Applying Spectral Methods to Software Clustering

Ali Shokoufandeh, Spiros Mancoridis, Matthew Maycock
Department of Computer Science
Drexel University, Philadelphia, PA, USA
E-mail{ashokouf,smancori,ummaycoc}@mcs.drexel.edu

Abstract

The application of spectral methods to the software clustering problem has the advantage of producing results that are within a known factor of the optimal solution. Heuristic search methods, such as those supported by the Bunch clustering tool, only guarantee local optimality which may be far from the global optimum. In this paper, we apply the spectral methods to the software clustering problem and make comparisons to Bunch using the same clustering criterion. We conducted a case study, involving 13 software systems, to draw our comparisons. There is a dual benefit to making these comparisons. Specifically, we gain insight into (1) the quality of the spectral methods solutions; and (2) the proximity of the results produced by Bunch to the optimal solution.

1. Introduction and Motivation

Views of a software system's structure are typically represented as directed graphs. When these graphs become large, clustering algorithms can be used to partition them. A variety of criteria have been used to partition software graphs. A reasonable criterion is to partition a graph so that clusters exhibit high cohesion but low coupling. We used this criterion in our earlier work on the Bunch clustering system [21] and use this same criterion in this work.

Software clustering tools create abstract structural views of the resources (e.g., subsystems, modules, interfaces, classes) and relations (e.g., procedure calls, inheritance relationships) present in the source code. These views, which can be considered a "road map" of a system's structure, can help software engineers cope with the complexity of software development and maintenance.

The first step of a typical design extraction process (see Figure 1) is to determine the resources and relations in the source code and store the resultant information in a database. Readily available source code analysis tools – supporting a variety of programming languages – can be used for this step [7, 8, 19]. After the resources and relations have been stored in a database, the database is queried and a Module Dependency Graph (MDG) is created. For now, consider the MDG to be a directed graph that represents the software modules (e.g., classes, files, packages) as nodes, and the relations (e.g., function invocation, variable usage, class inheritance) between modules as directed edges. Once the MDG is created, clustering algorithms can be used to partition the MDG. The clusters in the partitioned MDG represent subsystems that contain one or more modules, relations, and possibly other subsystems. The final result can be visualized and browsed using a graph visualization tool [28].

Figures 2 and 3 show the MDG and partitioned MDG, respectively, of a file system. The file system is a C++ program that was written at the AT&T Research Labs. This program, which consists of 50,830 lines of C++ code, implements a file system service that 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. In this example, the modules of the MDG are C++ source files. Each edge in the MDG represents at least one relationship between program entities in the two corresponding source modules.

In this paper we answer two fundamental questions pertaining to the software clustering problem:

1. *How can a software engineer determine – within a reasonable amount of time and computing resources – if the solution produced by a software clustering algorithm is good or not?*
2. *Can an algorithm be created that guarantees a solution – within a reasonable amount of time and computing resources – that is close to the ideal solution?*

From a practical aspect, the answers to these questions are important because they provide increased confidence to software engineers who analyze systems. From a theoretical aspect, these answers are important because they provide an approximation algorithm to a known NP-Hard problem, in addition to a method for comparing any solution,

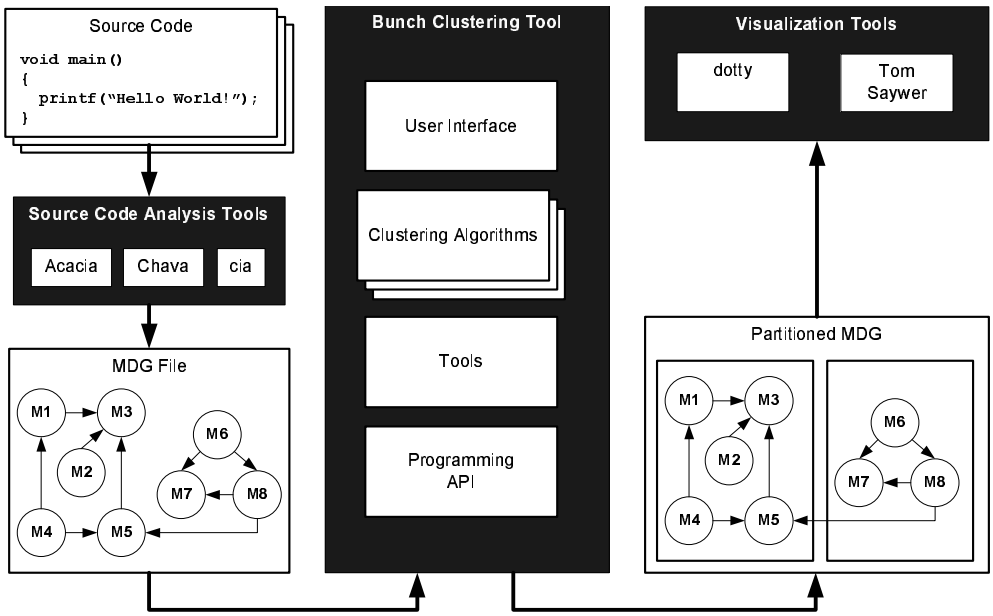

Figure 1. The design extraction process

even those produced by other algorithms that use the same clustering criterion we do (i.e., coupling-cohesion tradeoff), to the optimal solution.

The rest of the paper is structured as follows: Section 2 outlines related work, Section 3 outlines our spectral methods for clustering software graphs. Section 4 evaluates our methods and compares them to an existing method, namely Bunch. Finally, Section 5 outlines our conclusions and plans for future work.

2. Related Work

The primary bodies of related work are from the areas of software clustering and combinatorial optimization.

2.1. Software Clustering

Many of the clustering techniques published in the literature can be categorized by the way they create clusters. A survey article published by Wiggerts [41] is a good starting point to learn about software clustering. Hutchens and Basili [15] developed an algorithm that clusters procedures into modules by measuring the interaction between pairs of procedures. Schwanke et al [32, 33] introduced the notion of using design principles such as low coupling and high cohesion to create clusters. Choi and Scacchi [9] describe a clustering technique based on maximizing the cohesiveness of clusters by evaluating the exchange of resources between modules. Müller et al [27] implemented several software clustering heuristics in the Rigi tool that (a) measure the relative strength between interfaces, (b) identify omnipresent modules, and (c) use similarity of module names. Clustering based on similar patterns in implementation information (*e.g.*, module file names and directory structure) has been investigated by Anquetil et al [2, 3] and Tzerpos et al [39]. Concept analysis [20, 40, 1] has also been explored in the software clustering research. Our research on the Bunch system is based on using heuristic search techniques [22, 11, 21, 26] to determine clusters using the "low coupling and high cohesion" criterion.

The Application of data mining approaches to the software clustering problem was investigated by Sartipi et al [31, 30]. The authors' clustering approach involves using data mining techniques to annotate nodes in a software graph with association strength values. These values are used to partition the graph into clusters.

Now that a plethora of approaches to software clustering exist, the validation of clustering results is starting to attract the interest of the Reverse Engineering research community. Many of the clustering techniques published in the literature present case studies, where the results are evaluated by the authors or by the developers of the systems being studied. This evaluation technique is very subjective. Recently, researchers have begun developing infrastructure to evaluate clustering techniques, in a semi-formal way, by proposing similarity measurements [2, 3, 25]. These measurements enable the results of clustering algorithms to be compared to each other, and preferably to be compared to an

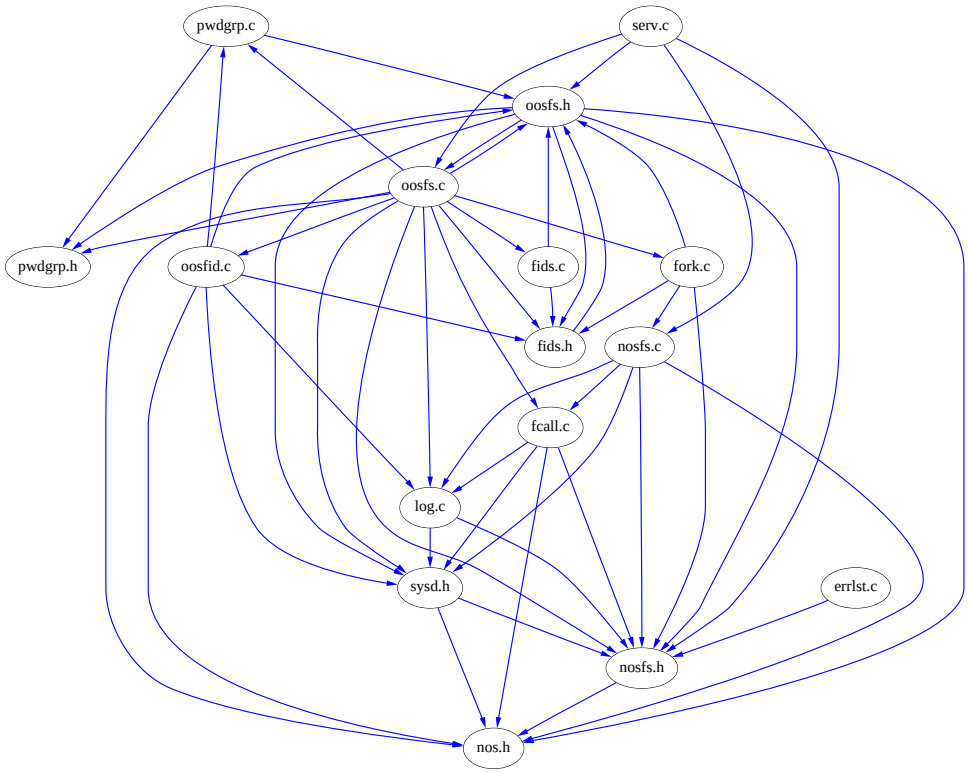

Figure 2. MDG of a file system

agreed upon "benchmark" standard. Note that the "benchmark" standard needn't be the optimal solution in a theoretical sense. Rather, it is a solution that is perceived as being "good enough".

Existing clustering techniques neither provide a guarantee on the quality of their solutions nor any indication of a solution's proximity to the optimum. Bunch, for example, uses several methods to find solutions, such as hill-climbing and genetic algorithms. Hill-climbing only guarantees local optimality, but makes no guarantees of global optimality. Genetic algorithms are another type of search, like hill-climbing, that does not guarantee the quality of its solution, not even with respect to local extrema. Neither method indicates how good a solution is with respect to the optimal solution. Not being able to meet either of these criteria is unsatisfactory.

2.2. Combinatorial Optimization

Most of the research on polynomial-time approximation algorithms for graph clustering is concentrated on finding the lower-bounds of graph bisection methods (see [29] for a survey). There has been some work in the early 1970s on the eigen-value characterization of the upper and lower bounds of objective functions that are closely related to the software clustering problem. The idea of using the eigenvalues to find the partitions of undirected graphs originated in the work of Donath and Hoffman, as well as Fiedler [10, 12].

These eigen-value characterizations have been studied to design efficient algorithms for a variety of problems in the segmentation, grouping, verification and matching of graphs arising in the domains of computer vision and data mining [34, 37, 35, 36, 23, 24].

Recently, researchers have paid attention to approximation algorithms for graph partitioning [4, 38, 42, 14]. Most of the these algorithms are based on solving simplified (relaxed) forms of the partitioning problem. The main benefit underlying these techniques is that the relaxations produce tractable problems. In related work [16] we showed how to solve simplified variations of relaxed optimization problems, such as matrix scaling and balancing, in polynomial time.

3. Spectral Methods for Software Clustering

We define a Module Dependency Graph $MDG = (M, R)$, where M is the set of named modules in the software system, and $R \subseteq M \times M$ is a set of ordered pairs

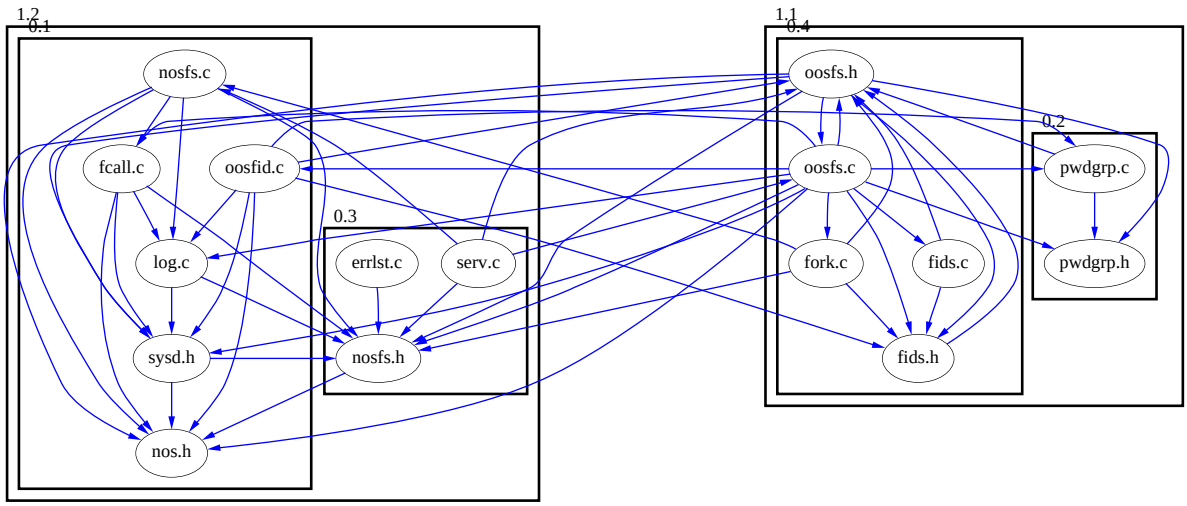

Figure 3. Clustered MDG of a File System

$\langle u, v \rangle$ which represent the source-level relationships that exist between modules u and v.

Given the MDG of a software system, we search for a "good" partition of the MDG. We accomplish this by treating clustering as an optimization problem where the goal is to maximize the value of an objective function. This function characterizes the trade-off between coupling (*i.e.*, connections between the components of two distinct clusters) and cohesion (*i.e.*, connections between the components of the same cluster). We refer to our objective function as the *Modularization Quality* (*MQ*) of an MDG partition. *MQ* adheres to our assumption that well-designed software systems are organized into cohesive clusters that are loosely interconnected. *MQ* is designed to reward the creation of highly-cohesive clusters and penalize excessive inter-cluster coupling.

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{\mu_i}{\mu_i + \varepsilon_i} & otherwise \end{cases} \quad (1)$$

The *MQ* for an MDG partitioned into k clusters is calculated by summing the *Cluster Factor* (*CF*) for each cluster in the partitioned MDG. The Cluster Factor, CF_i, for cluster i is defined as a normalized ratio between the total number of internal edges and the total number of external edges that originate in cluster i and terminate in other clusters. We refer to the internal edges of a cluster as intra-edges (μ_i) and the edges between two distinct clusters as inter-edges (ε_i).

Let us assume that we are interested in partitioning the nodes of an MDG into two sets M_1 and M_2 that maximizes the MQ function. Let X be an $n = |M|$-dimensional indicator vector, with $x_i = 1$ if module i belongs to M_1 in the optimal bisection, and -1 otherwise. Also, let $\deg(u)$ denote the out-degree of module u in the MDG, and $\mathcal{L}$ denote the Laplacian matrix of the MDG, i.e., $\mathcal{L}_{u,u} = d(u)$, $\mathcal{L}_{u,v} = -1$ if $u \neq v$ and $\langle u, v \rangle$ is a source-level relation between modules u and v, and 0 otherwise. Using the fact that $CF_i = 1 - \frac{\varepsilon_i}{\mu_i + \varepsilon_i}$, *MQ* can be reformulated as:

$$\begin{aligned} MQ &= 2 - \left(\tfrac{\varepsilon_1}{\mu_1+\varepsilon_1} + \tfrac{\varepsilon_2}{\mu_2+\varepsilon_2} \right) \\ &= 2 - \left(\frac{\sum_{x_i>0, x_j<0} -\mathcal{L}_{i,j} x_i x_j}{\sum_{x_i>0} \deg(i)} + \frac{\sum_{x_i<0, x_j>0} -\mathcal{L}_{i,j} x_i x_j}{\sum_{x_i<0} \deg(i)} \right) \end{aligned} \quad (2)$$

Hence, we can reformulate the optimal bisection of an MDG as finding an appropriate $\{-1, 1\}$ assignment of indicator variables x_i, $1 \leq i \leq |M|$, that minimizes the following expression:

$$MQ^*(X) = \frac{\sum_{x_i>0, x_j<0} -\mathcal{L}_{i,j} x_i x_j}{\sum_{x_i>0} \deg(i)} + \frac{\sum_{x_i<0, x_j>0} -\mathcal{L}_{i,j} x_i x_j}{\sum_{x_i<0} \deg(i)} \quad (3)$$

Let $\mathcal{D}$ denote an $|M| \times |M|$ diagonal matrix with $\mathcal{D}_{u,u} = d(u)$, α denote the normalized degree of modules in set

M_1 (i.e., $\alpha = \frac{\sum_{x_i>0} d(i)}{\sum_i d(i)}$), and $\mathbf{e}$ denote an identity vector whose entries are all 1s. Then the optimization problem in (3) can be reformulated as an integer-programming problem of the following quadratic form:

$$MQ^* = \frac{(\mathbf{e}+X)^t \mathcal{L}(\mathbf{e}+X)}{4\alpha \mathbf{e}^t \mathcal{D}\mathbf{e}} + \frac{(\mathbf{e}-X)^t \mathcal{L}(\mathbf{e}+X)}{4(1-\alpha)\mathbf{e}^t \mathcal{D}\mathbf{e}} \quad (4)$$

If $Y = (\mathbf{e}+X) - \frac{\alpha}{1-\alpha}(\mathbf{e}-X)$, the optimal solution to the MDG bisection in (4) can be obtained from the following optimization problem:

$$\begin{array}{ll} MQ^* & = \text{Minimize } \frac{Y^t \mathcal{L} Y}{Y^t \mathcal{D} Y} \\ \text{Subject to:} & y_i \in \{1, \frac{\alpha}{1-\alpha}\},\ 1 \le i \le |M| \\ & Y^t \mathcal{L} e = 0 \end{array} \quad (5)$$

Removing the constraints $y_i \in \{1, \frac{\alpha}{1-\alpha}\},\ 1 \le i \le |M|$, from the optimization problem in (5) results in the well-known eigen-value problem known as Rayleigh's quotient [13]. It is known that the minimizer of any quadratic form $\frac{X^t \mathcal{L} X}{X^t X}$ is an eigen-value of the MDG's adjacency matrix A. Using the change of variable $Z = \mathcal{D}^{\frac{1}{2}} Y$ will reduce the optimization problem in (5) to computing the eigen-vector corresponding to second smallest eigen-value of matrix $\mathcal{D}^{\frac{-1}{2}} \mathcal{L} \mathcal{D}^{\frac{-1}{2}}$.

In the ideal case, the entries of the solution to the optimization problem in (5) assume one of two discrete values, and the values of the entries can be used to determine partitions M_1 and M_2. Unfortunately, the entries of an eigen-vector can assume any real value since we removed the constraints $y_i \in \{1, \frac{\alpha}{1-\alpha}\},\ 1 \le i \le |M|$ from our optimization problem.

In the absence of a binary solution to (5) we can sort the entries of eigen-vector Y and find an appropriate splitting point that will generate a partition $\{M_1, M_2\}$ that maximizes MQ in (2). After partitioning the MDG into clusters M_1 and M_2, we can run the bisection procedure recursively, in a top-down fashion, on the sub-MDGs induced by sets M_1 and M_2. Each branch of this recursion terminates when a further partitioning of its clusters does not improve the value of MQ.

3.1 Recursive Bisection Algorithm

Our recursive bisection clustering algorithm can be summarized as follows:

1. Given an MDG on software modules M, construct the diagonal matrix of degrees $\mathcal{D}$ to create the Laplacian matrix $\mathcal{L}$.
2. Define the eigen-equation $\mathcal{L}X = \lambda \mathcal{D}X$ for the $|M|$-dimensional vector X. Then, compute all of the roots of this system (the eigen-values and eigen-vectors) using standard techniques [13].
3. The eigen-vector corresponding to the smallest non-zero eigen-value will be used as the characteristic vector for the bisection. Use the entries of this vector to split the modules of the MDG so that the break-point maximizes the new value of MQ.
4. If the bisection improves the quantity of MQ, then bisect each sub-MDG obtained in the previous step, recursively. Otherwise, stop the clustering algorithm.

In order to improve the quality of the recursive bisection algorithm, we added two post-processing steps. A module u in cluster $\mathcal{C}$ is an *isolated* element, if all the incoming or outgoing edges adjacent to u are from modules outside of $\mathcal{C}$. In the clean-up step, we first remove all isolated modules from every cluster generated by the recursive bisection. Then, we re-execute the algorithm on the induced MDG defined on these modules. Finally, if after the clean-up step there are still isolated modules, we try to locate more suitable alternative clusters (i.e., that result in a higher MQ value) to house these modules.

3.2 Solution Quality Guarantee of the Recursive Bisection Algorithm

There is a rich body of work in the Computer Science Theory community on the evaluation of clustering algorithms. Most of the recent work in this area has focused on measuring the notions of *conductance volume* and *normalized inter-cluster volume* [5, 6, 18, 17]. Conductance volume for a set is defined as the ratio of the number of edges inside the set over the total number of edges adjacent to the vertices in the set. Similarly, the normalized inter-cluster volume is defined as the ratio of edges leaving the set over the total number of edges adjacent to the vertices in the set. In fact, the conductance and normalized inter-cluster volumes are the two main terms of our MQ function.

Formally, a clustering $\{\mathcal{C}_1, ..., \mathcal{C}_\ell\}$ of M is called an (α, ϵ)-clustering if the conductance volume of each cluster is at least α, and the normalized inter-cluster volume is at most an ϵ-fraction of the total number of edges. Assume that (α^*, ϵ^*) denotes the conductance and normalized inter-cluster volumes for the optimal clustering on an MDG. Recently, Kannan et al [17] showed that if the measure of quality for a cluster is the normalized volume, then any recursive approximate-cut algorithm will generate a clustering with a conductance volume of $\tilde{\alpha} = \frac{\alpha^*}{c_1 \log^2 |M|}$ and a normalized inter-cluster volume of $\tilde{\epsilon} = c_2 \epsilon \log^2 |M|$, for absolute constants c_1 and c_2. They, subsequently, generalized their result

to show that if the measure of quality for a cluster S is any function of the following form:

$$\Phi(S) = \frac{\sum_{u \in S, v \notin S} A_{u,v}}{\min(\mathrm{Vol}(S), \mathrm{Vol}(M \setminus S))} \quad (6)$$

(where the Vol of a set is the number of edges of the set and $A_{u,v}$ is the adjacency structure between u and v) then the optimization algorithm that is based on this function will have the same performance guarantee, albeit with different constants c_1 and c_2. It is easy to see that the formulation of MQ in (4) satisfies a similar condition on every cluster and, thus, will have a similar performance guarantee. In short, the recursive bisection algorithm will generate clusters that have a conductance volume within a factor $\frac{1}{c_1 \log^2 n}$ of the optimal and an inter-cluster volume within a factor of $c_2 \log^2 n$ of the optimal.

4. Evaluation

For small graphs, our algorithm gives excellent performance. For larger graphs, however, the performance is not as good. Computing eigen-values takes cubic time, and bisecting a graph recursively can, in the worst case, takes $n-1$ iterations, giving a worst-case complexity of $\Theta(n^4)$. The results, however, are deterministic, unlike other clustering techniques that use hill-climbing and genetic algorithms.

We have compared the results of our algorithm with the results produced by Bunch on the software systems described in Table 1. The MDGs of these systems are graphs where the nodes are classes (for Java and C++) and files (for C) and the edges are function or method calls and variables usage. Table 2 gives a comparison of the result and time ratios. For small systems (< 100 modules), the fitness function values for recursive bisection are within 0.85 of Bunch's fitness value results. For large systems, the results of recursive bisection are never below 0.639 of Bunch's results. This is understandable, as it is difficult for our algorithm to compensate for early mistakes that are not corrected by its clean-up phases. Bunch, however, can detect and correct such problems when it looks at the neighbors of it's current state and sees improvement in the correction. This problem is more prominent in the larger graphs because there is more opportunity for error.

System Name	Description
Compiler	Turing Language Compiler
LSLayout	Layout Algorithm
Boxer	Graph Drawing Tool
Mini-Tunis	Small Operating System
ISpell	Unix Spell Checker
Modularizer	MDG Graph Generator
RCS	Revision Control System
Bison	Compiler compiler
CIA	C Source Code Analyzer
Grappa	Graph Drawing Applet
Swing	Java GUI Library
Linux Kernel	Kernel for the Linux OS
Proprietary Compiler	Industrial Strength Compiler

Table 1. The systems of our case study

System Name	Nodes	Time-Ratio	MQ-Ratio
Compiler	13	0.859	1.033
LSLayout	17	0.895	0.962
Boxer	18	0.953	0.986
Mini-Tunis	20	0.929	0.980
ISpell	24	0.931	0.902
Modularizer	26	0.939	0.966
RCS	29	0.854	0.879
Bison	37	1.037	0.918
CIA	38	1.024	0.944
Grappa	86	1.796	0.881
Swing	413	13.559	0.755
Linux Kernel	916	35.268	0.639
Proprietary Compiler	939	21.893	0.691

Table 2. Recursive Bisection versus Bunch (the ratios are Bisection / Bunch)

5. Conclusions & Future Work

There are many good software clustering algorithms. Software clustering, however, is known to be NP-hard, and, thus, for clustering algorithms to be useful, they must provide sub-optimal answers or face an exponential running time.

We have shown that our spectral clustering algorithm gives a bounded approximation of the optimal clustering. Our algorithm, however, is generally worse than Bunch in quality of solution and running time, and only gets worse as the size of input increases. This observation implies that Bunch yields answers within a bounded approximation of the optimal solution, and does so efficiently.

In the future we hope to generalize our technique to produce solutions that may be better than those produced by Bunch. Specifically, for a *k-section*, $2 < k \leq n$, instead of assigning a variable x_i, $1 \leq i \leq n$, we can assign a k-dimensional vector $X^{(i)}$ to each module M_i in the MDG.

The vector's entries can be either 0s or 1s. Intuitively, in an optimal solution, all modules with similar vectors will belong to the same cluster. More specifically, let X denote an $n \times k$ matrix for which the non-zero entries of column j represent the nodes contained in cluster S_j, $1 \leq j \leq k$, and its i-th row corresponds to vector $X_{(i)}$, $1 \leq i \leq n$. In addition, let A represent the adjacency matrix of the MDG, and D denote the $n \times n$ diagonal matrix with $D_{i,i} = d_i$. Then, the optimization problem in (2) can be generalized to an arbitrary value of k as follows:

$$\textbf{Maximize} \qquad MQ = \sum_{1 \leq i \leq k} \frac{X_{(i)}^T A X_{(i)}}{X_{(i)}^T D \mathbf{e}}$$

$$\textbf{Subject to} \qquad \mathbf{e}^T X^T X \mathbf{e} = n$$
$$X \mathbf{e}_k = e,$$
$$X_{i,j} \in \{0,1\},\ 1 \leq i \leq n,\ 1 \leq j \leq n$$

where $\mathbf{e}_k$ is a vector with entry k equal 1, and 0 everywhere else, I_n is the identity matrix of order n.

Note that the constraints of this optimization problem guarantee exactly one non-zero entry in each row of matrix X. Hence, each module can belong to exactly one of the k clusters $\{S_1, ..., S_k\}$.

A second approach to generalizing the bisection algorithm is to use the higher order eigen-vectors given by the solution of (5). An argument can be made to show that the first eigen-vectors corresponding to first k eigen-vectors of the Laplacian matrix are the non-integral solutions that subpartition the first $k-1$ parts in an optimal way. To make this approach practical we must bound the rounding error for each eigen-vector computation to control the quality of the solution. The higher-order eigen-vectors must satisfy all of the conditions of the optimization problem in (5).

Acknowledgments

This research is sponsored by grants CCR-9733569 and CISE-9986105 from the National Science Foundation (NSF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] N. Anquetil. A comparison of graphis of concept for reverse engineering. In *Proc. Intl. Workshop on Program Comprehension*, 2000.

[2] N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with hierarchical clustering algorithms as software remodularization methods. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[3] N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering*, 1999.

[4] T. Asano. Approximation algorithms for max sat: Yannakakis vs, 1997.

[5] Y. Azar, A. Fiat, A. R. Karlin, F. McSherry, and J. Saia. Spectral analysis of data. In *ACM Symposium on Theory of Computing*, pages 619–626, 2001.

[6] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. In *ACM Symposium on Theory of Computing*, pages 626–635, 1997.

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

[8] Y. Chen, E. R. Gansner, and E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *Proceedings of the European Conference on Software Engineering/Foundations of Software Engineering*, 1997.

[9] S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1990.

[10] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17:420–425, 1973.

[11] D. Doval, S. Mancoridis, and B. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, 1999.

[12] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czech. Math. J.*, 25(100):619–633, 1975.

[13] G. Golub and C. V. Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[14] Q. Han, Y. Ye, H. Zhang, and J. Zhang. On approximation of max-vertex-cover. In *17th International Symposium on Mathematical Programming*, Atlanta, Georgia, 2000.

[15] D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, Aug. 1995.

[16] B. Kalantari, L. Khachiyan, and A. Shokoufandeh. On the complexity of matrix balancing. *SIAM Journal on Matrix Analysis and Applications*, 18(2):450–463, 1997.

[17] R. Kannan, S. Vempala, and A. Vetta. On clusterings: Good, bad and spectral. In *Proc. 41st Symposium on Foundations of Computer Science, FOCS'00*, Redondo Beach, CA, 2000.

[18] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins. The Web as a graph: Measurements, models, and methods. In T. Asano, H. Imai, D. T. Lee, S. Nakano, and T. Tokuyama, editors, *Proc. 5th Annual Int. Conf. Computing and Combinatorics, COCOON*, volume 1627. Springer-Verlag, 1999.

[19] J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 314–325, 1999.

[20] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, 1997.

[21] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, Aug. 1999.

[22] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th Intl. Workshop on Program Comprehension*, 1998.

[23] D. McWherter, M. Peabody, W. C. Regli, and A. Shokoufandeh. Transformation invariant shape similarity comparison of models. In *Proc. ASME Design Engineering Technical Confs.*, 2001.

[24] D. McWherter, M. Peabody, A. Shokoufandeh, and W. C. Regli. Database techniques for archival of solid models. In *Proc. 6th ACM/SIGGRAPH Symp. on Solid Modeling and Applications*, pages 78–87, 2001.

[25] B. Mitchell and S. Mancoridis. Craft: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'01)*, October 2001.

[26] B. Mitchell, S. Mancoridis, and M. Traverso. An architecture for distributing the computation of software clustering algorithms. In *Proceedings of the IEEE/IFIP Working International Conference on Software Architecture (WICSA'01)*, August 2001.

[27] H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.

[28] S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, 1994.

[29] R.B.Boppana. Eigevalues and Graph Bisection: An Average Case Analysis. In *Proc. of the 28 Annual Symposium on Computer Science*, pages 280–285, 1988.

[30] K. Sartipi and K. Kontogiannis. Component clustering based on maximal association. In *Proc. Working Conf. on Reverse Engineering (WCRE'01)*, 2001.

[31] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architectural design recovery using data mining techniques. In *Proc. of the European Conf. on Software Maintenance and Reengineering (CSMR'00)*, 2000.

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

[33] R. Schwanke and S. Hanson. Using Neural Networks to Modularize Software. *Machine Learning*, 15:137–168, 1998.

[34] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[35] A. Shokoufandeh and S. Dickinson. Applications of bipartite matching to problems in object recognition, 1999.

[36] A. Shokoufandeh, S. Dickinson, K. Siddiqi, and S. W. Zucker. Indexing using a spectral encoding of topological structure. In *Proc. Computer Vision and Pattern Recognition*, pages 491–497, 1999.

[37] K. Siddiqi, A. Shokoufandeh, S. Dickinson, and S. Zucker. Shock graphs and shape matching, 1999.

[38] Sviridenko. Best possible approximation algorithm for MAX SAT with cardinal constraint. In *APPROX: International Workshop on Approximation Algorithms for Combinatorial Optimization*, 1998.

[39] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension driven clustering. In *Proceedings of the Working Conference in Reverse Engineering (WCRE'00)*, 2000.

[40] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proc. International Conference on Software Engineering*, Aug. 1999.

[41] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proc. Working Conference on Reverse Engineering*, 1997.

[42] U. Zwick. Outward rotations: A tool for rounding solutions of semidefinite programming relaxations, with applications to MAX CUT and other problems. In *ACM Symposium on Theory of Computing*, pages 679–687, 1999.