Three Algorithms for Analyzing Fractal Software Networks

Mario Locci, Giulio Concas, Roberto Tonelli, Ivana Turnu
Department of Electrical and Electronic Engineering
University of Cagliari
piazza d'Armi – 09123 Cagliari
ITALY
{mario.locci, concas, roberto.tonelli, ivana.turnu}@diee.unica.it
http://www.diee.unica.it

Abstract: - In this work we propose an algorithm for computing the fractal dimension of a software network, and compare its performances with two other algorithms. Object of our study are various large, object-oriented software systems. We built the associated graph for each system, also known as software network, analyzing the binary relationships (dependencies), among classes. We found that the structure of such software networks is self-similar under a length-scale transormation, confirming previous results of a recent paper from the authors. The fractal dimension of these networks is computed using a Merge algorithm, first devised by the authors, a Greedy Coloring algorithm, based on the equivalence with the graph coloring problem, and a Simulated Annealing algorithm, largely used for efficiently determining minima in multi-dimensional problems. Our study examines both efficiency and accuracy, showing that the Merge algorithm is the most efficient, while the Simulated Annealing is the most accurate. The Greeding Coloring algorithm lays in between the two, having speed very close to the Merge algorithm, and accuracy comparable to the Simulated Annealing algorithm.

Key-Words: - **Complex Systems, Complex Networks, Self-similarity, Software Graphs, Software Metrics, Object-Oriented Systems.**

1 Introduction

Software systems are characterized by being composed of *software modules*, which are related on each other. This characteristic holds irrespectively of the specific technology or language used for developing the system. In a system written in C or Fortran language the modules are the functions, which call each other, or the source code files, which include, and are included by, other files. In an object-oriented (OO) system, the modules can be classes and interfaces, source code files holding them, or packages, with decreasing granularity. Among OO classes and interfaces, many relationships are possible, such as inheritance, composition, dependency, instantiation, implementation.

A software system composed by modules, can be easily mapped to a graph, or a network, being graph nodes the software modules, and graph edges the relationships between modules. We will call *software network* such a graph. It is already well known that software networks have the characteristics of complex networks, i.e. are scale-free and small-world [1-4] [15]. A recent paper by Song et al. [5] demonstrated that the structure of complex networks can also be self-similar under a length-scale transformation, and showed how to calculate their *fractal dimension* using the "box counting" method.

This finding was applied to software networks computed on the classes and class relationships of large Smalltalk and Java systems, which were shown to exhibit a consistent self-similar behavior [6]. Moreover, a significant correlation seems to hold between the fractal dimension computed for various OO systems, and standard metrics related with software quality [6] [13] [14]. It is worth noting that the fractal dimension is just a single number that characterizes a whole network, and hence a whole software system, while complexity metrics are computed on every module of the system – think for instance to Chidamber and Kemerer OO metrics suite [7]. Obviously, the whole system can be characterized by some statistics computed on all modules, but this is not the same of having just one consistent, synthetic measure as with fractal dimension.

For this reason, we believe that the fractal dimension of software networks is a significant metric

describing the regularity of the software structure. It is therefore important to have efficient and reliable algorithms to compute it. In fact, as it will be shown in the following, the box counting algorithm is NP-complete, and its exact computation for large networks cannot be practically accomplished.

In this paper we recall the definition and meaning of fractal dimension in software networks, and then present and compare three different algorithms to compute it – Greedy Coloring, a Merge Algorithm devised by the authors, and Simulated Annealing – discussing the results.

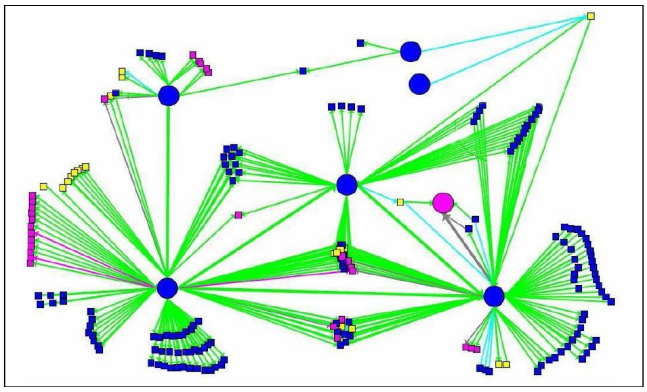


Fig. 1. Example of a portion of class graph for Eclipse.Classes, abstract classes and interfaces, as well as different relationships, have different colors.

2 The Fractal Dimension of Software Networks

2.1 Object-Oriented Systems as Networks

The basic building block of OO programming is the class, composed of a data structure and of procedures able to access and process these data. The data structure is made up of fields (instance or class variables) that represent the state of an object.

A class has also a behavior expressed in terms of methods that represent the procedures able to access and process the data structure. Classes may be defined at various levels of complexity, and are related across different kinds of binary relationships, such as inheritance, composition and dependence, which are well-known properties of OO design.

For the software systems considered in this work, but also in general, there is not unique way for building a software network. One first difference can be introduced at the vertex level, discriminating among simple classes, abstract classes, or interfaces.

Another difference can exist at the link level, for example distinguishing among directed or un-directed links. It is also possible to consider only particular kinds of links, like dependencies alone, and so on.

In this work we decided to consider only the classes (not abstract classes or interfaces) as vertex, and dependencies and inheritance as binary relationships.

Analyzing the source code of an OO system, it is possible to build its class graph —a graph whose nodes are the classes, and the graph edges represent directed relationships between classes (fig 1).

In this graph, the in-degree of a class is the number of edges directed toward the class, and is related to the usage level of this class in the system, while the out-degree of a class is the number of edges leaving the class, and represents the level of usage the class makes of other classes in the system. It has already been shown that OO software networks exhibit the scale-free and small-world properties, and thus can be considered complex networks. The in-degree distributions are power laws with exponent $\gamma \approx 2.5$ [1], [3], while the out-degree distributions are more controversial, and are mainly log-normal or Double-Pareto distributions [3], [8].

The dispute among log-normal or double Pareto (or even simple Pareto) is not purely academical. In fact it involves the validity and the suitability of different models of software process production, and presents potentially important practical implications on software quality and costs.

With regard to this paper the possible paths on the software graph, and thus the distances among nodes, may be different when considering the out-links or the in-links network. This may influence the final value of the fractal dimension. Thus we decided to consider only un-directed links.

2.2 Fractals

Before studying software networks scaling properties we need to introduce the basic concept of fractal dimension. Systems possessing fractal dimensions fill the space in a counterintuitive manner.

One of the easy-to-understand and well known

examples of fractals is the Cantor set.

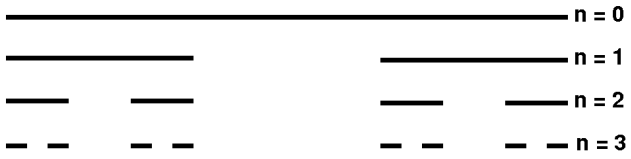


Fig. 2 Cantor set with three steps.

Let us consider a segment of unit length. For the sake of clarity, we label its left extreme with the abscissa zero, and its right extreme with the abscissa one. To obtain the Cantor set we delete pieces of this segment applying an infinite iterative process, and look at the remaining set of points (fig. 2).

At the first step we subtract the inner third of the segment (any other fraction works well). The remaining set are two segments of length one third each.

At the second step we subtract from each of these two segment their central third. The remaining set are four segments of length one ninth each.

According to this process, at the n^{th} step we obtain 2^n segments each of length 3^{-n} each, and the set's total length is $(2/3)^n$.

Let us examine the limit for large n. Clearly the number of segments grows to infinity, each segment becoming of zero length, namely a point. But such set is not a simple set of discrete points. In fact it is not countable. Actually it has the power-of-continuum, namely it is in a one-to-one correspondence with the original unit length segment.

The reasoning is the following. Consider the first step. The two segments may be identified with two numbers, zero for the left segment, and one for the right segment. For the segments at the second step we can add another binary digit, again zero for the left segment and one for the right segment. Thus the four segments at the second step are identified by two binary digits: 00, 01, 10 and 11, from left to right. In general, each segment at the n^{th} step may be identified by a sequence of n binary digits. In the limit of large n we can label each point of the remaining set by an infinite sequence of binary digits. But each point of the unit length segment is identified, in binary notation, by the same infinite sequences. In other words, all the points among zero and one may be coded in a fractional binary number. This provides a one-to-one mapping among the original segment and the remaining set of points. Apparently the subtraction of an infinite number of pieces does not modify the segment! Nevertheless they are clearly two different sets.

We can also consider another peculiarity. Let us calculate the length of all the subtracted sub-segments.

At the first step we subtracted one segment of length one third. At the second step we subtracted two other segments, each of length one ninth. In general at the n^{th} step we have to consider $2^{(n-1)}$ segments of length 3^{-n}. We obtain the following series providing the total length:

$$L_{tot} = \sum_{n=1}^{\infty} 2^{n-1}/3^n$$

This sum converges to one. Thus at the end we subtracted all the original segment's length! These are the paradoxes we have to face when dealing with fractal objects.

Even if the two sets are in a one-to-one correspondence, they substantially fill the available space in a different manner. Roughly speaking, the segment fills all the available space, while the Cantor set (the set of remaining points) leaves lots of holes: it is non-continuous. This is the key observation to define the fractal dimension.

In fact, if we partition the available space in cells, with varying sizes or diameters, we can note the difference among the two kinds of sets if we look at how many cells are filled or empty. In the segment case, regardless of the cells diameter, all the cells will be full. Thus the number of filled cells grows with the inverse of the cells diameter. In the case of the Cantor set this is not true, and the number of filled cells grows with a *fractional power* of the inverse of the diameter.

More formally, in the first case, we simply partition the segment in N_ε identical subsegments of diameter ε (in one dimension all the partition cells are simply segments, while in dimension D we can use D-dimensional cubes). The relationship among the two is $N_\varepsilon = 1/\varepsilon$, and thus the number of filled cells scales with the inverse power of the diameter. If we use a different segment length we obtain a multiplicating factor in front of the previous formula.

In two dimensions, instead of a segment we partition a unit square, using ε-side subsquares. Their number is given by $N_\varepsilon = 1/\varepsilon^2$, and scales with the second inverse power of the diameter.

Generalizing to dimension D, a unit D-cube will be partitioned by ε-side D-cubes, and their total number will be $N_\varepsilon = 1/\varepsilon^D$, scaling with the D inverse power of the diameter. All these powers are integer, that we are use to call "integer dimensions".

In the case of Cantor set, for the covering partition at step *n* we need $N_\varepsilon = 2^n$ cells of diameter $\varepsilon = 3^{-n}$. Then the scaling of N_ε with ε is:

$$N_\epsilon \sim \epsilon^{-\log(2)/\log(3)} = \epsilon^{-K}$$

with K < 1. This defines the fractal dimension of the Cantor set as *d = log(2)/log(3)*, and indicates the scaling of the number of non empty cells with diameter, or, in a sense, how many points fill the space in a range of the diameter, a concept that will be extended for the network's fractal dimension.

This way of proceed to calculate the fractal dimension is known as the box-counting method, since the partition divides the space into equal boxes.

The value log(2)/log(3) clearly indicates that Cantor set does not fill the space nor like a one dimensional segment, neither like a set of disjoint point, whose fractal dimension is zero.

The Cantor set, by construction, shows an important signature of fractal sets, the self-similarity: at any length scale the set looks the same. The structure reveals always the same details when viewed at different magnifications, and there is not lower limit, or something like "atomic" components, from which all the set is built.

While for mathematical sets this is exactly true at any length scale, for real objects the scaling regime and self-similarity hold only in a limited range of lengths, and finite size or granularity effects are revealed out of these limits.

2.2 Fractal Dimension of OO Networks

A recent study [5] found that the structure of complex networks is often also self-similar, and it is possible to calculate their fractal dimension using the box-counting method. As we saw, this method consists in covering the entire set, the network in this case, with the minimum number of boxes N_B of linear size (diameter) l_B. For a given network *G* and box size l_B, a box is a set of nodes where all distances l_{ij} between any two nodes *i* and *j* in the box are smaller than l_B. If the number of boxes scales with the linear size l_B following a power law (see eq. (1)), then d_B is the fractal dimension, or box dimension, of the graph [5]:

$$N_B(l_B) \sim l_B^{-d_B} \qquad (1)$$

The computation of the fractal coefficient of a network is thus a two-step one. First, an assessment of the self-similarity of the network has to be done, computing the minimum number of boxes covering the network, varying l_B from one to a given number, usually 10 or 20. This is the most computational intensive step. Then, one has to check whether $N_B(l_B)$ is linear in a log-log plot, showing a power-law behavior. This check is somewhat subjective, though it is possible to compute confidence intervals to this purpose. Eventually, an estimate of d_B is made fitting the plot with an LMS algorithm.

It has been already reported that OO software networks related to classes of large Smalltalk and Java systems show a patent self-similar behavior, with fractal dimension between 3.7 and 5.1 [6]. So, for OO software networks it is important to have efficient and reliable algorithms able to compute their fractal dimension.

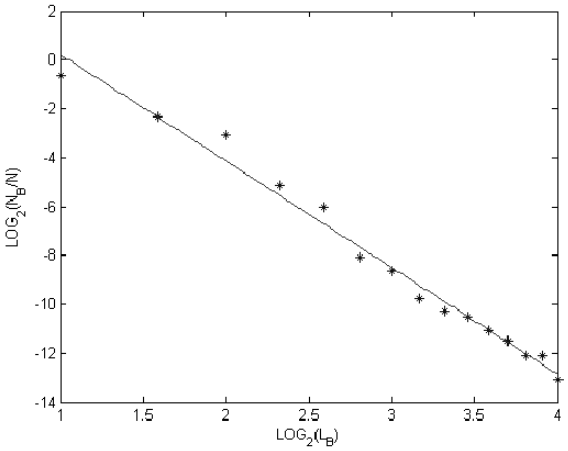
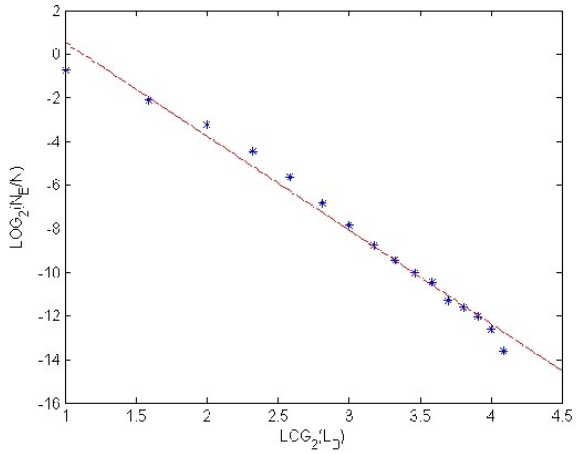


Fig. 3. Log-log plot of N_B vs. l_B for JDK 1.5.0.

Fig. 3 shows the box counting analysis of the software network related to JDK 1.5.0 Java system. The log-log plot of N_B vs. l_B reveals a self-similar structure. The slope of the fit is 4.24; this value is the fractal dimension d_B for JDK 1.5.0



.Fig. 4. Log-log plot of N_B vs. l_B *for Eclipse 2.1.3.*

In fig. 4 we report the box counting result for Eclipse 2.1.3, whose software graph has partially been shown in the first figure. Also for this system the log-log plot of N_B vs. l_B is clearly linear, and the system exhibits the power-law scaling. The slope provides 4.31 for the box-counting fractal dimension of this software network.

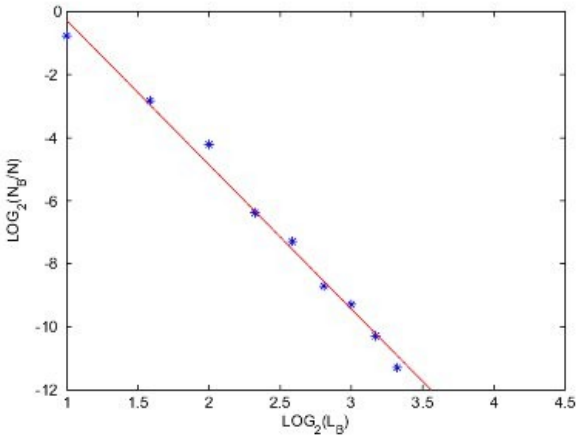


Fig. 5. Log-log plot of N_B *vs.* l_B *for VWorks 7.3.*

In fig. 5 we show the same plot for another software system, VWorks 7.3. Once again it shows a self-similar structure and a power-law relationship among N_B and l_B. Here the value provided by the box-counting method for the fractal dimension is 4.54.

2.3 Computing the Fractal Dimension

Song et. al. in their first paper [5] do not give details about how they actually computed the fractal dimension. Subsequently, Concas et al. shortly presented a simple algorithm for computing d_B [6]. Later, Song et al. demonstrated that this computational problem is equivalent to the graph coloring problem, and consequently took advantage of the many well-known greedy algorithms to perform this task [9]. Here we compare three algorithms both in terms of performance and precision – greedy coloring as in [9], a merge algorithm similar to that reported in [6], and simulated annealing, which is considered one of the best approaches to find the global minimum of difficult, multi-modal problems.

2.3.1 Greedy Coloring (GC)

Song et al. demonstrate that the box counting problem can be mapped to the graph coloring problem, which is known to belong to the family of NP-hard problems. Vertex coloring is a well-known procedure, where colors are assigned to each vertex of a network, so that no edge connects two identically colored vertexes [10]. We used the greedy algorithm described by Song et al. For this implementation we need a two-dimensional matrix c_{il} of size $N \times l_B^{max}$, whose values represent the color of node i for a given box size $l = l_B$. The algorithm works in the following way [9]:

(1) Assign a unique id from 1 to N to all network nodes, without assigning any colors yet.

(2) For all l_B values, assign a color value 0 to the node with id=1, i.e. $c_{il} = 0$.

(3) Set the id value $i = 2$. Repeat the following until $i = N$.

(a) Calculate the distance l_{ij} from i to all the nodes in the network with id j less than i.

(b) Set $l_B = 1$.

(c) Select one of the unused colors $c_{jl_{ij}}$ from all nodes $j<i$ for which $l_{ij} \geq l_B$. This is the color c_{jl_B} of node i for the given l_B value.

(d) Increase l_B by one and repeat (c) until $l_B = l_B^{max}$..

(e) Increase i by 1.

This greedy algorithm is very efficient, since it can cover the network with a sequence of box sizes l_B performing only one network pass.

The main steps are illustrated in fig. 6, where the example shows the case $l_B = 3$. Starting from the network G representing the software graph, we build the dual network G', obtained from G connecting two nodes when their distance, calculated in the original graph G, is larger than, or equal to, l_B. Next we use a greedy algorithm for vertex coloring in G'. Then we go back to the original network determining the box covering on G.

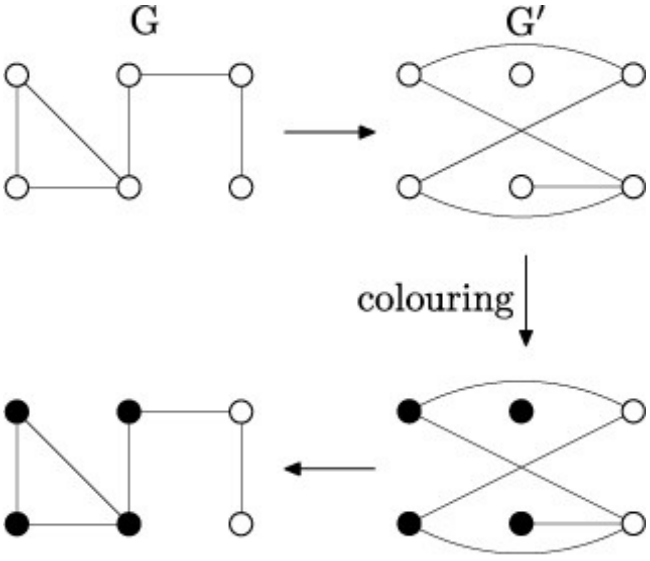


Fig. 6: Construction of the dual network G' for a given box size (here $l_B = 3$), where two nodes are connected if they are at a distance $l \geq l_B$. We use a greedy algorithm for vertex coloring in G', which is then used to determine the box covering in G, as shown in the plot.

2.3.2 Merge Algorithm (MA)

This method is based on the union of two or more clusters into a third one. Two clusters are merged if the distance between them is less than l_B. MA uses the configuration at l_B to obtain the starting point for the successive aggregation at $l_{B+1} = l_B + 1$.

In the initial configuration each cluster c_k contains only a node, so each node is marked with a different label. Let n be the number of nodes of the network, and l_{max} the maximum value for l_B. The algorithm works in the following way:

```
l_B = 2;
C ≡ {c_1, c_2 , c_3 ,..., c_n};
while l_B ≤ l_max;
    D ≡  Φ;
    repeat
        get a random cluster c_k from C;
        C' ≡ {c_j ∈ C| d(c_k,c_j) ≤ l_B};
        get a random cluster c_i from C';
        ĉ = merge(c_k,c_j);
        C = C - {c_k,c_j};
        D = D ∪ {ĉ};
    until size(C) < 2 or C'= Φ  ∀c ∈ C;
    D = D ∪ C;
    N_B = size(D);
    l_B := l_B +1;
    C = D;
end while;
```

In order to find the set **C'** we use an efficient burning algorithm to determine in a single step all clusters belonging to **C'**.

2.3.3 Simulated Annealing(SA)

The MA described above is an efficient method to estimate the fractal dimension, and the base for Simulated Annealing algorithm. SA is a class of algorithms inspired by the annealing process in metallurgy [11]. In the SA context, a box partition (box covering) is the state S of the physical system and the number of boxes N_B is the "internal energy" in that state.

In order to consider a neighbor state S' of the current state S we compute three fundamental operations:

- movement of nodes;
- creation of new clusters;
- union of clusters.

If S' is a solution worse that S, there is a probability to accept the state S' even if it has the energy $E(S') > E(S)$.

A new state or partition with boxes of size l_B is obtained from the current state by moving nodes and

merging clusters. Let A and B be two generic clusters of the current partition. We define the following operations:

- **movement**: a node is moved from A to B if B diameter doesn't exceed l_B, and A includes at least two nodes;
- **creation**: a node is taken from cluster A to form a new cluster;
- **merge**: all clusters are merged by using the merge algorithm described in section 2.3.2..

At each "temperature" we perform k_1 movements and k_2 creations of nodes, and a single merge of all clusters by using MA. We always accept a better or equal solution, while we accept a solution S' worse than S with probability:

$$p = e^{-\frac{E(S')-E(S)}{T}} \quad (2)$$

At each step the system is cooled down to a lower temperature $T_0 = cT$, where $c < 1$ is the cooling constant. The typical starting temperature T is about 0.6 and the typical values of k_1 and k_2 are 5000 and 5, respectively. Similar values are used by Zhou et al. in their implementation of the SA algorithm [12]. In deeper detail, the algorithm works in the following way:

```
create first configuration S using MA
for j (j = 1, 2, ....,k3) do
    move k1 nodes;
    create k2 new clusters;
    if E(S') ≤ E(S) then
        S := S'
    else
        get a random number RND;
        if RND < exp(-(E(S')-E(S))/T) then
            S := S'
        endif;
    endif;
    merge clusters using MA algorithm;
    T := cT;
endFor;
```

We perform about 5000 steps at each temperature, and then reduce T. The number of outer cycles (temperature reductions) is k_3, and it is set to 20, with cooling constant c set to 0.995 [12].

3 Results

We implemented in Java the three algorithms and compared their performance in terms of speed and quality of the result. In fact, being the box partitioning problem NP-complete, on large networks its exact solution is not feasible. Consequently, it is not enough to have a fast algorithm to compute the box partitioning, but the results must be trusted, in the sense that the partitioning found should be close enough to the global minimum to guarantee the consistency of the results. We tested the goodness of the results by repeatedly running the same algorithm, selecting randomly the initial configuration. We then checked the variance of the resulting estimate of $N_B(l_B)$ for various values of l_B, which in turn depends on the number of boxes found in each partitioning.

We used for the tests the software network related to Java JDK 1.5 system, which includes the standard Java libraries and development tools. The JDK network has 8499 nodes and 42048 edges, so it can be considered a quite large network.

3. Execution speed

We computed the execution speed on the whole computation of d_B, which is what actually matters, running the three algorithms starting from random configurations of the initial box partitioning and performing 100 times the computation. The results for a PC with Windows XP and a processor Intel Core 1.4 GHz are reported in Table 1and Table 2.

Table 1. Average execution times for d_B computation on JDK 1.5 class graph (8499 nodes and 42048 links).

Algorithm	Time (s)	d_B
GC	410	3.96
MA	289	4.24
SA	8807	4.06

Table 1I. Average execution times for d_B computation on the E. Coli protein interaction network graph (2859 nodes and 6890 links).

Algorithm	Time (s)	d_B
GC	13	3.44
MA	21	3.57
SA	1177	3.47

As you can see, the most efficient algorithm is MA, and this is confirmed also by other test runs on other networks, not reported here for the sake of brevity. GC is *still* very efficient, while SA is much worse as regards execution speed, being at least one order of magnitude slower.

It has to be pointed out that these results may depend not only from the system analyzed, but also from the particular release. In fact, changes of the software structure among various releases are reflected in changes of their fractal dimension. Depending on the system, these changes may be more or less pronounced.

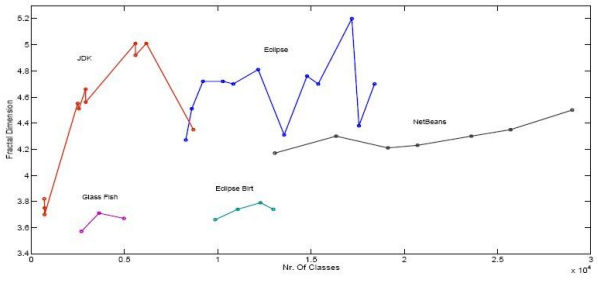


Fig. 7. Fractal dimension for different versions of the analyzed systems, as a function of the release version, according to the number of classes of each version.

In fig. 7 we report the results of the fractal dimension for various versions of different software systems. It can be noted that, for Glass Fish, Eclipse Birt, Netbeans, the fractal dimension is quite stable among releases, while for JDK and Eclipse, it shows major variations. This is an index of major topological changes in the structure of the graph associated to these systems, and, consequently, of major modifications applied onto the software.

Regarding the quality of results, they look similar but not exactly the same. This is discussed in detail in the next section.

3.2 Result Quality

We computed the reliability of the three tested algorithms by testing for their repeatability in 1000 runs on a smaller network than the whole JDK 1.5

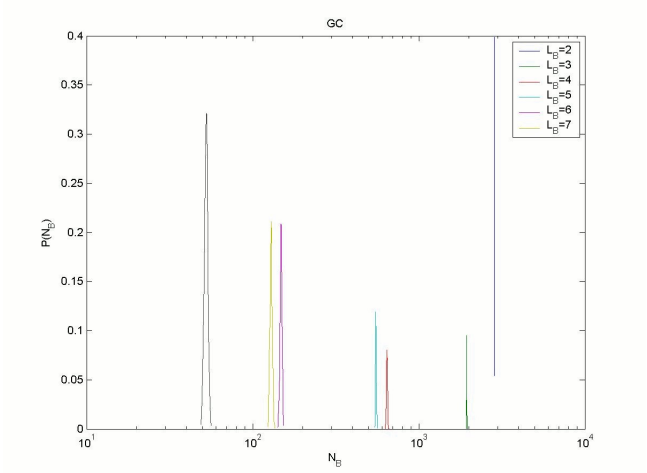


Fig. 8. Empirical distributions of the values of N_B for six values of l_B, for GC algorithm run 1000 times.

software graph, the E. Coli protein interaction network [5]. This network has 2859 nodes and 6890 edges. We varied l_B, from 2 to 7. Figs. 8, 9 and 10 show the empirical distributions of the values of N_B for each value of l_B, and for GC, MA and SA algorithms, respectively.

As you can see, GC and SA algorithms show a very small dispersion of the resulting values of N_B, showing that both are highly reliable. On the other hand, the results of Fig. 8 regarding MA algorithm show a much higher dispersion. Consequently, despite its high performances, we deem that MS algorithm is not suitable for the computation of software networks fractal dimension.

We report in Fig. 11 the standard deviation of the computed N_B for the three algorithms, for eight values of l_B, from 2 to 9. Fig. 11 confirms the previous results on the reliability of the three algorithms.

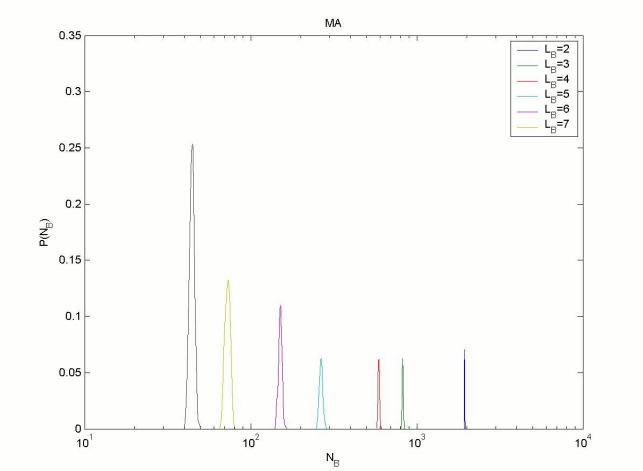


Fig. 9. Empirical distributions of the values of N_B for six values of l_B, for MA algorithm run 1000 times.

The standard deviation of MA results is consistently higher than that of GC and SA. The latter algorithms are quite similar, with a slightly better average performance of SA over GC on the eight test values of l_B.

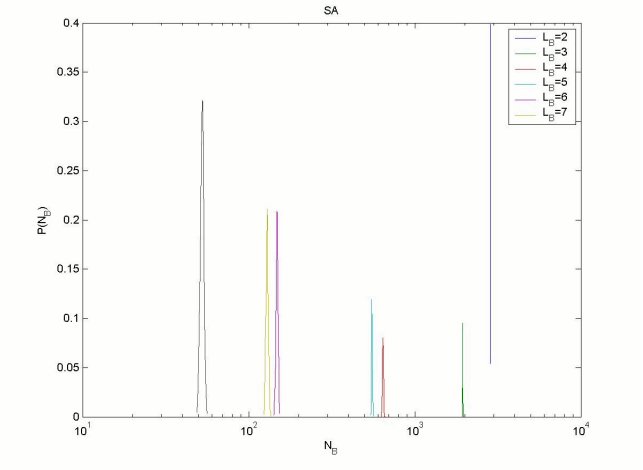


Fig. 10. Empirical distributions of N_B for six values of l_B, for SA algorithm run 50 times.

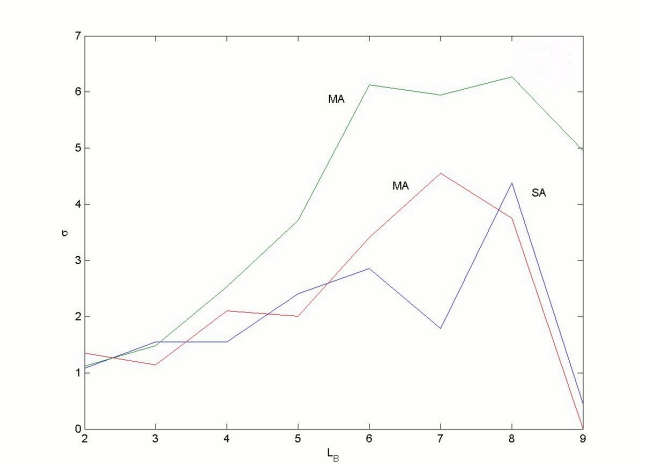


Fig. 11. Standard deviations of the values of N_B for eight values of l_B, for MA algorithm run 1000 times.

4 Conclusion

The fractal dimension of software networks has the potential to be a significant, synthetic metric describing the regularity of the structure of a software system, and moreover it has been proven to be correlated to source code quality metrics of OO systems. It is therefore important to have efficient and reliable algorithms to compute it.

In this paper we presented three different algorithms to compute the fractal dimension of networks, which to our knowledge cover all the approaches proposed in literature. These algorithms – Greedy Coloring, Merge Algorithm, and Simulated Annealing, have been described and compared using the software network related to Java JDK 1.5 open source system and, for the purpose of assessing the algorithm reliability, also using a smaller protein interaction network.

We found that SA is the best algorithm in terms of precision, but it is by far the worst in terms of speed. The time performance of MA is better than GC for large networks but the greedy coloring produces more precise solutions. In conclusion, the Greedy Coloring algorithm, based on the equivalence of the box counting problem with the graph coloring problem, looks the best compromise, having speed comparable to MA, and accuracy comparable with SA.

References:

[1] S. Valverde, R. Ferrer-Cancho, and R. Sole´, Scale-Free Networks from Optimal Design. *Europhysics Letters*, vol. 60, 2002, pp. 512-517.

[2] C. Myers, "Software Systems as Complex Networks: Structure, Function, and Evolvability of Software Collaboration Graphs". *Physical Rev. E*, vol. 68, 2003.

[3] G. Concas, M. Marchesi, S. Pinna, and N. Serra, Power-Laws in a Large Object-Oriented Software System, *IEEE Transactions on Software Engineering*, vol. 33, No. 10, 2007, pp. 687-708.

[4] P. Louridas, D. Spinellis and V. Vlachos, Power Laws in Software. *ACM Trans. Software Eng. and Method.*, Vol. 18, No. 1, 2008.

[5] C. Song, S. Havlin and Makse H. A., Self-similarity of complex networks, *Nature*, vol. 433, pp. 392-395, and related supplementary information, 2006.

[6] G. Concas, M. Locci, M. Marchesi, S. Pinna, and I. Turnu, Fractal dimension in software networks, *Europhysics Letters*, vol. 76, 2006, pp. 1221-1227.

[7] S. Chidamber, and C. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Trans. Software Eng*., vol. 20, no. 6, pp. 476-493, June 1994.

[8] G. Concas, M. Marchesi, A. Murgia, R. Tonelli, I. Turnu, Stochastic models of software development activities, *submitted for publication*.

[9] C. Song, L.K. Gallos, S. Havlin, H. A. Makse, How to calculate the fractal dimension of a complex network: the box covering algorithm, *Journal of Statistical Mechanics*, P03006, 2007.

[10] D.W. Matula, G. Marble and J.D. Isaacson, Graph Coloring Algorithms. In *Graph Theory and Computing* (Ed. R. Read). New York: Academic Press, pp. 109-122, 1972.

[11] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing. *Science,* vol. 220, 1983, pp. 671-680.

[12] W.X. Zhou, Z.Q. Jiang, D. Sornette, Exploring self-similarity of complex cellular networks: the edge-covering method with simulated annealing and log-periodic sampling. *Physica A* vol. 375, No. 2, 2007, pp. 741-752.

[13] L. Lazic, S. Popovic, N. E. Mastorakis, A Simultaneous Application of Combinatorial Testing and Virtualization as a Method for Software Testing. *WSEAS Trans. on Inf. Sci, and App.*, Vol. 6, 2009.

[14] L. Lazic, N. E. Mastorakis OptimalSQM: Integrated and Optimized Software Quality Management. *WSEAS Trans. on Inf. Sci, and App.*, Vol. 6, 2009.

[15] V. Podgorelec, Improved Mining of Software Complexity Data on Evolutionary Filtered Training Sets. *WSEAS Trans. on Inf. Sci, and App.*, Vol. 6, 2009.