Surpriser: Manual

D. Gamermann[a]

[a]Department of Physics, Universidade Federal do Rio Grande do Sul (UFRGS) - Instituto de Física , Av. Bento Gonçalves 9500 - Caixa Postal 15051 - CEP 91501-970 - Porto Alegre, RS, Brasil.

Abstract

Instructions for the use of package `Surpriser` to study graph communities.

1 Introduction

The main methods and functions in the package are programmed in C and callable through python via the python C API. Upon installation, the C code and functions are compiled. The process is straightforward in Linux (given that the `python-dev` packages are properly installed in the system), but less experienced Windows users might have difficulties. If you do science, consider using Linux and open source software.

This package was programmed in order to study the community structure of complex networks, using as quality parameter the surprise [1].

A network is a set of K nodes between which one finds n links. Mathematically, we describe our networks as symmetric $K \times K$ matrices whose elements ij are either 0 or 1, indicating whether node i has connection with node j.

The network's nodes may be divided into communities. A community is a subset of the nodes. The network community structure is defined by a partition: to every node in the network a number between 1 and N_c is assigned[1], indicating to which community it belongs. Here N_c is the total number of communities of the network.

The surprise is a measure that depends on the partition of the nodes of a network into communities. It is defined as[2] [1]:

$$S = -\ln \sum_{j=p}^{\min(M,n)} \frac{\binom{M}{j}\binom{F-M}{n-j}}{\binom{F}{n}} \tag{1}$$

where n is the total number of links in the network, F is the total number of possible links given the value of K (links in a complete graph with K nodes: $K(K-1)/2$), p is the number of links inside communities given the community partition and M is the total number of possible links inside communities given the partition. If C_i is the number of nodes in community i, then

[1]Actually, python starts counting elements from zero, not one. So, when in the text we refer to community 1, in python it would be community number 0.

[2]We use and programmed everything using the natural logarithm (ln), though when first defined by its authors, the surprise was evaluated using base 10 logarithm (log).

$$M = \sum_{i=1}^{N_c} \frac{C_i(C_i-1)}{2}. \quad (2)$$

Even for small networks, the number of different ways into which one may group its nodes into any given number of communities is huge (a complex combinatorics problem), and which is the best way to divide the graph's nodes into communities is an open problem in the field. The standard way to approach it is to define a quality function and use some heuristics in order to find a partition that maximizes this function. The most popular one is the modularity, but this has several drawbacks and pitfalls [2, 3]. We developed this package in order to maximize the surprise, instead.

The package is composed of four modules: `surprise`, `data`, `randoms` and `benchmark`. In the following sections we explain the objects and functions found in each module and how to use them with examples. The basic object programmed, which we call `Surpriser`, is found in the `surprise` module. We describe in the next section its attributes and methods.

2 The `surprise` Module

In this section we describe the `Surpriser` object, its initialization, attributes and methods and the other functions found in the `surprise` module.

2.1 `Surpriser` Initialization and Attributes

The basic element of the package is the `Surpriser` object. This will contain all the graph's information and its community structure. In order to create it, just call `Surpriser` with the M matrix as argument[3]:

```
>>> from Surpriser.surprise import *
>>>
>>> M = [[0, 1], [1, 0]]
>>> sur = Surpriser(M)
>>> print sur
< Graph/communities info :
   Number of nodes       : 2
   Number of links       : 1 (1)
   Number of communities : 2
   Number of intralinks  : 0 (0)
   Surprise              : 0.000000 >
```

In the above example, a trivial network with two connected nodes is created. By default, since no community structure was given in the `Surpriser` call, each node was allocated in its own community. This can be seen by the `show_communities` method.

```
>>> sur.show_communities()
community 0 (size 1) :  0,
community 1 (size 1) :  1,
```

[3] In the code snipets in this manual, python 2 syntax will be used. Small adaptions might have to be made in order to execute them in python 3, like instead of `print sur` to use `print(sur)`.

Let's create the object for the network in figure 1, which has had its community structure studied in [4]. From the figure, intuitively, one would associate two 4 nodes communities (the 4 nodes cliques in the extremes) and maybe assign another community to the 3 nodes path connecting the cliques.

In the code below, its M-matrix and `Surpriser` object is created, already making the above mentioned supposition about the network's community structure (from now on, unless strictly necessary, we will suppress the `import` commands from the examples).

```
>>> M = [[0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
...      [1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
...      [1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0],
...      [1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
...      [0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
...      [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0],
...      [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
...      [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1],
...      [0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1],
...      [0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1],
...      [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0]]
>>> communities = [ [0, 1, 2, 3], [4, 5, 6], [7, 8, 9, 10] ]
>>> sur = Surpriser(M, communities)
>>> print sur
< Graph/communities info :
   Number of nodes       : 11
   Number of links       : 16 (55)
   Number of communities : 3
   Number of intralinks  : 14 (15)
   Surprise              : 21.653068 >
>>> sur.show_communities()
community 0 (size 4) :  0, 1, 2, 3,
community 1 (size 3) :  4, 5, 6,
community 2 (size 4) :  7, 8, 9, 10,
```

Note that, in order to create the object with a preestablished community structure, in the `Surpriser` call, one must include an optional argument with a list where each element is another list with all nodes in a given community.

In the representation of the `Surpriser` object, one can see its attributes, shown below with the attributes names in place

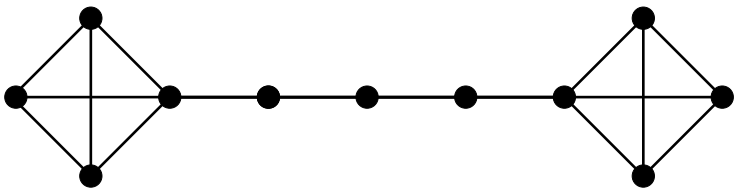

Figure 1: Network used in the example.

of their values and shown from the object:

```
< Graph/communities info :
   Number of nodes       : K
   Number of links       : nl (F)
   Number of communities : Nc
   Number of intralinks  : p (M)
   Surprise              : surprise >
>>> sur.K
11
>>> sur.nl
16
>>> sur.F
55
>>> sur.Nc
3
>>> sur.p
14
>>> sur.M
15
>>> sur.surprise
21.653067988321435
```

2.2 Methods

Most methods of the `Surpriser` object transform the community structure of the network. There are three basic ways to transform the communities:

- Merge → two communities may be merged into a single one.
- Exchange → an element of a given community is transferred to another community.
- Extract → an element of a given community is extracted (becomes its own community).

The three methods that control these basic operations are `merger`, `exchanger` and `extractor`. The arguments in the call of these methods are all optional, and they basically specify which community/ies should be tried, how many times and weather do the change or only do it if the surprise of the partition increases. The `merger` method, for example, with its default options reads:

```
Surpriser.merger(ic1=-1, ic2=-1, N=1, BfA=0)
```

Where `ic1` and `ic2` indicate the two communities that should be merged. When a number below zero is given, it will randomly select a community from all possible ones. The N parameter, indicates how many times it should try to do merges. If it is 1, it tries only once with the specified communities, if they were given, if it is bigger than 1, it randomly selects communities N times trying to do the merge and if it is 0, it will do a merge once, no matter the change in the surprise value, whereas if `N` is bigger than 0, it makes the merge only if the surprise value increases. If `BfA` (Best from All) is 1 instead

of 0, it ignores the value of `ic2`, and tries to merge `ic1` with all other communities, only effectively making the merge that increases the most the value of the surprise. This method will return the number of merges actually executed.

The calls to the other two methods receive very similar arguments:

```
Surpriser.exchanger(ic1=-1, ic2=-1, N=1, iex=-1, BfA=0)

Surpriser.extractor(ic1=-1, N=1, iex=-1, BfA=0)
```

The only difference is the optional `iex` that indicates which element from community `ic1` should be tried (to be exchanged or extracted). The `iex` should not be the node number (corresponding line in the M matrix), but its position in the community list. It should, therefore, be a number between zero and the community size minus one.

As an example, the network in figure 1 can have its full community structure revealed only by a series of merges. If the M-matrix is stored in variable `M`, multiple random merges may raise the surprise value to its maximum:

```
>>> sur = Surpriser(M)
>>> print sur
< Graph/communities info :
   Number of nodes       : 11
   Number of links       : 16 (55)
   Number of communities : 11
   Number of intralinks  : 0 (0)
   Surprise              : 0.000000 >
>>> sur.merger(N=100)
7
>>> print sur
< Graph/communities info :
   Number of nodes       : 11
   Number of links       : 16 (55)
   Number of communities : 4
   Number of intralinks  : 13 (13)
   Surprise              : 21.675463 >
>>> sur.show_communities()
community 0 (size 4) :  0, 1, 3, 2,
community 1 (size 4) :  9, 10, 8, 7,
community 2 (size 2) :  5, 6,
community 3 (size 1) :  4,
```

The returned number (7 in this case) is the total number of merges done. Note that actually, we found now a surprise value bigger than with the three communities we had created the object previously. With 4 communities, the surprise value is ~ 0.02 bigger and the central nodes are split into a community of two and another of one node[4]. Of course, nodes 4 and 6 are indistinguishable, and if we join nodes 4 and 5 into a community and leave node 6 alone in another, one should get the same surprise value:

[4]Note that in the `merger` call, the argument used was 100, this means that 100 random merges were tried. Since the process is random, in each execution of the above snippet a slightly different result may come.

```
>>> sur.exchanger(ic1=2, ic2=3, N=0, iex=0)
1
>>> sur
< Graph/communities info :
   Number of nodes       : 11
   Number of links       : 16 (55)
   Number of communities : 4
   Number of intralinks  : 13 (13)
   Surprise              : 21.675463 >
>>> sur.show_communities()
community 0 (size 4) :  0, 1, 3, 2,
community 1 (size 4) :  9, 10, 8, 7,
community 2 (size 1) :  6,
community 3 (size 2) :  4, 5,
```

In the `exchanger` call, `N=0` specifies that the element should be exchanged even if the surprise value is not improved, `ic1` is the community number from where an element will be removed, `ic2` the community to which it should be sent and `iex=0` specifies that the first element of the community (python counts from 0, not from 1) is the chosen one[5].

Two of the operations (extract and exchange) may be performed on a subcommunity level. Given the subgraph defined by a community, one may study the subcommunity structure of this community, and check if extracting a subcommunity or exchanging it will increase the overall surprise. The functions that perform this subcommunity analysis are:

```
Surpriser.subcommuniter(ic=-1, N=1, iex=-1)
Surpriser.subcommunity_exchanger(ic1=-1, ic2=-1, N=1, iex=-1, BfA=0)
Surpriser.subcommunity( ic )
```

The last one, `Surpriser.subcommunity` will return the `Surpriser` object representing the subgraph of the community. These methods, in order to produce the subcommunity structure of the chosen community, will execute the algorithm implemented in the surpriser method `stepper`, described below.

A greedy algorithm has been implemented which performs all merges, exchanges, extractions, and also the same operations over the subcommunities, in an ordered way until no operation is able to increase the surprise value any more. The order in which the operations are done may be differently chosen. In the default execution it proceeds as follows:

It starts a loop over the communities into which the graph is currently partitioned. In each community it will then start a loop in its nodes, check the community to which each of the node's neighbors belong and if they are not the same it tries first to merge the two communities, if it does not succeed, it tries to exchange an element between the two communities. Having finished the loop over the community nodes, it will, while successful, perform first single element extractions from the community and after this, subcommunity extractions. When it is done with the extractions, it tries all possible subcommunity exchanges. After the loop over all communities is finished, it starts all over again, until after going through all the communities it does not find a single operation that improves the value of the surprise. The call for this method with all its optional arguments is:

[5]Note that if the random process resulted in a different partition, these values might have to be different in order to do the same operation as described here. Actually a first random merge may actually link nodes 4 with 3 or 6 with 7, in which case the final partition will not be the maximum possible value for the surprise, unless extractions or exchanges are also made.

`Surpriser.stepper(ord=1, subcoms=1)`

There are four possible orders in which the operations are performed. With `ord=1`, it will perform exactly the algorithm described above. If `ord=2`, the loop over the communities tries to first merge only with the best possible community (increasing the surprise the most), and then, until exhaustion, tries to extract elements, then subcommunities, then to exchange elements and finally subcommunities. If `ord=3`, each operation is made to exhaustion in each community before jumping to the next operation in the same order as described before. For any other value of `ord`, the operations are made to exhaustion over each community, but in the following order: first all merges, then all extractions and finally all exchanges. If `subcoms=1`, the operations are also performed over the subcommunities (`Surpriser.subcommuniter` and `Surpriser.subcommunity_exchanger`), otherwise only over the communities themselves. In the tests we made, usually the default algorithm runs faster than the other versions and, though the other versions seem greedier for the merges are always done to the best match, not necessarily they render better results. Since the algorithm always run its loops in an ordered way and, as will be discussed soon, the surprise surface over the partition space is very rugged (very difficult to identify an global maximum among many local ones), the same initial partition ordered differently might result in a different maximum value for the final surprise with a different final partition.

Finally, it is also possible to perform a Monte-Carlo approach instead of the greedy algorithms described above. All operations can be performed by the methods with the tag `_an`, with a temperature indicated. In this case, the object computes the change in the surprise value when an operation is performed. If it increases the surprise, the change is accepted, if it decreases the surprise, the change is accepted with a probability given by $e^{-\frac{\Delta S}{T}}$ where ΔS is how much the surprise decreased (absolute value) and T is the temperature which was set by the method call. If no temperature was specified, the default value is 1. The methods to perform this annealing are:

`Surpriser.merger_an( ic1=-1, ic2=-1, N=1, T=1. )`

`Surpriser.exchanger_an( ic1=-1, ic2=-1, N=1, T=1. )`

`Surpriser.extractor_an( ic1=-1, N=1, T=1. )`

`Surpriser.subcommuniter_an( isc=-1, N=1, T=1. )`

`Surpriser.subcommunity_exchanger_an( isc=-1, N=1, T=1. )`

`Surpriser.montecarlo_step( T=1., K=self.K, subcoms=0 )`

The last method, `montecarlo_step`, performs a full Monte-Carlo step, it will randomly perform each one of the five annealing operations K times. In the next section an example with these functions will be shown.

Finally, the surpriser object has other methods to help inspect its current community structure and the underlying graph as a whole. We list them below, with their arguments.

- `Surpriser.community( ic )` → returns a list with the nodes in community `ic`.
- `Surpriser.connected( i, j )` → returns the element ij of matrix M. In other words, 1 if nodes i and j are connected and 0 otherwise.
- `Surpriser.linksin( list )` → the `list` argument should be a list of nodes (possible community) and the method returns a tuple where the first element is the number of internal links in the community defined by `list` and the second element the number of external links in the community.

- `Surpriser.partition()` → returns the current partition of the network's nodes into communities, i.e. a list with K elements where element i is the community of node i.
- `Surpriser.checkN( im=-1, iex=-1, iec1=-1, iec2=-1, iscex=-1, isc1=-1, isc2=-1 )` → this method can be used to study the changes that would be made in the surprise values by merges, exchanges, extractions, ...
- `Surpriser.shake( subcoms=1 )` → Performs all exchanges and/or subcommunity exchanges that do not alter the surprise value for the partition. The argument `subcoms`, as in the other functions where it appears, indicates whether to perform or not the operations at the subcommunity level as well.

In section 3 (Module `data` and Examples) we explain in more details the uses of these methods illustrating them with some, well, examples...

2.3 Functions in the `surprise` module

The module also has functions to study the partitions and the surprise itself.

First note that the evaluation of the surprise itself in equation (1), may quickly present problems due to machine precision, since real world networks are usually composed of thousands of nodes and in the evaluation of the surprise one must compute many combinations of the links in different numbers of communities, which involves the factorial of big numbers divided by even bigger ones. The strategy we used to evaluate this expression was to allocate in memory the logarithms of the factorials and factor out the biggest one from the summation in equation (1). The functions needed to compute the elements in the evaluation of the surprise and the surprise itself are implemented in:

- `fact( N )` → returns the natural logarithm of the factorial of N ($\ln N!$).
- `gammas( M, N )` → returns the natural logarithm of the combination of M elements N by N ($\ln \binom{M}{N}$).
- `surprise( M, F, nl, p )` → returns the surprise in equation (1).

The other functions in the package may be used to study the different partitions one may generate from a network:

- `compare( partition1, partition2 )` → evaluates the variation of information [5] between two partitions. A partition is a list with K elements where element i is the community of node i.

And finally, two functions to perform an embedding of a set of data which can be used to produce visualizations:

- `ChiGrad( matr, coords, gamma=-1., N=0, dlim=0.1 )` → Evaluates the gradient of a coordinate embedding.
- `embedding( matr, coords=[], gamma=-1., dlim=0.1, lamb=2., adj=0.05, eps=1.e-10 )` → evaluates an embedding.

Again the use of these functions will be clearer in the next section.

3 Module `data` and Examples

Along with the package, there is some data over which one may work in order to test the package. One may import the data which comes along with the distribution with:

```
>>> from Surpriser.data import *
```

This will import the M-matrices of the network in figure 1 (variable `Mtoy`) and two real world networks: the metabolic network of *Synechocystis sp. PCC 6803* (variable `Msyn`) and the PPI (protein-protein interaction) network of *Mycoplasma genitalium G37* (variable `M243273`). It also imports two lists indicating to which metabolite the nodes in the synechocystis network (`syn`) are associated and to which proteins the nodes in the PPI network are. These lists are in variables `NODESsyn` and `NODES243273`. The data for the production of the syn network was taken from KEGG database [6] and for the PPI network from STRING database [7]. Details about the construction of these networks and the study of their (and others) characteristics can be found in [8].

Having imported the data, the first example we showed in the previous section could be done differently:

```
>>> sur = Surpriser(Mtoy)
>>> sur.stepper()
(0, 0, 7, 0, 0)
>>> print sur
< Graph/communities info :
   Number of nodes       : 11
   Number of links       : 16 (55)
   Number of communities : 4
   Number of intralinks  : 13 (13)
   Surprise              : 21.675463 >
```

One interesting characteristic of the surprise (and actually of, probably, any quality function one can choose to in order to study the community structure of a graph) is the existence of many possible local maxima, which of course greatly complicates the task of finding the actual maximum (if it is indeed unique). As we showed, for the toy graph of figure 1 there are two possible partitions that result in the same maximum surprise. The authors of [3] propose an interesting approach to visualize this issue.

First, we used the annealing functions of the package in order to produce thousands of different partitions. Then, using the package function `compare`, which evaluates the variation of information between two partitions, we compared each partition we generated to each other, producing a distance matrix d, whose element d_{ij} is the variation of information (VI) between partitions i and j divided by the logarithm of eleven (since the network has 11 nodes, the VI between two partitions is a number between 0 and ln(11) and therefore the elements of the distance matrix will be numbers between 0 and 1).

We worked here (to produce the following figures) with 1336 partitions from the thousands we produced. In figure 2 one can see the histogram of the surprise values for these different partitions.

Then we proceeded to produce an embedding of the partitions: to each partition, we want to assign a two dimensional vector $\vec{r}_i = (x_i, y_i)$ in such a way that the euclidean distance between the vectors $\vec{r}_i$ and $\vec{r}_j$ are as similar as possible to the distances d_{ij} computed from the variation of information. To do that we implemented an steepest descent algorithm that walks over the $\vec{r}_i$ parameter space searching the minimum of χ^2 defined as:

$$\chi^2 = \sum_{i=1}^{N} \sum_{j>i|d_{ij}<d_{lim}} d_{ij}^{\gamma} \left(d_{ij} - \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right)^2 \tag{3}$$

where the first sum runs over all partitions while the second sum runs over the partitions whose distance to the i^{th} partition is less than some predetermined value d_{lim}. This sum will be smaller the closer that the euclidean distances between the

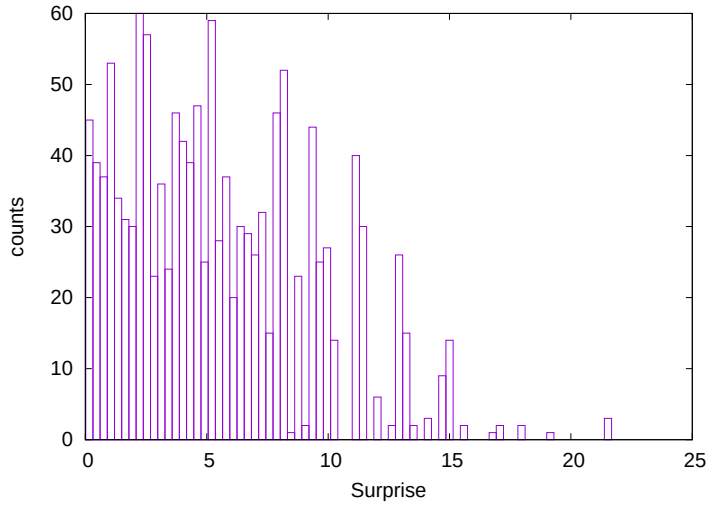

Figure 2: Histogram for the surprise values of 1336 different partitions of the toy network.

vectors $\vec{r}_i$ are to the variation of information evaluated in the matrix d_{ij}. Finally, parameter γ can be used to give more importance to partitions that are closer ($\gamma < 0$) or further ($\gamma > 0$) away from each other. The steepest descent algorithm implemented in function `embedding` evaluates the gradient of χ^2 in parameter space and updates the values x_i and y_i in the direction opposite to this gradient (towards smaller values of χ^2). In the function call, two parameters control the evolution of the algorithm: `lambda` and `adj`. `lambda` is how far in the direction opposite to the gradient it should start walking and `adj` is how to adapt `lambda` after an successful or failed step (it will be decreased or increased, respectively, by a proportion equal to it and therefore `adj` should be a value between 0 and 0.9). The algorithm should stop when the modulus of the gradient reaches a value below `eps`, defined in the function call, but it will also stop before reaching the desired precision if `lambda` gets smaller than machine precision or if it fails more than 5000 times to perform a successful iteration.

The data used here also comes with the package distribution (variables `partsToy` and `surpsToy`). Below we present the script used to produce figure 3, where in the xy plane are the coordinates $\vec{r}_i$ and in the z-axis the surprise value corresponding to each partition. The surface plot was produced using gnuplot dgrid3d command via the GNUplot.py package, which should be installed in order to perform some of the commands below. The user who wishes, can export the resulting coordinates and use it with another plotting library or software.

```
from Surpriser.surprise import *
from Surpriser.data import *
from math import log
from Gnuplot import Gnuplot as gplot # This and the line below can only be executed
from Gnuplot import Data as gdata    # if the GNUplot.py package is installed in the system.


g = gplot(persist=1) # to produce the plots


N = len(partsToy) # partitions in data


matr = [[0. for ii in xrange(N)] for jj in xrange(N)]
for ii in xrange(N):
```

```
    part1 = partsToy[ii]
    for jj in xrange(ii+1, N):
        part2 = partsToy[jj]
        dd = compare(part1, part2)/(log(11.)) # VI
        matr[ii][jj] = dd
        matr[jj][ii] = dd

coords = embedding(matr, coords=[], gamma=-1., dlim=1., lamb=.2, adj=.05, eps=1.e-9)

a, b, c = ChiGrad(matr, coords, gamma=-1., dlim=1.)
print c/(2*N) # gradient modulus per coordinate

xs = [coord[0] for coord in coords]
ys = [coord[1] for coord in coords]
zs = [surpsToy[ii] for ii in xrange(N)] # surprises from data
g("set cont") # following commands from GNUplot
g("set pm3d")
g("unset su")
d = gdata(xs, ys, zs, with_="lines")
g("set zlabel 'Surprise' rotate by 90")
g("set xlab 'x'")
g("set ylab 'y'")
g("set dgrid3d 150,150 qnorm 5")
g.splot(d)
```

Note that in the `embedding` function call above, the optional argument `coords` is an empty list (could actually have been left omitted). This should be an initial guess on the coordinates, but if left empty, the points are dispersed around the origin at random.

In figure 3 one can clearly see how ruggedy the surprise surface is in this visual representation. Actually the surprise is not a continuous function of its variables, since the variables upon which it depends are discrete, but this representation shows that similar partitions may have different values of the surprise making it a complex task to find an absolute maximum.

Finally, let's see what the method `Surpriser.checkN` is all about. Given an state of the partition (`Surpriser` object), its method `checkN` can be called in five different ways in order to check all possible changes in the surprise resulting from the five different basic operations:

- optional argument `im` → merges of community `im` to all other communities.
- optional argument `iex` → extraction of each element from community `iex`.
- optional arguments `iec1` and `iec2` → exchange of each element from `iec1` to `iec2`.
- optional argument `iscex` → extraction of each subcommunity from community `iscex`.
- optional arguments `isc1` and `isc2` → exchange of each subcommunity from `isc1` to community `isc2`.

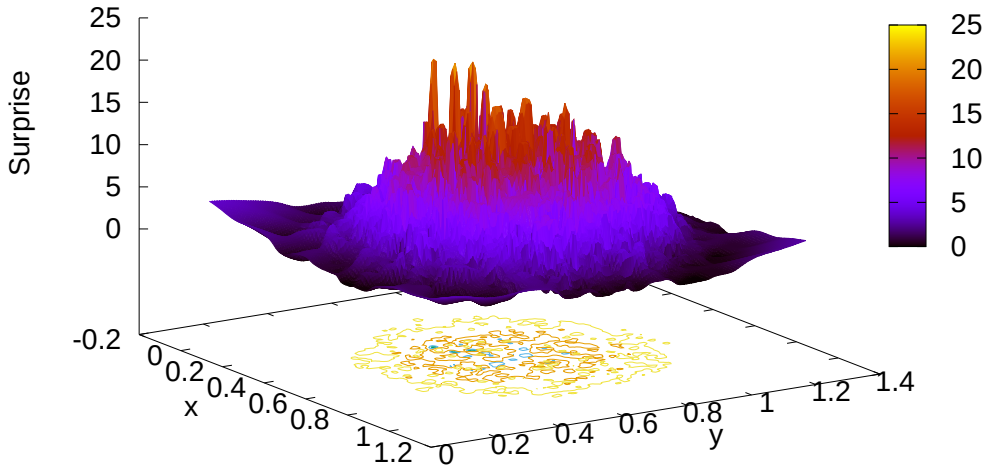

Figure 3: Surprise surface over the partition space.

In every call, it will return the result of only one of the above possibilities, if arguments are given to more than one, it performs only the first one appearing in the order above. To understand the method, consider the commands below analyzing the metabolic network of the cyanobacteria *Synechocystis* (from the package's data):

```
>>> syn = Surpriser(Msyn)
>>> syn.stepper()
(26, 157, 397, 60, 45)
>>> merges = syn.checkN(im=0)
```

In my computer (CORE i7, 16Gb ram) the stepper method takes around 20 seconds to run while it performs 26 extractions, 157 exchanges, 397 merges, 60 subcommunities extractions and 45 subcommunities exchanges. The surprise value reached is $\sim$ 3748.37 and the number of communities in this partition is 252. After execution of the above commands, the `merges` list has 252 elements, indicating what would be the change in the surprise value if community 0 were merged to each other community in the partition. Note that this community cannot be merged with itself, but the list has 252 elements anyway. Element `im` of the returned list will always be zero.

The stepper method reaches a point where it finds no possible improvement in the surprise value. So, it should come as no surprise that the maximum value of the `merges` list in this case is 0. (element `im`). Let's check the least bad possible merge:

```
>>> max(merges) # is 0.,
0.0
>>> merges.index(max(merges)) # the value of the first element since im=0
0
>>> ma = max(merges[1:])
>>> ima = merges.index(ma)
```

```
>>> print ima, ma
242 -1.12395317036
```

The above commands are telling us that, if we merge communities 0 and 242, the surprise value will decrease 1.12389405263, and this is the merge that decreases the surprise value the least among all possible merges of community 0 (there can be other merges that cause, at most, the same decrease).

Let's now study possible extractions:

```
>>> c3 = syn.community(3); print c3
[223, 222, 951, 391, 952, 953]
>>> extrs = syn.checkN(iex=3)
>>> ma = max(extrs)
>>> mi = min(extrs)
>>> print "Maximum : %f (%i)\nMinimum : %f (%i) "%(ma, extrs.index(ma), mi, extrs.index(mi))
Maximum : -3.124128 (0)
Minimum : -7.923214 (1)
```

After execution of these commands, `extrs` list will have 6 elements (the size of community 3). The maximum[6] possible change in surprise from extracting an element from this community is -3.124128 if the first element (223) is extracted and the minimum possible change is -7.923214. We can check what this node represents with another list in the package's data:

```
>>> print NODESsyn[223]
3-Sulfopyruvate
```

Let's now search for a situation similar to the one we had with the toy network: a change in the partition that keeps the surprise unaltered.

```
>>> pairs = []
>>> for i1 in xrange(syn.Nc):
...     for i2 in [ele for ele in range(syn.Nc) if ele!=i1]:
...         changes = syn.checkN(iec1=i1, iec2=i2)
...         if 0. in changes:
...             pairs.append( (i1, i2) )
...
>>> print len(pairs)
5
```

The above tells us that there are four pairs of communities where we find this situation where a given node can be in either community resulting in the same surprise value for the partition. Let's check one of these situations:

```
>>> print pairs[0]
(28, 201)
>>> c28 = syn.community(28)
>>> changes = syn.checkN(iec1=28, iec2=201)
```

[6] Note that these are actually negative values, and in this sense we are talking about maximum and minimum here, but the maximum "disturbance" in the surprise value would be with the extraction that results in the minimum change in the surprise.

```
>>> ich = changes.index(0.); print ich
5
>>> print c28[ich], NODESsyn[c28[ich]]
522 D-Ribulose1,5-bisphosphate
```

Therefore, node 522 (D-Ribulose1,5-bisphosphate), which is currently in community 28, could be in community 201 resulting in no change in the surprise value. Let's perform the operation and check:

```
>>> syn.exchanger(ic1=28, ic2=201, N=0, iex=5)
1
>>> print syn
< Graph/communities info :
   Number of nodes       : 967
   Number of links       : 2651 (467061)
   Number of communities : 252
   Number of intralinks  : 1121 (3904)
   Surprise              : 3748.366873 >
```

Let's now check the subcommunity operations over community 0:

```
>>> subCs = syn.subcommunity(0)
>>> print subCs
< Graph/communities info :
   Number of nodes       : 61
   Number of links       : 350 (1830)
   Number of communities : 16
   Number of intralinks  : 145 (231)
   Surprise              : 130.856080 >
>>> subExs = syn.checkN(iscex=0)
>>> ma = max(subExs)
>>> print subExs.index(ma)
1
>>> sc1 = subCs.community(1)
>>> c0 = syn.community(0)
>>> sc1p = [c0[ele] for ele in sc1]
>>> print sc1p
[683, 682, 777]
>>> nodes = [NODESsyn[ele] for ele in sc1p]
>>> print nodes
['Cob(II)yrinatea,cdiamide', 'Cobyrinate', 'Cobyrinatec-monamide']
>>> print ma
-3.15305928893
```

The call to the method `subcommunity` returns the `Surpriser` object representing the subgraph composed of the nodes in community 0. The above commands are telling us that inside this community, nodes 683, 682 and 777, representing[7] metabolites Cob(II)yrinatea_cdiamide, Cobyrinate and Cobyrinatec-monamide form subcommunity 1 and extracting this subcommunity from the community will decrease the value of the surprise 3.15322041956 points. Lets check:

```
>>> surpi = syn.surprise
>>> syn.subcommuniter(0, N=0, iex=1)
1
>>> print syn.surprise-surpi
-3.15305928893
```

Extracting the subcommunity resulted in the predicted change...

Now let's try to find the optimal partition with the annealing methods. The default algorithm of the `stepper` method found the optimal value of surprise for the metabolic network contained in `Msyn` to be around 3748.37 and it took in my computer a bit less than 20 seconds to run. With the code below, we start to build a community structure with random merges and then run the annealing over it:

```
>>> sur = Surpriser(Msyn)
>>> sur.merger(N=1000000)
819
>>> T = 2.
>>> kk = 0
>>> for ii in xrange(200):
...     k = sur.montecarlo_step(T=T, K=sur.K, subcoms=1)
...     kk += k
...     T = 0.95*T
...
>>> print sur
< Graph/communities info :
   Number of nodes       : 967
   Number of links       : 2651 (467061)
   Number of communities : 279
   Number of intralinks  : 975 (3511)
   Surprise              : 3186.664261 >
>>> print kk
52454
```

The merger command in the snippet above tries to perform one million random merges in the network. The average time it takes to run is 2.6s (in my computer). The number of successful operations done is around 830 (in the snippet above it was 819) and the average surprise value reached is 2229 for an average number of communities equal to 136. Runing the annealing process took my computer a bit more than 30 seconds (50% longer than the default algorithm). After the merges that create a starting community structure, the annealing process begin with a temperature of 2. The resulting partition

[7] Note how we carefully associated the references in one object to the nodes in the other object.

after 52454 successful operations has 279 communities with a surprise value of 3186, way below the 3748 reached by the algorithm and the temperature is too low for this process to execute any successful operation over the current partition.

Let's now try something else: can we use the annealing to improve upon the partition found by the algorithm?

First let's think what would a good temperature be. For takling this issue, we run the code below:

```
sur = Surpriser(Msyn)
sur.stepper()


changes1 = []
changes2 = []
changes3 = []
changes4 = []
changes5 = []
for ii in xrange(sur.Nc):
    bla = sur.checkN(im=ii)
    changes1.extend(bla[ii+1:])
    bla = sur.checkN(iex=ii)
    changes2.extend(bla)
    for jj in xrange(ii+1, sur.Nc):
        bla = sur.checkN(iec1=ii, iec2=jj)
        changes3.extend(bla)
        bla = sur.checkN(iec1=jj, iec2=ii)
        changes3.extend(bla)
        bla = sur.checkN(isc1=ii, isc2=jj)
        changes4.extend(bla)
        bla = sur.checkN(isc1=jj, isc2=ii)
        changes4.extend(bla)
    bla = sur.checkN(iscex=ii)
    changes5.extend(bla)
```

Now we have lists with all possible changes in the surprise value after running all possible partition modifications over the current state. The evaluation of these values took around 30 seconds. The averages of the values are:

`changes1` → merges: -4.63 ± 6.91

`changes2` → extractions: -8.41 ± 13.29

`changes3` → exchanges: -9.63 ± 13.15

`changes4` → subcommunity exchanges: -7.99 ± 23.18

`changes5` → subcommunity extractions: -5.95 ± 22.72

Each operation has a different (but all similar) typical value. We executed then the snippet below: an annealing starting with a temperature of 0.2. Less than 0.02% of all 420966 possible operations computed in the `changes` lists have variation of surprise equal or smaller than this value. The annealing process programmed stops running when, after two full montecarlo steps, no successful operation is made:

```
T = .2
stop = False
iszero = 0
while not stop:
    k = sur.montecarlo_step(T=T, K=10*sur.K, subcoms=1)
    if k==0:
        if iszero>1:
            stop = True
        else:
            iszero += 1
    else:
        iszero = 0
    T = 0.99*T
```

The annealing took around 10 minutes to run and improved a bit (∼0.1%) surprise value for the new partition:

```
>>> print sur
< Graph/communities info :
   Number of nodes       : 967
   Number of links       : 2651 (467061)
   Number of communities : 254
   Number of intralinks  : 1115 (3808)
   Surprise              : 3752.044684 >
```

Though one can see that the default algorithm programmed in the `stepper` method does not reach the global maximum of surprise (and even after the annealing one can not know if it has been reached), the improvements upon it might be too small to justify the extra computational time to keep searching.

Now, Let's come back to the study of changes that result in the same surprise, but this time let's focus in the exchange of subcommunities and in the other network present in the data:

```
>>> sur = Surpriser(M243273)
>>> sur.stepper()
>>> pairs = []
>>> for i1 in xrange(sur.Nc):
...     for i2 in [ele for ele in range(sur.Nc) if ele!=i1]:
...         changes = sur.checkN(isc1=i1, isc2=i2)
...         if 0. in changes:
...             pairs.append( (i1, i2) )
...
>>> print len(pairs)
5
```

The execution of the surpriser algorithm in this case, takes longer than in the `syn` graph for, though this network has less nodes, it is a much denser graph. The surprise reached by the algorithm is around 5663.49 and the partition has 56 different

communities. Some of the snippets in the following might also take a bit longer (minutes) to run.

We find one subcommunity that can be interchanged between communities resulting in the same value for the surprise. Let's check the metabolites in one of this:

```
>>> print pairs[0]
(1, 54)
>>> c1 = sur.community(1)
>>> subc = sur.subcommunity(1)
>>> changes = sur.checkN(isc1=1, isc2=54)
>>> ich = changes.index(0.)
>>> subcns = subc.community(ich)
>>> subcnsp = [c1[ele] for ele in subcns]
>>> print [NODES243273[ele] for ele in subcnsp]
['MG_189', 'MG_160']
```

So, we found a subcommunity of two elements that may be exchanged between communities 1 and 54 without altering the value of the surprise. Note, however, that these changes not altering the surprise value may open the opportunity that other merges, extractions, exchanges, ... do alter now the surprise values. In fact, our tests show that once the `stepper` method reaches its stop point, if changes not altering the surprise value are done to the partition, running again the stepper after these changes, might improve a little bit the value of the surprise (around 0.05% in some tests we performed which, for this PPI network, for example, raises the value from 5663.488667 to 5666.340193). Another `Surpriser` method does precisely this: it will perform once every change in the graph not altering the value of the surprise. This method is called `shake`.

```
>>> sur.shake()
(6, 2)
```

The results (6 and 2) are the number of single element exchanges that were done and the number of subcommunity exchanges (more than one element) that were done (2 in this case).

Along with the `data`, three other functions are also imported: `check`, `partToComs` and `comsToPart`.

The function `check` should receive as argument an `Surpriser` object. It will perform simple checks on the properties of the object and return `True` if it is all ok, or print an alert message and return `False` otherwise. This function was programmed for debugging purposes when programming the package and is here as courtesy. In principle, no change in the surpriser object should ever result in an object failing the checks, and if any bug is ever encountered, please try this function and inform me which message the function warns the test has failed.

The other function transforms a partition in a list of communities or vice-versa. The list of communities are used in the surpriser calls, while the partition list is returned by the `partition` method and used by the `compare` function.

```
>>> sur = Surpriser(Mtoy)
>>> sur.stepper()
(0, 0, 7, 0, 0)
>>> part = sur.partition()
>>> print part
[0, 0, 0, 0, 2, 2, 3, 1, 1, 1, 1]
```

```
>>> print partToComs(part)
[[0, 1, 2, 3], [7, 8, 9, 10], [4, 5], [6]]
>>> print sur.linksin( [7, 8, 9, 10] )
(6, 1)
```

In the above example, we also show the use of method `linksin`. Among nodes 7, 8, 9 and 10, there are 6 connections (number of connections inside a clique of size 4) and one external connection which is the one between nodes 7 and 6.

4 The `benchmark` module

In this module (`Surpriser.benchmark`) one can also find the implementation of a new benchmark to analyze the performance of this (and any other) community detection algorithm. The benchmark is a simple implementation of the intuitive idea that the ideal community would be a clique in the graph. In this benchmark, each community is initially defined as a clique and the cliques can have their connections degraded with a given probability and connections between the cliques may be created with another probability. Apart from the cliques, a fraction r of the graph nodes are randomly connected in order to give rise to communities formed by single nodes (not coming from cliques, though the list of cliques with which the benchmark is created may be composed of single nodes). The probabilities p and q may be functions of the cliques sizes.

There is one object and three functions in this module:

- `MBenchmark(cliques, r=0.01, cycle=False)`
- `Pielou(coms)`
- `Pielouer(Nc, pie, imin=1, imax=100, Nt=100, eps=.01)`
- `PielouerNodes(Nc, pie, Nmin, Nmax, imin=1, imax=100, Nt=100, eps=.01)`

The first is the benchmark object which should receive a list of cliques sizes (the communities sizes) upon creation plus optional arguments (`r` and `cycle`). The other functions are to evaluate the pielou index [9] for the communities sizes or to generate numbers with a given value for their pielou index.

When creating the benchmark, `cliques` should be a list containing the cliques sizes and `r` should be a number indicating the fraction of the graphs' nodes that should be created outside the cliques. Note that, the total number of nodes inside communities will be the sum of the `cliques` list and rK (the product of the total number of nodes in the network by the parameter r) is the number of nodes that do not belong to any community (are their own communities). Therefore, the total number of nodes, K, in the network will be given by $K = int\left(\frac{sum(cliques)}{1-r}\right)$.

Finally, if the last argument, `cycle`, is evaluated as `True`, each clique will have at least one connection to the adjacent clique in the list and the last clique with the first, they will be forming a ring.

The benchmark network is initially created as fully connected cliques disconnected to each other and a fraction of nodes rK randomly connected to the cliques. This structure can be degraded in two ways: connections inside the cliques may be removed and connections between the cliques can be created. The methods that control the degradation are:

- `degradP(func1=lambda cs, pars:.8, params1=())`
- `degradQ(func2=lambda cs1, cs2, pars:.01, params2=(), singles=True)`

When `degradP` is called, each link inside each community may be removed with a probability given by `func1(cs, params1)`, where `cs` is the community size and `params1` are parameters of the function. The method `degradQ` works in a similar way,

but here each possible link between two communities (cliques) might be created with probability given by `func2(cs1, cs2, params2)`, where `cs1` and `cs2` are the two cliques sizes and `params2` parameters the function may receive. In this last method, if `singles=True`, the possible links between the isolated nodes and each possible community may also be created with probability controlled by `func2` and `params2`.

To illustrate the use of the benchmark, consider the code below[8]:

```
>>> from Surpriser.surprise import *
>>> from Surpriser.data import *
>>> from Surpriser.benchmark import *
>>>
>>> cliques = [5, 5, 3, 4, 3] # 5 cliques
>>> r = .05
>>> cbar = sum(cliques)/5.
>>> Ke = 5*cbar/(1-r) # Expected number of nodes
>>> bench = MBenchmark(cliques, r)
>>> coms = partToComs(bench.partition)
>>> print coms
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12], [13, 14, 15, 16], [17, 18, 19], [20]]
>>> p = .2
>>> q = .05
>>> bench.degradP(func1=lambda c, par: p)
>>> bench.degradQ(func2=lambda c1, c2, par: q)
>>> print bench.incliques, bench.betweenc
26 10
```

In the above example, the last two numbers, 26 and 10, are the number of connections inside the communities and between them after degradation. One cannot predict the exact values for these numbers, but we can evaluate their expected values, given p and q (the functions were just constant numbers in the above example). If c_i is the size of community i, then the expected value for the number of links inside communities $\langle l_{ins} \rangle$ and the expected value for the number of links between communities $\langle l_{bet} \rangle$ are:

$$\langle l_{ins} \rangle = \sum_{i=1}^{N_c} (1-p) \frac{c_i(c_i-1)}{2} \tag{4}$$

$$\langle l_{bet} \rangle = \sum_{i=1}^{N_c} \sum_{j<i} q c_i c_j + rK \tag{5}$$

Let's check in the above example:

```
>>> lins = sum([(1-p)*ci*(ci-1)/2. for ci in cliques])
>>> comsizes = [len(com) for com in partToComs(bench.partition)]
>>> lbet = sum([q*ci*cj for ii, ci in enumerate(comsizes) for cj in comsizes[:ii]]) + Ke*r
```

[8]Note that the degradation and clique connection process is random, therefore, each time this snippet of code is executed, the result might be a bit different.

```
>>> print lins, lbet
25.6 9.95263157895
```

The result is close to the values 26 and 10, though note that in each execution of the snippet, a different result might come up, since the generation of the benchmark contains a random process, as well as the degradation of it. But if the process is repeated many times, the average result will be closer and closer to the above numbers:

```
>>> bet = []
>>> ins = []
>>> for ii in xrange(100000):
...     bench = MBenchmark(cliques, r)
...     bench.degradP(func1=lambda c, par: p)
...     bench.degradQ(func2=lambda c1, c2, par: q)
...     bet.append(bench.betweenc)
...     ins.append(bench.incliques)
...
>>> print sum(bet)*1./100000
9.90463
>>> print sum(ins)*1./100000
25.58878
```

Finally, we can execute the surpriser algorithm over the graph and compare with the original partition:

```
>>> from math import log
>>>
>>> sur = Surpriser(bench.M)
>>> sur.stepper()
(3, 1, 14, 0, 0)
>>> compare(bench.partition, sur.partition())/log(sur.K)
0.078267404240154
```

The `compare` function returns the variation of information between two partitions, which is a number between 0 (if they are exactly equal) and $\log(K)$ (the maximum possible value) where K is the length of the partition (the number of nodes). In the above example, the discrepancy was around 8%, which means that around 1 or 2 nodes in 21 (K) are misplaced in the partition obtained by the algorithm with respect to the benchmarked one.

In the above example, the internal connections of all cliques were degraded with the same probability p, but it is reasonable to think that bigger cliques will have a bigger degradation probability than smaller ones. For instance, a clique of size two, if degraded can no longer be considered to be a community for its members will not be connected any more. For taking into account the effect that the community sizes have in the probabilities p and q, the functions `func1` and `func2` in the `degradP` and `degradQ` methods, should not be left constant.

To show how this works, consider the example below, where the degradation probability grows linearly from 0, when the clique size is 2 until it reaches 80% when the clique size is 50.

```
>>> func = lambda cs, params: params[0]*cs + params[1] # linear function
```

```
>>> cliques = range(2, 51, 3)
>>> nc = len(cliques)
>>> r = .05
>>> q = 0.005
>>> params = (1./60, -1./30) # func(2, params)=0 and func(50, params) = 0.8
>>> cbar = sum(cliques)*1./nc
>>> Ke = nc*cbar/(1-r)
>>> bench = MBenchmark(cliques, r)
>>> bench.degradP(func1=func, params1=params)
>>> bench.degradQ(func2=lambda c1, c2, pa: q) # q is left constant
>>> coms = partToComs(bench.partition)
>>> print bench.incliques, bench.betweenc
2834 518
>>>
>>> lins = sum([(1-func(ci, params))*ci*(ci-1)/2. for ci in cliques])
>>> comsizes = [len(com) for com in partToComs(bench.partition)]
>>> lbet = sum([q*ci*cj for ii, ci in enumerate(comsizes) for cj in comsizes[:ii]]) + Ke*r
>>> print lins, lbet
2856.0 525.858157895
```

Note that in the `MBenchmark.degradP` method call, the degradation probability function (`func1`) is a two variable function, the first variable is the clique size and the second a list with the function's parameters (even if the function has a single parameter, it should be given as a single element list). The probability of connections between cliques (given by the `MBenchmark.degradQ` method) is also a function (`func2`), in principle of three variables, the first two being the two cliques sizes and the third the function's parameters. In the above example, this function is just a constant (q).

In order to check the results, another function has been implemented in order to analyze the `Surpriser` object:

```
surStats(sur, pprint=False)
```

This function will return a tuple with the parameters estimated from the community structure contained in the `Surpriser` object: it evaluates the global p, q and r from the object and return it along with two lists, the first containing the cliques (communities with more than 2 elements) and the second the p value inside each community. The snippet below exemplifies its use in the above created network:

```
>>> sur = Surpriser(bench.M, coms)
>>> pp, qq, rr, coo, ps = surStats(sur)
>>> pp
0.6149979622333922
>>> qq
0.004924442145266069
>>> rr
0.04946236559139785
>>> coo = [len(ele) for ele in coo]
```

```
>>> for ii, ele in enumerate(coo):
...     print "size: %3i - obs: %1.4f, func: %1.4f"%(ele, ps[ii], func(ele, params))
...
size:   2 - obs: 0.0000, func: 0.0000
size:   5 - obs: 0.1000, func: 0.0500
size:   8 - obs: 0.1071, func: 0.1000
size:  11 - obs: 0.2182, func: 0.1500
size:  14 - obs: 0.1319, func: 0.2000
size:  17 - obs: 0.2206, func: 0.2500
size:  20 - obs: 0.2947, func: 0.3000
size:  23 - obs: 0.3557, func: 0.3500
size:  26 - obs: 0.3723, func: 0.4000
size:  29 - obs: 0.4655, func: 0.4500
size:  32 - obs: 0.5000, func: 0.5000
size:  35 - obs: 0.5479, func: 0.5500
size:  38 - obs: 0.6245, func: 0.6000
size:  41 - obs: 0.6402, func: 0.6500
size:  44 - obs: 0.7061, func: 0.7000
size:  47 - obs: 0.7530, func: 0.7500
size:  50 - obs: 0.8106, func: 0.8000
```

The last output in this snippet is the observed degradation probability inside each community and the value of the linear function for the community size. The `surStats` function might also receive an optional second argument that, if equal to `True`, will print in the screen a summary of the statistics:

```
>>> pp, qq, rr, coo, ps = surStats(sur, True)
# Comunities : 40
# Cliques    : 17
av click size: 26.000000 +- 15.149257
p            : 0.614998
pbar         : 0.402849 +- 0.252212
q            : 0.004924
r            : 0.049462
```

The difference between `p` and `pbar` presented by the output summary is that `p` is globally evaluated for the whole community structure (all intracommunity links divided by the total possible number given the identified communities) while the values presented for `pbar` are the average and standard deviation of the `ps` list which contains the values of p evaluated for each community separately.

The three functions in this package deal with the diversity in the community sizes. The Pielou index [9] measures how even a partition of elements into groups is. It is taken from a biology context, where one wants to measure, for different interacting species, how even the distribution of the individual organisms in the different group of species is.

Let's take the number K of elements and divide them into the N_c communities. Each group i has C_i elements in it,

therefore, $\sum_{i=1}^{N} c_i = K$.

Now, take an element at random. The probability that it belongs to community i is $p_i = \frac{c_i}{K}$. The more even the communities sizes are, the more uncertain one can be on the community of a randomly selected element. The pielou index PI is a measure of this uncertainty. First, evaluate the entropy for the p_i distribution:

$$H = -\sum_{i=1}^{N} p_i \ln p_i, \tag{6}$$

its maximum value is $\ln N$ in the case that $p_i = \frac{1}{N}$ (most even case) and its minimum is 0 if one has all elements in the same community (one is in this case 100% certain on a randomly selected element). Thus, one can construct a normalized index (between 0 and 1) dividing H by $\ln N$ and this is the pielou index:

$$PI = \frac{H}{\ln K}. \tag{7}$$

The package function `Pielou` measures, for a list of numbers, its pielou index given by equation 7. This function should receive upon calling a list of numbers representing the community sizes. The other two functions, `Pielouer` and `PielouerNodes` can be used to generate a list of numbers with a desired value for its corresponding pielou index. Both functions must receive as mandatory arguments the number of communities (`Nc`) and the desired pielou index (`pie`), the optional argument `eps` whose default value is 0.01 specifies the accepted tolerance in the desired value of PI (the algorithm returns the generated list when its corresponding pielou index is between `pie-eps` and `pie+eps` the other optional arguments `imin` and `imax` control the minimum and maximum accepted sizes for a community while the other optional argument `Nt` controls how many times (`Nt*Nc`) the algorithm should try number substitutions in order to achieve `pie` before stopping. The function `PielouerNodes` has two extra mandatory arguments in order to control the minimum and maximum number of total nodes in the graph (the sum of the community sizes should be bigger than `Nmin` and smaller than `Nmax`).

```
>>> coms = Pielouer(10, 0.75) # 10 communities with PI=0.75
>>> print coms, sum(coms)
[25, 12, 12, 1, 9, 24, 98, 15, 95, 5] 296
>>> print Pielou(coms)
0.759392321411
>>> coms = Pielouer(10, 0.75, eps=0.0001) # 10 communities with PI=0.75 with greater precision
>>> print coms, sum(coms)
[12, 18, 1, 16, 96, 77, 5, 2, 9, 49] 285
>>> print Pielou(coms)
0.750025475811
```

If these numbers are now used as cliques sizes in the generation of a benchmark network, for example, the first graph would have 296 nodes and the second 285 (if r=0). If one wants a given number of nodes in the networks, one should use the other pielouer function:

```
>>> coms = PielouerNodes(10, 0.75, 320, 330) # 10 communities with PI=0.75
Best pielou achieved after 1000 tries is 0.867901.
```

```
>>> print coms
[1, 3, 16, 63, 22, 71, 16, 41, 38, 57]
>>> print Pielou(coms)
0.867901035382
```

Here one sees an example when the algorithm failed after `Nt*Nc` attempts. One can try to increase `Nt`:

```
>>> coms = PielouerNodes(10, 0.75, 320, 330, imin=2, Nt=10000) # 10 communities with PI=0.75
>>> print coms, sum(coms)
[62, 15, 99, 23, 2, 72, 46, 2, 2, 2] 325
>>> print Pielou(coms)
0.75717468992
```

Now, say one wants a given exact number of nodes. It is only a matter of setting `Nmin=Nmax`:

```
>>> coms = PielouerNodes(10, 0.75, 320, 320, imin=2, Nt=10000) # 10 communities with PI=0.75
>>> print coms, sum(coms)
[14, 2, 49, 72, 2, 14, 2, 97, 2, 66] 320
>>> print Pielou(coms)
0.7431058202
```

5 The `randoms` Module

Along with the package, this module has been introduced with some functions in order to deal with random numbers and statistics, in particular, dealing with the power-law scale-free distribution. In the benchmark introduced above and some of the analysis done[9], one already sees the need of a random number generator and a function to produce the statistics of a set of numbers. These two are included in the present module as:

- `stats(numbers, assymetry=0)` → returns the average, standard deviation and, if the `assymetry` parameter is evaluated as `True`, the skewness for a set of numbers.
- `drand()` → generates a homogeneous random number between 0 and 1.

Below a simple example for their use:

```
>>> from Surpriser.randoms import *
>>>
>>> N = 1000000
>>> nums = [drand() for ii in xrange(N)]
>>> print stats(nums, assymetry=1)
(0.4998283690074808, 0.28866528035922, 0.00046975546204043345)
>>>
```

In this example, 1000000 random numbers were generated and the descriptive statistics for the set is evaluated. Note that the expected average, standard deviation and skewness for the homogeneous distribution between 0 and 1 are respectively $\frac{1}{2}$, $\sqrt{\frac{1}{12}}$ and 0.

[9] Therefore, the `Surprise.data` module has dependency on some functions of the `Surprise.randoms` module.

A distribution that has been the focus of much attention in the study of graphs is the power-law distribution. In this distribution the probability of generating a given number is proportional to the number to some negative power: $p(x) \propto x^{-\gamma}$, where γ is a parameter.

If x can assume only integer values bigger than some x_0, the precise form of the distribution reads:

$$p(x/\gamma, x_0) = \begin{cases} \frac{x^{-\gamma}}{\zeta(\gamma, x_0)} & x \geq x_0 \\ 0 & x < x_0 \end{cases} \tag{8}$$

$$\zeta(\gamma, x_0) = \sum_{x=x_0}^{\infty} x^{-\gamma} \tag{9}$$

where $\zeta(\gamma, x_0)$ is the Riemann zeta function (modified such that the sum starts at a minimum value x_0). This distribution has γ and x_0 as parameters. The parameter x_0 is an integer indicating the smallest number in the distribution.

If x can assume any real value bigger than some x_0, the distribution reads:

$$p(x/\gamma, x_0) = \begin{cases} \frac{\gamma-1}{x_0}\left(\frac{x}{x_0}\right)^{-\gamma} & x \geq x_0 \\ 0 & x < x_0 \end{cases} \tag{10}$$

In both cases (x real or integer), the distribution is only defined (can be normalized) for γ bigger than 1, it only has finite average for γ bigger than 2 and the distribution's standard deviation is only defined if γ is bigger than 3.

The following functions evaluate the expressions related to these distributions:

- `drand_SF(gamma=2.3, x0=1.)` $\rightarrow$ returns a random number between x0 and infinity with continuous power-law distribution (Eq. (10)).
- `irand_SF(gamma=2.3, x0=1, iMAX=4000000)` $\rightarrow$ returns an integer random number between x0 and infinity ($\sim$`iMAX`) with discrete power-law distribution (Eq. (8)).
- `zeta(gamma, xmin=1, N=20)` $\rightarrow$ evaluates the zeta Riemann function $\zeta(\gamma, x_0)$ (Eq. (9)).
- `dzetadx(gamma, xmin=1, N=20)` $\rightarrow$ evaluates the first derivative with respect to γ of the zeta Riemann function $\frac{d}{d\gamma}\zeta(\gamma, x_0)$.

The implementation of the Riemann zeta function evaluates a sum (not the one in Eq. (9), which converges too slowly) and the optional argument `N` in the last two functions controls the number of terms in this sum. Our numerical experiments show that the precision achieved, even for values of γ close to 1, are smaller than the machine precision summing 20 terms. The function `irand_SF` does have to evaluate the sum in Eq. (9) until reaching some random probability and the optional argument `iMAX` controls the maximum number of terms in this sum, which will therefore be the maximum possible random integer that the function returns. If the sum reaches this value, a number close to and slightly bigger than `iMAX` is returned, but note that in these cases, the actual returned number should have been even bigger, therefore the function actually failed[10]. The probability that this sum passes the value in `iMAX` can be evaluated from the distribution in Eq. (8):

$$p(i > iMAX) = 1 - \sum_{x=x_0}^{iMAX} \frac{x^{-\gamma}}{\zeta(\gamma, x_0)} \tag{11}$$

[10]Working with much bigger values for `iMAX` is of course possible, but increases the running time.

In the snippet below, given $x_0 = 1$, this is evaluated for different values of γ:

```
>>> gammas = [1.01, 1.05, 1.1, 1.5, 2.0, 2.3, 2.5]
>>> for gamma in gammas:
...     som = 0.
...     for ii in xrange(1, 4000001):
...         som += ii**(-gamma)
...     som /= zeta(gamma)
...     print "For gamma=%1.2f : %3.12f %s"%(gamma, (1.-som)*100, "%")
...
For gamma=1.01 : 85.403692624039 %
For gamma=1.05 : 45.442666406296 %
For gamma=1.10 : 20.659783339617 %
For gamma=1.50 : 0.038279336016 %
For gamma=2.00 : 0.000015198166 %
For gamma=2.30 : 0.000000140733 %
For gamma=2.50 : 0.000000010955 %
```

When working with values of γ slightly above 1, it is important to pay close attention to this issue.

Finally, the last four functions in the module deal with the estimation of the parameter γ for a set of numbers:

- `gamma_MLE(xi, x0=1, uncert=0, gamma0=2.3, eps=1.e-8, nsum=20)` $\rightarrow$ estimates the parameter γ that maximizes the likelihood for a set of numbers assuming the distribution in Eq. (8).
- `gamma_MLE_cont(xi, x0=1., uncert=0, eps=1.e-8, delt=0.02)` $\rightarrow$ estimates the parameter γ that maximizes the likelihood for a set of numbers assuming the distribution in Eq. (10).
- `lnL(xi, gamma=2.3, x0=1, nsum=20)` $\rightarrow$ evaluates the logarithm of the likelihood for a set of numbers using the distribution in Eq. (8).
- `lnL_cont(xi, gamma=2.3, x0=1.)` $\rightarrow$ evaluates the logarithm of the likelihood for a set of numbers using the distribution in Eq. (10).

Given a set of numbers that one assumes comes from a given distribution $p(x_i)$, the log-likelihood of the set is given by:

$$\ln \mathcal{L} \quad = \quad \sum_{i=1}^{N} \ln p(x_i) \tag{12}$$

The functions `gamma_MLE` and `gamma_MLE_cont` evaluate for a list of numbers (`xi`), given a value for x_0, the value of the parameter γ that maximizes the likelihood in Eq. (12). The optional parameter `uncert` in these functions indicate if the function should also return the lower and upper uncertainty for the γ parameter, these being the points around γ where the likelihood is increased by half point. The other optional arguments in these functions are of numerical nature: the parameter `gamma0` indicates where the algorithm (steepest descent) should start looking; the parameter `eps` indicates when the search finishes (reaches the desired precision or which value of the derivative have to be considered null); `nsum` is the same parameter as `N` in the evaluation of the Riemann-zeta function and `delt` is the value of $\Delta\gamma$ for the numerical evaluation of derivatives.

Finally, the functions `lnL` and `lnL_cont` evaluate the log-likelihood given by equation (12) for a set of numbers `xi` and given values for γ and x_0. In the snippet below random numbers with scale-free distribution are generated and these functions are used in order to cross-check the quality of the produced set.

```
>>> nums = [irand_SF(2.5, x0=2) for ii in xrange(1000)]
>>> gamma, uncs = gamma_MLE(nums, uncert=1, x0=2)
>>> print "gamma = %1.6f ^ + %1.6f _ - %1.6f"%(gamma, uncs[0], uncs[1])
gamma = 2.497919 ^ + 0.049426 _ - 0.048302
>>> ln1 = lnL(nums, gamma=gamma, x0=2)
>>> ln2 = lnL(nums, gamma=gamma+uncs[0], x0=2)
>>> ln3 = lnL(nums, gamma=gamma-uncs[1], x0=2)
>>> print "Diference 1: %1.6f \nDiference 2: %1.6f "%(ln1-ln2, ln1-ln3)
Diference 1: 0.500000
Diference 2: 0.500000
```

Here 1000 numbers are generated and the γ for the set is evaluate as $\gamma = 2.471170^{+0.048492}_{-0.047393}$. Also the difference between the log-likelihood in the central value and in the uncertainty limits is checked.

The same can be done for the continuous distribution:

```
>>> nums = [drand_SF(1.9, x0=3) for ii in xrange(1000)]
>>> gamma, uncs = gamma_MLE_cont(nums, uncert=1, x0=3)
>>> print "gamma = %1.6f ^ + %1.6f _ - %1.6f"%(gamma, uncs[0], uncs[1])
gamma = 1.921007 ^ + 0.029433 _ - 0.028819
>>> ln1 = lnL_cont(nums, gamma=gamma, x0=3)
>>> ln2 = lnL_cont(nums, gamma=gamma+uncs[0], x0=3)
>>> ln3 = lnL_cont(nums, gamma=gamma-uncs[1], x0=3)
>>> print "Diference 1: %1.6f \nDiference 2: %1.6f "%(ln1-ln2, ln1-ln3)
Diference 1: 0.500000
Diference 2: 0.500000
```

6 Bugs and Feedback

These functions have been thoroughly tested in a linux mint system using python 2.7.17 and python 3.6.9.

Please email any detected bugs, suggestions or your feedback to the author (Daniel Gamermann: gamermann@gmail.com).

7 License

References

[1] Rodrigo Aldecoa and Ignacio Marín. Deciphering network community structure by surprise. *PloS one*, 6(9):e24195, 2011.

[2] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[3] Benjamin H Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Physical Review E*, 81(4):046106, 2010.

[4] Rodrigo Aldecoa and Ignacio Marín. Surprise maximization reveals the community structure of complex networks. *Scientific reports*, 3:1060, 2013.

[5] Marina Meilă. Comparing clusterings by the variation of information. In *Learning Theory and Kernel Machines*, pages 173–187. Springer Berlin Heidelberg, 2003.

[6] Minoru Kanehisa, Yoko Sato, Masayuki Kawashima, Miho Furumichi, and Mao Tanabe. Kegg as a reference resource for gene and protein annotation. *Nucleic Acids Research*, 44(D1):D457–D462, 2016.

[7] D. Szklarczyk, A. Franceschini, S. Wyder, K. Forslund, D. Heller, J. Huerta-Cepas, M. Simonovic, A. Roth, A. Santos, K. P. Tsafou, M. Kuhn, P. Bork, L. J. Jensen, and C. von Mering. STRING v10: protein-protein interaction networks, integrated over the tree of life. *Nucleic Acids Res.*, 43(Database issue):D447–452, Jan 2015.

[8] D. Gamermann, J. Triana-Dopico, and R. Jaime. A comprehensive statistical study of metabolic and protein–protein interaction network properties. *Physica A: Statistical Mechanics and its Applications*, 534:122204, November 2019.

[9] E.C. Pielou. The measurement of diversity in different types of biological collections. *Journal of Theoretical Biology*, 13:131–144, December 1966.