Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software xxx (2004) xxx–xxx
The Journal of Systems and Software
www.elsevier.com/locate/jss

Spectral and meta-heuristic algorithms for software clustering

Ali Shokoufandeh, Spiros Mancoridis *, Trip Denton, Matthew Maycock

Department of Computer Science, College of Engineering, Drexel University, Philadelphia, PA, USA

Received 1 April 2003; received in revised form 16 July 2003; accepted 2 March 2004

Abstract

When large software systems are reverse engineered, one of the views that is produced is the system decomposition hierarchy. This hierarchy shows the system's subsystems, the contents of the subsystems (i.e., modules or other subsystems), and so on. Software clustering tools create the system decomposition automatically or semi-automatically with the aid of the software engineer.

The Bunch software clustering tool shows how meta-heuristic search algorithms can be applied to the software clustering problem, successfully. Unfortunately, we do not know how close the solutions produced by Bunch are to the optimal solution. We can only obtain the optimal solution for trivial systems using an exhaustive search.

This paper presents evidence that Bunch's solutions are within a known factor of the optimal solution. We show this by applying spectral methods to the software clustering problem. The advantage of using spectral methods is that the results this technique produces are within a known factor of the optimal solution. Meta-heuristic search methods 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. We conducted a case study to draw our comparisons and to determine if an efficient clustering algorithm, one that guarantees a near-optimal solution, can be created.

1. Introduction and motivation

Legacy software systems are often reverse engineered in order to extract design information that can be used by software engineers to aid the software maintenance effort. Views of a software system's design are typically represented as directed graphs, where the nodes represent software entities such as functions, classes, modules, or files, and the directed edges represent binary relations between those entities such as function invocation, variable use, inheritance, module imports, and file inclusion. When these graphs become large, clustering algorithms can be used to partition them in order to make them easier to comprehend.

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 (Mancoridis et al., 1999) and use this same criterion in this work.

Software clustering tools create abstract structural views of the entities and relations 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 Fig. 1) is to determine the entities and relations in the source code and store the resultant data in either a database, as many source code analysis tools do (Chen, 1995), or a set of files, as many code fact extractors do (Holt et al., xxxx). This data can then be queried by the user to obtain information about the code's struc-

* Corresponding author.
E-mail addresses: ashokouf@cs.drexel.edu (A. Shokoufandeh), smancori@cs.drexel.edu (S. Mancoridis), tdenton@cs.drexel.edu (T. Denton), ummaycoc@cs.drexel.edu (M. Maycock).

0164-1212/$ - see front matter © 2004 Published by Elsevier Inc.
doi:10.1016/j.jss.2004.03.032

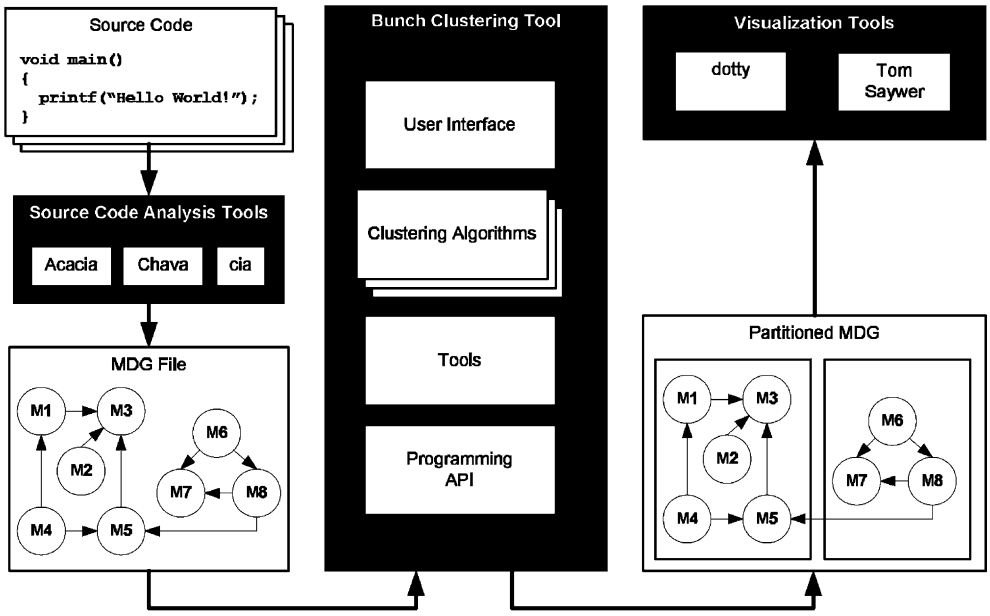

Fig. 1. The design extraction process.

ture. For example, one can find out if a function is reachable from another function in the same program.

In our work we use readily available source code analysis tools for this step (Chen et al., 1997; Korn et al., 1999). After the entities and relations have been stored in a database, the database can be queried to derive a Module Dependency Graph (MDG). 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 such as Dotty (North and Koutsofios, 1994).

Figs. 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 over 50K 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 (e.g., function invocation, variable usage) between program entities in the two corresponding source modules.

The Bunch software clustering tool applies meta-heuristic search algorithms (Clark et al., 2003) such as hill-climbing (Russell and Norvig, 2002), simulated annealing, and genetic algorithms (Goldberg, 1989). It has been shown, by extensive experimentation (Mitchell and Mancoridis, 2002; Mitchell and Mancoridis, 2003), that Bunch produces good results consistently and quickly. However, one known limitation of meta-heuristic search algorithms is their inability to guarantee the proximity of their solutions to the optimal solution. Another limitation is that meta-heuristic search algorithms often give poor results because they converge to local optimum solutions that are far from the optimal one. The fact that Bunch produces consistent results across many types of systems indicates that the search space has one or more basins of attraction that all converge to solutions of similar quality. However, we do not know if these solutions are close to the optimal solution.

In this paper we answer the following question:

Can an efficient software clustering algorithm, one that guarantees near-optimal solutions, be created?

From a practical aspect, the answers to this question is important because it provides an increased confidence to software engineers who analyze systems. From a theoretical aspect, the answer is important because it provides an approximation algorithm to a known NP-Hard problem (Garey and Johnson, 1979) as well as leads to a method for comparing clustering solutions (ones that use the same clustering criterion we do) to the optimal solution.

The rest of the paper is organized as follows: Section 2 reviews related research in (a) clustering algorithms for

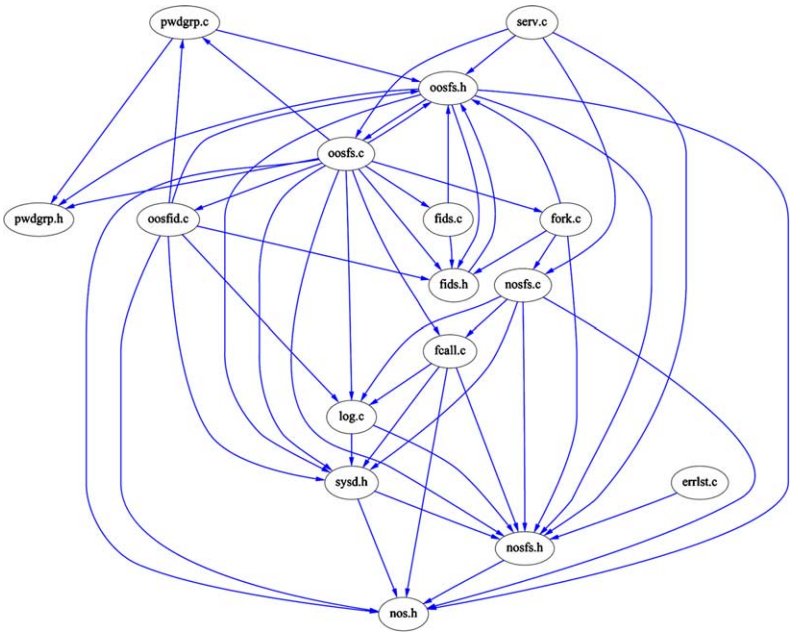

Fig. 2. MDG of a file system.

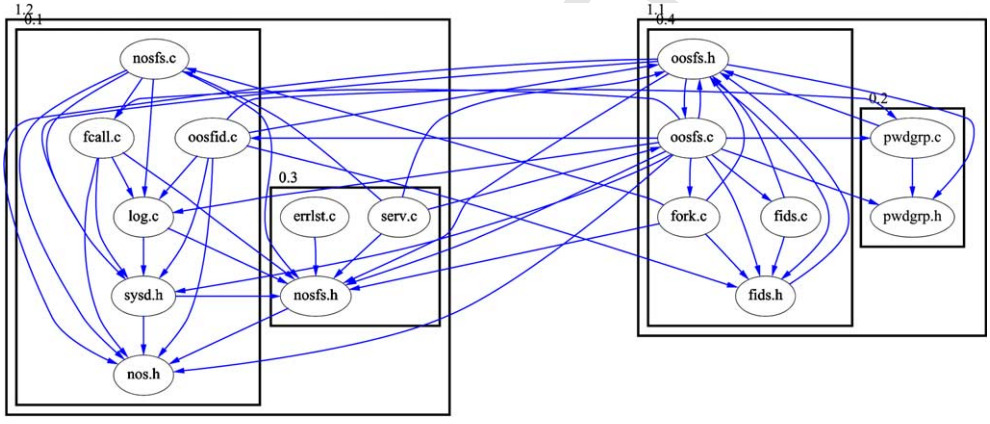

Fig. 3. Clustered MDG of a file system.

reverse engineering, (b) the clustering algorithms supported by Bunch, and (c) polynomial-time approximation algorithms for graph clustering; Section 3 describes our Spectral algorithm, which guarantees solutions that are within a known factor of the optimal solution in polynomial time; Section 4 describes a case study consisting of 13 software systems to compare the results produced by Bunch with those produced by the Spectral algorithm; Section 5 concludes the paper by identifying future research opportunities.

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 by Wiggerts (1997) is a good starting point to learn about software clustering. Hutchens and Basili (1995) developed an algorithm that clusters procedures into modules by measuring the interaction between pairs of procedures. Schwanke (1991) and Schwanke and Hanson (1998) introduced the notion of using design principles such as low coupling and high cohesion to create clusters. Choi and Scacchi (1990) describe a clustering technique based on maximizing the cohesiveness of clusters by evaluating the exchange of resources between modules. Müller et al. (1993) implemented several software clustering heuristics in the Rigi tool that (a) measure the relative strength between inter-

faces, (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. (1999), Anquetil and Lethbridge (1999) and Tzerpos and Holt (2000). Concept analysis (Lindig and Snelting, 1997; van Deursen and Kuipers, 1999; Anquetil, 2000) has also been explored in the software clustering research. Our research on the Bunch system is based on using meta-heuristic search techniques (Mancoridis et al., 1998; Doval et al., 1999; Mancoridis et al., 1999; Mitchell et al., 2001) 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. (2000) and Sartipi and Kontogiannis (2001). 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 broad range 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 (Anquetil et al., 1999; Anquetil and Lethbridge, 1999; Mitchell and Mancoridis, 2001). These measurements enable the results of clustering algorithms to be compared to each other, and preferably to be compared to an agreed upon "benchmark" standard. Note that the "benchmark" standard need not 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 hill-climbing, which 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.

2.1.1. The Bunch clustering algorithm

The Bunch tool implements a variety of meta-heuristic search algorithms to cluster graphs. Bunch's hill-climbing clustering algorithms (Mancoridis et al., 1998) start by generating a random partition of the *MDG*. Modules from this partition are then rearranged systematically in an attempt to find an "improved" partition. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. The hill-climbing search algorithm eventually converges when no improved partitions of the *MDG* can be found.

The Bunch genetic algorithm (GA) (Doval et al., 1999) uses operators such as selection, crossover, and mutation to determine a "good" partition of the *MDG*. This technique is especially good at finding solutions quickly, but we have found that the quality of the results produced by Bunch's hill-climbing algorithms are typically better. Hence, in this paper we will only be concerned with the hill-climbing algorithms.

Although each of Bunch's search algorithms works differently, they all examine partitions from the very large search space of *MDG* partitions. Note that the number of *MDG* partitions, the Bell number, is $O(N!)$, where N is the number of modules in the *MDG*. Thus, Bunch's search algorithms require a way to determine if one *MDG* partition is "better" than another. To address this need we define an objective function, which we call *Modularization Quality* (*MQ*), to evaluate the relative "quality" of *MDG* partitions.

The *MQ* function (see Eqs. (1) and (2)) works by calculating a value which we call the *Cluster Factor* (CF) for each cluster. Given an *MDG* partitioned into k clusters, *MQ* is calculated by summing CF for each cluster of the partitioned *MDG*. CF_i for cluster i ($1 \leqslant i \leqslant k$) is defined as a normalized ratio between the total weight of the internal edges (edges within the cluster) and half of the total weight of external edges (edges that exit or enter the cluster). The weight of the external edges is split in half in order to apply an equal penalty to both clusters that are connected by an external edge. We refer to the internal edges of a cluster as intra-edges (μ_i), and the edges between two distinct clusters i and j as inter-edges ($\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ respectively). If edge weights are not provided by the *MDG*, we assume that each edge has a weight of 1.

The *MQ* measurement design is based on the assumption that good software systems consist of a set of highly-cohesive subsystems (clusters in the *MDG*) that are loosely coupled together.

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 Boppana, 1988 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. Our algebraic formulation of software clustering problem is based on a matrix form representation of

a graph known as the Laplacian. The study of the connection between Laplacian and the eigen-values to find the partitions of undirected graphs originated in the work of Donath and Hoffman, as well as Fiedler (Donath and Hoffman, 1973; Fiedler, 1975). These spectral properties have also been used in a variety of graph algorithms, particularly algorithms for finding small separators in graphs (Guattery and Miller, 1998; Pothen et al., 1990). For a comprehensive survey of Laplacian matrices and their spectral properties the reader is referred to Chung's manuscript (Chung, 1997). Our approach is based on an integer programming formulation of the clustering problem. Integer programming deals with problems of maximizing or minimizing a multivariable objective function subject to linear and nonlinear constraints. Due to the robustness of the general model, a remarkably rich variety of problems can be formulated using this technique (Nemhauser and Wolsey, 1988).

Spectral clustering is a general framework for partitioning the rows and columns of matrices derived from the data in terms of few of their eigenvectors. In most cases the clustering problem will be related to a variation of cuts in graphs, and the spectral method will be an underlying technique for the approximation of graph partition problems (Chung, 1997; Spielman and Teng, 1996). For example, Sarkar and Soundararajan (2000) presented a spectral clustering technique based on an average cut measure. This measure is defined as the proportion of the total cut-link weight, normalized by the sizes of the partitions. Shi and Malik (2000) defined the notion of normalized cut for perceptual grouping and used spectral properties of distance matrices to construct such grouping in computer vision. For an extensive survey of recent results in spectral partitioning the reader is directed to (Ng et al., 2001) and (Kannan et al., 2000).

Eigen-value characterizations have also been applied 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 (Shi and Malik, 2000; Siddiqi et al., 1999; Shokoufandeh and Dickinson, 1999; Shokoufandeh et al., 1999; McWherter et al., 2001a; McWherter et al., 2001b). Recently, researchers have paid attention to approximation algorithms for graph partitioning (Asano, 1997; Sviridenko, 1998; Zwick, 1999; Han et al., 2000). Most of 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 (Kalantari et al., 1997) 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

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 objective 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).

What makes this an appropriate objective function is that it is based on a classical engineering tradeoff which states that well-designed systems are a loose synthesis of highly complex components. Software is almost always designed with this tradeoff in mind. For example, the structure of a compiler is typically a simple pipeline of compilation phases, where each phases is implemented as a complex set of lower-level software components. Our objective function was designed with this tradeoff in mind. Although we could have used other criteria for clustering the structure of a software system (e.g., similar names in files or functions), our objective function will cluster the structure of software to recover the architecture of the system. Specifically, the objective function is designed to identify the set of complex components that constitute individual software capabilities as well as to identify how these components interact with other complex components in the rest of the system.

We refer to our objective function as an additive function of cluster factors (CF):

$$MQ = \sum_{i=1}^{k} \mathrm{CF}_i \tag{1}$$

Here, CF_i for cluster i ($1 \leqslant i \leqslant k$) is a function of the normalized ratio between the total weight of the internal edges (μ_i) and the total weight of the external edges ($\varepsilon_{i,j}$ and $\varepsilon_{j,i}$):

$$\mathrm{CF}_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum_{\substack{j=1, \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \text{otherwise.} \end{cases} \tag{2}$$

We will translate this formulation of the MQ function into an optimization problem using the matrix representation of an MDGs. Let us assume that we are interested in partitioning the nodes of an MDG into two sets C_1 and C_2 that maximizes the MQ function. To formalize this objective function, we assign an indicator variable x_i to every module i in our MDG, which can have a value of $+1$ or -1. Specifically, $x_i = +1$ if module i belongs to C_1 in the optimal bisection of the MDG, and -1 otherwise.

We will start our reformulation by introducing some algebraic notations related to the structure of MDG. Let $\mathscr{A}$ denote the adjacency matrix of MDG, i.e., $\mathscr{A}_{u,v} = 1$ if $u \neq v$ and $\langle u, v\rangle$ is a source-level relation between modules u and v, let $\delta(u)$ denote the degree of module u in the MDG, and $\mathscr{L}$ denote the Laplacian matrix of the MDG, i.e., $\mathscr{L}_{u,u} = \delta(u)$, $\mathscr{L}_{u,v} = -1$ if $\mathscr{A}_{u,v} = 1$, and 0 otherwise.

We start our reformulation of MQ by translating each CF_i in terms of indicator variables x_is and Matrix $\mathscr{L}$. Since μ_1 and μ_2 respectively represent the total weight of the internal edges in C_1 and C_2, we will have:

$$2\mu_1 = \sum_{\substack{x_i>0,\\ x_j>0}} \mathscr{A}_{i,j}x_ix_j = \sum_{\substack{x_i>0,\\ x_j>0}} -\mathscr{L}_{i,j}x_ix_j. \tag{3}$$

Using the definition of Laplacian matrix $\mathscr{L}$, we can see that for every $i \in \{1,\ldots,|M|\}$, where M denotes the set of modules in the MDG:

$$\mathscr{L}_{i,i} = -\left(\sum_{\substack{x_i>0,\\ x_j>0}} \mathscr{L}_{i,j}x_ix_j + \sum_{\substack{x_i>0,\\ x_j<0}} \mathscr{L}_{i,j}x_ix_j\right), \tag{4}$$

the above in turn implies:

$$\mu_1 = \frac{1}{2}\sum_{x_i>0}\left(\mathscr{L}_{i,i} - \sum_{x_j<0} -\mathscr{L}_{i,j}x_ix_j\right), \tag{5}$$

similarly:

$$\mu_2 = \frac{1}{2}\sum_{x_i<0}\left(\mathscr{L}_{i,i} - \sum_{x_j>0} -\mathscr{L}_{i,j}x_ix_j\right). \tag{6}$$

So far we have reformulated the numerator of CF_1 and CF_2 in matrix form. Next, we rewrite the denominator $\mathscr{D}_i$ of each cluster factor CF_i in terms of matrix $\mathscr{L}$, and degree values δ. Observe that each denominator $\mathscr{D}_i$ is a function of total number of incident edges to C_i. Specifically:

$$\mathscr{D}_1 = \sum_{x_i>0}\mathscr{L}_{i,i} = \sum_{x_i>0}\delta(i), \tag{7}$$

and

$$\mathscr{D}_2 = \sum_{x_i<0}\mathscr{L}_{i,i} = \sum_{x_i<0}\delta(i). \tag{8}$$

We can use (5) and (7) to rewrite CF_1 in following form:

$$\mathrm{CF}_1 = \frac{\sum_{x_i>0}\mathscr{L}_{i,i} - \sum_{\substack{x_i>0,\\ x_j<0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i>0}\delta(i)} = 1 - \frac{\sum_{\substack{x_i>0,\\ x_j<0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i>0}\delta(i)}.$$

Similarly, using (6) and (8) we have:

$$\mathrm{CF}_2 = \frac{\sum_{x_i<0}\mathscr{L}_{i,i} - \sum_{\substack{x_i<0,\\ x_j>0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i<0}\delta(i)} = 1 - \frac{\sum_{\substack{x_i<0,\\ x_j>0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i<0}\delta(i)}.$$

We can combine these reformulation of CF_1 and CF_2 and restate the MQ in quadratic matrix form as follows:

$$MQ = 2 - \left(\frac{\sum_{\substack{x_i>0,\\ x_j<0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i>0}\delta(i)} + \frac{\sum_{\substack{x_i<0,\\ x_j>0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i<0}\delta(i)}\right).$$

This quadratic form can be interpreted as an optimization problem where the $+1$ and -1 values are assigned to variables x_i such that the quantity MQ is maximized. Clearly, MQ is maximized if the term in parentheses is minimized. This translates to the following minimization problem:

$$MQ^*(X) = \text{Minimize}\left(\frac{\sum_{\substack{x_i>0,\\ x_j<0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i>0}\delta(i)} + \frac{\sum_{\substack{x_i<0,\\ x_j>0}} -\mathscr{L}_{i,j}x_ix_j}{\sum_{x_i<0}\delta(i)}\right), \tag{9}$$

subject to $x_i \in \{-1,1\}$, $1 \leqslant i \leqslant |M|$.

The solution of this quadratic optimization problem is closely related to the spectral properties of the Laplacian matrix $\mathscr{L}$, and falls in the general context of computing the eigenvalues and eigenvectors of matrix $\mathscr{L}$. Similar techniques have been used for optimization problems arising in heat propagation of continuous systems, combinatorial techniques in graph partitioning, and image segmentation in computer vision (Chung, 1997; Shi and Malik, 2000). In the remaining of this section we will establish the connection between an approximation solution for the optimal bisection in (9) and the spectral properties of matrix $\mathscr{L}$.

Let Δ denote an $|M| \times |M|$ diagonal matrix with $\Delta_{u,u} = \delta(u)$. Also let α denote the normalized degree of modules in set C_1 (i.e., $\alpha = \frac{\sum_{x_i>0}\delta(i)}{\sum_i \delta(i)}$), and $\mathbf{e}$ denote an identity vector whose entries are all 1s. Then the optimization problem in (9) can be reformulated as an integer-programming problem of the following quadratic form:

$$MQ^* = \frac{(\mathbf{e}+X)^t\mathscr{L}(\mathbf{e}+X)}{4\alpha\mathbf{e}^t\Delta\mathbf{e}} + \frac{(\mathbf{e}-X)^t\mathscr{L}(\mathbf{e}+X)}{4(1-\alpha)\mathbf{e}^t\Delta\mathbf{e}}, \tag{10}$$

where X is the desired unknown vector of size $|M|$ whose entries have to be either $+1$ or -1. Using the substitution $\beta = \frac{\alpha}{1-\alpha}$ and the expansion of individual terms, we can restate (10) as:

$$MQ^* = \frac{(1+\beta)(X^t\mathscr{L}X + \mathbf{e}^t\mathscr{L}\mathbf{e})}{\beta\mathbf{e}^t\Delta\mathbf{e}} + \frac{2(1-\beta)\mathbf{e}^t\mathscr{L}X}{\beta\mathbf{e}^t\Delta\mathbf{e}} + \frac{2\beta X^t\mathscr{L}X}{\beta\mathbf{e}^t\Delta\mathbf{e}} - \frac{2\beta\mathbf{e}^t\mathscr{L}\mathbf{e}}{\beta\mathbf{e}^t\Delta\mathbf{e}} = \frac{((\mathbf{e}+X)-\beta(\mathbf{e}-X))^t\mathscr{L}((\mathbf{e}+X)-\beta(\mathbf{e}-X))}{\beta e^t\Delta\mathbf{e}}.$$

Observe that:

$$\sum_{x_i>0}\delta(i) = \beta\sum_{x_i<0}\delta(i),$$

which implies:

$$\begin{aligned}\beta e^t\Delta\mathbf{e} &= \beta\sum_i\delta(i) = \beta\left(\sum_{x_i<0}\delta(i)+\sum_{x_i>0}\delta(i)\right)\\ &= \beta\left(\sum_{x_i<0}\delta(i)+\beta\sum_{x_i<0}\delta(i)\right)\\ &= \beta\sum_{x_i<0}\delta(i)+\beta^2\sum_{x_i<0}\delta(i)\\ &= \sum_{x_i>0}\delta(i)+\beta^2\sum_{x_i<0}\delta(i).\end{aligned}$$

Let $Y = (\mathbf{e}+X)-\beta(\mathbf{e}-X)$, then:

$$\begin{aligned}Y^t\Delta Y &= ((e+X)-\beta(\mathbf{e}-X))^t\Delta((e+X)-\beta(\mathbf{e}-X))\\ &= (e+X)^t\Delta(e+X)-\beta(e+X)^t\Delta(e-X)\\ &\quad -\beta(e-X)^t\Delta(e+X)+\beta^2(e-X)^t\Delta(e-X)\\ &= (e+X)^t\Delta(e+X)+\beta^2(e-X)^t\Delta(e-X)\\ &= \sum_{x_i>0}\delta(i)+\beta^2\sum_{x_i<0}\delta(i) = \beta e^t\Delta\mathbf{e}.\end{aligned}$$

Using these equalities and the definition of vector Y, the optimal solution to the MDG bisection in (10) can be obtained from the following optimization problem:

$$MQ^* = \text{Minimize}\frac{Y^t\mathscr{L}Y}{Y^t\Delta Y} \tag{11}$$

Since the ith entry of vector X (i.e., x_i) can attain only the values $\{-1,+1\}$, the corresponding value in vector Y (i.e., y_i) can have only one of the two values $\{\frac{\alpha}{\alpha-1},1\}$. Removing the constraint $y_i \in \{\frac{\alpha}{\alpha-1},1\}$, $1 \leqslant i \leqslant |M|$, from the optimization problem in (11) results in the well-known eigen-value problem known as Rayleigh's quotient (Golub and Loan, 1996). It is known that the minimizer of any quadratic form $\frac{X^t\mathscr{L}X}{X^tX}$ is an eigenvalue of the MDG's Laplacian matrix $\mathscr{L}$. Using the change of variable $Z = \Delta^{\frac{1}{2}}Y$ will reduce the optimization problem in (11) to computing the eigenvector corresponding to second smallest eigenvalue of matrix $\Delta^{\frac{-1}{2}}\mathscr{L}\Delta^{\frac{-1}{2}}$.

In the ideal case, the entries of the solution to the optimization problem in (11) assume one of two discrete values, and the values of the entries can be used to determine clusters C_1 and C_2. Unfortunately, the entries of an eigenvector can assume any real value since we removed the constraint $y_i \in \{\frac{\alpha}{\alpha-1},1\}$, $1 \leqslant i \leqslant |M|$ from our optimization problem. Removing this constraint makes this problem tractable. In the absence of a binary solution to (11) we can use a deterministic rounding schema that is commonly used for the solution of integer programming problems (Nemhauser and Wolsey, 1988): sort the entries of eigenvector Y and find an appropriate splitting point that generates a partition $\{C_1, C_2\}$ that maximizes MQ in (9) (see Section 3.1 for details).

To understand why computing the optimal clustering at each spectral level produces a globally good clustering, it should be mentioned that through a similar line of argument that resulted in equation (11), one can show the eigenvector corresponding to third smallest eigenvalue is the relaxed solution for optimally sub-dividing the first two clusters obtained using the second eigenvector (Fiedler, 1975). In fact, the strategy of computing consecutive eigenvalues can be extended for subsequent clusters. In practice (Boppana, 1988), however, the rounding of real-valued solutions to discrete integer values for all values of k will generate highly unstable solutions. As a result, after partitioning the MDG into clusters C_1 and C_2, we can run the bisection procedure recursively, in a top–down fashion, on the sub-MDGs induced by sets C_1 and C_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 Δ to create the Laplacian matrix $\mathscr{L}$.
2. Define the eigenequation $\mathscr{L}X = \lambda\Delta X$ for the $|M|$-dimensional vector X. Then, compute all of the roots of this system (the eigenvalues and eigenvectors) using standard techniques (Golub and Loan, 1996).
3. The eigenvector corresponding to the smallest non-zero eigenvalue is 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 $\mathscr{C}$ is an *isolated* element, if all the incoming or outgoing edges adjacent to u are from modules outside of $\mathscr{C}$. In the clean-up step, we first remove all isolated modules from every cluster generated by the recursive bisection. Then, we re-execute the algo-

rithm 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* (Azar et al., 2001; Charikar et al., 1997; Kleinberg et al., 1999; Kannan et al., 2000). 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. It is interesting that the MQ formulation was discovered in 1998 by Mancoridis et al. (1998) before the concepts of conductance volume and normalized inter-cluster volume were articulated in the theory community. This shows that our concept of coupling and cohesion shows up elsewhere with an almost identical mathematical formulation.

Formally, a clustering $\{\mathscr{C}_1, \ldots, \mathscr{C}_l\}$ 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. (2000) 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:

$$\Phi(S) = \frac{\sum_{u \in S, v \notin S} A_{u,v}}{\min(\mathrm{Vol}(S), \mathrm{Vol}(M \setminus S))}, \tag{12}$$

(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 (10) 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.

Table 1
The systems of our case study

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

Table 2
Bunch versus recursive bisection (RSB)

	Bunch		RSB		Bunch/RSB	
File	Secs	MQ	Secs	MQ	MQ	Secs
Bison	0.12	2.63	1.00	2.57	1.021	0.115
Boxer	0.07	3.10	0.20	3.06	1.013	0.370
CIA	0.15	2.85	2.20	3.29	0.866	0.069
Compiler	0.05	1.34	0.10	1.50	0.894	0.470
Grappa	0.28	12.69	8.00	12.05	1.053	0.035
Ispell	0.10	2.32	1.00	2.14	1.084	0.102
LSLayout	0.07	1.86	0.10	1.79	1.038	0.700
Modulizer	0.10	2.76	0.20	2.69	1.027	0.475
Mini-Tunis	0.09	2.21	0.10	2.15	1.028	0.870
RCS	0.12	2.21	1.00	2.06	1.072	0.116
Swing	9.51	45.21	613.00	40.36	1.120	0.016
Linux kernel	69.26	42.68	19981.00	31.78	1.343	0.003
Proprietary compiler	74.71	45.37	7003.00	36.90	1.230	0.011

4. Evaluation

For small graphs, our algorithm gives excellent performance. For larger graphs, however, the performance is not as good. Computing eigenvalues takes cubic time, and bisecting a graph recursively can, in the worst case, take $n-1$ 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. The fitness function values and execution performance time for recursive bisection are quite close to those produced by Bunch except for the last two systems, which are quite large (roughly 10 times more nodes in the MDG graphs). 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 its current state and sees improvement in the correction. This problem is more prominent in the larger graphs because there is more opportunity for error.

5. Conclusions and 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*, $2 < k \leqslant n$, instead of assigning a variable x_i, $1 \leqslant i \leqslant n$, 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, $1 \leqslant j \leqslant k$, and its ith row corresponds to vector $X_{(i)}$, $1 \leqslant i \leqslant n$. 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 (9) can be generalized to an arbitrary value of k as follows:

$$\begin{aligned} \text{Maximize} \quad & MQ = \sum_{1 \leqslant i \leqslant k} \frac{X_{(i)}^T A X_{(i)}}{X_{(i)}^T D \mathbf{e}} \\ \text{Subject to} \quad & \mathbf{e}^T X^T X \mathbf{e} = n, \\ & X \mathbf{e}_k = e, \\ & X_{i,j} \in \{0,1\}, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant n, \end{aligned} \tag{648}$$

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 $\{S_1, \ldots, S_k\}$.

A second approach to generalizing the bisection algorithm is to use the higher order eigenvectors given by the solution of (11). An argument can be made to show that the first eigenvectors corresponding to first k eigenvectors of the Laplacian matrix are the non-integral solutions that sub-partition the first $k-1$ parts in an optimal way. To make this approach practical we must bound the rounding error for each eigenvector computation to control the quality of the solution. The higher-order eigenvectors must satisfy all of the conditions of the optimization problem in (11).

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

Anquetil, N., 2000. A comparison of graphs of concepts for reverse engineering. In: Proceedings of the International Workshop on Program Comprehension.

Anquetil, N., Lethbridge, T., 1999. Recovering software architecture from the names of source files. In: Proceedings of Working Conference on Reverse Engineering.

Anquetil, N., Fourrier, C., Lethbridge, T., 1999. Experiments with hierarchical clustering algorithms as software remodularization methods. In: Proceedings of Working Conference on Reverse Engineering.

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

Azar, Y., Fiat, A., Karlin, A., McSherry, F., Saia, J., 2001. Spectral analysis of data. In: ACM Symposium on Theory of Computing, pp. 619–626.

Boppana, R., 1988. Eigenvalues and graph bisection: an average case analysis. In: Proceedings of the 28 Annual Symposium on Computer Science, pp. 280–285.

Charikar, M., Chekuri, C., Feder, T., Motwani, R., 1997. Incremental clustering and dynamic information retrieval. In: ACM Symposium on Theory of Computing, pp. 626–635.

Chen, Y., 1995. Reverse engineering. In: Krishnamurthy, B. (Ed.), Practical Reusable UNIX Software. John Wiley & Sons, New York, pp. 177–208, Chapter 6.

Chen, Y., Gansner, E.R., Koutsofios, E., 1997. A C++ data model supporting reachability analysis and dead code detection. In: Proceedings of the European Conference on Software Engineering/ Foundations of Software Engineering.

Choi, S., Scacchi, W., 1990. Extracting and restructuring the design of large systems. In: IEEE Software. pp. 66–71.

Chung, F.R.K., 1997. Spectral graph theory. CBMS Regional Conference Series in Mathematics.

Clark, J., Dolado, J.J., Harman, M., Hierons, R., Jones, B., Lumkin, M., Mitchell, B.S., Mancoridis, S., Rees, K., Roper, M., Shepperd, M., 2003. Reformulating software engineering as a search problem. Journal of IEE Proceedings—Software 150 (3), 161–175.

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

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

Fiedler, M., 1975. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. Czechoslovak Mathematical Journal 25 (100), 619–633.

Garey, M., Johnson, D., 1979. Computers and Intractability. W.H.Freeman.

Goldberg, D., 1989. Genetic Algorithms in Search, Optimization & Machine Learning. Addison Wesley.

Golub, G., Loan, C.V., 1996. Matrix Computations, third ed. Johns Hopkins University Press.

Guattery, S., Miller, G.L., 1998. On the quality of spectral separators. SIAM Journal on Matrix Analysis and Applications 19, 701–719.

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

Holt, R., Malton, A., Dean, T., Cppx: Open source c++ fact extractor. Available from <http://swag.uwaterloo.ca/cppx/>.

Hutchens, D., Basili, R., 1995. System structure analysis: clustering with data bindings. IEEE Transactions on Software Engineering 11, 749–757.

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

Kannan, R., Vempala, S., Vetta, A., 2000. On clusterings: good, bad and spectral. In: Proceedings of 41st Symposium on Foundations of Computer Science, FOCS'00, Redondo Beach, CA.

Kleinberg, J., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A., 1999. The Web as a graph: Measurements, models, and methods. In: Asano, T., Imai, H., Lee, D.T., Nakano, S., Tokuyama, T. (Eds.), Proceedings of 5th Annual International Conference on Computing and Combinatorics, COCOON, vol. 1627. Springer-Verlag.

Korn, J., Chen, Y., Koutsofios, E., 1999. Chava: reverse engineering and tracking of Java Applets. In: Proceedings of the 6th Working Conference on Reverse Engineering, pp. 314–325.

Lindig, C., Snelting, G., 1997. Assessing modular structure of legacy code based on mathematical concept analysis. In: Proceedings of International Conference on Software Engineering.

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

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

McWherter, D., Peabody, M., Regli, W.C., Shokoufandeh, A., 2001a. Transformation invariant shape similarity comparison of models. In: Proceedings of ASME Design Engineering Technical Conferences.

McWherter, D., Peabody, M., Shokoufandeh, A., Regli, W.C., 2001b. Database techniques for archival of solid models. In: Proceedings of 6th ACM/SIGGRAPH Symposium on Solid Modeling and Applications, pp. 78–87.

Mitchell, B., Mancoridis, S., 2001. 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.

Mitchell, B., Mancoridis, S., Traverso, M., 2001. An architecture for distributing the computation of software clustering algorithms. In: Proceedings of the IEEE/IFIP Working International Conference on Software Architecture, WICSA'01.

Mitchell, B.S., Mancoridis, S., 2002. Using heuristic search techniques to extract design abstractions from source code. In: Proceedings of the AAAI Genetic and Evolutionary Computation Conference, GECCO'02.

Mitchell, B.S., Mancoridis, S., 2003. Modeling the search landscape of metaheuristic software clustering algorithms. In: Proceedings of the AAAI Genetic and Evolutionary Computation Conference, GECCO'03.

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

Nemhauser, G.L., Wolsey, L.A., 1988. Integer and Combinatorial Optimization. John Wiley and Sons, New York.

Ng, A., Jordan, M., Weiss, Y., 2001. On spectral clustering: analysis and an algorithm. In: Advances in Neural Information Processing Systems, number 14.

North, S., Koutsofios, E., 1994. Applications of graph visualization. In: Proceedings of Graphics Interface.

Pothen, A., Simon, H.D., Liou, K.-P., 1990. Partitioning sparse matrices with eigenvectors of graphs. SIAM Journal of Matrix Analysis and Applications 11, 430–452.

Russell, S., Norvig, P., 2002. Artificial intelligence: a modern approach. Series in Artificial Intelligence. Prentice Hall, Englewood Cliffs, NJ.

Sarkar, S., Soundararajan, P., 2000. Supervised learning of large perceptual organization: graph spectral partitioning and learning automata. IEEE Transaction on Pattern Analysis and Machine Intelligence 22 (5), 504–525.

Sartipi, K., Kontogiannis, K., 2001). Component clustering based on maximal association. In: Proceedings of Working Conference on Reverse Engineering (WCRE'01).

Sartipi, K., Kontogiannis, K., Mavaddat, F., 2000. Architectural design recovery using data mining techniques. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'00).

Schwanke, R., 1991. An intelligent tool for re-engineering software modularity. In: Proceedings of 13th International Conference on Software Engineering.

Schwanke, R., Hanson, S., 1998. Using neural networks to modularize software. Machine Learning 15, 137–168.

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

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

Shokoufandeh, A., Dickinson, S., Siddiqi, K., Zucker, S.W., 1999. Indexing using a spectral encoding of topological structure. In: Proceedings of Computer Vision and Pattern Recognition, pp. 491–497.

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

Spielman, D., Teng, S.H., 1996. Spectral partitioning works: planar graphs and nite element meshes. In: 37th Annual Symposium on Foundations of Computer Science, (FOCS), pp. 96–105.

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

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

van Deursen, A., Kuipers, T., 1999. Identifying objects using cluster and concept analysis. In: Proceedings of International Conference on Software Engineering.

Wiggerts, T., 1997. Using clustering algorithms in legacy systems remodularization. In: Proceedings of Working Conference on Reverse Engineering.

Zwick, U., 1999. 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, pp. 679–687.

Ali Shokoufandeh is an Assistant Professor in the Department of Computer Science at Drexel University. He received his B.Sc. in computer science from the University of Tehran, and his M.Sc. and Ph.D. degrees in computer Science from Rutgers University in 1996 and 1999, respectively. His research focuses on the design and analysis of algorithms, combinatorial optimization, graph theory, and pattern recognition. He is the recipient of the Center for Discrete Mathematics and Theoretical Computer Science (A National Science Foundation and Technology Center) Graduate Award, in 1998 and 1999. He is a member of SIAM, IEEE, and IEEE Computer Society.

Spiros Mancoridis is an Associate Professor in the Department of Computer Science at Drexel University. He received his B.Sc. in computer science with honors from Acadia University in Canada, and his M.Sc. and Ph.D. degrees in computer science from the University of Toronto in 1992 and 1996, respectively. His research focuses on software engineering and specifically in reverse engineering, software maintenance, and software security. In 1997 he received a CAREER award from the National Science Foundation. He is a member of the IEEE and the ACM.

Trip Denton received a B.S. from Drexel University where he is currently a graduate student. He has worked in the software industry for 20 years and holds two patents. He has taught at Moore College of Art, Philadelphia College of Art, Hussian Art School, Cabrini College, The Franklin Institute, and Drexel University. His primary research interests include clustering algorithms, approximation algorithms, semidefinite programming, and canonical sets.

Matthew Maycock is an undergraduate student at Drexel Univesity majoring in Mathematics. He has served as the corresponding secretary for the Drexel University Mathematics and Computer Science Student Society.