Computing Program Modularizations Using the k-Cut Method

Christopher Jermaine
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
jermaine@cc.gatech.edu

Abstract

The problem of producing modularizations of huge, legacy software systems which lack an obvious modular structure is an important one. Such modularizations allow maintenance and reuse of smaller, more manageable pieces of program source code, and also may provide insights into the overall structure of the software system. In this paper, we consider the application of an algorithm from graph theory known as the k-cut method *to the domain of computing program modularizations. We describe the k-cut method, its associated algorithms, and the problem's application to reverse engineering of legacy systems. We also describe our experience in applying the method.*

Keywords: module, reverse engineering, k-cut

1. Introduction

Since large software systems that were originally developed years or even decades ago are still in widespread use today, the capability to derive an intimate understanding of the inner workings of a huge program by examining the program source code is vital. The process of developing such an understanding is known as *reverse engineering*. However, reverse engineering in not a trivial problem. Developing an understanding of a huge, legacy system is often exceedingly difficult, due at least in part to the following reasons:

- Because of the passage of time, the original developers of the program may no longer be present to aid in understanding the program. For this reason, the development of powerful methods applicable in the absence of *a priori*, human input is important.
- Adequate documentation of the program's inner workings and overall structure may not exist, especially if the program has grown steadily throughout its life due to frequent, non-systematic, ad-hoc additions of new functionality to the system.
- Often, such legacy systems were developed without the aid of a programming language supporting structured program development. For example, language facilities such as modules, packages, classes, etc. that may help provide structure to a program may be entirely absent from the programming language. This may cause the program to be difficult to understand later. The underlying hierarchical layout of source code files may be the only clue as to program organization. However, with a huge, complicated directory tree, file organization alone may be wholly inadequate to determine structure.

It is important to note that even in the face of these substantial difficulties, reverse engineering is still an important problem. The resources that have gone in to program development and maintenance over a long life-cycle may be huge. If substantial program functionality must be added or changed in a long-lived, brittle program, the cost of re-developing the entire program may be prohibitive, making maintenance of the existing system the only option. If this is the case, the ability to extract an understanding of the overall program structure, even in the face of the aforementioned difficulties, is crucial.

In this paper, we describe a tool that can aid in the reverse engineering process. Our approach is a simple, divide-and-conquer method: given a huge program comprised of multiple subprograms (or functions), we have developed a method that breaks that program into *modules*, or groups of related func-

0-7695-0303-9/99 $10.00 © 1999 IEEE

tions. This provides the software engineer with an additional layer of abstraction: first, only the inner workings of each module need to be understood; then a study of the interactions among modules can support an understanding of the overall program structure.

Our method is based on the observation that a module can be defined as a group of functions having *high cohesion* and *low coupling*. That is, interactions among functions *within* a module are frequent, but interactions *among* functions in different modules are rare.

This begs the question: what are *interactions*? A possible approximation for the concept of an *interaction* between functions is that of locating *function invocations*. Thus, when one function calls another (as determined through static program analysis), we say that there is an *interaction* among the functions. The network of function invocations that make up a large program can be represented using a call graph, where nodes or vertices in the graph represent functions, and edges represent invocations. The problem of computing a modularization of a large program then becomes the following: given a program call graph, can we break the graph into subgraphs (modules) such that invocations among modules (coupling) is *minimized*, and invocations within modules (cohesion) is *maximized*?

Unfortunately, in the general case, the answer is no. This is due to the inherent intractability of the problem [3]. Widely known as the *k-cut problem* in graph theory, it is probably not possible to solve efficiently. However, somewhat efficient approximations do exist [11], and approximation in order to produce a good program modularization is the method we will consider here.

The paper is organized as follows. In Section 2, we discuss the k-cut problem in detail, focusing on the special case of determining a modularization of a program call graph. In particular, approximation efficiency, even with an input graph containing thousands of functions, is of primary concern. In Section 3, we discuss a metric that may be used to help choose an appropriate degree of modularization. In Section 4, we discuss our implementation and experience using the method on actual program call graphs. Section 5 presents some related work; we conclude the paper in Section 6.

2. Minimal k-Cuts

In this section, we give a formal definition of the *minimal k*-cut of a graph. We then show in more detail why a k-cut of a program call-graph may provide a useful modularization for that program, and discuss the complexity issues involved in computing a minimal k-cut of a call graph.

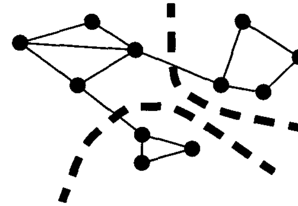

Figure 1: A graph and the associated minimal 2-cut.

2.1 k-Cuts and Program Modularization

Informally, a minimal k-cut is a set of k - 1 "cuts" across edges in a graph, breaking the graph into disjoint subgraphs, such that the number of edges "cut" by the partitioning is minimized. For a simple example, of a k-cut of a simple, undirected graph with k equal to two, refer to Figure 1.

More formally, given a graph $G = <V, E>$, where V is a set of vertices and E a set of edges of the form (v_1, v_2), the *minimal k-cut of G* is defined as follows:

Definition 1. The minimal k-cut of a graph G is the partition of E into k sets $A_1, A_2, \ldots, A_k$, such that *size* $(\{(v_1, v_2) \in E \mid \exists i, j,\ i \neq j \wedge v_1 \in A_i, v_2 \in A_j\})$ is *minimized.*

Why might the computation of a minimal k-cut for a program call-graph provide a good candidate modularization for that program? The intuition behind the method is as follows. It can be expected that in a program, functions that most naturally belong to the same module will frequently call many of the same functions. Those functions, which, in turn, are called by common routines are likely to be part of the same module as well. In fact, the strongest indication that two functions belong to the same module is the observation that one of the routines calls the other. Thus, we can expect that *within* a module, function invocations are common; however, *between* modules, function invocations tend to be rare unless the routine called provides an interface to its module. Thus, there is likely to be a good deal of cohesion within program modules. Because of these observations, the computation of a minimal k-cut of a call graph may suggest a good modularization for a program since it chooses the modularization such that function calls across module boundaries are minimized.

For a simple example of why a k-cut might be useful, we consider the call graph of Figure 2 In this

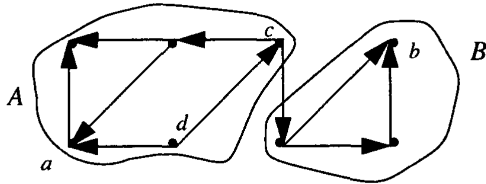

Figure 2: A program call-graph and the corresponding minimal 2-cut.

case, a minimal k-cut with a k value of 2 is computed to produce the modularization shown, breaking the program into modules A and B. For this simple graph, the method chooses what seems to be the most suitable modularization of the program.

It should be noted that while the graph of Figure 2 happens to demonstrate how the k-cut method can be useful, it also depicts a particularly difficult case in many ways for other modularization techniques. More conventional modularization methods relying on the notion of dominance or a clear hierarchical tree structure (for example [2]), are likely to find difficulty with this particular program for several reasons. First, while the graph is acyclic, it is not a tree, and the hierarchical structure is not totally clear. And second, locating the dominant node in the graph, d, is of little help in computing the modularization. This is because under the ideal program modularization, both of its children are located in the same module.

2.2 Computing a Minimal k-Cut

While the computation of a k-cut may suggest a good modularization of a call-graph, its computational complexity presents a challenge. In general, computing the k-cut of a graph is an NP-hard problem [3]. Given that the size of a large call graph may be on the order of thousands or tens of thousands of functions, an exponential-time computation of the minimal k-cut is not practical. This renders an exact solution infeasible in most cases.

It is, however, possible to *approximate* the k-cut of a graph and produce a set of cuts that are within a factor of two times the optimal cost [11], where the cost of the k-cut is defined to the number of edges removed by the cuts. The algorithm to compute the approximate-optimal k-cut is simple, and is outlined as follows:

Algorithm Approximate_*k*_Cut

1) Compute a set of *Gomory-Hu cuts* [8] for the graph.
2) Sort the Gomory-Hu cuts in order of increasing weight.
3) Cut the graph using the k *least* costly of the Gomory-Hu cuts.

We now describe each of these steps in detail.

2.2.1 Computing Gomory-Hu Cuts

In this section, we will define what it means to compute a set of *Gomory-Hu cuts* for a call graph, and then discuss how thus may be done. First, we define a *Gomory-Hu tree* for an undirected graph $G = <V, E>$. Informally, a Gomory-Hu tree for a graph G is a spanning tree for that graph having integer-weighted edges such that for any pair of vertices v_1, v_2 in G, the minimum weight on the unique path between v_1 and v_2 in the tree is equal to the maximum flow between v_1 and v_2 in the original graph G. We will give an example of such a tree in Figure 3 below. More formally:

Definition 2. A *Gomory-Hu tree* for a graph G is a tree $G_{G\text{-}H}$ with vertices $V_{G\text{-}H} = V$ and a set of weighted edges $E_{G\text{-}H}$ of the form (v_1, v_2, w) such that $\forall v_1, v_2 \in V$, the *maximum flow* between vertices v_1, v_2, in G is $min\ \{w \mid (v_3, v_4, w) \in E_{G\text{-}H} \wedge (v_3, v_4, w)$ is on the unique path from v_1 to v_2 in $G_{G\text{-}H}\}$.

The above definition relies, in turn, on the definition of the *maximum flow* for a graph G:

Definition 3. The *maximum flow* for an undirected graph G between vertices v_1 and v_2 is the maximum number of *mutually exclusive* paths (paths having no common edge) between v_1 and v_2 in G.

Note that the above definitions relate only to *undirected* graphs, whereas the call graph for a program is a directed graph. This presents a problem for applying these methods and definitions directly to a call graph. To deal with this, we simply take the easiest way out: we transform a directed call graph into an undirected graph by removing all notion of direction from function invocation. We argue that, as far as the k-cut modularization method is concerned, this can be done without any important loss of information from the call graph. Since we are primarily interested in maximizing cohesion within the functions of a call graph, we are not really concerned in the direction of a call; instead, we are more interested in the fact that the invocation in either direction exists.

To give an example of a Gomory-Hu tree for a call graph, we again consider the call graph of Figure 2. The corresponding Gomory-Hu tree for that graph is given in Figure 3. To demonstrate that this is, in fact, a Gomory-Hu tree, we select two nodes from

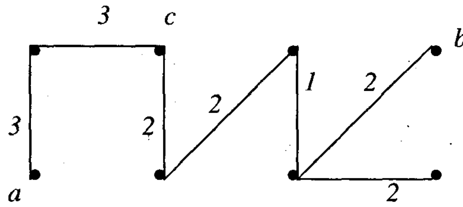

Figure 3: A Gomory-Hu tree for the call-graph of Figure 2.

the graph at random, say, a and b. The smallest weight on the unique path from a to b is one. Referring back to Figure 2, we find that the maximum number of paths containing no common edges from a to b is indeed one (once we have eliminated edge direction from the graph). Likewise, the maximum number of such paths between nodes a and c is 3, as predicted from the Gomory-Hu tree.

All of this begs the question: how might one compute a set of Gomory-Hu cuts for a call graph? To answer that, we give a simplified version of the algorithm (given in more detail in [8]):

Algorithm Gomory_Hu

1) Pick two vertices, v_1 and v_2, at random from the graph.
2) Compute the minimum 2-cut of the graph subject to the constraint that v_1 and v_2 are on *separate sides of the cut.*
3) Remove the cut edges from the graph, and recursively compute Gomory Hu trees for each of the two, newly disconnected subgraphs.

It follows from this algorithm that the number of minimal 2-cuts that must be computed as the entire Gomory-Hu tree is built scales linearly with the size of the call graph, since the process is repeated until the entire graph has been disconnected by these repeated 2-cuts. Due to the linear relationship between call graph size and the number of minimal 2-cuts that must be computed, efficiency of the minimal 2-cut computation is of primary concern to the method, an issue we turn to now.

2.2.2 Difficulty of the Minimal 2-Cut Problem

Given the great number of times it must be computed, ideally, we would like something close to a linear-time algorithm to compute the minimal 2-cut of a graph, given the constraint that the two vertices v_1 and v_2 must be on opposite sides of the cut. We note that this special case of the minimal k-cut problem is also known as the *max-flow/min-cut* problem [5] and is one of the most widely studied of all of the network flow problems. Fortunately, when the value of k is restricted to two, the problem is no longer intractable.

However, it is still not an easy problem to solve quickly. In the general case where each edge is allowed to have a real-number-valued weight, the fastest max-flow/min-cut algorithm is a special case of a more general set of algorithms known as *preflow-push* algorithms [6]. The fastest preflow-push algorithm available to solve this problem runs in $O(VElg(V^2/E))$ time [7]. Considering that in order to compute the Gomory-Hu tree for a call graph, such an algorithm must be run $O(V)$ times, the overall approximation time would then be $O(V^2Elg(V^2/E))$. Given that the number of vertices V for a call graph could number in the tens of thousands, this is not a practical algorithm for our purposes.

However, this is the fastest method for a *general* graph. In the specific case of a program call-graph, we can assume that the value of the maximum flow between any two nodes is quite small (we will use the common notation f^* to denote the maximum flow between two nodes). This is due to the fact that functions are generally limited in the number of other functions that they call or all called by them. Given two randomly selected functions, the number of distinct, mutually exclusive call sequences (meaning that each pair of call sequences share no common invocations) by which one routine could cause the other to be invoked is likely to be quite small (say, less than ten). Thus, in the corresponding graph, f^* will be small as well. Because of this, we can use a version of a class of max-flow/min-cut algorithms that are collectively known as the *Ford-Fulkerson method* [5]. Algorithms in this class generally run in $O(Ef^*)$ time, yielding an overall $O(EVf^*)$ running time to approximate the k-cut for a call-graph. Since the number of function invocations present in an average function can be expected to be relatively insensitive to overall program size (at least once a program achieves a certain size) f^* can be expected to be insensitive to the size of a call graph. In practice then, the Ford-Fulkerson method provides what is essentially a quadratic algorithm with respect to call graph size.

2.2.3 Fast Computation of a Minimal 2-Cut

In this section, we overview an implementation of the Ford-Fulkerson algorithm for computing a max-flow/min-cut for the special case of an undirected program call graph. The basic algorithm is as follows:

Algorithm Ford-Fulkerson

1) Do until there are no more paths from v_1 to v_2.
   2) Pick any path from v_1 to v_2.
   3) Walk along this path. For each edge traversed during the walk:
      4) If the edge is *undirected*, change to a directed edge *in the opposite direction of travel;*
      5) Otherwise, if the edge is *directed*, chance the edge to an undirected edge.
6) Output all vertices (nodes) still reachable from v_1 as being on one side of the minimum cut; all other vertices are on the other side.

Before we give an example of the operation of the algorithm on the call graph of Figure 1, we discuss a few of the more non-obvious aspects of the algorithm. The algorithm is based on the dichotomy of the *maximum flow* and the *minimum cut* of a graph. By repeatedly selecting arbitrary *flows* (paths) from our source node (v_1) to the destination node (v_2), we eventually reach a point where no more paths are possible. At this point, the graph is saturated, and there exists some cut across the graph where no more edges exist that are not part of some path from source to destination. This cut marks the *narrowest* point of the graph (where it is the fewest edges in width), and corresponds to the minimal 2-cut separating v_1 and v_2.

Before we demonstrate the method, one other point bears further discussion. That is the slight complexity associated in altering edge directions in steps 4) and 5) of the Ford-Fulkerson algorithm. This complexity is required due the idea of a *residual flow.* Sometimes, as a path (flow) is chosen, it actually comprises two different flows comprising part of the maximum flow from source to destination. To demonstrate this, we consider Figure 4.

Figure 4(a) shows one path from source to destination. If this path had been chosen during the first iteration in step 2), it is clear that there are no subsequent paths that do not share edges with this path. However, this path does *not* constitute a max-flow/min-cut for the graph (the maximum number of paths, and hence the maximum flow from source to destination is two, and traces along the outside of the diamond). To solve this problem, we allow subsequent paths to reuse edges, *but only in the opposite direction that they had previously been traversed.* This effectively allows us to *undo* a portion of a previous path, and is the notion of a *residual flow.* Thus, as we attempt to choose our next path, we are faced with a graph that appears as in Figure 4(b). There *is*

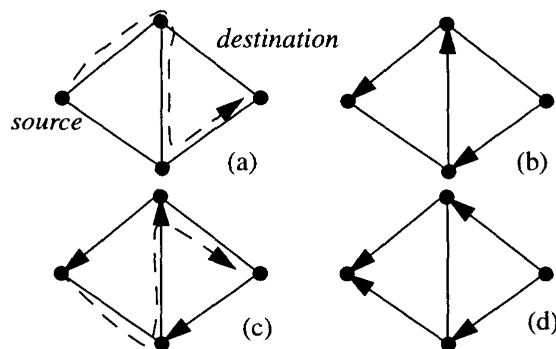

Figure 4: Demonstrating the need for directional edges in the Ford-Fulkerson algorithm.

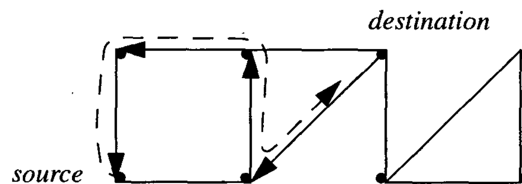

Figure 5: First Step of the Ford-Fulkerson computation.

one more path, as is shown in Figure 4(c); at this point, the graph is saturated, as is shown in Figure 4(d).

We now demonstrate the method on the call graph of Figure 2. First, we transform the directed call-graph into an undirected graph. Then, given the source and a destination nodes selected randomly by the Gomory-Hu algorithm, a search is performed through the graph to find some path from source to destination. Along that path, the non-directional edges are replaced with directed edges going in opposite direction of the path. This is shown in Figure 5.

Next, we repeat the process, subject to the constraint that we can only follow directed edges along their indicated direction. Also, rather than replacing directed edges in the opposite direction, we replace them with non-directional edges. At this point, during our second search, there is only one path from source to destination, and we take it. We show the state of the computation in Figure 6.

After our second graph traversal, there are no more paths from source to destination, and we then end the max-flow/min-cut computation. The number of path traversals is equivalent to f^*, the maximum flow from source to destination. To find the minimum cut, we then determine all of the nodes that are reach-

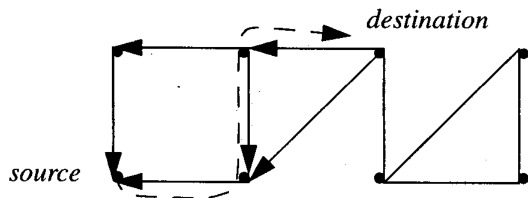

Figure 6: After the second iteration of the Ford-Fulkerson method.

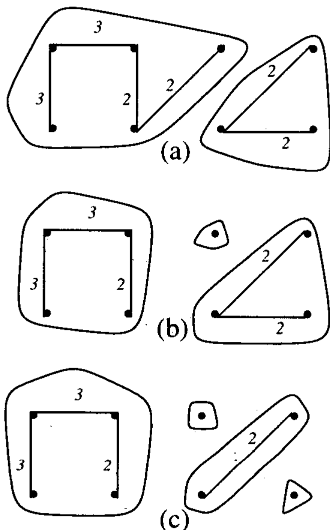

Figure 7: approximate-minimal 2, 3, and 4 cuts of a call graph.

able from the source (in this case, only the source node is reachable from itself). This is the corresponding minimum cut of the graph.

2.2.4 Using a Gomory-Hu Tree to Approximate a *k*-Cut

Once the Ford-Fulkerson algorithm has been used repeatedly by the Gomory-Hu algorithm to produce a Gomory-Hu tree for the call graph, the final step is to use the tree to approximate a k-Cut. This is a very simple matter and consists of simply removing, in sequence, the edges with the lightest weight from the graph. This is demonstrated in Figure 7.

Figures 7(a), (b), and (c) show the approximate k-cuts for k values of 2, 3, and 4, respectively. As we will discuss in Section 3, this demonstrates a particularly attractive quality of the method we describe here. Once the Gomory-Hu tree for a graph has been constructed, it is a trivial matter to compute k-cuts for any value of k. This is useful because an appropriate value of k is typically not known *a priori*. Since little additional computation is required once a k-cut has been computed for a given k, cuts for different values of k can be tried, and a measure of the quality of the associated modularization can be applied to determine the preferred value for k.

3. Determining an Appropriate Value for *k*

As stated above, there is no obvious way suggested directly by these algorithms to determine an appropriate value of k in order to choose from among the possible modularizations for a program. In particular, this is a serious problem if the size of the call graph, and hence the number of candidate modularizations, is quite large, and so each modularization cannot be feasibly be considered in sequence by a human expert.

To state the problem in a different manner, the classical k-cut problem requires that a value of k be supplied *a priori*. In fact, the set of all nodes in a call graph, with all edges removed, is always a minimal k-cut for that graph (with k set to one less than the size of the graph). However, it is likely not an appropriate modularization for the program. The difficulty is that when the method is applied to the problem of program modularization, it cannot be assumed that the optimal number of modules can be guessed at beforehand.

To help solve this problem and determine an appropriate value for k, we developed a metric to use in measuring the quality of a modularization having a number of modules $k + 1$. Intuitively, this metric measures the difference between the *expected* and *actual* numbers of inter- and intra-module function invocations in a given modularization. Modularizations with a relatively large number of inter-module invocations (high coupling) and a small number of intra-module invocations (low cohesion) will have a correspondingly high associated value for this metric.

Our metric, which we call *sociability*, hinges on the idea of the expected number of calls within and among modules. The essential intuition is that modularizations where the overall number of calls within and among modules are as one would naively expect from simply randomly selecting a modularization are in fact rather uninteresting. However, modularizations where there are a startlingly large number of internal calls within modules, and also an unexpectedly small number of external calls among modules, are more appropriate. To describe these notions more

formally, we now define expectation for inter- and intra-module function invocations.

3.1 Expected Extent of Calls Within a Module

Assuming no knowledge about the structure of a call graph other than the total number of nodes in each module and the number of invocations each module is involved in (that is, the number of calls made either from or to that module) and finally the total number of function calls in the entire program, we might naively expect that the number of *internal calls*, or calls existing entirely *within* a module, are proportional to the size of the module when compared to the size of the entire program. That is, as the size of the module grows with respect to the size of the entire program, a correspondingly higher proportion of the calls that module is involved in should be internal calls. More exactly, let:

nodes be the total number of nodes (functions) in the graph;

$nodes_i$ be the number of nodes in a given module i; and let

$calls_i$ be the number of function invocations that are either made by a function in module i, or are made of a function in module i, or both. That is, this is any edge in the call graph originating at or going to a function in module i.

Then:

Definition 3. Let $expi_i$ be the expected number of invocations of functions located in module i that are also initiated from module n. Then $expi_i$ is:

$$\frac{nodes_i}{nodes} \times calls_i$$

3.2 Expected Number of External Calls

Likewise, assuming no specific knowledge about the structure of the call graph, we can also compute the expected number of external calls. Let:

$expe_i$ be the expected number of function invocations that are either made by a function in module i, or are made of a function in module i, but *not* both.

Then we can define the expected number of external calls as follows:

Definition 4. The expected number of external calls $expe_i$ is:

$$\frac{nodes - nodes_i}{nodes} \times calls_i$$

3.3 The Sociability of a Modularization

Given Definitions 3 and 4, computing the sociability of a modularization is straightforward. For each module, we consider the difference between the expected number of internal function calls and the actual number and the difference between the expected number of external function calls and the actual number, as described in Definition 5:

Definition 5. Let int_i be the number of internal function calls within module i and let e_i be the number of external function calls made of functions within module i or made by functions within module i, but not both. Then the *sociability* of a modularization is:

$$\sum_{i=1}^{k+1} (int_i - expi_i) + (expi_i - e_i)$$

There are several considerations regarding Definition 5 that benefit from explanation. First, we note the lack of normalization in the formula that would allow comparisons of modularization across graphs. This is because this metric is introduced here primarily as a means to compare the suitability of different modularizations of the same program, and is not meant to be used to compare modularizations of separate programs. Second, we note that the value for this summation can indeed be negative (we will give such an example in Section 4.2) if the modularization in question is especially poor.

A final consideration is the suitability of the algorithms described in Section 2 for use with this metric. Since the bulk of the computational complexity involved in approximating the minimal k-cut is incurred during the computation of the Gomory-Hu tree for the graph, the addition of the use of the metric to determine the best value for k does little to add additional running time to the algorithm. The same Gomory-Hu tree can be used to compute a k-cut for each value of k, in succession, with each additional k-cut computed in linear time with respect to graph size once the first k-cut has been computed.

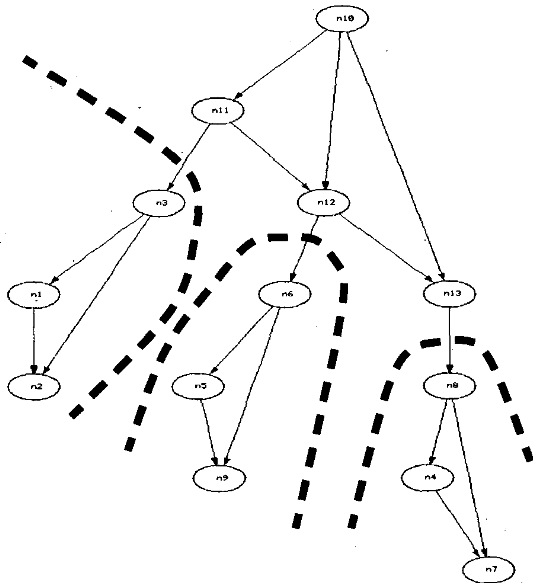

Figure 8: A call graph with an obvious optimal value for k (k = 4).

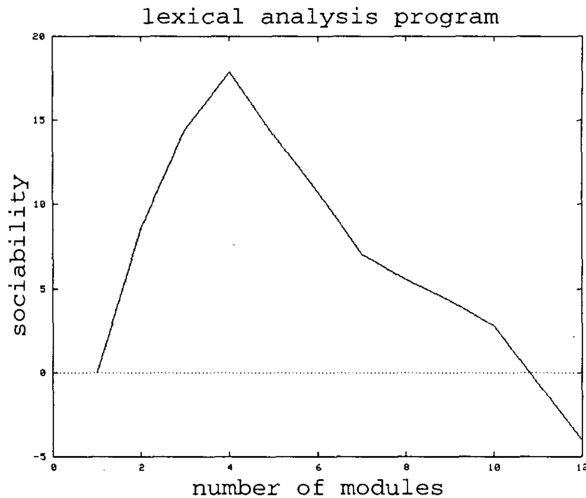

Figure 9: Plot of the sociability for each of the 13 modularizations of the lexical analysis program.

4. Evaluating the *k*-Cut Method in Practice

Having now described the method, in this section we discuss our experience with using the *k*-cut modularization method (that is, the approximate *k*-cut algorithms combined with the sociability metric) on real program call graphs. Using these real graphs as examples, we show that the method is able to produce appropriate modularizations for program call graphs and also demonstrate the scalability of the method for use on very large graphs. We also discuss some fundamental limitations of the method that become clear when it is applied to real programs that likely limit the method's utility in practice. We discuss how these observations provide a springboard for future, related work in applying graph algorithms to the problem of program modularization.

4.1 Experimental Testbed

To test the feasibility of the use of the k-cut method in practice, we have tested the method using many different program call graphs. We discuss three particular examples now:

- The first program used as input to the method was a small C program written to perform lexical analysis. The program consists of approximately 250 lines of code and is made of 13 separate functions.
- The next test was concerned primarily with scalability. The test involved computing a k-cut for the source code of the NCSA Mosaic World Wide Web browser. This particular software system is composed of 112,000 lines of C source code and 1644 functions.
- The final test involved using the method on two intermediate-sized programs that are small enough to allow an interpretation of the quality of the results obtained. These programs were a FORTRAN program called BTTSIM written to simulate projectile flight (32 subroutines) and a subset of the NCSA Mosaic source tree (55 functions).

4.2 Lexical Analysis Program

As indicated, we first tested the method on the small, lexical analysis program. For each value of *k*, a *k*-cut is constructed and for each *k*-cut, the associated sociability value for the associated modularization is computed. The modularization chosen for the lexical analysis program by the method is depicted in Figure 8. A plot of the sociability of the various modularizations suggested by the *k*-cut method is shown in Figure 9. The sociability of the modularization containing four modules is greatest, and from examination of Figure 8 it is clear that this modularization is most appropriate. While in this case the appropriate choice of *k* is very clear from Figure 9, in practice, we have found that the plots associated with some programs may have several peaks and require that a human expert choose from among a set of

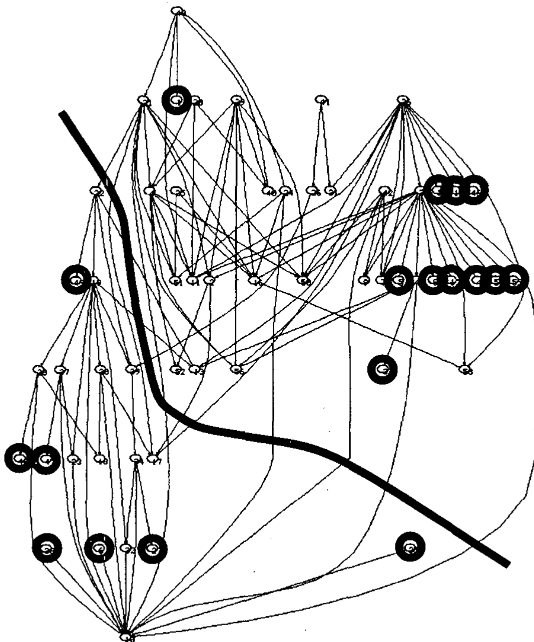

Figure 10: A modularization of the Mosaic subset (modules are indicated by dark circles) and a more appropriate cut, indicated by a dark line.

modularizations. In general, such human interpretation of the sociability plot is useful when determining an appropriate number of modules.

4.3 NCSA Mosaic

The next test was primarily concerned with evaluating the scalability of the method, and its applicability to rather large program call graphs. Scalability is a primary concern, since even though approximation algorithms for the k-cut method are used to compute only an approximate-optimal modularization, engineering a set of algorithms that are fast enough for use with very large programs is a primary concern of this work. To test the method's scalability, we ran our implementation on the call graph for NCSA Mosaic, which contains 1644 functions. Because of its large size, Mosaic is likely to be within an order or magnitude or two of the size of most large, real-life, legacy systems.

The results we obtained were encouraging. The method, running on a 200MHz Pentium Pro processor, was able to produce a modularization of Mosaic in slightly more than two minutes. Since running time is expected to scale approximately quadratically with respect to program call graph size, we expect that a modularization of a program ten times the size would take on the order of three hours of CPU time.

In terms of main memory requirements, our implementation requires that the call graph be stored in a two dimensional matrix made of $3n^2$ bits. It would be possible to implement the method so that memory requirements with respect to graph size are not quadratic. However, even with this quadratic memory requirement a 20,000 function call graph would still require only about 300Mb of main memory, which is probably not an unreasonable requirement.

4.4 Qualitative Results

In order to demonstrate the quality of the modularizations produced for larger program call graphs, we ran the method on the BTTSIM source code and on a subset of the Mosaic source tree. The method performed rather well on the BTTSIM source code, producing appropriate modularizations. However, some problems with the method became evident through examination of the modularizations produced by the method for the Mosaic subset. We show the modularization produced in Figure 10. The method chose to place each of the 18 functions that were called by or call only one other function into its own module, and placed all of the remaining 37 functions into a single, monolithic module.

The reason for this anomaly is that a minimal k-cut chooses to make cuts so as to minimize the number of edges removed, or cut. If there are many singular functions that are loosely attached to the graph, as well as a dense, highly connected core in the graph, the loosely connected routines will be detached and put into their own, tiny modules. Because the slicing off of single routines in this way often minimizes the cost of the k-cut, obvious cuts such as the one indicated in Figure 10 that slice through unexpected, weak spots in the graph are not chosen.

We have considered several fixes for this problem, including the combination of a dominance-based method with the k-cut algorithm. For example, nodes clearly dominated in a hierarchical graph could be considered to be inseparable from their parents (since they would likely best be placed into the same modules) and the associated edges not considered during the cutting process. However, this particular fix does not address the fundamental problem, and would not alleviate the situation where a node is called by *two* different routines, but where a more optimal cut across more than two edges should ideally be made. For this reason, we believe that ulti-

mate solution is to integrate the notion of sociability more tightly into the algorithms of Section 2. Desirable cuts like the one shown in Figure 10 would then be chosen not because they minimize the weight of the cut, but because they maximize the sociability of the resulting modularization.

5. Related Work

The problem discussed in this paper has been that of automatic and unassisted program architecture extraction. A related area of work is the development of more general tools for program architecture extraction. Such tools typically contain a parser for extraction of fundamental information about the program, some sort of storage manager for this information, and a set of additional tools that allow analysis of the extracted data. The storage manager might be a relational database system, for example, and the analysis tools may include support for graphical visualization and ad hoc querying of the information obtained about the program. Two such tools that have been built are Rigi [10] and Dali [9]. A comparison of these tools and several others is given in [1]. In Rigi, for example, a hierarchy is constructed with each level corresponding to a given level of abstraction for the software system. Combining of lower-level systems into higher-level systems in the hierarchy is based on the notions of cohesion and coupling, which are similar to the k-cut method. The tool allows and aides the user in manipulations of the hierarchy that eventually arrive at an architecture for the system.

However, these interactive tools are much more general purpose than the methods described in this paper. The k-cut method is narrower in scope and is a tool that may be of use within these larger tools. For example, a potential use of the method would be to suggest an initial organization that could be presented to the user as a starting point during the process of software architecture extraction.

There are alternatives to the method we have presented here that can be used as a basis for the more specific problem of automatically suggesting modularizations for large, legacy systems. We briefly focus on two: those methods based on *data clustering* and those methods based on *dominance analysis*.

Data clustering refers to a set of algorithms for partitioning a large set of data items into a number of smaller, disjoint sets of data items such that the items in each set are somehow all similar to one another. Clustering has been studied widely, and applied to many problems. For a general discussion of clustering methods and how they may be applied to the problem of reverse engineering large, legacy software systems, we refer the reader to [12]. In general, data clustering methods are at least somewhat similar to the methods presented here since they rely on some sort of metric to quantify the degree of relation of two different data objects. This metric may be arbitrarily complex, depending on the specific method (so long as the triangle inequality holds), and may rely even on some notion of the semantics or control flow of the program. Using the k-cut method however, functions are simply considered similar if they call or are called by a set of functions that are somehow related (the strongest relationship occurs when each call one another). Another key difference is that the k-cut method is a graph theoretic approach. The k-cut method can thus avoid many of the well-studied pitfalls inherent to cluster analysis, at the cost of a simpler notion of object similarity. We refer to [3] for a discussion of some of the issues that must be considered when clustering, and to [12] for an excellent descriptions of some reverse engineering methods that make use of cluster analysis.

Another, and somewhat more dissimilar approach is the set of methods based on the notion of dominance (an example of a broader method that makes use of dominance is [2]). In such methods, the hierarchical structure of a program is used in an attempt to infer some sort of organization. As was discussed in Section 4, it is precisely the case where the program to be analyzed has such a structure that the k-cut method has a propensity to create tiny modules. However, the k-cut method appears to be most applicable precisely in those situations where dominance analysis fails; that is, when no clear, hierarchical organization exists.

6. Conclusions

In this paper, we have explored the use of a class of graph-based algorithms known as *minimal k-cut algorithms* on the problem of producing modularizations of huge, legacy software systems where no current modularization exists. We have chosen (and modified when necessary) several algorithms from the area of graph theory in order to build an implementation of an approximate k-cut method that exhibits acceptable performance, even when used with very large software systems. We have developed a metric called *sociability* that is used to determine appropriate values for k and control the extent of the modularization. We have shown that the algorithms described are scalable, and produce good results on

many call graphs. Finally, we are currently working on modifications to the method that will address the shortcomings which we described. Specifically, we wish to more tightly couple the notion of sociability with our algorithms, so that cuts which maximize the sociability of the resulting modularizations are specifically chosen.

References

[1] M. N. Armstrong and C. Trudeau. Evaluating Architectural Extractors. *Proceedings of the Fifth Working Conference on Reverse Engineering*, 1998.

[2] E. Burd and M. Munro. Assisting Human Understanding to Aid the Targeting of Necessary Reenginerring Work. *Proceedings of the Fifth Working Conference on Reverse Engineering*, 1998.

[3] E. Dalhous, D. S. Johnson, C. Papadimitriou, P. Seymour and M. Yannakakis. The Complexity of Multiway Cuts. *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, 1992.

[4] B. Everitt. *Cluster Analysis*. Halstead Press, New York, NY, 1993.

[5] L. R. Ford, Jr and D. R. Fulkerson. Maximum Flow Through a Network. *Canadian Journal of Mathematics* 8 (1956).

[6] A. Goldberg. Efficient Graph Algorithms for Sequential and Parallel Computers. *PhD Thesis, Department of EE and CIS, MIT*, 1987.

[7] A. Goldberg and R. Trajan. A New Approach to the Maximum Flow Problem. In *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing*, 1986.

[8] R. E. Gomory and T. C. Hu. Multi-Terminal Network Flows. *SIAM Journal* 9, 4 (1961).

[9] R. Kazman and J. Carriere. Playing Detective: Reconstructing Software Architecture from Available Evidence. In *Journal of Automated Software Engineering*, 1998.

[10] H. Muller. Understanding Software Systems Using Reverse Engineering Technologies: Research and Practice. *Proceedings of the 18th In. Conf. on software Engineering*, 1996.

[11] H. Saran and V. Vazirani. Finding *k*-Cuts Within Twice the Optimal. *SIAM Journal of Computing*, 24, 1 (1995).

[12] T. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. *Proceedings of the Fourth Working Conference on Reverse Engineering*, 1997.