Soft Comput
DOI 10.1007/s00500-007-0218-3

FOCUS

# On the evaluation of the Bunch search-based software modularization algorithm

**Brian S. Mitchell · Spiros Mancoridis**



**Abstract** The first part of this paper describes an automatic reverse engineering process to infer subsystem abstractions that are useful for a variety of software maintenance activities. This process is based on clustering the graph representing the modules and module-level dependencies found in the source code into abstract structures not in the source code called subsystems. The clustering process uses evolutionary algorithms to search through the enormous set of possible graph partitions, and is guided by a fitness function designed to measure the quality of individual graph partitions. The second part of this paper focuses on evaluating the results produced by our clustering technique. Our previous research has shown through both qualitative and quantitative studies that our clustering technique produces good results quickly and consistently. In this part of the paper we study the underlying structure of the search space of several open source systems. We also report on some interesting findings our analysis uncovered by comparing random graphs to graphs representing real software systems.

**Keywords** Reverse engineering · Software clustering · Search algorithms

B. S. Mitchell (✉) · S. Mancoridis
Department of Computer Science,
Software Engineering Research Group at Drexel University,
Philadelphia, USA
e-mail: bmitchel@cs.drexel.edu
URL: http://www.cs.drexel.edu/~bmitchel

S. Mancoridis
e-mail: spiros@cs.drexel.edu
URL: http://www.cs.drexel.edu/~spiros

## 1 Introduction

Modern software systems tend to be large and complex, thus appropriate abstractions of their structure are created to simplify program maintenance activities, which often go on for years after the initial system is released. Ideally, these abstractions are documented, however, such documentation is often out-of-date or non-existent. Since source code is often the only accurate documentation available to software developers and maintainers, the reverse engineering community has been developing tools to recover abstract views of the software architecture directly from source code.

According to Shaw and Garlan (1996), the software architecture of a system consists of a description of the system elements, interactions between the system elements, patterns that guide the construction of the system elements, and constraints on the relationships between the system elements. For smaller systems the elements and relations may be modeled using source code entities such as procedures, classes, method invocation, inheritance, and so on. However, for larger systems, the desired entities may be abstract (i.e., high-level), and modeled using architectural components such as subsystems and subsystem relations Lakhotia (1997).

Reverse engineering architectural components from the source code of an existing system is a challenging task. Module boundaries are defined by programmers by grouping sets of related functions/methods, macros, and properties/variables into source code files or classes. The relations between the modules can be easily derived by extracting the module-level relations from the source code. As systems grow, engineers often aggregate these modules into higher–level entities called subsystems. For very large systems, the hundreds subsystems themselves might be collected into other subsystems, resulting in a subsystem hierarchy. The goal is to define subsystems that contain entities that

are related based on a specified set of criteria. Since the subsystems are not specified in the source code, they must be inferred using other methods. Some of the criteria investigated to derive subsystems include using source code component similarity Hutchens and Basili (1985), Schwanke (1991); Choi and Scacchi (1999); Müller et al. (1992), concept analysis Lindig and Snelting (1997); van Deursen and Kuipers (1999), and implementation information such as module, directory, and/or package names Anquetil et al. (1999).

Our research addresses the above goal by using a combinatorial optimization approach Affenzeller and Mayrhofer (2002) to cluster the module-level structure represented by the source code into architectural-level subsystems. We accomplish this objective by modeling the software clustering process as a search problem. Our technique uses source code analysis tools to first create a graph of the system structure, where the nodes are modules (e.g., Java classes, C/C++ files), and the edges are a binary relations that represent the module-level dependencies (e.g., method calls, inheritance). The clustering approach evaluates the quality of a graph partition, and uses heuristics to navigate through the search space Clark et al. (2003) of all possible graph partitions. Since finding the optimal graph partition is NP-hard Garey and Johnson (1979) (Discussed in Sect. 2), we have developed several promising heuristic approaches to solving this problem, and outline our findings in the remaining sections of this paper.

Our search-based clustering approach has been implemented in a tool called Bunch. Bunch starts by generating a random solution from the search space, and then improves it using evolutionary computation algorithms. The search is guided by an objective function called modularization quality (MQ). We define the quality of a clustering result as a measure of MQ, which is discussed in Sect. 2.2. Bunch rarely produces the exact same result on repeated runs of the tool. However, its results are very similar – this similarity can be validated by inspection on small systems and by using similarity measurements Mitchell and Mancoridis (2001) that have been designed to compare software clustering results on larger systems.

The above observations intrigued us to answer why Bunch produces similar and high-quality results so consistently, especially since the probability of finding a good solution by random selection is extremely small. The key to our approach is to model the search landscape of each system undergoing clustering, and then to analyze how Bunch produces results within this landscape.

The first part of this paper describes the search-based software clustering approach that has been implemented in the Bunch tool. The second part defines how the search landscape is modeled and how it is used for evaluation purposes.

## 2 Software clustering with Bunch

The goal of the software clustering process is to partition a graph of the source-level entities and relations into a set of clusters (i.e., subsystems). Since graph partitioning is known to be NP-hard Garey and Johnson (1979), obtaining a good solution by random selection or exhaustive exploration of the search space is unlikely. Bunch overcomes this problem by using evolutionary search techniques based on hill climbing and genetic algorithms Mitchell (1997); Doval et al. (1999).

In the preparation phase of the clustering process, source code analysis tools Chen et al. (1997); Chen (1995) are used to parse the code and build a repository of information about the entities and relations in the system. A series of scripts are then executed to query the repository and create the module dependency graph (MDG). The MDG is a graph where the source code components are modeled as nodes, and the source code dependencies are modeled as edges. There are many different ways to create a MDG based on the desired granularity of the source code components (e.g., classes/modules versus functions/methods) and their programming language specific dependencies. Since our goal is to perform architectural analysis on software systems, we will use MDGs where the nodes are modules that represent source code files for procedural languages such as C, and classes for object oriented programming languages such as C++ and Java. A directed edge in the MDG $(u, v)$ represents the set of resources that module $u$ uses in module $v$. The edge will be weighted to reflect the total number of dependencies (all function calls, inheritance relations, imports, etc.) that exist between modules $u$ and $v$.

Once the MDG is created, Bunch generates a random partition of the MDG and evaluates the "quality" of this partition using a fitness function that is called modularization quality (MQ) Mitchell (2002). The MQ function is designed to reward cohesive clusters and penalizes excessive inter-cluster coupling. MQ increases as the *intraedges* (i.e., internal edges contained within a cluster) increase and the *interedges* (i.e., external edges that cross cluster boundaries) decrease.

Given that the fitness of an individual partition can be measured, search-based algorithms are used in the clustering phase in an attempt to improve the MQ of the randomly generated partition.

Once Bunch's search algorithms converge, a result can be viewed as XML Holt et al. (2000); GXL or using the dotty Gansner et al. (1993) graph visualization tool.

### 2.1 A small software clustering example

This section presents an example illustrating how Bunch can be used to cluster the JUnit system. JUnit is an open-source unit testing tool for Java, and can be obtained online from http://www.junit.org. JUnit contains four main packages: the

framework itself, the user interface, a test execution engine, and various extensions for integrating with databases and J2EE. The JUnit system contains 32 classes and has 55 dependencies between the classes.

For the purpose of this example, we limit our focus to the framework package. This package contains seven classes, and nine inter-class relations. All remaining dependencies to and from the other packages have been collapsed into a single relation to simplify the visualization.

Figure 1 depicts the process that Bunch used to partition the MDG of JUnit into subsystems. In the left corner of this figure we show the MDG of JUnit. Step 2 illustrates the random partition generated by Bunch as a starting point in the search for a solution. Since the probability of generating a good random partition is small, we expect the random partition to be a poor solution. This intuition is validated by inspecting the random partition, which contains two singleton clusters and a disproportionably large number of interedges (i.e., edges that cross subsystem boundaries).

The random partition shown in Step 2 is converted by Bunch into the final result shown in Step 3 of Fig. 1. The solution proposed by Bunch is consistent with the expected design of a unit testing framework as it grouped the test case modules, the test result modules, and the runtime validation modules into clusters.

The overall result shown in Step 3 of Fig. 1 is a good result, but Bunch's search algorithms are not guaranteed to produce exactly the same solution for every run. Thus we would like to gain confidence in the *stability* of Bunch's clustering algorithms by analyzing the search landscape (Sect. 4) associated with each MDG.

### 2.2 Measuring modularization quality

Bunch supports several functions Mitchell (2002) for evaluating the quality of a modularization. Each function was designed to balance the tradeoff between the coupling and cohesion of the individual clusters in the partitioned MDG. The goal of MQ is to reward cohesive clusters (subsystems) by minimizing inter-cluster coupling. Since the modularization quality (MQ) function is called many times during the clustering process, it needs to be efficient so that Bunch can be applied to large systems.

Formally, the MQ measurement[1] for an MDG partitioned into $k$ clusters is calculated by summing the cluster factor (CF) for each cluster of the partitioned MDG. The cluster factor, $CF_i$, for cluster $i$ ($1 \leq i \leq k$) 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 the external edges (edges that exit or enter the cluster). We split the weight of the external edges in half in order to apply an equal penalty to both clusters that are are connected by an external edge.

We refer to the internal edges of a cluster as intra-edges ($\mu_i$), and the edges between two distinct clusters $i$ and $j$ as inter-edges ($\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ respectively). If edge weights[2] are not provided by the MDG, we assume that each edge has a weight of 1. Also, note that $\varepsilon_{i,j} = 0$ and $\varepsilon_{j,i} = 0$ when $i = j$ because these are intra-edges ($\mu$). Below, we define the MQ calculation:

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \text{otherwise} \end{cases}$$

Figure 2 illustrates an example MQ calculation for an MDG consisting of eight modules that are partitioned into three subsystems. The value of MQ is approximately 1.924, which is the result of summing the Cluster Factor for each of the three subsystems. Each CF measurement is between 0 (no intra-edges in the subsystem) and 1 (no edges to or from other subsystems). For example, the CF for Subsystem 2 in Fig. 2 is 0.4 because there is 1 intra-edge, and three inter-edges. Applying these values to the expression for CF results in $CF_2 = 2/5$.

The complexity of calculating MQ is $O(|E|)$, where $|E|$ is proportional to the number of edges in the MDG Mitchell (2002).

In practice MQ can be calculated more efficiently by exploiting domain knowledge from the clustering algorithm. This optimization is based on the observation that the MQ objective function is calculated by summing the Cluster Factors. Furthermore, any decomposition of an MDG can be transformed into another decomposition of an MDG by executing a series of move operations. As an example consider Fig. 3, "Decomposition 1" can be transformed into "Decomposition 2" by executing the following move operations: move M4 to subsystem 1, then move M5 to subsystem 1.

Given the above example, we notice that each move operation impacts exactly two clusters. When a node is moved from one cluster to another, its source cluster and destination cluster are changed, but all other clusters remain the same with respect to the total weight of the $\mu$ and $\varepsilon$ edges. Revisiting the design of the MQ measurement, recall that each cluster has an associated Cluster Factor value. Thus, if we maintain the various cluster factor values, and the MQ value, we can update MQ incrementally by only calculating

[1] We have investigated several MQ functions, which are documented elsewhere Mitchell (2002).

[2] Note that MDGs with edge weights allow the clustering algorithm to take into account the strength of the coupling between modules. For example, a pair of modules that call each other 10 times should be more likely to be placed into the same cluster than a pair of modules that only interact with each other once.

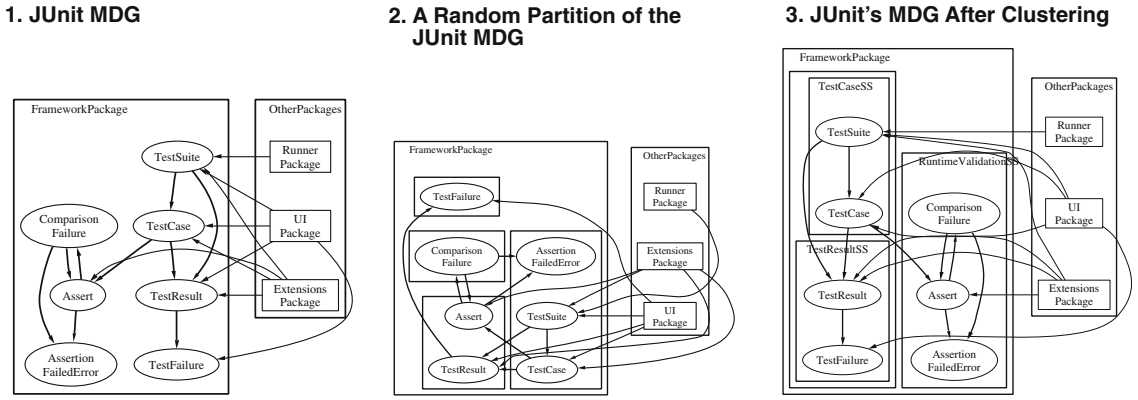


**Fig. 1** JUnit example

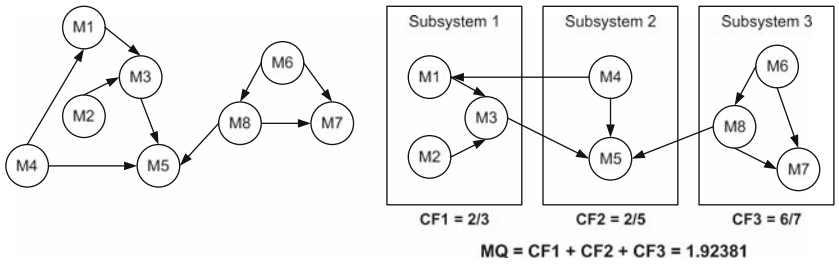


**Fig. 2** TurboMQ calculation example

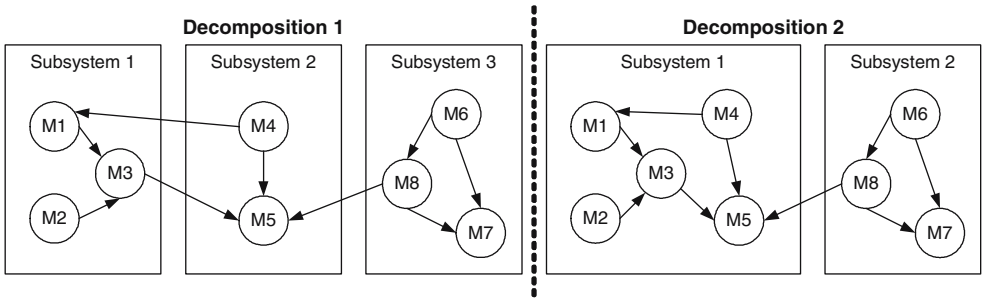


**Fig. 3** Two similar partitions of a MDG

the two cluster factors that change (i.e., the source and destination Cluster Factors). This optimization effectively allows us to update the value of MQ incrementally in constant $O(1)$ time Mitchell (2002).

## 3 Clustering algorithms

This section describes a family of hill-climbing search algorithms that have been implemented in Bunch. All of Bunch's hill-climbing clustering algorithms start with a random partition of the MDG. Modules from this partition are then systematically rearranged in an attempt to find an improved partition with a higher MQ. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. This hill-climbing approach eventually converges when no additional partitions can be found with a higher MQ.

As with traditional hill-climbing algorithms, each randomly-generated initial partition of the MDG eventually converges to a local maximum. Unfortunately, not every initial partition of the MDG produces an acceptable sub-optimal solution. We describe two ways that we deal with this problem in Sect. 3.1.

Bunch's hill-climbing algorithms move modules between the clusters of a partition in an attempt to improve MQ. This task is accomplished by generating a set of neighboring partitions (NP).

We define a partition NP to be a neighbor of a partition $P$ if NP is exactly the same as $P$ except that a single element

of a cluster in partition $P$ is in a different cluster in partition NP. If partition $P$ contains $n$ nodes and $k$ clusters, the total number of neighbors is bounded by $(n \cdot k)$.

The hill-climbing algorithm also uses a calibration threshold $\eta$ ($0\% \leq \eta \leq 100\%$) to calculate the minimum number of neighbors that must be considered during each iteration of the hill-climbing process. A low value for $\eta$ results in the algorithm taking more small steps prior to converging, and a high value for $\eta$ results in the algorithm taking fewer large steps prior to converging. Our experience has shown that examining many neighbors during each iteration (i.e., using a large threshold such as $\eta \geq 75\%$) increases the time the algorithm needs to converge to a solution.

Prior to introducing the adjustable threshold $\eta$, Bunch only considered the two extremes of the threshold—use the first neighboring partition found with a better MQ as the basis for the next iteration ($\eta = 0\%$), or examine all neighboring partitions and pick the one with the largest MQ as the basis for the next iteration ($\eta = 100\%$). In some of our other papers Mancoridis et al. (1998, 1999); Mitchell et al. (2001) we called the first algorithm next ascent hill-climbing (NAHC), and the second algorithm steepest ascent hill-climbing (SAHC).

Migrating to a single hill-climbing algorithm, instead of having two distinct implementations (i.e., NAHC, SAHC), has given our clustering algorithm more flexibility by allowing users to configure the hill-climbing behavior. Thus, by adjusting the neighboring threshold $\eta$, the user can configure the generic hill-climbing algorithm to behave like NAHC, SAHC, or any hill-climbing algorithm in between.

### 3.1 Simulated annealing (SA)

The basic hill-climbing clustering algorithm attempts to take a partition and improve it by examining a percentage of its neighboring partitions. A well-known problem of hill-climbing algorithms is that certain initial starting points may converge to poor solutions (i.e., local optima). To address this problem, the Bunch hill-climbing algorithm does not rely on a single random starting point, but instead uses a collection of random starting points. The result with the largest MQ from the initial population of starting points is selected as the solution.

Another way to overcome the above described problem is to use simulated annealing Kirkpatrick et al. (1983). Simulated Annealing is based on modeling the cooling processes of metals, and the way liquids freeze and crystallize. When applied to optimization problems, simulated annealing enables the search algorithm to accept, with some probability, a worse variation as the new solution of the current iteration. As the computation proceeds, the probability diminishes. The slower the *cooling schedule*, or rate of decrease, the more likely the algorithm is to find an optimal or near-optimal solution. Simulated Annealing techniques typically represent the cooling schedule with a *cooling function* that reduces the probability of accepting a worse variation as the optimization algorithm runs.

Bunch provides an extension facility to enable users to define and integrate a custom cooling function into the clustering process. Bunch includes an example cooling function that has been shown to work well Mitchell and Mancoridis (2002), and can be used as a reference implementation for other researchers interested in developing their own cooling functions.

The cooling function distributed with Bunch is called *SimpleSA* and is designed to determine the probability of accepting a worse, instead of a better, partition during each iteration of the clustering algorithm. The idea is that by accepting a worse neighbor, occasionally the algorithm will "jump" to a new area in the search space. The SimpleSA cooling function is designed to respect the properties of the Simulated Annealing cooling schedule, namely: (a) decrease the probability of accepting a worse move over time, and (b) increase the probability of accepting a worse move if the rate of improvement is small. The SimpleSA cooling function shown below returns the probability $P(A)$ of accepting a non-improving neighbor during the hill-climbing process:

$$P(A) = \begin{cases} 0 & \triangle MQ \geq 0 \\ e^{\frac{\triangle MQ}{T}} & \triangle MQ < 0 \end{cases}$$

$$T(k+1) = \alpha \cdot T(k)$$

Each time a probability $P(A)$ is calculated, $T(k)$ is reduced. The initial value of $T$ (i.e., $T(0)$) and the rate of reduction constant $\alpha$ are established by the user. Furthermore, $\triangle MQ$ must be negative, which means that the MQ value has decreased. Once a probability is calculated, a uniform random number between 0 and 1 is chosen. If this random number is less than the probability $P(A)$, the partition is accepted as the basis for the current iteration of the clustering process.

#### *3.1.1 Building block partitions*

When the set of neighbors for the current partition is evaluated, each node in the MDG is moved not only to all of the other clusters in the partition, but also to a new singleton cluster. Because MQ tries to minimize coupling, it is rare that the introduction of a singleton cluster results in an improved MQ value. This characteristic of MQ introduces an undesired side-effect, namely, the hill-climbing algorithm often finds it unfavorable to create new clusters.

To address this problem, we relaxed our-first definition of a neighboring partition slightly. During the hill-climbing process, in addition to examining neighboring partitions, we also examine a new partition type, called a building block

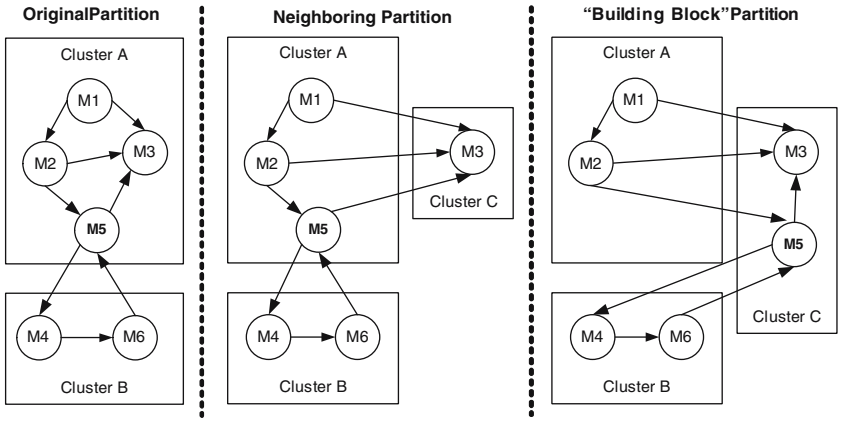


**Fig. 4** Example of a building block partition

partition. We define partition BBP to be a building block partition of $P$ if BBP is exactly the same as $P$ except that: (a) BBP contains one additional cluster, and (b) the new cluster contains exactly two nodes from the MDG. We illustrate the difference between a neighboring and a building block partition in Fig. 4.

Technically, there are $O(|V|^2)$ building blocks if we pair each module with all of the other modules when creating a building block. However, when a node is moved into a singleton cluster, we only create building blocks with other nodes that are connected to this node by an edge (relation) in the MDG. This optimization reduces the total number of building blocks from $O(|V|^2)$ to $O(|E|)$.

It should be noted that there are many ways to define building block partitions. Above we suggest creating a new partition containing exactly two nodes as long as the nodes in this new partition are connected by a relation in the MDG. This approach can be generalized to create larger building block partitions consisting of three or more nodes from the MDG. In practice we have found that creating building blocks of size two works well for stimulating Bunch to create new clusters when needed.

## 4 Case study

If Bunch is to be useful to software maintainers, its users must have confidence that the results it produces are consistent. In Sect. 3 we described that the Bunch hill-climbing algorithm starts by generating a random solution from the search space, and then improves it using search algorithms. As expected, Bunch rarely produces the exact same result on repeated runs of the tool since the search space is enormous. However, its results are very similar—this similarity can be validated by inspection on small systems, by using similarity measurements Mitchell and Mancoridis (2001), or by using the subjective feedback provided by the system developers. Since this outcome is not intuitive, we were intrigued to answer why Bunch produces similar results so consistently, and how the commonality in the clustering results can be used to evaluate the effectiveness of Bunch's clustering algorithms.

Since Bunch has several user-configurable parameters, we also wanted to investigate the impact of altering these parameters on the clustering results so that we can provide configuration guidance to Bunch users. The remainder of this section presents a substantial case study to investigate the questions proposed above.

### 4.1 The impact of altering Bunch's clustering parameters

Table 1 describes the five systems that we used in the first part of this case study. We selected these systems because they vary in size. Our basic test involved clustering each system a total of 1,050 times. Each system was clustered a total of 50 times with the adjustable clustering threshold (discussed in Sect. 3) $\eta$ set to 0%. The clustering threshold $\eta$ was then incremented by 5% and another 50 clustering runs was performed. This process was repeated a total of 21 times until $\eta$ reached 100%.

We then repeated the above test for each of the systems described in Table 1, this time using the simulated annealing

**Table 1** Systems examined in the configuration case study

| System Name | System description & MDG size |
|---|---|
| compiler | A small compiler. |
| | MDG: modules = 13, relations = 32 |
| ispell | An open source spell checker. |
| | MDG: modules = 24, relations = 103 |
| rcs | An open source version control system. |
| | MDG: modules = 34, relations = 163 |
| dot | A graph drawing tool Gansner et al. (1993). |
| | MDG: modules = 42, relations = 256 |
| swing | The Java user interface class library. |
| | MDG: modules = 413, relations = 1,513 |

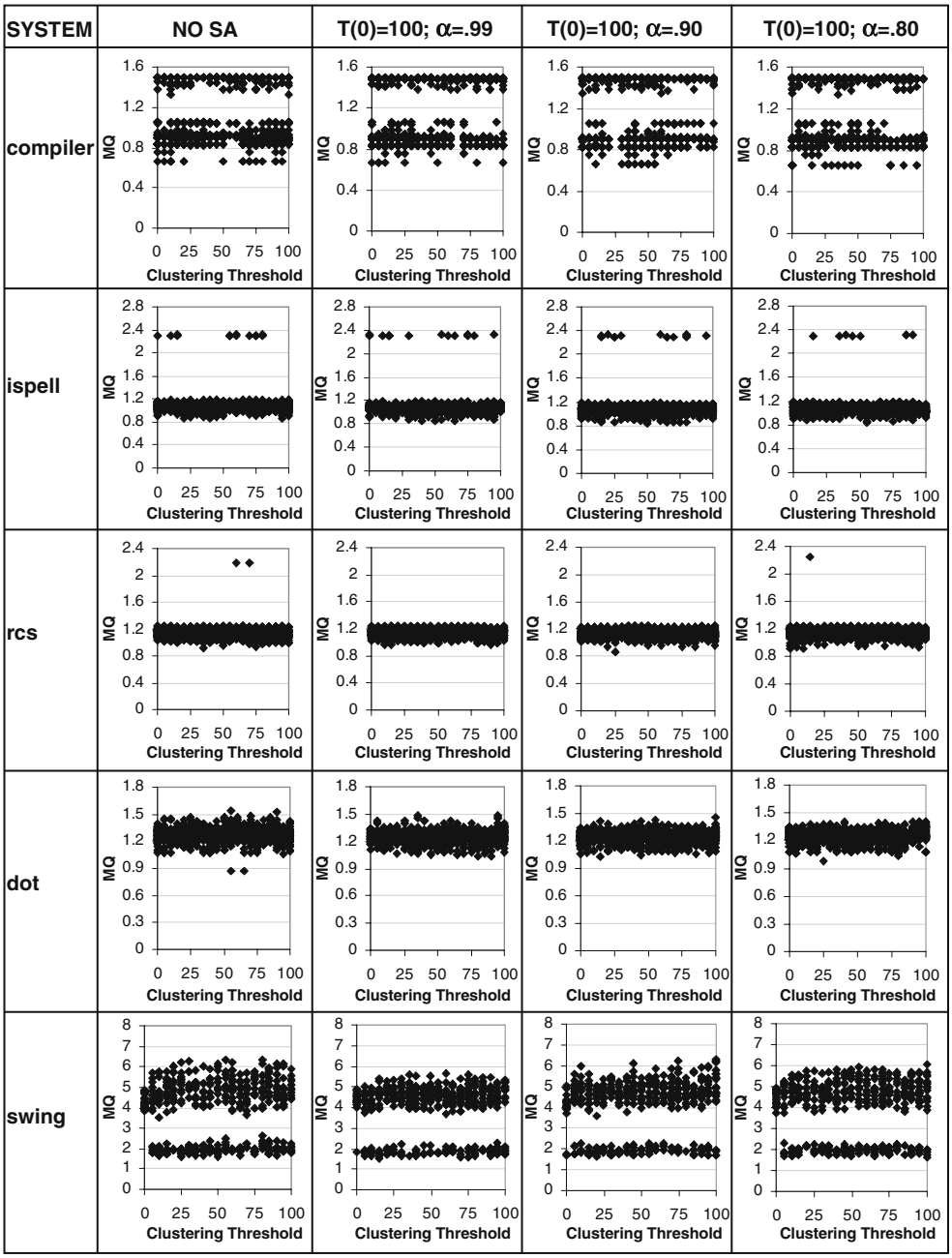


**Fig. 5** Case study results—MQ versus $\eta$ scatter plot

(SA) feature. Each of these tests altered the parameters used to initialize the *cooling function* that was described in Sect. 3.1. For these tests we held the initialization value for the starting temperature constant, $T(0) = 100$, and varied the cooling rate as follows: $\alpha = \{0.99, 0.9, 0.8\}$.[3] The results from the case study are illustrated in Figs. 5 and 6. The first column (NO SA) represents the test where simulated annealing was disabled. The subsequent columns represent the impact of altering the cooling rate parameter $\alpha$ in our simulated annealing implementation.

The following observations were made based on the data collected in the case study:

- Figure 5 shows that although increasing $\eta$ increased the overall runtime and number of MQ evaluations, altering $\eta$ did not appear to have an observable impact on the overall quality of the clustering results. The data in Fig. 5 also

[3] We only used larger values of $\alpha$ ($0 < \alpha < 1$) since smaller values would effectively disable the SA feature after a small number of iterations.

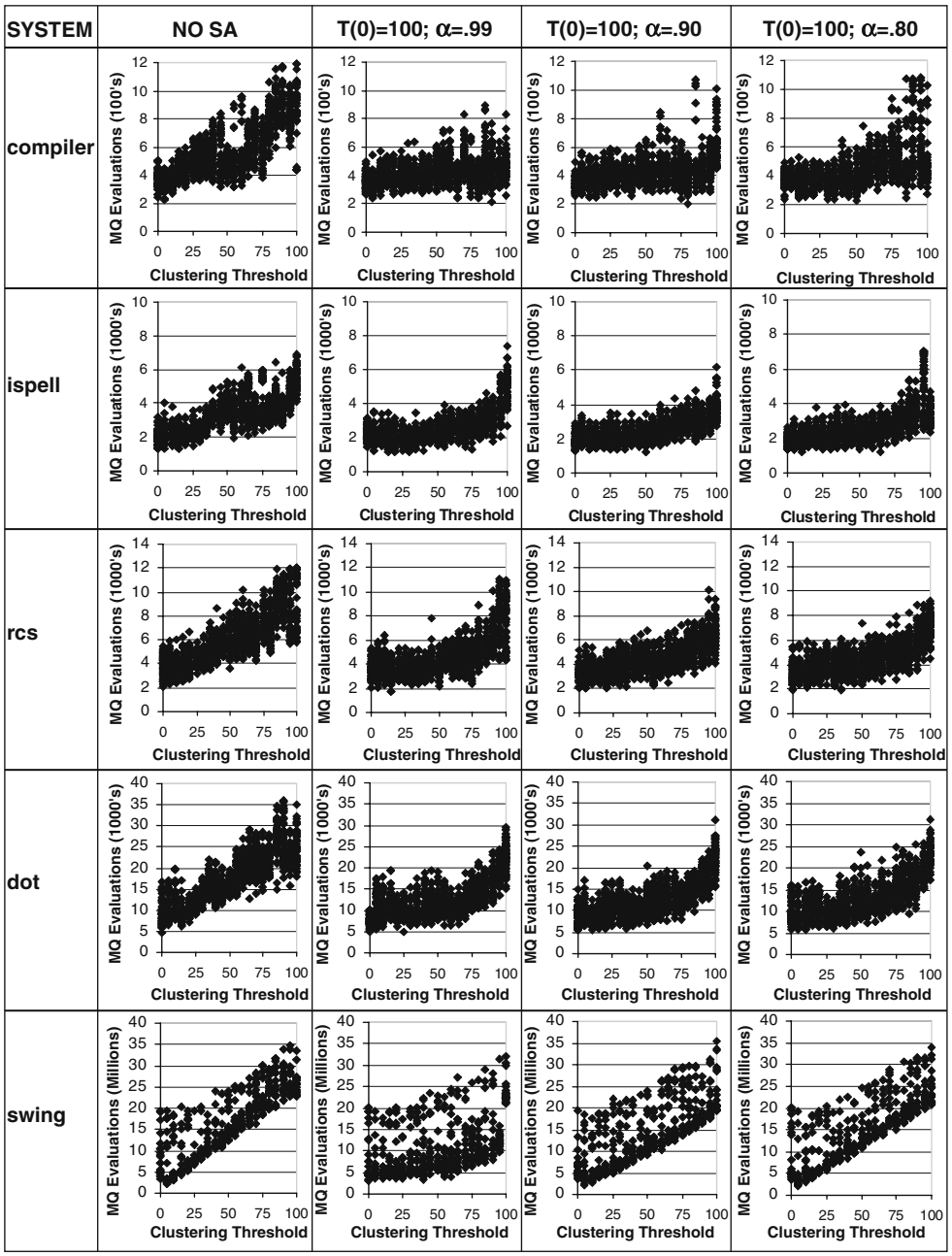


**Fig. 6** Case study results—MQ evaluations versus $\eta$ scatter plot

shows that our simulated annealing implementation did not improve MQ. Although this outcome was not uprising, the simulated annealing algorithm did appear to help reduce the total runtime needed to cluster each of the systems in this case study. We do not really understand why this happened so consistently across all of the systems we examined (see Table 2), and would like to investigate this outcome further in future research.

Figure 6 shows the number of MQ evaluations performed for each of the systems in the case study. Since the overall runtime is directly related to the number of MQ

**Table 2** Reduced percentage of MQ evaluations associated with using simulated annealing

| System | Simulated annealing parameters | | |
|---|---|---|---|
| | $T(0) = 100$ $\alpha = 0.99$ (%) | $T(0) = 100$ $\alpha = 0.90$ (%) | $T(0) = 100$ $\alpha = 0.80$ (%) |
| Compiler | 27 | 26 | 23 |
| ispell | 22 | 25 | 21 |
| rcs | 25 | 27 | 26 |
| Dot | 28 | 29 | 25 |
| Swing | 32 | 15 | 10 |

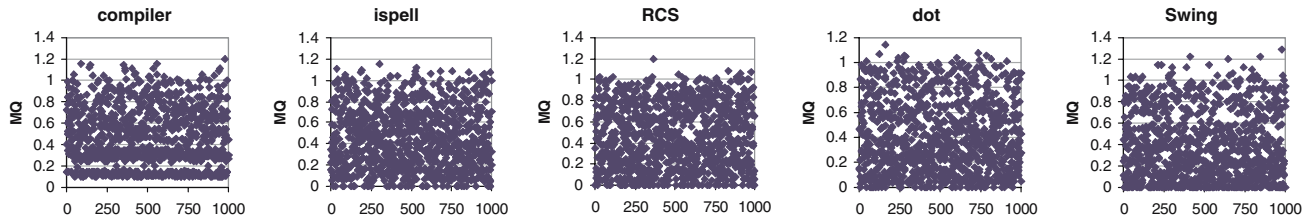


**Fig. 7** Case study results—random partition scatter plot

evaluations, it appears that the use of our SA cooling technique is a promising way to reduce the clustering time. Table 2 compares the average number of MQ evaluations executed in each SA test to the average number of MQ evaluations executed in the non-SA test. For example using $T(0) = 100$ and $\alpha = 0.99$ reduced the number of MQ evaluations needed to cluster the swing class library by an average of 32%.

- Figure 5 indicates that the hill-climbing algorithm converged to a consistent solution for the ispell, dot and `rcs` systems.
- Figure 5 shows that the hill-climbing algorithm converged to one of two families of related solutions for the compiler and swing systems. For the compiler system, 53.7% of the results were found in the range $0.6 \leq \text{MQ} \leq 1.0$, and 46.3% of the results were found in the range $1.3 \leq \text{MQ} \leq 1.5$. For the swing system, 27.3% of the results were found in the range $1.5 \leq \text{MQ} \leq 2.5$, and 72.7% of the results were found in the range $3.75 \leq \text{MQ} \leq 6.3$.
- Figure 5 shows two interesting results for the ispell and rcs systems. For the ispell system, 34 out of 4,200 samples (0.8%) were found to have an MQ around 2.3, while all of the others samples had an MQ value in the range $0.8 \leq \text{MQ} \leq 1.2$. For the rcs system, 3 out of 4,200 samples (0.07%) were found to have an MQ around 2.2, while all of the other samples had MQ values concentrated in the range of $1.0 \leq \text{MQ} \leq 1.25$. This outcome highlights that some rare partitions of an MDG may be discovered if enough runs of our hill-climbing algorithm are executed.
- As expected, the clustering threshold $\eta$ had a direct and consistent impact on the clustering runtime, and the number of MQ evaluations. As $\eta$ increased so did the overall runtime and the number of MQ evaluations. This behavior is illustrated consistently in Fig. 6.
- Figure 7 illustrates 1,000 random partitions for each system examined in this case study. The random partitions have low MQ values when compared to the clustering results shown in Fig. 5. This result provides some confidence that our clustering algorithms produce better results than examining many random partitions, and that the probability of finding a good partition by means of random selection is small.
- As $\alpha$ increased, so did the number of simulated annealing (non-improving) partitions that were incorporated into the clustering process. In Fig. 8 we show the number of SA partitions integrated into the clustering process for the swing class library. As expected, the number of partitions decreased as $\alpha$ decreased. Although we only show this result for swing, all of the systems examined in this case study exhibited this expected behavior.

## 4.2 Modeling the search landscape

Table 3 describes the systems used in the second part of our case study, which consist of seven open source systems and six randomly generated MDGs. The goal of this part of the case study is to investigate the stability of the Bunch's software clustering algorithms. Since Bunch's search algorithms are not guaranteed to produce exactly the same solution for every run, we wanted to gain insight into the *search landscape* to investigate why the results produced by Bunch are so consistent. By modeling the search landscape through a variety of different views we also discovered several aspects of Bunch's clustering results that would not have been obvious by examining an individual clustering result.

Some explanation is necessary to describe how the random MDGs were generated. Each random MDG used in this study consists of 100 modules, and has a name that ends


**Fig. 8** Case study results—simulated annealing partitions used for clustering swing

**Table 3** Application descriptions

| Application name | Modules in MDG | Relations in MDG | Application description |
|---|---|---|---|
| Telnet | 28 | 81 | Terminal emulator |
| PHP | 62 | 191 | Internet scripting language |
| Bash | 92 | 901 | Unix terminal environment |
| Lynx | 148 | 1,745 | Text-based HTML browser |
| Bunch | 220 | 764 | Software clustering tool |
| Swing | 413 | 1,513 | Standard Java user interface framework |
| Kerberos5 | 558 | 3,793 | Security services infrastructure |
| Rnd5 | 100 | 247 | Random graph with 5% edge density |
| Rnd50 | 100 | 2,475 | Random graph with 50% edge density |
| Rnd75 | 100 | 3,712 | Random graph with 75% edge density |
| Bip5 | 100 | 247 | Random bipartite graph with 5% edge density |
| Bip50 | 100 | 2,475 | Random bipartite graph with 50% edge density |
| Bip75 | 100 | 3,712 | Random bipartite graph with 75% edge density |

with a number. This number is the edge density of the MDG. For example a 5% edge density means that the graph will have an edge count of $0.05 \times (n(n-1)/2)$, which is 5% of the maximum number of edges that can exist in a graph of $n$ modules. Since the MDG for most software systems is sparse (the average density for the real systems in Table 3 is 17%), we wanted to investigate how Bunch behaved by comparing the clustering results to sparse random MDG's. We also wanted to investigate how sparse random MDG's compared to dense random MDG's, as well as how dense random MDG's compared to the MDG's of real systems.

Each system in Table 3 was clustered 100 times using Bunch's default settings. It should be noted that although the individual clustering runs are independent, the *search landscapes* plotted in Figs. 9 and 10 are ordered by increasing MQ. This sorting highlights some results that would not be obvious otherwise. The study presented in the remaining part of this section examines two different types of search landscapes—the structural landscape and the similarity landscape.

#### *4.2.1 Studying the stability of the hill-climbing algorithm*

This section describes the structural search landscape, which highlights similarities and differences from a collection of clustering results by identifying trends in the structure of graph partitions. Thus, the goal of the structural search landscape is to validate the following hypotheses:

- We expect to see a relationship between MQ and the number of clusters. Both MQ and the number of clusters in the partitioned MDG should not vary widely across the clustering runs.
- We expect a good result to produce a high percentage of intra-edges (edges that start and end in the same cluster) consistently.
- We expect repeated clustering runs to produce similar MQ results.
- We expect that the number of clusters remains relatively consistent across multiple clustering runs.

Figure 9 shows the structural search landscape of the open source systems, and Fig. 10 illustrates the structural search landscape of the random graphs used in this study.

The overall results produced by Bunch appear to have many consistent properties. This observation can be supported by examining the results shown in Figs. 9 and 10:

- By examining the views that compare the cluster counts (i.e., the number of clusters in the result) to the MQ values (far left) we notice that Bunch tends to converge to one or two "basins of attraction" for all of the systems studied. Also, for the real software systems, these attraction areas appear to be tightly packed. For example, the PHP system has a point of concentration were all of the clustering results are packed between MQ values of 5.79 and 6.11. The number of clusters in the results are also tightly packed ranging from a minimum of 14 to a maximum of 17 clusters. An interesting observation can be made when examining the random systems with a higher edge density (i.e., RND50, RND75, BIP50, BIP75). Although these systems converged to a consistent MQ, the number of clusters varied significantly over all of the clustering runs. We observe these wide ranges in the number of clusters, with little change in MQ for most of the random

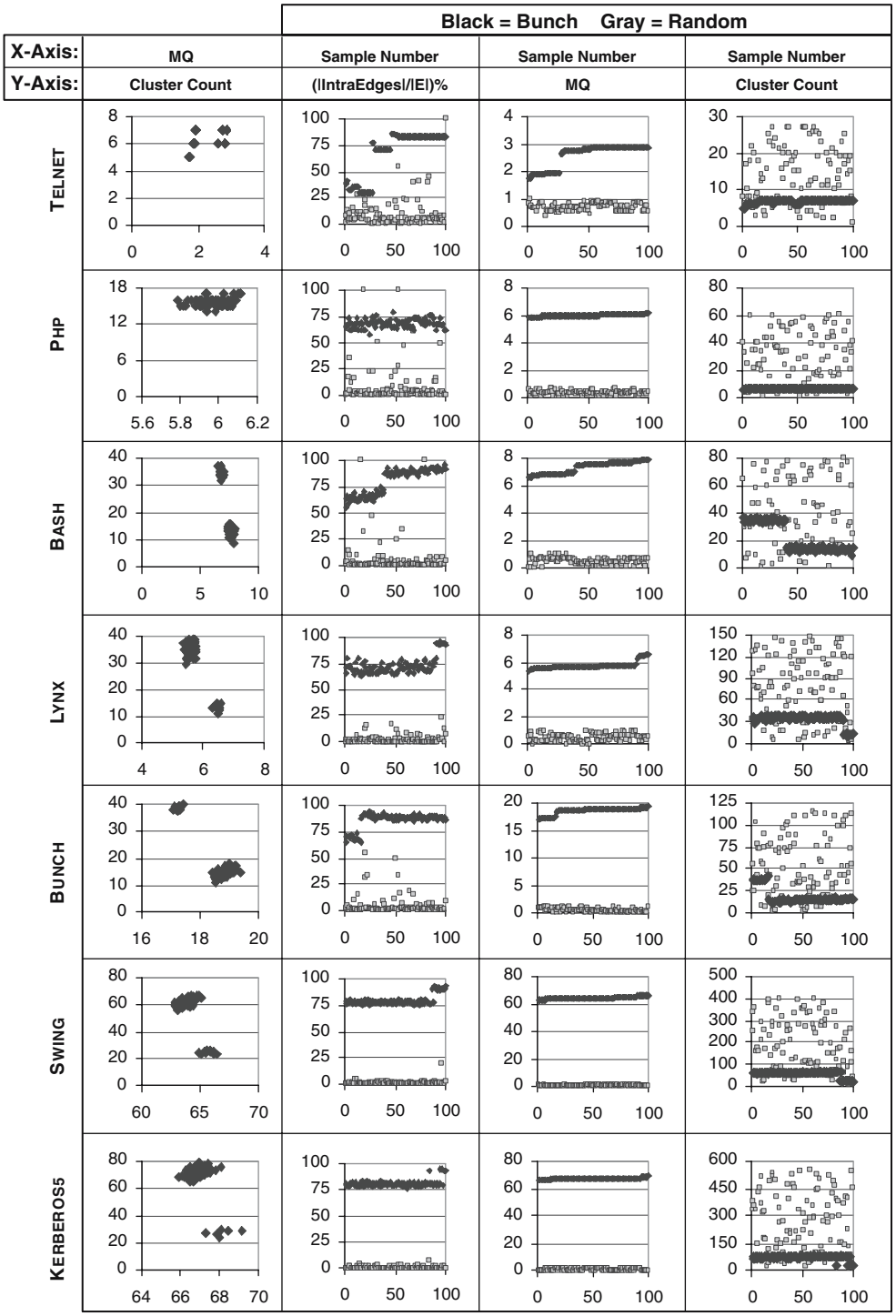


**Fig. 9** The structural search landscape for the open source systems

systems, but do not see this characteristic in the real systems.

- The view that shows the percentage of intra-edges in the clustering results (second column from the left) indicates that Bunch produces consistent solutions that have a relatively large percentage of intra-edges. To determine the average number of intra-edges, the number of edges that were internal to clusters in the MDG $|IntraEdges|$ was divided by the total number of edges in the graph $|E|$. Also, since the 100 samples were sorted by MQ, we observe that the intra-edge percentage increases as the MQ values increase. This result was expected as the goal

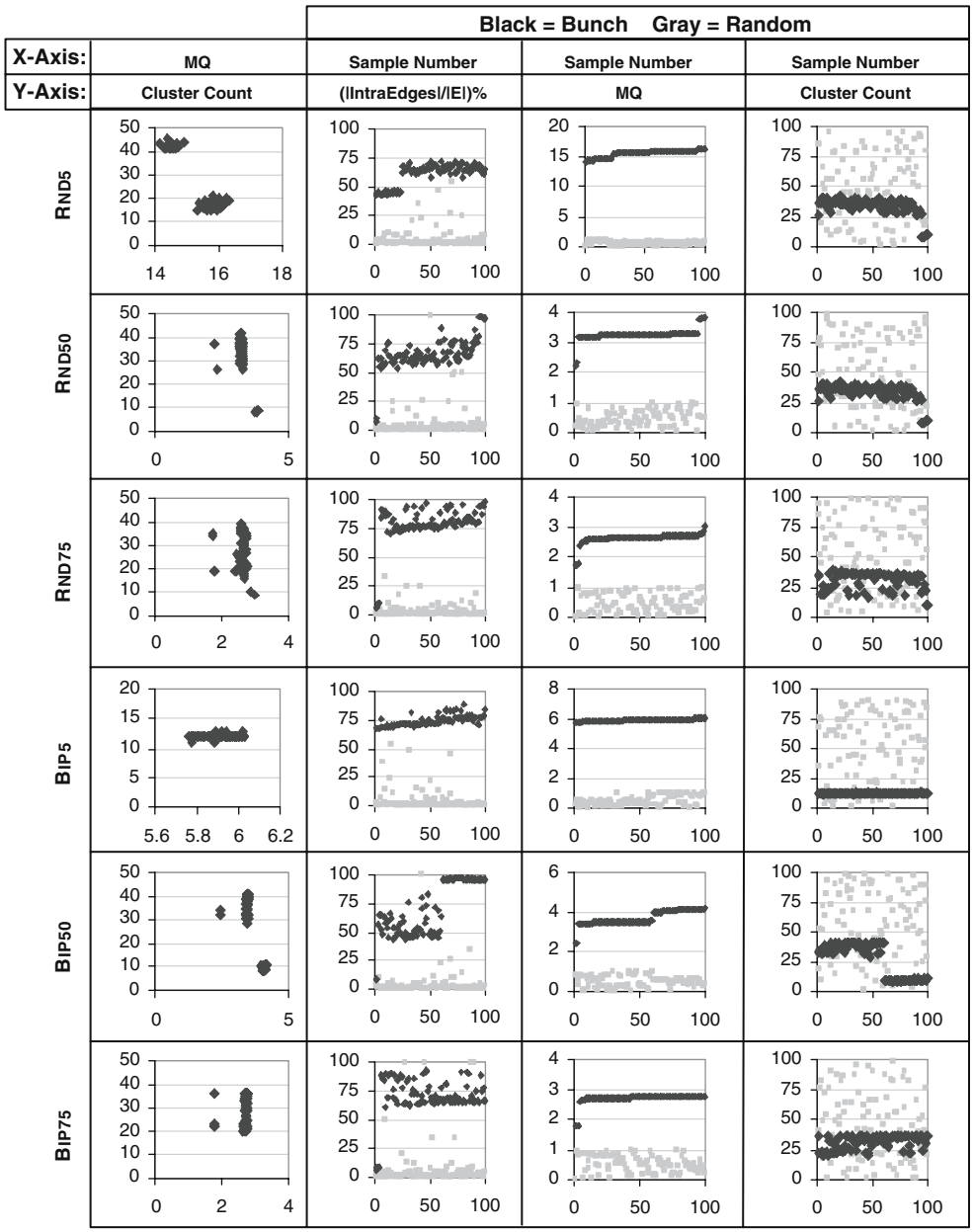


**Fig. 10** The structural search landscape for the random MDGs

of the search is to maximize MQ, which is designed to reward intra-edges. By inspecting all of the graphs for this view it appears that the probability of selecting a random partition (gray data points) with a high intra-edge percentage is rare.

- The third view from the left shows the MQ value for the initial random partition, and the MQ value of the partition produced by Bunch. The samples are sorted and displayed by increasing MQ value. Interestingly, the clustered results produce a relatively smooth line, with points of discontinuity corresponding to different basins of attraction. Another observation is that Bunch generally improves MQ much more for the real software systems, when compared to the random systems with a high edge density (RND50, RND75, BIP50, BIP75).
- The final view (far right column) compares the number of clusters produced by Bunch (black) with the number of clusters in the random starting point (gray). This view indicates that the random starting points appear to have a uniform distribution with respect to the number of clusters. We expect the random graphs to have from 1 (i.e., a single cluster containing all nodes) to $N$ (i.e., each node is placed in its own cluster) clusters. The y-axis is scaled to the total number of modules in the system (see Table 1).

This view shows that Bunch always converges to a "basin of attraction" regardless of the number of clusters in the random starting point. This view also supports the claim made in the first view where the standard deviation for the cluster counts appears to be smaller for the real systems than they are for the random systems.

When examining the structural views collectively, the degree of commonality between the landscapes for the systems in the case study is surprisingly similar. We do not know exactly why Bunch converges this way, although we speculate that this positive result may be based on a good design property of the MQ fitness function. Section 2.2 described that the MQ function works on the premise of maximizing intra-edges, while at the same time, minimizing inter-edges. Since the results converge to similar MQ values, we speculate that the search space contains a large number of isomorphic configurations that produce similar MQ values Mitchell and Mancoridis (2001). Once Bunch encounters one of these areas, its search algorithms cannot find a way to transform the current partition into a new partition with higher MQ.

#### 4.2.2 *Examining similarity across different clustering results*

This section describes the similarity search landscape, which focuses on modeling the extent of similarity across all of the clustering results. For example, given an edge $\langle u, v\rangle$ from the MDG, we can determine, for a given clustering run, if modules $u$ and $v$ are in the same or different clusters. If we expand this analysis to look across many clustering runs we would like to see modules $u$ and $v$ consistently appearing (or not appearing) in the same cluster for most of the clustering runs. The other possible relationship between modules $u$ and $v$ is that they sometimes appear in the same cluster, and other times appear in different clusters. This result would convey a sense of dissimilarity with respect to these modules, as the $\langle u, v\rangle$ edge drifts between being an inter-edge and an intra-edge.

The main observation from analyzing the structural search landscapes in the previous section is that the results produced by Bunch are stable. For all of the MDGs we observe similar characteristics, but we were troubled by the similarity of the search landscapes for real software systems when compared to the landscapes of the random graphs. We expected that the random graphs would produce different results when compared to the real software systems.

In order to investigate the search landscape further we measured the degree of similarity of the placement of nodes into clusters across all of the clustering runs to see if there were any differences between random graphs and real software systems. Bunch creates a subsystem hierarchy, where the lower levels contain detailed clusters, and higher levels contain clusters of clusters. Because each subsystem hierarchy produced by Bunch may have a different height, we decided to measure the similarity between multiple clustering runs using the initial (most detailed) clustering level.

The procedure used to determine the similarity between a series of clustering runs works as follows:

1. Create a counter $C_{\langle u,v\rangle}$ for each edge $\langle u, v\rangle$ in the MDG, initialize the counter to zero for all edges.
2. For each clustering run, take the lowest level of the clustering hierarchy and traverse the set of edges in the MDG; If the edge $\langle u, v\rangle$ is an intra-edge increment the counter $C_{\langle u,v\rangle}$ associated with that edge.
3. Given that there are $N$ clustering runs, each counter $C_{\langle u,v\rangle}$ will have a final value in the range of $0 \leq C_{\langle u,v\rangle} \leq N$. We then normalize the $C_{\langle u,v\rangle}$, by dividing by $N$, which provides the percentage $P_{\langle u,v\rangle}$ of the number of times that edge $\langle u, v\rangle$ appears in the same cluster across all clustering runs.
4. The frequency of the $P_{\langle u,v\rangle}$ is then aggregated into the ranges: {[0,0], (0,10], (10,75), [75,100]}. These ranges correspond to no (zero), low, medium and high similarity, respectively. In order to compare results across different systems, the frequencies are normalized into percentages by dividing each value in the range set by the total number of edges in the system ($|E|$).

Using the above process across multiple clustering runs enables the overall similarity of the results to be studied. For example, having a large *zero* and *high* similarity is good, as these values highlight edges that either never or always appear in the same cluster. The *low* similarity measure captures the percentage of edges that appear together in a cluster less than 10% of the time, which is desirable. However, having a large *medium* similarity measure is undesirable, since this result indicates that many of the edges in the system appear as both inter- and intra-edges in the clustering results.

Now that the above approach for measuring similarity has been described, Table 4 presents the similarity distribution for the systems used in the case study. This table presents another interesting view of the search landscape, as it exhibits characteristics that differ for random and real software systems. Specifically:

- The real systems tend to have large values for the *zero* and *high* categories, while the random systems score lower in these categories. This indicates that the results for the real software systems have more in common than the random systems do.
- The random systems tend to have much larger values for the *medium* category, further indicating that the similarity

**Table 4** The similarity landscape of the case study systems

| Application name | Edge density percent | Similarity distribution ($S$) | | | |
|---|---|---|---|---|---|
| | | **Zero (%)** ($S = 0\%$) | **Low (%)** ($0\% < S \leq 10\%$) | **Medium (%)** ($10\% < S < 75\%$) | **High (%)** ($S \geq 75\%$) |
| Telnet | 21.42 | 34.56 | 27.16 | 13.58 | 24.70 |
| PHP | 10.10 | 48.16 | 19.18 | 11.70 | 20.96 |
| Bash | 21.52 | 58.15 | 22.86 | 6.32 | 12.67 |
| Lynx | 16.04 | 49.28 | 30.64 | 8.99 | 11.09 |
| Bunch | 3.17 | 48.70 | 20.23 | 9.36 | 21.71 |
| Swing | 1.77 | 61.53 | 13.81 | 9.84 | 14.82 |
| Kerberos5 | 2.44 | 57.55 | 19.06 | 9.75 | 13.64 |
| Rnd5 | 5.00 | 12.14 | 54.65 | 24.71 | 8.50 |
| Rnd50 | 50.00 | 32.46 | 37.97 | 29.49 | 0.08 |
| Rnd75 | 75.00 | 33.80 | 30.39 | 35.81 | 0.00 |
| Bip5 | 5.00 | 37.27 | 20.97 | 13.46 | 28.30 |
| Bip50 | 50.00 | 47.21 | 23.89 | 28.60 | 0.30 |
| Bip75 | 75.00 | 29.90 | 38.36 | 31.74 | 0.00 |

of the results produced for the real systems is better than for the random systems.

- The real systems have relatively low edge densities. The SWING system is the most sparse (1.77%), and the BASH system is the most dense (21.52%). When we compare the real systems to the random systems we observe a higher degree of similarity between the sparse random graphs and the real systems then we do between the real systems and the dense random graphs (RND50, RND75, BIP50, BIP75).
- It is noteworthy that the dense random graphs typically have very low numbers in the *high* category, indicating that it is very rare that the same edge appears as an intra-edge from one run to another. The result is especially interesting considering that the MQ values presented in the structural landscape change very little for these random graphs. This outcome also supports the isomorphic basin of attraction conjecture proposed in the previous section, and the observation that the number of clusters in the random graphs vary significantly.

The collective results show that investigating similarity is important when evaluating software clustering results. Analysis of the structural search landscape showed similar behavior for both the real and random *MDGs*. Since the MQ value is based on the topology of the partitioned MDG, the MQ measurement for two or more partitions of an MDG will be similar if the number of clusters, intra-, and inter-edges are relatively the same. The previous section highlighted that Bunch did a good job at finding partitions of the MDG that have similar MQ values for both real or random systems. In order to be valuable to software engineers, the clustering results must produce similar solutions, where pairs of modules find themselves in the same or different clusters consistently. Table 4 shows that a majority of the modules for real systems fall into the zero and high categories, which is desirable. However, the Similarity Distribution for the random systems shows that a majority of the modules fall into the low and medium categories. This outcome is not desirable. For example, except for Bip5 system, at least 25% of the modules in the random systems were categorized in the medium category, which means that in any given clustering solution, over 25% of the modules are likely to be placed in different cluster on another run.

## 5 Related work

Robert Schwanke's tool Arch Schwanke (1991) implemented a heuristic semi–automatic approach to software clustering. Arch is designed to help software engineers understand the current system structure, reorganize the system structure, integrate advice from the system architects, document the system structure, and monitor compliance with the recovered structure. Similar to Bunch, Schwanke's clustering heuristics are based on the principle of maximizing the cohesion of procedures placed in the same module, while at the same time, minimizing the coupling between procedures that reside in different modules. Arch also supports a feature called maverick analysis. With maverick analysis, modules are refined by locating misplaced procedures. These procedures are then prioritized and relocated to more appropriate modules. Schwanke also investigated the use of neural networks and

classifier systems to modularize software systems Schwanke and Hanson (1998).

In Hausi Müller's work, one sees the basic building block of a cluster change from a module to a more abstract software structure called a subsystem. Also, Müller's tool, Rigi Müller et al. (1992, 1993), implements many heuristics that guide software engineers during the clustering process. Many of the heuristics discussed by Müller focus on measuring the "strength" of the interfaces between subsystems. A unique aspect of Müller's work is the concept of an omnipresent module, which is a feature that was later adopted by Bunch. Müller contends that omnipresent modules should not be considered when performing cluster analysis because they obscure the system's structure. One last interesting aspect of Müller's work is the notion that the module names themselves can be used as a clustering criteria. Building on this work Anquetil and Lethbridge (1999) used similarity in module names to guide their clustering approach.

Like Bunch, Choi and Scacchi's (1999) technique is based on maximizing the cohesion of subsystems. Their clustering algorithm starts with a directed graph called a resource flow diagram (RFD) and uses the articulation points of the RFD as the basis for forming a hierarchy of subsystems. Each articulation point and connected component of the RFD is used as the starting point of forming subsystems.

Most approaches to clustering software systems are bottom-up. These techniques produce high-level structural views of a software system by starting with the system's source code. Murphy's work with Software Reflexion Models Murphy et al. (2001) takes a different approach by working top-down. The goal of the Software Reflexion Model is to identify differences that exist between a designer's understanding of the system structure and the actual organization of the source code. By understanding these differences, a designer can update their conceptual model, or the source code can be modified to adhere to the organization of the system that is understood by the designer.

Eisenbarth, Koschke and Simon Eisenbarth et al. (2001) developed a technique for mapping a system's externally visible behavior to relevant parts of the source code using concept analysis. Their approach uses static and dynamic analyses to help users understand a system's implementation without upfront knowledge about its source code.

Tzerpos' and Holt's ACDC clustering algorithm Tzerpos and Holt (2000) uses patterns that have been shown to have good program comprehension properties to determine the system decomposition. The authors define 7 *subsystem patterns* and then describe their clustering algorithm that systematically applies these patterns to the software structure. After applying the subsystem patterns of the ACDC clustering algorithm, most, but not all, of the modules are placed into subsystems. ACDC then uses orphan adoption Tzerpos and Holt (1997) to assign the remaining modules to appropriate subsystems.

Koschke (2000) thesis examines 23 different clustering techniques and classifies them into connection-, metric-, graph-, and concept-based categories. Of the 23 clustering techniques examined, 16 are fully-automatic and 7 are semi-automatic. Koschke also created a semi-automatic clustering framework based on modified versions of the fully-automatic techniques discussed in his thesis. The goal of Kochke's framework is to enable a collaborative session between his clustering framework and the user. The clustering algorithm does the processing, and the user validates the results.

Using Bunch's MQ function, Mahdavi et al. (2003) combined results from multiple hill climbing runs to identify software clusters, an approach similar to the one that we used for evaluating software clustering results Mitchell and Mancoridis (2001). Harman et al. (2005) also investigated reported on fitness functions to guide a software clustering GA, and compared their functions to our MQ function.

We also created a tool called CRAFT Mitchell and Mancoridis (2001) as a platform to compare the results produced by different software clustering algorithms. This tool visualized the difference between the solutions produced by multiple clustering algorithms by measuring the similarity between the results. The CRAFT tool used two similarity measurements we developed called MeCl and EdgeSim Mitchell and Mancoridis (2001), which improved on some problems that we discovered with another similarity measurement developed by Tzerpos and Holt Tzerpos and Holt (1999) called MoJo. The particular problem that we discovered was with the large deviation in MoJo's similarity results when comparing similar partitioned graphs with a large number of isomorphic modules. Wen and Tzerpos (2004) have since updated the MoJo measurement, and performed a comparative analysis to demonstrate how the updated measurement can be used to measure the similarity between the results produced by different clustering algorithms.

A more recent study by Wu et al. (2005) compared six clustering algorithms (including Bunch) on five large open source systems. They examined the clustering algorithms using three different criteria including stability (a small change to the system results in a small change in the clustering results), authoritativeness (how close is the result of the clustering algorithm to an authoritative result), and extremity of the cluster distribution (does the clustering algorithm avoid many large and many small clusters). Bunch scored the worst in the stability comparison, however, the authors used the MoJo measurement Tzerpos and Holt (1999) as the basis to determine stability, which we found was not the best measure of similarity Mitchell and Mancoridis (2001). Bunch scored the best in the extremity test, and did very well in the authoritativeness test (only outranked slightly by a complete

linkage algorithm using the Jaccard coefficient Anquetil et al. (1999)).

## 6 Conclusions

Reverse engineering subsystem structures from source code provides valuable insight into the software architecture that is useful for a variety of software maintenance activities. This information is especially helpful when other forms of traditional design documentation are outdated or not available. This paper described our approach to inferring subsystem structures by searching for good partitions of the module dependency graph.

By modeling the search landscape we highlighted several aspects of Bunch's clustering results that would not have been obvious by examining an individual clustering result. We also gained some intuition about why the results produced by Bunch had many common structural properties independent of whether the MDGs represented real systems or were randomly generated.

Our case study produced some interesting results, some of which were surprising. We expected that calibrating the clustering threshold ($\eta$) used to guide the hill climbing process would either improve MQ or reduce variability in the clustering results, neither was found to be true. Also, although our simulated annealing technique did not find partitions of the MDG with a higher MQ, it did reduce the number of MQ evaluations, and therefore the overall clustering runtime for the systems that we examined. We also observed that our clustering algorithm tended to converge to one or two "neighborhoods" of related solutions for all of the systems that we examined. Finally, some rare solutions, with a significantly higher MQ, were discovered for the ispell and rcs systems.

Practitioners relying on software engineering tools must have confidence that they will produce useful results on their particular system. This is especially important for a system like Bunch, which is unlikely to return the exact same result from one run to the next. Much of the previous research on evaluating software clustering results relied on small empirical studies, subjective feedback from developers, or measuring the similarity of a clustering result against a small collection of reference systems. In this paper we demonstrated how a large collection of clustering results can be used to model the search landscape of any system, and how the search landscape can be used to evaluate search–based clustering algorithms like Bunch.

**Acknowledgments** The authors would like to thank the National Science Foundation (grant numbers CCR-9733569 and CISE-9986105), AT&T and Sun Microsystems for their generous support of this research. 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, US Government, AT&T, or Sun Microsystems.

## References

Shaw M, Garlan D (1996) Software architecture: perspectives on an emerging discipline. Prentice-Hall, Englewood Cliffs

Lakhotia A (1997) A unified framework for expressing software subsystem classification techniques. J Syst Softw 36(3):211–231

Hutchens D, Basili R (1985) System structure analysis: clustering with data bindings. IEEE Trans Softw Eng 11:749–757

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

Choi S, Scacchi W (1999) Extracting and restructuring the design of large systems. IEEE Softw 66–71

Müller H, Orgun M, Tilley S, Uhl J (1992) Discovering and reconstructing subsystem structures through reverse engineering. In: Technical report DCS-201-IR, Department of Computer Science, University of Victoria

Lindig C, Snelting G (1997) Assessing modular structure of legacy code based on mathematical concept analysis. In: Proceedings of international conference on software engineering

van Deursen A, Kuipers T (1999) Identifying objects using cluster and concept analysis. In: International conference on software engineering, ICSM'99. IEEE Computer Society, pp 246–255

Anquetil N, Fourrier C, Lethbridge T (1999) Experiments with hierarchical clustering algorithms as software remodularization methods. In: Proceedings of working conference on reverse engineering

Affenzeller M, Mayrhofer R (2002) Generic heuristics for combinatorial optimization problems. In: Proceedings of the 9th international conference on operational research (KOI)

Clark J, Dolado J, Harman M, Hierons R, Jones B, Lumkin M, Mitchell BS, Mancoridis S, Rees K, Roper M, Shepperd M (2003) Reformulating software engineering as a search problem. J IEE Proc Softw 150(3):161–175

Garey MR, Johnson DS (1979) Computers and Intractability. Freeman WH, San Francisco

Mitchell BS, Mancoridis S (2001) Comparing the decompositions produced by software clustering algorithms using similarity measurements. In: Proceedings of international conference of software maintenance

Mitchell M (1997) An introduction to genetic algorithms. The MIT Press, Cambridge

Doval D, Mancoridis S, Mitchell BS (1999) Automatic clustering of software systems using a genetic algorithm. In: Proceedings of software technology and engineering practice

Chen Y, Gansner E, Koutsofios E (1997) A C++ data model supporting reachability analysis and dead code detection. In: Proceedings of 6th European software engineering conference and 5th ACM SIGSOFT symposium on the foundations of software engineering

Chen Y (1995) Reverse engineering. In: Krishnamurthy B (ed) Practical reusable UNIX software, chap 6, pp 177–208. Wiley, New York

Mitchell BS (2002) A heuristic search approach to solving the software clustering problem. PhD Thesis, Drexel University, Philadelphia

Holt RC, Winter A, Schurr A (2000) Gxl: toward a standard exchange format. In: Proceedings of working conference on reverse engineering

GXL GXL: Graph eXchange Language: Online guide. http://www.gupro.de/GXL

Gansner ER, Koutsofios E, North SC, Vo KP (1993) A technique for drawing directed graphs. IEEE Trans Softw Eng 19(3):214–230

Mancoridis S, Mitchell BS, Rorres C, Chen Y, Gansner ER (1998) Using automatic clustering to produce high-level system organizations of source code. In: Proceedings of 6th international workshop on program comprehension

Mancoridis S, Mitchell BS, Chen Y, Gansner ER (1999) Bunch: a clustering tool for the recovery and maintenance of software system

structures. In: Proceedings of international conference of software maintenance, pp 50–59

Mitchell BS, Traverso M, Mancoridis S (2001) An architecture for distributing the computation of software clustering algorithms. In: The working IEEE/IFIP conference on software architecture (WICSA 2001)

Kirkpatrick S, Gelatt JR. CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220:671–680

Mitchell BS, Mancoridis S (2002) Using heuristic search techniques to extract design abstractions from source code. In: Proceedings of the genetic and evolutionary computation conference

Schwanke R, Hanson S (1998) Using neural networks to modularize software. Mach Learn 15:137–168

Müller H, Orgun M, Tilley S, Uhl J (1993) A reverse engineering approach to subsystem structure identification. J Softw Maintenance Res Practice 5:181–204

Anquetil N, Lethbridge T (1999) Recovering software architecture from the names of source files. In: Proceedings of working conference on reverse engineering

Murphy G, Notkin D, Sullivan K (2001) Software reflexion models: bridging the gap between design and implementation. IEEE Trans Softw Eng 364–380

Eisenbarth T, Koschke R, Simon D (2001) Aiding program comprehension by static and dynamic feature analysis. In: Proceedings of the IEEE international conference of software maintenance (ICSM 2001)

Tzerpos V, Holt RC (2000) ACDC: an algorithm for comprehension-driven clustering. In: Proceedings of working conference on reverse engineering, pp 258–267

Tzerpos V, Holt RC (1997) The orphan adoption problem in architecture maintenance. In: Proceedings of working conference on reverse engineering

Koschke R (2000) Evaluation of automatic re-modularization techniques and their integration in a semi-automatic method. PhD Thesis, University of Stuttgart

Mahdavi K, Harman M, Hierons R (2003) A multiple hill climbing approach to software module clustering. In: Proceedings of the IEEE international conference of software maintenance (ICSM 2003)

Mitchell BS, Mancoridis S (2001) CRAFT: a framework for evaluating software clustering results in the absence of benchmark decompositions. In: Proceedings of working conference on reverse engineering

Harman M, Swift S, Mahdavi K (2005) An empirical study of the robustness of two module clustering fitness functions. In: Proceedings of the genetic and evolutionary computation conference

Tzerpos V, Holt RC (1999) MoJo: a distance metric for software clustering. In: Proceedings of working conference on reverse engineering

Wen Z, Tzerpos V (2004) Evaluating similarity measures for software decompositions. In: ICSM, pp 368–377

Wu J, Hassan A, Holt R (2005) Comparison of clustering algorithms in the context of software evolution. In: Proceedings of the IEEE international conference of software maintenance (ICSM 2005)