Computing the Fractal Dimension of Software Networks

Mario Locci, Giulio Concas, Ivana Turnu
Department of Electrical and Electronic Engineering
University of Cagliari
piazza d’Armi – 09123 Cagliari
ITALY
{mario.locci, concas, michele}@diee.unica.it
http://www.diee.unica.it

Abstract: - Given a large software system, it is possible to associate to it a graph, also known as software network, where graph nodes are the software modules (packages, files, classes or other software entities), and graph edges are the relationships between modules. A recent paper by some of the authors demonstrated that the structure of software networks is also self-similar under a length-scale transformation, and calculated their fractal dimension using the “box counting” method. In this paper we describe three possible algorithms for the computation of the fractal dimension of software networks, and compare them. We show that a Merge Algorithm first devised by the authors is the most efficient, while Simulated Annealing is the most accurate. A Greedy Coloring algorithm, based on the equivalence of the box counting problem with the graph coloring problem, seems nevertheless the best compromise, having speed comparable to the Merge Algorithm, and accuracy comparable with Simulated Annealing.

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]. 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]. 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.

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.

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. 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 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.

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. This method consists in covering the entire network with the minimum number of boxes \( N_B \) of linear size \( l_B \). 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]:

\[
N_B(l_B) \sim l_B^{-d_B}
\]  (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.

![Log-log plot of \( N_B \) vs. \( l_B \) for JDK 1.5.0.](image)
Fig. 1 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.

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. $c_{i1} = 0$.
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 $C_{j|i}$ from all nodes $j<i$ for which $l_{ij} \geq l_B$.
   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.

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:

$\hat{l}_B = 2$;
$C = \{ c_1, c_2, c_3, \ldots, c_n \}$;
while $l_B \leq l_{max}$;
   D = Φ;
   repeat
      get a random cluster $C_k$ from $C$;
      $C' = (C_j \in C \mid d(C_k, C_j) \leq \hat{l}_B)$;
      get a random cluster $C_1$ from $C'$;
      $\hat{c} = merge(C_k, C_1)$;
      $C = C - \{ C_k, C_j \}$;
      $D = D \cup \{ \hat{c} \}$;
      until size($C$) < 2 or $C' = \Phi \ \forall C \in C$;
   D = D $\cup$ C;
   $N_B = size(D)$;
   $\hat{l}_B := \hat{l}_B + 1$;
   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 than \( S \), 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:

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

At each step the system is cooled down to a lower temperature \( T_0 = cT \), where \( c \) 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:

```java
create first configuration \( S \) using MA
for j \( (j = 1, 2, \ldots, k_3) \) do
  move \( k_1 \) nodes;
  create \( k_2 \) new clusters;
  if \( E(S') \leq E(S) \) then
    \( S := S' \)
  else
    get a random number RND;
    if RND < \( \exp(-\frac{E(S')-E(S)}{T}) \) 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.

### 3.1 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 1.
Table 1. Average execution times for $d_B$ computation on JDK 1.5 class graph.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (s)</th>
<th>$d_B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>410</td>
<td>3.96</td>
</tr>
<tr>
<td>MA</td>
<td>289</td>
<td>4.24</td>
</tr>
<tr>
<td>SA</td>
<td>8807</td>
<td>4.06</td>
</tr>
</tbody>
</table>

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.

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 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. 2, 3 and 4 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. 3 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. 5 the standard deviation of the computed $N_B$ for the three algorithms, for eight values of $l_B$, from 2 to 9. Fig. 5 confirms the previous results on the reliability of the three algorithms. 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 $l_B$. 
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: