AN APPROACH FOR MEASURING THE SOFTWARE MODULARITY BASED ON THE BURSTY EVOLUTION OF FUNCTIONAL DEPENDENCIES

AJAY RAJ TEDLAPU
Bachelor of Technology, GITAM University, 2015

A thesis submitted
in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

AN APPROACH FOR MEASURING THE SOFTWARE MODULARITY BASED ON THE BURSTY EVOLUTION OF FUNCTIONAL DEPENDENCIES

AJAY RAJ TEDLAPU

Date of Defence: December 13, 2019

Dr. Daya Gaur Thesis Supervisor	Professor	Ph.D.
Dr. Shahadat Hossain Thesis Examination Committee Member	Professor	Ph.D.
Dr. Robert Benkoczi Thesis Examination Committee Member	Associate Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

To my parents, brother and my fiancee.

Abstract

Modular Design of a software system is one of the parameters which defines the complexity of a software system. If the software is built as one whole module, then it makes testing a long process. Also, updating the software will make a significant impact on the whole system code because of the dependencies.

We propose a methodology to study and visualize the evolution of the modular structure of a network of functional dependencies in a software system. We used the Understand C++ tool for analyzing the dependencies and Gephi to produce the network. Our method analyzes the modularity of the software and identifies specific periods of significant activities, which are known as the evolutionary hot spots in software systems. As a case study, we analyzed the modular structure of Octave during its life cycle beginning from 1993 to the present.

Acknowledgments

I would like to convey my gratitude to my supervisor Dr. Daya Gaur, who guided and motivated me throughout my research. I also want to thank my committee members Dr. Shahadat Hossain and Dr. Robert Benkoczi for their constant support and motivation. I would also like to thank Dr. Muhammad Khan for guiding me in my research.

I would like to thank Dr. Howard Cheng for being the Examination chair of my thesis defense. I am grateful to SGS, Dr. Daya Gaur for helping me financially throughout my Masters in Canada.

It is a great pleasure to thank my parents and my brother for encouraging me throughout my life. Special thanks to my fiancee Abha Goel for her support and understanding me.

Finally, I would like to thank my friends Shahul Shaik, Leila Karimi, Akalanka Galappaththi and Disha Devaiya for being part of my journey these two years in Canada.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Software systems play a vital role in every sector such as Banking, Education. In days when software systems did not exist, considerable human resources and time had to be devoted to even simplest of the tasks. For example, to transfer money from one account to another account, one had to go to the respective bank and fill a form. Now, with the help of software systems, we can quickly transfer any amount to any account within minutes, even across different countries. Many such conditions contributed to the rise of software systems.

The complexity of the software system depends on the set of requirements. Based on the requirements, the software system will be designed and released, but the requirements might evolve, such as upgrading the software to the newer version or adding new functionality. Let us suppose we need to add new functionality to the system. If the system has an excellent modular design, then the effect of the change would be restricted to the module in which the change is undergoing. If the system has a poor modular design, then the effect of the change can be the whole system functionality.

Software systems need to be updated regularly based on the available versions and functionality, therefore software systems will undergo continuous changes. If the software is left unmodified for more extended periods, then the software will be outdated in terms of both functionality and also applicability. For example consider the Windows operating system, the first version of the Windows operating system Windows 1.0 was released in 1985. The

Windows 1.0 operating system has been developed based on the requirements as of 1985. Imagine using the Windows 1.0 operating system on the presently available computer system. So any software needs to be updated regularly.

As stated in [32], the software should be consistent with the new changes and also should be adaptable. Software is said to have a good design if the changes made to the software do not impact the entire software. If the changes made in a software impact major areas in software, then the change impact assessment should be made across all the affected areas of the software. Defects can also be introduced in the software during the process of update and change if the design is weak.

A software system is said to be modular if it groups the entities in the entire system into modules based on their functionality such that all the files with similar functionality will be in one module. This reduces the interactions across modules. These interactions are known as dependencies. As stated in [15] to assess the impact of changes made in a software system understanding the dependency relationships is essential. If a file ’A’ uses functionality in file ’B’ then file ’A’ is said to be dependent on file ’B’. A change in file ”B” because of the dependency forces a change in file ”A”. This dependency is called direct dependency. There are also the indirect dependencies where if file ”A” is dependent on file ”B” and file ”B” is dependent on file ”C”. Then file ”A” is indirectly depended on file ”C”. Most of the dependencies in a module comprise indirect dependencies.

For a system to be modular, the elements in the module should be strongly coupled, and the elements across the modules should be weakly connected. The strongly connected components, coupling, and cohesion are metrics to analyze the dependency information of a software system. If two files ’A’ and ’B’ are committed together at the same time, then files are said to have a co-change relation. As stated in [15], the co-change relationships between the files produce the logical dependencies. There are many research works available that help us analyze the modularity of the software system based on metrics like co-change and coupling as well. The evolution of software systems can also be done using design structure matrices and design rule theory, as in [26].

1.2 Motivation

One of the methods for achieving modularity is the use of information hiding. Information hiding groups the design decisions of a particular program that can be changed later and keeps them isolated so that even if the design decision is changed, it will not affect the other parts of the program. Refactoring is the process of restructuring the code and changing the internal behaviour without changing the external behaviour. Refactoring eliminates duplicate code and improves reusability. By Refactoring we can improve software modularity. Information hiding, Refactoring, Encapsulation, Polymorphism affects the modularity of a software system.

By making the system modular, it helps the software developer in debugging the error. Debugging is a process used by a developer to resolve an issue or a defect. Debugging entails analyzing the source code. If the system is not modular, then the developer has to check several files in order to determine the actual cause, which is time-consuming. The developer has to do an impact assessment in order to confirm that the changes made in order to fix the defect do not affect the rest of the code. This too takes time.

Software modularity also eliminates code repetition. It promotes code reusability, which also decreases the length of the code. If the system is not modular, then the project might become large involving more than one developer on the same project which might cause design problems because it is recommended to have one developer handle at least one complete functionality. The main point that motivates us is the effect of these techniques on the overall design of the system.

In this thesis, for the case study, we have considered GNU Octave software because of its three evolutionary phases. The first phase called the early development phase during the period 1993 to 1997, where the primary developer Eaton is the chief developer. The second phase is the period from 1997 to 2001, where Eaton concentrated more on maintenance than development. The last or the current phase is post-2001, where the transition of GNU Octave to open development has occurred, and several developers are involved. There might be significant changes occurred during every phase that affects the modularity of the system. The main point that motivates us is, whether we can identify the significant changes and if those significant changes improved the modularity of the system.

Requirements determine the complexity of the software systems because the requirements will evolve based on the need for functionality. The various requirements forced Eaton in 1997 to divide the branch in the repository into the staging and production branches, to keep the impact of the immediate changes away from the end-users (until they are thoroughly tested). Once the new changes are available in the staging branch, he spent time merging the completed changes with the production branch. These changes can improve the modularity of the system, or they can decrease the modularity of the system. If the resulting software is not modular, then the whole system is impacted because of the next set of changes in the requirements. Testing the software would take much time as one has to test the impact made by the change in the whole system. We have considered Octave as our case study because Octave underwent many changes and revisions which might have impacted the modularity of the system.

1.3 Contributions

The modularity of the software system in its design is essential. There are several metrics for analyzing the design of the software system based on co-change and functional dependencies. Few design methods like code refactoring, encapsulation also affect the modularity of a software system. We propose a methodology to identify the bursts of these local changes which affect the design of the system over its entire life cycle.

In this thesis, we propose a methodology to extract and visualize the evolution of the modular structure of a network of functional dependencies in a software system. We can also identify the significant changes that lead to a change in the modular structure. We have used the Understand C++ tool for analyzing dependencies and Gephi to produce and visualize the network. Our method analyzes the software and identifies specific periods of significant activities, which are known as the evolutionary hot spots in software systems. Using our methodology, we can identify in GNU Octave the points in time (over the period 1993 to 2019) where significant changes occurred, leading to a change in the modular structure of the software.

Our is a three-phase approach where, in Phase 1, we analyze the software based on the functional dependencies and produce a dynamic network. In Phase 2, we identify the modular structure of the network by using the weighted clustering algorithm. In Phase 3, we extract the arrival and removal event streams from the dynamic network. We assume that the significant effects on the modularity are due to the addition or removal of dependencies. We use Kleinberg's algorithm [20] to identify the bursts based on arrival and removal event streams. We have implemented our methodology and tested in on GNU Octave software over its entire life cycle.

1.4 Organization

There are six chapters in this thesis. The first (this) chapter deals with the introduction, motivation, and contribution. In the second chapter, we introduced the algorithms used in our thesis and described the related work.

In Chapter 3, we have explained our overall methodology and described the three phases in our thesis. For each phase, we also describe the various tools and methods used in the thesis along with the mathematical formulation. In chapter 4, we have explained our case study and the implementation process for each of the three phases.

In chapter 5, we have shared and analyzed the results for each phase. In chapter 6, we have the conclusions for the thesis. The scope of future work is also discussed.

Chapter 2

Preliminaries and Related Work

2.1 Kleinberg's Algorithm

Our main goal is to find the bursts of significant activity and analyze whether these bursts of significant activity lead to changes in the modularity of the system. To find out the bursts of activity, we have used the work done by Kleinberg [20] based on the two-state automaton. Kleinberg's [20] goal is to find outbursts of activity of a topic in several documents. Kleinberg modelled the bursts in the emails by looking for significant events for a specific topic. The bursts are said to occur when there is an increase in the arrival of emails on a particular topic. The behaviour of the bursts of email activity is very unpredictable, sometimes the frequency may be high and sometimes the frequency may be low.

2.1.1 Bursty Model

To find the bursts of activity in email streams for specific topics, Kleinberg modelled the process of generation of the email stream using an Infinite State Automaton. The infinite model has states with probabilities which are directly proportional to the arrival rates of the email for a particular topic. If the state transition moves from low probability state to high probability state, then it is considered as a burst of activity. The probabilities are assigned for the individual states based on the data. The cost of a state transition is also determined using the data. The main advantage of assigning the costs to the state transition is that it helps in finding the longer bursts instead of finding shorter bursts of activity. The high probability states corresponding to the higher intensity of bursts also helps in providing the hierarchical structure in email streams [20].

The arrival of email messages based on the particular topic is considered to be an exponential distribution. As stated in [20] the arrival of messages cannot be predicted and the gap x between the messages m and $m+1$ is distributed according to the density function $f(x) = \alpha e^{-\alpha x}$ where α is greater than zero. That is, the probability of the gap in time between the email message arrivals to be more than x is equal to $e^{-\alpha x}$. Here α is the rate of the email arrivals. The model of bursts has periods of higher α along with the periods of lower α. The bursty model extends the formulation by displaying the periods of lower arrival rates along with the periods of higher arrival rates [20]. Kleinberg constructed the model with multiple states, where the arrival rates of email depends on the current state. The infinite-state model helps us in analyzing the hierarchical structure whereas the two-state model is the basic model.

2.1.2 Two State Model

As stated by Kleinberg [20] the two-state modelling of bursts is the fundamental model where the automaton has two states q_0 and q_1 where q_0 is for low state and q_1 is for the high state for a specific event stream. So, when the automaton is in q_0, then the arrival of messages is slower when compared to automaton in q_1. In between the messages, the automaton undergoes state transition with probability $p \in (0,1)$. The probability that it does not change the state is given by $(1-p)$ for some p.

So the automaton starts in q_0 and changes the state with probability p. The email messages arrive in batches, some of which are relevant to the topic, and some are not relevant. The main goal is to model the bursts based on relevant message streams.

Consider there are n batches of email messages, and the tth batch contains r_t relevant messages out of d_t total messages. Let r = $(r_1, r_2, ..., r_n)$ and $d = (d_1, d_2, ..., d_n)$ and let $R = \sum_{t=1}^{n} r_t$ and $D = \sum_{t=1}^{n} d_t$. For state q_0 the probability of fraction of relevant documents is $p_0 = R/D$ and similarly for state q_1 the probability of fraction of relevant documents is

$p_1 = p_0 s$, where 's' is a scaling parameter.

The cost of the state sequence $q = (q_1, q_2, ..., q_n)$ producing the observed email stream if the automaton is in state q_0 at time 't' is given by

$$\sigma(0, r_t, d_t) = -\log\left(\binom{d_t}{r_t} p_0^{r_t}(1-p_0)^{d_t - r_t}\right)$$

Similarly, the cost function for the automaton to fit in state q_1 at time 't' is given by

$$\sigma(1, r_t, d_t) = -\log\left(\binom{d_t}{r_t} p_1^{r_t}(1-p_1)^{d_t - r_t}\right)$$

There is also a cost assigned for state transition. For a burst between [t_1, t_2], the weight of the burst is given by

$$\sum_{t=t_1}^{t_2} \sigma(0, r_t, d_t) - \sigma(1, r_t, d_t).$$

In [20], they have conducted experiments with an email stream of data. With the help of infinite-state automaton, they were able to extract and analyze the hierarchical structure in email streams.

With the help of Kleinberg's method [20] we can use a two-state automaton that can model our event stream and identify the bursts of significant activity. Based on the two-state automaton and relevant event stream, we can assign a cost for each sequence of state transitions. The goal is to identify the state sequence that best explains the observed events. Given the automaton and the relevant event stream, we need to find the state sequence for the entire batch of the relevant event stream. The minimum cost state sequence is found by using Viterbi's algorithm [35] as described in the next section.

2.2 Viterbi's Algorithm

Viterbi's algorithm is named after Andrew Viterbi. In 1967 Andrew Viterbi proposed the algorithm which is a dynamic programming algorithm for finding the hidden states to

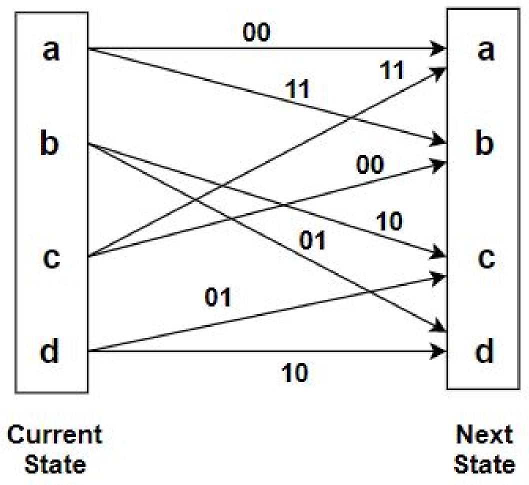


Figure 2.1: Code Trellis

decode convolutional codes. There were several applications of the algorithm; the most crucial application of the Viterbi's algorithm is the maximum likelihood decoding of the convolutionally coded digital sequences over the channel with many distractions or noise.

The objective of Viterbi's algorithm is to identify the best path through the trellis that is closest to the received data bit sequence. The algorithm operates by computing a cost/metric or discrepancy for every possible path in the trellis. The metric for a particular path is defined as the Hamming Distance between the sequence represented by that path and the received sequence. At each state in the trellis, the algorithm compares the paths and chooses the path with lower hamming distance.

Let us consider an example, which will help us understand the Viterbi's algorithm. Let the received data bit sequence is 11 01 10. The code trellis is shown in figure 2.1. The code trellis is a graphical representation of a code where every path represents a codeword.

We start from initial state a_0 then based on the code trellis we have to indicate all the possible paths from current state to next state as shown in the figure 2.2. a_0 is the current state and a_1 is the next state, 00 is the output which will be received for moving from a_0 to a_1. The desired output in the step 1 is indicated in the red color and the hamming distance for each node is calculated and is indicated in blue color. The hamming distance is the

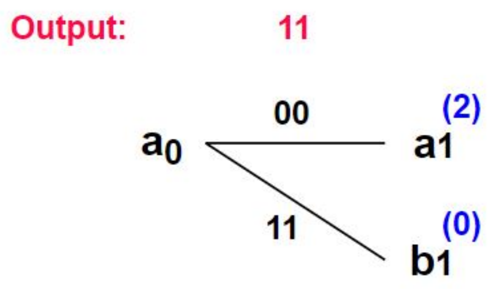


Figure 2.2: Step 1 in Viterbi's Algorithm

metric here which is the difference in the received output bit and the desired output bit. In step 2, from each node a_1, b_1 the possible next states is indicated along with the computed hamming distance in figure 2.3. Similarly, in each step the hamming distance is calculated and is indicated in blue color at the top of each node. At the final step, as shown in the figure 2.3 we received 8 paths and the total hamming distance is calculated for each path and is displayed in the green colour. The viterbi's algorithm chooses the path which has less hamming distance when compared with other paths. So, the path from a_0, b_1, d_2, d_3 has the less hamming distance and if we compared the output received in this path is 11 01 10 which is the received data bit sequence. Viterbi's algorithm computes the metric in each step and chooses the path with lower metric to receive the data bit sequence.

In section 2.1 we have the cost assigned for the states based on the probability distribution and a cost is also assigned for the state transitions. Now the problem is identical to the problem of finding the minimum cost state sequence. We used Viterbi's algorithm [35] to determine the state sequence based on the cost function. For each node or state, the algorithm compares two paths entering the state. The path with the lower metric is retained, and the other is discarded.

2.3 Related Work

The link between class dependencies and co-change is explained with empirical evidence in [13]. The authors make the following point. If dependencies (either logical or

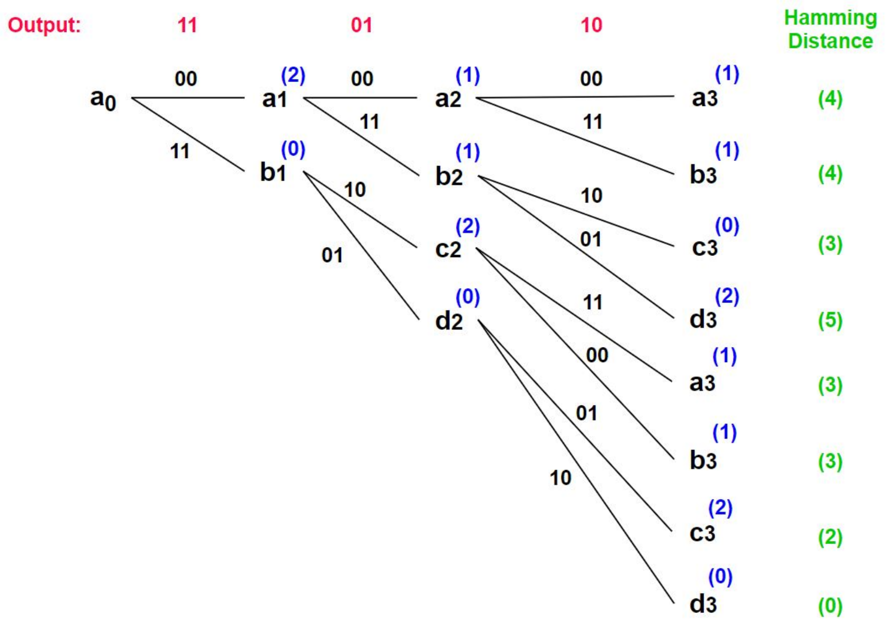


Figure 2.3: Finding the path with minimum hamming distance

functional) connect the modules, then they should have also exhibit co-change behaviour. They have stated that most of the dependencies in the whole system are only due to the few active dependencies. To characterize the refactoring effect, dependencies alone are not enough; we need to see the changelogs of a project. They have also stated the impact of differences on dependent and independent modules. They answered the question of co-change propagation along with the depth of dependency structure with empirical evidence. This paper provides a bias as it shows us that dependencies cause co-change, but at the same time, it also provided results based on the Lorenz curve that the co-changes are only due to small subset of dependencies. It is also not a good idea to remove all the dependencies because they provide reusability of code. So we have to identify particular areas which have more impact on co-change. In order to identify the ”hot spots” (the specific area to change the code structure to reduce dependency) this paper tells us that dependency alone will not suffice, but we also need changelogs (functional dependencies) in a project.

The approach for detecting changes to software modularity is stated in [34]. They have described a few examples where the detection of the software (un)-modularity violation is hard even while testing the software system. Their approach states that if two files are changed simultaneously multiple times due to the requirement changes and if those two files belong to different modules, then there is a modularity violation. They have implemented their approach on two open-source software systems, Hadoop Common and Eclipse JDT. They were able to find out the modularity violations and can show that the system has lousy software modularity. There are several other approaches to detect software modularity violations. However, this paper proved to be one of the best to determine the modularity violations.

In [14], it is stated that it is challenging to determine the change propagation because of the structural dependencies of several classes involved in the software system. They have analyzed four open-source projects coded in Java and determined the impact of structural dependencies over co-change propagation. They have shown that if a file ”A” depends on file ”B” then file ”A” and file ”B” do not need to commit at the same time. However, if file ”A” structurally depends on file ”B” then the two files' files' A” and ”B” might co-change. They have also stated that the structural dependencies alone do not cause co-change propagation. They have observed several other cases which caused co-change propagation.

The software development will undergo many changes over its life cycle, and the changes that affect the system are analyzed in [10]. They have analyzed the data history of the software and compared it with the structural data for software clustering. They have also provided a design that measures the quality of software decomposition. They have worked on projects based on Java language, and their results lead to impact analysis, software and bug prediction.

In [5], they have analyzed and stated whether the software which is developed based on SPL technology has dependencies which in turn cause co-changes along with its dependency structure. They have worked on five software projects which have used SPL technology, and their results clearly described the relation between dependencies and change propagation and stated that on reducing the dependencies the change propagation would not be controlled.

In [7], they have provided a mathematical model and implemented a case study to predict the change propagation based on development structure and impact analysis. They have stated that the number of files which will be affected by a change is directly proportional to the complexity of the software system. It is stated that reduced complexity in a software system might decrease the effect of a change in a software system.

In [27], the design structure matrix is formed by extracting the dependencies and by applying the requisite DSM algorithms. They described a tool that extracts the dependencies present in a software system. This tool also indicates the defects or the problems associated with the dependency structure, that is the problems that violate the software modularity are also identified. They have presented an approach to maintain the architecture of the complex software systems, and the violations to software modularity have been indicated.

In [16], it is stated that most of the developers do not make a note of the dependency information in a software document or in the form of comments. Only the developer will have an idea about that dependency relation. This paper deals with the logical dependencies among the modules rather than static dependencies. They have described a methodology where they have analyzed the historical process of the code development involved. They have measured the coupling of the system based on the revision history.

In [22], they have defined the propagation cost and the clustering cost of the design structure matrix based on functional dependencies. Based on the type of dependencies, the clustering is then performed. Another design structure matrix approach for determining the co-change modularity of a software system using commit logs is in [4]. The authors pointed out the drawbacks with the propagation cost stated in [22] and described the weighted propagation cost and weighted clustering cost to measure the co-change modularity of the software systems. They have also implemented their methods as a case study on GNU Octave. It was successful in determining the co-change modularity of GNU Octave over its entire life cycle. They have used co-change dependencies but did not consider functional dependencies. The relation between the co-change and functional dependencies is described in [1].

In our thesis to identify the bursts in Phase 3, we have used Kleinberg's method [20]. In his paper, Kleinberg presented a methodology for modeling bursts of occurrence of a specific topic in the email stream so that they can be easily identified. Kleinberg's motive is to properly organize his personal email as it is overloaded with many numbers of emails. There were many research papers involved in text indexing to develop email interfaces that can organize the mails according to the topic. Kleinberg modeled the bursts of activity for the occurrence of a particular topic in a document stream using an infinite-state automaton. A burst of activity is identified if there is a state transition from a lower state to a higher state. Using an infinite state model, they can extract the hierarchical structure of the emails from the bursts. Kleinberg proposed a two-state model where he assigned a cost to each state sequence according to the binomial distribution with the probability of occurrence of relevant emails. The problem of finding the bursts is now equal to the problem of finding the minimum cost state sequence. Given the state sequence, when the automaton is in high state corresponds to the burst of activity.

Kumar et al. in [21] extended the work done by Kleinberg [20] by analyzing the community structure of blog space the information from blog pages. They also proposed a methodology to identify the communities by pruning and extracting bursty communities. In their results, they have identified the community structure, and they found out that blogs that give rise to the communities are significantly more long-lasting than a typical blog.

Qi He et al. [29] proposed a new temporal representation of text streams using bursty features. The bursty text representation is different from a regular text representation. As the bursty text dynamically represents the documents over time. In their model, the burst is considered as a significant amount of text content over a particular topic. The representation of the document is also dynamic as it depends on its publication date. Their work is motivated by research in the field of Topic Detection and Tracking (TDT). They have two main steps, bursty feature identification, and bursty feature representation. They have used Kleinberg's [20] two-state model in order to identify the bursty features. They have classified the states based on the emission rate for each feature. They have computed the burstiness of each feature across all topics.

Andreas et al. [2] proposed a model to evaluate two separate tasks. Data Association is one of the tasks where a topic is assigned to each data point and Intensity tracking models the bursts and changes in intensities of topics over a period of time. They proposed an extension of Factorial Hidden Markov Model for the topic intensity tracking for data association. They state that in Kleinberg's model [20] the data association and burst detection are viewed as two separate tasks and that could add bias to the model. They presented an approach that combines the two tasks of associating data and tracking of intensity into a single model.

Kleinberg's model of identifying bursts is used for identifying the bursts of arrival of documents over time for a particular query in [25]. Whenever we ran a query the result will be comprised of documents related to the query along with the documents that are not relevant to the query. They classify the data stream as either bursty i.e the automaton in high state and non-bursty when the automaton is in low state which depends on whether the result contains more documents relevant to the query or less documents relevant to the query. Cost is assigned whenever there is a state transition from low to high state. The weight of the burst is also computed for a period where the automaton is in high state.

Chapter 3

Methodology

Dependencies play a crucial role in the software system. File ”A” can depend on file ”B” in various ways. File ”A” can call a member variable of file ”B”. File ”A” can use a member function of file ”B”. File ”A” can import file ”B”. File ”A” can extend file ”B”. So a change in file ”B”, due to the requirement changes can cause a change in file ”A”. A software system contains many such dependencies. For a system to be modular, the elements in the module should be ”strongly” connected, and the elements across the modules should be ”weakly” connected.

We produced a network based on the dependency structure of software, obtained from its repository during a specific period. We analyzed the software dependencies using the Understand C++ tool and produced the dependency table for each year during a specified period. We built the dynamic network of the dependencies using the Gephi tool. The dynamic network helps us in visualization and the extraction of the data for a specific period. The nodes in the graph represent files and edges represent the dependency. Labels on the edges represent time intervals. Disjoint time intervals are represented as a list of labels. This is a directed graph, and the direction is essential.

Grouping the functions or files which achieve similar tasks is called a modular grouping [31]. Strongly connected components in the graph as a module. The number of strongly connected components has been used as a measure of the modularity. The higher the number of strongly connected components, the more modular the system. However, this measure has some drawbacks. Therefore, we used the method of MacCornack et al. [22].

This method was recently extended in [4] to handle a weighted clustering cost. We use the method in [4] to determine the modules of the network for each year and the clustering cost (which is a measure of modularity).

Requirements determine the complexity of the software systems. Conditions will keep changing based on their necessity. Whenever there is a new requirement, then we need to add or remove some functionality in our software system. This change can affect system behaviour. We consider two types of events. One is the arrival event stream, and the other is the removal event stream. These events can improve modularity or decrease modularity. Our goal is to identify these significant events and analyze whether these significant changes made during the life cycle of a project have improved the modularity or decreased the modularity.

Ours is a three-phase approach, as shown in Figure 3.1.

- Phase 1: Analyze the software, produce a dynamic network, identify the strongly connected components and visualize it using Gephi.
- Phase 2: Identify the modular structure of the network using the DSM methodology.
- Phase 3: For each module, and each event stream identify the periods of significant activity called bursts during the entire life cycle of the project using the method of Kleinberg.

In Phase 1, we cloned the repository, downloaded the source code, analyzed the source code and produced a dynamic network [36]. In Phase 2, the modules associated with the network are identified. Finally, in Phase 3, we determine the periods in time where significant events occurred over the entire life cycle of the project.

3.1 Phase 1: Dynamic Network using Gephi

The source code to be analyzed is cloned to the local working repository. Understand C++ can examine the source code in various programming languages like C, C++, Java,

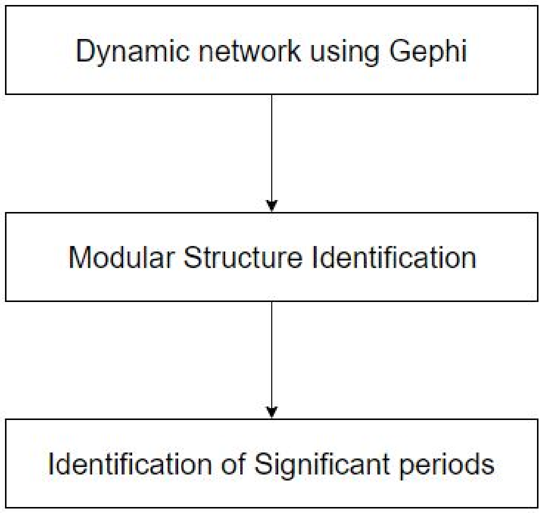
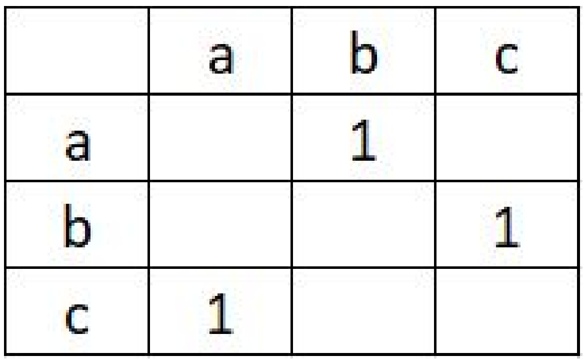

Dynamic network using Gephi
Modular Structure Identification
Identification of Significant periods

Figure 3.1: Phases Involved



	a	b	c
a		1	
b			1
c	1		

Figure 3.2: Example Dependency Matrix Report

Table 3.1: Edges table derived from dependency matrix report

Source	Target
a	b
b	c
c	a

Fortran, Objective-C, Objective-C++. Understand C++ has two types of analyzers, the fuzzy analyzer, and the strict analyzer. The fuzzy analyzer can analyze projects like C, C++, Java except for Objective-C and Objective-C++. For Objective-C and Objective-C++ we have to use the strict analyzer. Consequently, the fuzzy analyzer in Understand C++ is used to analyze the project and generate a dependency report which is a matrix where the rows and columns are files. The methodology followed in this thesis is explained by a simple example where the source code of the software comprises three files a, b and c. The source code is given as input to Understand C++ to analyze. Which generates the dependency matrix report, as shown in Figure 3.2.

The matrix is a 3 x 3 matrix because there are only three files associated with the software. The dependency matrix report indicates the dependencies between the three files. The entry in the second-row third column is 1; this means that the file ’a’ depends on file ’b,’ i.e. file ’a’ calls the member function/variable of file ’b’. Similarly, from the dependency matrix, we can infer that file ’b’ depends on file ’c’ and file ’c’ depends on ’a’. These are the direct dependencies associated with the software. There is also an indirect dependency associated, file ’a’ depends on file ’b’, and file ’b’ depends on file ’c’. So, file ’a’ indirectly depends on file ’c’ because there is a path from file ’a’ to file ’c’ through file ’b’. The edge table associated with the dependency matrix is generated, as shown in Table 3.1.

Now, this edges table is loaded in a Gephi tool to produce a network. In Gephi, we can analyze the network by loading either edge table or node table or the adjacency matrix. In this example, we load the edges table. The network produced from the edge table looks like as shown in Figure 3.3.

The network produced is a static network to it convert into the dynamic network we

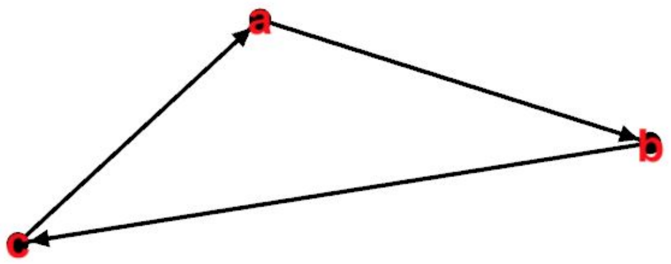


Figure 3.3: Static Network from Edges Table Using gephi

Table 3.2: Edges Table with Start date and End date

Source	Target	Start Date	End Date
a	b	2000/01/01	2000/12/31
b	c	2001/01/01	2001/12/31
c	a	2002/01/01	2002/12/31

need to know the start date and end date of these edges. This information can be gathered from the repository. So after determining the start date and end date, the edge table looks like as shown in Table 3.2.

In Gephi we can load the edge table 3.2 and produce a dynamic network. In a dynamic network, an edge is visible only when the current date is in between the start date and end date of the edge. When the current date is not in between the start date and end date of the edge then the edge is not visible. Dynamic networks help us gather information about the edges present during a certain period. Once the dynamic network is generated, using Gephi, we can extract the edge table for every year during the entire life cycle. Then in phase 2, we will illustrate the methodology used to identify the modular structure.

3.2 Phase 2: Identification of the Modular Structure

There are several methods to identify the modules associated with a network. The detection of communities is done in [30] based on the bipartite cliques. The method followed in [21] has two stages. The first stage is pruning, where the main file or the seed of the community is identified. The second stage is expansion, where they identify the neighbours of the seed and extend the community. The approach used in this thesis is different from all these approaches.

We used the method of weighted clustering described in [4]. It is a greedy algorithm where it improves the solution in each iteration by making the best local move. An entry in adjacency matrix of a directed graph, $(D)_{a,b}$ denotes the number of interactions between two files *a* and *b*. $(D^k)_{a,b}$ is the number of walks between a and b. This approach is used to cluster the DSM's formed in phase one [9].

At the start, each file belongs to its own cluster. Let 't' be the total number of files. Let D the functional dependency matrix. Cost is assigned to every dependency based on whether the two files *a* and *b* belong to the same or different modules. If both the files *a* and *b* belong to the same module *m* then the cost assigned to the dependency is $e^D(a,b) * s(m)^2$ where e^D is the exponential matrix of the functional dependency matrix *D* and $s(m)$ is the number of files in module *m*. If both the files are in different modules, then the cost assigned to the dependency is $e^D(a,b) * t^2$. During the initial step, the sum of the cost of all dependencies is the total cost. This algorithm in [4] improves the local move in every iteration until no local move can further decrease the cost beyond the threshold. The resultant is a clustering matrix where we can find the module information of each file.

3.3 Phase 3: Identifying the bursts of significant activity

In phase 1, we have produced the DSMs, and to these DSMs, we have applied the clustering algorithm in phase 2 to identify the modular structure. We have a dynamic network where significant changes have occurred during a few years, and we intend to find the bursts in time where these significant events occurred during the entire life cycle using Kleinberg's approach. This process of community extraction and burst analysis was also done in [21]. However, we use DSMs to cluster as opposed to the method of community detection.

In software based on the requirement, there will be many changes like adding a file, removing a file, creating or removing a dependency. We are treating the addition or removal of a dependency in the software network as meaningful events. In this phase, we will describe the methodology of how to identify the bursts in time where these significant events occur.

The bursts of the events occur interspersed with the periods where this is no activity, i.e. the bare distributed over the entire life cycle. We have to notice that events that occur in different modules might be occurring at the same time or at different times. In case of events that happen simultaneously in modules 1 and 2, we have to be able to detect the most significant activity that happened among both the modules. In this thesis, we have considered the addition of dependencies as one event stream and removal of dependencies as another event stream. For both of these event streams based on the modular structure identified in phase 2, we have to find out the periods in time where these significant changes occurred. We have to analyze whether these changes have made the software system more modular.

Before describing the method that we have used in our thesis, let us quickly review a simple method to identify the bursts. One way of identification of bursts is through a threshold. We can plot a graph based on the activity in every module over the entire time period and then decide on a threshold. The period where the activity has crossed the threshold in the module, forms as a burst. We can apply this method of identification of burst to each module and each event stream (arrival and removal). But the main problem associated with this technique is the choice of the threshold. It will be challenging to identify the burst when the rate of arrival stream or removal stream varies significantly.

3.3.1 Kleinberg's Method

For phase 3, we have used Kleinberg's method [20]. They have proposed a method to model bursts in such a way they can be easily detected. Kleinberg's [20] approach is based on using an infinite-state automaton for modelling the stream of activity where the bursts appear as transitions to specific states. In their approach, the automaton has states that represent the steady-state and a bursty state of events. They have also proposed a two-state model.

The two-state model is the basic model where the automaton has two states which represent low and high volumes of the events. When the automaton is in the low state, then there will be no significant changes to the event rate. But when the automaton is in high state there will be significant change to the event rate. Costs are associated with state transitions. Now our problem is to find a state sequence that minimizes the cost and best explains the observed sequence of events.

They have also proposed the infinite state model for capturing the bursts of higher significant activity with very fewer gaps occurring over the time interval. Here the states are defined based on the gaps between the occurrence of bursts. The costs are assigned in an infinite-state model, similar to the two-state model. Now identical to the two-state model, the problem is of finding the state sequence which minimizes the cost while explaining the observed events.

They have also conducted experiments with an email stream of data and published their findings. With the help of infinite-state automaton, they have been able to extract and analyze the hierarchical structure in email streams.

In this thesis, we are concerned only with the arrival and removal of dependencies as events. Our goal is to find the bursts of significant arrival or removal activity over its entire life cycle. As stated by Kleinberg [20] the two-state modelling of bursts is the fundamental model where the automaton has two states q_0 and q_1 where q_0 is the low state and q_1 is the high state. So, when the automaton is in q_0, the arrival or removal of dependencies is slower compared to when the automaton in q_1. The automaton undergoes state transition with probability $p \in (0,1)$. The probability that it does not change the state is given by $(1-p)$.

So the automaton starts in state q_0 and changes the state with probability p. The arrival or removal of dependencies happens in batches, of which some are relevant dependencies, and some are not relevant dependencies. If both the files in a dependency belong to the same module, then it is recorded as a relevant dependency, irrelevant otherwise. The main goal is to model the bursts based on relevant message streams.

Consider there are n batches of arrival dependencies, and the tth batch contains a_t relevant arrival dependencies out of d_t total dependencies. Let $a = (a_1, a_2, ..., a_n)$ and $d = (d_1, d_2, ..., d_n)$ and let $A = \sum_{t=1}^{n} a_t$ and $D = \sum_{t=1}^{n} d_t$. For state q_0 the probability of fraction of relevant arrival dependencies is $p_0 = A/D$ and similarly for state q_1 the probability of fraction of relevant arrival dependencies is $p_1 = p_0 * s$ where $s > 1$. The parameters p_0 and p_1 corresponds to thresholds for generating relevant arrival or removal dependencies in small and large numbers respectively.

The cost function for the fit of the model in state q_0 at time 't' is given by

$$\sigma(0, a_t, d_t) = -\log\left(\binom{d_t}{a_t} p_0^{a_t} (1-p_0)^{d_t - a_t}\right)$$

Similarly, the cost function for the automaton in state q_1 at time 't' is given by

$$\sigma(1, a_t, d_t) = -\log\left(\binom{d_t}{a_t} p_1^{a_t} (1-p_1)^{d_t - a_t}\right)$$

There is also a cost assigned for state transition. For a burst between [t_1, t_2], the weight of the burst is given by

$$\sum_{t=t_1}^{t_2} \sigma(0, a_t, d_t) - \sigma(1, a_t, d_t).$$

The cost is assigned for each module and every year for the arrival and removal event streams.

With the help of Kleinberg's method [20] we can use a two-state automaton that can model our event stream and identify the bursts of significant activity. Based on the two-state automaton and event stream, we can assign the costs to each state and also for the state transitions. Now we have to find the state sequence which minimizes the total cost, described in the next section.

3.3.2 Minimum cost state sequence

In section 3.3.1 we have the cost assigned to the states based on the probability distribution and the cost is also assigned for the state transitions. Now the problem is identical to the problem of finding the minimum cost state sequence. We used Viterbi's algorithm [35] to determine the state sequence based on the cost function.

This algorithm operates by computing a metric or discrepancy for every possible path. For each node or state, the algorithm compares two ways of entering the state. The path with the lower metric is retained, and the other is discarded. The pseudo-code is shown below.

Algorithm 1: Minimum Cost state sequence

```
Initially at time t=1, the cost matrix C[0,1] = S[0,1] and C[1,1] = S[1,1].
TC = state transition cost.
B = All the values in back pointer matrix is initialized with one.
for $t \leftarrow 2$ to $N$ do
    C[0,t] = min(C[0, t-1] , C[1, t-1]) + S[0, t];
    if C[0, t-1] > C[1, t-1] then
        B[0,t] = 2;
    end
    C[1,t] = min(C[0, t-1] + TC , C[1, t-1]) + S[1, t];
    if C[0, t-1] + TC > C[1, t-1] then
        B[1,t] = 2;
    end
end
```

The algorithm starts the state sequence with state zero. Here $S[i,t]$ is the variable that stores the fit of the model for that particular state, where i denote the state 0 or state 1 and t denotes the time, which ranges from (1 to n). Similarly, $C[i,t]$ is the variable that stores the minimum cost based on the previous time $(t-1)$. Given the metric for each state and state transition, it compares the metric of the previous state at time $t-1$ and decides the next state by choosing the lower metric among the possible paths. In our thesis, the metric is the cost of the model to be in that particular state.

3.3.3 Mathematical Formulation

Let q_0 and q_1 represent low state and high state, respectively. Let p_0 be the probability of occurrence or removal of a fraction of relevant dependencies in q_0. Let (1-p_0) be the probability of occurrence or removal of a fraction of irrelevant dependencies in state q_0. Let us choose two parameters s, γ both greater than zero where s is a scaling parameter which controls the resolution and γ controls the rapid change of states. Generally, γ will be set to a default value of 1.

We define, $p_0 = D_t/T_t$, where D_t is the set of dependencies added/removed to the module at time t and T_t is the total number of dependencies added or removed at time t. The probability that relevant dependencies are added or removed in state q_1 is sp_0.

The fit of the model in the state q_i is defined as

$$\sigma(i, D_t, T_t) = -\log\left(\binom{T_t}{D_t} p_i^{D_t}(1-p_i)^{T_t-D_t}\right)$$

Let n be the number of batches. The cost of transitioning across states from q_0 to q_1 is defined as

$$cost(q_0, q_1) = \gamma \log n$$

The cost of the remaining state transitions are zero. The weight of a significant burst in the interval $[a, b]$ is defined as

$$\sum_{t=a}^{b} \sigma(0, D_t, T_t) - \sigma(1, D_t, T_t).$$

Using the weight of the bursts, we can determine the relative order of the occurrence of the bursts. The calculation of the cost of the paths is done separately for both the arrival stream and removal stream. The cost is assigned for the fit of the model in a particular state i.e either q_0 or q_1 as in Kleinberg's algorithm [20]. Then with the help of Viterbi's algorithm [35], the minimum cost state sequence is found, and the weight of the bursts is calculated to determine the relative order of the bursts across different modules. So the identification of the bursts is done with the help of Kleinberg's algorithm [20]. This helps us to identify the bursts and analyze them for any significant activity and whether it has improved the system modularity or it decreased the system modularity. This two-stage process of identification of community and then bursts is also done based on several activities in social networks in [21]. In our thesis, detection of modularity is done based on [4] by the weighted clustering algorithm.

For the better understanding of the methodology, consider an example where the number of batches (n) = 10, s = 2.0 and $\gamma = 1$. For a Module M, let us consider the arrival of relevant dependencies as A = [3, 6, 2, 8, 1, 1, 2, 1, 1, 3] , and the total number of dependencies as T = [16, 20, 10, 10, 15, 12, 14, 12, 11, 10]. As shown in the figure 3.4, the top row corresponds to the low state (q_0) and bottom row corresponds to high state (q_1). The sigma values $\sigma(i, A_t, T_t)$ are assigned for each state and for each time (t = 1 to n). The cost is assigned for the state transition from low state (q_0) to high state (q_1), i.e γ log n. Given a state sequence, contiguous time when the automaton is in high state corresponds to a significant burst of dependencies. Now our goal is to determine the state sequence by comparing and selecting the path that has minimum cost.

The algorithm starts from state q_0, computes the cost and updates the cost matrix for each time t. At time t = 1, the the cost value is same as the sigma value for both the states. At time t = 2, the algorithm compares the cost of state 0 with the cost of state 1 in the previous time (i.e, t = 1) and updates the cost with the minimum cost. Similarly, the cost matrix is updated at each time.

In the figure 3.4 the values inside the nodes are the sigma values, the solid edges have cost γ log n and the dashed edges have cost 0. The shortest path is decided based on

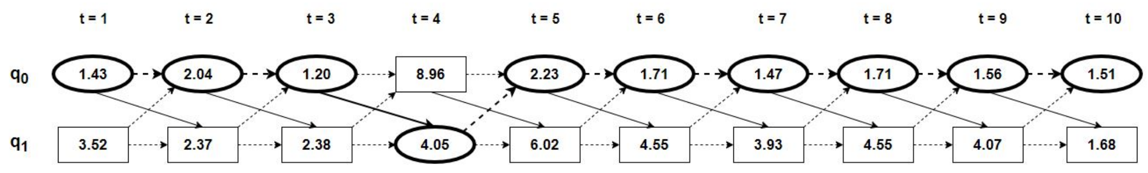


Figure 3.4: The state sequence

minimum cost and is represented by the elliptical shaped nodes. Given the time, state sequence whenever the automaton is in high state corresponds to the significant burst, here in this example the burst of dependencies is at time t = 4. The weight of the burst is 4.91. So for the module M, there is an arrival burst of dependencies at time t = 4.

Chapter 4

Case study and Implementation

4.1 Case Study: Introduction

In this thesis, we have used GNU Octave for the case study. GNU Octave is free, the popular software that is used for numerical computations. Octave is coded in C++. Octave is similar to MATLAB, and the syntax is matrix-based and supports different operations on matrix. Octave was introduced in the year 1988. It was associated with an undergraduate-level textbook in Chemistry at the University of Texas [8]. Eaton and others built some specialized tools for solving the design problems for chemical reactors. It was coded Fortran. They noticed that it is difficult to debug and analyze the code in Fortran. This motivated them to build an easily understandable and flexible language, Octave.

The design was kept compatible with MATLAB because they thought it would help the users right away, without the need to learn a new language [8]. The first full-time development of the Octave started in January 1992. The first release was on January 4, 1993. From then on Octave has undergone many releases and revisions.

John W. Eaton was the primary developer of Octave in the initial period. Eaton started working on Octave interpreter on February 20, 1992. Basic functionality was added to the initial version. The first version was available on January 4, 1993. Eaton was the only developer during the initial phase. After the release, in the next year, users started giving comments for new features and some reported bugs. The next public release was on January 12, 1995. After the release in January 1995, there was not much functionality added in that same year because Eaton concentrated on project cleanups [8].

Eaton divided the production branch into two branches in 1997, one as a staging branch and another one for production. So that the addition of the new functionality does not affect the main production branch. All the new changes are kept in the staging branch or the development branch. In May 1997, Eaton decided to make Octave as a part of the GNU project. On December 7, 2000, Eaton stepped aside from the development of Octave and made it an open-source software where any developer can contribute to the development of the software. From 2001 till present GNU Octave has undergone many releases with many people contributing to the project around the world.

The life cycle of GNU Octave can be classified into the following three stages [4].

1. Initial Phase: This is the start phase of the project. Eaton alone was the developer in this stage. This phase is from 1993 to 1997.

2. Maintenance Phase: In 1997 Octave joined the GNU project and became GNU Octave. Eaton mainly concentrated on stabilizing the software in this stage. This phase lasted from1997 to 2001.

3. Open source development: In 2001 Eaton decided to step away from Octave. He made it an open-source software so that contributors across the world can develop the software. This phase is from 2001 to the present.

The three phases above are from [4]. They analyzed the commit logs and using the co-change DSM's they were able to distinguish these three phases.

The Octave source code is maintained in a Mercurial repository. The Octave code base is mainly of .c, .cc, .cpp and .h files. In this thesis, we analyzed the Octave source code using functional dependencies. We consider the addition and removal of dependencies as events in our model. Our goal is to find the bursts of significant events over the entire life cycle. We also want to know whether these changes lead to an increase in modularity.

Requirements determine the complexity of the software systems. The requirements also evolve based on the necessity. Because of the frequent changes to the requirements, Eaton in 1997 divided the branch into a staging and production branch. He did not want the system to be impacted because of the new changes. These changes can improve the modularity of the system or can decrease the modularity of the system. If the software is not modular then the whole system is impacted by a single change. Testing the software would also take a lot of time. We use Octave as our case study because it underwent many changes and revisions. We want to study the impact of these changes on the modularity of the system.

4.2 Implementation

In our thesis, we have analyzed the software code of GNU Octave to identify the bursts of significant activity over the entire life cycle to measure its impact on software modularity. Our is a three-phase approach where we analyze the software and produce a dynamic network in the first phase, identify the modular structure of the network in the second phase and the last phase, we identify the bursts of significant activity over the entire life cycle of the project. In chapter 3, we have discussed the entire process. In this chapter, we will describe the implementation of each phase. We also describe the usage of tools like Understand C++ and Gephi for code analysis and graph visualization in phase 1. For phase 2, we describe the clustering algorithm used for the identification of the modular structure of the entire software. For phase 3, we describe the implementation of Kleinberg's method and Viterbi's algorithm for identifying the significant bursts of activity.

4.2.1 Phase 1: Dynamic network using gephi

We used Understand C++ tool to analyze the dependency structure of the code. We then used the Gephi tool for graph visualization. Phase 1 that produces a dynamic network is shown in figure 4.1 :

- Step 1: Extraction of GNU Octave source code from the repository
- Step 2: Extraction of edge table for network visualization

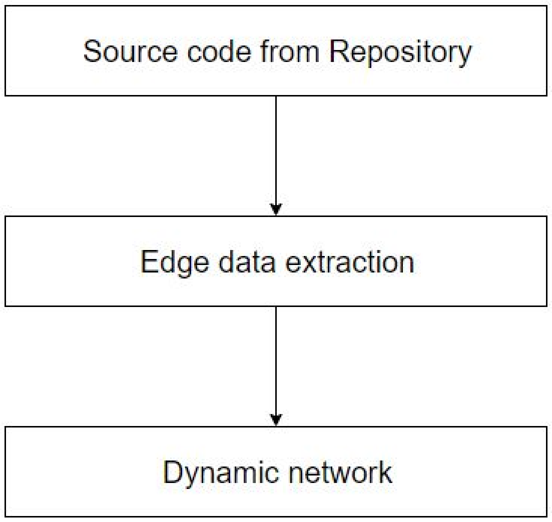


Figure 4.1: Steps involved in Phase 1

- Step 3: Visualization of the dynamic network using Gephi

In Step 1, we cloned the mercurial repository and downloaded the source code of GNU Octave. In Step 2, the edges which represent the dependency between the files are extracted with the help of the Understand C++ tool. Step 3, uses the edge table along with their start date (when the edge is formed) and end date (when the edge is removed). This updated edge table is imported in Gephi to obtain a dynamic network. The details are explained in the further sections.

Step 1: Extraction of GNU Octave source code from repository:

The GNU Octave source code is maintained in Mercurial Repository. Mercurial is free software that is licensed under GNU. There are many other control version systems like GIT, CVS. But Eaton in the year 1997 chose Mercurial as the source control management tool for Octave because it is platform-independent, fast and has distributed architecture, unlike CVS.

The first release of Octave was in the year 1993. From the year 1993 to 2019, we cloned the repository to a local working repository for every month based on the latest revision made in that particular month. For example, consider the January 1993, we search

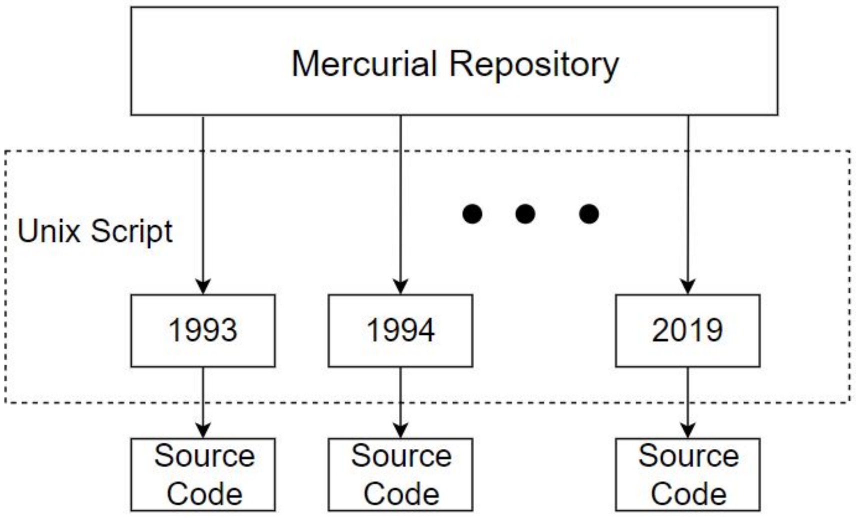


Figure 4.2: Extraction of GNU Octave source code

for the latest revision before January 31, 1993. If there are two revisions made on January 12 and January 25 as we are looking for the latest revision. In the month of January before 31st, the local working repository will be updated with the source code as of January 25, 1993. This process is repeated every month and every year from Aug 1993 to July 2019. We have cloned the repository and extracted the source code for every month because there might be a dependency created in month one, and it may get removed in another month. Such dependencies which are essential might be obscured from the analysis if a coarser unit of time is used. Once we have the source code for every month from 1993 to 2019, the source code forms an input for Step 2. An edge table extraction method is described in the next section.

Step 2: Extraction of edge table for network visualization:

In Step 1, we have cloned the source code of Octave from mercurial repository from August 1993 to July 2019 every month. In total there are 312 versions of the source code pulled from the repository based on revisions made in as many months. Now, we describe how the data is extracted for the network visualization.

Step 2 has four steps, as shown in figure 4.3.

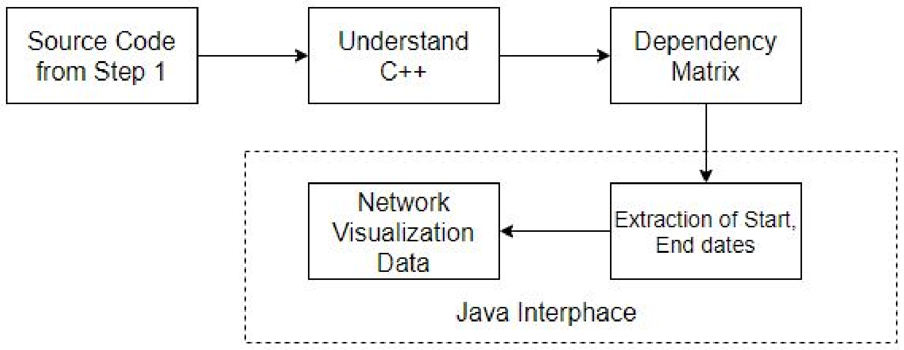


Figure 4.3: Extraction of edge table for network visualization

1. Understand C++: Understand C++ is a tool that helps developers analyze their code and project structure. It helps in project flow visualization and also in debugging. It can perform code analysis and give us a functional dependency report. So, the source code which is extracted from the mercurial repository for every month is uploaded as a specific project in Understand C++ for analysis. We used fuzzy analyzer option of Understand C++ because we have .c, .cc , .cpp and .h files. We generated a dependency matrix report of the project, which is a CSV file. The dependency matrix report has a record of all the functional dependencies that exist in the project. Note that this process is executed for every month from August 1993 to July 2019. So in total, there are 312 dependency matrix reports generated.

2. Dependency Matrix: We have 312 dependency matrix reports. From these dependency matrices, we have to extract the dependency information as a table of all the edges. We have written a java program that reads all these 312 dependency matrices one by one and then produce an edge(dependency) table for each dependency matrix report. Care is taken to ensure that the indices remain unchanged over the months.

3. Extraction of Start and End dates: Given are the 312 edge table starting from August 1993 to July 2019. Now we need to sum up all the edges in these 312 tables and also extract the start date and end date for each edge. Few edges might repeat in every month. We have written a java program to extract the start date and end date of each

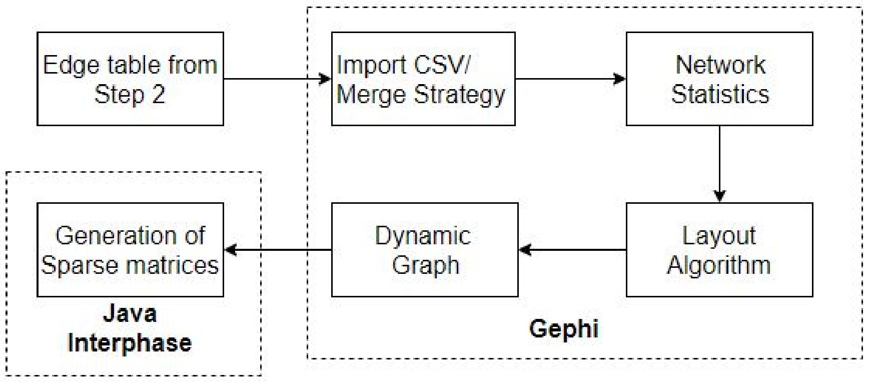


Figure 4.4: Dynamic Network using gephi

edge. For example, if there is an edge between two files a and b in the first month, August 1993, then the start date of the edge is recorded as August 1, 1993. If the same edge is present in the second and third months but not present in the fourth month, then the end date for the edge between the files a and b is recorded as of October 31, 1993, because if the edge is not present in the fourth month. It also happens that an edge can reappear then there will be two entries in the table. So in this process, the java program converts the 312 dependency tables into one single edge table with columns Source, Target, Start Date, and End date.

4. Data for network visualization: So far, the dependency information is extracted to produce one a single file called edge table. This edge table has the complete information of the edges (dependencies) over the entire life cycle of the project and is used to produce a dynamic network using Gephi as described in the next step.

Step 3: Dynamic network using Gephi:

We upload the data in the edge table into Gephi. It will automatically detect the edges and ask us to choose a merge strategy because edges might be repeated. So we have to choose a merge strategy, and we decide not to merge those edges. By default, the repeating edges are merged.

Once we choose the merge strategy, the next process is shown in figure 4.4 option generates the network statistics. In the network statistics tab, we are provided with many operations that can be performed on the network. The main operations are to determine the modularity, strongly connected components, degree, and clustering coefficient. After computing these statistics for the network, we change the colors of the nodes and edges based on a modularity class. We also change the size of the nodes based on the degree. The next step in the process is to apply a layout algorithm. This layout algorithm makes the network clearly visible and prevents the overlap of nodes and edges. Since we have a large network there are few layout algorithms like Openord, YifanHu to choose from.

Once the layout algorithm is applied, the network looks clear. Now to construct the dynamic graph, we have to merge the two columns Start date and End date and then create a time interval. Once we create the time interval, then we will be provided with an option called enable timeline. Once we enable the timeline then in Gephi we can see the timeline from August 1993 to July 2019. We can animate the graph and visualize the evolution of edges over the entire period.

When we set a particular time period, then the edges with interval in the time period are visible, the rest of the edges are not visible. Finally, we generate the sparse matrices which are given as input to phase 2.

For the generation of the sparse matrices for every year, we have to export the edge table from the dynamic network. We set the start date and end date for a particular year then we export the edge table from Gephi as a CSV file which will have the information of edges belonging only to that year.

We have written a java program that will examine all the CSV's generated for each year from 1993 to 2019 and convert the edge table into a sparse matrix. Generally, the node name contains the full directory information, for example, ”octave/src/example.c” . In phase two (where we find the modularity structure) it will be difficult to manipulate the sparse matrix as indexed by these strings. So the sparse matrix is generated by assigning numbers to every node. There are 9256 nodes in total, so numbers from 1 to 9256 are assigned to every node and the number assigned to a node remains the same for all years. The sparse matrix is then generated for each year based on the edge table exported from Gephi. The sparse matrix also contains the information on the number of non-zeros and the dimension of the matrix. This sparse matrix is given as an input to phase 2 for the identification of the Modular structure in the network.

4.2.2 Phase 2: Identification of Modular Structure

We used the method in [4], the weighted clustering algorithm. It is a greedy algorithm where it improves the solution in each iteration by making the best local move possible.

Initially, each file is in its own module. The cost assigned to the files a and b, if both the files are in the same module is $e^D(a,b) * s(m)^2$ where e^D is the exponential matrix of the functional dependency matrix D and $s(m)$ is the number of files in module m. If both the files are in different modules then the cost assigned is $e^D(a,b) * t^2$ where t is the total number of files. At every step, the algorithm tries to reduce the cost by placing the files in different modules. The moves are considered in a random order. The move which improves the cost is considered [4]. The algorithm continues to improve the cost in every iteration, and it stops when the cost cannot be further improved.

The result is also a sparse matrix where it contains the information of the file and its associated module. This resultant sparse matrix is generated for every year. These matrices form the input for phase 3, where we detect the bursts of significant activity. The bursts are identified in phase 3 based on the modular structure of the year 2019 because it is the latest version that is currently in use.

4.2.3 Phase 3: Identifying the bursts of significant activity

In phase 1, we have produced the DSMs, and in phase 2, we have applied the clustering algorithm to find the modular structure. In this phase, we intend to find the bursts of significant activity over the entire time period from 1993 to 2019. These activities include

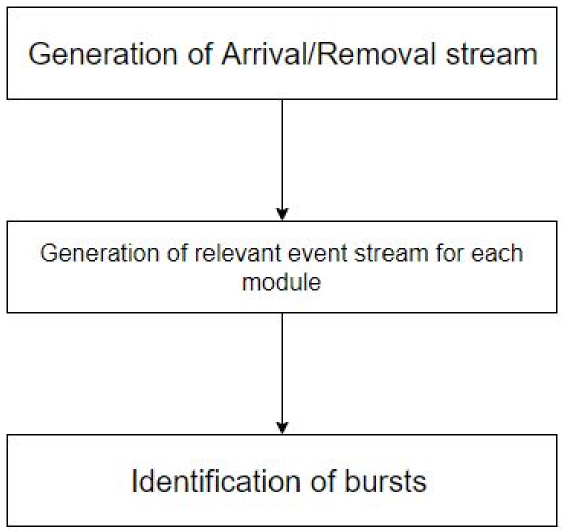

Generation of Arrival/Removal stream
Generation of relevant event stream for each module
Identification of bursts

Figure 4.5: Steps involved in Phase 3

adding or removing a dependency. These bursts are interspersed with the time when there is minimum or no activity. The implementation steps are shown in figure 4.5.

- Step 1: Generation of Arrival stream and Removal stream data.
- Step 2: Generation of relevant event stream for each module.
- Step 3: Identification of the bursts of significant activity

To identify the bursts of significant activity, we need the edge table for every year from phase 1. Based on the edge table we generate the arrival and removal streams in step 1. In step 2, for every module, we generate the relevant event stream and also the total number of events in each year. Finally, in step 3, we apply Kleinberg's algorithm [20] and Viterbi's algorithm [35] to find the bursts of significant activity.

Step 1: Generation of arrival/removal event stream:

We have the edge table from the dynamic network produced in phase 1 for every year from 1993 to 2019.

1. Arrival Stream: We have written a java program where it compares the edge table for the current year with the previous year edge table to find the edges added in the

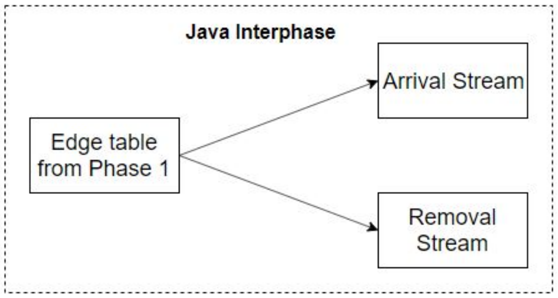


Figure 4.6: Generation of arrival/removal event stream

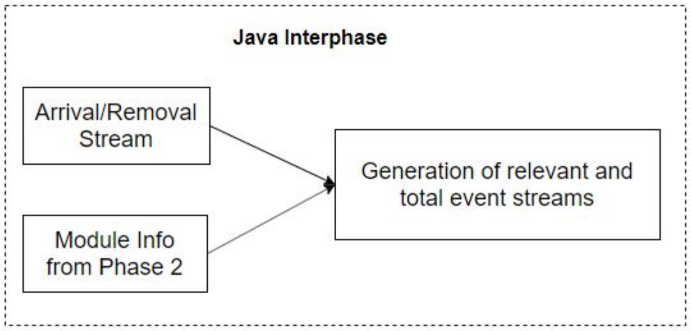


Figure 4.7: Generation of relevant event stream for each module

current year. If the edge is not present in the previous year, then the edge is added to the arrival stream of the next year. The same process is carried out for all the years, and an arrival stream is produced. Arrival stream is a CSV file containing the edge added for that particular year.

2. Removal Stream: For the removal stream, we do similar to the arrival stream. But, we compare the edges in the previous year against the edges in the current year. If an edge is present in the previous year and is not present in the current year, then the edge is added to the removal stream. So the removal stream is also generated for all the years. Removal stream is a CSV file containing the edge removed in that particular year.

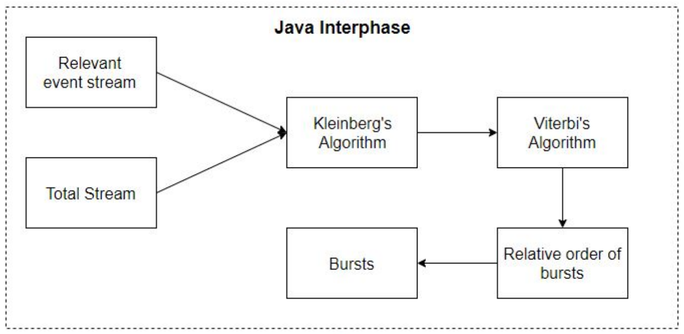


Figure 4.8: Identification of the bursts of significant activity

Step 2: Generation of relevant event stream for each module:

The Arrival stream and Removal stream are used to identify the relevant events and computing the total number of events. We have written a java program to perform the task. In this thesis, we have considered the modular structure of 2019 for identification of bursts.

The arrival/removal stream for each year consists of an edge list. We parse every arrival/removal stream and for each edge, we check whether the two files are in the same module or in a different module. The module information is obtained from the Module info file generated in phase 2. If both the files are in the same module, then it counts as a relevant dependency for that module in that particular year. Similarly, we generate the relevant dependency count for each module from 1993 to 2019.

The total dependency stream is the addition of all arrivals or removals for a module in the entire period from 1993 to 2019. The sum of the arrival/removal activity in a year across all the modules is the total activity in that year. This relevant event stream and the total stream is given as input to step 3, where we use Kleinberg's algorithm [20] combined with Viterbi's algorithm [35] to identify the bursts.

Step 3: Identification of the bursts of significant activity:

The relevant dependencies generated for each module and both the arrival and removal streams are called relevant event streams. The total stream is the total number of arrival/removal dependencies for a particular year. For every year, we know the number of arrival/removal dependencies and the total arrival/removal dependencies which are given as input to Kleinberg's algorithm. We have written a java program to use the arrival/removal stream, total stream to calculate the probabilities p_0 and p_1 given by $p_0 = \sum_{t=1}^{n} A_t/T_t$ where A_t is the set of arrival dependencies of the module at time t and T_t is the total number of arrival dependencies at time t. Similarly for the removal stream $p_0 = \sum_{t=1}^{n} R_t/T_t$ where R_t is the set of removal dependencies of the module at time t. The probability for the generating relevant dependencies in-state q_1 is p_1 given by $p_1 = s * p_0$ where s is a scaling parameter.

A cost is assigned for the model to fit the states, and a cost for the state transitions is also assigned. Once the cost is assigned then using Viterbi's algorithm the minimum cost state sequence is found by comparing the cost of all the paths from the previous state to the current state and chooses the path with minimum cost. For a particular year, if a path is in state 1, then there is a burst of significant activity that year. Once the bursts are formed then the weight of the burst in the interval $[a,b]$ is defined as

$$\sum_{t=a}^{b} \sigma(0, A_t, T_t) - \sigma(1, A_t, T_t)$$

$$\sum_{t=a}^{b} \sigma(0, R_t, T_t) - \sigma(1, R_t, T_t)$$

for arrival and removal streams respectively. This program will generate a CSV containing the weight of the bursts for each module in a particular year, where we have identified the significant activity. The CSV is then parsed by another java program which compares the weight of the bursts for every year across all modules. The module which has the highest weight is recognized as a burst of significant activity for that year. A similar process is carried out for all the years, and the bursts of significant activity are determined. In the next chapter, we will discuss the implementation of each phase and discuss the impact of these significant bursts of changes on the overall design of the software (modularity).

Chapter 5

Results

In this chapter, we analyze the results obtained in all three phases and compare it to the co-change modularity is studied for the same in [4]. In this thesis, we analyzed the functional dependencies of GNU Octave. Understand C++ tool provides the dependency information, and by using Gephi, we produced a dynamic network of the GNU Octave over its entire life cycle. Our goal is to identify the points in time called bursts of significant activity and to analyze their impact on the modularity.

5.1 Phase 1: Dynamic network using Gephi

For the software to have good design structure modularity is an important factor [3]. Given the software code with any number of files, the dependency relation is said to exist if a file uses another file syntactically. These dependencies affect the modular structure of the software. To get the dependency data, we need to analyze the code of the software.

We have written a Unix script that has cloned the repository based on the latest revision made every month from August 1993 to July 2019 a total of 312 months. The running time for the script is approximately 18 hours when executed on an Intel(R) Core i7-8700 CPU with 16 GB RAM. In total more than 25,000 revisions made in Octave were processed by our script.

We know that Octave has three stages in its evolution. The initial phase, the maintenance phase, and the open-source development phase. We plotted a graph to depict the revision history over its entire life cycle. The number of revisions made in each year from 1993

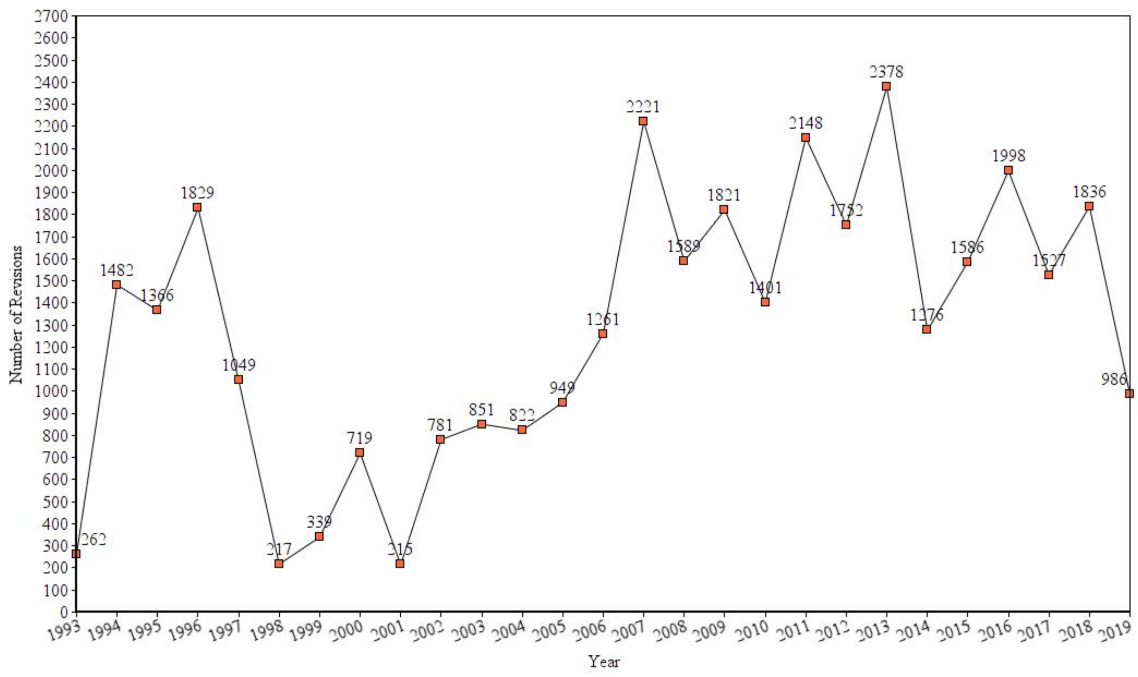


Figure 5.1: Revisions made per each year

to 2019 is shown in figure 5.1. This information will give us an idea about the years with significant activity and years with less activity.

The figure 5.1 shows the three stages in the evolution of Octave. The first initial stage is from 1993 to 1997. During the initial stage, there are several new files added to the software system leading to several revisions as we can see in the figure. During the years 1994, 1995, 1996, several revisions were made by Eaton himself. The second stage is the maintenance stage from 1998 to 2001. During this phase Eaton [8], mainly concentrated on project cleanups and structure maintenance. During this period no new functionality is added. So, the number of revisions is also less.

In the third stage from 2002 to 2019, is the open development stage. Many developers contribute to development. There is a rapid increase in the number of revisions made during this period. There were major releases in the years 2007, 2012 and 2013. In the year 2013, there is a major release for Octave and this year has recorded the most number of revisions for Octave when compared to all other years.

Table 5.1: Dependency information using Understand C++

Year	Number of Files	Number of dependencies	Cost
1993	248	1276	1184330000
1994	384	2875	76015600000
1995	558	4269	260310000000
1996	767	7230	13879600000000
1997	1001	9194	39227200000000
1998	795	6769	28734000000000
1999	823	7054	4188030000000
2000	887	7364	8833260000000
2001	922	7533	11820500000000
2002	896	7674	19402600000000
2003	1047	8527	181437000000000
2004	975	8642	313889000000000
2005	2615	34801	4662877063373252
2006	1330	11532	1.7585E+015
2007	3534	29245	5135229347974849
2008	2949	21137	1349852685704362
2009	3517	24679	1922930000000000
2010	3729	24498	2139240000000000
2011	5185	37581	1.5491541662E+016
2012	6571	35001	2.4907671911E+016
2013	7163	39657	2.9707353741E+017
2014	4223	25913	2984150000000000
2015	4224	25740	3986520000000000
2016	4308	26736	1.0383898996E+016
2017	4298	27577	9.82903183E+015
2018	4426	28693	2.1067391697E+016
2019	4416	26340	1.8041365978E+016

Using Understand C++, we produced the 312 dependency matrices for 312 months from August 1993 to July 2019. The java program for extracting the dependency information and creating an edge (dependency) table from all the 312 dependency matrices took about 7 minutes and 15 seconds. Table 5.1 has the information on the number of files and the number of dependencies recorded for each year. We also extracted the start date and end date information for each edge (dependency) using this program. As shown in Table 5.1 there is a drop in the total number of dependencies from 34801 to 29245, but there is an overall increase in the number of modules. In the years 2012, 2013 there were major releases 3.6.1 and 3.8.0 which corresponds to the highest number of modules. In the year 2014, there is an increase in modularity as the number of dependencies, modules drop for the release 3.8.1. From the year 2017 to 2019 the system seems to be more stable as the number of modules remains constant. The cost shown in Table 5.1 is obtained in Phase 2.

By using the edge information, a dynamic graph is created in Gephi. We have imported the edge table into the data laboratory. A static network was produced initially. To convert it into a dynamic network, we create a time interval based on the start and end dates. From the dynamic network, we can obtain the information of the edges present during a particular period several network statistics such as Modularity class, degree, strongly connected components can now be computed. The modularity class identifies the communities in the network using strongly connected components. Nodes in a community are assigned one color. Similarly, we size the nodes depending on their degree. To rearrange the graph by grouping the nodes of the same community together and also to prevent node and edge overlapping there are several layout algorithms.

ForceAtlas Layout:

ForceAtlas Layout can be used for networks with nodes ranging from 1 to 10,000. It took around 12 hours to run this layout algorithm in Gephi. It uses edge weights and it is slower compared to other layout algorithms. The layout still has a few node overlaps as

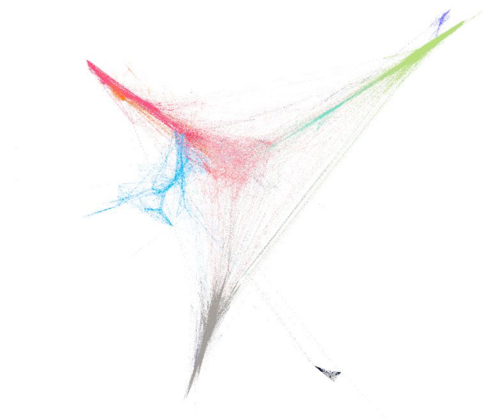


Figure 5.2: ForceAtlas Layout view of the network

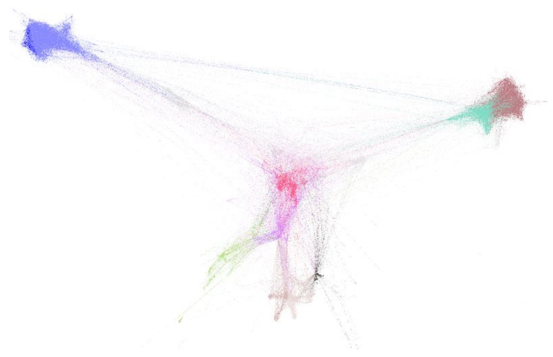


Figure 5.3: ForceAtlas2 Layout view of the network

shown in figure 5.2.

ForceAtlas2 Layout:

ForceAtlas2 Layout is a faster method and can be used for networks with nodes ranging from 1 to 1000000. It took approximately 2 minutes to run this layout algorithm in Gephi. The layout still has few edge overlaps, as shown in figure 5.3.

Yifan Hu Layout:

Yifan Hu Layout can be used for complex networks with nodes ranging from 100 to 100000. It took 1 minute and 30 seconds to run this layout algorithm in Gephi. It uses edge weights, and it is fast when compared to force atlas2 layout algorithm. The network ishown in figure 5.4.

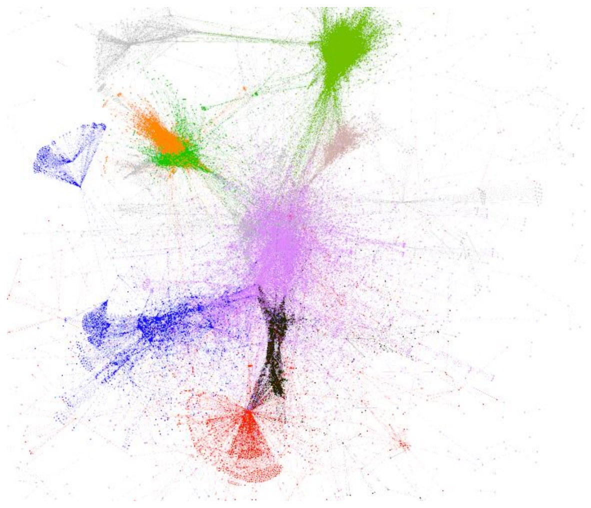


Figure 5.4: Yifan Hu Layout view of the network

OpenOrd Layout:

OpenOrd Layout can be used for networks with nodes ranging from 100 to 1000000. It took approximately 1 minute to run this layout algorithm in Gephi. It uses edge weights and it is fast when compared to the Yifan Hu layout algorithm. This algorithm has a fixed number of iterations. The resulting layout is as shown in figure 5.5.

5.2 Phase 2: Identification of Modular Structure

From the dynamic graph in Phase 1, we can extract the edge (dependency) table for each year. The java program which converts the edge table into a sparse matrix takes approximately 2 minutes for all years in the combine. The sparse matrices are then used to detect the modules by assigning a cost for each edge. The figure 5.6 is built from Table 5.1.

There are 497667 dependencies in total for all the years. There is an increase in the number of modules. There were only 12 modules in the year 1993, but in 2019 there were 66 modules. We compared the Number of Modules as shown in figure 5.7 to the Number of dependencies as shown in figure 5.6. During the initial stage of the GNU Octave, the number of dependencies increased and the number of modules increased at a slower rate till the year 2004. In 2005 there is a sudden increase in the number of dependencies from

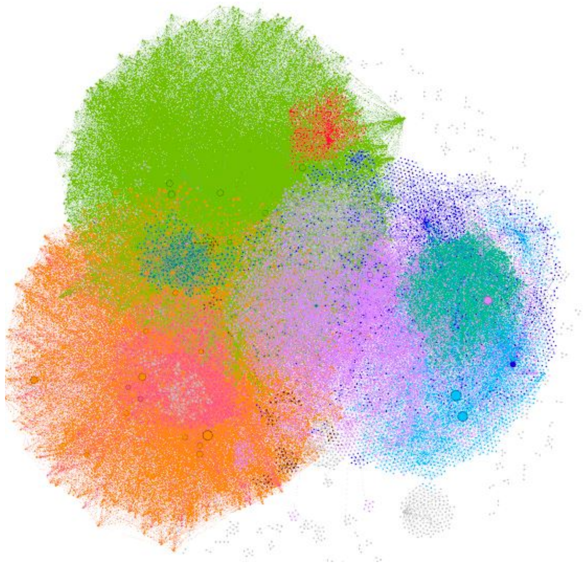


Figure 5.5: OpenOrd Layout view of the network

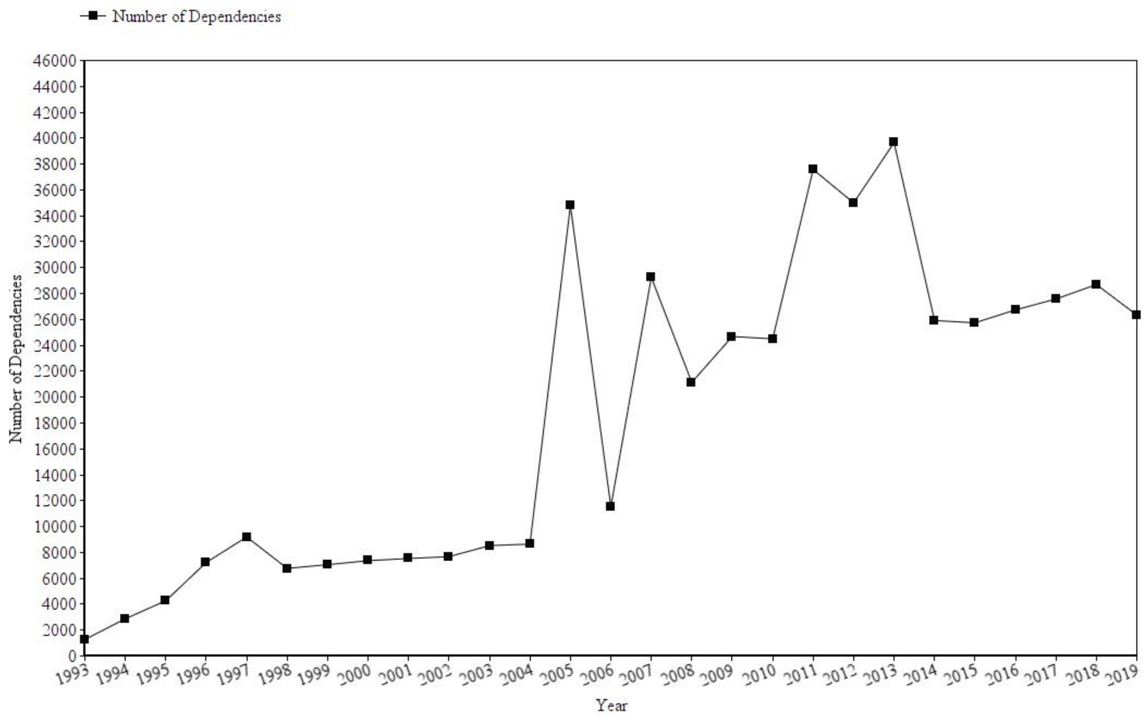


Figure 5.6: Number of Dependencies for each year

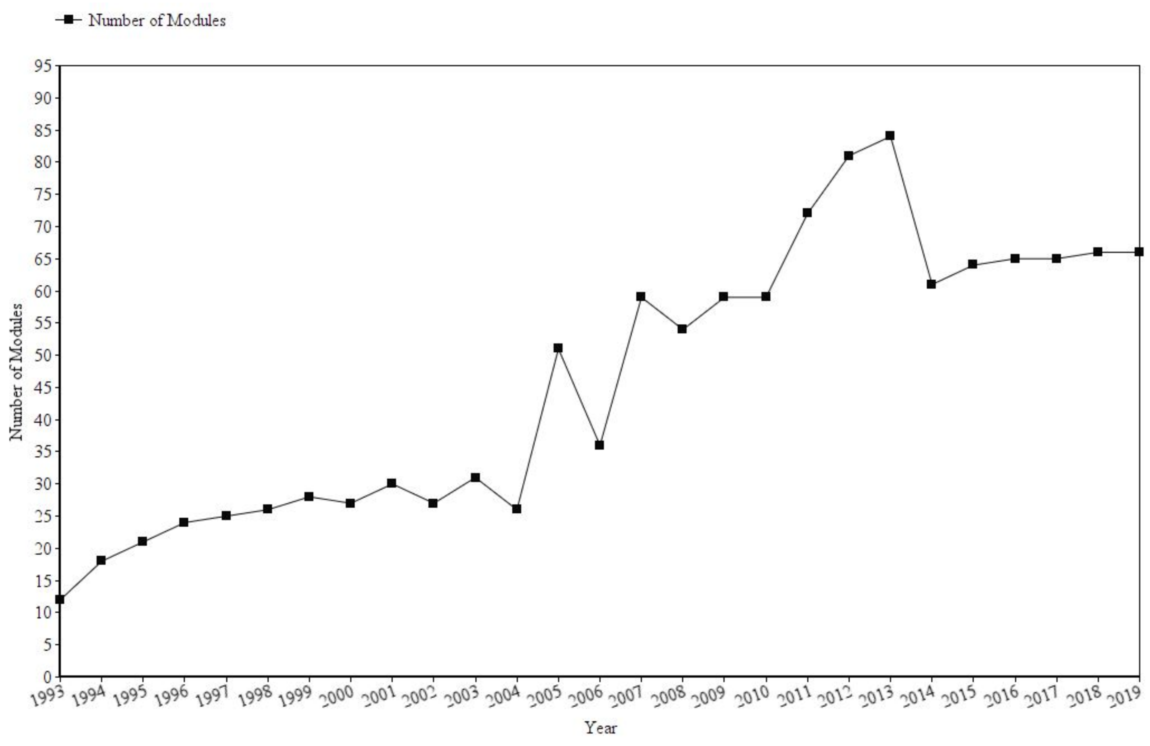


Figure 5.7: Number of Modules for each year

8642 to 34801, and also in the modularity. This sudden increase in the dependencies is because in 2005 Eaton moved out from Octave. Many contributors were involved in the development of new functionality. Many dependency relations are created. There is again a substantial decrease in the number of dependencies during the year 2006. There is also a decrease in the modularity of the software. In the year 2013, there is an increase in the number of dependencies (nearly 40000) and the number of modules in the system is also high when compared with the rest of the years. There was a major release in the year 2013. There were 84 modules in the year 2013. There is a drop in several dependencies, and also in the modularity in the year 2014. From 2015 onwards the system is more stable because there are no more fluctuations in the number of modules. In [18], it is observed that modularity can be increased by increasing the connectivity among the files. We also observed that when there is a major release then there are many dependencies created or removed. This improved the modularity of the system. The next goal is to find the bursts of significant activity which we will describe in phase 3.

5.3 Phase 3: Identifying the bursts of significant activity

The DSMs are extracted from the dynamic network produced in Phase 1 to which we have applied the clustering algorithm in Phase 2 and identified the modular structure of the network. In Phase 3, our goal is to find the bursts in time where significant activity occurs. For identifying the bursts of significant activity, we need the arrival stream and removal stream events.

The java program which extracts the arrival and removal stream from the edge table for each year runs approximately 2 minutes on Intel(R) Core i7-8700 CPU with 16 GB RAM. The arrival and removal streams are as shown in Table 5.2. The program to generate the relevant and total event streams runs for approximately 1 minute on the same configuration (Intel(R) Core i7-8700 CPU with 16 GB RAM). The table 5.2 is in sync with the releases made. From 1993 to 1997, there is an increase in the number of dependencies but for the removal stream, there are not many removals during this period. This clearly explains that Eaton concentrated on adding new features in the initial period. There is a sudden increase in the number of removals from 414 in 1996 to 2960 in 1997. This is because from 1997 to 2001 Eaton concentrated on clearing the project structure. No new functionality was added in this period. From the year 2005, there is a significant change in both arrival and removal streams because of the open-source development with many developers contributing to the new features in the software.

The relevant event and total event streams are given as input to the algorithm which is coded in java. It takes approximately 6 minutes to process all the modules and to find out the bursts. An edge (dependency) is counted as a relevant dependency for a module only if the two files belong to the same module. The relevant dependencies for each module and the total dependency information is input for phase 3. We got the bursts of activity as shown in figure 5.8 for arrival stream and figure 5.9 for removal stream.

In Phase 3, we worked with the modular structure of the dependencies in the year 2019. The table 5.3 shows the module number and its size. It also shows the year of activity.

Table 5.2: Arrival and removal streams

Year	Number of arrivals	Number of removals
1993	1276	0
1994	1546	31
1995	1454	239
1996	3079	414
1997	3508	2960
1998	418	1260
1999	382	161
2000	439	98
2001	495	88
2002	739	632
2003	1601	601
2004	1835	1737
2005	17855	16007
2006	1408	440
2007	18158	10401
2008	1180	192
2009	3111	364
2010	1011	420
2011	16279	22323
2012	14333	16854
2013	9728	2190
2014	1201	740
2015	1089	966
2016	2011	1165
2017	2791	2196
2018	4056	2668
2019	1942	3550

Table 5.3: Bursts of Significant activity based on Modular Structure of the year 2019

Module	Size	Arrival Year	Removal Year
49	16	1994	–
30	12	1996	–
20	39	1997	–
8	35	2005	2005
6	54	2007	2007
27	25	2009	2009
52	8	2011	2011
40	26	2013	2013
9	21	2016	2017
16	8	2017	–
19	18	2018	2018

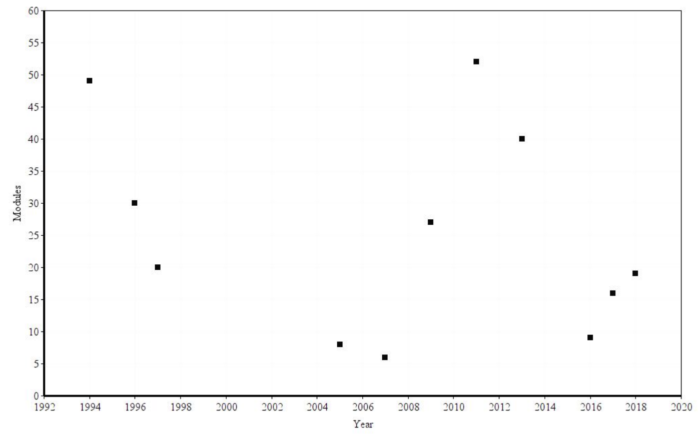


Figure 5.8: Arrival Bursts of significant activity

During the initial period of the Octave that is when Eaton was the chief developer the table 5.3 shows that the significant events are only arrival events during the period 1994, 1996 and 1997. We can also see that the number of modules has improved from 18 in 1994 to 25 in 1997. From this observation, we can infer that these significant bursts in the years 1994,1996 and 1997 have improved the modularity.

During the maintenance phase of the GNU Octave, which is from 1997 to 2001, Eaton mainly concentrated on maintaining the software rather than adding new features. There are no significant bursts for arrivals or removals. In the open development phase, when many developers were contributing, there were many releases, so there are significant arrival and

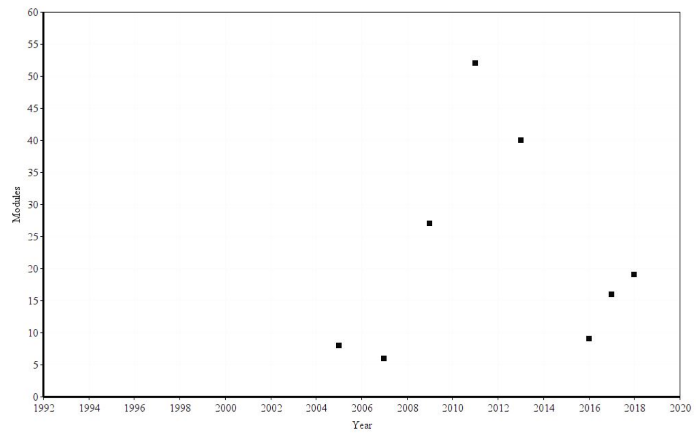


Figure 5.9: Removal Bursts of significant activity

removal bursts in 2005, 2007, 2009, 2011 and 2013. The significant burst of arrivals and removals in the year 2007 improved the modularity as it improved the cost as shown in Table 5.1. From the burst of the year 2007 we can observe that the local changes made in the year 2007 has improved the modularity. There are significant arrival events in the year 2016 which decreased the modularity as the cost is decreased. So, with the arrival burst of the year 2016 we can observe that the local changes made in the year 2016 has decreased the modularity. So, we observe that not all local changes will increase modularity. But after the year 2016, the significant bursts in 2017 and 2018 have made the system stable as the number of modules remains at a constant number.

This thesis confirms the approach in [4] on functional dependencies [4] and provides an implementation to identify the significant activity which is effecting the system modularity. The results of this study are consistent with the observation regarding the evolution of modularity in [18]. Our study can also be used for predicting software defects.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we identified the bursts of significant activity and analyzed the impact of these changes on the modularity of the software over its entire life cycle. We extracted the functional dependency information of the code based by cloning the repository every month and analyzed the dependency using Understand C++. We used the Gephi to produce a dynamic network based on the dependency table from Understand C++. We used the clustering algorithm in [4] to identify the modules. This modular strcuture, along with the arrival and removal stream extracted from the dynamic network helped us to run the algorithm based on [20] and [35] to identify the bursts of significant activity.

In this thesis, we observed that these bursts have made the system modular and more stable, and we also identified the points in time over the entire life cycle of GNU Octave. Similar to [11], it is observed that local changes made in the system improved the modularity of the software. In [4] they have analyzed the GNU Octave based on the co-change dependencies where we have analyzed based on functional dependencies, and we have also identified the bursts of significant activity which made the system modular.

6.2 Future Work

1. We have analyzed only for the .c, .cc, .cpp, .h files of the octave software. There are also .m files that can be analyzed.

2. We can analyze the error logs and also identify the impact of the changes made to fix

those errors based on the commit logs on the overall modularity of the system as in [33].

3. The same methodology can be applied on another software to identify the hotspots which change the modular structure of the software system.

References

[1] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134:120–137, 2017.

[2] Krause Andreas, Leskovec Jure, and Guestrin Carlos. Data association for topic intensity tracking. *Proc. of 23rd international conference on Machine Learning, ACM*, 2006.

[3] C.Y. Baldwin and K.B. Clark. 1999. Design rules:the power of modularity vol. 1. 1999.

[4] Robert Benkoczi, Daya Gaur, Shahadat Hossain, and Muhammad Ali Khan. A design structure matrix approach for measuring co-change-modularity of software products. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 331–335. IEEE, 2018.

[5] Cafeo B.P., Cirilo Elder, Garcia Alessandro, Dantas Francisco, and Lee Jaejoon. Feature dependencies as change propagators: An exploratory study on perfective maintenance of software product lines. *Information Software Technology*, 2016.

[6] Ken Cherven. *Mastering Gephi Network Visualization*. Packt Publishing Ltd, 2015.

[7] Clarkson Dr.P. John, Simons Caroline, and Eckert Dr. Claudia. Predicting change propagation in complex design. *J. Mechanical Design*, 2004.

[8] John W. Eaton. Octave: Past, present and future. *In proc. of the 2nd International Workshop on Distributed Statistical Computing*, 2001.

[9] Steven D Eppinger and Tyson R Browning. *Design structure matrix methods and applications*. MIT press, 2012.

[10] Beck Fabian and Diehl Stephan. On the impact of software evolution on software clustering. *Empirical Software Engineering*, 2013.

[11] Miguel A Fortuna, Juan A Bonachela, and Simon A Levin. Evolution of a modular software network. *Proceedings of the National Academy of Sciences*, 108(50):19985–19989, 2011.

[12] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 1991.

[13] Markus Michael Geipel and Frank Schweitzer. The link between dependency and cochange: Empirical evidence. *IEEE Transactions on Software Engineering*, 38(6):1432–1444, 2012.

[14] Oliva Gustavo, Ansaldi and Gerosa Marco, Aurelio. Experience report: How do structural dependencies influence change propagation? an empirical study. *IEEE*, 2015.

[15] Gall Harald, Hajek Karin, and Jazayeri Mehdi. Detection of logical coupling based on product release history. *International Conference on Software Maintenance*, 1998.

[16] Gall Harald, Jazayeri Mehdi, and Krajewski Jacek. Cvs release history data for detecting logical couplings. *Proc. 6th International workshop on Principles of Software Evolution IWPSE*, 2003.

[17] Y. F. Hu. Efficient and high quality force-directed graph drawing. *The Mathematica Journal*, 2005.

[18] Clune Jeff, Mouret Jean-Baptiste, and Hod Lipson. The evolutionary origins of modularity. 2013.

[19] Devangana Khokar. *Gephi Cookbook*. Packt Publishing Ltd, 2015.

[20] Jon Kleinberg. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery*, 7(4):373–397, 2003.

[21] Ravi Kumar, Jasmine Novak, Prabhakar Raghavan, and Andrew Tomkins. On the bursty evolution of blogspace. *World Wide Web*, 8(2):159–178, 2005.

[22] Alan MacCormack, John Rusnak, and Carliss Y Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7), 2006.

[23] S. Martin, W. M. Brown, R. Klavans, and K. Boyack. Openord: An open-source toolbox for large graph layout. *SPIE Conference on Visualization and Data Analysis (VDA).*, 2011.

[24] Bastian Mathieu, Heymann Sebastien, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. *Third International AAAI Conference on Weblogs and Social Media*, 2009.

[25] Michael Mathioudakis, Nilesh Bansal, and Nick Koudas. Identifying, attributing and describing spatial bursts. *Proc. of the VLDB Endowment*, 2010.

[26] M.LaMantia, Y.Cai, A.MacCormack, and J.Rusnak. Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases. *IEEE/IFIP Conference Software Architecture*, 2008.

[27] Sangal Neeraj, Jordan EV, Sinha Vineet, and Jackson Daniel. Using dependency models to manage complex software architecture. *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications*, 2005.

[28] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media Inc, 2009.

[29] He Qi, Chang Kuiyu, Lim Ee-Peng, and Zhang Jun. Bursty feature representation for clustering text streams. *Proc. of 2007 SIAM Internationsl Conference on Data Mining*, 2007.

[30] S. Rajagopalan R. Kumar, P. Raghavan and A. Tomkins. Trawling the web for cyber communities. *WWW8/Computer Networks*, 31:1481–1493, 1999.

[31] Milev Roberto, Muegge Steven, and Michael Weiss. Design evolution of an open source project using an improved modularity metric. *Carleton University*, 2009.

[32] S.Bohner and R.Arnold. Software change impact analysis. *IEEE*, 1996.

[33] Richard W. Selby and Victor R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, 1991.

[34] Wong Sunny, Cai Yuanfang, Kim Miryung, and Dalton Micheal. Detecting software modularity violations. *ICSE*, 2011.

[35] Andrew J Viterbi. A personal history of the Viterbi algorithm. *IEEE Signal Processing Magazine*, 23(4):120–142, 2006.

[36] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *In the Proceedings of 26th International Conference on Software Engineering (ICSE '04)*, 2004.

Appendix A

Bursts based on different modular structure

This appendix included the tables which displays the bursts that are generated based on the modular structures of the years 2001, 2004 and 2005.

Table A.1: Bursts of Significant activity based on Modular Structure of the year 2001

Module	Size	Arrival Year	Removal Year
2	308	1994	–
1	398	2001	2001
24	20	2005	2005
4	27	2007	2007

Table A.2: Bursts of Significant activity based on Modular Structure of the year 2004

Module	Size	Arrival Year	Removal Year
2	328	1994	–
1	420	2005	2005
15	21	2007	2007
1	420	2011	2011
1	420	2012	2012

Table A.3: Bursts of Significant activity based on Modular Structure of the year 2005

Module	Size	Arrival Year	Removal Year
2	872	1997	–
1	1105	2005	2005
6	29	2007	2007
4	47	2009	2009
1	1105	2013	2013
30	48	2018	2018

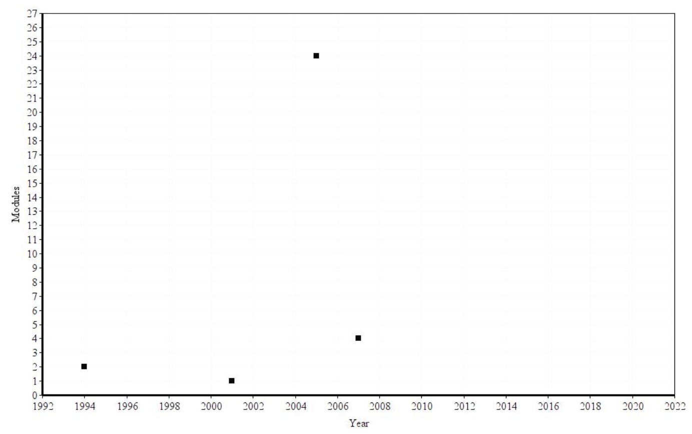


Figure A.1: Arrival Bursts of significant activity based on Modular Structure of 2001

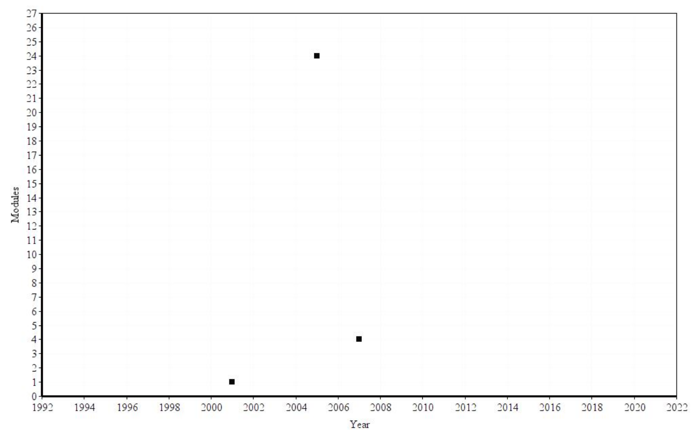


Figure A.2: Removal Bursts of significant activity based on Modular Structure of 2001

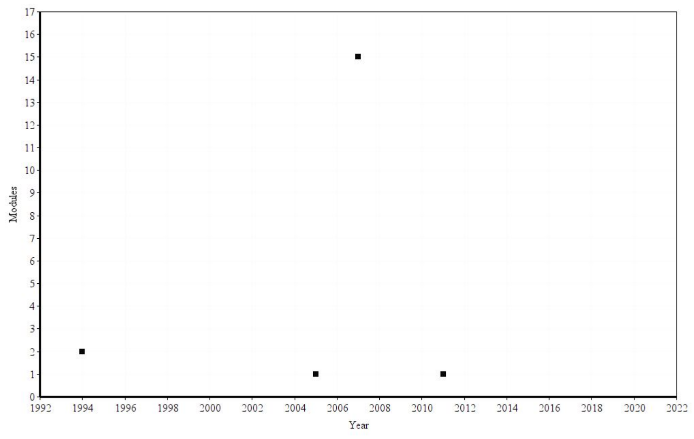


Figure A.3: Arrival Bursts of significant activity based on Modular Structure of 2004

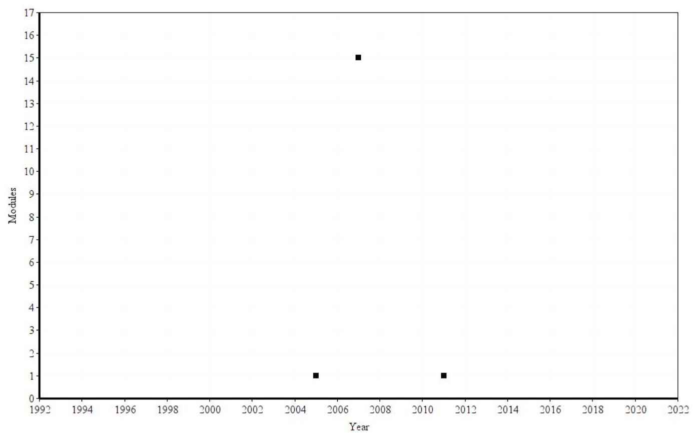


Figure A.4: Removal Bursts of significant activity based on Modular Structure of 2004

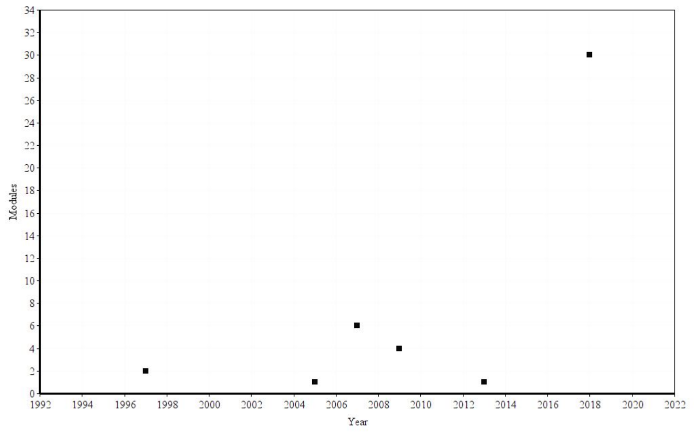


Figure A.5: Arrival Bursts of significant activity based on Modular Structure of 2005

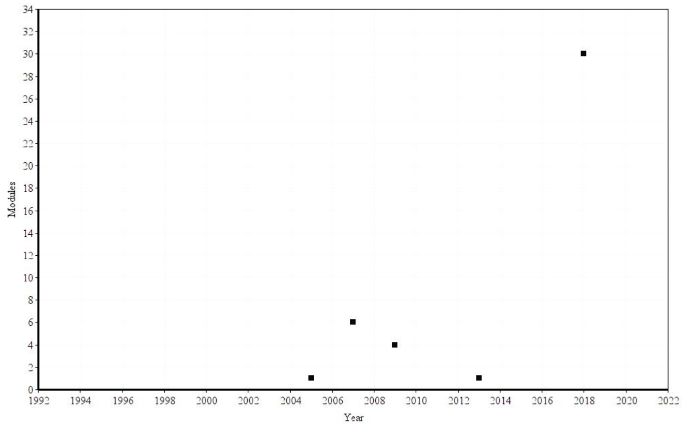


Figure A.6: Removal Bursts of significant activity based on Modular Structure of 2005

Appendix B

Preliminaries and Related Concepts

B.1 Introduction

In this chapter, we will discuss the tools and concepts used in this thesis. In the first section we describe the tool Understand C++ used for analyzing the code structure, in the next section we describe the tool Gephi used to produce graphs or networks and in the last section we will discuss the concepts of mercurial repository on how to clone the repository and how to check the revisions according to date.

B.2 Understand C++

Understand C++ is an integrated development environment that was built to help users or developers edit, view, analyze their code. It helps in visualizing the program flow and also makes debugging a lot easier. It has a fully loaded C++ editor. Understand has the following features:

1. Static Analysis:
    - (a) Code Analysis: It helps to see the interactions between functions, classes and also helps to see the variables declared in the entire project.
    - (b) Graph Visualization: The interactions between various functions or classes in the whole project can be visualized in graphs.
    - (c) Dependency: The dependencies can be viewed based on functions in the project or class dependencies or architecture dependencies.
2. Reports:
    - (a) The dependency reports can be exported either as a CSV, HTML file or into a database file.
    - (b) We can get the error report which includes the details of the error such as the exact line number in the code.
    - (c) We can even compare files and folders in the project.

Understand C++ tool has a project configuration menu where we have to set up our project details like the languages used in our project, project location, etc. Understand C++ supports with two analyzers fuzzy and strict. In the Fuzzy analyzer, we don't have to

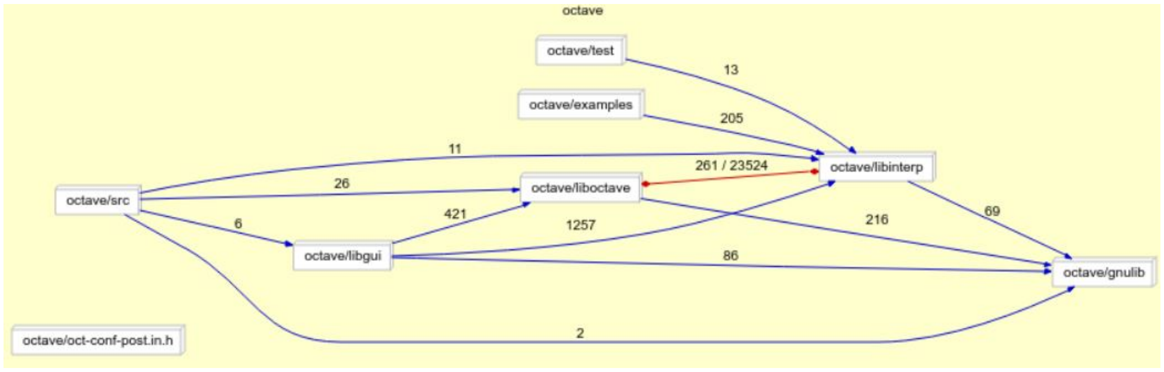


Figure B.1: Dependency Graph based on Project Architecture

compile the code to understand to analyze the project. Even if the project has errors fuzzy analyzer still analyzes the entire project and produces the dependency report. But in the case of a strict analyzer we have to compile the code to analyze the project, if the project has errors then understand cannot analyze the project. For C/C++ projects we can use the fuzzy analyzer, but for other complex projects, we have to use the strict analyzer. Understand C++ supports multiple languages like C, C++, Java, Objective-C, Objective-C++ and so on.

The following are the steps involved to analyze the project in Understand C++.

Step 1 : Run Understand C++ and click on ”New Project”.

Step 2 : Enter the Project Name and the project directory.

Step 3 : Then select the language that is used in the project and select fuzzy or strict based on the requirement.

Step 4 : Then we have to specify the project root directory location. Note that this directory will be watched by understand so that any changes made in this directory will automatically reflect in Understand C++.

Step 5 : Once the project is added then the project will be analyzed automatically.

Step 6 : In Step 3, if we have selected fuzzy then go to Step 7, if we have selected strict then go to Step 8.

Step 7 : We do not have to compile the project. Once the project is analyzed we can start generating the reports.

Step 8 : For Strict analyzer, we have to compile the project, so we have to add includes, macros of the project under the project configuration menu. Unless the project is compiled we cannot generate the reports.

The dependency graph for an example project looks like as shown in Figure B.1. This figure shows the dependencies based on the architecture of the project. For example, the octave/test has 13 dependencies from octave/libinterp. Similarly, we can produce various graphs or reports based on class dependencies or functional dependencies.

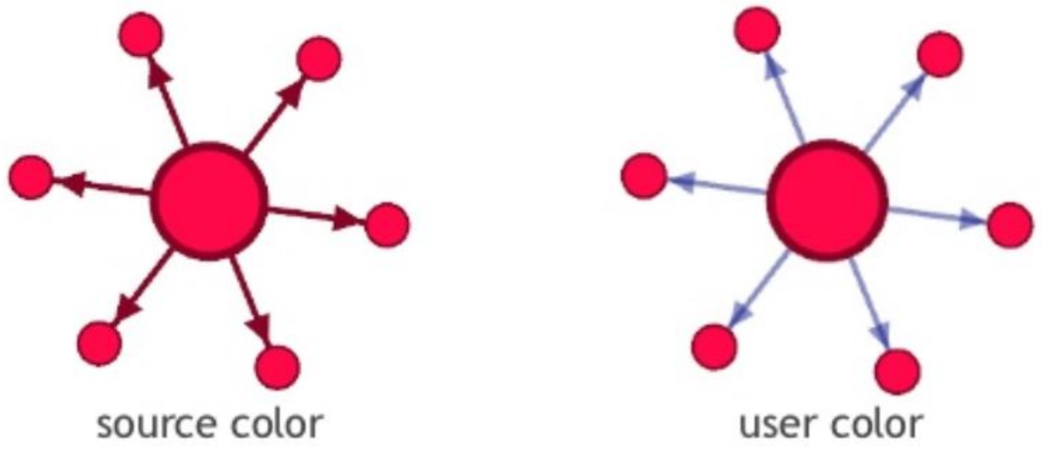


Figure B.2: Difference between edges colors

To generate a report of the dependency structure in the menu have to select reports then there will be three types of reports available to generate. One is a dependency report based on file dependencies where the report generates the dependency matrix csv only based on dependencies between files in the project. We can also generate a report based on architectural dependencies where the report consists of dependency relations based on the architecture of the project. We can also generate a report based on class dependencies, wherein the report we can see the details of a class depending on another class. It gives the information of all class dependencies recorded in the project.

B.3 Gephi

Gephi is a tool used for network visualization and analysis. It is an open-source software where it is maintained in GitHub. Gephi helps us to analyze large networks by clustering or filtering techniques[19]. A network is comprised of nodes and edges. There are various properties associated with a network such as In Degree, Out degree, Modularity and so on. These properties will be discussed later. Gephi can run many statistics on the network and helps to improve the visualization making the network more clear. We can also filter the network based on the requirements, in the filter options we can add queries that we need[24].

The following are some important functionality used in gephi.

Edges : The Color of the edges can be configured. By default, the color of the edges will be the same as the color of the source node as shown in B.2.

Labels : The size, color of the node labels and edges labels can be configured. The size of the labels can be either fixed for all the nodes or can also vary according to the size of the node.

Overlap : Gephi also provides a function to eliminate labels overlapping. To eliminate label overlapping, we have to go to the Layout panel and select ”label adjust” and then click on Run.

These are the following steps involved to produce a network.

Step 1 : Run Gephi and click on ”New Project”.

Step 2 : Go to the Data Laboratory Tab and import the data either as an edges table or as a nodes table or an adjacency list.

Step 3 : While importing if there are any parallel edges do not forget to choose merge strategy.

Step 4 : Now we should be able to view a graph. We can adjust the edge thickness.

Step 5 : Now we can go to the Statistics tab and run the necessary statistics like Modularity, Degree, Connected Components.

Step 6 : Once the statistics are completed then we can rank the nodes according to their degree. For doing this, go to the Ranking tab and choose a Degree from the list. Select the color and apply it. Now the nodes will be colored based on the degree rankings.

Step 7 : We can also change the size of the nodes similarly based on the degree in the ranking tab.

Step 8 : Gephi can be able to detect communities by the metric Modularity.

Step 9 : Go to partition and select Modularity Class, then we can see all the communities in the network. We can assign a different color to each community and then click on apply.

Step 10 : We can hide nodes or edges from visibility in the network by filtering the graph. For example, if you would like to see only the nodes with degree greater than 10. Then we can go to the Filters tab, Topology and drag Degree Range to the Queries Tab and select the range then click on filter.

Step 11 : The last step is to export the graph as a png or pdf. We can also take a screenshot by clicking on the camera symbol at the bottom of the visualization tab.

If the graph is small with less number of nodes the graph there might be few overlapping of nodes and overlapping of node labels. So to remove overlapping we need to select Label Overlap function under the layout tab. In case the graph is too large with more nodes then the graph looks very unpleasant where even nodes might overlap and the edges might be too large. To solve these problems gephi provides different layout algorithms where we can expand the graphs, prevent node and label overlapping, and also groups the nodes of the same community together such that the edges will not be large[19]. There are many layout algorithms available both in the 2D model and also in the 3D model.

- ForceAtlas Layout : ForceAtlas Layout can be used for networks having nodes ranging from 1 to 10,000. Its complexity is O(N^2). It is slow when compared to other layout algorithms. Before running the algorithm we can change a few settings depending on the requirement. By selecting Adjust by size option we can eliminate node overlapping. It uses edge weights.

- Fruchterman-Reingold Layout: This layout can be applied to networks with nodes size ranging from 1 to 1000. This layout algorithm has also had a complexity of O(N^2). It does not use edge weights[12].

- Yifan Hu Layout: This layout is suitable for complex networks having nodes ranging from 100 to 100000. It is very fast when compared to other layout algorithms. Its complexity is O(N*(log(N)))[17].

- ForceAtlas 2 Layout: This is designed for large networks and with reducing complexity when compared to ForceAtlas. This algorithm can be applied to a network with nodes ranging from 1 to 1000000 nodes. Its complexity is O(N*(log(N))) and uses edge weights.

- OpenOrd Layout: This algorithm works for networks with nodes ranging from 100 to 1000000. This algorithm has a fixed number of iterations. Its complexity is O(N*(log(N))) and uses edge weights[23].

Gephi also provides a method to produce dynamic network[6]. For a network to be dynamic the nodes or edges should be having a start date and end date. The advantages of dynamic networks are we can make the nodes in the network visible only if the current date is in between the start date and end date of the node.

These are the following steps for producing a dynamic network using gephi. Dynamic networks help us in improving network visualization.

Step 1 : Run Gephi to load the static network by opening the graph file. When we open we can see the number of nodes, edges in the graph.

Step 2 : In Data Laboratory tab and for the start date and end date columns are mandatory. For a network to be dynamic with nodes or edges should have a start date and end date.

Step 3 : If we want to display the nodes only based on start dates, i.e the nodes will be visible from the start date till the end then go to step 4, otherwise step 5.

Step 4 : By using Merge columns, select start date in the left tab and move to the right tab. Then go to step 7.

Step 5 : If we want to display the nodes based on their start date and end date then go to step 6.

Step 6 : Merge columns will merge the selected start date and end date.

Step 7 : In the merge strategy there will be an option called ”Create time interval”. which can change the configuration of the time interval in configuration settings.

Step 8 : Interval column will be created in the data laboratory. Once we have merged the columns of start date and end date we will be able to enable timeline.

Step 9 : Now the timeline will be enabled. We can go to Overview and create an animation for the visualization of the graph.

B.4 Mercurial Repository

Requirements evolve in a continuous manner which rises to modifications in the current version of the software system. In case if the requirement is changed and we want to revert to the previous version, it would not be a good practice to revert all those modifications manually to get the previous version. It involves a lot of time, effort because if we revert those modifications manually then we have to test the system to make sure that all the modifications are reverted. So, revision control solves this problem by maintaining the older version of the code so that we can revert to older versions or newer versions whenever required [28].

There are different kinds of repositories such as Git, Mercurial, CVS. We are discussing mercurial because we took Octave as our case study to determine the periods in time where significant events occurred. Octave is maintained using a mercurial repository. Mercurial repository is easy to customize, easy to learn and use [28]. To install Mercurial for windows we can use TortoiseHg. TortoiseHg provides with graphical interface and command-line interface [28].

In this thesis, we have used the mercurial repository to clone the files according to the revisions and download the various versions of the Octave source code to analyze the dependency network which will be discussed more in Chapter 3.

B.5 Java Integration

In this thesis to achieve some tasks we have implemented our methodology using java as source code. We have used Java because it is the most popularly used programming language that follows the object-oriented design principle. The main advantage of Java over C++ is the runtime error detection mechanism is handled by the system whereas in C++ the runtime error detection is handled by the user.

Java is free software by Oracle where we can download the JDK (Java Development Kit) from the Oracle website. The Java project was started in the year 1991. Java 13 is the latest version of Java which is released in September 2019. We can use threads in Java unlike in C++. Java provides guarantees for security such as Security-related APIs and Byte-Code verification. Java supports multithreading functionality where it can perform multiple tasks at the same time.

B.5.1 Compiling the Java source code

The Java source code is written in an IDE, notepad or using a text editor. The file containing the source code should be saved as .java file. We can use the javac command to compile the java code. When we use the javac command then the source code is compiled and will be translated into a byte code if there are no errors and a file with .class is created. This .class file is then compiled or interpreted to execute the program. The byte code which is in .class file is then converted into the machine-understandable code with the help of JVM(Java Virtual Machine). The platform independence of Java is achieved through JVM.

B.5.2 Eclipse IDE

Eclipse is an integrated development environment(IDE) software that provides an ability to write and compile the Java source code. Eclipse helps us to write java code faster and also tells the error associated with the code. Eclipse provides with many built-in plugins, we can also clone the GIT repository to eclipse. The other IDE's are Codenvy, IntelliJ, NetBeans, Greenfoot and so on.

B.5.3 HashMap - Java collection framework

HashMap comes under the Java collection framework. Java HashMap class has two parameters Key and Value. It stores the value based on the unique key. To retrieve the value we need to pass the key as a parameter. For example, in HashMap if we store '1' as key and 'abc' as its value then to retrieve the value we need to use the get method of HashMap and use '1' as key to getting the value 'abc'. HashMap does not support duplicate keys.

B.5.4 CSV reader/writer

The CSV reader or CSV writer is a part of OpenCSV library of Java. Using CSV reader we can read the whole CSV file at once and store as ArrayList of a String array. With the help of CSV writer, the data present in ArrayList of String array can be written in the form of CSV. CSV stands for Comma Separated Values.

Appendix C

Dynamic Graph of GNU Octave

The dynamic graph of GNU Octave based on the functional dependencies can be viewed from the following the link *http://www.cs.uleth.ca/˜gaur/papers/dynamicGraph.mp4*.