Network Models in the Evaluation of Software Clustering Algorithms

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

***Abstract*—Software modularization recovery algorithms help to understand how software systems decompose into modules, but they**

Software modularization recovery algorithms... bla bla

***Keywords*-reverse engineering; software modularization; empirical evaluation; complex networks;**

In the context of our research, we consider classes as the components

I. Abstract

Software clustering algorithms automatically recognize a system's modular structure by analyzing its implementation. Due to the lack of well document software systems, though, the issue of testing these algorithms is still underexplored, limiting both their adoption in the industry and the development of better algorithms. We propose to rely on software models to produce arbitrarily large test sets. In this paper we consider three such models and analyze how similar the artifacts they produce are from artifacts from real software systems.

II. Introduction

One of the most obvious ways to reduce the time needed to develop or maintain a software system is to assign more developers to it. When combined with poor management, though, adding developers may in fact raise the development time, due to the increasing communication cost between developers [1]. Poor management may result in developers working on overlapping tasks, writing duplicated code, and inadvertently breaking the code written by another developer.

In order to take advantage of a team of developers, therefore, a key issue is the ability to decompose a system into weakly coupled modules, so that each module is maintained by a distinct group of developers [2]. Because of the weak coupling between modules, the need for communication between groups of developers is reduced, and so is the development time.

The ability of decomposing a system into modules depends on global knowledge about how its implementation components (e.g., functions, classes, variables) interact. Often, specially in legacy systems, this information is not readily available. Instead, it must be extracted from the system's source code. Even then the task of finding modules may be overwhelming because of the large number of components and interactions.

The task can be made easier by the use of software clustering algorithms, also known as also known as architecture recovery algorithms [3]. They automatically decompose a software system into weakly coupled modules by analyzing the network of dependencies between its components.

An important question for the adoption of a software clustering algorithm is whether the decomposition it finds for a system is similar to a reference decomposition for that system, i.e., a decomposition found by a group of experienced developers. To answer this question one needs to test the algorithm by applying it to a sample of systems with known reference decompositions, and then compare the two decompositions for each system by means of a metric such as MoJo [4] and EdgeSim [5].

Unfortunately there are few systems with known reference decompositions. Furthermore, because it is costly to obtain reference decompositions [6], there are few empirical studies about software clustering algorithms, most of them based on a couple of small and medium systems [7]–[9].

In this research we propose to evaluate software clustering algorithms by applying them to synthetic, i.e., computer generated, component dependency networks with built-in decompositions. With this approach it is feasible to test algorithms with arbitrarily large samples in a controlled manner.

Of course, it is desirable that the synthetic networks resemble networks extracted from real software systems. Hence, in this paper, we describe a study about three models that synthesize networks with built-in decompositions. We show empirically that all three models are capable of synthesizing networks that resemble software networks.

The remaining sections are organized as follows. .... Section 2, ...

III. Software Networks

The implementation of a software system can be viewed as a network of interacting components. In a object-oriented system, for example, classes interact with other classes through mechanisms such as inheritance and aggregation. By abstract the particular mechanisms of interaction, it is possible to build a network of dependencies between components, represented as a directed graph. This kind of network is the input for many software clustering algorithms [7], [10]–[12].

In this paper we will study networks of dependencies between classes in object-oriented systems. We will call any such a network a *software network*.

Network theory research studies general properties of many types of networks by using statistical analysis. In the last decade, it has been found that many networks arising from sociology, biology, technology and other domains present remarkable structural similarities. It has been shown that, in these networks, the distribution of vertex degrees is a power law, i.e., the number of vertices connected to k edges, N(k), is proportional to k^{gamma}, where γ is a positive constant. These networks are called scale-free networks.

Network theory has been applied to software networks and it was shown that they are also scale-free networks [13], [14].

IV. Network Models

Many models were proposed to explain the formation of scale-free networks. These models are simple algorithms that can be proven, either formally or empirically, to synthesize networks that are scale-free. Most models, though, do not account for modular structure, a feature present in some networks, in particular software networks. In this section we present three models that synthesize directed networks with built-in modular decompositions: BCR+, CGW and LR.

A. BCR+

The BCR model [15] aims to model the network of hyperlinks between web pages as a directed graph without modules. We have developed an extension to this model, called BCR+, that adds modules to the construction of the network. The model accepts the following parameters:

- number of vertices, n;
- a graph, G;
- three probabilities, p, q, and r, such as $p + q + r = 1$;
- a probability, mu;
- two real numbers, δ_{in} and δ_{out}.

In a network with modules, one can define a module dependency graph (MDG) as a graph where the two following conditions hold:

- each vertex represents a module in the original network;
- there is an edge from M_1 to M_2 in the MDG only if there is an edge from v_1 to v_2 in the original network, where $v_1 \in M_1$ and $v_2 \in M_2$.

The BCR+ model synthesizes networks whose MDG is equal to G by adding vertices and edges to an initial network until it reaches n vertices. The initial network is isomorphic to G: it contains one vertex for each module and one edge for each edge in G, connecting vertices whose corresponding modules are connected. Thus, the MDG for the generated network is equal to G from the very beginning.

After that, the algorithm consists of successive applications of one out of three operations on the network: (1) adding a vertex with an outgoing edge; (2) adding a vertex with an ingoing edge; (3) adding a edge between existing vertices. In operation 3, the edge can connect either vertices that belong to the same module or vertices in distinct modules.

The choice of the vertices that will be connected by a new edge, although non-deterministic, is not fully random. The probability that a particular vertex v is choosen, P(v), is proportional to a function of the in-degree or the out-degree of the node. We say that we choose a vertex within a set of vertices S according to a function f if

$$P(v) \;=\; \frac{\mathrm{f}(v)}{\sum_{v \in S} \mathrm{f}(v)}.$$

The denominator is a normalizing factor that assures that the probabilities sum to 1.

Before we present the algorithm in detail, using pseudo-code, we must define some functions:

- **in-degree(v)**: the number of edges that enter the vertex v;
- **out-degree(v)**: the number of edges that leave the vertex v;
- **module(v)**: the module to which the vertex v belongs;
- **out-neighbors(v)**: the set of all vertices that are connected to the vertex v by an edge starting in v.
- **same-module(v)**: the set of all vertices that are in the same module as the vertex v, except for v itself and vertices in out-neighbors(v).
- **other_modules(v)**: the set of all vertices that are in modules that are connected to v's module (in the graph G) by an edge starting in v's module, except for vertices in out-neighbors(v).

After the initial network is created, the algorithm proceeds as follows:

We can see that the probabilities p, q, and r control how often each operation is performed. Because the operation associated with probability r does not add any vertices, greater values of r imply more edges in the resulting network. It is easy to see that nodes connected to many ingoing edges are more likely to get another ingoing edge. The parameter δ_{in} can alleviate the handicap 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

```
while the number of nodes in the network is less than n:
  choose one of the operations according to probabilities (p, q, r)
  operation 1: (add a vertex with an outgoing edge)
    choose a vertex w within V according to f(x) = din + in-degree(x)
    add a new vertex, v, to module(w)
    add an edge from v to w
  operation 2: (add a vertex with an ingoing edge)
    choose a vertex w within V according to f(x) = dout + out-degree(x)
    add a new vertex, v, to module(w)
    add an edge from w to v
  operation 3: (add an edge between two existing vertices)
    choose a vertex v within V according to f(x) = dout + out-degree(x)
    choose a case according to probabilities (mu, 1 - mu)
    case 1: (distinct modules)
     choose a vertex w within other\_modules(v) according to f(x) = din + in-degree(x)
      add an edge from v to w
    case 2: (same module)
      choose w within same-module(v) according to f(x) = din + in-degree(x)
      add an edge from v to w
```

v_2 with in-degree 8. If $\delta_{in} = 0$, v_2 is twice more likely to receive a new incoming edge; if, otherwise, $d \in= 4$, v_2 is only $\frac{3}{2}$ more likely to receive the edge. The same reasoning applies to δ_{out}.

The parameter mu controls the proportion of edges between vertices in distinct modules. Lower values of mu lead to networks with weakly coupled modules.

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 network. 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 CITE.

B. CGW

The CGW model [16] 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 $\alpha \geq -1$

Just like BCR+, this is a growth model. Its initial network is composed of two vertices belonging to the same module and a directed edge between them. The remaining $m - 1$ modules are initially empty.

After that, the algorithm consists of successive applications of one out of four operations on the network, 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.

Whenever an edge is added from a vertex v to a vertex w (operations 1, 2, and 3), v is chosen randomly, and w is chosen according to the following probability function:

$$P_v(w) = \frac{f_v(w)}{\sum_{w \in S, f_v(w)}},$$

where

$$f_v(w) = \begin{cases} 1 + degree(w) \cdot (1 + \alpha) & \text{if } module(w) = \\ & = module(v) \\ 1 + degree(w) & \text{otherwise} \end{cases}$$

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, more strong is the tendency. For $\alpha = 0$ there is no bias, the two kinds of edges are equaly likely.

Because the original paper did not provide an implementation for the model, we now present the pseudo-code for our implementation:

Unlike BCR+, this model does not allow constraints on the connection between modules, however it accounts for the rewiring and the removal of edges.

C. LF

The LF model [17] is a very flexible model that can generated 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

```
operation 1: (adding a vertex with e1 outgoing edges)
  create a new vertex v and add it to a randomly chosen module
  repeat e1 times:
    choose a vertex w according to f\_v(w)
    add an edge from v to w

operation 2: (adding e2 edges)
  repeat e2 times:
    choose a vertex v randomly
    choose a vertex w according to f\_v(w)
    add edge from v to w

operation 3: (rewiring e3 edges)
  repeat e3 times:
    choose a vertex v randomly
    choose an edge e randomly within edges that start in v
    choose w according to f\_v(w)
    remove e
    add an edge from v to w

operation 3: (removing e4 edges)
  repeat e4 times:
    remove a randomly chosen edge
```

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 edges weights are discarded and the modules are non overlapping. 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).

V. Characterization of Software Networks

Our research hypothesis is that at least one of the presented models can synthesize networks that resemble software networks. A central issue, thus, is how to measure similarity between networks. In order to be useful, a similarity metric must be able to differentiate between software and non-software networks. In this section we present such a metric, together with an experiment that evaluates its usefulness by applying it to both software and non-software networks.

A. Similarity Between Networks

In a recent work, Milo et al. [18] 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 configuration of directed edges, as shown in Figure 1.

CITE igraph

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 distinct domains.

Following the work by Milo et al. [19], 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):

$$\mathrm{sim}(a, b) \;=\; \mathrm{cor}(\mathrm{TCP}(a), \mathrm{TCP}(b)),$$

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.

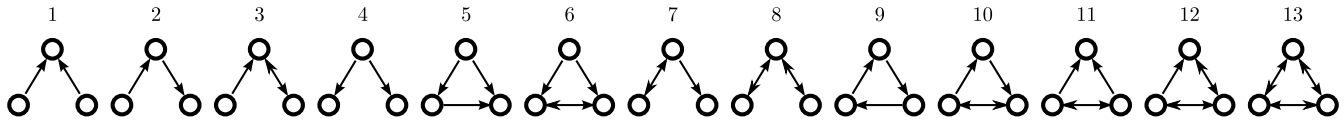

Figure 1. The 13 network triads.

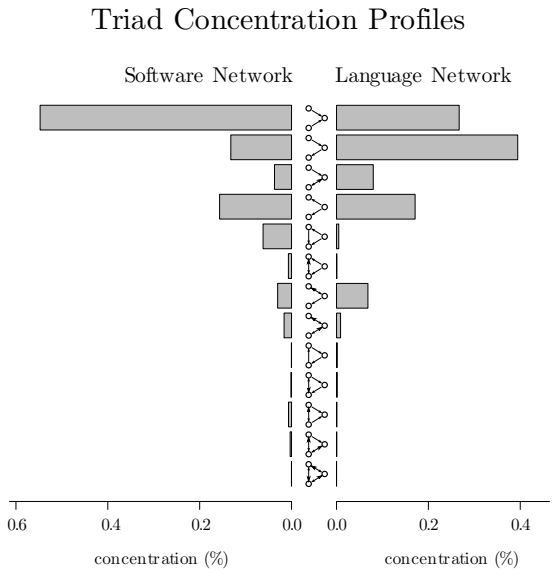

Figure 2. Triad concentration profiles (TCP) for two networks. On the left, network extracted from the software system JabRef (see the Appendix). On the right, word adjacency network for the Japanese language [19].

B. Data Set

To support the evaluation of the metric, we have collected 131 networks. 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 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. These networks are freely available on the Internet and have been used in previous researches. CITE

C. Evaluation of the Similarity Metric

In order to evaluate the similarity metric, we measured the similarity between the networks in the data set. A suitable 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:

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

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 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 common structural patterns

TODO: Boxplot by network class (software, metabolic, neural, social etc.)

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

Two networks, though, showed high S-score, above 0.83. The network of links between blogs on politics showed S-score 0.97. The neural network of the worm C. Elegans also showed a high S-score (0.88). Further investigation is needed in order to discover the reasons behind these high S-scores and whether auxiliary metrics can differentiate these networks from software networks.

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.

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

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, such a model can be evaluated by its precision and recall. Consider a 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, since 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 greater than the second greater non-software network S-score. With this value, we have both a high recall (95.4%) and a high precision (96.9%).

VI. 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 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. After that, we present rules for the choice of parameter values that lead to software-like networks.

A. Synthetic Data Set

We want to investigate if, with a proper choice of parameters, a model is capable of synthesizing a network that resembles software networks. 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. In this section we describe the combinations of parameters values used for each model.

B. BCR+ networks

We have chosen five different module dependency graphs, which where extracted from actual dependencies between archives of five different software systems of our sample: GEF (2 archives), iBATIS (4 archives), MegaMek (8 archives), findbugs (16 archives), and zk (32 archives).

For the remaining parameters, the following values were given:

- $p, q, r \in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, 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 after the creation of 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.

C. 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: $m \in \{2, 4, 8, 16, 32\}$.

Notice 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.

D. 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 [20], 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+	$\alpha \geq 0.7 \Rightarrow S$ $\alpha < 0.7 \Rightarrow N$	82.4%
CGW	$p_1 \geq 0.5 \Rightarrow S$ $p_1 < 0.5 \Rightarrow N$	82.3%
LF	$\gamma \geq 2.44 \Rightarrow S$ $\gamma < 2.44 \Rightarrow N$	78.9%

E. Results

Each synthesized network was classified as software-like or non software-like, using the classification model presented in section V-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 from data mining 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. We are interested in rules for networks classified as software-like and in the accuracy of these rules, 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% in all models.

VII. CONCLUSION AND FUTURE WORK

We have shown empirically that network models found in the literature can synthesize networks that resemble the network of 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, 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 studied with real networks [21]. After that we will be able to compare the results obtained by the 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 [19];
- 4 word adjacency networks [19];
- 3 protein structure networks [19];
- 2 social networks of positive sentiment [19];
- 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

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

REFERENCES

[1] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201835959

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[3] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, and H. Verjus, "Towards a process-oriented software architecture reconstruction taxonomy," in *Proc. 11th European Conference on Software Maintenance and Reengineering CSMR '07*, 2007, pp. 137–148.

[4] 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, moJo.

[5] 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, edgeSim and MeCl.

[6] 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, kosche-Eisenbarth (KE) measure.

[7] 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, precision, Recall.

[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] 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, bunch.

[11] 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.

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

[13] 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.

[14] S. Valverde and R. V. Sol, "Hierarchical small worlds in software architecture," no. Directed Scale-Free Graphs, 2003. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:cond-mat/0307278

[15] 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.

[16] 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.

[17] 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

[18] 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

[19] 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

[20] 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

[21] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Proc. 21st IEEE International Conference on ICSM'05 Software Maintenance*, 2005, pp. 525–535.

[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