Modular Network Models for Class Dependencies in Software

Rodrigo Souza
Departamento de Sistemas e Computação
Universidade Federal de Campina Grande
Campina Grande, Brazil
rodrigorgs@gmail.com

***Abstract*—Software clustering algorithms can automatically decompose a software system into modules by analyzing the network of dependencies between its components (e.g., classes in object-oriented systems). Empirical evaluation of these algorithms is difficult because few software systems have reference decompositions to be compared with the decompositions found by the algorithms. Alternatively, the algorithms can be evaluated by applying them on computer-generated networks with built-in decompositions, but the validity of this approach depends on the similarity between real and computer-generated networks. In this paper we present three network models and show that, with a proper choice of parameters, they can generate networks that are indistinguishable from class dependency networks.**

***Keywords*-software clustering; empirical evaluation; complex networks.**

I. Introduction

Large software systems need to be decomposed in modules in order to be effectively maintained by groups of developers working concurrently. Software clustering algorithms [1]–[4], also known as architecture recovery algorithms, find suitable decompositions by analyzing the dependencies between components in a software system (e.g., classes in object-oriented systems) and then grouping together highly interdependent components.

Although many software clustering algorithms have been proposed, there is little empirical evaluation regarding the external quality of the decompositions they find. One important test for a software clustering algorithm is to apply it to a collection of software systems with known reference decompositions made by experienced developers [2]. It is expected that a good algorithm finds decompositions that are similar to reference decompositions, which can be measured by a similarity metric such as MoJo [5] or EdgeSim [6].

Unfortunately, there are few publicly available reference decompositions [7] and, as a result, there are few studies that compare software clustering algorithms. Also, because it is costly to obtain reference decompositions for large systems, most studies test the algorithms with a couple of small and medium systems [2], [8], [9].

An alternative approach is to run the algorithms on synthetic, i.e., computer- generated, component dependency networks with built-in reference decompositions. This approach enables one to test algorithms with arbitrarily large samples in a controlled manner. Nevertheless, this approach is often overlooked, mainly because networks synthesized by naive random network models are very dissimilar from real component dependency networks.

In this paper, we present three recently proposed network models and show empirically that they are capable of synthesizing networks that resemble software component dependency networks. We limit our analysis to networks of static dependencies between classes in object-oriented systems written in Java.

The paper is organized as follows: Section II presents three network models. Section III demonstrates a method to tell if a network resembles a software component dependency network. Section IV shows empirically that the three models synthesize networks that resemble software component dependency networks. Section V discusses the impact of this research and future work.

II. Network Models

Network theory studies general properties of many types of networks by using statistical analysis. In the last decade, remarkable similarities have been found in networks arising from domains as diverse as sociology, biology, technology, and linguistics. It has been shown that, in many networks, the distribution of vertex degrees is a power law, i.e., the number of vertices connected to k edges is proportional to $k^{-\gamma}$, where γ is a positive constant. Networks that share this property are called scale-free networks [10].

Network theory has been applied to class dependency networks, represented as directed unweighted graphs, and it was shown that they are also scale-free [11], [12]. Thus, the study of the structure of object-oriented software systems can benefit from the vast literature available about scale-free networks.

Scale-free network models are algorithms that can be proven, either formally or empirically, to synthesize networks that are scale-free. In this section we present three models that synthesize scale-free networks with built-in modular decompositions: BCR+, CGW, and LF. The networks are represented as directed unweighted graphs with modules, which makes them suitable for testing software clustering algorithms.

A. BCR+

The BCR model [13] synthesizes directed scale-free networks without modules. We have developed an extension to this model, called BCR+, that adds modules to the construction of the network. The BCR+ model accepts the following parameters:

- number of vertices, n;
- a directed graph of modules, G;
- three probabilities, p_1, p_2, and p_3, summing to 1;
- constant μ, with $0 \leq \mu \leq 1$;
- base in-degree, δ_{in};
- base out-degree, δ_{out}.

The graph G contains one vertex for each module that will be created and determines a "supply" relationship between modules. We say that module M_2 is a supplier of module M_1 if, and only if, G contains an edge from M_1 to M_2.

Networks generated by the model can contain both internal edges and external edges. An internal edge connects two vertices in the same module. An external edge connects vertices in distinct modules. In this model, an external edge from a vertex $v_1 \in M_1$ to a vertex $v_2 \in M_2$ is only allowed if M_2 is a supplier of M_1.

The parameter μ controls the proportion of external edges in the network. Lower values of μ lead to networks with fewer external edges.

First the model creates a network in which each module contains exactly one vertex. Then all allowed external edges are added. Finally, the networks is modified by sucessive choices between three operations that add vertices or edges to the network, until it reaches n vertices. Each time, the probability of choosing the i-th operation is p_i.

Before describing the operations, some definitions are needed. The expression in-degree(x) means the number of edges that enter the vertex x; out-degree(x), likewise, is the number of edges that leave vertex x. The expression "to choose a vertex according to f(x)" means that the probability of choosing a vertex x is given by the following probability function:

$$\mathrm{P}(x) = \frac{\mathrm{f}(x)}{\sum_i \mathrm{f}(i)}$$

(the denominator is a normalizing factor, so the sum of probabilities is 1).

The three operations are described next:

1) *Adding a vertex with an outgoing edge.* An existing vertex, w, is chosen according to $\mathrm{f}(x) = \delta_{in} +$ in-degree(x). A new vertex, v, is added to the module of w, together with an edge from v to w.
2) *Adding a vertex with an ingoing edge.* An existing vertex, w, is chosen according to $\mathrm{f}(x) = \delta_{out} +$ out-degree(x). A new vertex, v, is added to the module of w, together with an edge from w to v.
3) *Adding an edge between existing vertices.* A vertex, v, is choosen from the network according to $\mathrm{f}(x) = \delta_{out} +$ out-degree(x). Then an edge is added from vertex v to an existing vertex w, which is chosen according to one of the following cases:
   a) with probability μ, w is chosen among vertices in modules that are suppliers of the module of v;
   b) with probability $1 - \mu$, w is chosen among vertices in that same module as v.

It is easy to see that nodes with high in-degree are more likely to receive new ingoing edges. The parameter δ_{in} can reduce this bias by providing a base in-degree that is applied to all vertices when computing the probabilities. Consider two vertices, v_1 with in-degree 4, and v_2 with in-degree 8. If $\delta_{in} = 0$, v_2 is twice more likely to receive a new incoming edge; if, otherwise, $\delta_{in} = 4$, v_2 is only $\frac{3}{2}$ more likely to receive the edge. The same reasoning applies to δ_{out}.

The BCR+ model is a growth model, meaning that the network is created vertex by vertex, growing from an initial network. It can, therefore, simulate the evolution of a software system. Moreover, it can simulate the evolution of a software system subject to constraints in module interaction, as is the case with top-down system design methodologies.

B. CGW

The CGW model [14] was proposed to model the evolution of software systems organized in modules. It accepts the following parameters:

- number of vertices, n;
- number of modules, m;
- probabilities p_1, p_2, p_3, p_4, summing to 1;
- natural numbers e_1, e_2, e_3, e_4;
- constant α, with $\alpha \geq -1$.

Because the model is described in detail in the original paper, only an intuitive notion of the meaning of the parameters is provided here. This model is similar to BCR+ in which it grows an initial network by sucessive applications of operations. In this case, there are four operations, which are chosen according to probabilities p_1, p_2, p_3, and p_4:

1) adding a vertex with e_1 outgoing edges;
2) adding e_2 edges;
3) rewiring e_3 edges;
4) removing e_4 randomly chosen edges.

The constant α controls the proportion of edges that connect vertices from distinct modules. For $\alpha = -1$, most edges will connect distinct modules. For $\alpha > 0$, most edges will connect vertices in the same module, and the greater the α, the stronger the tendency. For $\alpha = 0$ there is no bias, the two kinds of edges are equaly likely.

C. LF

The LF model [15] is a very flexible model that can generate weighted directed networks with overlapping modules, that is, in which a vertex can belong to more than one module. Unlike the previous models, this is not a growth model: all vertices are generated at once and then the edges are added.

There is also a special version of the model in which the edge weights are discarded and the modules do not overlap. We used the original implementation of this version, available at http://santo.fortunato.googlepages.com/inthepress2. It accepts the following parameters:

- number of nodes, n;
- average in-degree, k, with $k < n$;
- maximum in-degree, max_k, with $k \leq max_k < n$;
- mixing parameter, μ, with $0 \leq \mu \leq 1$;
- minus exponent for the degree distribution, γ;
- minus exponent for the module size distribution, β;
- minimum module size, min_m;
- maximum module size, max_m, with $max_m \geq min_m$;

The sizes of the modules are selected from a power law distribution with exponent $-\beta$. The mixing parameter, μ, is the proportion of edges in the network that connect vertices from distinct modules. In this model, some combinations of parameter values are unfeasible. For example, if $n = 100$ then min_m cannot be 60 (otherwise there would be modules smaller than the minimum module size).

III. Characterization of Software Networks

In the remaining of this paper, we will use the expression "software network" to refer to a network of static dependencies between classes of a software system, in contrast to networks studied in other domains.

Our research hypothesis is that at least one of the presented models can synthesize networks that are similar to software networks. A central issue, thus, is how to measure similarity between networks.

We already know that the models synthesize networks that, just like software networks, are scale-free. This single property, though, is insufficient to prove the hypothesis, since many known scale-free networks are not software networks (e.g., biologic networks). Therefore, the similarity metric should be able to distinguish between software and non-software networks.

In this section we present a similarity metric and validate it by appplying it to a data set containing both software and non-software networks.

A. Similarity Between Networks

In a recent work, Milo et al. [16] proposed to characterize networks by analyzing their triad concentration. A triad is a network with three vertices in which all vertices are connected. There are only 13 distinct triads, one for each possible configuration of directed edges, as shown in Figure 1.

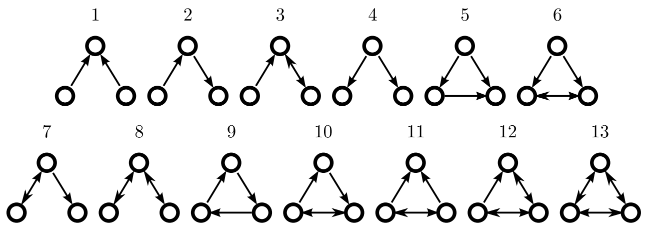

Figure 1. The 13 network triads.

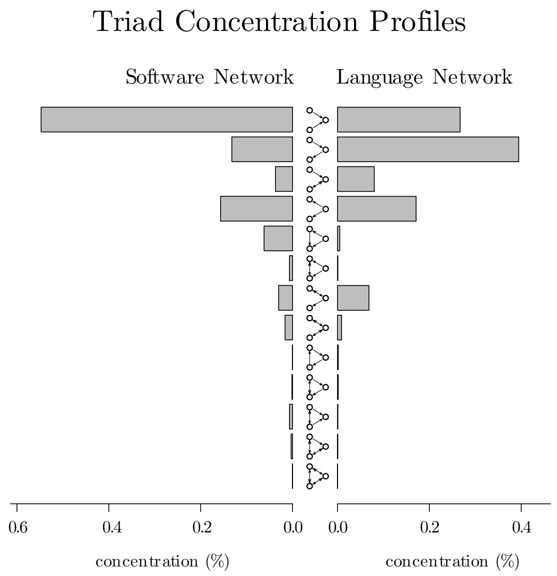

Figure 2. Triad concentration profiles (TCP) for two networks. On the left, network extracted from the software system JabRef (see the Appendix). On the right, word adjacency network for the Japanese language [17].

By counting how many times each triad appears in a network, one can build a triad concentration profile (TCP), which is a vector with 13 numbers that summarize the local structure of the network. Figure 2 shows the TCP for networks from two distinct domains.

Following another work by Milo et al. [17], similarity between two networks can be measured by computing Pearson's correlation coefficient between the corresponding TCPs, which yields a value between -1 (most dissimilar) and 1 (most similar):

$$\text{similarity}(a, b) \ = \ \text{cor}(\text{TCP}(a), \text{TCP}(b)),$$

where a and b are networks, TCP(x) is the triad concentration profile for network x, and cor(x, y) is Pearson's correlation coefficient.

B. Data Set

To support the evaluation of the metric, we have collected 131 networks from many different domains. The networks are described in detail in the Appendix.

1) Software networks: We have collected 65 software systems written in Java, with size ranging from 111 to 35,363 classes. Java was chosen for being a popular programming language in which many open source systems have been written. The software networks, representing static dependencies between classes, were extracted with the tool Dependency Finder[1].

2) Non-software networks: We have collected 66 networks from distinct domains, such as biology, sociology, technology, and linguistics, with size ranging from 32 to 18163 vertices. These networks are freely available on the Internet and have been previously studied in the literature.

C. Evaluation of the Similarity Metric

For the purposes of this research, a similarity metric must fullfil two conditions: (i) it must yield high similarity between software networks, and (ii) it must yield lower similarity between software networks and networks from other domains.

Using the data set we can define S-score, a metric that represents how much a particular network resemble software networks. It is defined as the average similarity between the network and a sample of software networks:

$$\text{S-score}(a) \ = \ \frac{\sum_{s \in S} \text{similarity}(a, s)}{|S|},$$

where S is the set of sample software networks, and $|S|$ is the number of networks in S. In this work we use the full software data set consisting of 65 software networks as our sample.

We used the tool igraph [18] to extract the TCP for each network in the data set. Then, we measured the S-score for each software network, which ranged from 0.83 to 0.98. The average S-score was 0.97 and the standard deviation, 0.03. The high average S-score and the low standard deviation show that the metric successfully characterizes software networks by capturing their common structural patterns.

Then we measured the S-score for each non-software network. The majority of the networks (97.0%) had a S-score lower than 0.83, which is the lowest S-score for software networks in the sample. Some networks, e.g., the friendship networks between students, showed negative S-score, meaning that they are very different from software networks.

Two networks, though, showed high S-score: the network of links between blogs on politics, with S-score 0.97, and the neural network of the worm C. Elegans, with S-score 0.88. Further investigation is needed in order to discover the reasons behind the high values and whether auxiliary metrics can differentiate these networks from software networks.

[1]Available at http://depfind.sf.net.

D. A Network Classification Model

Although the S-score of a network tells how close it is from software networks, it does not tell whether a network is close enough that it can be considered software-like. What is needed is a binary classification model that distinguishes software-like networks from the other networks. The distinction can be made by choosing a suitable S-score threshold. Networks with S-score below the threshold are considered dissimilar from software networks; only networks with S-score above the threshold are considered software-like.

As we have shown on the previous section, there are non-software networks with high S-scores, hence it is impossible to build a perfect classification model, regardless of the threshold. Nonetheless, a classification model can be evaluated by its precision and recall. Consider our data set with both software and non-software networks. Let S be the set of all software networks, and L the set of all networks that were classified by the model as software-like. The precision of the model is

$$\text{precision} : \ \frac{S \cap L}{L},$$

and the recall is

$$\text{recall} : \ \frac{S \cap L}{S}.$$

Increasing the threshold has the effect of reducing the recall, because fewer software networks are classified as software-like. Decreasing the threshold has the effect of reducing the precision, because more non-software networks are classified as software-like.

The choice of a proper threshold, thus, depends on whether it is more important to have high precision or high recall. Because our research hypothesis is that networks synthesized by the presented models are software-like, higher precision means a stronger test, as fewer networks are classified as software-like.

To get 100% precision, the threshold needs to be 0.98, so the non-software network with highest S-score is below the threshold. The recall in this case, though, would be too low, because most software networks would be misclassified. So we chose the value 0.88, that is immediatly above the second greater S-score for a non-software network. With this value, we have both high recall (95.4%) and high precision (96.9%).

IV. Evaluation of Network Models

In the previous section it was shown that many networks, although scale-free, can be distinguished from software networks by a simple classification model based on triad concentration profiles. In this section we show empirically that the three network models previously presented can synthesize networks that are indistinguishable from software networks. The experiment consists of synthesizing networks using many combinations of parameters from the three models, and then classifying each network as software-like or non software-like. Because the possible combinations of parameter values are infinite, we have set the number of vertices to 1000 and then varied the remaining parameters in discrete steps.

A. BCR+ networks

We have chosen five different module graphs, which where extracted from actual dependencies between Java archives distributed with five different software systems of our sample: GEF (2 archives), iBATIS (4 archives), MegaMek (8 archives), findbugs (16 archives), and zk (32 archives). Because many of these archives are intended to be reused in distinct projects, they provide a good coarse-grained approximation to the concept of module.

For the remaining parameters, the following values were chosen:

- $p_1, p_2, p_3 \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, with $p + q + r = 1$ and $p + q > 0$;
- $\delta_{in}, \delta_{out} \in \{0, 1, 2, 3, 4\}$;
- $\mu \in \{0.0, 0.2, 0.4, 0.6\}$.

We chose $p+q > 0$ because otherwise no node would be added to the initial network. We also avoided large values for μ in order to ignore networks with strongly coupled modules.

In total, 9,500 BCR+ networks were synthesized.

B. CGW networks

The parameters values were chosen as such:

- $p1, p2, p3, p4 \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, with $p1 > 0$ and $p1 + p2 + p3 + p4 = 1$;
- $e1, e2, e3, e4 \in \{1, 2, 4, 8\}$;
- $\alpha \in \{-1, 0, 1, 10, 100, 1000\}$;
- number of modules: $m \in \{2, 4, 8, 16, 32\}$.

Note that e_i has no effect when $p_i = 0$; in this case e_i was just set to zero.

In total, 38,790 CGW networks were synthesized.

C. LF networks

The following values were chosen:

- mixing parameter: $\mu \in \{0.0, 0.2, 0.4, 0.6\}$;
- degree exponent distribution: $\gamma \in \{2.18, 2.70, 3.35\}$;
- module size distribution: $\beta \in \{0.76, 0.99, 1.58\}$;
- average degree: $k \in \{5, 10, 15, 25\}$;
- maximum degree: $max_k \in \{58, 157, 482\}$;
- minimum module size: $min_m \in \{1, 10, 273\}$.

In order to choose these values, we analyzed software networks from our sample with approximately 500 to 2,000 nodes, so no network was much bigger or much smaller than the synthetic networks. We computed the exponents for the degree and module size distributions using the maximum likelihood estimation method [19], and then chose the minimum, median and maximum values. For k, $maxk$, and min_m, the values extracted from the software networks were divided by the number of nodes in the network and then multiplied by 1000. We then selected the minimum, median and maximum values. The parameter max_m was left unbound to avoid impossible combinations of parameters.

In total, 1,296 LF networks where synthesized.

Table I
RESULTS FOR THE CLASSIFICATION OF SYNTHETIC NETWORKS

Model	Networks classified as software-like
BCR+	21.18%
CGW	19.40%
LF	31.25%

Table II
RULES FOR PREDICTING THE CLASSIFICATION OF A SYNTHETIC NETWORK. S STANDS FOR SOFTWARE-LIKE AND N STANDS FOR NON SOFTWARE LIKE.

Model	Rule	Accuracy
BCR+	$p_1 \geq 0.7 \Rightarrow S$ $p_1 < 0.7 \Rightarrow N$	82.4%
CGW	$p_1 \geq 0.5 \Rightarrow S$ $p_1 < 0.5 \Rightarrow N$	82.3%
LF	$\gamma < 2.44 \Rightarrow S$ $\gamma \geq 2.44 \Rightarrow N$	78.9%

D. Results

Each synthesized network was classified as software-like or non software-like, using the classification model presented in section III-D. The results are summarized in Table I.

All models synthesized both software-like and non software-like networks. The proportion of software-like networks was greater than 19% for all models, discarding the possibility that this result was obtained by pure chance.

Of course, this result is of little practical value unless there is a relationship between parameter values and S-score. For the purpose of this research, it is important to know which values are more likely to lead to software-like networks.

The algorithm 1R [20] from machine learning was used to help discover such relationship. It analyzes the parameters and the classification of each network and finds a rule that relates the value of one single parameter with the classification. Such rules can be evaluated according to their accuracy, i.e., the proportion of networks that are correctly classified. The rules found by 1R are shown in Table II.

The rules are very simple and, thus, easy to follow. Despite their simplicity, they have high accuracy, approximately 80% for all models.

V. CONCLUSION AND FUTURE WORK

We have shown empirically that network models found in the literature can synthesize networks that resemble the network of static dependencies between classes in object-oriented systems. This result supports the use of synthetic networks in the evaluation of software clustering algorithms.

The use of synthetic data is common in distributed computing research, but still underexplored in software engineering research. Because many reverse engineering tasks rely on dependency data [21], we expect this work to have impact beyond the software clustering community.

We accept that it is important to evaluate the algorithms with real software networks, but we argue that the use of synthetic networks in a complementary manner can give researchers new insights about the algorithms. First, the use of models allows the creation of large test sets, thus diminishing the small sample effects. Moreover, the networks are created in a controlled way, according to model parameters, so it is possible to study the behavior of the algorithms with different parameter values.

In a future work, we intend to use synthetic networks in the evaluation of software clustering algorithms that were previously tested with real networks. After that we will be able to compare the results obtained by the two approaches.

Appendix

This appendix lists the networks used in this work.

A. *Software Networks*

Systems hosted by SourceForge (http://sourceforge.net/):

- AbaGuiBuilder-1.8
- alfresco-labs-deployment-3Stable
- aoi272
- stendhal-0.74
- battlefieldjava-0.1
- checkstyle-5.0
- dom4j-1.6.1
- findbugs-1.3.8
- freetts-1.2.2-bin
- ganttproject-2.0.9
- geoserver-2.0-beta1-bin
- geotools-2.5.5-bin
- gfp_0.8.1
- hibernate-distribution-3.3.1.GA-dist
- hsqldb_1_8_0_10
- iBATIS_DBL-2.1.5.582
- iReport-nb-3.5.1
- JabRef-2.5b2-src
- jailer_2.9.9
- jalopy-1.5rc3
- jasperreports-3.5.2-project
- jfreechart-1.0.13
- pentaho-reporting-engine-classic-0.8.9.11
- jGnash-2.2.0
- jgraphpad-5.10.0.2
- jmsn-0.9.9b2
- juel-2.1.2
- JXv3.2rc2deploy
- makagiga-3.4
- MegaMek-v0.34.3
- iFreeBudget-2.0.9
- mondrian-3.1.1.12687
- oddjob-0.26.0
- openxava-3.1.2
- pdfsam-1.1.3-out
- pjirc_2_2_1_bin
- pmd-bin-4.2.5
- proguard4.3
- smc_6_0_0
- squirrel-sql-3.0.1-base
- squirrel-sql-3.0.1-standard
- tvbrowser-2.7.3-bin
- villonanny-2.3.0.b02.bin
- rapidminer-4.4-community
- zk-bin-3.6.1

Systems hosted in other sites:

- ArgoUML-0.28
- GEF-0.13-bin
- Hl7Comm.1.0.1
- IRPF2009v1.1
- broker-4.1.5 (OurGrid)
- dbwrench
- ec2-api-tools
- ermodeller-1.9.2-binary
- flyingsaucer-R8
- gdata-src.java-1.31.1
- guice-2.0
- gwt-windows-1.6.4
- jai-1_1_4-pre-dr-b03-lib-linux-i586-08_Jun_2009
- jakarta-tomcat-5.0.28-embed
- juxy-0.8
- myjgui_0.6.6
- peer-4.1.5 (OurGrid)
- subethasmtp-3.1
- thinkui_sqlclient-1.1.2
- worker-4.1.5 (OurGrid)

B. *Networks from Other Domains*

- 5 friendship networks from Facebook [22]
- 3 electronic circuit networks [17]
- 4 word adjacency networks [17]
- 3 protein structure networks [17]
- 2 social networks of positive sentiment [17]
- 43 metabolic networks [23]
- Protein interaction network for yeast [24]
- Links between political blogs [25]
- Neural network of C Elegans [26]
- Network “beta3sreduced” (unknown source)
- Network “czech” (unknown source)
- Network “ecoli-metabolic” (unknown source).

Acknowledgment

Thanks to Dalton, Jorge, Christina, Garcia, Charles, Fabiola, Italo, Roberto, Jemerson, Sandra, and Lancichinetti.

References

[1] 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. th International Workshop on Program Comprehension IWPC '98*, 1998, pp. 45–52.

[2] N. Anquetil and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in *Proc. Sixth Working Conference on Reverse Engineering*, 1999, pp. 235–255.

[3] V. Tzerpos and R. C. Holt, "Acdc: an algorithm for comprehension-driven clustering," in *Proc. Seventh Working Conference on Reverse Engineering*, 2000, pp. 258–267.

[4] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 150–165, 2005.

[5] V. Tzerpos and R. C. Holt, "Mojo: a distance metric for software clusterings," in *Proc. Sixth Working Conference on Reverse Engineering*, 1999, pp. 187–193.

[6] B. S. Mitchell and S. Mancoridis, "Comparing the decompositions produced by software clustering algorithms using similarity measurements," in *Proc. IEEE International Conference on Software Maintenance*, 2001, pp. 744–753.

[7] R. Koschke and T. Eisenbarth, "A framework for experimental evaluation of clustering techniques," in *Proc. 8th International Workshop on Program Comprehension IWPC 2000*, 2000, pp. 201–210.

[8] O. Maqbool and H. A. Babri, "Hierarchical clustering for software architecture recovery," vol. 33, no. 11, pp. 759–780, 2007.

[9] R. A. Bittencourt and D. D. S. Guerrero, "Comparison of graph clustering algorithms for recovering software architecture module views," in *CSMR '09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 251–254.

[10] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, p. 509, 1999. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/9910332

[11] C. R. Myers, "Software systems as complex networks: structure, function, and evolvability of software collaboration graphs." *Phys Rev E Stat Nonlin Soft Matter Phys*, vol. 68, no. 4 Pt 2, p. 046116, Oct 2003.

[12] S. Valverde, R. Ferrer Cancho, and R. V. Solé, "Scale-free networks from optimal design," *Europhysics Letters*, vol. 60, pp. 512–517, Nov. 2002.

[13] B. Bollobs, C. Borgs, J. Chayes, and O. Riordan, "Directed scale-free graphs," in *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 132–139.

[14] T. Chen, Q. Gu, S. Wang, X. Chen, and D. Chen, "Module-based large-scale software evolution based on complex networks," *8th IEEE International Conference on Computer and Information Technology*, pp. 798—803, 2008.

[15] A. Lancichinetti and S. Fortunato, "Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities," 2009. [Online]. Available: http://arxiv.org/abs/0904.3940

[16] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks." *Science*, vol. 298, no. 5594, pp. 824–827, October 2002. [Online]. Available: http://dx.doi.org/10.1126/science.298.5594.824

[17] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon, "Superfamilies of evolved and designed networks." *Science*, vol. 303, no. 5663, pp. 1538–1542, March 2004. [Online]. Available: http://dx.doi.org/10.1126/science.1089167

[18] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[19] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-law distributions in empirical data," 2007. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0706.1062

[20] R. C. Holte, "Very simple classification rules perform well on most commonly used datasets," *Mach. Learn.*, vol. 11, no. 1, pp. 63–90, April 1993. [Online]. Available: http://dx.doi.org/10.1023/A:1022631118932

[21] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, "Empirical studies in reverse engineering: state of the art and future trends," *Empirical Softw. Engg.*, vol. 12, no. 5, pp. 551–571, 2007.

[22] A. L. Traud, E. D. Kelsic, P. J. Mucha, and M. A. Porter, "Community structure in online collegiate social networks," 2008. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0809.0690

[23] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A. L. Barabasi, "The large-scale organization of metabolic networks," *Nature*, vol. 407, no. 6804, pp. 651–654, October 2000. [Online]. Available: http://dx.doi.org/10.1038/35036627

[24] H. Jeong, S. P. Mason, A.-L. Barabasi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *NATURE v*, vol. 411, p. 41, 2001. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0105306

[25] L. A. Adamic and N. Glance, “The political blogosphere and the 2004 u.s. election: divided they blog,” in *LinkKDD ’05: Proceedings of the 3rd international workshop on Link discovery*. New York, NY, USA: ACM Press, 2005, pp. 36–43. [Online]. Available: http://dx.doi.org/10.1145/1134271.1134277

[26] D. J. Watts and S. H. Strogatz, “Collective dynamics of ’small-world’ networks.” *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998. [Online]. Available: http://dx.doi.org/10.1038/30918