2013

Automated Software Architecture Extraction Using Graph-based Clustering

John Thomas Chargo
Iowa State University

Automated software architecture extraction

using graph-based clustering

by

John Thomas Chargo

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Suraj Kothari, Major Professor

Tien Nguyen

Joseph Zambreno

Iowa State University

Ames, Iowa

2012

DEDICATION

I would like to dedicate this thesis to the friends, family, and teachers who have inspired, supported, and motivated me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my sincere thanks for those that helped me in conducting research and writing this thesis. First and foremost I'd like to thank my major professor, Dr. Suraj Kothari for his direction and guidance throughout the development of this thesis. His knowledge of software engineering and experience in guiding graduate research has been invaluable. I'd also like to thank my committee members for their contributions to this work: Dr. Tien Nguyen and Dr. Joseph Zambreno.

ABSTRACT

As the size and complexity of software grows developers have an ever-increasing need to understand software in a modular way. Most complex software systems can be divided into smaller modules if the developer has domain knowledge of the code or up-to-date documentation. If neither of these exist discovery of code modules can be a tedious, manual process.

This research hypothesizes that graph-based clustering can be used effectively for automated software architecture extraction. We propose methods of representing relationships between program artifacts as graphs and then propose new partitional algorithms to extract software modules from those graphs. To validate our hypothesis and the partitional algorithms a new set of tools, including a software data miner, cluster builder, graph viewer, and cluster score calculator, were created. This toolset was used to implement partitional algorithms and analyze their performance in extracting modules. The Xinu operating system was used as a case study because it has defined modules that can be compared to the results of the partitional algorithm.

CHAPTER 1. OVERVIEW

As the size and complexity of software grows developers have an ever-increasing need to understand software in a modular way. It is impractical for one person to understand all the code in a large system. However, most complex software systems can be divided into many smaller modules. For instance, an operating system might contain modules that handle tasks, file I/O, networking, or memory management. If a developer can identify these modules the task of understanding the code becomes easier. If the developer with knowledge of these modules sets out to modify the software, he or she won't need to understand everything about all modules, but rather the module he or she is modifying and how it interacts with the other modules.

Frequently the existence of code modules is known to the developers only after they gain significant domain knowledge of the software. For instance, a developer working on operating system software would probably know there are file, network, and task scheduling subsystems. Without significant domain knowledge discovering these modules can be a tedious process. Cross-cutting connections further complicate this process by obscuring module boundaries. This research focuses on methods to automate the discovery and identification of code modules.

Recently the use of clustering techniques borrowed from other fields have made their way into research in the software engineering field. This research summarizes these existing methods in an overview of how clustering and graphs have been applied to software. It then explores the use of graphs for software clustering and visualization. Included is a description of a test bed that was developed for experimentation and analysis of clustering applied to software. This test bed allowed us to develop our own graph-based partitional clustering algorithm and compare it to another popular clustering algorithm, the weighted combined algorithm.

CHAPTER 2. BASICS OF CLUSTERING AND GRAPHS

In general, clustering is a process of grouping together inter-related items based on common properties or features of the items. By grouping together similar items, complex data sets can be divided into multiple smaller sets that are more manageable. As a practical application, these smaller sets could be divided amongst members of a team if the complex set is more than a single developer could manage. Successful grouping of common items can decrease the scope of data that must be understood or processed at one time and can therefore simplify complex systems.

Clustering of data has been applied within many disciplines as a method of analyzing data. In biology taxonomy is used to cluster groups of biological organisms based on common properties. For instance, mammals are grouped together; reptiles are grouped together, etc. Through this clustering a biologist need not be an expert on all animal species. Instead he or she can understand properties of an animal based simply on the cluster in which it exists. Clustering is similarly used in astrophysics, health sciences, and chemistry.

Distance is a common notion in clustering used as a heuristic for measuring how similar items are to each other. Items sharing a small distance are very similar whereas those with a large distance are very different. Distance can be an abstract measure. For example, the distance between two strings could be how many characters they have in common, how similar the length of the strings are, or how close they are alphabetically. Regardless of the definition of distance, many clustering algorithms function by grouping items with the smallest distances together. There are exponentially many cluster combinations for any data set. Clustering algorithms work efficiently to create meaningful clusters, those with the smallest distance between items in the clusters and maximum spacing between clusters[9].

Graphs are mathematical structures used to model relationships between objects. A graph

contains vertices, or nodes which represent entities and edges that represent the relationship between those entities. Relationships represented as edges can be physical relationships; in the case of transportation network graphs nodes could be cities and edges can represent the roads that connect them. Edges can also be abstract or virtual, representing some other relationship nodes have in common. Graphs are commonly used in computer science because they allow for intuitive visualization of relationships by modeling vertices as dots or circles and edges as lines connecting those dots or circles.

Significant research has been done on the properties of. Problems solved by graphs include graph coloring, routing, network flows, and covering.

Of particular relevance to this research is the use of graphs to represent and analyze flow networks. Flow networks, frequently used for transportation network diagrams, have weighted edges that connect a source node to a sink node through intermediate nodes. Flow problems include those like the maximum flow problem which seeks to find the path with most capacity, based on edge weight, from a source to a sink node.

Algorithms like the Ford-Fulkerson Algorithm and Edmond-Karps Algorithm are available to compute the maximum flow of a flow network. These maximum-flow algorithms are augmented by the max-flow min-cut theorem which states that the maximum flow from a source to a sink in a flow network is equal to the minimum capacity, which if removed, would result in no flow between the source and the sink[9]. Finding the minimum-cut is a mathematically proven way of splitting a graph into two smaller graphs.

It is theorized that if one could adequately model software as graphs, that such algorithms from the graph theory field could be used to break complicated sets of software artifacts into smaller sets that represent the software's modules.

CHAPTER 3. SOFTWARE CLUSTERING AND GRAPHS

There are numerous methods of quantifying the quality of software. Many such methods analyze the amount of coupling and cohesion in a given codebase. Well-designed software will exhibit low coupling, the degree in which a module depends on other modules, and high cohesion, the degree in which elements of a module belong together[8]. Software elements in code with low coupling and high cohesion will naturally group together.

When these natural groupings, or clusters are known, software understanding and maintenance becomes easier because developers may be able to limit the scope of their work to the cluster under review. Research into the use of clustering on software aims to exploit the natural sets that exist within well-designed code to expose underlying secrets of the software.

Likewise, graphs can be used as a tool to pictorially visualize relationships implicit in the software. Software itself includes several sets of artifacts that are interrelated. At a macro level software can be viewed as a series of functions that are in the same classes or a series of header files that include each other. At a micro level software can be viewed as individual code statements that call each other or access the same variable. These relationships can all be modeled as graphs.

Software developers can use function call graphs to visualize a sometimes complicated web of function calls. Control flow branches can be visualized to simplify test coverage analysis. Beyond those visualization techniques, it has been shown that techniques from the graph theory field can be applied to software if the correct program artifact relationships are graphed[2].

Another such method of modeling program relationships as graphs is the source graph, described in [13] . In a source graph nodes represent code files, functions, data types, and variables. Edges represent the contain and use relationships between the nodes. Included in source graphs are sub-graphs like uses graphs, data dependency graphs, definition-use graphs,

and is-component-of graphs. These sub-graphs are used by [4] as a property-based measure of software. Through existing research it is apparent that modeling software as graphs can provide a valuable abstraction for software analysis.

CHAPTER 4. PROPOSED APPROACH

As mentioned previously and detailed in the related works section, research has been done on methods of visualizing software and methods of applying clustering to software. A goal of this research was to explore new methods of combining these two areas of study to further a developer's ability to understand and maintain software.

It was hypothesized that applying clustering to software graphs could be used not only to extract software modules but also to display it in a way that is easy for a developer to understand. To accomplish this we begin by deciding how to represent software as graphs and then how to apply clustering to those graphs.

There are numerous different relationships inherent in software and because graphs provide an abstraction to construct powerful algorithms to analyze those relationships there are numerous ways to model software as graphs. Each of these approaches has its advantages and disadvantages.

We began with a simple call graph, which is a classic way of modeling call relationships. In a call graph functions are represented as nodes. Two nodes share a directed edge if one of the functions calls the other function. This representation reveals functional flow but we found it didn't give a complete enough view of the software for module extraction. Call relationships don't reveal hidden control mechanisms like the use of mutexes and semaphores for synchronization of multi-threaded programming. Call relationships also don't reveal the flow of data through shared memory or mailboxes.

While call relationship graphs are valuable, other software relationships can provide a more relevant graph for object-oriented software. Commonly accepted object-oriented design principles dictate the grouping of related types and functions that operate on those types. This research proposes taking advantage of this principle by relating functions based on the data

types they access rather than just their call relationships. For example, two functions using the common wait and signal functions for thread synchronization wouldn't be related in a call graph; there is no direct function call from one to another. However, the semaphore type that these two functions use is common between the functions. Using the usage of types to relate functions gives a data set that includes hidden control.

We propose modeling this type usage relationship in a graph that uses functions as nodes that share an edge if the two functions access a common data type. The edges are therefore not directed. It can be noted that two functions could share multiple different data types and theorized that functions sharing many data types are more closely related than those sharing fewer data types. To improve the clustering process edges were weighted based on the number of data types the source and sink nodes share.

After deciding on the software relationships to model, analysis can be done on that graph to uncover information about the software. As previously mentioned, it was theorized that clustering based on the software graph could be used to extract software modules. Creating clusters from graphs has been widely researched. However, we believe that the generic clustering algorithms fail to take advantage of some truths hidden in the software. For instance, code with low coupling and high cohesion will form natural groupings. Because the graphs, and the relationships contained, are of a particular type we can make assumptions about how clusters in the graph should present themselves.

Methods of clustering graphs, like using the Edmonds-Karp maximum flow algorithm combined with the min-cut theorem to divide graphs into smaller graphs, have been proven to provide optimal results[6]. A limitation of this and similar algorithms is that the runtime complexity of each iteration is $O(NE^2)$, where N is the number of nodes and E is the number of edges. Software projects can easily involve hundreds if not thousands of functions which makes implementing these algorithms on software impractical.

By exploiting the assumption that well-designed object oriented code will naturally cluster, we theorize that computationally-complex optimal clustering isn't necessary. We propose that a much simpler algorithm that approximates clusters is more than sufficient to provide valuable software module extraction.

CHAPTER 5. RELATED RESEARCH

Using graphs to model software has been researched significantly. Some of these graphs, such as the Program Dependence Graph (PDG) have been in use since the late 1980s as a method to optimize compilers. [7] Other research has focused on the automated extraction of software architecture from code. A summary of the existing research related to our hypothesis provides background and demonstrates the relevance of this research.

5.1 Star Diagrams: Designing Abstractions out of Existing Code

Recently several different research efforts have focused on methods of using functions and data types to visualize code. One such method of visualizing code is the star diagram, which is described in [3]. The author of [3] describes a visualization technique to help a developer reengineer existing legacy systems into an object-oriented system. The author points out the steps necessary to reengineer such a system. First, the programmer must identify all the uses of a data structure that is going to be encapsulated into object-oriented objects. Next, the programmer groups similar computations on the data structure. From there the programmer can plan his or her task and begin manipulating expressions.

The star diagram is a visualization that gives a software engineer a bottom-up view of the source code which allows for easier reengineering. The diagram, as shown in figure 5.1, starts with a single root node on the left that denotes all the references to the data structure under analysis. Each operation directly referencing that data structure is connected to the right of that root node by an edge. The operation consuming the result of this reference is connected to the right of those operations, creating a tree.

These star diagrams are a visualization technique but [3] doesn't propose automated pro-

cessing on those diagrams to perform the re-engineering. Nevertheless, the paper proposes a valuable visualization technique for software, and shows how proper visualization techniques can benefit developers. An automated process for refactoring code is described in [12].

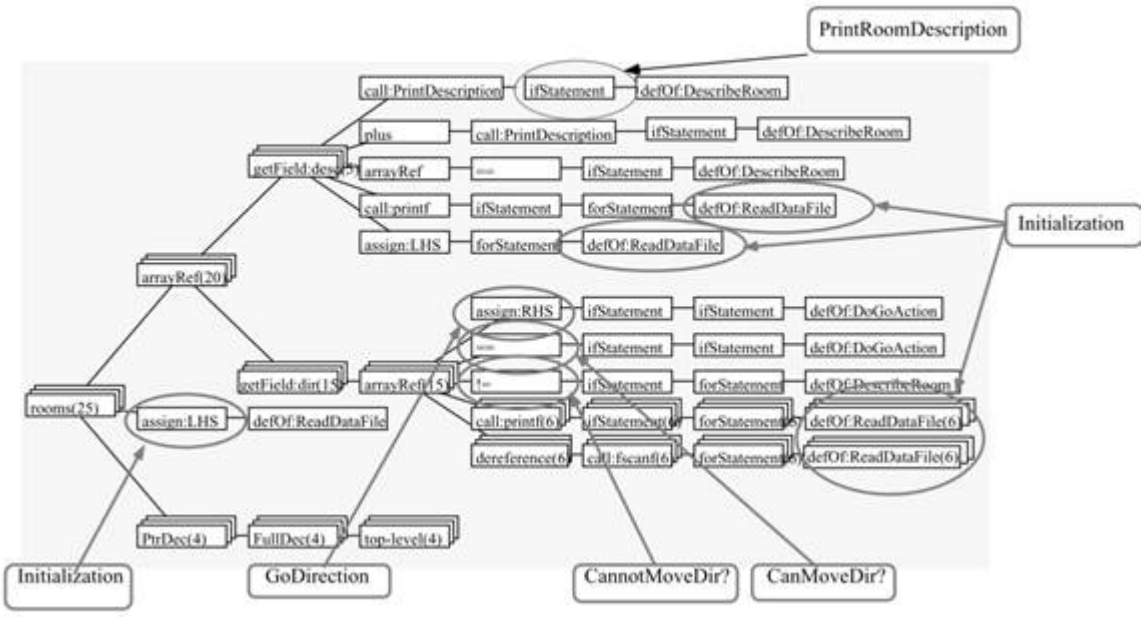

Figure 5.1 Star Diagram

5.2 Clustering Software Using Knowledgebase

As summarized in [1], software engineers frequently get source code as their most updated source of information about the software. For various reasons documentation can become outdated, limited or nonexistent. Therefore, [1] proposes using software clustering as a method of identifying subsystem structures. More specifically, [1] proposes the use of a knowledgebase to perform the clustering.

Knowledgebases are used in the artificial intelligence field. A knowledgebase is a set of data in the form of rules that describes knowledge about a topic. In the case of software the knowledgebase would be the repository of information describing how software entities might

be related and the weights assigned to those relations.

For instance, a knowledgebase might consider function calls, type usage, functions calling the same function, naming prefixes, etc. Each of these considerations are assigned a weight inside the knowledgebase. Building this knowledgebase is a tedious manual process, but once it is developed on generic entities it can be used across multiple software projects with minor modification.

Once the knowledgebase is developed it is then possible to measure the similarity between two software entities. Once the similarity between all software entities under examination is determined [1] creates a similarity matrix that is used to identify subsystems.

The knowledgebase identifies "soul clusters" from some obligatory subsystems with known properties. As an example given in [1] a library management system would have subsystems like "Book" or "Member". With soul clusters identified all other entities are compared with the soul clusters. If entities are similar enough to the soul clusters defined in the knowledgebase they are included in the soul cluster.

The remaining software entities are considered "candidate clusters." These candidate clusters are merged together based on their similarity measurement from the knowledgebase. Similarity matrices are built and clusters are joined iteratively until a threshold stopping criteria is met.

In this research, it is concluded that as a knowledgebase improves so does the accuracy of the software cluster. However, the process of building the knowledgebase can be labor intensive and still requires knowledge of the software in order to implement correctly.

5.3 Term Weighting Schemes for Labeling Clusters

Research done in [14] acknowledges that software clusters are valuable to developers, but recognizes that those clusters could be difficult to understand if they aren't labeled correctly. To overcome this difficulty [14] proposes the use of a term weighting scheme to automate cluster labeling.

Term weighting schemes are used in the information retrieval field to weight terms based on their importance in a document. Four different term weighting schemes, Inverse Document

Frequency (IDF), RF, Odds Ratio (OR), and chi-square, were analyzed using clusters from the CD_Net and Xfig_d software systems. Terms used in software artifacts like the NET term in a function NET_SV_AddrToString were weighted, with the heaviest weighted terms becoming the cluster label. In their analysis the chi-squared method obtained the most meaningful labels.

5.4 Weighted Combined Algorithm

As described in [11], research has been done exploring the use of clustering techniques for reverse engineering and software architecture recovery. There are several available similarity measures, but those need to be tailored to software. Therefore, [11] examines a variety of those measures and proposes a new algorithm for finding inter-cluster distance.

To create any clusters, first the similarity between entities must be measured. Similarity can either be based on a direct link or sibling link approach. The direct link measures how close two entities are related, such as a function calling another function. A sibling link approach measure similarity based on shared features. In this case software entities can be viewed as graph nodes sharing edges that represent features the nodes have in common. The sibling link approach lends itself well to software and is therefore used in [11].

After the type of similarity is decided there are several similarity measures available to measure how similar the entities are. As with other research, association coefficients, distance measures, and correlation coefficients are considered. Association coefficients calculate similarity based on binary features: either a feature is present or it isn't. Common association coefficient methods include the Jaccard coefficient, simple coefficient, and Sorensen-Dice coefficient. Likewise various distance measures are considered, including the Euclidean distance, Canberra distance, and Minkowski distance. Distance measures calculate the dissimilarity between entities. The Pearson product moment correlation coefficient is also presented.

After the measure of distance between entities has been calculated, the entities can be clustered. Maqbool [11] asserts that clustering algorithms fall into two categories: partitional and hierarical. Partitional algorithms start with an initial partition and then modify it iteratively until the final partitions are found. Hierarchical algorithms, on the other hand, can either build clusters by starting with nothing and iteratively connecting nodes, or starting with everything

connected and splitting nodes until clusters are formed. Several different linkage algorithms exist to measure a cluster's distance from other clusters, including single linkage, complete linkage, weighted average linkage, and un-weighted average linkage.

To be able to claim successful clustering there must be a method available to assess or validate the performance of a clustering algorithm. Maqbool [11] summarizes three common validation studies: external assessment, internal assessment, and relative assessment. With external assessment the results of the clustering algorithm are compared to an expert decomposition obtained by a subject matter expert of the code under analysis. Precision and recall is a common method of comparing that expert decomposition with the algorithm's results. The precision and recall method is described later in this paper because it is used to compare the results of this research to an expert decomposition.

Maqbool and Barbi [11] go on to summarize a limitation of their previously published combined algorithm and proposes the weighted combined algorithm to overcome that limitation. The weighted combined algorithm operates by creating a feature vector for each cluster which is the binary OR of the feature vector of each of the individual entities in the cluster. This feature vector includes information about what data is accessed by the entities.

Entity	Feature Vector
A	{1 1 0 0 0 }
B	{ 0 1 1 0 0}
C	{ 0 0 0 1 1}
D	{ 1 0 1 1 0}

Figure 5.2 Weighted Combined Algorithm Feature Vector

After the feature vector for each cluster is determined a similarity matrix using the Jaccard coefficient is created. The most similar entities in that matrix are clustered together. The weighted combined algorithm, unlike the combined algorithm, maintains the number of entities

in a cluster that access each specific feature instead of starting over each iteration.

	A	B	C	D
A	-	1/3	0	1/4
B	1/3	-	0	1/4
C	0	0	-	1/4
D	1/4	1/4	1/4	-

Entity	Feature Vector
AB	{1 1 1 0 0 }
C	{ 0 0 0 1 1}
D	{ 1 0 1 1 0}

Figure 5.3 Weighted Combined Algorithm Combined Feature Vector

In addition to proposing the weighted combined algorithm, [11] tests the algorithm using the code for Xfig, an open source drawing tool, and Bash, a Unix shell. The tests analyzed the weighted combined algorithm using a variety of similarity measures and compared the weighted combined algorithm to the complete algorithm. The weighted combined algorithm was shown to yield better results, judged by a higher precision/recall crossover point, than the complete algorithm as well as the combined algorithm.

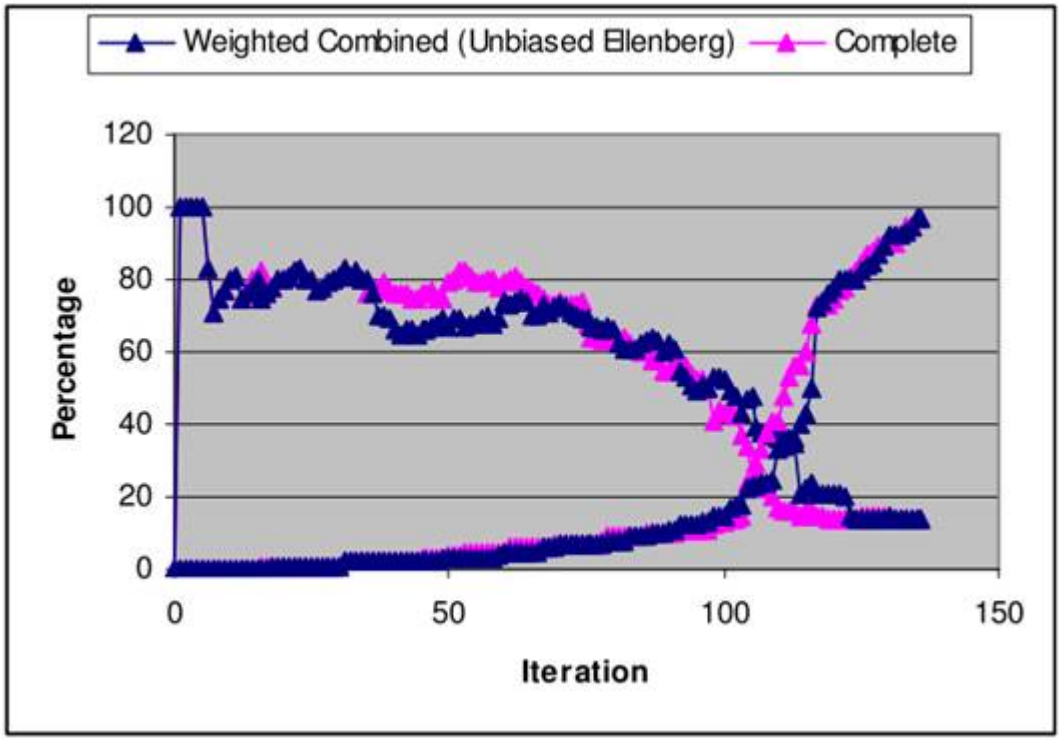

Figure 5.4 Weighted Combined Algorithm vs Complete, Xfig f_files Subsystem

5.5 Hierarchical Clustering for Software Architecture Recovery

The paper [10] reviews several different methods of using clustering for software architecture recovery. In doing so, it provides an analysis of the behavior of a variety of similarity and distance measures as they apply to software clustering. As an overview of clustering, [cite hierarchical clustering] introduces both formal and non-formal software features that can be used as entities for similarity analysis. Examples of formal features could be function calls or accessing variables. Non-formal features might be things like comments or the developer's name. Once the features to analyze are identified, clustering algorithms relate them through similarity measures like distance measures, correlation coefficients, and association coefficients. These similarity measures then allow the algorithms to cluster similar entities together until a threshold number of clusters are formed.

Some similarity measures will yield similar results between multiple entities. As discovered

experimentally in our research and presented in [10], arbitrary decisions in clustering can cause poor algorithm performance.

In addition to providing an overview of clustering methods, [10] includes a summary of comparative studies done between a variety of software clustering algorithms including the Single Linkage Algorithm (SLA), Complete Linkage Algorithm (CLA), Combined Algorithm (CA) and the Weighted Combined Algorithm (WCA) detailed above. In some of the studies summarized, WCA was shown to extract understandable software architecture. As such, it was analyzed and used in this research for comparison purposes.

CHAPTER 6. EXPERIMENTAL SETUP

To research graph-based clustering on software four experimental tools were created. They include a data mining tool, cluster builder tool, graph viewer tool, and cluster scoring tool.

The purpose of the data miner tool is to gather program artifacts from the software under analysis. As proposed, our graphs represent the relationships between functions using the types they access. The data mining tool, developed in Java, extracts those relationships from software using Ensoft's Atlas Java APIs. The Atlas tool indexes C code and then provides a query language to find program artifacts. The data miner tool was developed separately from other tools to increase the flexibility of this research. It outputs query results as a standard comma-separated values document that the cluster builder tool can use.

The software cluster builder tool was developed in Java with a purpose of doing analysis on the data created by the data mining tool. The clustering tool acts as a test bed for clustering algorithm development as described later in this paper. Rather than duplicating work to develop a graph implementation and related analysis framework, this research considered many open source graph implementation libraries. Notable libraries considered included the Java Universal Network/Graph Framework (JUNG) and JGraph. Due to scalability concerns and ease of implementing custom algorithms, JGraphT was chosen as the graph framework for the clustering tool. JGraphT is an open source Java library of graph structures and algorithm developed by Barak Neveh and contributors.

Within the software clustering tool, using the JGraphT library, a graph clustering algorithm framework was implemented. This framework allows for rapid prototyping of clustering algorithms using data output from the data mining tool. Output from the software clustering tool such as resulting clusters is logged to file and a visualization of the resulting clusters is rendered using the JGraph graph visualization and layout library.

The graph viewer tool was developed in Java and has a purpose of displaying the graph clusters for visual analysis. The graph viewer tool uses the JGraph graph visualization engine developed at the Swiss Federal Institute of Technology in Zurich. JGraph was ideal for this implementation because it separates out the layout, facade, and graph implementation to allow for real-time GUI-based manipulation of large graphs.

The cluster scoring tool is used to analyze the output of the cluster builder. This tool performs precision and recall analysis, comparing cluster output to manually computed truth data, to grade the clustering algorithm's correctness.

CHAPTER 7. PARTITIONAL ALGORITHM DEVELOPMENT

Mining data from software is useless without a method of parsing the data to reveal useful information. Even in small software projects the amount of mined data can be overwhelming, especially without domain knowledge of the software. The bulk of this research is finding and analyzing effective algorithms to use graphs for automated software architecture extraction from software code artifacts.

Once a software graph is created the architecture can be extracted using clustering. Clustering algorithms such as the Ford-Fulkerson algorithm finds the maximum flow of a graph. Combined with the max-flow min-cut theorem, the Ford-Fulkerson algorithm can be used to find perfect minimum cut clusters in a graph. However, the algorithm is computationally complex and is therefore impractical for use in even moderate or large software projects.

While there are several algorithms in various fields that provide mathematically correct clustering results, we chose to exploit relationships inherent in well-designed object-oriented code to create an algorithm that generates cluster approximations using a reasonable amount of processing power on large sets of software artifacts.

Throughout the course of researching our hypothesis our approximation algorithm underwent several iterations. The results of those iterations were compared with truth data gathered through domain knowledge of the software under analysis to provide improvements through the iterations.

Our algorithm iterations were tested using C code for the Xinu operating system. Xinu is a Unix-like operating system developed by Douglas Comer at Purdue University. Xinu is a small operating system, but with 263 functions and 65 types it provides a viable codebase for analysis. Those functions and types include operating system components like process, memory, I/O, timer management, and interprocess communications. [5]

7.1 Algorithm 1

The first algorithm attempt started with a connected graph and then relied on a connectivity analysis to identify the nodes and edges in the cluster.

In the Xinu operating system it was discovered that 52 of the 263 functions don't share any common data types with other functions in Xinu. These functions therefore are disconnected at the beginning of the clustering algorithm.

The remaining 211 functions are connected as a single large graph cluster that must be split to provide valuable information about the underlying software architecture. The algorithm operates by splitting the cluster(s) until each has below a threshold number of nodes. To split the clusters, this algorithm simply removes the edge with the smallest weight. Due to the splitting nature of this algorithm we refer to it as the partitional algorithm.

Removing the edge with the smallest weight resulted in valuable clusters, but also resulted in many "orphaned" nodes, nodes that would be in clusters by themselves. These orphaned nodes resulted in many small clusters which did little to further our goal of extracting the software architecture.

To illustrate, the below figure shows an example software graph. With this algorithm the minimum cut might be the edge between F and G. Instead, the algorithm would remove the edge between A and B, which would leave A in a cluster by itself.

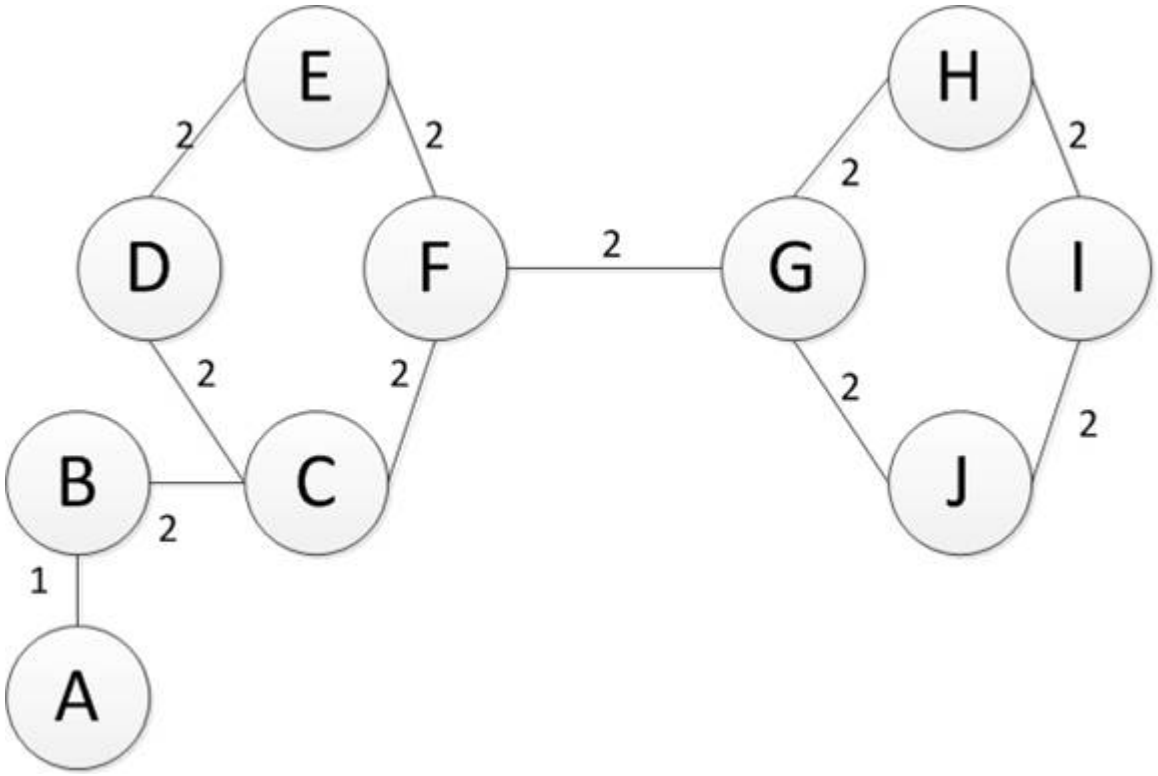

Figure 7.1 Partitional Algorithm 1 Example

7.2 Algorithm 2

For our second iteration of the partitional algorithm, the first algorithm was modified in an effort to solve the orphaned node issue. Instead of blindly removing the smallest weighted edge the algorithm would remove the smallest edge whose source and sink nodes both had at least two edges. In doing this, the algorithm will not create orphaned nodes as it did in the first iteration.

With this small change the algorithm was still only partially effective in removing orphaned nodes. Additionally, after comparison with the ideal solution developed using domain knowledge of Xinu, it became clear that the algorithm's output was far from correct.

The following illustration shows the inefficiency of algorithm two. Once again the ideal minimum cut would be the edge between F and G. Unlike partitional algorithm one, the edge between A and B wouldn't be removed. However, that would then leave a tie for other edges.

The algorithm as written would remove the edge between B and C because it was the first edge the algorithm came across.

In analysis of a real software project it was discovered that there are frequently ties for the smallest edge. Arbitrarily breaking those ties could produce sub-optimal results.

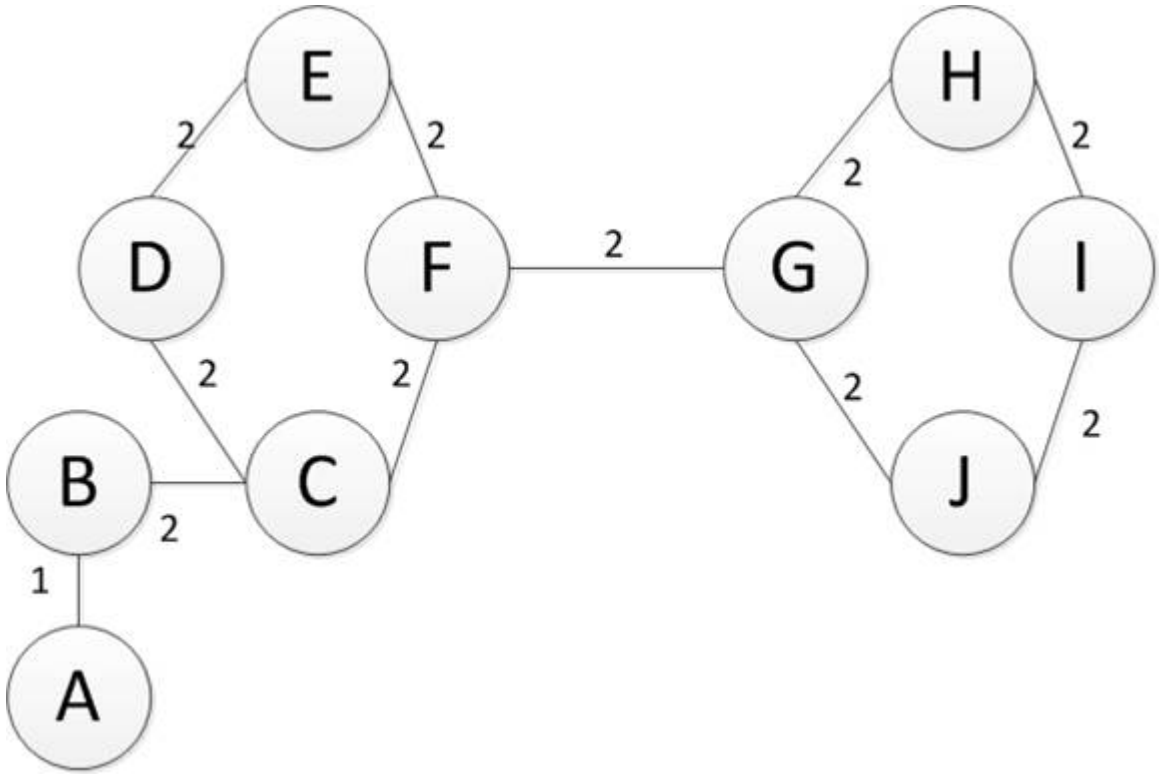

Figure 7.2 Partitional Algorithm 2 Example

7.3 Algorithm 3

One of the limitations of the second partitional algorithm implementation is that ties in edge weights aren't handled in a meaningful way. Instead, the first edge of the lowest weight that the algorithm comes across would be removed. As shown in the diagram illustrating the limitation of algorithm two this can be less than ideal.

In an attempt to make the output of the algorithm more meaningful for a developer wishing to extract the software architecture, details were added to the algorithm to determine what to do in a "tie" situation. In algorithm three the smallest weighted edge whose removal wouldn't

result in an orphaned node, and whose source and sink nodes had the fewest number of other edges would be removed, iteratively, until clusters were below a set threshold.

As an example consider the graph below. Algorithm three would treat edges with a weight of two as the smallest edges whose removal wouldn't orphan a node. To break the tie between edges with weight two, the algorithm looks at an edge x and then calculates how many edges the source and sink nodes for x have. For instance, in this graph, edge D-E would have a score of three because E has two edges and D has two edges. Note that the edge between D and E isn't counted twice.

Partitional algorithm implementation three would score edge D-E with a three as well as H-I and I-J. This is the lowest score so one of those edges would be removed. While this simple example illustrates the improvement in the algorithm, it still isn't an ideal grouping because removing those edges wouldn't result in a smaller cluster.

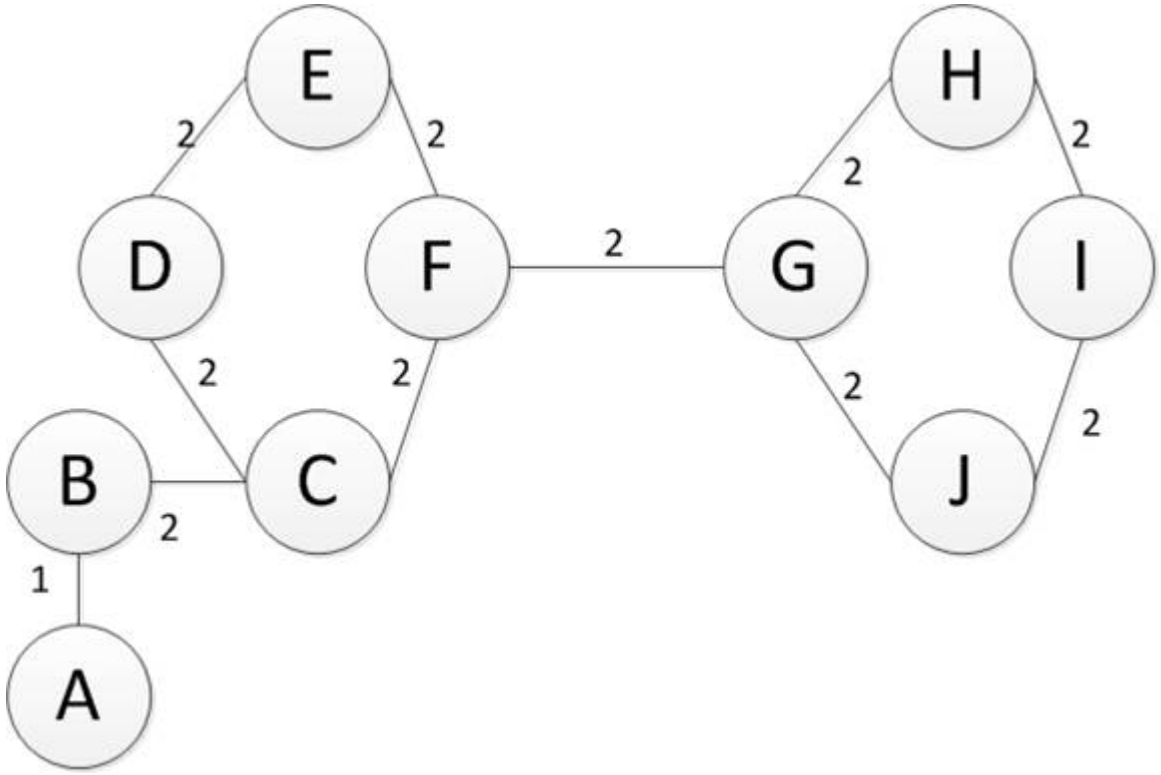

Figure 7.3 Partitional Algorithm 3 Example

7.4 Algorithm 4

Partitional algorithm iteration four improves on algorithm three by modifying the criteria for breaking "ties" in a graph. Algorithm four removes the smallest weighted edge that has the greatest ratio of edge weight to the total edge weight between source and sink nodes that doesn't result in an orphaned node.

In creating this algorithm it was theorized that functions that share many data types could be interface functions to a subsystem. These interface functions would tie two clusters together. Therefore, using the defined ratio would remove tied edges which would be more effective at separating subsystems into their own clusters.

To demonstrate, in the below graph algorithm four would treat edges with a weight of two as the smallest edges whose removal wouldn't orphan a node. To break the tie between all the edges of weight two, the algorithm looks at an edge x, and then calculates the ratio between the weight of edge x to the total weight of all edges connected to the source and sink of x. For instance, the score for the edge D-E would be calculated as:

$w(d-e)/(w(d-e)+w(e-f)+w(c-d)) = 2/6 = 1/3.$

The adjusted weights of edges are listed in red. Using these adjusted weights the algorithm would chose to remove edge F-G, because it has the smallest combined weight. In this case the removal of F-G is a minimum cut for the graph.

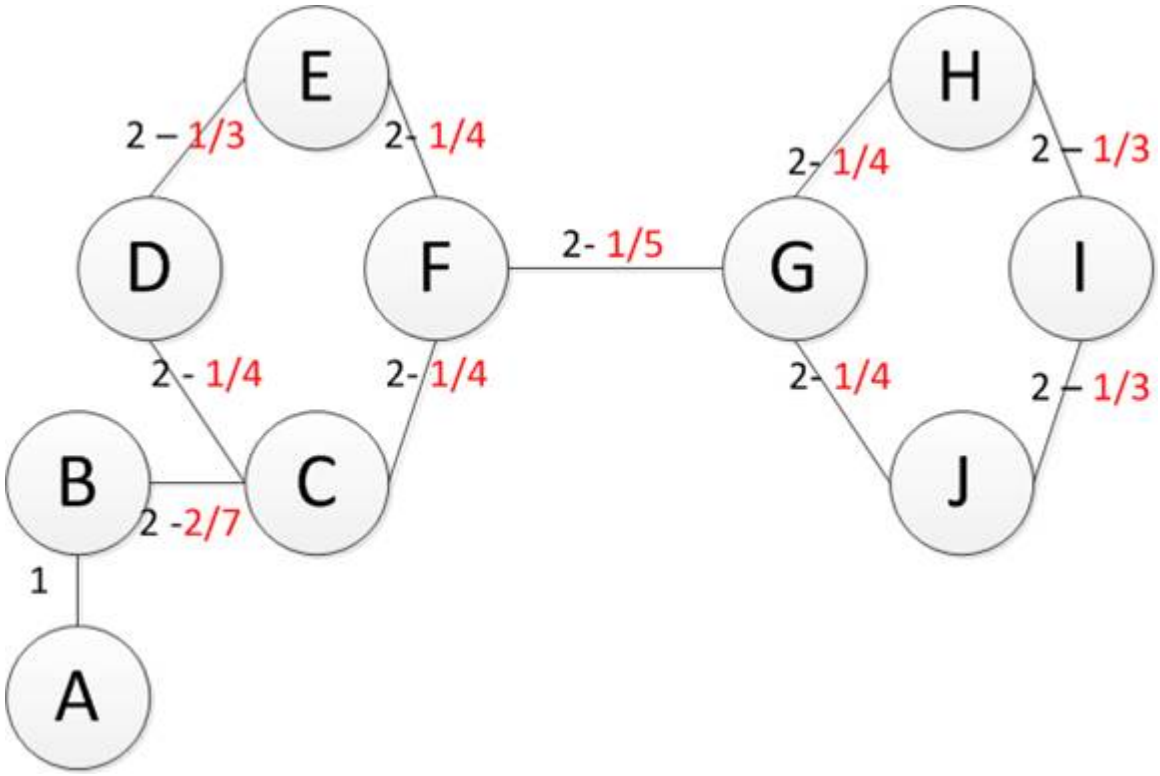

Figure 7.4 Partitional Algorithm 4 Example

CHAPTER 8. RESULTS

Using the data miner, graph cluster builder, graph viewer, and cluster scoring tools developed our hypothesis that software modules could be automatically extracted using graph-based clustering was tested. The tests were done using the Xinu operating system and subsystems of the Linux operating system. These tests provided for improvements in the partitional algorithm that allow it to successfully extract software modules.

8.1 Partitional Clustering Algorithm

As described in the Partitional Algorithm Development section, the partitional algorithm makes decisions to remove edges to split large clusters into smaller clusters until all clusters are below a threshold size.

Tuning this threshold size to its optimal was done first through manual inspection and then, as described later in this paper, through precision and recall analysis. Increasing the threshold size increases the number of functions in a cluster, but decreases the overall number of clusters the algorithm will produce. The goal is to have that threshold so the resulting clusters would describe subsystems of the software.

Appendix Table A.1 shows the clusters found when applying the partitional clustering algorithm to the Xinu operating system using a cluster threshold of 15. The types (edges) used to make up each cluster are displayed to reveal details of how the algorithm made clustering decisions. Because the partitional algorithm is based on graphs and a graph viewer tool was developed, output from the algorithm can be displayed pictorially, as shown in figure 8.1.

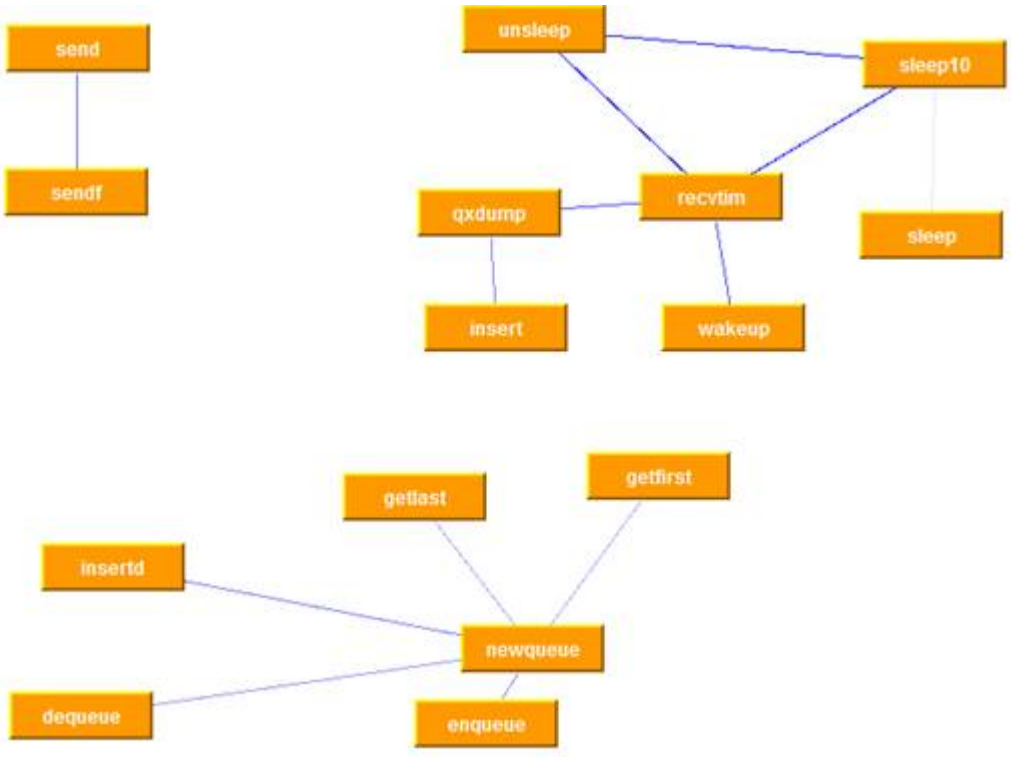

Figure 8.1 Graph Viewer Output of Sample Clusters

While these results intuitively seem valuable there is a desire to quantify their correctness. Such quantification can score the algorithm for comparison to other algorithms or to variations of the same algorithm.

As used in [11], the scoring of this algorithm's results is done using the precision and recall measure. Precision and recall is a measure of relevance used in the pattern recognition and information retrieval field. The measure compares a data set to a truth set in two different ways: precision, which is the fraction of retrieved data that is relevant; and recall, which is the fraction of relevant data that is retrieved.

$Precision = (PairsinTest \cap PairsinTruthData)/(PairsinTest)$

$Recall = (PairsinTest \cap PairsinTruthData)/(PairsinTruthData)$

If only singleton, or orphaned, nodes exist in clusters it would be easy to see that the results would be zero recall but 100 percent precision. Likewise if there were only one large cluster the system would have 100 percent recall but zero percent precision. Precision naturally decreases

as the size of clusters increases because more of the nodes in the cluster aren't relevant when compared to truth data. Inversely, recall naturally increases as cluster size increases because smaller clusters can mean less relevant data per cluster. The point where recall and precision intersect is important because it can reveal the optimal tuning for cluster size. Alternatively, computing the f-score of the precision and recall can be used to measure the overall accuracy. The f-score is the weighted average of precision and recall, and can be computed as:

$f - score = 2 * (precision * recall)/(precision + recall)$

To judge the effectiveness of any algorithm there must be truth data to which it can be compared. In the case of Xinu, partial truth data of the major subsystems was gathered from experts with domain knowledge in the software.

Table 8.1 Xinu Subsystem Truth Data

Subsystem	**Functions**
Process	chprio, create, ctxsw, kill
Files	ckmode, dfalloc, dfdsrch, ibclear, ibfree, ibget, ibfree, ibnew, ibput, lfclose, lfgetc, lfinit, lfputc, lfread, lfsdfree, lfseek, lfsetup, lfsflush, lfsnewd, lfwrite
Disks	dscntl, dsinit, dsinter, dskenq, dskqopt, dskstrt, dsopen, dsread, dsseek, dswrite
TTY	ttycntl, ttygetc, ttyiin, ttyinit, ttyoin, ttyopen, ttyputc, ttyread, ttywrite
Networking	arp_in, arp_find, ethinit, ethinter,ethread, ethrstrt, ethwrite, ethwstrt, icmp_in, ip_in, ipsend, rap_in, mkarp, netin, netdump, netinit, dgalloc, dgclose, dgcntl, dgdump, dginit, dgread, dgwrite, route, udpecho, udpsend
Utility - cross-cutting	close, conf, freebuf, getbuf, getpid, getpath, getnet, getname, getprio, ioerr, kprintf, open, panic, putc, read, ready. Resched, wait, signal, suspend
Messaging	pcount, preate, pdelete, pinit, perceive, psend, preset
Semaphores	receive, send, scount, screate, sdelete, signal, wait, signal, sreset

This truth data, while only a partial set of functions in Xinu, was used to determine how effective the researched algorithm was. Our cluster scoring tool can then be used to compute precision, recall, and f-score measuring the pairs of functions, in the same cluster, that were present in both truth data and experimental data.

Table 8.2 Partitional Algorithm Precision and Recall Score

Threshold	**Precision**	**Recall**	**F-Score**
2	0.213115	0.01656051	0.030733
5	0.198413	0.063694268	0.096432
10	0.218519	0.150318471	0.178113
15	0.191489	0.206369427	0.198651
20	0.189641	0.303184713	0.233333
25	0.194872	0.338853503	0.247442
30	0.155738	0.338853503	0.213398
35	0.146315	0.346496815	0.205749
40	0.148746	0.445859873	0.223072
45	0.128019	0.472611465	0.201466
50	0.128019	0.472611465	0.201466
		Maximum:	0.247442

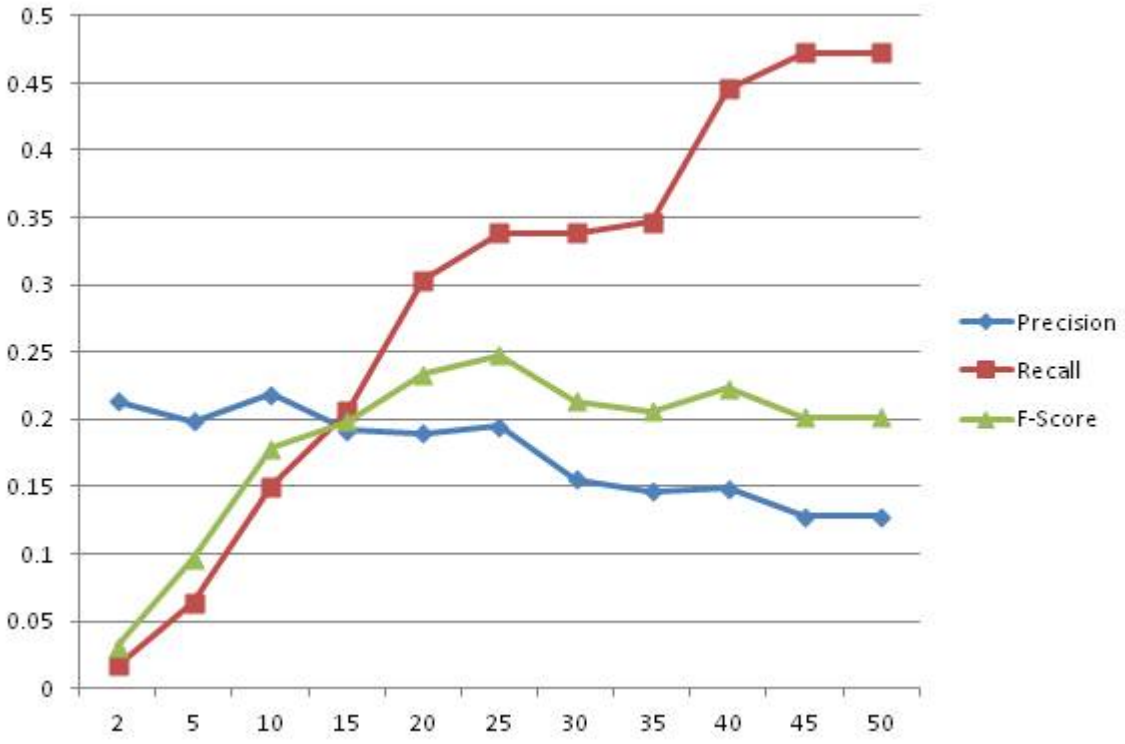

Figure 8.2 Precision, Recall and F-Score of Xinu Analysis

Analysis shows, as expected, an intersection of precision and recall. This intersection, shown when all clusters have less than 15 nodes in them, is used as the optimal tuning threshold by some research. However, due to the fact that recall increases quicker than precision decreases, the maximum f-score actual occurs when 25 is used as the algorithm threshold.

8.2 Weighted Combined Algorithm

As described in the related works section, related research has analyzed methods of visualizing software and methods of using clustering to understand software. Some research, like that done on the weighted combined algorithm, examines the use of clustering algorithms to extract software modules.

The partitional algorithm uses connected graphs as the basis of clustering for the automated extraction of software modules. The weighted combined algorithm also uses clustering, though not based on graphs, to extract software modules. To provide a basis for supporting our

hypothesis, the weighted combined algorithm was implemented in Java using the Atlas API and used to analyze the same Xinu operating system code the partitional algorithm analyzed.

The weighted combined algorithm starts with stand-alone software artifacts and then iteratively joins those artifacts together to form clusters. The number of iterations the algorithm goes through is used to tune the algorithm's performance.

As expected, the weighted combined algorithms provided valuable clusters that give an approximate representation of the software's underlying modules. These clusters are can be found in Appendix B.1.

As with the partitional algorithm, the precision and recall analysis method was applied to compare the results of the weighted combined algorithm to the truth data found in table 8.1.

Table 8.3 Weighted combined Algorithm Precision and Recall Score

Iterations	**Precision**	**Recall**	**F-Score**
30	0.205128	0.010191083	0.019417
60	0.213592	0.028025478	0.04955
90	0.194286	0.043312102	0.070833
120	0.251479	0.108280255	0.15138
135	0.275269	0.163057325	0.2048
150	0.223438	0.182165605	0.200702
165	0.204983	0.230573248	0.217026
180	0.17475	0.289171975	0.21785
210	0.026766	0.755414013	0.0517
		Maximum:	0.21785

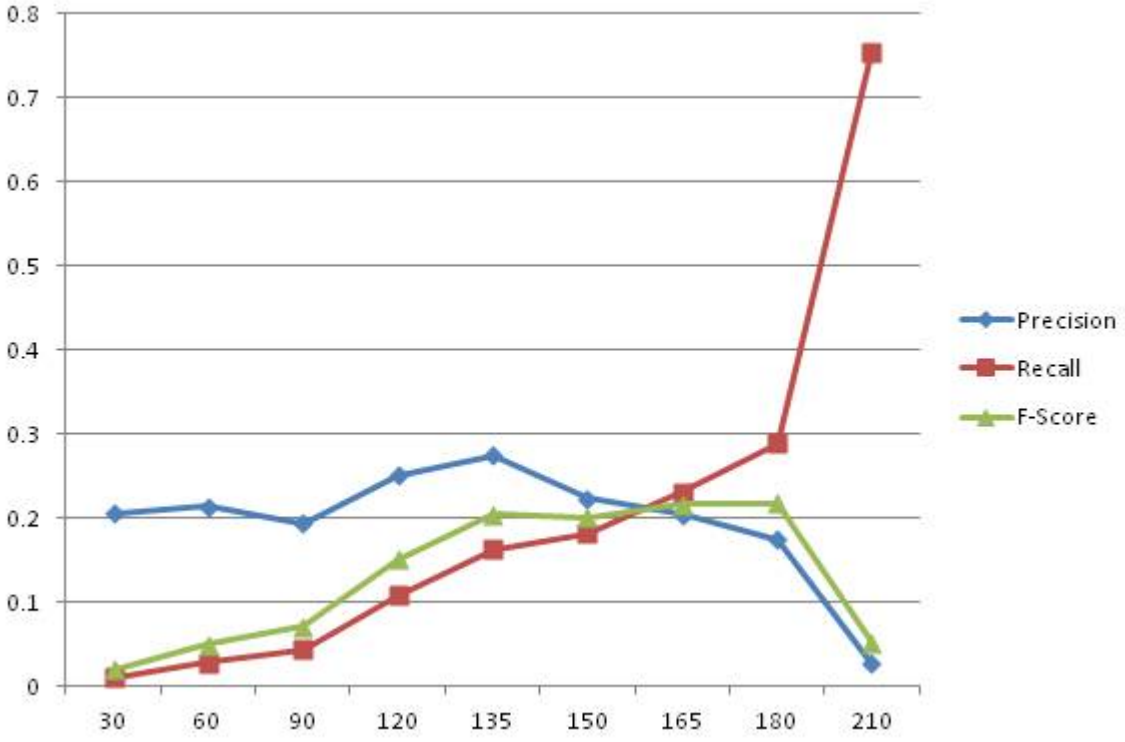

Figure 8.3 Weighted Combined Algorithm Precision, Recall and F-Score

CHAPTER 9. CONCLUSIONS AND FUTURE WORK

An overarching goal of this research was to develop methods of using tools to aid in the development and maintenance of complex software systems. Since developers frequently find themselves with large amounts of code and only small amounts of documentation or domain knowledge, effective methods of extracting software modules from code would be of great value to the developer.

It was hypothesized that graph-based clustering algorithms could be used to automatically extract software modules. After identifying the software relationships to visualize using graphs, we examined methods of clustering those graphs.

This research developed a set of tools for software data mining, visualization, clustering, and cluster evaluation. These tools allowed us to develop an algorithm showing that software module extraction is possible through graph clustering.

After a survey of existing literature, we found research done on methods of visualizing software and research on using clustering for module extraction. We found no research that used graphs to create software clusters for module extraction. The weighted combined algorithm uses a different method for module extraction so we implemented that algorithm as a baseline to validate our hypothesis.

To quantitatively validate our hypothesis that graph-based clustering can be used for software modules extraction we used the precision and recall method of pattern recognition to compare the results from the partitional algorithm and weighted combined algorithm to truth data.

Results indicate that our algorithm operates with comparable accuracy to the weighted combined algorithm and that these results are valuable for module extraction. Furthermore, by using graph-based clustering we are able to view results pictorially. The automatically-

extracted modules combined with a graph visualization of results gives developers knowledge of software that was previously only available to those with significant domain knowledge or extensive documentation.

While our hypothesis was validated, the notion of using graph-based software clustering for software module extraction is new, which gives many opportunities for future work. This research noted that the partitional algorithm and the weighted combined algorithm shared a precision and recall crossover point that indicates differences between the algorithms. Future research could compare the strengths and weaknesses of the two algorithms in an attempt to develop a more accurate hybrid algorithm. Additionally, it may be possible to combine several software relationships together with the type-use relationship this research used to provide more optimal module extraction.

This and other related research has shown that software clustering can be used effectively to extract software modules. This capability, especially if enhanced through future research, can increase a developer's ability to maintain and develop complex software systems.

APPENDIX A. PARTITIONAL ALGORITHM CLUSTER RESULTS

Table A.1 Paritional Algorithm Xinu Clusters

Cluster	**Elements**
Cluster 1:	x_date getutim
Types in cluster 1:	clktime
Cluster 2:	control remove rename access
Types in cluster 2:	devsw.dvcntl devtab devsw
Cluster 3:	_mkinit mark
Types in cluster 3:	mkmutex nmarks
Cluster 4:	pdelete psend _ptclear freebuf bpdump pinit pcreate getbuf preceive poolinit mkpool pcount x_bpool preset
Types in cluster 4:	ports pt.ptstate pt ptmark MARKER marks nmarks pt.ptssem pt.pthead pt.pttail ptnode pt.ptrsem pt.ptseq ptnode.ptnext ptnode.ptmsg ptfree pt.ptmaxcnt bptab nbpools bpool.bpnext bpool.bpsem bpool bpmark bpool.bpsize ptnextp
Cluster 5:	mount naminit ndump unmount namrepl mprint
Types in cluster 5:	nament.nrepl nam Nam.nnames Nam nament nam.nametab nament.ndev nam.nnames nament.npre Nam.nametab devsw.dvname devtab devsw
Cluster 6:	signaln sreset scount screate signal sdelete
Types in cluster 6:	sentry.semcnt semaph sentry.sstate sentry.sqhead sentry
Cluster 7:	dskbcpy dumkdl
Types in cluster 7:	dskdbp
Cluster 8:	suspend resume
Types in cluster 8:	pentry.pstate proctab pentry pentry.pprio

Table A.1 (Continued)

Paritional Algorithm Xinu Clusters Continued

Cluster 9:	enqueue insertd getlast newqueue dequeue getfirst
Types in cluster 9:	qent.qprev qent q qent.qnext qent.qkey
Cluster 10:	clkinit stopclk strtclk
Types in cluster 10:	preempt defclk clkdiff clockq slnempty
Cluster 11:	send sendf
Types in cluster 11:	pentry.phasmsg pentry.pstate proctab Bool pentry pentry.pmsg
Cluster 12:	getpid recvclr receive
Types in cluster 12:	currpid pentry.phasmsg proctab Bool pentry pentry.pmsg
Cluster 13:	sleep10 unsleep wakeup recvtim sleep insert qxdump
Types in cluster 13:	clkruns pentry.pstate qent.qkey proctab qent sltop q qent.qnext clockq pentry slnempty currpid qent.qprev
Cluster 14:	setdev setnok kill addarg
Types in cluster 14:	proctab pentry pentry.pdevs pentry.pnxtkin pentry.pstate pentry.pbase
Cluster 15:	freemem mdump getmem getstk nulluser x_snap x_mem
Types in cluster 15:	memlist.mnext memlist mblock.mnext maxaddr mblock.mlen end mblock pentry.pstate edata etext pentry proctab pentry.pstklen
Cluster 16:	sysinit newsem chprio pxdump resched getprio x_ps wait ready newpid create
Types in cluster 16:	pentry.pstate sentry.sqtail pentry proctab semaph sentry.sstate sentry currpid nextsem numproc Bool pentry.plimit pentry.phasmsg pentry.pprio pentry.pargs pentry.pname pentry.paddr pentry.pbase nextproc rdyhead pentry.psem pentry.pregs qent q
Cluster 17:	rfread
Types in cluster 17:	-
Cluster 18:	getc
Types in cluster 18:	-
Cluster 19:	putc
Types in cluster 19:	-

Table A.1 (Continued)

Paritional Algorithm Xinu Clusters Con't

Cluster 20:	x_mount init iosetvec lfwrite rfwrite seek close lfread open ioinit x_devs dscntl write read devdump
Types in cluster 20:	devsw.dvname devtab devsw devsw.dvovec devsw.dvivec devsw.dvminor devsw.dvioblk devsw.dvcsr
Cluster 21:	rfcntl rfmkpac rfalloc rfopen rfio rfinit rfclose rfseek rfdump x_rf rfsend
Types in cluster 21:	rfinfo.device Rf.device Rf rfinfo rfinfo.rmutex Rf.rmutex Rf.rftab rfblk.rf_state rfinfo.rftab rfblk rfblk.rf_pos rfblk.rf_mode rfblk.rf_name rfblk.rf_dnum rfblk.rf_mutex devsw.dvioblk devsw
Cluster 22:	dsksync dsinter dsread dskqopt dsseek dswrite dsinit dskenq dskstrt
Types in cluster 22:	dreq dreq.drop dreq.drdba dreq.drpid dreq.drstat dreq.drbuff DBADDR dreq.drnext dsblk dsblk.dreqlst devsw.dvioblk dskrbp currpid devsw dtc.dt_csr dsblk.dcsr dtc xbdcb.xcntl xbdcb.xop xbdcb.xladdr dtc.dt_xdar dsblk.ddcb dtc.dt_car xbdcb.xunit xbdcb.xcount dtc.dt_xcar xbdcb xbdcb.xmaddr dtc.dt_dar
Cluster 23:	lfinit dfalloc
Types in cluster 23:	flblk flblk.fl_pid fltab
Cluster 24:	ibget ibput
Types in cluster 24:	dskdbp IBADDR iblk
Cluster 25:	lfsnewd lfsdfree
Types in cluster 25:	dsblk dir.d_fblst devsw.dvioblk freeblk dsblk.dflsem dir freeblk.fbnext DBADDR devtab dsblk.ddir devsw
Cluster 26:	dsopen dumkfs dfdsrch ibnew lfsetup lfputc ibfree ibclear dumkil iblfree lfgetc lfclose dudir lfseek lfsflush
Types in cluster 26:	flblk Bool flblk.fl_mode devsw.dvioblk flblk.fl_pos flblk.fl_dch flblk.fl_dev fdes flblk.fl_dent devsw flblk.fl_iba flblk.fl_bptr iblk fdes.fdiba flblk.fl_buff iblk.ib_dba flblk.fl_iblk DBADDR IBADDR flblk.fl_ipnum dsblk devtab dsblk.ddir dir flblk.fl_iblk.ib_dba dir.d_fblst dir.d_nfiles dir.d_id dir.d_iblks dir.d_filst fdes.fdname dir.d_files fdes.fdlen iblk.ib_next dsblk.dflsem iblk.ib_byte flblk.fl_pid currpid

Table A.1 (Continued)

Paritional Algorithm Xinu Clusters Con't

Cluster 27:	x_rls echoch x_creat
Types in cluster 27:	Bool
Cluster 28:	ttyiin ttyoin tdump1 ttycntl ttywrite rststate erase1 ttyinit kputc ttyread writcopy ttygetc ttyputc savestate eputc
Types in cluster 28:	Bool tty.oflow tty.oheld csr.ctstat tty.ostop tty csr.crbuf csr tty.ibuff tty.iecho tty.ihead tty.isem tty.ioaddr tty.imode tty.iintpid tty.iintr tty.ieof tty.ieofc tty.ierase tty.iintrc tty.ikill tty.icursor tty.ierasec tty.ifullc tty.ostart tty.ikillc tty.icrlf csr.ctbuf tty.obuff tty.otail tty.osem tty.ebuff tty.ehead tty.odsend tty.etail tty.itail tty.ohead devsw.dvminor devsw devtab devsw.dvcsr csr.crstat saveps savedev savectstat savecrstat tty.evis tty.ieback devsw.dvioblk tty.ocrlf
Cluster 29:	rfputc ttyopen ethstrt ethinit rfgetc ethread ethwstrt ethrstrt ethinter
Types in cluster 29:	devsw.dvnum devsw etblk.ercmd etblk.eioaddr dcmd.dc_st1 etblk dcmd dqregs dqregs.d_csr etblk.etdev Eaddr etblk.etpaddr dqsetu etblk.etrpid devsw.dvioblk etblk.etrsem etblk.ewcmd dcmd.dc_st2 dcmd.dc_buf etblk.etwsem dcmd.dc_flag dcmd.dc_bufh etblk.etlen dqregs.d_wcmd etblk.etwtry dqregs.d_wcmdh etblk.etsetup dcmd.dc_len dqregs.d_rcmd dqregs.d_rcmdh
Cluster 30:	shell lexan x_help
Types in cluster 30:	cmds Shl shvars cmdent cmdent.cmdnam shvars.shncmds Shl.shncmds shvars.shtok Shl.shtktyp Shl.shtok shvars.shtktyp
Cluster 31:	dgparse dgdump x_dg
Types in cluster 31:	dgblk.dg_faddr dgblk IPaddr dgblk.dg_fport dgblk.dg_mode dgblk.dg_state dgtab dgblk.dg_lport dgblk.dg_dnum dgblk.dg_xport
Cluster 32:	icmp_in netout
Types in cluster 32:	epacket.ep_data epacket ip ip.i_paclen ip.i_dest IPaddr

Table A.1 (Continued)

Paritional Algorithm Xinu Clusters Con't

Cluster 33:	arpfind adump x_routes arpinit
Types in cluster 33:	Arp.atabnxt arpent.arp_Ead arpent arpent.arp_dev Arp.atabsiz Arp Eaddr arpblk.atabsiz arpblk.atabnxt Arp.arptab IPaddr arpblk arpblk.arptab arpent.arp_Iad arpent.arp_state st
Cluster 34:	mkarp dgmcntl getpath getnet route ipsend getaddr arp_in sndrarp ethwrite rarp_in netin
Types in cluster 34:	eheader IPaddr eheader.e_ptype epacket.ep_data epacket epacket.ep_hdr epacket.ep_hdr.e_ptype arppak.ar_op arppak.ar_tha arppak.ar_tpa etblk.etpaddr arppak.ar_sha Net epacket.ep_hdr.e_dest arppak Eaddr netinfo arppak.ar_spa etblk eheader.e_dest netinfo.netpool Net.netpool Net.gateway netinfo.gateway arpent arpblk.arpwant Arp.arpwant Arp.arppid Arp Arp.arptab arpblk arpblk.arppid arpblk.arptab arpent.arp_state Bool Net.mavalid netinfo.mavalid netinfo.mynet Net.mynet arpent.arp_Ead arpent.arp_dev netinfo.myaddr Net.myaddr devsw.dvioblk devtab devsw eheader.e_src arpblk.rarppid Arp.rarppid
Cluster 35:	dgmopen dgclose dginit dgalloc
Types in cluster 35:	dgblk.dg_state dgblk.dg_dnum dgtab dgblk Bool netq netinfo dgblk.dg_netq netq.valid netq.uport netinfo.netqs Net Net.netqs
Cluster 36:	ip_in x_net nqalloc netdump getname udpnxtp netinit
Types in cluster 36:	netq.xport netq netinfo Net.nover netq.uport netinfo.netqs netinfo.nover Net netinfo.ndrop Net.netqs netq.pid Net.ndrop Bool netq.valid Net.nmutex Net.mavalid netinfo.nxtprt netinfo.npacket netinfo.netpool netinfo.nmutex netinfo.mavalid Net.npacket Net.nxtprt Net.netpool netinfo.mnvalid Net.mnvalid

Table A.1 (Continued)

Paritional Algorithm Xinu Clusters Con't

Cluster 37: Types in cluster 37:	udpsend rwhod dgcntl dgread x_who rwhoind login dgwrite x_uptime ip.i_data udp.u_udplen udp.u_data udp epacket.ep_data epacket ip udp.u_sport rwent.rwmach rwent Rwho.rwnent rwhopac.rw_btim rwent.rwslast rwhopac.rw_load rwinfo rwent.rwusers rwent.rwload rwent.rwlast rwinfo.rwnent rwinfo.rwcache rwhopac.rw_sndtim rwhopac Rwho.rwcache rwhopac.rw_host Rwho rwent.rwboot Shl Bool shvars.shlast shvars.shlogon Shl.shuser MARKER Shl.shmark shvars.shuser Shl.shlast shvars shvars.shmark Shl.shused Shl.shlogon shvars.shused marks nmarks netinfo netinfo.netpool IPaddr Net.netpool Net dgblk devsw.dvioblk dgblk.dg_xport dgblk.dg_mode devsw xgram.xg_faddr dgblk.dg_lport xgram.xg_data xgram.xg_fport xgram

APPENDIX B. WEIGHTED COMBINED ALGORITHM RESULTS

Table B.1 Weighted Combined Algorithm Xinu Clusters

Cluster	**Functions**
Cluster 0:	Qdumph
Cluster 1:	Sndrarp
Cluster 2:	hl2vax
Cluster 3:	Userret
Cluster 4:	x_rm
Cluster 5:	Getpid
Cluster 6:	Sleep
Cluster 7:	Namopen
Cluster 8:	x_rls
Cluster 9:	Tqdump
Cluster 10:	Qdumpa
Cluster 11:	Iosetvec
Cluster 12:	Ckmode
Cluster 13:	net2hs
Cluster 14:	Udpecho
Cluster 15:	Blkcopy
Cluster 16:	Ionull
Cluster 17:	Getname
Cluster 18:	Ctxsw
Cluster 19:	Prdumpa
Cluster 20:	Cksum
Cluster 21:	vax2hl

Table B.1 (Continued)

Weighted Combined Algorithm Xinu Clusters Continued

Cluster 22:	Rfmkpac
Cluster 23:	Dgmcntl
Cluster 24:	Nulluser
Cluster 25:	Prdumph
Cluster 26:	x_snap
Cluster 27:	Tdump
Cluster 28:	Nammap
Cluster 29:	x_sleep
Cluster 30:	Outint
Cluster 31:	dgparse dgdump x_dg
Cluster 32:	x_reboot
Cluster 33:	Main
Cluster 34:	Udpsend
Cluster 35:	Clkinit
Cluster 36:	x_cp
Cluster 37:	Ascdate
Cluster 38:	rfcntl rfsend
Cluster 39:	Xdone
Cluster 40:	x_creat
Cluster 41:	Gettime
Cluster 42:	shell x_help
Cluster 43:	Kprintf
Cluster 44:	getutim x_date
Cluster 45:	Setclkr
Cluster 46:	dot2ip
Cluster 47:	Restart
Cluster 48:	Clkint
Cluster 49:	dginit lfinit dfalloc dgalloc
Cluster 50:	x_kill

Table B.1 (Continued)

Weighted Combined Algorithm Xinu Clusters Continued

Cluster 51:	Blkequ
Cluster 52:	icmp_in netout ipsend
Cluster 53:	Netnum
Cluster 54:	x_close
Cluster 55:	x_echo
Cluster 56:	x_unmou
Cluster 57:	lexan addarg
Cluster 58:	Dumkdl
Cluster 59:	x_mv
Cluster 60:	rwhod x_who rwhoind login x_uptime
Cluster 61:	Prdump
Cluster 62:	Dgcntl
Cluster 63:	net2hl
Cluster 64:	dsksync dsinter dsread dskqopt dsinit dswrite dsseek dskstrt dskenq
Cluster 65:	Tdumph
Cluster 66:	ethstrt ethinit ethread ethwstrt ethrstrt ethinter
Cluster 67:	_mkinit mark
Cluster 68:	freemem mdump getstk getmem x_mem
Cluster 69:	Inint
Cluster 70:	Panic
Cluster 71:	Ioerr
Cluster 72:	Rwho
Cluster 73:	ip2name
Cluster 74:	dsopen lfsnewd dumkfs dfdsrch ibnew lfsetup lfsdfree ibput lfputc ibfree iblfree lfclose lfgetc dudir lfseek lfsflush
Cluster 75:	ttyoin ttyiin ttycntl tdump1 rststate ttyinit ttyread kputc writcopy ttygetc ttyputc savestate eputc
Cluster 76:	x_exit
Cluster 77:	hl2net

Table B.1 (Continued)

Weighted Combined Algorithm Xinu Clusters Continued

Cluster 78:	setdev suspend setnok resume chprio getprio ready newpid
Cluster 79:	ibclear ibget dumkil dskbcpy
Cluster 80:	control ioinit remove rename ttywrite access
Cluster 81:	pdelete pcount psend _ptclear pinit pcreate preset perceive
Cluster 82:	Ethwrite
Cluster 83:	rfalloc rfopen rfio rfinit x_rf rfdump rfseek rfclose
Cluster 84:	x_cat
Cluster 85:	hs2net
Cluster 86:	mount naminit unmount ndump namrepl mprint
Cluster 87:	newsem signaln sreset scount screate signal sdelete
Cluster 88:	x_net ip_in dgmopen dgread nqalloc dgclose netdump udpnxtp netinit dgwrite
Cluster 89:	arpfind adump mkarp getpath getnet route x_routes arp_in getaddr rarp_in netin arpinit
Cluster 90:	Stopclk
Cluster 91:	putc getc write read seek
Cluster 92:	mkpool freebuf x_bpool bpdump getbuf poolinit
Cluster 93:	sleep10 sysinit send pxdump resched strtclk unsleep kill recvclr recvtim x_ps wait create dequeue enqueue insertd wakeup getlast sendf newqueue insert receive qxdump getfirst
Cluster 94:	Qdump
Cluster 95:	lfread ttyopen rfputc dscntl rfgetc rfread lfwrite rfwrite
Cluster 96:	open x_devs x_mount init devdump close
Cluster 97:	erase1 echoch

BIBLIOGRAPHY

[1] Adnan, M. N., Islam, M. R., and Hossain, S. (2011). Clustering software systems to identify subsystem structures using knowledgebase. *2011 Malaysian Conference in Software Engineering*, pages 445–450.

[2] Allen, E. (2002). Measuring graph abstractions of software: An information-theory approach. *Software Metrics, 2002. Proceedings. Eighth IEEE ...*.

[3] Bowdidge, R. (2007). Star diagrams: Designing abstractions out of existing code. *00 PS LA'96 Workshop on Transforming Legacy ...*, pages 1–5.

[4] Briand, L., Morasca, S., and Basili, V. (1996). Property-based software engineering measurement. *Software Engineering, IEEE ...*, 22(1).

[5] Comer, D. (2011). *Operating System Design: The Xinu Approach*. CRC Press.

[6] Edmonds, J. and Karp, R. M. (1972). Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264.

[7] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

[8] Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. *... of the International Symposium on Applied ...*, pages 1–10.

[9] Kleinberg, J. and Tardos, E. (2006). *Algorithm Design*. Addison-Wesley.

[10] Maqbool, O. (2007). Hierarchical clustering for software architecture recovery. *Software Engineering, IEEE*, 33(11):759–780.

[11] Maqbool, O. and Babri, H. (2004). The weighted combined algorithm: a linkage algorithm for software clustering. *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 15–24.

[12] O'Connor, A., Shonle, M., and Griswold, W. (2005). Star diagram with automated refactorings for Eclipse. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange - eclipse '05*, pages 16–20.

[13] Sartipi, K. and Kontogiannis, K. (2003). On modeling software architecture recovery as graph matching. *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 224–234.

[14] Siddique, F. and Maqbool, O. (2011). Analyzing Term Weighting Schemes for Labeling Software Clusters. *2011 15th European Conference on Software Maintenance and Reengineering*, pages 85–88.