of using *total number of* F_1 in this way we make it faster and we save a lot of memory space and time. Also we improve the computing of F2 and Fk while we keep all the computation process in file so it is scalable to handle large dataset as shown in Table 4 the maximum forest size that we can handle in the new implementation of TreeMiner for two lager open source systems Qt and HippoDraw.

Table 4. The Maximum Forest Size for the new implementation of TreeMiner

System	Maximum Number of Nodes in Each Tree	Maximum Number of Trees	Maximum Number of Node in Forest	Maximum Forest Size
Qt	238,107	99,412	26,398,180	82.5 MB
HippoDraw	71,444	3,735	1,367,806	4.2 MB

CHAPTER 4

RESEARCH APPROACH

In this chapter, we describe how TreeMiner is used to mine the corpus to discover uncovered patterns that have both textual and syntactic similarities. The proposed approach applies the TreeMiner over a source code. Here, a description of how TreeMiner is used for code completion is given. The process is as follows. First the source code is pre-processed into an AST/ASG. Then the encoding process takes place to represent the AST in horizontal format. Finally the TreeMiner used to discover uncovered patterns. As shown in Figure 8, there are four steps for patterns discovery process using TreeMiner. Each of these steps will be now described in more detail.

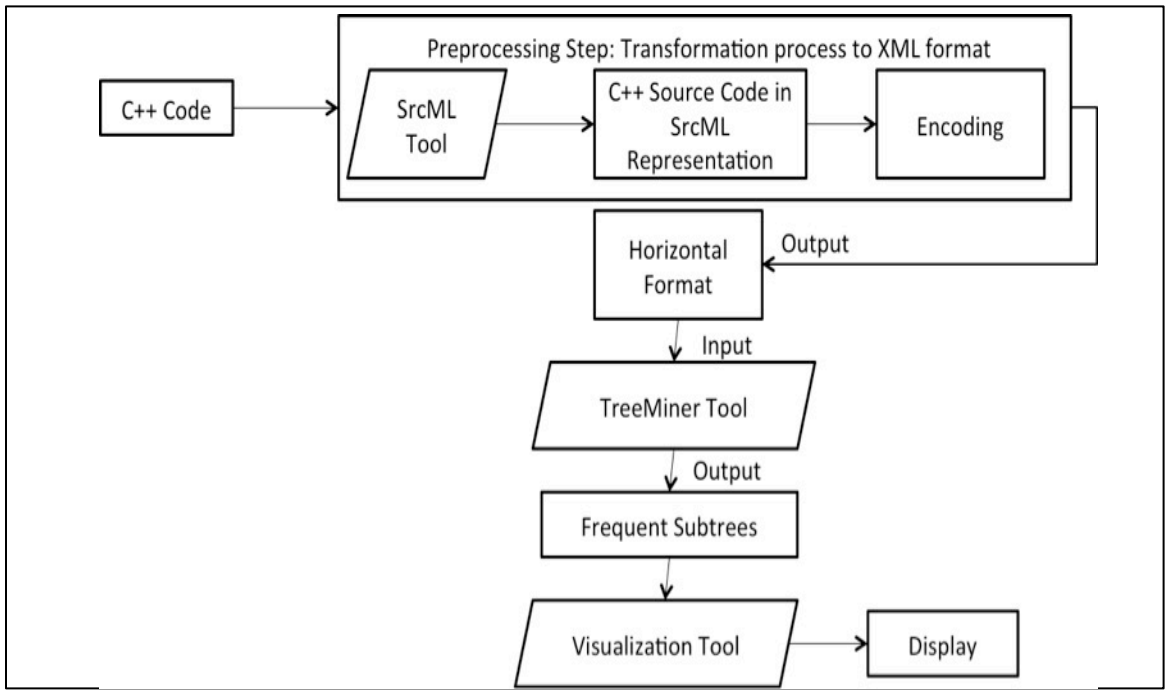


Figure 8. Patterns Discovery Process Using TreeMiner

4.1. Overview of Corpus Generation

We use srcML tool [Collard, Decker, Maletic 2011] to transform the C++ source code to XML format as a first preprocessing step. srcML is an XML representation that supports both document and data views of source code. The format supports lightweight static program analysis using standard XML tools, while at the same time preserving all original lexical information. A very usable and efficient tool to translate C/C++ to/from srcML is freely available[1]. As a next step we developed a program in C++ to encoding source code that in XML format to represent it in horizontal format to be input to the TreeMiner tool to get the frequent subtrees.

4.2. The Encoding Process

After building the corpus, as a next step, the corpus is encoded. In Figure 9, we show the encoding algorithm for the tree *T* as well as for each subtree. So we start at the root of the tree and add 1 to the string. The next node in preorder traversal is labeled 1, which added to the encoding. Then we backtrack to the root adding -1 and follow down to the next node, adding 2 to the encoding. Finally we backtrack to the root adding -1 to the string.

---

[1] See www.sdml.info for srcML downloads and documentation.

```
Encoding Algorithm (T)

Input: T = Tree, n = nod G= Global variable set.        // (C++ source code in SrcML),

Output: (T', l) = labeled tree                          // (Horizontal Format)

1. Set [T'] = Ø;
2. Start from the root node n0
3. For all n ϵ T do
4. Depth-First Preorder (T, n, G):
5. For each n ϵ T do
    1. If node ni is not labeled do
    2. If ni ϵ G do
        1. T'. Push ni (l);        // add the current nod's label l to T' if it is not global node
    3. If n is a leaf node than    // whenever we Backtrack from child to its parent add -1
        1. T'. Push ni (-1);
    4. Whenever Backtrack from n to its parent do
        1. T'. Push ni (-1);
    5. Recursively call Depth-First Preorder (T, n);
6. Return (T', l);
```

Figure 9. The Encoding Algorithm

Here we define a *class signature* as a frequency distribution of method stereotypes for a class. We use the class signature to infer a class's stereotype. In this section, we summarize how we defined and automatically identified method stereotypes as this forms the basis for the signature. Specifics of the class signature are then presented.

4.3. Applying TreeMiner

After encoding process, as a next step, the horizontal format will be the input to TreeMiner tool to generate the frequent subtrees. As shown in Figure 10, the main three steps in TreeMiner tool. First step will take the horizontal format to compute the F1 method to generate the frequent 1-subtrees. In the second step F2 method will generate

the vertical format and the frequent 2-subtrees. In the third step will compute the Fk method will cover all frequent k-subtrees.

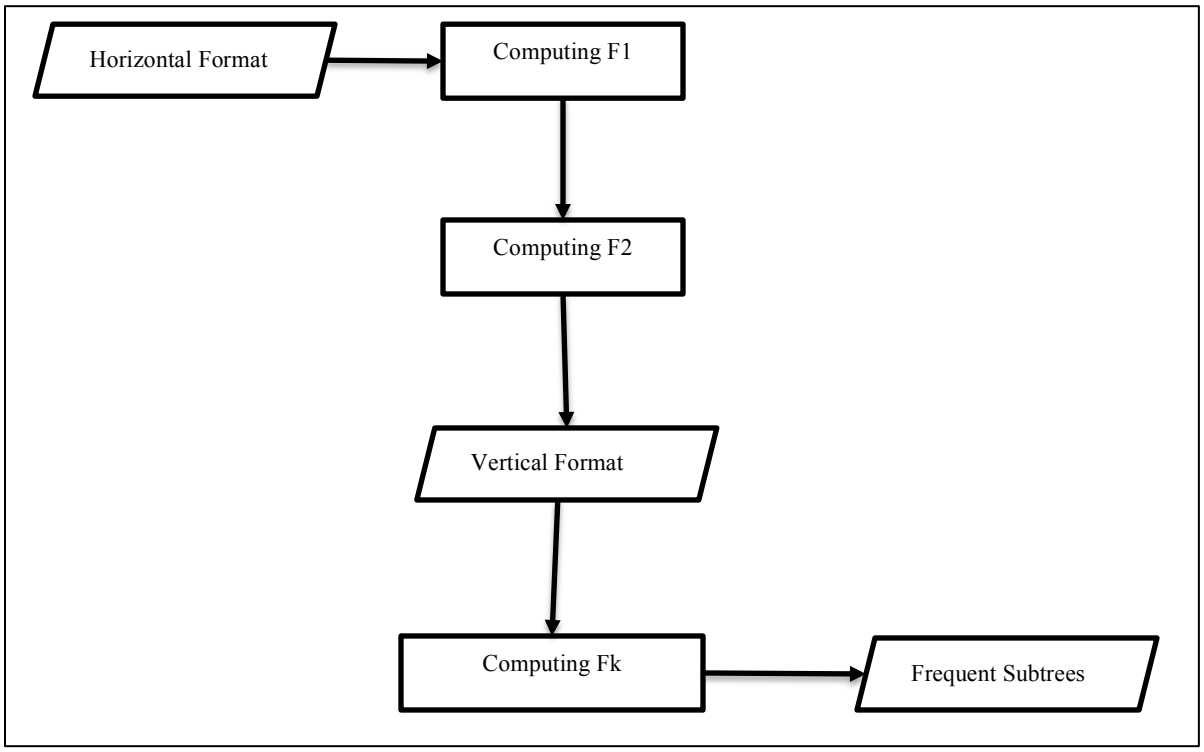


Figure 10. TreeMiner Steps

4.4. Code Completion

In this section, we will present our proposed technique for code completion problem. Code completion is a novel technique to improve the efficiency of code writing by supporting code completion of full statement or code block based on frequent subtrees patterns learned from a corpus of existing code and to reduce the burden of memorizing the details of programming languages and the API usages. In another hands code completion technique shows how other developers have used the

API in similar situations. Our code completion approach has two stages: the offline stage, and the online stage. Figure 11 shows the code completion process.

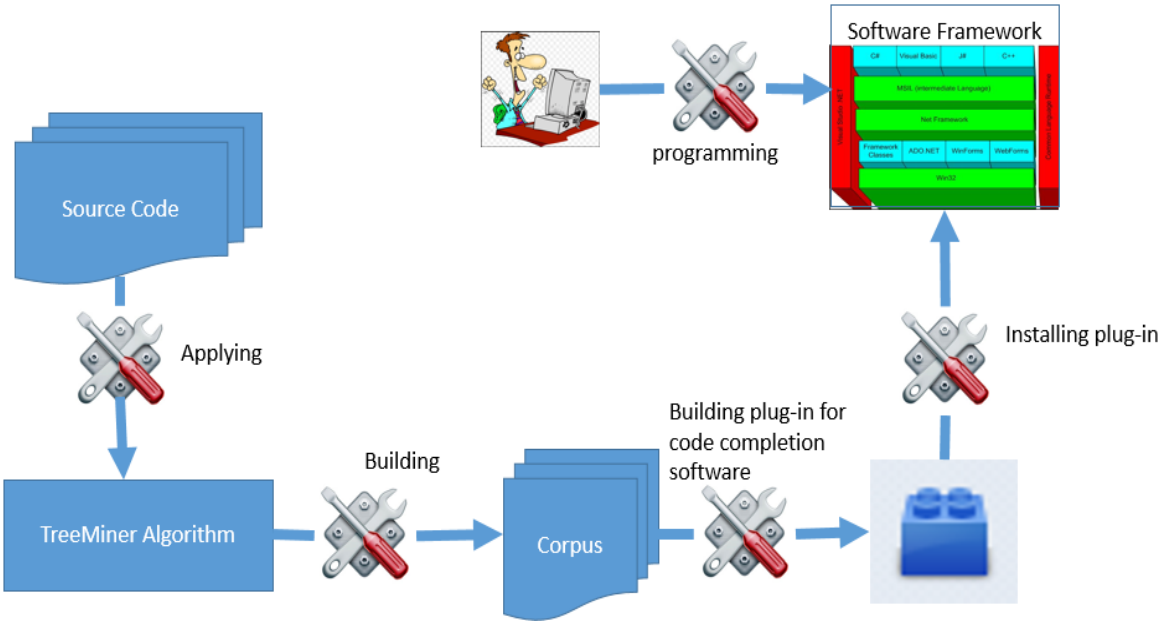


Figure 11. Code Completion Process

4.4.1. The Offline Stage

In this stage we used the TreeMiner algorithm to mine existing systems to find the most frequent subtrees patterns to build the corpus, which will be used in the next stage. The corpus will be then used as a database to be integrated with a plug-in software that will provide the user with graphical user interface (GUI) to interact with programmer in the online stage of code completion technique. Figure 12 shows the code completion plug-in framework.

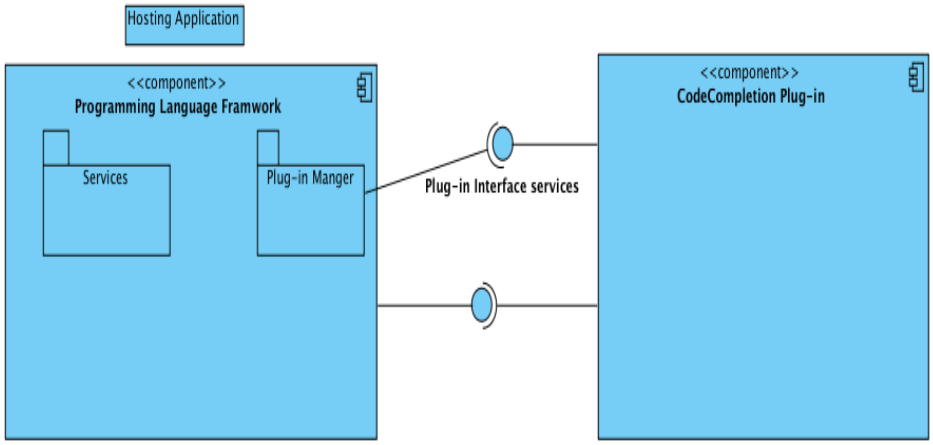


Figure 12. Code Completion Plug-in Framework

4.4.2. The Online Stage

In this stage we will use one of the graph mining technique called Link prediction, this will be used to predict the future pattern while the programmer is typing and will show a list of suggested statements or blocks to complete the rest of the statement or function. After building the code completion, plug-in will be installed and integrated with the programing framework. Figure 13 shows an example of code completion technique. The programmer types some code and once the software detect and recognized string such as class name, function name, object name, or variable name identifier it will presents a list of suggested statements, this list will be sorted based on the frequency, to the programmer which contains the complete statement or code block, and the programmer makes a choice with his or her moues or the keyboard arrow.

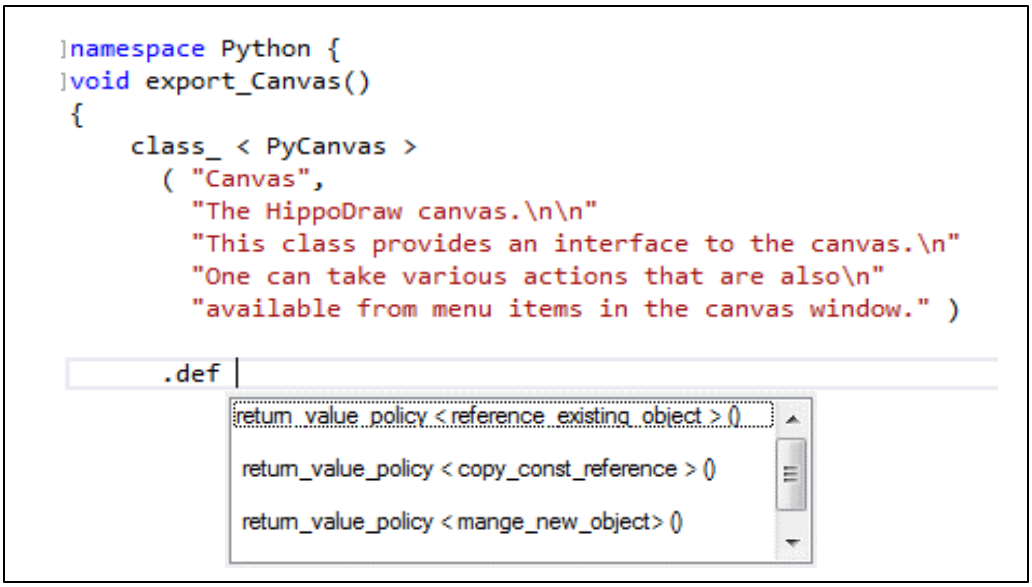


Figure 13. An example of code completion technique

CHAPTER 5

DISCOVERING PATTERNS

In this chapter, we apply TreeMiner technique to six open-source systems, HippoDraw, Qt-examples-5.3, Qt_examples5.4.2, ClanLib-4.0.0, wxWidgets-3.0.2, and calligra-2.9.5 (Koffice) all of these open-source system use qt. HippoDraw is written in C++ and provides a data analysis environment. It is a widely used application and has features for data analysis processing and visualization with an application GUI interface. It can be used as a Python extension module, allowing users to use HippoDraw data objects with the full power of the Python language. Its library consists of approximately 50K lines of source code and over 300 classes. HippoDraw 1.21.3 release is chosen in this case study. To use HippoDraw in the study we transform the source code to XML format using srcML tool than an encoding process took place to represent XML format in the horizontal format. Table 5 shows, the number of function in HippoDraw, number of node, and the maximum number of node. We consider each function as a single tree.

Table 5. HippoDraw System

Total Number of Tree	Total Number of Node	Maximum Node Id
3,735	951,716	12,692

5.1. Results and Observation

The following represents the results and observations from the exploratory case study where we tried to answer the following questions:

- Can we apply graph-mining techniques to software engineering data?
- How can we apply graph mining in different spheres in software engineering?

The first method is computing F_1 that works as follow; for each node $i \in T$ *(tree)* we do, Incrementing *i's* count in one dimensional vector, then storing it in F1_file (contain Node id, frequency, and TreeId), Output F1_file (List of frequent 1-subtrees). The second method is Computing F_2 that is working as follow; firstly we compute the supports of each candidate by using two-dimensional vector of size *(Max_NodeId* Max_NodeId)* instead of using (*total number of* F_1*)* and F2-file to store it. Secondly we compute the prefix classes for the Eqclass. Finally we generate the scope_list (vertical representation) for each frequent node in F_1 *(*$i \in F_1$*)*. The output are; Prefix classes and frequent 2-subtrees. Finally, the last method in TreeMiner is to compute Fk; the input to the method is a set of elements of class *[P]* prefix class along with their scope lists. The frequent subtrees are generated by joining the scope lists for each item. Before the joining step, we have the pruning step to ensure that subtrees of the resulting tree are frequent; otherwise we can avoid the join. Table 6 shows the summary of F_1, F_2, and F_3 methods: number of frequent items and the total run time.

Table 6. Summary of F1, F2, and F3 methods

Methods	Total number of frequents items	Total run time
F1	143,496	02:50.40 Min
F2	900,772	01:45:18.37 Min
F3	51,790	02:14:10 Min

5.2. Sample of Result

This section shows a sample of HippoDraw frequent patterns for each method in the TreeMiner algorithm. In the first method F1 we found that the total number of frequent items was 143,498. The prefix class is empty in this step. We have two sets of frequent items; the first set is the frequent items related to C++ code structures (if, for, while, etc.); the second set is the frequent identifier names and function names. Table 7 shows some of the frequent items included in these two sets.

Table 7. Sample of frequents items F1

Frequent Items Related to C++ Code Structures	Frequency	Frequent Identifier Names and Function Names	Frequency
Void	2417	setRange	147
If	4024	Range	418
For	612	Axis	1103
While	234	m_axis	128
Else	972	getY	87
Try	81	getX	84
Switch	76	const_withd	27
Catch	85	getHeight	68
PyArray_TYPES	32	Rect	803
getEnum	19	getDrawRect	40
Class_	33	def	309
NumArrayTuple	37	return_value_policy	84
Bases	24	copy_const_refernce	42
DataSource	345	refrence_existing_object	20
setLabels	16	mange_new_object	16
getLabels	34	return_by_value	6

In the second method F2, the total number of frequent 2-subtrees is 900,772.the prefix class contains one item on it. We have two sets of frequent 2-subtrees, first set includes the code structure statement, and the second set includes identifier names, function names. Table 8 shows some examples of these frequent 2-subtrees. We use this notation ➔ to represent the parent child relation.

Table 8. Sample of Frequent 2-Subtrees

Frequents 2-subtrees	Frequency
if {for{}}	460
for {for{}}	90
for{push_back}	235
for {getDrowRect}	25
while {for{}}	29
setRange ➔ (range)	38
setRange ➔(Axes::x)	20
(range.) ➔(setRange)	7
(m_x_axis.)➔(setRange)	13
(m_y_axis.)➔(setRange)	13
def-> return_value_policy	84
(Plotter.)➔(setRange)	6
(m_binner)➔(setRange)	10

def -> replaceColumn	8
Observable -> notifyObservers () m_canvas_view -> notifyObservers () DataRep -> notifyObservers () DataSource -> notifyObservers () Ntuple -> notifyObservers () CanvasWindow -> notifyObservers () CanvasView -> notifyObservers () plotterBase -> notifyObservers ()	20 14 8 5 11 4 3 5

In the third method F3, the total number of frequent 3-subtrees is 59,239. Table 9 shows some examples of frequent 3-subtrees. The prefix class consists of two items P [P_1, P_2].

Table 9. Frequent 3-Subtrees

Frequents 3-subtrees	Frequency
(setRange)➔(Axes :: x, range)	12
(if) ➔ {for{➔ (puch_back)}}	84
(setRange)➔(Axes :: x, range, false)	9
(setRange)➔ (axis, low, high)	14
(setRange)➔(m_axis, range, true, false)	11
(setRange)➔ (axis, low, high, false)	11

(setRange)➔ (low, high, pos)	9
(def)➔return_value_policy➔ (refernce_existing_object)	22
(def) ➔ (return_value_policy) ➔ (copy_const_refernce)	42
(def)➔ (return_value_policy) ➔ (mange_new_object)	16
(def) ➔ (return_value_policy) ➔ (return_by_value)	6
FunctionController➔ {fcontroller, FunctionController{instance ()}}	36
If {(! Fcontroller) ➔ (hasFunction ())➔ return}	13
Ifdef {HAVE_NUMPY}	14
Class_{bases <>}	24
def {“setLabels”, setLabels (label)} def {“replaceColumn”, replaceColumn () }	3
def {“getLabels”, getLabels (label) }	4
def {“addColumn”, addColumn ()} def {“replaceColumn”, replaceColumn () }	7
def {“replaceColumn”, replaceColumn () }	15
PyArray_TYPES➔ {getEnum ➔ {return ➔ PyArray_TYPES}}	19

5.3. TreeMiner Performance

The common metrics to measure the TreeMiner performance are number of nodes, number of trees in the forest, the size of the forest, the forest size, and the total

run time. Our evaluation consist of two issues the scalability, and the effect of using different value of support.

We now analyze the performance of the TreeMiner tool, all the experiments were performed on a 3.20GHz with 6.00 GB memory running Ubuntu (64bit) Linux. Timings included the preprocessing costs.

Scalability comparison: Table 10 shows the six open source systems with the number of tree, total number of nodes, and the time. Figure 15 shows how the TreeMiner toll scale with increasing number of nodes (forest size) in the forest, from 1 million to 16 million nodes. We find a linear increase in the run time with increasing forest size.

Table 10. Six Open Source Systems

System	Size	Number of Trees	Total number of node	Time
HippoDraw	3.1 MB	3735	1,367,806	3:40:20
Qt-examples-5.3	7.1 MB	2001	2,200,738	6:11:39
Qt_examples-5.4.2	8.1 MB	2124	2,389,936	6:39:58
ClanLib-4.0.0	10.6 MB	1702	3,092,596	8:58:25
wxWidgets-3.0.2	57.5 MB	4328	16,744,298	15:30:25

Calligra-2.9.5 (Koffice)	78.8 MB	9170	23,131,655	23:21:19

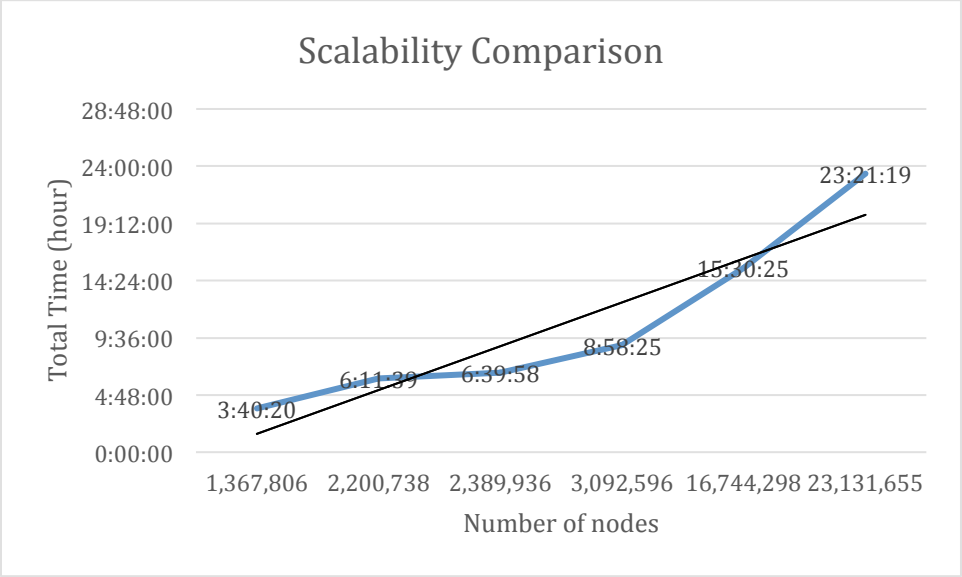


Figure 14. Salability of TreeMiner

TreeMiner performance Vs. Support comparison: Figure 16 shows the performance of TreeMiner for the HippoDraw with different value of support, for the highest support it's outperforms than the lowest supports. We find a linear increase in the run time with increasing support. There are two reasons for this. First, we need to pay the cost of generating the subtree of each new pattern, and this adds significant overhead, especially for lower supports, because there are many frequent patterns. Second, when we have lower support the length of the patterns is too long.

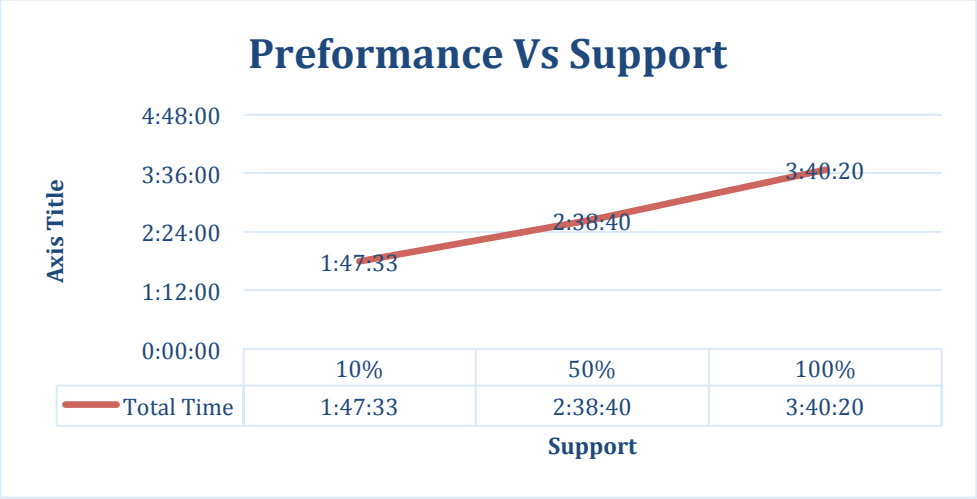


Figure 15. TreeMiner Performance

5.4. Patterns Discovered

The open source medium sized software system HippoDraw are used in the case study. Our TreeMiner tool was applied to extract frequent subtree patterns. The source code of HippoDraw is well written, follows a consistent object oriented style, and the documentation is available.

5.4.1. HippoDraw

In this section we present the patterns discovered and their correlation to code completion. Figure 17 shows a list of some discovered patterns.

(setRange)➔(AXES :: X, RANGE)
(if) ➔ {for{➔ (puch_back)}}
(setRange)➔(Axes :: x, range, false)
(setRange)➔ (axis, low, high)
(setRange)➔(m_axis, range, true, false)
(setRange)➔ (axis, low, high, false)
(setRange)➔ (low, high, pos)
(def)➔return_value_policy➔ (refernce_existing_object)
(def) ➔ (return_value_policy) ➔ (copy_const_refernce)
(def)➔ (return_value_policy) ➔ (mange_new_object)
(def) ➔ (return_value_policy) ➔ (return_by_value)
FunctionController➔ {fcontroller, FunctionController {instance ()}}
If {(! fcontroller) ➔ (hasFunction ())➔ return}
PlotterBase ➔{ (plotter) getPlotter()}
If {(!plotter) return}
Ifdef {HAVE_NUMPY}
Class_{bases <>}
def {“setLabels”, setLabels (label)} def {“replaceColumn”, replaceColumn () }
def {“getLabels”, getLabels (label) }

def {“addColumn”, addColumn ()} def {“replaceColumn”, replaceColumn () }
def {“replaceColumn”, replaceColumn () }
PyArray_TYPES➔ {getEnum ➔ {return ➔ PyArray_TYPES}}

Figure 16. HippoDraw Discovered Patterns

For example we can look at *FunctionController* in depth. *FunctionController* is one of the frequent pattern each time it is called create and instant called *fcontroller* that frequently occurs with *FunctionController* and it called another function called *instance ()*. Moreover, there is a relation between (*FunctionController➔ {fcontroller, FunctionController {instance ()}})* pattern and *If{(!fcontroller) ➔ (hasFunction ())➔ return}* pattern its also occurs together. Figure 18 shows a real example from HippoDraw system.

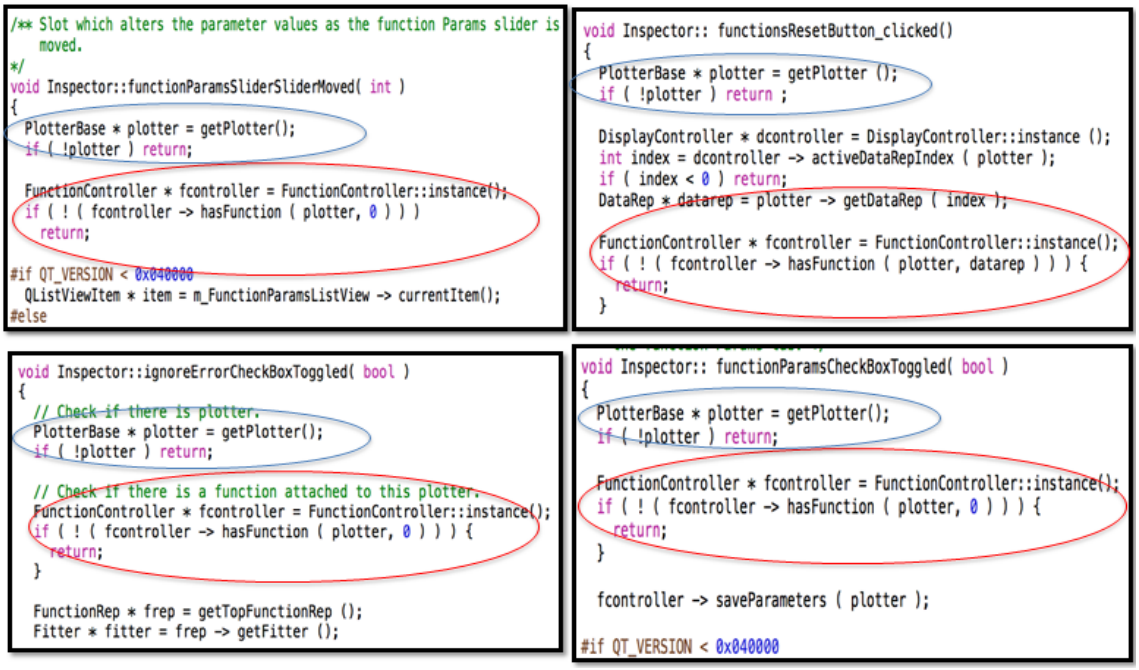


Figure 17. HippoDraw Result

Another example of HippoDraw patterns is *PyArray_TYPES,* which frequently occurs with function called getEnum in different trees as Figure 19 shows a sample of HippoDraw source code.

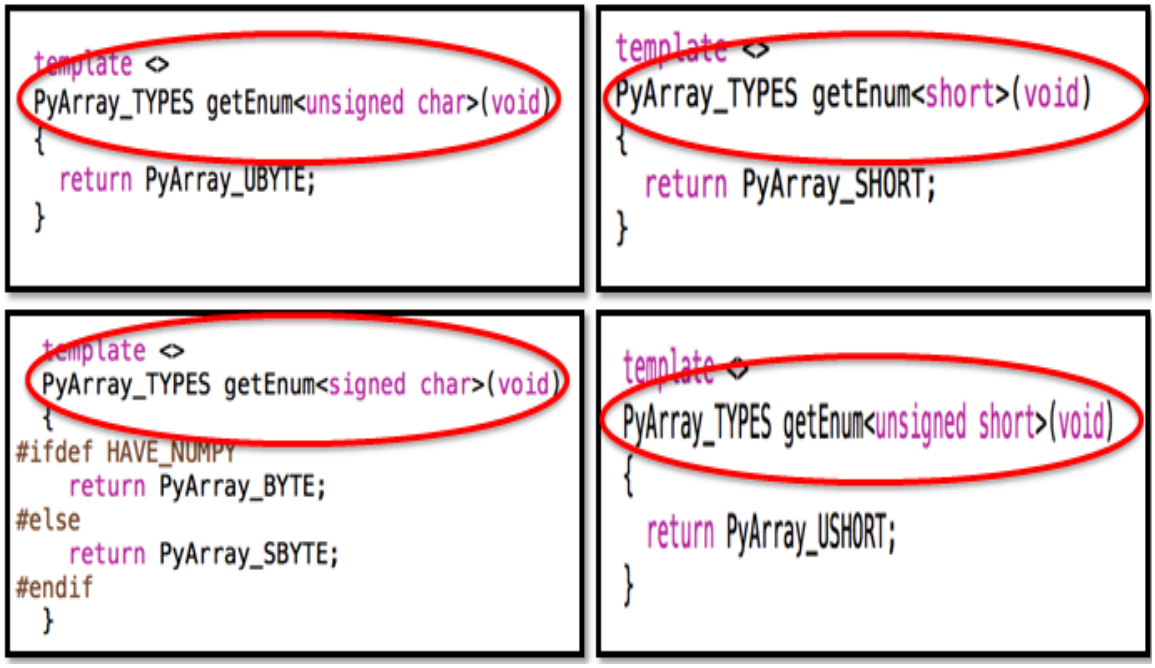


Figure 18. Sample of HippoDraw Source Code

Another frequent function called *def,* has a number of frequent parameters that occurs frequently such as, *{"setLabels", setLabels (label)}, {"replaceColumn", replaceColumn ()}, {"getLabels", getLabels (label)}, {"addColumn", addColumn ()}, {"replaceColumn", replaceColumn ()}, (return_value_policy➔ (refernce_existing_object)), and (return_value_policy) ➔ (copy_const_refernce)*. As show in Figure 20.

Figure 19. Def Function Usages

5.5. Discussion

The results show that the use of the TreeMiner algorithm will cover useful patterns that can be used in different software engineering problems since it capture the relation between patterns. These patterns will be used in the code completion technique. We plan to extend this work by studying more Qt API. Also it shows that we can use the TreeMiner algorithm to solve code completion problem, which is one of the software engineering problems, so we can improve programming productivity by recommending relevant codes and automatically filling in code. setRange is one of

the frequently used functions in HippoDraw system with different parameters data type. Forth more it was called in different classes with frequent objects. As shown in Figure 20, which shows how we can use TreeMiner results to solve code completion problems for example if the programmers want to use the setRange function we will give them a list of, calling statements with the parameters list to choose from.

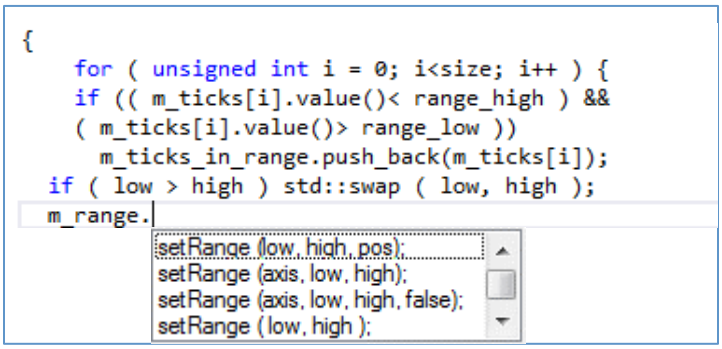


Figure 20. Code Completion Example

CHAPTER 6

EVALUATION

To evaluate the approach we need to first mine a set of patterns and then apply them in the context of our problem of code completion. We compare the results of our automatic TreeMiner tool with that of human experts. The evaluation methodology is to first apply TreeMiner to mine a portion of the HippoDraw system as a training set. Next we mine a later part of the HippoDraw system (the testing set) manually and see if the resulted patterns from the training set can be accurately predict patterns in the testing set that would be suggested in the code completion method. Table 11 shows the number of trees, the number of nodes, and the maximum node ID in the training set.

Table 11. The Training Dataset

Number Of Trees	Number Of Nodes	Maximum Node Id
2,490	687,698	12,692

6.1. Discovered Patterns in the Training Set

We allocated 2/3rd of HippoDraw system to the training dataset. This training dataset contains 2,490 functions (or trees). The remainder was allocated to the test dataset 1,245 functions (or trees). We configured TreeMiner tool to mine frequent

subgraph patterns with minimum support of five for all frequent subgraph patterns. Table 12 shows a sample of discovered patterns in the training dataset with support.

The usefulness of the discovered frequent subgraph patterns are shown in how well the training dataset will predict the future patterns for the code completion methods.

Table 12. Discovered Patterns in the Training Dataset

Discovered Patterns	Support
(*PlotterBase* ➔ *{ (plotter) getPlotter()})*	15
(If ➔ *{(!plotter) return})*	12
(def)➔*(PyCanvas,* return_value_policy*)*➔(refernce_existing_object)	8
(NumArrayTuple➔ *{(nt)(dynamic_Cost{(NumArrayTuple), (m_dataSource)})*	6
Class_*➔{bases <>, boost{noncopyable}}*	13
Class_*➔{NumArrayTuple, addColumn}*	17
nt (hippo➔*{NumArrayTuple})*	6

6.2. Validation Metrics

In this section we present metrics used to validate our approach these metrics are: *recall, precision, and coverage.* To measure the usefulness of code completion predation we use the recall and coverage [Kagdi, Maletic, Sharif 2007].

Let suppose that we have the training dataset as GT = {gt_1, gt_2… gt_n}, the testing dataset as GS = {gs_1, gs_2… gs_m}, and the frequent discovered subgraph patterns as GP = {gp_1, gp_2… gp_i}. To predict these frequent subgraph patterns, the training dataset would be queried for candidates that suggested what was actually used in the testing dataset.

Metrics definition:

Covered subgraph patterns is a discovered subgraph pattern in the testing set for which there is at least one-candidate subgraph patterns suggested from the training dataset.

Coverage is the percentage of the total number of covered patterns to the total number of subgraph patterns in the testing dataset.

$$Coverage = \frac{CoveredPatterns}{|NumberofPattern\sin Trainingset|} \times 100\%$$

Recall is the percentage of the total number of correctly covered patterns to the total number of patterns in the testing dataset.

$$Recall = \frac{CorrectlyCoveredPatterns}{|NumberofPatterninTrainingset|} \times 100\%$$

Coverage and recall metrics are measure of the completeness of the training dataset in predicting the testing dataset subgraph patterns.

Correctness for any given covered pattern in the evaluation-set, only that pattern is suggested from the training-set to any of its elements in the testing set, the percentage of the number relevant patterns over the number of suggested candidates of all its elements

$$Correctness = \frac{Numberofrelevant Patterns}{|Numberofsuggestedcandidates|} \times 100\%$$

6.3. Predicting Patterns in the Testing Set

We use the subgraph patterns discovered in the training dataset for code completion prediction on the testing dataset. This step is preformed manually to mine frequent subgraph patterns with minimum support of three for all frequent subgraph patterns. Table 13 shows the size of testing dataset.

Table 13. The Testing dataset

Number Of Trees	Number Of Nodes	Maximum Node Id
1,245	264,018	8,814

For the sample of the discovered patterns in the training set (in table 12) was predicted about 70% (5/7) in the testing set patterns and we discovered an uncovered patterns in the training set additional patterns was added and covered. As Table 14 shows the covered patterns and uncovered patterns with support, and recall percentages. Furthermore, the usefulness of the uncovered subgraph patterns can be shown in how well the training dataset predicts the existence of subgraph patterns in

the Testing dataset to found all the API usages. We are planning to extend our evaluation to cover all the five open source system in our case study. Table 15 shows the coverage and recall percentages for the covered and uncovered patterns. Also Figure 21 shows the recall and the coverage percentages. Furthermore, Table 16 shows the *Correctness* for Patterns discovered from training set correctly predicted in the test set, these are the four patterns with the highest support produced from the training set.

Table 14. A sample of the Discovered Patterns in the Testing Dataset and Training Dataset

Training set Patterns		Testing dataset Patterns	
Patterns	Support	Patterns	Support
(*plotterbase ➔ { (plotter) getplotter()}*)	15	(*plotterbase ➔ { (plotter) getplotter()}*)	4
(If ➔ {(!Plotter) return})	12	*(If ➔ {(!Plotter) return})*	4
(def)➔*(pycanvas,* return_value_policy*)*➔(refernce_existing_object)	8	(def)➔*(*return_value_policy*)*➔(refernce_existing_object)	5
(numarraytuple➔ {(nt)(dynamic_Cost{(numarraytuple), (m_datasource)})	6	*M_project➔ {new (DyHistzDProject) }*	6
Class_➔{bases <>,	*13*	*Class_➔{bases <>}*	8

boost{noncopyable}}			
Class_➔{numarraytuple, addcolumn}	*17*	*FunctionProjector➔ {(fn), (m-Projector)}*	18
Nt(hippo ➔{numarraytuple})	*6*	*Hippo➔{numarraytuple}*	4

Table 15. The Covered and Uncovered Patterns Recall and Coverage percentages

Patterns	Recall %	Coverage %
Covered Patterns in training set	72%	78%
Uncovered Patterns in the training set	28%	22%

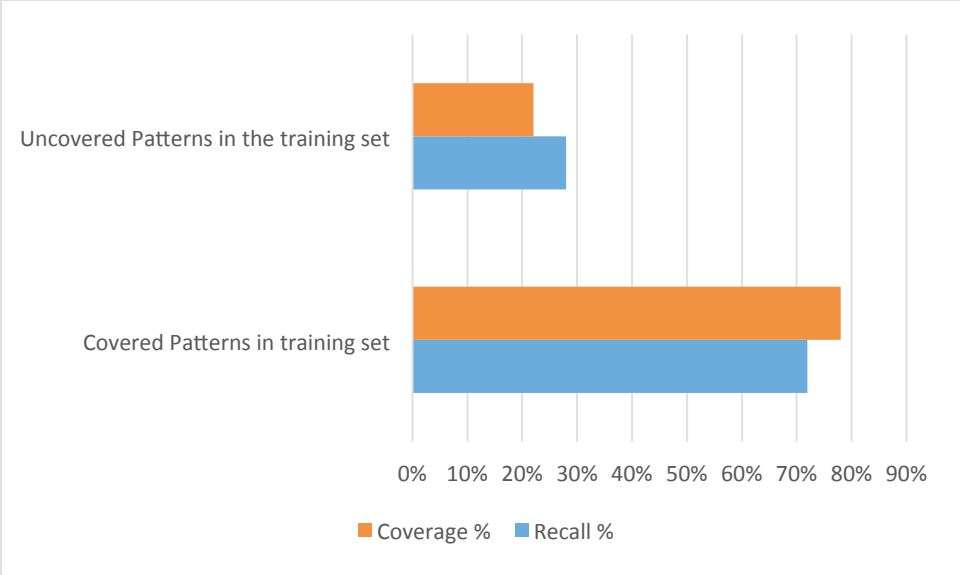


Figure 21. The Recall and Coverage Percentages

Table 16 the Correctness of the Patterns discovered from training set correctly predicted in the test set

Pattern	Support	Correctness
(def(return_value_policy (refernce_existing_object, copy_const_refernce, mange_new_object, return_by_value)))	41	74%
(def (DataSource (setTitle, getLabels)))	12	25%
(If (!Plotter, return))	12	66%
Class_ (bases <>)	13	62%

6.4. Discussion

We found that our TreeMiner algorithm works well in discovering patterns we believe that our approach is applicable to any open source software. As we can see from our finding such as *(def)➔(return_value_policy)➔(refernce_existing_object)*, and *Class_➔{bases <>}* these patterns in the testing set are more general patterns than in the training set. We do not claim that our result would generalize to any systems that are representing different domains. We found that TreeMiner is working better in finding patterns in the training because the number of patterns in the training set is larger than in the testing set moreover the discover patterns in the training set is also founded in the testing set.

CHAPTER 7

CONCLUSIONS

The dissertation addresses several research issues related to Mining Software Repositories (MSR). The goal of the research is to apply graph-mining technique to the domain of software engineering in order to solve a relevant problem. In particular the work focuses on using the graph mining technique of Frequent Subgraph Mining. Frequent subgraph mining identifies which subgraphs in a given dataset occur most often. The work proposed here is to apply frequent subgraph mining to the source code of large-scale software systems. Specifically, the abstract syntax tree of the source code will be mined. This will allow for the discovery of patterns that have both textual and syntactic similarities.

This is a major departure from existing techniques in pattern mining of source code. Current methods and research examines only textual similarities and frequency. Syntactic similarities are not accounted for or considered. The software engineering problem that will be addressed is automated code completion in the context of API usage. That is, frequent patterns of the use of an API will be uncovered from existing software systems that use the particular API. These uncovered patterns will then be used in the prediction of a user's intent while typing in source code. When a feature of the API is used a number of options will be given to the user to select from. These options will be a complete statement and correct usage of the API.

An introduction of the problem is given along with an overview of the literature on graph mining and possible applications to software engineering. Preliminary findings include the enhancement of existing frequent subgraph mining tools to scale for the very large abstract syntax graphs of software. The results of mining relatively large open source software systems are also presented.

7.1. Contributions

The main contributions of the research are to examine the applicability of using graph-mining techniques into software engineering domain to solve software engineering problems by discovering useful pattern in source code.

The work investigates two main questions:

- Can graph-mining techniques be successfully applied to software engineering data?
- What types of relevant, practical, software engineering problems can be addressed with these techniques?

To date, there is little or no research that applies graph-mining techniques to a practical software engineering problem. This work strives to identify how the different types of graph mining methods can be applied to different software engineering domains. The results will identify a variety of research problems along with the specifics of the work undertaken here.

The ultimate goal of the work is to uncover frequent patterns in the usage of specific APIs to support the task of automated code completion. This has the potential to

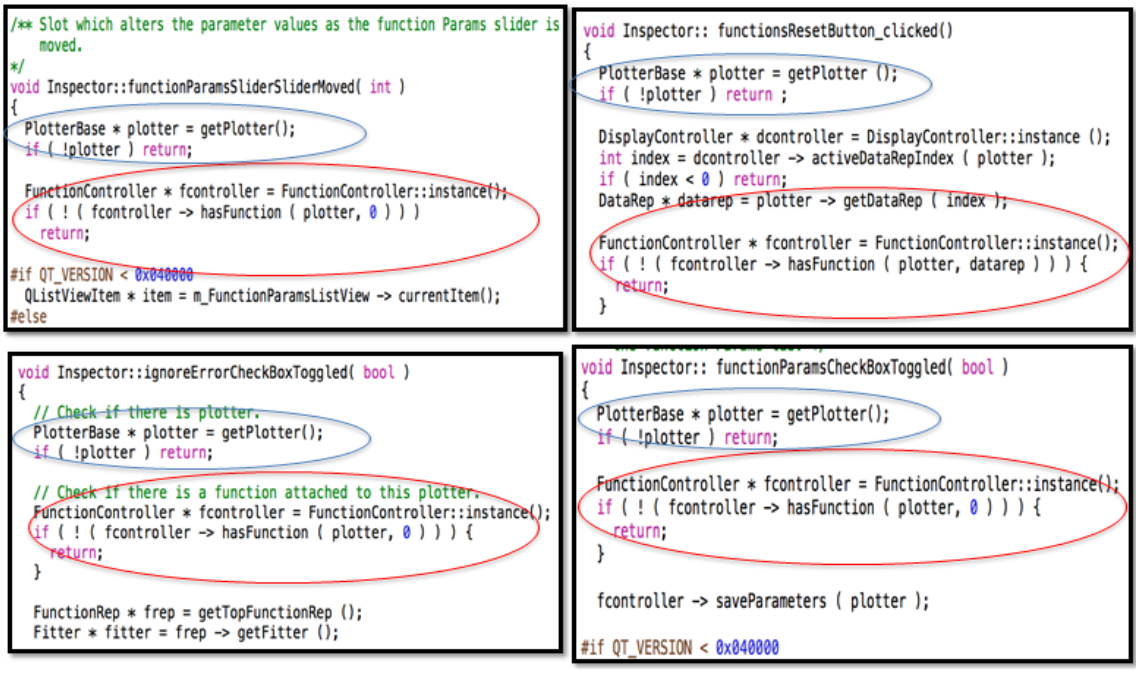
greatly improve programmer productivity in the context of correctly using an API or similar library.

7.2. Future work

This work forms for a number of research areas in mining software repository, and we plan to extend our work in the code completion problems and API usages. We plan to build the code completion tool based on the TreeMiner tool result. The proposed code completion tool could be used not only to improve the programming also to improve the use of APIs.

APPENDIX A

SOURCE CODE OF HIPPODRAW



```
/** Slot which alters the parameter values as the function Params slider is
    moved.
*/
void Inspector::functionParamsSliderSliderMoved( int )
{
  PlotterBase * plotter = getPlotter();
  if ( !plotter ) return;

  FunctionController * fcontroller = FunctionController::instance();
  if ( ! ( fcontroller -> hasFunction ( plotter, 0 ) ) )
    return;

#if QT_VERSION < 0x040000
  QListViewItem * item = m_FunctionParamsListView -> currentItem();
#else
```

```
void Inspector:: functionsResetButton_clicked()
{
  PlotterBase * plotter = getPlotter ();
  if ( !plotter ) return ;

  DisplayController * dcontroller = DisplayController::instance ();
  int index = dcontroller -> activeDataRepIndex ( plotter );
  if ( index < 0 ) return;
  DataRep * datarep = plotter -> getDataRep ( index );

  FunctionController * fcontroller = FunctionController::instance();
  if ( ! ( fcontroller -> hasFunction ( plotter, datarep ) ) ) {
    return;
  }
```

```
void Inspector::ignoreErrorCheckBoxToggled( bool )
{
  // Check if there is plotter.
  PlotterBase * plotter = getPlotter();
  if ( !plotter ) return;

  // Check if there is a function attached to this plotter.
  FunctionController * fcontroller = FunctionController::instance();
  if ( ! ( fcontroller -> hasFunction ( plotter, 0 ) ) ) {
    return;
  }

  FunctionRep * frep = getTopFunctionRep ();
  Fitter * fitter = frep -> getFitter ();
```

```
void Inspector:: functionParamsCheckBoxToggled( bool )
{
  PlotterBase * plotter = getPlotter();
  if ( !plotter ) return;

  FunctionController * fcontroller = FunctionController::instance();
  if ( ! ( fcontroller -> hasFunction ( plotter, 0 ) ) ) {
    return;
  }

  fcontroller -> saveParameters ( plotter );

#if QT_VERSION < 0x040000
```

Figure 22. HippoDraw Result

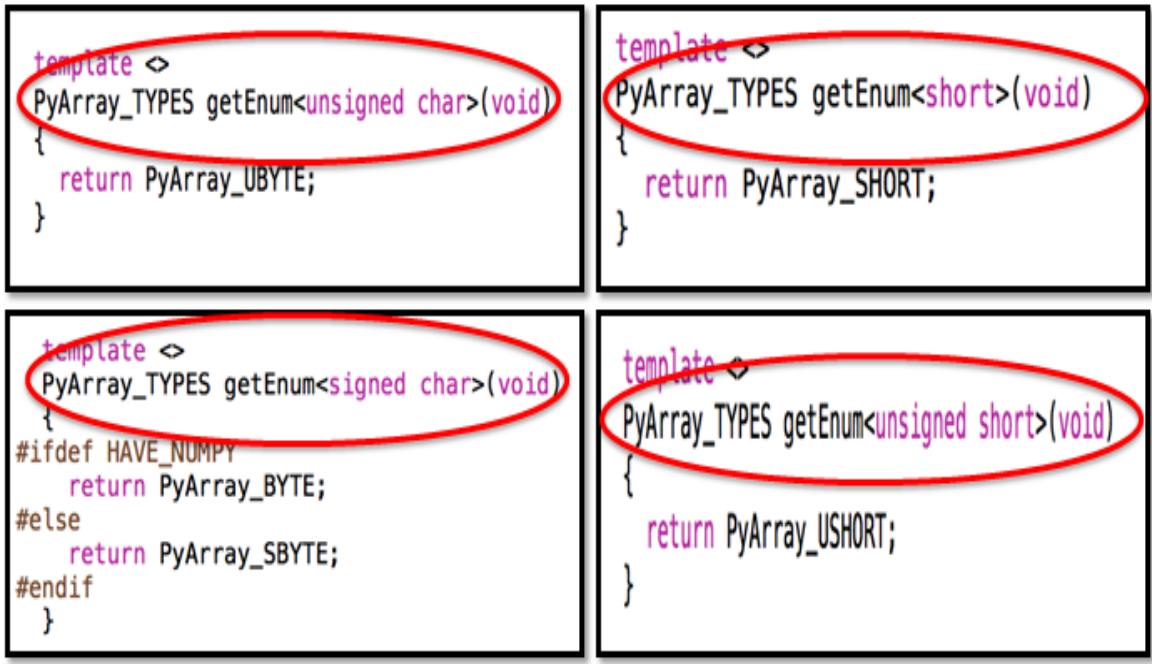
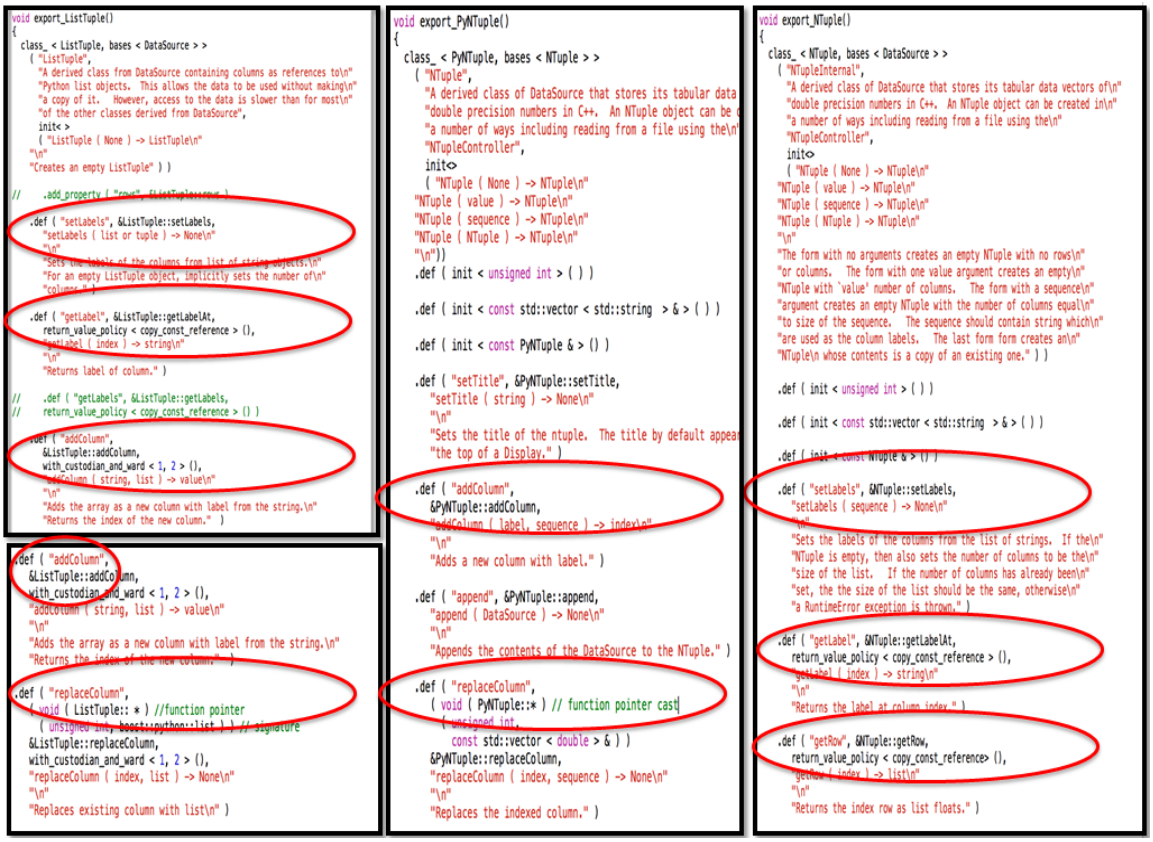


Figure 23. Sample of HippoDraw Source Code



```
void export_ListTuple()
{
  class_ < ListTuple, bases < DataSource > >
    ( "ListTuple",
      "A derived class from DataSource containing columns as references to\n"
      "Python list objects.  This allows the data to be used without making\n"
      "a copy of it.  However, access to the data is slower than for most\n"
      "of the other classes derived from DataSource",
      init< >
      ( "ListTuple ( None ) -> ListTuple\n"
    "\n"
    "Creates an empty ListTuple" ) )

//     .add_property ( "rows", &ListTuplessrows )

    .def ( "setLabels", &ListTuple::setLabels,
       "setLabels ( list or tuple ) -> None\n"
       "\n"
       "Sets the labels of the columns from list of string objects.\n"
       "For an empty ListTuple object, implicitly sets the number of\n"
       "columns." )

    .def ( "getLabel", &ListTuple::getLabelAt,
       return_value_policy < copy_const_reference > (),
       "getLabel ( index ) -> string\n"
       "\n"
       "Returns label of column." )

//     .def ( "getLabels", &ListTuple::getLabels,
//     return_value_policy < copy_const_reference > () )

    .def ( "addColumn",
       &ListTuple::addColumn,
       with_custodian_and_ward < 1, 2 > (),
       "addColumn ( string, list ) -> value\n"
       "\n"
       "Adds the array as a new column with label from the string.\n"
       "Returns the index of the new column."  )
```

```
.def ( "addColumn",
  &ListTuple::addColumn,
  with_custodian_and_ward < 1, 2 > (),
  "addColumn ( string, list ) -> value\n"
  "\n"
  "Adds the array as a new column with label from the string.\n"
  "Returns the index of the new column."  )

.def ( "replaceColumn",
  ( void ( ListTuple:: * ) //function pointer
    ( unsigned int, boost::python::list ) ) // signature
  &ListTuple::replaceColumn,
  with_custodian_and_ward < 1, 2 > (),
  "replaceColumn ( index, list ) -> None\n"
  "\n"
  "Replaces existing column with list\n" )
```

```
void export_PyNTuple()
{
  class_ < PyNTuple, bases < NTuple > >
    ( "NTuple",
      "A derived class of DataSource that stores its tabular data
      "double precision numbers in C++.  An NTuple object can be
      "a number of ways including reading from a file using the\n"
      "NTupleController",
      init<>
      ( "NTuple ( None ) -> NTuple\n"
    "NTuple ( value ) -> NTuple\n"
    "NTuple ( sequence ) -> NTuple\n"
    "NTuple ( NTuple ) -> NTuple\n"
    "\n"))
    .def ( init < unsigned int > ( ) )

    .def ( init < const std::vector < std::string  > & > ( ) )

    .def ( init < const PyNTuple & > () )

    .def ( "setTitle", &PyNTuple::setTitle,
       "setTitle ( string ) -> None\n"
       "\n"
       "Sets the title of the ntuple.  The title by default appea
       "the top of a Display." )

    .def ( "addColumn",
       &PyNTuple::addColumn,
       "addColumn ( label, sequence ) -> index\n"
       "\n"
       "Adds a new column with label." )

    .def ( "append", &PyNTuple::append,
       "append ( DataSource ) -> None\n"
       "\n"
       "Appends the contents of the DataSource to the NTuple." )

    .def ( "replaceColumn",
       ( void ( PyNTuple::* ) // function pointer cast
         ( unsigned int,
           const std::vector < double > & ) )
       &PyNTuple::replaceColumn,
       "replaceColumn ( index, sequence ) -> None\n"
       "\n"
       "Replaces the indexed column." )
```

```
void export_NTuple()
{
  class_ < NTuple, bases < DataSource > >
    ( "NTupleInternal",
      "A derived class of DataSource that stores its tabular data vectors of\n"
      "double precision numbers in C++.  An NTuple object can be created in\n"
      "a number of ways including reading from a file using the\n"
      "NTupleController",
      init<>
      ( "NTuple ( None ) -> NTuple\n"
    "NTuple ( value ) -> NTuple\n"
    "NTuple ( sequence ) -> NTuple\n"
    "NTuple ( NTuple ) -> NTuple\n"
    "\n"
    "The form with no arguments creates an empty NTuple with no rows\n"
    "or columns.  The form with one value argument creates an empty\n"
    "NTuple with `value' number of columns.  The form with a sequence\n"
    "argument creates an empty NTuple with the number of columns equal\n"
    "to size of the sequence.  The sequence should contain string which\n"
    "are used as the column labels.  The last form form creates an\n"
    "NTuple\n whose contents is a copy of an existing one." ) )

    .def ( init < unsigned int > ( ) )

    .def ( init < const std::vector < std::string  > & > ( ) )

    .def ( init < const NTuple & > () )

    .def ( "setLabels", &NTuple::setLabels,
       "setLabels ( sequence ) -> None\n"
       "\n"
       "Sets the labels of the columns from the list of strings.  If the\n"
       "NTuple is empty, then also sets the number of columns to be the\n"
       "size of the list.  If the number of columns has already been\n"
       "set, the the size of the list should be the same, otherwise\n"
       "a RuntimeError exception is thrown." )

    .def ( "getLabel", &NTuple::getLabelAt,
       return_value_policy < copy_const_reference > (),
       "getLabel ( index ) -> string\n"
       "\n"
       "Returns the label at column index." )

    .def ( "getRow", &NTuple::getRow,
       return_value_policy < copy_const_reference> (),
       "getRow ( index ) -> list\n"
       "\n"
       "Returns the index row as list floats." )
```

Figure 24. Def Function Usages

RERERENCES

[Chuntao, Michele 2013] Chuntao Jiang, F. C., Michele Zito (2013). “A survey of frequent subgraph mining algorithms." In Proceedings of the *Knowledge Engineering Review*, vol. Volume 28 no. No.1 , pp. Pages 75-105. Endika Bengoetxea, (2002). The graph matching problem, PhD Thesis.

[Kollias, Sathe, Grama 2014] Kollias, G., Sathe, M., Schenk, O., & Grama, A. (2014). “Fast parallel algorithms for graph similarity and matching”. In Proceedings of the Journal of *Parallel and Distributed Computing*, 74(5), 2400-2410.

[Schaeffern 2007] Schaeffer, S. E. (2007). “Graph clustering”. In Proceedings of the *Computer Science Review*, 1(1), 27-64.

[Lee, Ruan, Jin, Aggarwa 2010] Lee, V. E., Ruan, N., Jin, R., & Aggarwal, C. (2010). “A survey of algorithms for dense subgraph discovery”. In Proceedings of the *Managing and Mining Graph Data* (pp. 303-336). Springer US.

[Yu, Cheng 2010] Yu, J. X., & Cheng, J. (2010). “Graph reachability queries: A survey”. In Proceedings of the *Managing and Mining Graph Data* (pp. 181-215). Springer US.

[Al Hasan, Zaki 2011] Al Hasan, M., & Zaki, M. J. (2011). “A survey of link prediction in social networks”. In Proceedings of the *Social network data analytics* (pp. 243-275). Springer US.

[Xie, Jian 2006] Xie, Tao, and Jian Pei. (2006) "MAPO: Mining API usages from open source repositories." In Proceedings of the 2006 *international workshop on Mining software repositories*, pp. 54-57. ACM, 2006.

[Beyer, Noack 2005] Beyer, D., & Noack, A. (2005). “Clustering software artifacts based on frequent common changes”. *In Program Comprehension*, 2005. IWPC 2005. In Proceedings of the 13th International Workshop on (pp. 259-268). IEEE.

[Cheng et al. 2009] Cheng Hong, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. (2009) "Identifying bug signatures using discriminative graph mining." In Proceedings of the e*ighteenth international symposium on Software testing and analysis*, pp. 141-152. ACM, 2009.

[Dallmeier, Thomas 2007] Dallmeier Valentin, and Thomas Zimmermann. (2007) "Extraction of bug localization benchmarks from history." In Proceedings of the *twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 433-436. ACM, 2007.

[Di et al. 2006] Di Fatta, Giuseppe, Stefan Leue, and Evghenia Stegantova. (2006) "Discriminative pattern mining in software fault detection." In Proceedings of the 3rd international workshop on Software quality assurance, pp. 62-69. ACM, 2006.

[Eichinger et al. 2008] Eichinger, Frank, Klemens Böhm, and Matthias Huber. (2008) "Mining edge-weighted call graphs to localise software bugs." In *Machine Learning and Knowledge Discovery in Databases*, pp. 333-348. Springer Berlin Heidelberg, 2008.

[Eichinger et al. 2010] Eichinger, Frank, Victor Pankratius, Philipp WL Große, and Klemens Böhm. (2010) "Localizing defects in multithreaded programs by mining dynamic call graphs." In Proceedings of the *Testing–Practice and Research Techniques*, pp. 56-71. Springer Berlin Heidelberg, 2010.

[Liu, et al. 2005] Liu, Chao, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. (2005) "SOBER: statistical model-based bug localization." In Proceedings of the *ACM SIGSOFT Software Engineering Notes* 30, no. 5 (2005): 286-295.

[Marcus et al. 2004b] Marcus, A., Sergeyev, A., Rajlich, V., and Maletic, J., (2004b), "An Information Retrieval Approach to Concept Location in Source Code." In Proceedings of 11th IEEE Working Conference on Reverse Engineering (WCRE'04), pp. 214-223.

[Zhong et al. 2009] Zhong, Hao, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. (2009) "MAPO: Mining and recommending API usage patterns." In Proceedings of the *ECOOP 2009–Object-Oriented Programming*, pp. 318-343. Springer Berlin Heidelberg, 2009.

[Kolter, Marcus 2006] Kolter, J. Zico, and Marcus A. Maloof. (2006) "Learning to detect and classify malicious executables in the wild." The Journal of *Machine Learning Research* 7 (2006): 2721-2744.

[Mihai, et al. 2008] Christodorescu, Mihai, Somesh Jha, and Christopher Kruegel. (2008) "Mining specifications of malicious behavior." In Proceedings of the *1st India software engineering conference*, pp. 5-14. ACM, 2008.

[Han et al. 2009] S. Han, D. R. Wallace, and R. C. Miller, (2009)"Code completion from abbreviated input." In Proceedings of the *Automated Software Engineering (ASE*). IEEE Computer Society, 2009, pp. 332-343.

[Nguyen et al. 2012] Nguyen, Anh Tuan, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar Al-Kofahi, and Tien N. Nguyen. (2012) "Graph-based pattern-oriented, context-sensitive source code completion." In Proceedings of the *34th International Conference on Software Engineering*, pp. 69-79. IEEE Press, 2012.

[Little, Miller 2009] Little, Greg, and Robert C. Miller. (2009) "Keyword programming in Java." In Proceedings of the *Automated Software Engineering* 16, no. 1 (2009): 37-71.

[Nguyen et al. 2009] Tung Nguyen, Hoan Nguyen, Nam Pham, Jafar Al-Kofahi, and Tien Nguyen , (2009) “Graph-based Mining of Multiple Object Usage Patterns". Proceedings of the *7th joint meeting of the European software engineering conference* and the *ACM SIGSOFT symposium on The foundations of software engineering*, August 24-28, 2009, Amsterdam, The Netherlands.

[Mohammed Zaki 2005] Zaki, M. J. (2005). “TREEMinER: An Efficient Algorithm for Mining Embedded Ordered Frequent Trees.” In Proceedings of the *Advanced Methods for Knowledge Discovery from Complex Data* (pp. 123-151). Springer London.

[Tairas, Gray 2009] Tairas, R. and Gray, J., (2009), "An information retrieval process to aid in the analysis of code clones." In Proceedings of the *Empirical Software Engineering*, vol. 14, no. 1, pp. 33-56.

[Grant, Cordy 2010] Grant, S. and Cordy, J. R., (2010), "Estimating the Optimal Number of Latent Concepts in Source Code Analysis." In Proceedings of

International Working Conference on Source Code Analysis and Manipulation, pp. 65-74.

[Kagdi, Collard, Maletic 2007] Huzefa Kagdi, M. L. Collard., and Jonathan I. Maletic, (2007) "A survey and taxonomy of approaches for mining software repositories in the context of software evolution." In the Journal of *Software Maintenance and Evolution: Research and Practice (JSME)*, vol. Volume 19, no. No. 2, 2007, pp. Pages 77-131.

[Michail 1999] Michail, A., (1999) "Data Mining Library Reuse Patterns in User-Selected Applications." In Proceedings of the *14th International Conference on Automated Software Engineering (ASE'99)*. Cocoa Beach, Florida, USA: IEEE Press, 1999, pp. pages 24-33.

[Kagdi, Maletic 2006] Kagdi, H., & Maletic, J. (2006, September). “Mining for co-changes in the context of web localization”. In Proceedings of the *Web Site Evolution*, 2006. WSE'06. Eighth IEEE International Symposium on (pp. 50-57). IEEE.

[Michail 2000] Michail, A., "Data Mining Library Reuse Patterns using Generalized Association Rules." In Proceedings of the *2000 International Conference on Software Engineering, 2000.* Limerick: IEEE 2000, pp. Pages 167 - 176.

[Miltiadis, Charles 2013] Miltiadis Allamanis , Charles Sutton, (2013) “Mining source code repositories at massive scale using language modeling.” In Proceedings of the *10th Working Conference on Mining Software Repositories*, May 18-19, 2013, San Francisco, CA, USA

[Collard, Decker, Maletic 2011] Collard, Michael L., Michael John Decker, and Jonathan I. Maletic. (2011) “Lightweight transformation and fact extraction with the srcML toolkit”. In proceeding of *Source Code Analysis and Manipulation* (SCAM), 2011 11th IEEE International Working Conference on, pp. 173-184. IEEE, 2011.

[Kagdi, Maletic, Sharif 2007] Kagdi, H., Maletic, J.I., Sharif, B., (2007) "Mining Software Repositories for Traceability Links." In the Proceedings of the *15 IEEE International Conference on Program Comprehension* (ICPC 2007), Banff Canada, June 26-29, 2007, pp. 145-154.