Modeling the Search Landscape of Metaheuristic Software Clustering Algorithms

Brian S. Mitchell and Spiros Mancoridis

Department of Computer Science
Drexel University, Philadelphia PA 19104, USA,
{bmitchell, Spiros.Mancoridis}@drexel.edu,
WWW home page: http://www.mcs.drexel.edu/~{bmitchel, smancori}

Abstract. Software clustering techniques are useful for extracting architectural information about a system directly from its source code structure. This paper starts by examining the Bunch clustering system, which uses metaheuristic search techniques to perform clustering. Bunch produces a subsystem decomposition by partitioning a graph formed from the entities (e.g., modules) and relations (e.g., function calls) in the source code, and then uses a fitness function to evaluate the quality of the graph partition. Finding the best graph partition has been shown to be a NP-hard problem, thus Bunch attempts to find a sub-optimal result that is "good enough" using search algorithms. Since the validation of software clustering results often is overlooked, we propose an evaluation technique based on the search landscape of the graph being clustered. By gaining insight into the search space, we can determine the quality of a typical clustering result. This paper defines how the search landscape is modeled and how it can be used for evaluation. A case study that examines a number of open source systems is presented.

1 Introduction and Background

Since many software systems are large and complex, appropriate abstractions of their structure are needed to simplify program understanding. Because the structure of software systems are usually not documented accurately, researchers have expended a great deal of effort studying ways to recover design artifacts from source code.

For small systems, source code analysis tools [3, 9] can extract the source-level components (*e.g.*, modules, classes, functions) and relations (*e.g.*, method invocation, function calls, inheritance) of a software system. However, for large systems, these tools are, at best, useful for studying only specific areas of the system.

For large systems there is significant value in identifying the abstract (high-level) entities, and then modeling them using architectural components such as subsystems and their relations. Subsystems provide developers with structural information about the numerous software components, their interfaces, and their interconnections. Subsystems generally consist of a collection of collaborating

source code resources that implement a feature or provide a service to the rest of the system. Typical resources found in subsystems include modules, classes, and possibly other subsystems.

The entities and relations needed to represent software architectures are not found in the source code. Thus, without external documentation, we seek other techniques to recover a reasonable approximation of the software architecture using source code information. Researchers in the reverse engineering community have applied a variety of software clustering approaches to address this problem. These techniques determine clusters (subsystems) using source code component similarity [18, 4, 17], concept analysis [10, 5, 1], or information available from the system implementation such as module, directory, and/or package names [2].

In this paper we examine the Bunch software clustering system [12, 11, 13]. Unlike the other software clustering techniques described in the literature, Bunch uses search techniques to determine the subsystem hierarchy of a software system, and has been shown to produce good results for many different types of systems. Additionally, Bunch works well on large systems [16] such as `swing` and `linux`, and some work has been done on evaluating the effectiveness of the tool [14, 15].

Bunch starts by generating a random solution from the search space, and then improves it using search algorithms. 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 and by using similarity measurements [14]. Furthermore, as we will demonstrate later in this paper, we validated that the probability of finding a good solution by random selection is extremely small, thus Bunch appears to be producing a family of related solutions.

The above observations intrigued us to answer why Bunch produces similar results so consistently. The key to our approach will be to model the search landscape of each system undergoing clustering, and then to analyze how Bunch produces results within this landscape.

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 such that the clusters represent subsystems. Since graph partitioning is known to be NP-hard [7], obtaining a good solution by random selection or exhaustive exploration of the search space is unlikely. Bunch overcomes this problem by using metaheuristic-search techniques.

Figure 1 illustrates the clustering process used by Bunch. In the preparation phase, source code analysis tools 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.

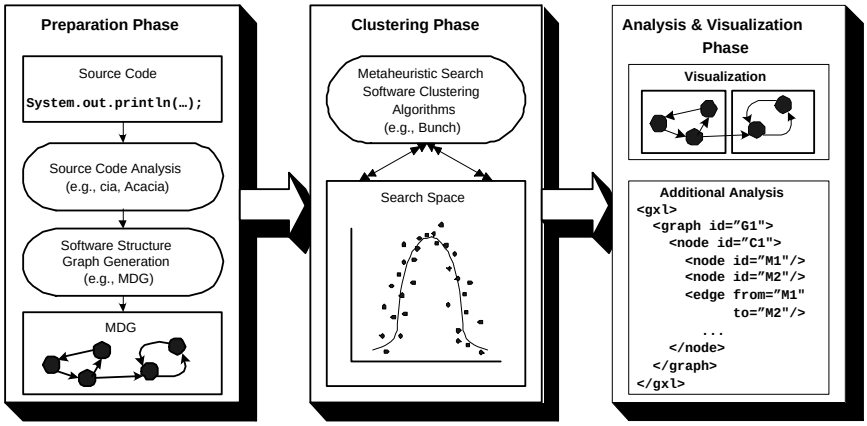

Fig. 1. Bunch's Software Clustering Process

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*) [16]. 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, metaheuristic search algorithms are used in the clustering phase of Figure 1 in an attempt to improve the *MQ* of the randomly generated partition. Bunch implements several hill-climbing algorithms [12, 16] and a genetic algorithm [12].

Once Bunch's search algorithms converge, a result can be viewed as XML [8] or using the dotty [6] graph visualization tool.

2.1 Clustering Example with Bunch

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 7 classes, and 9 inter-class relations. All remaining dependencies to and from the other packages have been collapsed into a single relation to simplify the visualization.

Figure 2 animates 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.

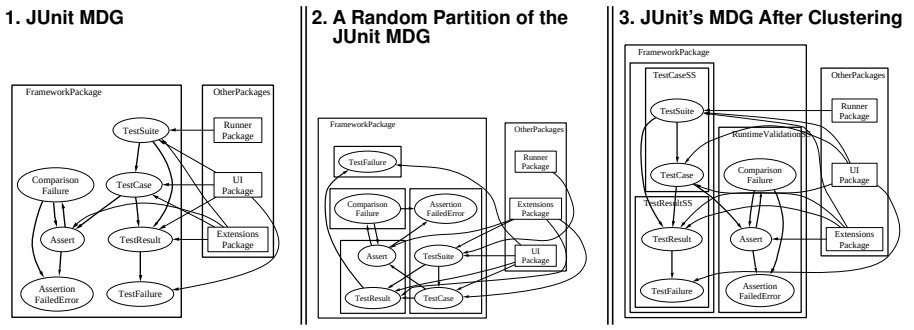

Fig. 2. JUnit Example

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 low quality solution. This intuition is validated by inspecting the random partition, which contains 2 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 Figure 2. The solution proposed by Bunch has many good characteristics 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 Figure 2 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 associated with each *MDG*.

3 Modeling the Search Landscape

This section presents an approach to modeling the search landscape of software clustering algorithms. The search landscape is modeled using series of views, because software clustering results have many dimensions, and combining too much detail into a single view is confusing.

The search landscape is examined from two different perspectives. The first perspective examines the structural aspects of the search landscape, and the second perspective focuses on the similarity aspects of the landscape.

3.1 The Structural Search Landscape

The structural search landscape 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 intraedges (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.

3.2 The Similarity Search Landscape

The similarity search landscape focuses on modeling the extent of similarity across all of the clustering results. For example, given an edge $< u, v >$ 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 $< u, v >$ edge drifts between being an interdedge and an intraedge.

4 Case Study

This section describes a case study illustrating the effectiveness of using the search landscape to evaluate Bunch's software clustering results. Table 1 de-

Table 1. 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

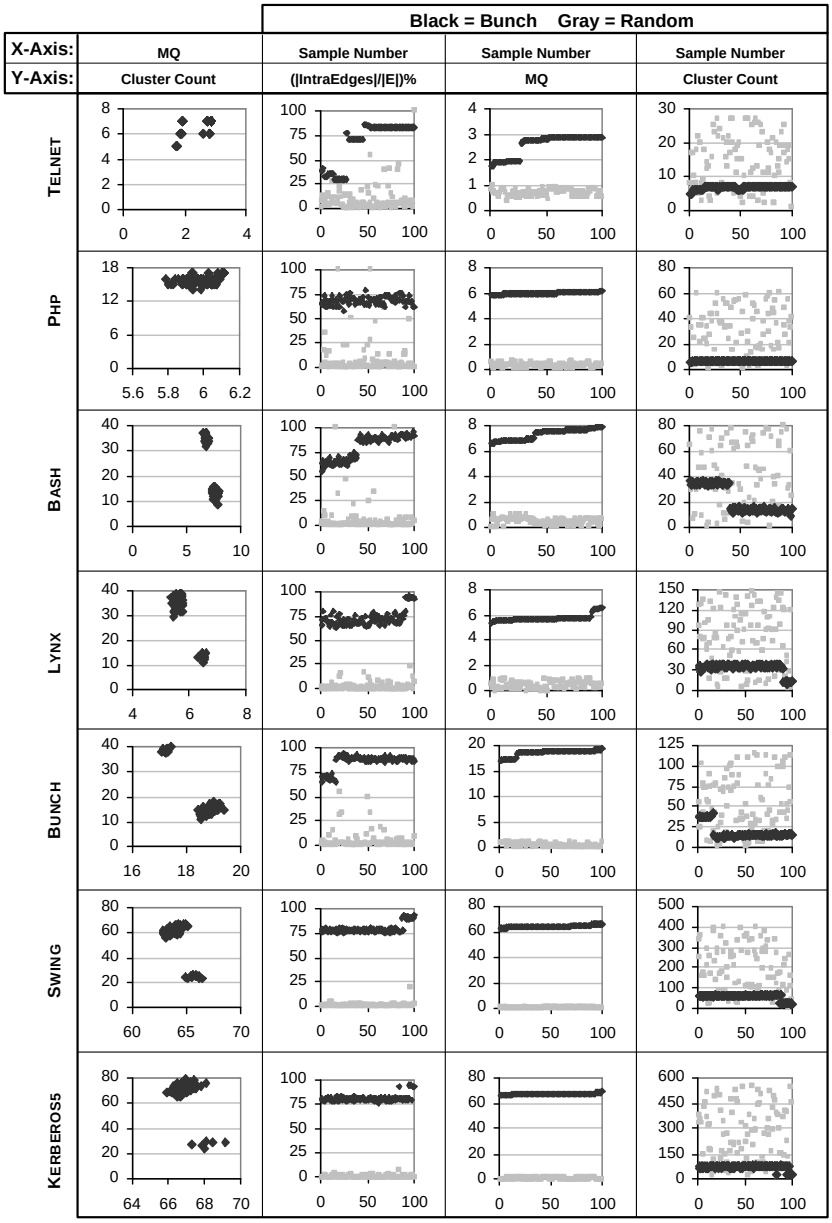

Fig. 3. The Structural Search Landscape for the Open Source Systems

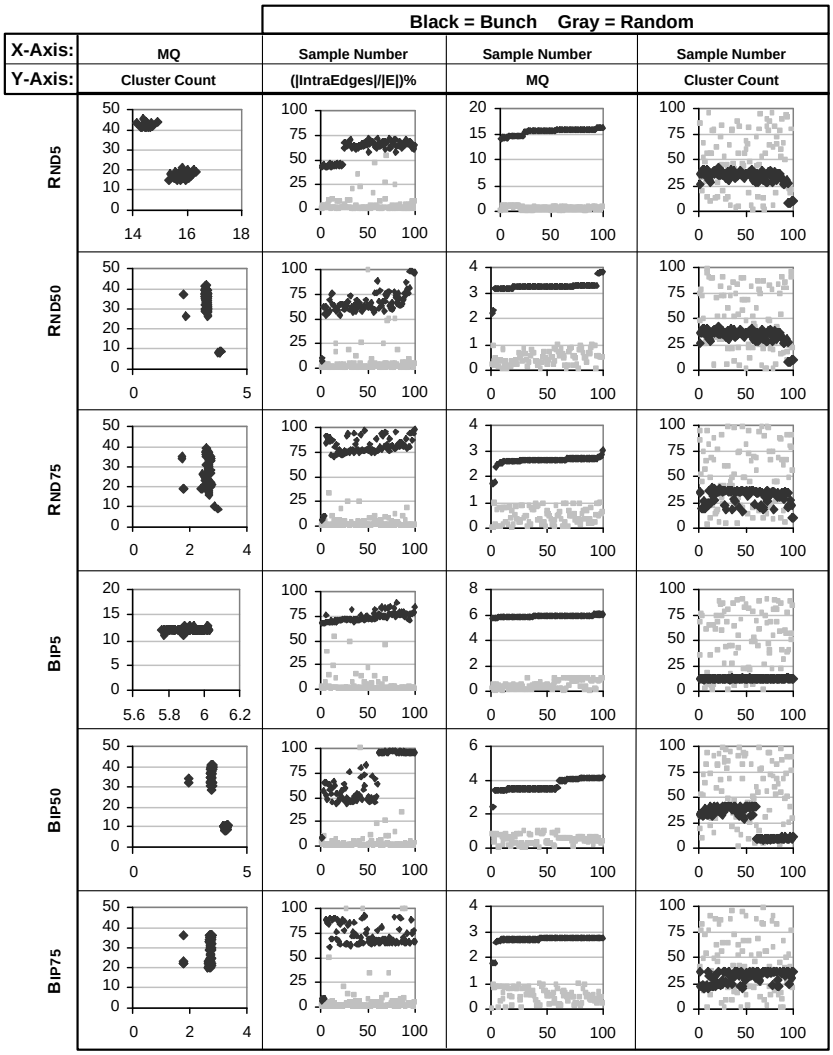

Fig. 4. The Structural Search Landscape for the Random MDGs

scribes the systems used in this case study, which consist of 7 open source systems and 6 randomly generated *MDG*s. Mixing the random graphs with real software graphs enables us to compare how Bunch handles real versus randomly generated graphs.

Some explanation is necessary to describe how the random *MDG*s were generated. Each *MDG* used in this case study consists of 100 modules, and has a name that ends 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 $0.05 * (n(n-1)/2)$, which is 5% of the total possible number of edges that can exist for a graph containing n modules.

Each system in Table 1 was clustered 100 times using Bunch's default settings.[1] It should also be noted that although the individual clustering runs are independent, the landscapes are plotted in order of increasing *MQ*. This sorting highlights some results that would not be obvious otherwise.

4.1 The Structural Landscape

Figure 3 shows the structural search landscape of the open source systems, and Figure 4 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 Figures 3 and 4:

- 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 systems, but do not see this characteristic in the real systems.
- The view that shows the percentage of intraedges in the clustering results (second column from the left) indicates that Bunch produces consistent solutions that have a relatively large percentage of intraedges. Also, since the 100 samples were sorted by *MQ*, we observe that the intraedge percentage increases as the *MQ* values increase. 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 intraedge 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).

[1] An analysis on the impact of altering Bunch's clustering parameters was done in a 2002 GECCO paper [16].

– 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 described that the *MQ* function works on the premise of maximizing intraedges, while at the same time, minimizing interedges. 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. Once Bunch encounters one of these areas, it's search algorithms cannot find a way to transform the current partition into a new partition with higher *MQ*.

4.2 The Similarity Landscape

The main observation from analyzing the structural search landscapes is that the results produced by Bunch are stable. For all of the *MDG*s 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,[2] 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:

[2] The height of the subsystem hierarchy produced by Bunch generally does not differ by more than 3 levels for a given system, but the heights of the hierarchy for different systems may vary dramatically.

1. Create a counter $C_{<u,v>}$ for each edge $< u, v >$ 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 $< u, v >$ is an intraedge increment the counter $C_{<u,v>}$ associated with that edge.
3. Given that there are N clustering runs, each counter $C_{<u,v>}$ will have a final value in the range of $0 \leq C_{<u,v>} \leq N$. We then normalize the $C_{<u,v>}$, by dividing by N, which provides the percentage $P_{<u,v>}$ of the number of times that edge $< u, v >$ appears in the same cluster across all clustering runs.
4. The frequency of the $P_{<u,v>}$ 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 intraedges in the clustering results.

Table 2. The Similarity Landscape of the Case Study Systems

Application Name	Edge Density Percent	Similarity Percentage (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

Now that the above approach for measuring similarity has been described, Table 2 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 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 intraedge 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.

5 Conclusions

Ideally, the results produced by Bunch could be compared to an optimal reference solution, but this option is not possible since the graph partitioning technique used by Bunch for software clustering is NP-hard. User feedback has shown that the results produced by Bunch are "good enough" for assisting developers performing program comprehension and software maintenance activities, however, work on investigating why Bunch produces consistent results had not been performed until now.

Through the use of a case study 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 several common properties regardless of whether the *MDG*s were real or randomly generated. A final contribution of this paper is that it demonstrates that modeling the search landscape of metaheuristic search algorithms is a good technique for gaining insight into these types of algorithms.

6 Acknowledgements

This research is sponsored by grants CCR-9733569 and CISE-9986105 from the National Science Foundation (NSF). Any opinions, findings, and conclusions or

recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

1. N. Anquetil. A comparison of graphis of concept for reverse engineering. In *Proc. Intl. Workshop on Program Comprehension*, June 2000.
2. N. Anquetil and T. Lethbridge. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering*, October 1999.
3. Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.
4. S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.
5. A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *International Conference on Software Engineering, ICSM'99*, pages 246–255. IEEE Computer Society, May 1999.
6. E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.
7. M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, 1979.
8. GXL: Graph eXchange Language: Online Guide. http://www.gupro.de/GXL/.
9. J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, October 1999.
10. C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, May 1997.
11. S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, pages 50–59, August 1999.
12. S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, June 1998.
13. B. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD thesis, Drexel University, Philadelphia, PA, USA, 2002.
14. B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proceedings of International Conference of Software Maintenance*, November 2001.
15. B. S. Mitchell and S. Mancoridis. CRAFT: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *Proc. Working Conference on Reverse Engineering*, October 2001.
16. B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of Genetic and Evolutionary Computation Conference*, 2002.
17. H. Müller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5:181–204, 1993.
18. R. Schwanke and S. Hanson. Using Neural Networks to Modularize Software. *Machine Learning*, 15:137–168, 1998.