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 modularization recovery 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. [1]

II. Introduction

Development of large-scale software systems is a challenge.

A key to success is the ability to decompose a system into weakly-coupled modules, so each module can be developed by a distinct team. Failing to do so results in duplicated code, non-parallelism, one's work impacting another's work etc.

The ability do modularize depends decisively on a vast knowledge about the system, how its different parts interact to accomplish the system's goal.

Unfortunately, in the case of legacy systems, such knowledge isn't available. Depending on its size, it might take months to understand the system so well as to find a good modularization. XXX [2]

POR ISSO SURGIRAM software modularization recovery algorithms, also known as software clustering algorithms or software architecture recovery algorithms. In its most common flavor, these algorithms analyze the dependencies between implementation components, such as classes, and then group them into modules such as there are few dependencies between classes in distinct modules.

Software modularization recovery algorithms can, therefore, do in minutes what a person would spend weeks or months. The question is: are the found modularizations good? Are they similar to what a person would find? To answer this question it's essencial to perform empirical evaluations envolving systems with known reference modularizations.

The empirical evaluations consist of selecting a collection of systems with known reference modularizations and then applying the algorithms to the systems. The modularizations found by the algorithms are then compared to the reference decompositions by a metric such as MoJo [3] or Precision-Recall.

Unfortunately there are few systems with known reference modularizations and, because to obtain reference modularizations is costly, there are few empirical studies, and most of them consider a couple of small and medium systems.

We therefore propose to use synthetic, i.e., computer-generated, software dependency networks, to evaluate software modularization recovery algorithms. These networks are generated by parametrizable models and have an embedded reference modularizations. The goal of an algorithm is, thus, to find modularizations that are similar to the reference modularization embedded in the network. With this approach we can CONTAR COM a large volume of test data that is composed of networks of different sizes and controllable characteristics.

Of course the success of this approach depends on the realism of the synthetic networks, ie, how well they resemble networks extracted from real software systems. In this paper we study three models and show that all of them are, by means of a careful parameter choosing, capable of producing realistic software networks.

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

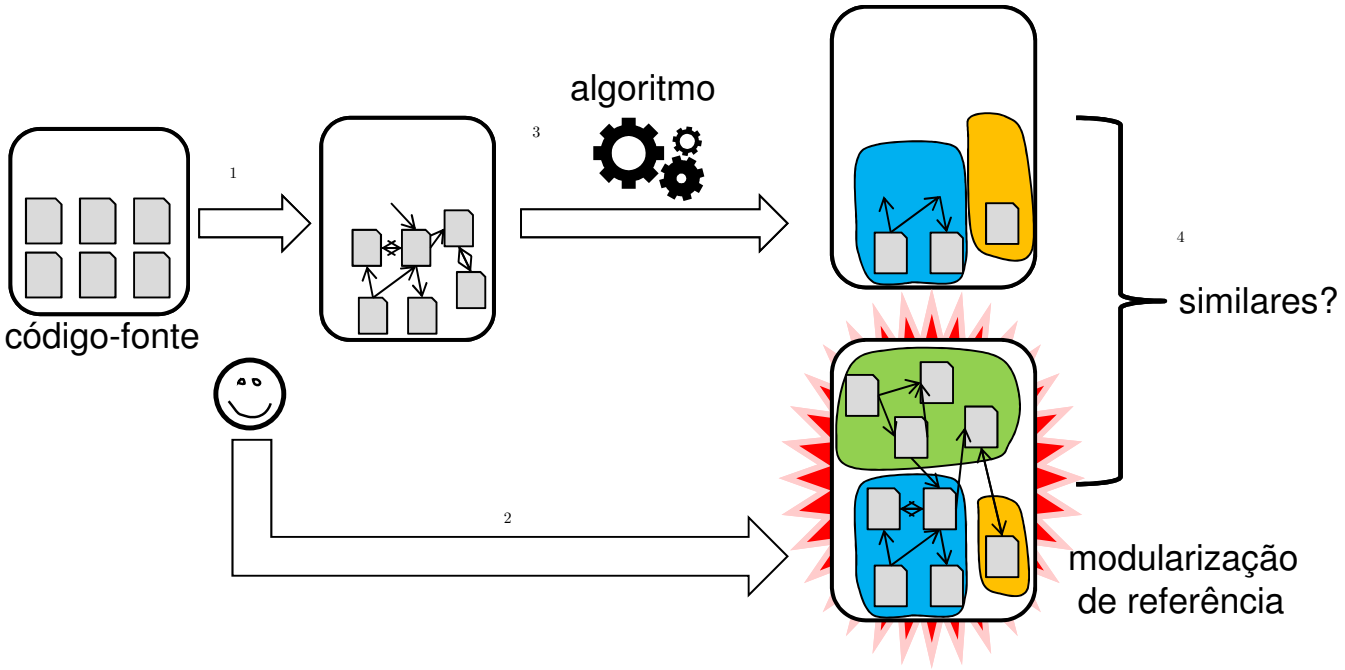


Figure 1. Evaluation of a software modularization recovery algorithm

III. Related Work

IV. Software Modularization Recovery: an Overview

Aka software architecture recovery, software architecture reconstruction, software clustering...

It's an task of reverse engineering, as defined by Tonella [4]...

DEFINITIONS

external edge...

directed graph, (un)weighted

V. Network Theory

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 number of vertices connected to k edges, N(k), is proportional to k^{gamma}, where γ is a positive constant. These networks were called scale-free networks.

Network theory has been applied to software networks and it was shown that they are also scale-free networks [5], [6].

(in fact we can't argue the degree distribution is perfectly fit by a power law. Anyway, the distribution is much more assymetric than normal ou Poisson's)

software indegree distribution (power law), outdegree distribution (not so power law)

VI. 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 generate networks that are scale-free.

Glossary: preferential attachment

In common: the were proven formally to generate scale-free directed networks.

A. BCR plus

We propose an extension to BCR model... which was proposed to model the links between web pages. [7]

The BCR model aims to model the network of hyperlinks between web pages as a directed graph without modules [7]. It was proven analitically to generate scale-free networks. We have developed an extension to this model, which we will call BCR+, that adds the concept of module. 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, din and dout.

In a network with modules, one can define a MDG as a graph where the two following conditions hold: * each vertex represents a module in the original network; * there is an edge from M1 to M2 in the MDG only if there is an edge from v1 to v2 in the original network, where v1 in M1 and v2 in M2.

The BCR+ model generates 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 are connected. Thus, the networks's MDG 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. 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 of the out-degree of the node. We say that we choose a vertex within a set V according to a function f if

P(choose v) = f(v) / sum(v in V, 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 explain some definitions. V is the set of vertices in the network being generated. This set grows as the algorithm is executed. Being v a vertex, we define the following three functions:

* out-neighbors(v): the set of all vertices that are connected to v by an edge starting in v. * same-module(v): the set of all vertices that are in the same module as v (except v itself). * 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.

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

while the network has less than n vertices: 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 w's module 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 w's module 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) - neighbors(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) - neighbors(v)) according to f(x) = din + in-degree(x) add an edge from v to w end

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, and the same reasoning can be applied to outgoing edges. The parameter din can alleviate the handicap by providing a "base in-degree" that is applied to all vertices when computing the probabilities. Consider two vertices, v1 with in-degree 4, and v2 with in-degree 8. If din = 0, v2 is twice more likely to receive a new incoming edge; if, otherwise, din = 4, v2 is only 3/2 more likely to receive the edge.

The parameter mu controls the proportion of edges between vertices in distinct modules. Lower values of mu lend to networks that are more modular.

The BCR+ model is a growth model, meaning that the network is generated 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 design methodologies CITE.

B. CGW

The CGW model was proposed to model the evolution of software systems organized in modules. It was proven formally to generate scale-free networks. [8] It accepts the following parameters:

* n, m * Probabilities p1, p2, p3, p4, summing 1 * Natural numbers e1, e2, e3, e4 * alpha

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. Because the original implementation of the model is not available, we describe in detail the algorithm we implemented:

while the network has less than n vertices: choose operation (p1, p2, p3, p4) operation 1: (adding a vertex with e1 edges) create a new vertex v and add it to a randomly chosen module do e1 times: choose a vertex w according to mbpa(v) add an edge from v to w operation 2: (adding e2 edges) do e2 times: choose a vertex v randomly choose a vertex w according to mbpa(v) add edge from v to w operation 3: (rewiring e3 edges) do e3 times: choose a vertex v randomly choose an edge e randomly within edges that start in v choose w according to mbpa(v) remove e add an edge from v to w operation 3: (removing e4 edges) do e4 times: remove and randomly chosen edge

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 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 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://.

VII. Experimental Setup

"softwareness"

Our hypothesis is that at least one of the presented models can synthesize networks that resemble software networks. We already know that they produce networks that, just like software networks, are scale-free. This single property, though, isn't enough to prove the hypothesis, since there are many known scale-free networks extracted from many distinct domains of knowledge. It is important to distinguish between software and non-software networks.

The approach we chose to test our hypothesis was to build a network classifier: an algorithm that analyses a network and classifies it as a software network or as a non-software network. If a classifier is capable of classifying with high accuracy both software and non-software networks, then we can apply it to synthetic networks and see if they are discernable from software networks. it gives a fair approximation apply it on synthetic networks. We expect that some synthetic networks will be so similar to software networks that they will be classified as software networks.

The experiment we conducted to evaluate the models, described in details in the following sections, can be summarized as such: * First of all, we collected both software and non-software networks * Then, we devised a classifier * After that, we applied the classifier to our data set and evaluated its accuracy, precision and recall. * After we came to an acceptable classifier, we synthesized many networks by varying the parameters on the models we presented. * Finally, we classified all synthesized networks as software or non-software.

After that we analysed the results and tried to understand which parameters of each model contribute mostly to the softwareness of their networks

A. Real Networks Data Set

For this experiment we have collected 65 software systems written in the Java programming language and 66 networks from other domains, such as biology, sociology and linguistics. Both the software systems and the networks are freely available on the Internet.

As of this writing, Java is a popular programming language, for which there is plenty of open source software CITE. Also, there are many dependency extractors for this language. We have extracted the dependency network for each software system using a tool called Dependency Finder CITE, for its easy integration with scripts. Dependency Finder looks for dependencies by means of static analysis of Java bytecode and outputs the dependency network in a XML file. Although this network is in a detailed level, showing dependencies between fine-grained implementation entities such as attributes and methods, we have abstracted it to a network of dependencies between classes. The network is directed and non-weighted. The size of the networks range from XXX to YYY.

The networks from other domains were collected in websites of complex network research groups. The domains include sociology, linguistics, biology and technology. The size of the networks range from XXX to YYY vertices.

B. Synthetic Data Set

In assessing the realism of a model, we want to know if, with a proper choice of parameters, the model is capable of producing a realistic network. In order to do so, we must generate many networks using different combinations of parameters.

Since most of the parameters can assume infinite distinct values, we chose to fix some of them and vary the others in discrete steps. In all models the number of vertices was fixed to 1000. We generated one network for each set of parameters. Here we describe the criteria we use to choose the parameters for each model.

1) BCR+: We chose five different module dependency networks, which where extracted from actual dependencies between modules of five different software systems of our sample, ranging from 2 to 32 modules. The module dependency networks are shown on Figure 2.

The probabilities p, q and r were given all possible values from 0.0 upto 1.0, in 0.2 steps, such as the sum of the probabilities was 1. Since the only events that create vertices are those associated with probabilities p and q, we imposed the additional restriction that p + q ¿ 0.

For deltain and deltaout we assigned the integer numbers from 0 to 4. TODO: why

Finally, we chose mu from 0.0 to 0.6 in 0.2 steps. It does not make sense to choose higher values since they mean that there will be more edges connecting different models.

Total: 9,500 networks.

2) LF: Like BCR, we choose mu ranging from 0.0 to 0.6 in 0.2 steps. For the remaining parameters we selected values from our sample of software systems. For degexp, ..., we picked the minimum, the median and the max values.

Total: 1,296 networks

3) CGW: p1,p2,p3,p4 ranging from 0.0 to 1.0 in 0.2 steps, with p1 ¿ 0, p1 + p2 + p3 + p4 = 1.0. e1,e2,e3,e4 in 1, 2, 4, 8 (except that when pi = 0, ei is ignored). 2*p4*e4 ¿= p1*e1 + p2*e2, so the number of edges created is at least twice the number of edges removed (to avoid long run times) alpha in -1, 0, 1, 10, 100, 1000. m in 2, 4, 8, 16, 32 (just like bcr and lfr)

Total: 38,790 networks

C. Classifier

In a recent work, Milo et al. proposed to distinguish networks of different domains by analyzing the triad con-

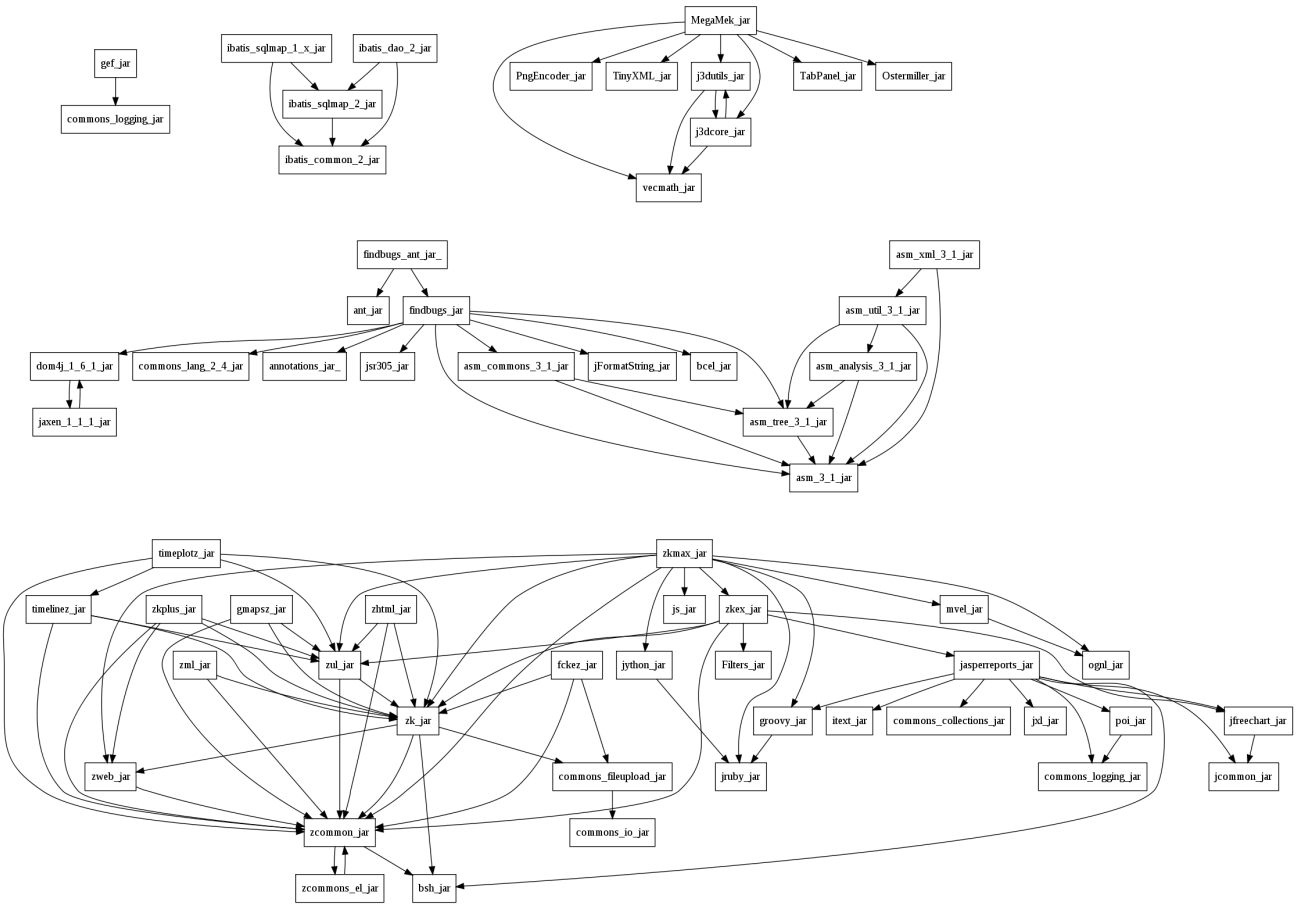


Figure 2. Module dependency graphs

centration of each network. A triad is simply 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 3. [9]

In order to characterize a network, one can count how many times each of the 13 triads appear on the network. The result is a triad concentration profile. Figure 1a show triads for a software system... Figure 2a show triads for network from domain X. 4

Then, in order to measure how similar two networks are, Milo et al computed Pearson's correlation coefficient between the two corresponding profiles.

CITE igraph

In order two measure the softwareness of a network, we measure the correlation between the network's profile and the profile of each of the 65 software networks in our sample. We then compute the average of these correlations and call it the S-score, that varies between -1 and 1. The greater the S-score, more likely that the network is a software network.

In order to build our classifier, we need a threshold for the S-score, so networks are classified as software networks only when they have S-score greater or equal to this threshold. We computed the S-score for each of the 65 software systems and found an average S-score of 0.97 with standard deviation of 0.03. Using the three-sigma rule from statistics, we chose the threshold as 0.97 - 3 * 0.3, that is 0.88.

The small standard deviation shows that triad concentration profiles indeed characterize well the software domain.

TODO: show histogram of S-scores for sw networks (x axis from -1.0 to 1.0) TODO: Is there need for cross validation in order to choose a threshold? - do cross-validation and choose threshold with max precision

1) Classifier Validation: In order to assess if the classifier is capable of distinguishing between software and non-software networks, we applied it to all the networks from our sample.

Most software networks were classified correctly, which is not surprising since the classifier was based on them.

Precision: X, recall: Y

Curiously, polblog had a very high S-score. This issue

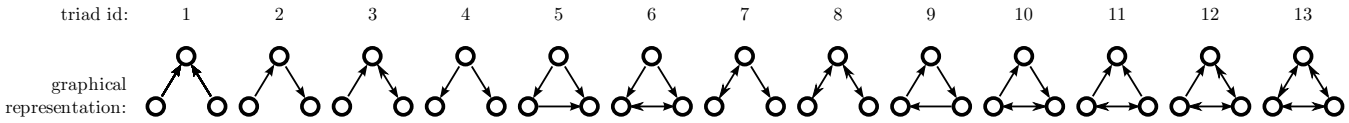
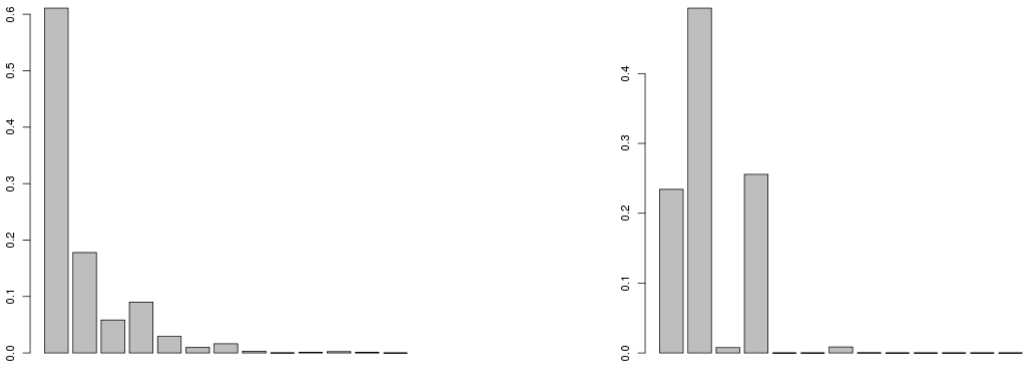


Figure 3. Triads (id vs. graph. reprs.)



(a) Software network

(b) Non-software network

Figure 4. Triad profiles

requires further investigation. Anyway, we cannot deny the possibility that it is really very software-like.

We could have chosen another value for the threshold, and it would affect the precision and the recall. For instance, had we chosen 0.82, we would end with 100% recall and x% precision. For us, though, it is more important to have a high precision, so we keep the threshold 0.88.

We also generated networks with the ER model, which is known to generate networks that are not scale-free and, thus, are very different from software networks. All the ER networks were classified as non-software.

VIII. Softwareness Evaluation

We used our classifier ...

Table: Model — Number of Networks — Realistic Networks — Percent %

Show some graphs: histogram of average correlations for each model. 5

All models produce networks that resemble software networks. For some parameters, though, the networks are not realistic.

A. Patterns in Parameters

1R

Naive Bayes

B. Homogeinity

Pick realistic networks from a model. Are they similar to each other? (see standard deviation) In other words: do the parameters make a difference?

Are they similar to networks generated by the other models? In other words: are the models equivalent?

IX. Threats to validity

Structural information isn't enough for a expert do produce a decomposition (s/he may use data such as names and external documentation).

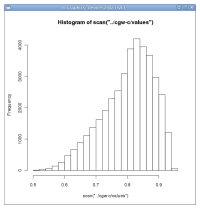
Even when considering only structural information, is it true that experts would find a decomposition similar to the reference decomposition imposed by the model?

We generated only one network for each set of parameters. (as we've shown, some parameters are redundant as they do not change significantly the realism)

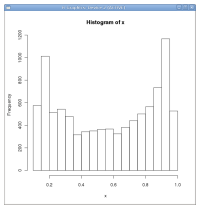
Some clustering algorithms use weights and they weren't studied here.

EV: We've only studied 65 systems, which is not that much.

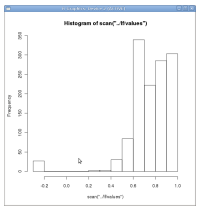
We only studied object-oriented systems implemented in Java. Maybe the results would be different if we studied systems implement in other languagens or using other paradigms. The choice of a particular technique for extracting dependencies (static analysis) may also have impact on the structure of the networks.



(a) Software network



(b) Non-software network



(c) Non-software network

Figure 5. Triad profiles

X. 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 [10]. After that we will be able to compare the results obtained by both approaches.

XI. Appendix A: list of networks

TODO: Show as Table: Name, Vertices, Edges

Software networks:

From SourceForge: 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

From other places:

ArgoUML-0.28 GEF-0.13-bin Hl7Comm.1.0.1 IRPF2009v1.1 broker-4.1.5 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 subethasmtp-3.1 thinkui_sqlclient-1.1.2 worker-4.1.5

Other networks:

3 circuit networks (circuit-s208 circuit-s420 circuit-s838)

5 facebook networks (facebook-Caltech36 facebook-Georgetown facebook-Oklahoma facebook-Princeton facebook-UNC28)

5 language networks (lang-english lang-french lang-japanese lang-spanish)

43 metabolic networks

3 protein networks (protein-a4j protein-AOR protein-eaw)

2 social networks (social-leader social-prison)

other networks:

polblogs

yeast

beta3sreduced celegansneural

czech ecoli-metabolic

XII. Conclusion

The conclusion goes here. this is more of the conclusion

Acknowledgment

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

References

[1] 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.

[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] 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.

[4] 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.

[5] 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.

[6] 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

[7] 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.

[8] 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.

[9] 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

[10] 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.