Using Automatic Clustering to Produce High-Level System Organizations of Source Code

S. Mancoridis, B. S. Mitchell, C. Rorres
Department of Mathematics & Computer Science
Drexel University
Philadelphia, PA, USA
{smancori, bmitchel, crorres}@mcs.drexel.edu

Y. Chen, E. R. Gansner
AT&T Labs - Research
Florham Park, NJ, USA
{chen,erg}@research.att.com

Abstract

This paper describes a collection of algorithms that we developed, and implemented as a tool, to facilitate the automatic recovery of the hierarchical structure of a software system from its source code. These automatically-produced structural views provide valuable insight to software professionals who are trying to understand the intricate organization of complex systems.

Keywords: Clustering, Program Understanding, Reverse Engineering, Software Structure, Optimization, Genetic Algorithms.

1 Introduction

One aspect that makes software maintenance an arduous task is the difficulty associated with understanding the intricate relationships that exist between the source code components. Frequently, this problem is exacerbated because the design documentation is out-of-date and the original system architect is no longer available for consultation. With no mechanism for gaining insight into the system design and structure, the software maintenance practitioner is often forced to make modifications to the source code without a thorough understanding of the system's organization. Because the requirements of heavily used software systems tend to change, it is inevitable that continually adopting an "ad-hoc" approach to maintenance will have a negative effect on the overall modularity of the system. Over time, the system structure may deteriorate to the point where the source code organization is so chaotic that it needs to be radically overhauled or abandoned.

Software Engineers have long known of the difficulties associated with maintaining software systems whose only documentation is limited to the source code. While software maintenance strategies based directly on source code are feasible for small systems; the size of many interesting software systems is often beyond a programmer's cognitive ability to determine the affect of a local change on the entire system. One way programmers typically cope with this complexity is by grouping (clustering) related procedures and their associated data into modules and classes. These clustering strategies, if done well, can provide valuable insight into the comprehension of a program because they capture the functional and data boundaries that exist within a software system.

While modules and classes do much to improve software development and maintenance practices, they are insufficient for supporting the design and ongoing maintenance of complex systems.

Complex systems often contain several hundreds of thousands of lines of code that are packaged into a large number of cooperating modules and classes. Fortunately, we often find that these systems are organized into identifiable clusters of modules and classes, called subsystems, that collaborate to achieve a higher-level system behavior[3]. By clustering the related modules and classes into subsystems, a higher-level organization of the software system is created that helps developers understand the structure of complex systems.

Unfortunately, the subsystem structure is not obvious from the source code structure. Our research therefore proposes an automatic technique that creates a hierarchical view of the system organization based solely on the components and relationships that exist in the source code. The first step in our technique is to represent the system modules (classes) and the module(class)-level relationships as a module dependency graph. We then use our algorithms to partition the graph in a way that derives the high-level subsystem structure based on the component-level relationships that are extracted from the source code. Fully automatic modularization techniques are not only useful to programmers who lack familiarity with a system. These techniques can also be used by system architects who want to compare the documented modularization with the automatically derived one, and improve the design by learning from the differences between the modularizations.

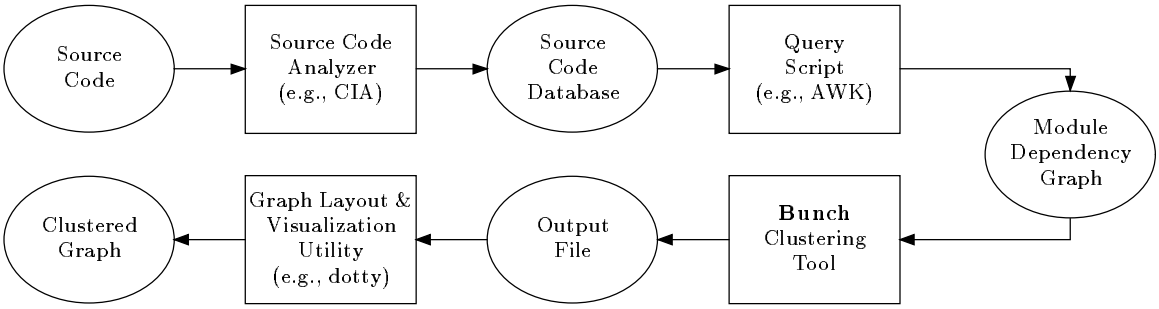


Figure 1: Automatic Software Modularization Environment

Figure 1 shows the architecture of our automatic software modularization environment. The first step in the modularization process is to extract the module-level dependencies from the source code and store the resultant information in a database. We used AT&T's CIA tool[1] (for C) and Acacia[2] (for C++) for this step, although there are commercial and public domain tools that could have been utilized to extract similar information. After all of the module-level dependencies have been stored in a database, we execute an AWK script to query the database, filter the query results, and produce, as output, a textual representation of the module dependency graph. Our clustering tool, called Bunch, applies our clustering algorithms to the module dependency graph, and emits a text-based description of the high-level structure of the systems organization. We use the AT&T dotty visualization tool[9] to read the output file from our clustering engine and produce a visualization of the results. In section 5 of this paper we elaborate on the operation and performance of Bunch.

The structure of the remainder of this paper is as follows: Section 2 presents a case study that illustrates the results of our automatic software modularization technique. Section 3 develops the pertinent aspects of our technique by formally quantifying inter-connectivity, intra-connectivity and modularization quality. Section 4 presents the algorithms we have implemented for clustering software components. Section 5 is dedicated to describing the operation and performance of Bunch,

our automatic software modularization tool. Section 6 presents related research in the area of software modularization; and describes how our approach compliments these techniques. Finally, we conclude this paper by outlining the research benefits and limitations of our work along with a discussion of our future plans to improve our technique.

2 An Example

Figure 2 shows the module dependency graph of a C++ program that implements a file system service. It allows users of a new file system nos to access files from an old file system oos (with different file node structures) mounted under the users' name space. Each edge in the graph represents at least one dependency relationship between program entities in the two corresponding source modules (C++ source files). For example, the edge between oosfid.c and nos.h is established due to 19 dependency relationships from the former to the latter. Among them is a reference relationship between the member function getStats of a class Fid defined in oosfid.c and a member mode of struct Dir defined in nos.h .

The program consists of 50,830 lines of C++ code, not counting the system library files. The Acacia tool parsed the program and detected 463 C++ program entities and 941 dependency relationships among them. Containment relationships between classes/structs and their members are excluded for consideration in the construction of the module dependency graph.

Even with the module dependency graph, it is not clear what major components are in this system. Applying our automatic modularization tool to the graph results in Figure 3 with two large clusters and two smaller ones in each. Several interesting observations can be made about these clusters:

1. It is obvious that there are two major components in this system. The right cluster mainly deals with the old file system while the left cluster deals with the new file system.

2. The clustering tool is effective in putting strongly-coupled modules like pwdgrp.c and pwdgrp.h in the same cluster even though the algorithm does not get any hints from the file names. The cluster is consistent with the designer's expectation.

3. On the other hand, just by looking at the module names, a designer might tend to associate oosfid.c with the right cluster. Interestingly, the algorithm decided to put it in the left cluster because of its associations with sysd.h and nos.h, which are mostly used by modules in the left cluster. The designer later confirmed that the partition makes sense because it is the main interface file used by the new file system to talk to the other file system. The file has to be rewritten (and renamed) for each interface to a different file system.

4. We cannot quite explain why a small cluster, consisting of errlst.c, erv.c, and nosfs.h, was created on the left. It might have been better to merge that small cluster with its neighbor cluster. A simple explanation is that our algorithm is only sub-optimal and may give a less-than-satisfactory answer in certain cases.

In the next two sections, we examine our algorithm in detail and shed some light on the heuristics used to drive our tool to obtain a sub-optimal solution.

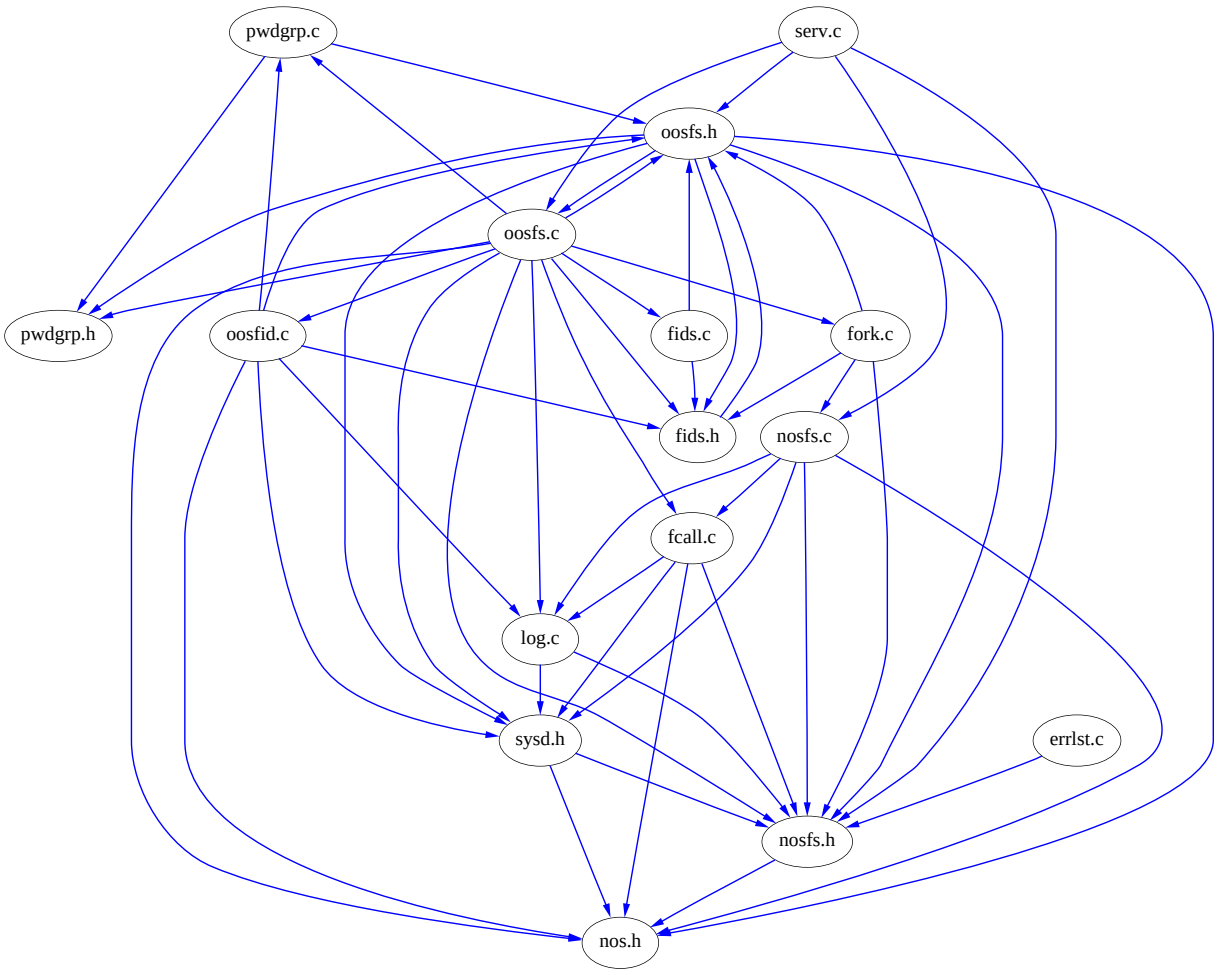


Figure 2: Module Dependency Graph of the File System

3 Automatic Software Modularization

Software systems contain a finite set of software components along with a finite set of relationships that govern how the software components interact with each other. Typical software components include classes, modules, variables, macros and structures; common relationships include import, export, inherit, procedure invocation, and variable access. The goal of our software modularization process is to automatically partition the components of a system into clusters (subsystems) so that the resultant organization concurrently minimizes inter-connectivity (i.e., connections between the components of two distinct clusters) while maximizing intra-connectivity (i.e., connections between the components of the same cluster). We accomplish this task by treating the clustering process as an optimization problem where our goal is to maximize an objective function based on a formal characterization of the tradeoff between inter and intra-connectivity.

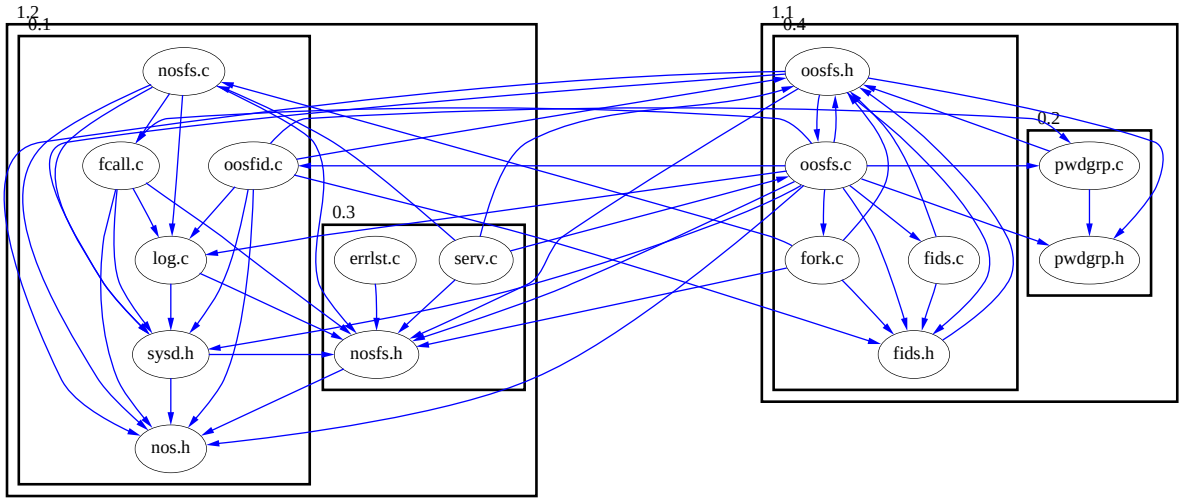


Figure 3: Automatically Produced High-Level System Organization of the Same File System

The clusters, once discovered, represent higher-level component abstractions of the system's organization. Each subsystem contains a collection of modules that either cooperate to perform some high-level function in the overall system (e.g., scanner, parser, code generator), or provide a set of related services that are used throughout the system (e.g., file manager, memory manager). A fundamental assumption underlying our approach is that well-designed software systems are organized into cohesive clusters that are loosely interconnected.

3.1 Intra-Connectivity

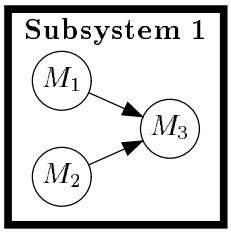
We define *Intra-Connectivity* (A) to be a measure of the connectivity between the components that are grouped together in the same cluster. A high degree of intra-connectivity indicates good subsystem partitioning because the modules grouped within a common subsystem share many software-level components. A low degree of intra-connectivity indicates poor subsystem partitioning because the modules assigned to a particular subsystem share few software-level components (limited cohesion). By maximizing the intra-connectivity measurement we increase the likelihood that changes made to a module are localized to the subsystem that contains the module. By localizing changes, we reduce the number of paths through which a modification to the software can result in errors being introduced and propagated into other subsystems.

Formally, the intra-connectivity measurement A_i of cluster i consisting of N_i components and μ_i intra-edge dependencies is:

$$A_i = \frac{\mu_i}{N_i^2}$$

This measurement is a percentage of the maximum number of intra-edge dependencies that can exist for cluster i, which is N_i^2. The value of A_i is bounded between the values of 0 and 1. A_i is 0 when the modules in a cluster do not share any software-level components; A_i is 1 when every

module within a cluster uses a software resource from all of the other modules in its cluster (i.e., the modules and dependencies within a subsystem form a complete graph). In Figure 4 we apply our intra-connectivity measurement to a cluster containing three modules and two intra-connections.



Number of modules in a subsystem, $N_i = 3$
Number of intra-edge dependencies, $\mu_i = 2$
Maximum intra-edge dependencies, $N_i^2 = 9$

$$A_i = \frac{\mu_1}{N_i^2} = \frac{2}{9} = 0.2222$$

Figure 4: Intra-Connectivity Example

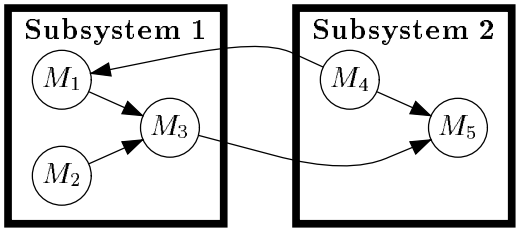
3.2 Inter-Connectivity

We define Inter-Connectivity (E) to be a measurement of the connectivity between two distinct clusters. A high degree of inter-connectivity is an indication of poor subsystem partitioning. Having a large number of inter-dependencies complicates software maintenance because changes to a module may affect many other parts of the system due to the subsystem interrelationships. A low degree of inter-connectivity is a desirable trait of a system organization and is an indicator that the individual clusters of the system are, to a large extent, independent. Therefore, changes applied to a module are likely to be localized to its subsystem, which reduces the likelihood of introducing errors into other parts of the system.

Formally, the inter-connectivity $E_{i,j}$ between clusters i and j (each consisting of N_i and N_j components, respectively) with ε_{ij} inter-edge dependencies is:

$$E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \frac{\varepsilon_{i,j}}{2 \times N_i \times N_j} & \text{if } i \neq j \end{cases}$$

As illustrated by the above expression for $E_{i,j}$, our inter-connectivity measurement is a fraction of the maximum number ($2 \times N_i \times N_j$) of inter-edge dependencies between clusters i and j. This measurement is bound between the values of 0 and 1. $E_{i,j}$ is 0 when there are no module-level dependencies between subsystem i and subsystem j; $E_{i,j}$ is 1 when each module in subsystem i depends on all of the modules in subsystem j and vice-versa.



Number of modules in subsystem 1, $N_1 = 3$
Number of modules in subsystem 2, $N_2 = 2$
Number of inter-edge dependencies, $\varepsilon_{1,2} = 2$

$$E_{i,j} = \frac{\varepsilon_{ij}}{2 \times N_i \times N_j} = \frac{2}{12} = 0.1666$$

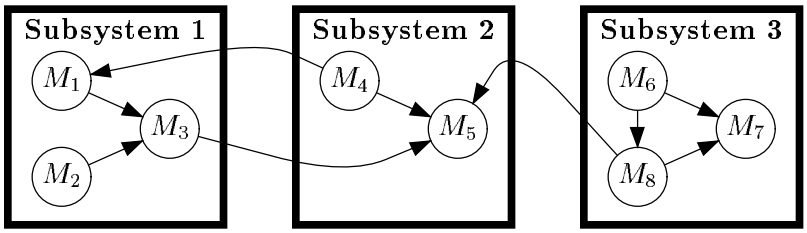
Figure 5: Inter-Connectivity Example

3.3 Modularization Quality

Recall that our goal is to discover a partitioning of the components of a software system that concurrently minimizes inter-connectivity and maximizes intra-connectivity. The Modularization Quality (MQ) measurement, which will be used as the objective function of our optimization process, is therefore defined as the measurement of the "quality" of a particular system modularization. Specifically, the MQ of a module dependency graph partitioned into k clusters, where A_i is the Intra-Connectivity of the i^{th} cluster and $E_{i,j}$ is the Inter-Connectivity between the i^{th} and j^{th} clusters, is:

$$MQ = \begin{cases} \frac{1}{k}\sum_{i=1}^{k} A_i - \frac{1}{\binom{k}{2}}\sum_{i,j=1}^{k} E_{i,j} & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases}$$

The MQ measurement demonstrates the tradeoff between inter-connectivity and intra-connectivity by rewarding the creation of highly cohesive clusters, while penalizing the creation of too many inter-edges. This tradeoff is established by subtracting the average intra-connectivity from the average inter-connectivity. We use the average values of A and E to ensure unit consistency in the subtraction because the Intra-Connectivity summation is based on the actual number of subsystems (k), while the Inter-Connectivity summation is based on the number of distinct pairs of subsystems, given by $\binom{k}{2}$. The MQ measurement is bounded between -1 (no cohesion within the subsystems) and 1 (no coupling between the subsystems). Figure 6 illustrates an example calculation of MQ.



$$MQ = \frac{\overbrace{0.22 + 0.25 + 0.33}^{\textbf{Average}A}}{3} - \frac{\overbrace{0.16 + 0.0 + 0.083}^{\textbf{Average}E}}{3} = 0.186$$

Figure 6: Modularization Quality Example

4 Automatic Software Modularization Algorithms

Now that the MQ measurement has been defined, we turn our attention to developing algorithms that start with the module dependency graph of the source code and produce, as output, a hierarchy of clusters that represents the subsystem structure of a software system. Figure 7 depicts the software modularization algorithms that are supported by Bunch. The optimal algorithm produces the best results, but it only works well for small systems. The other two algorithms are much

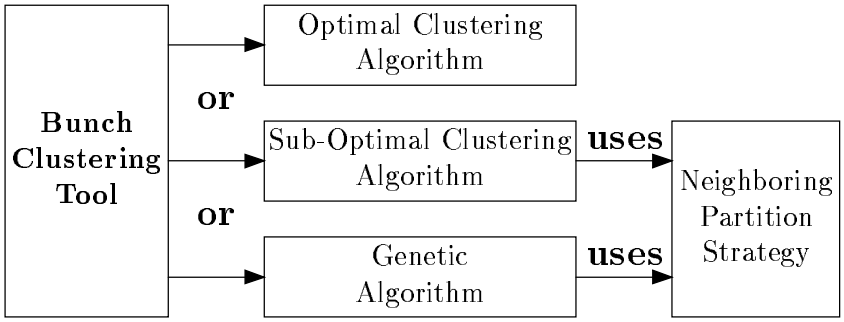


Figure 7: Automatic Clustering Algorithms Supported by Bunch

faster, but they may not produce an optimal result. The remainder of this section describes these algorithms in detail.

The first step in our automatic modularization process is to parse the source code and build a module dependency graph. Formally, a module dependency graph $MR = (M, R)$ consists of two components: (1) M is the set of named modules in the software system, and (2) $R \subseteq M \times M$ is a set of tuples of the form $\langle v, v \rangle$, which represents the source-level relationships that exist between module u and module v. Once the module dependency graph is constructed, we apply our modularization algorithms to MR. The remainder of this section develops some additional theory and then presents a collection of algorithms that we have implemented to automatically partition a software system.

4.1 Partitions of a Set

Consider the source code organization of a software system. Let S be a set of modules $M_1, M_2, \ldots, M_n$ where each module, M_i contains source code features (i.e., variables, macros, functions, procedures, constants). Let $\pi = A_1, A_2, \ldots, A_n$ be a set of non-empty subsets such that each $A_i \subseteq S$. π is a partition of set S if:

1. The subsets are a covering of S: $\bigcup_{i=1}^{n} A_i = S$
2. The subsets are mutually exclusive: $A_i \bigcap A_j = \emptyset, \forall i \neq j$

Given this definition, each subset A_i is a cluster of the partition. Also, a partition of S into k non-empty clusters is called a $k - partition$ of S.

Given a set S that contains n elements, the number $S_{n,k}$ of k-partitions of the set satisfies the recurrence equation:

$$S_{n,k} = \begin{cases} 1 & \text{if } k = 1 \text{ or } k = n \\ S_{n-1,k-1} + k \times S_{n-1,k} & \text{otherwise} \end{cases}$$

The entries $S_{n,k}$are called *Stirling numbers of the second kind.* Stirling numbers grow exponentially with respect to the size of S. Hence, a 5 node module dependency graph would have 52 distinct partitions, but a 15 node module dependency graph would have 1,382,958,545 distinct partitions.

4.2 The Optimal Clustering Algorithm

We now present our algorithm for determining the optimal clustering of a software system.

Algorithm

1. Let $S = M_1, M_2, \ldots, M_n$, where each M_i is a module in the software system.
2. Let MR be the graph representing the relationships between the modules in S.
3. Generate every partition of set S.
4. Evaluate MQ for each partition.
5. The partition with the largest MQ is the optimal solution.

We have successfully applied the Optimal Clustering Algorithm to systems of up to 15 modules. Beyond that, the search space (number of k-partitions of S) becomes so large that it cannot be explored in a reasonable timeframe. Clearly, sub-optimal techniques must be employed for systems with a large number of modules.

4.3 Neighboring Partitions

A critical step in our sub-optimal clustering technique is the ability to move modules between the clusters of the partition in an attempt to improve the MQ. This task is accomplished by generating a set of neighboring partitions (NP) for a partition.

We define a partition NP to be a neighbor of a partition P if and only if NP is exactly the same as P except that a single element of P is in a different cluster in partition NP. Figure 8 illustrates the process of determining all of the neighboring partitions of P.

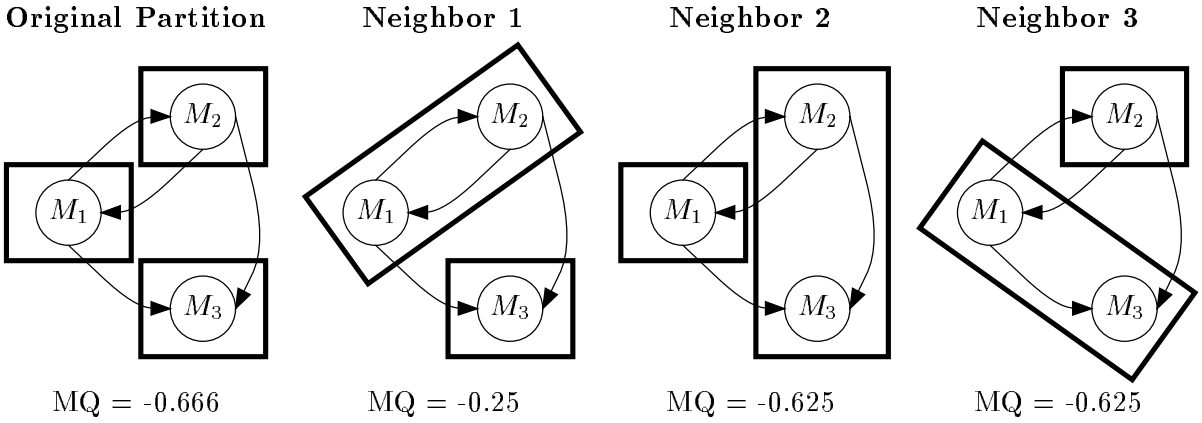


Figure 8: Neighboring Partition Example

Note that there are many other ways to define a neighboring partition. We chose this definition because it is simple to understand and implement while offering good execution performance.

In other automated software modularization techniques[10], a poor module movement decision early on can negatively bias the final results because there is no facility for moving a module once it has been placed into a cluster. A useful property of our neighboring partition approach is that the assignment of a module to a cluster is not permanent.

4.4 Sub-Optimal Clustering Algorithm

Because the search space required to enumerate all possible partitions of a software system becomes prohibitively large as the number of modules in the system increases, we direct our attention to developing a search strategy, based on hill-climbing [Mitc96] techniques, that quickly discovers an acceptable sub-optimal clustering result.

In summary, our sub-optimal clustering algorithm starts with a random partition and repeatedly finds better neighboring partitions until no neighboring partition can be found with a higher MQ.

Sub-Optimal Clustering Algorithm

1. Let $S = M_1, M_2, \ldots, M_n$, where each M_i is a module in the software system.
2. Let MR be the graph representing the relationships between the modules in S.
3. Generate an initial random[8] partition P of set S.
4. Repeat
   - Find a better neighboring partition NP (i.e., one such that $MQ(NP) > MQ(P)$).
   - If such an NP is found, let $P = NP$.

   until no further "improved" neighboring partitions can be found.
5. Partition P is the sub-optimal solution.

In our sub-optimal clustering algorithm, a better neighboring partition is discovered by going through the set of neighboring partitions of P, one-by-one, until a partition with a higher MQ is found.

4.5 A Genetic Algorithm Implementation

Our experimentation with the sub-optimal clustering algorithm has shown that, given an initial random starting partition, the algorithm will always converge to a local maximum. However, not all randomly generated initial partitions improve to an acceptable sub-optimal result. One approach to solving this problem is to run the experiment many times using different initial partitions and pick the experiment that results in the largest MQ as the sub-optimal solution. As the number of experiments increases, the probability of finding the globally optimal partition (based on the MQ) also increases.

Another more systematic approach to solving our optimization problem is based on Genetic Algorithms. Discovering an acceptable sub-optimal solution based on Genetic Algorithms involves starting with a population of randomly generated initial partitions and systematically improving

them until all of the initial samples converge. In this approach, the resultant partition with the largest MQ is used as the sub-optimal solution.

Genetic Algorithms[4] have been successfully applied to many problems that involve exploring large search spaces. They combine a survival-of-the-fittest technique with a structured and randomized information exchange to facilitate innovative search algorithms that parallel the natural biological selection process. Genetic algorithms are more than a randomized search; instead, they exploit historical data to speculate new information that is expected to yield improved results.

We now present our genetic search algorithm for finding a sub-optimal partition of a software system:

Genetic Search Sub-Optimal Clustering Algorithm

1. Let $S = M_1, M_2, \ldots, M_n$, where each M_i is a module in the software system.
2. Let MR be the graph representing the relationships between the modules in S.
3. Generate a random population of N random partitions of set S.
4. Repeat
   - Randomly select a percentage of partitions from the population and improve them by finding better neighboring partitions.
   - Generate a new population (from the existing one) by using a biased selection process that favors partitions with a larger MQ (the size of the population does not change but good partitions from the initial population might be duplicated while poor ones may be eliminated).

   until no improvement is seen for t generations, or until all of the partitions in the population have converged to their maximum MQ, or until the maximum number of generations ($MaxG$) has been reached.
5. Partition P in the final population with the largest MQ is the sub-optimal solution.

Our genetic clustering implementation has several user-configurable parameters. These include setting the population size (N), the convergence percentage (t), and the maximum number of generations to execute ($MaxG$) before concluding that the experiment did not converge.

4.6 Agglomerative Clustering

The algorithms presented in the previous section discovered partitions based on the graph (MR) that was formed by recovering the relationships that exist between the source code components. However, in most systems we are interested in finding a hierarchy of subsystems that capture the higher-order relationships that exist in the software. Therefore, agglomerative clustering has been integrated into this, and other software modularization techniques[6], because it provides a hierarchical view of the systems organization.

The first step in the agglomerative clustering process is to apply our standard software modularization algorithms to the MR graph. This activity discovers a partition, P_{lm}, which represents a

partition that has converged to a local maximum. We then build a new component-level graph by treating each cluster in P_{lm} as a single component. Furthermore, if there exists at least one edge between any two clusters in P_{lm} then there is an edge between their representative nodes in the new component-level graph. We then apply our clustering algorithms to the component-level graph in order to discover the next higher-level partition. This process is applied iteratively until all of the components have coalesced into a single cluster (i.e., the root of the subsystem decomposition hierarchy).

5 The Bunch Clustering Tool - Implementation and Performance

We have implemented the algorithms described above and applied them to many example software systems. Table 1 presents performance measurements for some common systems that were processed by Bunch. The results indicate that our algorithms are capable of efficiently discovering an acceptable sub-optimal partitioning of the software systems that we examined. The computation environment used for these experiments was a Pentium 166 computer with 80 Mb of RAM, running the WindowsNT 4.0 operating system. The execution times shown in Table 1 were collected running Bunch under the Microsoft J++ virtual Java machine. We experienced similar performance results using the Java just-in-time (JIT) compiler provided by Sun Microsystems in Solaris 2.7.

Interested readers may download a copy of our software, which was developed using the Java programming language, from the Drexel University Software Engineering Research Group (SERG) home page. The URL for SERG on the World Wide Web is http://www.mcs.drexel.edu/s̃erg.

System Name	System Type	Module Count	Module-Level Relationships	Execution Time
Ispell	Unix Spell Checker	22	98	22.793 sec.
Rcs	Version Control System	27	159	46.256 sec.
Mtunis	Small Operating System	20	57	17.876 sec.
Lu	Proprietary System	153	103	1 hour 24.924 sec.

Table 1: Bunch Tool Performance

6 Related Work

The problem of automatic modularization (also referred to as automatic clustering) has been extensively researched over the past two decades. Several semi-automatic techniques such as Rigi[5] and Arch[10] have been developed to address this problem. However, these techniques rely on the intervention from an architect who understands the system structure in order to produce good results. As a result, these techniques are of little help to someone who is not familiar with a software system, but is trying to understand its structure.

Hutchens and Basili[6] presented an automatic clustering technique based on data bindings. In their approach, a data binding is defined as an ordered triple $\langle p, x, q \rangle$, where p and q are procedures and x is a variable within the scope of p and q. Unfortunately, the use of data bindings as the basis for performing a software modularization has some shortcomings. Specifically, if the system modules

exhibit strong encapsulation (hide their data), or use abstract data types (e.g., Classes), then there will no way to determine their module-level relationships with data bindings because of the limited number of publicly accessible variables. Additionally, modularization with data bindings addresses the problem of clustering procedures and variables into classes and modules. Our desire is to cluster related modules and classes into subsystems, which is particularly useful when systems have a large number of modules. In addition to the bottom-up clustering approaches, which produce high-level structural views starting with the structure of the source code, research emphasis has been placed on using modularization information to improve the software maintenance process. The goal of the Software Reflexion Model[7] is to capture and exploit the differences that exist between the actual source code organization and the designer's mental model of the high-level system organization. The primary purpose of this technique is to streamline the amount of time it takes for someone unfamiliar with the system to understand the structure of its source code. The Orphan Adoption Problem[11] addresses the problem of incremental structural maintenance. The authors present an algorithm that accepts as input the name of a software resource (an orphan) and produces as output the name of the subsystem that has been chosen as the best parent (cluster) for the orphan.

7 Conclusions and Future Work

Experimentation with our clustering technique has shown good results for many of the systems that we have investigated. The primary method that we use to evaluate our results is to present an automatically generated modularization of a software system to the actual system designer(s) and ask for feedback on the quality of the results.

While we were able to produce good results for many of the systems that we examined, one known shortcoming with our current definition of modularization quality (MQ) is that it does not take into account the Interconnection Strength (IS)[5] of the relationships that exist between the modules in the software system. According to Mller et. al., IS is a measurement of the exact number of syntactic objects that are exchanged or shared between two modules. Thus, our clustering technique, which is based strictly on the topology of the module dependency graph, might not convey an accurate representation of a systems modularization when the magnitude of the interconnection strengths of the actual module relations differ significantly.

In order to address this shortcoming, we are currently working on an extension to our definitions of inter-connectivity, intra-connectivity and modularization quality that accounts for the weight of the module-level dependencies. We expect this extension to yield better results for systems in which the distribution of interconnection strength values is non-uniform. Our current assumption is that the IS strength of the module-level dependencies is equal to one.

References

[1] Y.-F. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

[2] Y.-F. Chen, E. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Cod e Detection. In *Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 1997.

[3] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, pages 80–86, June 1976.

[4] D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison Wesley, 1989.

[5] a. T. H. Mller, andM. Orgun and J. Uhl. Discovering and reconstructing subsystem structures through reverse engineering. Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, Aug. 1992.

[6] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, pages 749–757, Aug. 1995.

[7] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proc. ACM SIGSOFT Symp. Foundations of Software Engineering*, 1995.

[8] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms.* Academic Press, 2nd edition, 1978.

[9] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, pages 235–245, 1994.

[10] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

[11] V. Tzerpos and R. Holt. The orphan adoption problem in architecture maintenance. In *Working Conf. on Reverse Engineering (WCRE97)*, 1997.