SESMotifAnalyser Documentation

Release 2.3.0

Tim Seppelt

Feb 19, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Social-Ecological Systems have been analysed in may different ways using several statistical approaches. This software package, the SESMotifAnalyser or sma, was developed in the context of the work of Ö. Bodin et. al. Most of the terminology is taken from [BT12] and [BN16].

This software package consists of two major parts: a Python module for analysing SES' which is as also accessible in R and an extension of the R package `ergm.userterms` for analysing SES' in the framework of Exponential Random Graph Modelling.

This package is the result of two internship carried out at the Stockholm Resilience Centre in October 2018 and August 2019 by Tim Seppelt (t.seppelt-dev@posteo.de) under the supervision of Örjan Bodin (orjan.bodin@su.se). It is published under GNU GPLv3. Contact the author in case of any questions.

This file aims to provide a complete documentation of all components of sma. Particularly interesting are the following sections: *Terminology*, *An Example: Multi-level SENs* and *Appendix: The Motif Zoo*.

1.1 Installation

The Python package can be installed from PyPI using `pip`.

```
pip3 install sma
```

Per default, dependencies for drawing SENs will not be installed. Use the following command instead:

```
pip3 install sma[plots]
```

The version in PyPI might be slightly older than the most recent version on Gitlab.

Please follow these instructions for installing the R `ergm.userterms`.

1. download this repository, either with git or as archive.
2. call in R: `download.packages("ergm.userterms", type="source", destdir=".")`
3. unzip the just downloaded archive
4. copy the files from this repository from the folder `ergm.userterms/` to the respective folders in the unzipped archive
5. open a terminal in the folder containing the `ergm.userterms` folder
6. call `RCMD INSTALL ergm.userterms` or in Unix-like operating systems `R CMD INSTALL ergm.userterms`.
7. now the package is available in R and can be used after calling `library(ergm.userterms)`, see `example.R` for examples.

Check the documentation of R's `ergm.userterm` package for further details on the installation. See also *Integration in R*.

1.2 Motifr

Many functions of this package are available from within R. The corresponding R package is motifr developed by Mario Angst and the author of this package.

1.3 Terminology

A *social-ecological network (SEN)* is a simple directed or undirected graph[1] with two-coloured vertices and without self-loops. These two colours are called *ecological* or *social* and encoded using the nodal attribute `sesType` in `networkx` graph objects. The colour of a node is often referred to as its *level*. Per default, the ecological level is level zero (`sesType = 0`) and the social level is level one (`sesType = 1`). All nodes of one level form a *subsystem*, e.g. the *ecological / social subsystem*.

A *multi-level network* is a simple graph coloured vertices. In such a network more than two colours can occur, e.g. there are *three-level networks* with three levels and *two-level networks*, which are just SENs. The levels are named $0, 1, 2, \ldots$ and encoded using the nodal attribute `sesType`. Often the number of levels is not important and the terms multi-level network and SEN are used synonymously.

A *motif* is a subgraph of network typically consisting only of few nodes. The main purpose of this package is to count motifs and to analyse their distribution. There exist a large variety of motifs, and this package supports some of them. A full list of all supported motifs is given in *Appendix: The Motif Zoo*.

Motifs are identified by three pieces of information. These are,

1. the *motif class* which describes the isomorphism class of the subgraph induced by the motif (e.g., closed triangle, open triangle, etc.),
2. the *signature* which indicates of how many nodes from how many distinct levels the motif consists,
3. the *positions* which indicate from which levels the nodes are taken.

The motif class is specified by names, which can be looked up in *Appendix: The Motif Zoo*. Signatures are ordered tuples of integers, e.g. `(1, 2)` for triangles, `(1, 2, 1)` for a certain multi-level motif. The length of this tuple is the number of levels which occur in the motif. The entries specify how many nodes are taken from the levels. A full list of all supported signatures can be accessed by calling `sma.supportedSignatures()`.

Specifying the position of a motif is slightly more complicated and requires more subtle notions.

The difference between signature and positions becomes clearer when thinking of the signature as the amounts of nodes from each level in an abstract motif while viewing the positions as the actual levels (`sesTypes`) in concrete SEN.

Signature and motif class together specify the abstract structure of the motif. These two properties are often used to refer to a motif. For example, (1,2)-motifs of class II.C are closed triangles with one node on one level and the two other nodes on another level (these motifs are also called 3-motifs). Another example are (2,2)-motifs (or 4-motifs) of type I.D, i.e. four-cycles ranging across two levels. See the following list of detailed examples:

- motif class II.C, signature 1, 2, positions 0, 1: closed triangle with ecological distinct node and social non-distinct nodes (3E-motif)
- motif class I.C, signature 1, 2, positions 1, 0: open triangle with social distinct node and ecological non-distinct nodes (3S-motif)
- motif class I.D, signature 2, 2, positions 1,2: four-cycle with two nodes from level 1 and two nodes from level 2 in a (at least) three level network (4-motif)

Often the positions of a motif are implicit from the signature. This package implements a process for determining positions automatically called *Position matching*. In order to specify motifs in full generality two further objects, arities and roles, are used.

[1] This package was created for analysing undirected SENs and later extended to cover also directed networks. Hence most older functions work only for undirected SENs.

Arities and roles always come together. *Arities* are an ordered list of integers defining the number of nodes taken from specific levels in a motif. *Roles* a levels assigned to the non-zero entries in the list of arities.

Motif classes, signatures and positions can be stated using motif identifier strings. The *motif identifier* or *motif identifier string* of a motif is a string of the form `HEAD[CLASS]` or `HEAD`, where `CLASS` specifies the arities and roles of a motif and `CLASS` its motif class. `HEAD` is a comma-separated list of arities (e.g., `1,2,2`) or a comma-separated list of arities and roles, separated by colons (e.g., `1:0,2:2,2:1`). The example above has the motif identifier `0,2,1`, or more succinctly `1:2,2:1`. When referring in this case to open triangles, one gets `1:2,2:1[I.C]`.

1.3.1 Position matching

Position matching is the process of assigning concrete positions to the entries of the signature of a motif. This technical step is necessary for example when counting motifs. It is implemented in *sma.matchPositions()*.

Position matching takes as input arities and optionally roles. The output is a list of positions (list of `sesType`'s for every entry in the signature). The procedure works as follows:

1. Determine the signature of the motif. Based on the arities a supported signature is found. This is done by calling *sma.multiSignature()* and querying *sma.motifInfo()*, i.e. the database of all implemented motifs. *sma.multiSignature()* returns the entries of the signature in ascending order. The *sma.MotifInfo* returned by *sma.motifInfo()* contains the signature in proper order.
2. Determine the positions associated with the entries of the signature. This is done by calling *sma.matchPositions()*. Two cases are distinguished:
   - no roles given (`roles = []`): the position associated with the i-th entry s_i of the signature is the index of the j-th occurrence of the value s_i in the list of arities where the value s_i occurs $j-1$ times in $s_0, \ldots, s_{i-1}$.
   - roles given: it is assumed that roles and arities are of the same length, i.e. arities does not contain zero and is in fact a signature (perhaps in wrong order). Two steps are needed:
     1. Determine the indices of the entries of the signature in the list of arities. This is done by calling *sma.matchPositions()* without specifying roles, cf. bullet point above. This results in *prepositions*.
     2. Determine the levels corresponding to the signature entries. Roles are given in the order of the arities, but the signature has possibly a different order. The levels are selected in the order defined by the prepositions.

Consider the following examples:

```
# Example 1
arities            = [1, 2]
roles              = [] # no roles
ordered_signature  = sma.multiSignature(arities) # gives [1, 2]
motif_info         = sma.motifInfo(ordered_signature) # gives sma.Motif3Info
signature          = motif_info.signature # gives [1, 2]
positions          = sma.matchPositions(signature, arities, roles) # gives [0, 1]
# in the (1,2)-motif specified here the first level has sesType 0,
# the second sesType 1. This is a 3E-motif.

# Example 2
arities            = [2, 1]
roles              = [] # no roles
ordered_signature  = sma.multiSignature(arities) # gives [1, 2]
motif_info         = sma.motifInfo(ordered_signature) # gives sma.Motif3Info
signature          = motif_info.signature # gives [1, 2]
positions          = sma.matchPositions(signature, arities, roles) # gives [1, 0]
# in the (1,2)-motif specified here the first level has sesType 1,
# the second sesType 0. This is a 3S-motif.
```

(continues on next page)

(continued from previous page)

```
# Example 3
arities             = [2, 0, 2]
roles               = [] # no roles
ordered_signature   = sma.multiSignature(arities) # gives [2, 2]
motif_info          = sma.motifInfo(ordered_signature) # gives sma.Motif4Info
signature           = motif_info.signature # gives [2, 2]
positions           = sma.matchPositions(signature, arities, roles) # gives [0, 2]
# in the (2,2)-motif specified here the first level has sesType 0,
# the second sesType 2.

# Example 4
arities             = [2, 1, 2]
roles               = [0, 2, 1] # no roles
ordered_signature   = sma.multiSignature(arities) # gives [1, 2, 2]
motif_info          = sma.motifInfo(ordered_signature) # gives sma.Motif221Info
signature           = motif_info.signature # gives [2, 2, 1]
positions           = sma.matchPositions(signature, arities, roles) # gives [0, 1,␣
↪2]
# in the (2,2,1)-motif specified here the first level has sesType 0,
# the second sesType 1, the third sesType 2.
```

In order to test how the position matching works, call `sma.exemplifyMotif()`.

1.3.2 Baseline models

This package contains various tools for computing random graphs with respect to a specific baseline model and for analysing motif distributions in these baselines.

- *Fixed Densities Model* In this model it is assumed that the number of edges between level i and level j is fixed for all i, j. The specified number of edges is chosen from the set of all possible edges. This model is identified by `sma.MODEL_FIXED_DENSITIES`.
- *Erdős-Rényi Model* In this model edges between the various levels are drawn independently at random with respect to a fixed probability p_{ij}. The (refined) Erdős-Rényi model accounts for the various levels by allowing to specify different probabilities for the different levels. This model is identified by `sma.MODEL_ERDOS_RENYI`.
- *Actor's Choice Model* In this model all edges but the edges on one level (the level of the actors) are fixed. The edges on the level of the actors are chosen indepently with fixed probability as in Erdős-Rényi. This model is identified by `sma.MODEL_ACTORS_CHOICE`.

See also *Distribution of motifs in Erdős-Rényi random graphs*, *Simulate baselines* and *Generation of random SENs*.

1.4 Data format

On the Python side, the SES' objects are stored as `networkx.Graph` objects. The `networkx` module provides a wide range of well documented functions. Most of them are applicable to SES', although SES' have the special properties that they nodes belong to either the social or the ecological type.

This distinction is done by introducing the nodal attribute `sesType` which takes integers ≥ 0 as values. For two-level networks, these values are `sma.NODE_TYPE_SOC` (1) or `sma.NODE_TYPE_ECO` (0). Please follow this naming schema in order to ensure compatibility.

1.5 Some comments on MPNet

This software is somehow compatible to MPNet [WRP09] (version as of Aug 2019). It can read files that MPNet can read as well, cf. *sma.loadMultifileSEN()*, and files that are produced in MPNet's simulation mode, cf. *sma.loadMPNetSEN()*.

MPNet distinguishes a social subsystem (A) and an ecological subsystem (B) and a system connecting A and B, the X system. We will not refer to the latter in this documentation.

The edges in a SEN can be either between two A nodes (*sma.EDGE_TYPE_SOC_SOC*), between two B nodes (*sma.EDGE_TYPE_ECO_ECO*) or in the X subsystem (*sma.EDGE_TYPE_ECO_SOC*). These constants are for example used by *sma.edgesCount()*.

Note that the patterns recognised by MPNet are different to the motifs distinguished in our terminology. Since MPNet does not care about the existence of an edge whenever it is not required by a pattern, its patterns are in a way more general. For example, in MPNet's nomenclature a Star2AX motif is a 3E-motif of class II.B or II.C, since the additional edge is not of any interest. This leads to the following selection of correspondences between our quantities and the quantities measured in MPNet:

- The number of Star2AX patterns in MPNet equals the number of 3E-motifs of class II.B plus twice the number of 3E-motifs of class II.C.
- The number of C4AXB patterns in MPNet equals the number of 4-motifs of class I.D plus the number of 4-motifs of class V.D plus twice the number of 4-motifs of class III.D.

This list could be expanded to monstrous length.

1.6 Version history

Version 1.0 of this package was developed in summer 2018. In the following year it has been continuously extended. Continuous development comes with breaking changes:

- Version 2.0: The order and types of parameters of *sma.countMotifs()* have been changed. Functions like *sma.count4Motifs()* have been adapted accordingly.
- Version 2.0.4: In order to introduce a clearer nomenclature, `sma.isType`, `sma.is3Type` and `sma.is4Type` were renamed to *sma.isClass*, *sma.is3Class* and *sma.is4Class*.

Please see Gitlab Tags and CHANGELOG for more recent developments.

CHAPTER

TWO

AN EXAMPLE: MULTI-LEVEL SENS

In this section we will see how to analyse multi-level SENs with this framework. We will go through all steps of SEN analysis, from generation of random SENs to counting motifs of different classes. In general this software focuses on two-level networks where the first level (type 0) is referred to as ecological and the second level (type 1) is called social. In contrary to that we will in this section look into the analysis of three-level networks. We will refer to the levels as

0. Actor level
1. Venue level
2. Issue level

The levels are encoded as nodal attribute (`sesType`) taking 0, 1 or 2 as its value.

The Python script that contains the described analysis can be downloaded from Gitlab. It should be executable without further setup.

2.1 Generation of random multi-level SENs

Random multi-level SENs can be computed using an adapted Erdős-Rényi model. The probabilities of the single edges are all the same. We will use `sma.randomMultiSENs()` for generating a three-level network with 10 actors, 3 venues and 7 issues. We will need to define how many edges connecting the different types of nodes should be present in the random networks. These values must be given as a upper-triangular matrix.

```
1 nEdges = numpy.array([[11, 12,  6],
2                       [ 0,  2, 10],
3                       [ 0,  0,  5]])
4 generator = sma.randomMultiSENs([10,3,7],
5                                 nEdges,
6                                 names=['Actor', 'Venue', 'Issue'])
7 G = next(generator)
```

Note that the object obtained in line 4 is a Python generator. We have to call `next` on it in order to obtain the graph object `G`, cf. *Generation of random SENs*.

For the example discussed in this section we will used a fixed (previously random) graph available in the `data/`-folder. Let's load it.

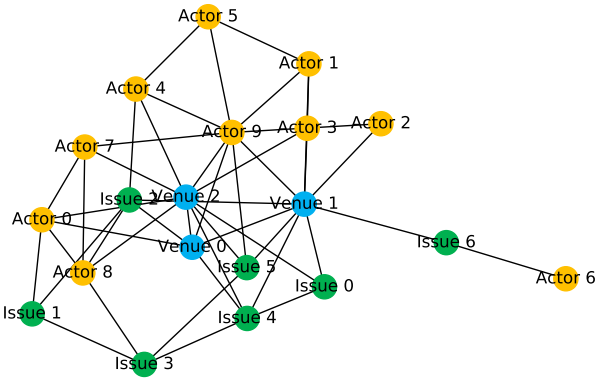
```
G = sma.loadSEN("data/dummy_multi_adj.csv",
                "data/dummy_multi_attr.csv",
                delimiter = ',')
```

2.2 Drawing a multi-level SEN

It may be useful to obtain a visual impression of the just generated random network. We will continue with the example from the previous section. For drawing the network we will use *sma.drawSEN()*.

```
cm = {0: '#ffc000', 1: '#00b0f0', 2: '#00b050'} # adapt the colors
sma.drawSEN(G, color_map = cm)
```

The result looks as follows:



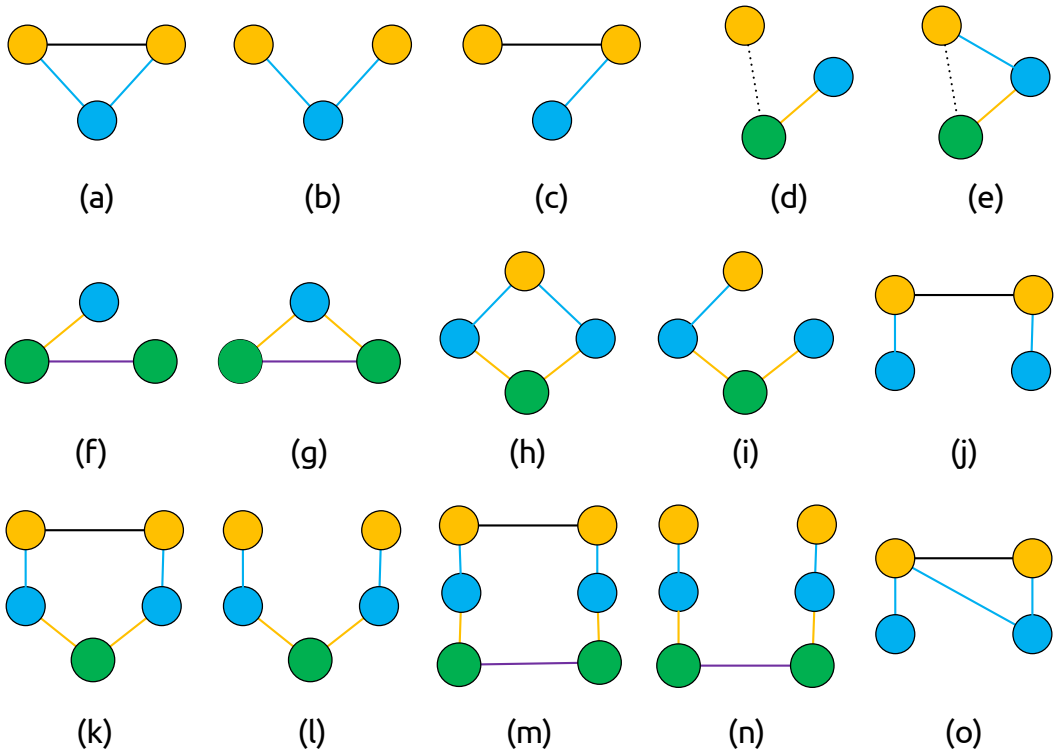
2.3 Listing motifs

Having a SEN at hand, we can now start to count the motifs in it. Motifs are tuples of nodes, which can be classified based on the edges linking the nodes. Motifs can consist of different amounts of nodes from the separate levels. These amounts are called the *signature* of the motif. In a two-level (social-ecological) network there are for example

- 4-motifs consisting of two social and two ecological nodes,
- 3E-motifs consisting of one ecological and two social nodes,
- 3S-motifs consisting of one social and two ecological nodes,
- 2-motifs consisting of one social and one ecological nodes,
- 3p-motifs consisting of three nodes of the same type.

This list could surely be prolonged. We will start with computing lists of all motifs of a given signature. In the next section we will use these lists to classify the motifs in them.

This software uses Python iterators called *Motif sources* to represent sets of motifs. In this introductory section we will look at the following motifs:



For obtaining a list of all motifs we use an instance of *sma.MultiMotifs*. As parameters we need to specify how many nodes should be taken from each level of the network. We have to call `list` on the object in order to compute the set of motifs:

```
# (a)-(c): set of all motifs consisting of two actors and one venue:
motifs1 = sma.MultiMotifs(G, 2, 1, 0)
list1 = list(motifs1) # [(('Actor 0', 'Actor 1'), ('Venue 0',), ()), ... ]

# (d)-(e): set of all motifs consisting of one actor, one venue and one issue
motifs2 = sma.MultiMotifs(G, 1, 1, 1)
list2 = list(motifs2) # [(('Actor 0',), ('Venue 0',), ('Issue 0',)), ...]

# (f)-(g): one venue, two issues
motifs3 = sma.MultiMotifs(G, 0, 1, 2)
list3 = list(motifs3) # [((), ('Venue 0',), ('Issue 0', 'Issue 1')), ...]

# (h)-(i): one actor, two venues, one issue
motifs4 = sma.MultiMotifs(G, 1, 2, 1)
list4 = list(motifs4) # [(('Actor 0',), ('Venue 0', 'Venue 1'), ('Issue 0',)), ...]
```

Some technical remarks: The motifs provided by *sma.MultiMotifs* are Python tuples consisting of one entry for each level of the network. The entries are themselves tuples consisting of the nodes. Sometimes these tuples are empty, i.e. `()`. The internal mechanisms of *sma.MultiMotifs* are fundamentally different from those of the classes for two-level SENs, e.g. *sma.FourMotifs*. Most tools are not compatible. This should not bother us here.

How many motifs did we obtain? The motifs for (h)-(i) are now stored in `list4`. Its length is 210. This is what we would expect since $\binom{10}{1}\binom{3}{2}\binom{7}{1} = 210$.

2.4 Classifying motifs

Every signature of motifs comes with different classes. All motif classes supported by this software are listed in *Appendix: The Motif Zoo*. This software uses objects called *Motif classificators* for classifying motifs. Since we are interested in motifs spanning across more than two levels, we will in particular be using *sma.MultiMotifClassificator*.

First we want to see how many motifs in `list1` are of classes (a), (b) and (c). Since here only two levels of the SEN are involved, namely the actors level and the venues level, *sma.MultiMotifClassificator* falls back to *sma.ThreeMotifClassificator*, a classificator for 3-motifs in two-level SENs. The possible classes outlined in *Terminology*. (a) correspondes to II.C, (b) to I.C and (c) to II.B. Again we need to specify how many nodes are taken from each level of the network.

The classificator object can then be used to classify individual motifs or lists of motifs using `map()`. Motifs can be counted using *sma.countMotifs()*.

```
classificator = sma.MultiMotifClassificator(G, 2, 1, 0)
classificator((('Actor 0', 'Actor 9'), ('Venue 0',), ())) # I.C = (b)
classes = list(map(classificator, list1)) # ['I.B', 'I.B', 'I.B', 'I.B', 'I.B' ...]

counts210 = sma.countMotifs(G,
                            sma.MultiMotifClassificator(G, 2, 1, 0),
                            sma.MultiMotifs(G, 2, 1, 0))
# {'I.A': 40, 'I.B': 48, 'I.C': 14, 'II.A': 9, 'II.B': 16, 'II.C': 8}

# or shortened:
counts210 = sma.countMultiMotifs(G, 2, 1, 0) # same output as before
```

In other words, (a) occurs 8 times in the example, (b) 14 times and (c) 16 times. We can count the occurrences of patterns (f) and (g) analogously: (f) corresponds to II.B while (g) corresponds to II.C.

```
counts012 = sma.countMultiMotifs(G, 0, 1, 2)
# {'I.A': 12, 'I.B': 24, 'I.C': 12, 'II.A': 5, 'II.B': 8, 'II.C': 2}
```

Note that *sma.MultiMotifClassificator* detects automatically what the signature and the roles of the motifs are. Based on the parameters `0, 1, 2` it automatically chooses the venue level as the origin of the distinct node in the motif described in *Terminology*.

We continue with (d) and (e). (d) corresponds to `6`, (e) to `7`. Apart from classifying all motifs we extract lists of motifs of the two interesting classes:

```
counts111 = sma.countMultiMotifs(G, 1, 1, 1)
# {0: 63, 1: 36, 2: 53, 3: 40, 4: 5, 5: 6, 6: 5, 7: 2}

motifsD = list(sma.MultiMotifs(G, 1, 1, 1)
               & sma.isClass(sma.MultiMotifClassificator(G, 1, 1, 1), 6))
# [(('Actor 1',), ('Venue 1',), ('Issue 5',)),...]

motifsE = list(sma.MultiMotifs(G, 1, 1, 1)
               & sma.isClass(sma.MultiMotifClassificator(G, 1, 1, 1), 7))
# [(('Actor 4',), ('Venue 0',), ('Issue 2',)),...]
```

For motif classes (j) and (o) need to specify `2, 2, 0` as parameter for the arities. *sma.MultiMotifClassificator* uses then an instance of *sma.FourMotifClassificator* for classifying the motifs. The classification follows Ö. Bodin, M. Tengo (2012). (j) corresponds to I.B, (o) corresponds to V.B.

```
counts220 = sma.countMultiMotifs(G, 2, 2, 0)
print('Motifs of class (j): %d' % counts220['I.B']) # 0
print('Motifs of class (o): %d' % counts220['V.B']) # 5
```

We treat the remaining cases of (h), (i), (k)-(n) analogously. Note that this software provides only a partial classification for motifs of these signatures. (h) is classified as 2, (i) as 1, (k) corresponds to I.B.2 while (l)

corresponds to I.A.2. (m) is classified as 2, (n) as 1. Motifs not matching these classes are classified either as -1 or as Unclassified.

```
# (h) and (i)
counts121 = sma.countMultiMotifs(G, 1, 2, 1)
print('Motifs of class (h): %d' % counts121[2]) # 5
print('Motifs of class (i): %d' % counts121[1]) # 18
# Which motif is of class (h)? Let's draw it!
motif = next(sma.MultiMotifs(G, 1, 2, 1)
             & sma.isClass(sma.MultiMotifClassificator(G, 1, 2, 1), 2))
subgraph = G.subgraph(motif[0]+motif[1]+motif[2])
sma.drawSEN(subgraph, color_map = cm, pos = sma.layer_layout(subgraph))

# (k) and (l)
counts221 = sma.countMultiMotifs(G, 2, 2, 1)
print('Motifs of class (k): %d' % counts221['I.B.2']) # 0
print('Motifs of class (l): %d' % counts221['I.A.2']) # 24

# (m) and (n)
counts222 = sma.countMultiMotifs(G, 2, 2, 2)
print('Motifs of class (m): %d' % counts222[2]) # 0
print('Motifs of class (n): %d' % counts222[1]) # 0
```

We finally give an overview over the motifs studied here, their classificators and the terminology used. This should also serve as part of the documentation of *sma.MultiMotifClassificator*.

Motif in the figure	Classificator	Classificator's terminology
Motif (a)	2, 1, 0	II.C
Motif (b)		I.C
Motif (c)		II.B
Motif (d)	1, 1, 1	6
Motif (e)		7
Motif (f)	0, 1, 2	II.B
Motif (g)		II.C
Motif (h)	1, 2, 1	2
Motif (i)		1
Motif (j)	2, 2, 0	I.B
Motif (o)		V.B
Motif (k)	2, 2, 1	I.B.2
Motif (l)		I.A.2
Motif (m)	2, 2, 2	2
Motif (n)		1

See also *Appendix: The Motif Zoo* for a full list of all motif classes.

CHAPTER

THREE

TERMS FOR ERGM

This section describes the terms which have been added to R's `ergm` package in order support the analysis of SES. See the documentation of the R packages `statnet`, `ergm` for `ergm.userterms` for mathematical and technical details.

All terms work only for undirected graphs. All of them have a `typeAttr` parameter which must be set to the name of the nodal attribute which states whether a node is ecological or social. This attribute must be either 0 for ecological nodes or 1 for social nodes. This guarantees compatability with the Python functions described in the other sections. On the Python side, this parameter is per default called `sesType`. Following this naming convention is strongly recommended.

The following terms are included in this package:

- `edgesPerDomain(typeAttr = 'sesType')` adds three change statistics counting the number of edges in each "domain", i.e. number of edges linking social and social, ecological and ecological and social and ecological nodes. See *`sma.edgesCount()`* for a similar Python function.
- `openTriangles(pointType, typeAttr = 'sesType')` adds one change statistic counting the number of open triangles (3-motifs I.C). `pointType` is the social-ecological type of the distinct node at the point of the triangle. Set this parameter to 0 to count the 3E-motif I.C and to 1 to count the 3S-motif I.C.
- `closedTriangles(pointType, typeAttr = 'sesType')` same as `openTriangles`, but for counting II.C motifs (closed triangles).
- `threeMotifs(pointType, typeAttr = 'sesType')` adds 6 change statistics for the number of all 3-motifs (I.A, . . . , II.C). `pointType` determines the social-ecological type of the distinct node. Set it to 0 to get the counts of 3E-motifs or to 1 to get the counts of 3S-motifs. Note, that this is a brute-force implementation. If you are just interested in the number of open or closed triangles (I.C or II.C motifs), use the terms `openTriangles` or `closedTriangles` which are implemented much more elegantly.
- `threeMotif(pointType, class, typeAttr = 'sesType')` adds one change statistic for the number of the 3-motifs specified by `class` (as a string, e.g `'I.C'`). `pointType` determines the social-ecological type of the distinct node. Set it to 0 to get the counts of 3E-motifs or to 1 to get the counts of 3S-motifs. Note, that this is a brute-force implementation. If you are just interested in the number of open or closed triangles (I.C or II.C motifs), use the terms `openTriangles` or `closedTriangles` which are implemented much more elegantly. Using this term is not more efficient than using `threeMotifs` because all motifs are classified in either case. Cf. *`sma.count3EMotifs()`* and *`sma.count3SMotifs()`*.
- `fourMotifs(typeAttr = 'sesType')` adds 28 change statistics counting the number of all 4-motifs (I.A, II.B, . . . , VII.D). Note, that this term is rather inefficiently implemented. In most cases, calling *`sma.count4Motifs()`* will be faster.
- `fourMotif(class, typeAttr = 'sesType')` adds one change statistic counting the number of motifs of the class defined by `class` (as a string, e.g `'I.C'`). Thus, this term returns one of the values computed by `fourMotifs`. Due to the complex nature of the motifs, using this term is not faster than computing the counts for all motifs. This term may be used for model fitting.

The terms can be used in all steps for ERGM modelling. For example, an ERGM model can be fitted using the terms:

```
library(ergm.userterms)

# let net be some network
model <- ergm(net ~ edgesPerDomain())
```

The terms can also be used for computing the network characteristics they represent:

```
# let net be some network
summary(net ~ fourMotifs())
```

CHAPTER

FOUR

INPUT/OUTPUT

This module provided input/output functions for reading and writing SENs in multiple formats using CSV files. Be aware, that the networkx package provides a wide range of other sophisticated input and output facilities.

Here functions for interacting with R's statnet are not listed. See *sma.translateGraph()* and the respective section for more details on this.

Most input functions will detect inconsistencies in the input data only to a certain extent. For example, it may happen that when inconsistent files are read nodes without `sesType`-attribute are added to the graph. Such nodes let many function used in analyses swallop up. Use *sma.untypedNodes()* to detect them. Statistical functions such as *sma.edgesCount()*, *sma.nodesCount()* or *sma.edgesCountMatrix()* (for multilevel SENs) can help to verify that data has been read correctly.

4.1 Input

`sma.`**`loadSEN`**(*adjacencyMatrix: str*, *types: str*, *delimiter: str = ';'*, *create_using: networkx.classes.graph.Graph = None*) → networkx.classes.graph.Graph

Loads a SEN in Madagascar format.

All parameters specifiy file names of the necessary CSV files.

A file stored at `types` might look like:

```
;soc
Ocean;0
Lake;0
City;1
```

A file stored at `adjacencyMatrix` might look like:

```
;Ocean;Lake;City
Ocean;0;1;0
Lake;0;0;1
City;0;0;0
```

See also *sma.loadDiSEN()*.

Return type `Graph`

Parameters

- **`adjacencyMatrix`** (`str`) – file name of the CSV file containing the adjacency matrix. The first line and the first row are skipped / treated as headings.
- **`types`** (`str`) – file name of the CSV file containing the social-ecological type encoded as either `0` or `1` for all nodes
- **`delimiter`** (`str`) – CSV delimiter, default is `;`.
- **`create_using`** (`Graph`) – None or an instance of `networkx.Graph` to create the network from. Use `networkx.DiGraph()` for obtaining a digraph.

Returns the graph containing all given data

sma.**loadDiSEN**(*adjacencyMatrix: str*, *types: str*, *delimiter: str = ';'*, *create_using: networkx.classes.graph.Graph = None*) → networkx.classes.digraph.DiGraph
Synonym of *sma.loadSEN()* for loading directed SENs.

sma.**loadMultifileSEN**(*adjacencyA: str*, *adjacencyB: str*, *adjacencyX: str*, *attributesA: list = []*, *attributesB: list = []*, *attributesX: list = []*) → networkx.classes.graph.Graph
Loads a SEN in Woodfire format.

All parameters specifiy file names of the necessary files.

Return type Graph

Parameters

- **adjacencyA** (str) – adjacency matrix for the social subsystem
- **adjacencyB** (str) – adjacency matrix for the ecological subsystem
- **adjacencyX** (str) – adjacency matrix for the links between the social and the ecological subsystems
- **attributesA** (list) – list of file names of CSV files containing nodal attributes for the social nodes
- **attributesB** (list) – list of file names of CSV files containing nodal attributes for the ecological nodes
- **attributesX** (list) – list of file names of CSV files containing nodal attributes for both social and ecological nodes. Social nodes are read first. Thus, their attributes must be listed at the beginning of the file

Returns the graph containing all given data

Raises **AssertionError** – if the dimensions of the files do not match

sma.**loadMPNetSEN**(*file: str*) → networkx.classes.graph.Graph
Loads a SEN generated by MPNet (version as of Aug 2019), e.g. in its simulation mode.

This function expects an M-file, e.g. thenetwork_Network_M_42.txt which contains an edge list. Graphs stored in MPNet's input format can be read using *sma.loadMultifileSEN()*.

MPNet's files are supposed to be Pajek-compatible although they cannot be read by networkx.read_pajek(). Boxes are translated to social nodes, ellipses to ecological nodes.

A file stored at file might look like:

```
*vertices 118
1 "" box ic Blue bc Black
2 "" box ic Blue bc Black
...
109 "" ellipse ic Red bc Black
110 "" ellipse ic Red bc Black
111 "" ellipse ic Red bc Black
*edges
1 13 1 c Blue
1 28 1 c Blue
1 36 1 c Blue
...
```

Return type Graph

Parameters **file** (str) – file name of the MPNet graph file

Returns the SEN represented by the file

4.2 Output

sma.**saveSEN**(*G: networkx.classes.graph.Graph*, *adjacencyFile: str*, *attributesFile: str*, ***kwargs*)

Stores a graph object in two files: a csv adjacency matrix and a csv table containing the nodal attributes. Use kwargs to pass arguments to pandas' to_csv facility.

For some reason, panda does not like it when the nodes are of different data types. Do not use e.g. integers and string simultaneously to name the nodes.

Parameters

- **G** (Graph) – the graph to save
- **adjacencyFile** (str) – file name for the adjacency matrix
- **attributesFile** (str) – file name for the nodal attributes (corresponds to attributesX in *sma.loadMultifileSEN()*).
- **kwargs** – additional parameters for pandas.DataFrame.to_csv().

CHAPTER

FIVE

GENERATION OF RANDOM SENS

See also Distribution of motifs in Erdős-Rényi random graphs and *Simulate baselines*.

Graphs can be generated randomly for statistical analysis. For answering more sophisticated questions the set of possible graphs can be refined, for example to only those graphs with a certain amount of edges per between specific levels.

Several baseline models are implemented in this package:

- *sma.MODEL_FIXED_DENSITIES* random graphs contain a fixed number of edges on the different levels and between the various levels. Most functions are based on this model, e.g. *sma.randomSimilarSENs()*, *sma.randomMultiSENsFixedDensities()*, etc.
- *sma.MODEL_ERDOS_RENYI* random graphs whose edges are chosen independently at random with respect to fixed probabilities. The refined Erdős-Rényi model used here accounts for the multitude of levels by working with separate edge probabilities for each pair of level. See *sma.randomMultiSENsErdosRenyi()*.
- *sma.MODEL_ACTORS_CHOICE* random graphs whose edges are chosen only on one level at random (there according to *sma.MODEL_ERDOS_RENYI*). The other levels are fixed.

Random graphs drawn from a less refined baseline model (no control over edges, or just overall density) can be generated using *sma.randomSENs()*.

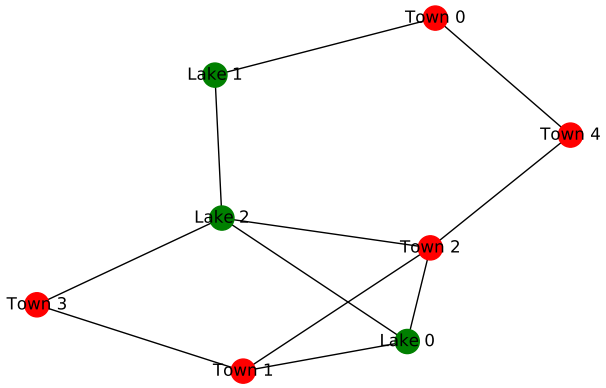
Technical remark Note that all functions in this sections are from the technical point of view Python generators. They do not return a single graph object but a generator object which can provide (infinitely) many graph objects. Have a look at the following example:

```
import sma

# get one random graph
G = next(sma.randomSENs(10,5))

# get a list of 10 random graphs
import itertools
n = 10
graphList = list(itertools.islice(sma.randomSENS(10,5), n)

# generate and draw random graph, cf. figure:
sma.drawSEN(next(sma.randomSENs(3, 5, 0.2)))
```

5.1 Random SENs from scratch

Most simply, randoms SENs can be generated based on elementary properties, e.g. the number of social / ecological nodes, the density or the number of edges per domain.

Example: Random network with three ecological and five social nodes, density 0.2.



sma.**randomSENs**(*ecoNodes: int*, *socNodes: int*, *density: float = None*, *ecoName: str = 'Lake'*, *socName: str = 'Town'*)

Generator which yields random graphs with a certain amount of ecological and social nodes. Optionally, a densitiy can be specified.

See also `sma.randomDiSENs()`.

Parameters

- **ecoNodes** (int) – number of ecological nodes
- **socNodes** (int) – number of social nodes
- **density** (float) – densitiy, i.e. number of edges / number of possible edges, cf. `sma.density()`. Set to None if the density does not matter
- **ecoName** (str) – name for the ecological nodes, i.e. 'Lake' for getting names like 'Lake 0', 'Lake 1' etc.
- **socName** (str) – name for the social nodes, i.e. 'Town' for getting names like 'Town 0', 'Town 1' etc.

sma.**randomSpecialSENs**(*ecoNodes: int*, *socNodes: int*, *edgesCounts: dict*, *ecoName: str = 'Lake'*, *socName: str = 'Town'*)

Generator which yields random graphs with a certain amount of ecological and social nodes and with certain amounts of social-social, ecological-ecological and social-ecological edges. See `sma.edgesCount()`.

This function implemented `sma.MODEL_FIXED_DENSITIES`.

Parameters

- **ecoNodes** (int) – amount of ecological nodes
- **socNodes** (int) – amount of social nodes
- **edgesCounts** (dict) – dict containing the amounts of social-social, ecological-ecological and social-ecological edges, labelled in accordance with `sma.edgesCount()`.
- **ecoName** (str) – name for the ecological nodes, i.e. 'Lake' for getting names like 'Lake 0', 'Lake 1' etc.
- **socName** (str) – name for the social nodes, i.e. 'Town' for getting names like 'Town 0', 'Town 1' etc.

sma.**randomDiSENs**(*ecoNodes: int*, *socNodes: int*, *density: float = None*, *ecoName: str = 'Lake'*, *socName: str = 'Town'*)

Generator which yields random directed graphs with a certain amount of ecological and social nodes. Optionally, a densitiy can be specified.

See also *sma.randomSENs()*.

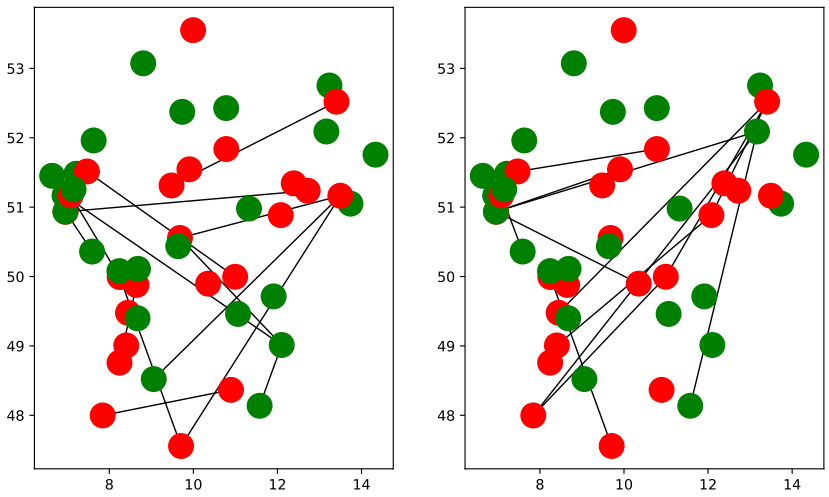
Parameters

- **ecoNodes** (int) – number of ecological nodes
- **socNodes** (int) – number of social nodes
- **density** (float) – densitiy, i.e. number of edges / number of possible edges, cf. *sma.density()*. Set to None if the density does not matter
- **ecoName** (str) – name for the ecological nodes, i.e. 'Lake' for getting names like 'Lake 0', 'Lake 1' etc.
- **socName** (str) – name for the social nodes, i.e. 'Town' for getting names like 'Town 0', 'Town 1' etc.

5.2 Random SENs based on given SENs

For more complex statistical analysis it may be desirable to compute a set of random graphs which shares certain properties with a given SEN. These properties can again be the number of social / ecological nodes or the density. Furthermore, random SENs which inherit all nodal attributes can be generated. This is especially useful when analysing the distribution of nodal properties in motifs.

Example: Consider a SEN which contains major German cities as nodes (the edges are chosen randomly). The network is not only aware of the social ecological type of the nodes (here chosen randomly) but also of their geographical location. Then *sma.randomSimilarAttributedSENs()* can be used to generate a new SEN of the same cities at their original location but with new random edges. The following figure shows the original SEN and a randomly generated one plotted with *sma.drawGeoSEN()*.



sma.**randomSimilarSENs**(*G: networkx.classes.graph.Graph*, *ecoName: str = 'Lake'*, *socName: str = 'Town'*)

Generator which yields random graphs similar to the given graph, w.r.t. the number of social and ecological nodes and the number of edges in the domains.

This function implemented *sma.MODEL_FIXED_DENSITIES*.

See *sma.edgesCount()* and *sma.randomSpecialSENs()*. See *sma.randomSimilarAttributedSENs()* for "more similar" random SENs.

Parameters

- **G** (Graph) – the graph the random graphs shall be similar to.
- **ecoName** (str) – name for the ecological nodes, i.e. 'Lake' for getting names like 'Lake 0', 'Lake 1' etc.
- **socName** (str) – name for the social nodes, i.e. 'Town' for getting names like 'Town 0', 'Town 1' etc.

sma.**randomSimilarAttributedSENs**(*G: networkx.classes.graph.Graph*)

Generator which yields random graphs similar to the given graph, w.r.t. all nodal attributes. The nodal data is extracted and a random SENs with equal amount of edges in each domain is generated.

This function implemented *sma.MODEL_FIXED_DENSITIES*.

Parameters **G** (Graph) – the graph the random graphs shall be similiar to.

5.3 Multi-level SENs

The front-end function *sma.randomSimilarMultiSENs()* can be used to call various generators for several baseline models listed in this section.

sma.**randomSimilarMultiSENs**(*G: networkx.classes.graph.Graph*, *names: list = ['Lake', 'Town', 'Issue', 'System', 'Entity', 'Item', 'Universe']*, *model: str = 'fixed_densities'*, *level: int = - 1*)

Generates random multi-level SENs "similar" to the given SEN. This function is a front-end for several generator functions which are chosen based on the given model parameter:

- *sma.MODEL_FIXED_DENSITIES*: see *sma.randomMultiSENsFixedDensities()*
- *sma.MODEL_ERDOS_RENYI*: see *sma.randomMultiSENsErdosRenyi()*
- *sma.MODEL_ACTORS_CHOICE*: see *sma.randomMultiSENsActorsChoice()*

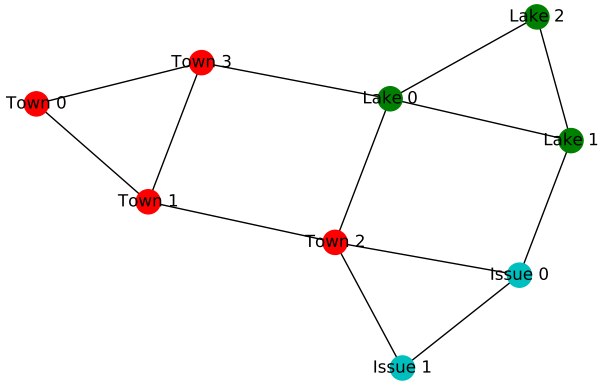
Parameters

- **G** (Graph) – the SEN
- **names** (list) – names for the classes of nodes, default *sma.MULTI_DEFAULT_NAMES*
- **model** (str) – baseline model, default *sma.MODEL_FIXED_DENSITIES*.
- **level** (int) – sesType of the variable level if model is set to *sma.MODEL_ACTORS_CHOICE*.

Returns generator yielding random SENs

Raises **NotImplementedError** – if no generators for the given model are implemented

The random multi SEN in the example in *sma.randomMultiSENsFixedDensities()* could look as follows:



sma.**randomMultiSENsFixedDensities**(*nVertices: list*, *nEdges: numpy.ndarray*, *names: list = ['Lake', 'Town', 'Issue', 'System', 'Entity', 'Item', 'Universe']*)

Generator for multi-level SENs.

This function implemented *sma.MODEL_FIXED_DENSITIES()*.

Example: Generate a SEN with 3 ecological nodes, 4 social actors and 2 issues. Then `nVertices` must be set to `[3,4,2]`. The number of edges is given by the matrix in `nEdges`. For a given multi-level SEN such a matrix can be computed using *sma.edgesCountMatrix()*. Entry (i, j) contains the number of edges connected vertices of types i and j.

```
import sma, numpy
nEdges = numpy.array([[3,2,1],[0,4,2],[0,0,1]])
# 3 edges from eco to eco, 2 edges from eco to soc, 1 edge from eco to issue
# 4 edges from soc to soc, 2 edges from soc to issue, 1 edge from issue to␣
↪issue

G = next(sma.randomMultiSENs([3,4,2], nEdges))
```

Given `nVertices`, *sma.randomEdgeCountMatrix()* can be used to compute a random admissible matrix for `nEdges`:

```
nVertices = [3,4,2]
G = next(sma.randomMultiSENs(nVertices, sma.randomEdgeCountMatrix(nVertices)))
```

Parameters

- **nVertices** (`list`) – a list of integers specifying how many nodes of each type should be generated
- **nEdges** (`ndarray`) – an upper-triangular matrix specifying how many random edges connecting nodes of certain type should be generated
- **names** (`list`) – names of the nodes of a certain type, cf. *sma.MULTI_DEFAULT_NAMES*.

sma.**randomMultiSENsErdosRenyi**(*nVertices: list*, *nEdges: numpy.ndarray*, *names: list = ['Lake', 'Town', 'Issue', 'System', 'Entity', 'Item', 'Universe']*)

Generator for random graphs drawn from an Erdős-Rényi model. The probabilities for the edges linking the various levels are extracted from `nEdges`. This function implements *sma.MODEL_ERDOS_RENYI*.

Parameters

- **nVertices** (`list`) – vector indicating the numbers of vertices on the various levels
- **nEdges** (`ndarray`) – matrix indicating numbers of edges, see *sma.edgesCountMatrix()*.
- **names** (`list`) – names for the nodes on the various levels, default *sma.MULTI_DEFAULT_NAMES*.

sma.**randomMultiSENsActorsChoice**(*G: networkx.classes.graph.Graph*, *level: int*)

Generator for random graphs drawn from an Actor's Choice mode. The probability for the edges on the variable level is extracted from the given network. This function implements *sma.MODEL_ACTORS_CHOICE*.

Parameters

- **G** (`Graph`) – the SEN
- **level** (`int`) – `sesType` of the variable level

CHAPTER

SIX

ANALYSE SENS, COUNT AND CLASSIFY MOTIFS

6.1 Count motifs

6.1.1 Front-end functions

sma.**countMotifsAuto**(*G: networkx.classes.graph.Graph*, **motifs*, *assume_sparse: bool = True*, ***kwargs*)

Front-end function for all counting functions in sma. This function determines automatically an ideal counter (sparse or dense) for a set of motifs. This reduces overhead and leads to an optimisation.

```
# let G by a SEN
# count open and closed social and ecological triangles
partial, total = sma.countMotifsAuto(G, "1,2[I.C]", "1,2[II.C]", "2,1[I.C]",
↪"2,1[II.C]")
print(partial)
print(total)
```

This function works for directed and undirected graphs. The directed case is distinguished by calling `networkx.is_directed()` on the given graph.

Parameters

- **G** (`Graph`) – the SEN
- **motifs** – list of motif identifier, see documentation
- **assume_sparse** (`bool`) – whether the network shall be assumed to be sparse
- **kwargs** – more arguments for the counter

Returns

tuple consisting of:

- a dict with the motif identifiers as keys and the corresponding counts as values
- a dict with all counts collected

Raises **ValueError** – in case of unrecognisable motif identifiers

sma.**countMultiMotifs**(*G: networkx.classes.graph.Graph*, **arities*, *roles=[]*, *iterator=None*, ***kwargs*)

Front-end function for counting multi-motifs. The default iterator is *sma.MultiMotifs*. As classificator serves *sma.MultiMotifClassificator*. All options of *sma.countMotifs()* are available.

Call *sma.supportedSignatures()* for an overview of all supported signatures.

This function works for directed and undirected graphs. The directed case is distinguished by calling `networkx.is_directed()` on the given graph.

See also *sma.countMultiMotifsSparse()*.

Parameters

- **G** (Graph) – the graph
- **arities** – arities for the motif source and the classificator
- **roles** – roles of the level, cf. *sma.MultiMotifClassificator*.
- **iterator** – a custom iterator, default is *sma.MultiMotifs*.
- **kwargs** – additional parameters for *sma.countMotifs()*

sma.**countMultiMotifsSparse**(*G: networkx.classes.graph.Graph*, **arities*, *roles=[]*, ***kwargs*)

Front-end function for several counting functions for sparse networks. It supports the same parameters for defining the motifs (arities, roles) as other multi-level counting functions. Please check the documentations of the respective functions as some of them might return partial counts only. A full list of supported motif signatures can be accessed via *sma.supportedSignatures()*.

This function works for directed and undirected graphs. The directed case is distinguished by calling networkx.is_directed() on the given graph.

See also *sma.countMultiMotifs()*.

Parameters

- **G** (Graph) – the SEN
- **arities** – arities of the motifs, cf. *sma.MultiMotifClassificator*.
- **roles** – roles of the levels, cf. *sma.MultiMotifClassificator*.
- **kwargs** – additional parameters for the counter functions, see above.

6.1.2 Dense networks

This section lists the most basic counting functions.

sma.**countMotifs**(*G: networkx.classes.graph.Graph*, *classificator: sma.classify.MotifClassificator*, *iterator: sma.iterate.MotifIterator*, *processes: int = 0*, *norm: bool = False*, *array: bool = False*, *chunksize: int = 10000*, *progress_report: bool = False*, *progress_total: int = - 1*)

General function for classifying and counting motifs.

Procedure:

1. motifs are drawn from the iterator
2. classificator is called for every motif
3. the number of occurrences is computed for every class defined in classes

Note, that this function provides for must cases to many options. See *sma.count4Motifs()*, *sma.count3EMotifs()* and *sma.count3SMotifs()*.

This function supports multiprocessing. Set processes to any value other than zero to use several processes to solve the task. Set processes to None to use multiprocessing.pool.Pool's default value.

Set norm to True to get normalized counts, i.e. in this case the sum of all counts equals one.

Set array to True to get an array where the indices match with the indices of the classes in classes instead of a dict mapping classes to their respective counts.

Parameters

- **G** (Graph) – the graph
- **classificator** (MotifClassificator) – a *sma.MotifClassificator* for classifying the motifs
- **iterator** (MotifIterator) – an iterator yielding motifs. Note, that the format of the motifs must be somehow compatible with the format used by the classificator. See *sma.iterate4Motifs()* etc.

- **processes** (int) – number of processes the task is split among for multiprocessing. Set processes to zero to disable multiprocessing.
- **chunksize** (int) – in multiprocessing mode size of each chunk that is sent to a child
- **norm** (bool) – whether the counts should be normalized in the way that they sum up to one.
- **array** (bool) – whether the results should be returned as array (indices map the indices in classes) or as dict mapping a class to its count
- **progress_report** (bool) – whether a progress report should be printed
- **progress_total** (int) – total number of motifs to count for progress report

Returns number of occurrences of the specified classes in either an array or a dict, cf. parameter array

sma.**count4Motifs**(*G: networkx.classes.graph.Graph*, *level0=1*, *level1=0*, ***kwargs*)
Front-end function for counting 4-motifs in a given graph. If iterator is None (default) *sma.iterate4Motifs()* is used.

See *sma.countMotifs()* for details. See also *sma.count4MotifsSparse()*.

Parameters

- **G** (Graph) – the graph
- **level0** (int) – sesType of the lower level
- **level1** (int) – sesType of the upper level
- **kwargs** – additional parameters for *sma.countMotifs()*

sma.**count3Motifs**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, ***kwargs*)
Front-end function for counting 3-motifs in a given graph.

See *sma.countMotifs()* for details. See also *sma.count3MotifsSparse()*.

Parameters

- **G** (Graph) – the graph
- **level0** – sesType of the lower level
- **level1** – sesType of the upper level
- **kwargs** – additional parameters for *sma.countMotifs()*

sma.**count3EMotifs**(*G: networkx.classes.graph.Graph*, *iterator=None*, ***kwargs*)
Front-end function for counting 3E-motifs (distinct ecological node) in a given graph. If iterator is None (default) *sma.iterate3EMotifs()* is used.

See *sma.countMotifs()* for details. See also *sma.count3MotifsSparse()*.

Parameters

- **G** (Graph) – the graph
- **iterator** – a custom iterator, if None (default) *sma.iterate3EMotifs()* is used.
- **kwargs** – additional parameters for *sma.countMotifs()*

sma.**count3SMotifs**(*G: networkx.classes.graph.Graph*, *iterator=None*, ***kwargs*)
Front-end function for counting 3S-motifs (distinct social node) in a given graph. If iterator is None (default) *sma.iterate3SMotifs()* is used.

See *sma.countMotifs()* for details. See also *sma.count3MotifsSparse()*.

Parameters

- **G** (Graph) – the graph

- **iterator** – a custom iterator, if None (default) *sma.iterate3SMotifs()* is used.
- **kwargs** – additional parameters for *sma.countMotifs()*

sma.**count3pMotifsLinalg**(*G: networkx.classes.graph.Graph*, *level*, *array: bool = False*, *progress_report: bool = False*)

Counter for plain 3-motifs using linear algebra.

This function counts plain 3-motifs not by enumerating and classifying all of them but by using linear algebra.

1. The adjacency matrix A of the level of interest is extracted.
2. The adjacency matrix powers A^2 and A^3 are computed.
3. The number of motifs of type 3 (closed triangles) is given by $\mathrm{tr}(A^3)/6$.
4. The number of motifs of type 2 (3-paths) is given by the sum of those upper triangular entries of A^2 where A is zero.
5. The number of motifs of type 1 (2-cliques) is given by the difference of total number of edges times the number of vertices minus 2 and twice the number of type-2-motifs and the three times the number of 3-motifs.
6. The number of motifs of type 0 (cocliques) is the remaining number of motifs.

The function runs in subcubic time in the number of vertices.

Parameters

- **G** (Graph) – the network
- **level** – sesType of the level
- **array** (bool) – whether an array should be returned
- **progress_report** (bool) – dummy parameter (progress reports not supported by ad hoc counters)

sma.**count2pMotifs**(*G: networkx.classes.graph.Graph*, *level*, *array: bool = False*, *progress_report: bool = False*)

Counts the number of plain 2-motifs by simply accessing the number of vertices and edges in the network on the respective level.

Parameters

- **G** (Graph) – the network
- **level** – sesType of the level
- **array** (bool) – whether an array should be returned
- **progress_report** (bool) – dummy parameter (progress reports not supported by ad hoc counters)

6.1.3 Sparse networks

All functions based on *sma.countMotifs()*, e.g. *sma.count3Motifs()*, iterate over all vertices to classify the motifs. For example, when classifying motifs consisting of n vertices the execution time of these functions is roughly $O(|V|^n)$. This is rather inefficient in most applications since networks in nature tend to be sparse, i.e. have only few edges.

Some functions return partial results only. For example, *sma.count4MotifsSparse()* with default parameters cannot compute correct counts for motifs IV.A-IV.D. Read the descriptions carefully.

Warning: The input graph must not contain self loops. Check using `networkx.nodes_with_selfloops()`.

sma.**count3MotifsSparse**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, *array: bool = False*, *progress_report: bool = False*)

Alternative function for counting 3-motifs optimised for sparse SENs, i.e. networks with $|E| \ll |V|^2$ for $|E|$ number of edges, V number of vertices.

This function counts 3-motifs with an asymptotic execution time of $O(|E||V|)$. Multilevel networks are supported. Use parameters `level0` (resp. `level1`) to specify the `sesType` of the distinct node (resp. the other nodes).

The performance improvement is achieved by iterating over all cross-level edges and in an inner loop over all vertices in `level1` distinct from the node contained in the edge. Then the following arithmetic steps are performed:

- the number of I.C and II.C motifs is halved since they are counted twice due to their symmetry
- the total number of edges that have been already encountered in the loop described above is computed. Here *sma.MOTIF3_EDGES* is used.
- it remains only to adjust the number of I.A (no edges at all) and of I.C (one edge in `level1` only) motifs. The number of I.C motifs is the difference of the total number of `level1`-`level1` edges counted with multiplicity and the number of already encountered edges of this type
- the number of I.A motifs is the total number of 3-motifs, cf. *totalMultiMotifs()*, and the amount of motifs already encountered

The penultimate step is based on the following observation: There are six different classes of 3-motifs. Let $m \in \mathbb{N}^6$ be a row vector denoting the numbers of occurences of the motif classes. Let $C \in \mathbb{N}^{6\times 2}$ denote the matrix in *sma.MOTIF3_EDGES* containing one row for every motif class with two entries each. The first entry represents the number of `level1` edged contained in this motif, the second the number of `level0` edges. Let V_i denote the set of vertices in level i, $E_{i,j}$ the set of edges from level i to level j. Then we have the following equalities

$$(mC)_0 = |V_0||E_{1,1}|, \quad (mC)_1 = (|V_1| - 1)|E_{0,1}|.$$

Both follow using a double counting principle.

Warning: The input graph must not contain self loops. Check using `networkx.nodes_with_selfloops()`.

See also *sma.count3Motifs()*.

Parameters

- **G** (`Graph`) – the SEN
- **level0** – `sesType` of the distinct node
- **level1** – `sesType` of the other nodes
- **array** (`bool`) – whether the output should be an array or a dict
- **progress_report** (`bool`) – whether a progress report should be printed

Returns array or dict with numbers of 3-motifs

sma.**count3SMotifsSparse**(*G: networkx.classes.graph.Graph*, ***kwargs*)

Wrapper for *sma.count3MotifsSparse()*. :type G: `Graph` :param G: the SEN :param kwargs: optional arguments for *sma.count3MotifsSparse()*.

sma.**count3EMotifsSparse**(*G: networkx.classes.graph.Graph*, ***kwargs*)

Wrapper for *sma.count3MotifsSparse()*. :type G: `Graph` :param G: the SEN :param kwargs: optional arguments for *sma.count3MotifsSparse()*.

sma.**count4MotifsSparse**(*G: networkx.classes.graph.Graph*, *level0=1*, *level1=0*, *guess_IV_A: int = - 1*, *array: bool = False*, *progress_report: bool = False*)

Alternative function for counting 4-motifs optimised for sparse SENs, i.e. networks with $|E| \ll |V|^2$ for $|E|$ number of edges, V number of vertices.

This function counts 4-motifs with an asymptotic execution time of $O(|E||V|^2)$. Multilevel networks are supported. Use parameters `level0` (resp. `level1`) to specify the `sesType` of the upper level (resp. the lower level) or keep the default values to count classical 4-motifs, i.e. as counted by *sma.count4Motifs()*.

See *sma.count3MotifsSparse()* for a more detailed description of the procedure.

Computing the numbers of motifs IV.A-IV.D: This function iterates over cross-level edges (and in an inner loop over one node from each of the two levels). This means that motifs without cross-level edges (IV.A to IV.D) cannot be recognised. However the number of these motifs can be reconstructed based on the numbers for the other motifs. After completing the iteration we know the numbers of all motifs except for the classes IV.A to IV.D. Based on the double counting formulas outlined in *sma.count3MotifsSparse()* we can compute the number of edges E_i in level i that we have not encountered yet. The number of remaining motifs M can be easily computed as well following *sma.total4Motifs()*. Then the following linear equations hold.

$$\begin{aligned} \text{IV.B} + \text{IV.D} &= E_0 \\ \text{IV.C} + \text{IV.D} &= E_1 \\ \text{IV.A} + \text{IV.B} + \text{IV.C} + \text{IV.D} &= M \end{aligned}$$

Unfortunately, this system of linear equation does not admit a unique solution. However, if the number of IV.A motifs is known (specified using parameter `guess_IV_A`), then the system can be easily solved by setting $M' = M -$ `guess_IV_A` and computing

$$\begin{pmatrix} \text{IV.B} \\ \text{IV.C} \\ \text{IV.D} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} E_0 \\ E_1 \\ M' \end{pmatrix} = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 0 & 1 \\ 1 & 1 & -1 \end{pmatrix} \begin{pmatrix} E_0 \\ E_1 \\ M' \end{pmatrix}.$$

If `guess_IV_A` is negative, the values for IV.A-IV.D will be set to -1. Note that in the case the sum of all returned values does not equal the integer returned by *sma.total4Motifs()* any more.

See also *sma.count4Motifs()*.

Parameters

- **G** (`Graph`) – the SEN
- **level0** (`int`) – `sesType` of the lower nodes, social in classical 4-motifs
- **level1** (`int`) – `sesType` of the upper nodes, ecological in classical 4-motifs
- **guess_IV_A** (`int`) – value for the number of IV.A or something negative if the the number of IV.A-IV.D motifs shall not be computed
- **array** (`bool`) – whether the output should be an array or a dict
- **progress_report** (`bool`) – whether a progress report should be printed

Returns array or dict with numbers of 4-motifs

sma.**count121MotifsSparse**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, *level2*, *array: bool = False*, *optimize_top_adjacent: bool = False*, *debug_unclassified: bool = False*, *progress_report: bool = False*)

Counts multi level motifs with arities 1, 2, 1, cf. *sma.Multi121MotifClassificator*. This function is faster than *sma.countMultiMotifs()* for sparse graphs, i.e. graphs with few edges. The speed up is achieved by (instead of iterating over all suitable tuples of nodes):

- iterating over all nodes in the lowest level
- for each of them, iterating over the adjacent nodes in the top level (or over all nodes from the top level if `optimize_top_adjacent = False`)
- for each of them, iterating over all nodes from the middle level that are adjacent to the node in the lowest level

With default parameters, this function counts motifs of all classes correctly. If `optimize_top_adjacent = True`, only classes `3` and `4` are counted correctly.

Parameters

- **G** (Graph) – the SEN
- **level0** – sesType of the upper level
- **level1** – sesType of the middle level
- **level2** – sesType of the lower level
- **array** (bool) – whether the output should be an array or a dict
- **optimize_top_adjacent** (bool) – whether the counting should be optimize for counting motifs of classes 3 and 4, see above.
- **debug_unclassified** (bool) – whether, in case that optimize_top_adjacent = True, the partial count for class -1 should be returned. Per default, the value is set to a negative value.
- **progress_report** (bool) – whether a progress report should be printed

sma.**count221MotifsSparse**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, *level2*, *array: bool = False*, *optimize_top_adjacent: bool = False*, *debug_unclassified: bool = False*, ***kwargs*)

Counts multi level motifs with arities 2, 2, 1, cf. *sma.Multi221MotifClassificator*. This function is faster than *sma.countMultiMotifs()* for sparse graphs, i.e. graphs with few edges. The speed up is achieved by (instead of iterating over all suitable tuples of nodes):

- iterating over all nodes in the lower level
- for each of them iterating over all pairs of combinations of two nodes from the top level, and
- two nodes from the middle level adjacent to the lower level node.

If optimize_top_adjacent = True, this function iterates only over top level nodes which are adjacent to the lower level node. In this case only motifs of classes *CLASS*.3 are counted correctly.

With default parameters, only motifs of classes *CLASS*.2 and *CLASS*.3 are counted correctly. The values for Unclassified, *CLASS*.1, *CLASS*.0 will be set to a negative value.

This function supports multiprocessing.

See also *sma.DenseMulti221Motifs*, the motif source used here for speed-up.

Parameters

- **G** (Graph) – the SEN
- **level0** – sesType of the upper level
- **level1** – sesType of the middle level
- **level2** – sesType of the lower level
- **array** (bool) – whether the output should be an array or a dict
- **optimize_top_adjacent** (bool) – whether the counting should be optimize for counting motifs of classes *CLASS*.3, see above.
- **processes** – number of processes for multiprocessing
- **chunksize** – chunksize for multiprocessing
- **debug_unclassified** (bool) – whether, the partial count for class Unclassified should be returned. Per default, the value is set to a negative value.
- **kwargs** – further parameters for *sma.countMotifs()*

sma.**count222MotifsSparse**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, *level2*, *array: bool = False*, *processes: int = 0*, *chunksize: int = 5*, *debug_unclassified: bool = False*, *progress_report: bool = False*)

Counts multi level motifs with arities 2, 2, 2, cf. *sma.Multi222MotifClassificator*. This function is faster than *sma.countMultiMotifs()* for sparse graphs, i.e. graphs with few edges. The speed up is achieved by (instead of iterating over all suitable tuples of nodes):

- iterating over all edges between two lower level nodes
- for each of them, iterating over all middle level nodes which are simultaneously neighbours of both ends of the edge and over all top level nodes which are simultaneously neighbours of both ends of the edge

This function counts only motifs of classes `3` and `4` correctly. All other counts are set to a negative value.

This function supports multiprocessing. Relatively small chunksizes are recommended.

Parameters

- **G** (`Graph`) – the SEN
- **level0** – `sesType` of the upper level
- **level1** – `sesType` of the middle level
- **level2** – `sesType` of the lower level
- **array** (`bool`) – whether the output should be an array or a dict
- **processes** (`int`) – number of processes for multiprocessing
- **chunksize** (`int`) – chunksize for multiprocessing
- **debug_unclassified** (`bool`) – whether, the partial count for class `-1` should be returned. Per default, the value is set to a negative value.
- **progress_report** (`bool`) – whether a progress report should be printed

6.1.4 Auxiliary functions

sma.**findIdealCounter**(*signature: list*, *motifs: list*, *directed: bool*, *assume_sparse: bool = True*)

Determines an ideal counter for a family of motifs. This function checks whether the given motifs can be counted with counters for sparse networks (if `assume_sparse = True`) and returns in this case the counter supplemented with additional parameters for the counter.

This function is a called by *sma.countMotifsAuto()*.

Parameters

- **signature** (`list`) – (ordered) signature
- **motifs** (`list`) – list of motif classes
- **directed** (`bool`) – whether the considered network is directed
- **assume_sparse** (`bool`) – whether the network is assumed to be sparse

Returns

tuple consisting of

- a counting function, e.g. *sma.countMultiMotifs()* or *sma.countMultiMotifsSparse()*,
- a dict with additional parameters for this counting function

6.2 Iterate through motifs

In this section basic motif iterators are described. They are sufficient for basal uses. See *sma.MotifIterator* for more advanced iterators and the functionalities for modelling abstract sets of motifs.

sma.**iterate4Motifs**(*G: networkx.classes.graph.Graph*, *level0=1*, *level1=0*)
Yields all tuples (s1,s2,e1,e2) where s1,s2 are social nodes, e1,e2 are ecological nodes

The tuples are not distinct w.r.t. their order, i.e. (1,2) = (2,1). The internal order of *sma.sesSubgraph()* is kept.

Parameters

- **G** (Graph) – the graph
- **level0** (int) – sesType of the upper nodes, usually *sma.NODE_TYPE_SOC*
- **level1** (int) – sesType of the lower nodes, usually *sma.NODE_TYPE_ECO*

sma.**iterate3Motifs**(*G: networkx.classes.graph.Graph*, *level0*, *level1*)
Yields all tuples (a, b1, b2) where a is a node from level0 and b1, b2 nodes from level1. This is a back-end function for *sma.iterate3EMotifs()* and *sma.iterate3SMotifs()*.

Parameters

- **G** (Graph) – the graph
- **level0** – sesType of the distinct node, e.g. in 3E-motifs *sma.NODE_TYPE_ECO*
- **level1** – sesType of the other nodes, e.g. in 3E-motifs *sma.NODE_TYPE_SOC*

sma.**iterate3EMotifs**(*G: networkx.classes.graph.Graph*, *level0=0*, *level1=1*)
Yields all tuples (e, s1, s2) where e is an ecological node and s1,s2 are social nodes.

The tuples are not distinct w.r.t. their order, i.e. (1,2) = (2,1). The internal order of *sma.sesSubgraph()* is kept.

See also *sma.iterate3Motifs()*.

Parameters

- **G** (Graph) – the graph
- **level0** (int) – sesType of the distinct node, here *sma.NODE_TYPE_ECO*
- **level1** (int) – sesType of the other nodes, here *sma.NODE_TYPE_SOC*

sma.**iterate3SMotifs**(*G: networkx.classes.graph.Graph*, *level0=1*, *level1=0*)
Yields all tuples (s, e1, e2) where s is a social node and e1, e2 are ecological nodes.

The tuples are not distinct w.r.t. their order, i.e. (1,2) = (2,1). The internal order of *sma.sesSubgraph()* is kept.

See also *sma.iterate3Motifs()*.

Parameters

- **G** (Graph) – the graph
- **level0** (int) – sesType of the distinct node, here *sma.NODE_TYPE_SOC*
- **level1** (int) – sesType of the other nodes, here *sma.NODE_TYPE_ECO*

sma.**sesSubgraph**(*G: networkx.classes.graph.Graph*, *sesType: int*, *source=None*) → filter
Filters only social or ecological nodes from a given graph. The returned object is a filter. If you need a networkx.Graph, you may want to call G.subgraph on the result.

```
# Let G be some SEN
socialNodes = list(sma.sesSubgraph(G, sma.NODE_TYPE_SOC))
socialGraph = G.subgraph(sma.sesSubgraph(G, sma.NODE_TYPE_SOC))
```

Return type filter

Parameters

- **G** (Graph) – the graph
- **sesType** (int) – the type (either social or ecological), see *sma.NODE_TYPE_ECO* and *sma.NODE_TYPE_SOC*.
- **source** – an alternative source for nodes, if None, G.nodes is used

Returns filter object with yields the nodes. Be aware that you will need to call list() on the result to obtain a list.

6.3 Classify motifs

This section features first basal functions which map an instance of a SEN and a motif (tuple of vertices) to its motif class (e.g. 'II.B'). In the second subsection motif classificators are introduced. Classificators are objects which wrap the former functions and provide additional structural information.

6.3.1 Front-end functions

sma.**motifTable**(*G: networkx.classes.graph.Graph*, *identifier: str*) → pandas.core.frame.DataFrame

Returns a pandas.DataFrame with one row for each instance of the motif specified by the given motif identifier string. If the identifier string specifies a motif class, e.g. 1,2[I.A] , then only motifs of the given class are returned. If the identifier string specifies a signature, e.g. 1,2, then a full list of all motifs of this signature is returned. In the latter case, the dataframe contains an additional column contain the motif classes.

The naming scheme of the columns is as follows: Each column is called levelA_nodeB where A is the sesType of the nodes in the column and B the index of the nodes amoung the nodes on the same level. This index stems from the internal order of the nodes and does not carry any specific meaning.

```
sma.motifTable(G, '1,2')

# level0_node0 level1_node0 level1_node1 class
# 0         Lake 0       Town 0       Town 1   I.B
# 1         Lake 0       Town 0       Town 2   I.B
# 2         Lake 0       Town 0       Town 3  II.A
# 3         Lake 0       Town 0       Town 4   I.B
# 4         Lake 0       Town 1       Town 2   I.C
# ..           ...          ...          ...   ...
# 95        Lake 9       Town 1       Town 3  II.B
# 96        Lake 9       Town 1       Town 4  II.C
# 97        Lake 9       Town 2       Town 3  II.B
# 98        Lake 9       Town 2       Town 4  II.C
# 99        Lake 9       Town 3       Town 4   I.B
#
# [100 rows x 4 columns]
```

This function works for directed and undirected graphs. The directed case is distinguished by calling networkx.is_directed() on the given graph.

Return type DataFrame

Parameters

- **G** (Graph) – the SEN
- **identifier** (str) – a motif identifier string

Returns DataFrame as described above.

6.3.2 Motif classificators

class sma.**MotifClassificator**(*G: networkx.classes.graph.Graph*)

Abstract super class of all motif classificators. Roughly speaking, a classificator maps a motif to its motif class, e.g. (Peter, Lake, Forrest) to 'I.C'.

This class cannot be used as classificator as it is abstract. Use one of its inheritors, e.g. *sma.FourMotifClassificator*.

Each instance of this class is callable mapping a motif to its class. For constructing such an object a SEN must be provided. The syntax is as follows, cf. *sma.ThreeMotifClassificator*.

```
import sma
# Let G be some SEN
motif = ('Peter', 'Lake', 'Forrest')
classificator = sma.ThreeMotifClassificator(G)
clss = classificator(motif) # e.g. 'I.C'
```

Each inheritor of this class must provide the following attributes. Note, that the values listed here are dummies. See the inheriting classes.

info()

Returns the motif database entry for the motifs classified by this classificator, i.e. a subclass of *sma.MotifInfo*.

class sma.**MultiMotifClassificator**(*G: networkx.classes.graph.Graph*, **arities: int*, *roles=[]*)

sma.MotifClassificator for classifying multi-motifs. This class is a wrapper class for several other motif classificators. Based on the given arities, i.e. the numbers of nodes from the different levels of the SEN, a suitable classificator is choosen automatically. This means that depending on the given parameters, the number of classes and their names is different.

See the documentation or query the motif database with *sma.motifInfo()* and *sma.supportedSignatures()* for a full list of supported motifs.

Every instances of this class is aware of the names and amount of classes that it uses for classification.

See also *sma.matchPositions()* and *sma.multiSignature()* for a description of the internal position matching mechanisms

Parameters

- **arities** – arities
- **roles** – list of indices

Raises **NotSupportedError** – whenever there is no matching classificator available

class sma.**FourMotifClassificator**(*G: networkx.classes.graph.Graph*)

sma.MotifClassificator for classifying 4-motifs. Wrapps *sma.classify4Motif()*.

class sma.**ThreeMotifClassificator**(*G: networkx.classes.graph.Graph*)

sma.MotifClassificator for classifying 3-motifs such as 3E- or 3S-motifs. Wrapps *sma.classify3Motif()*. See also *sma.ThreePMotifClassificator* for plain motifs.

class sma.**ThreePMotifClassificator**(*G: networkx.classes.graph.Graph*)

sma.MotifClassificator for classifying plain 3-motifs. These are motifs that do not respect whether the individual nodes are social or ecological.

See *sma.ThreeMotifClassificator* for the classificator that respects the types of the nodes.

class sma.**TwoMotifClassificator**(*G: networkx.classes.graph.Graph*)

sma.MotifClassificator for classifying 2-motifs. See *sma.classify2Motif()*.

class sma.**Multi111MotifClassificator**(*G: networkx.classes.graph.Graph*)

class sma.**Multi121MotifClassificator**(*G: networkx.classes.graph.Graph*)

class sma.**Multi221MotifClassificator**(*G: networkx.classes.graph.Graph*)

class sma.**Multi222MotifClassificator**(*G: networkx.classes.graph.Graph*)

class sma.**OneMotifClassificator**(*G: networkx.classes.graph.Graph*)
sma.MotifClassificator for classifying 1-motifs. Such motifs consist only of a single vertex.

class sma.**DiMotif11Classificator**(*G: networkx.classes.graph.Graph*)
Classificator for directed motifs with signature (1, 1).

class sma.**DiMotif2Classificator**(*G: networkx.classes.graph.Graph*)
Classificator for directed motifs with signature (2).

class sma.**DiMotif12Classificator**(*G: networkx.classes.graph.Graph*)
Classificator for directed motifs with signature (1, 2).

class sma.**DiMotif22Classificator**(*G: networkx.classes.graph.Graph*)
Classificator for directed motifs with signature (1, 2).

6.3.3 Basal functions

sma.**classify4Motif**(*G: networkx.classes.graph.Graph*, *motif*) → str
Classifies a 4-motif given as a 4-tuple of its nodes.

See *sma.binaryCodeToClass4Motifs()*.

Return type str

Parameters

- **G** (Graph) – the graph
- **motif** – the motif given as 4-tuple of its nodes

Returns class of the motif as string, e.g. 'I.C'

sma.**classify3Motif**(*G: networkx.classes.graph.Graph*, *motif*) → str
Classifies a 3-motif given as a 3-tuple of its nodes.

See *sma.binaryCodeToClass3Motifs()*.

Return type str

Parameters

- **G** (Graph) – the graph
- **motif** – the motif given as 3-tuple of its nodes

Returns class of the motif as string, e.g. 'I.C'

sma.**classify3pMotif**(*G: networkx.classes.graph.Graph*, *motif*) → int
Classifies a plain 3-motif given as a 3-tuple of its nodes.

Since the three vertices in the motif are not distinguishable, there exist four classes of motifs:

- type 0: no edges
- type 1: one edge
- type 2: two edges
- type 3: three edges

Return type int

Parameters

- **G** (Graph) – the graph
- **motif** – the motif given as 3-tuple of its nodes

Returns class of the motif as integer

sma.**classify2Motif**(*G: networkx.classes.graph.Graph*, *motif*) → bool
Classifies 2-motifs. These motifs consist of two vertices. Two classes of motifs occur. Either the two vertices are linked (type 1) or the two vertices not linked (type 0).

Return type bool

Parameters

- **G** (Graph) – the SEN
- **motif** – a pair of two vertices, the motif

sma.**classify111Motif**(*G: networkx.classes.graph.Graph*, *motif*) → int

sma.**classify121Motif**(*G: networkx.classes.graph.Graph*, *motif*) → int

sma.**classify221Motif**(*G: networkx.classes.graph.Graph*, *motif*) → str

sma.**classify222Motif**(*G: networkx.classes.graph.Graph*, *motif*) → int

6.4 Analyse triangles and higher order nodal properties

sma.**triangleCoefficient**(*G: networkx.classes.graph.Graph*, *node*, *othertype=None*) → float
Calculates the triangle coefficient of node, which is defined as follows:

Take all 3-motifs with the node as distinct node and compute the number of I.C and II.C motifs (open and closed triangles with the given node at the point). Divide the number of I.C motifs by the sum of number of I.C motifs and II.C motifs.

See *sma.cooccurrenceTable()* for a more general function.

Return type float

Parameters

- **G** (Graph) – the graph
- **node** – the node whose triangle coefficient shall be computed
- **othertype** – the sesType of the non-distinct nodes or None if all nodes with sesType different from the sesType of the distinct node shall be considered. A value must be specified when used with multi-level networks in order to avoid unexpected behaviour.

Returns II.C / (I.C + II.C), i.e. the ratio of closed triangles vs. all triangles, if no triangles are found, this function returns zero.

sma.**triangleCoefficients**(*G: networkx.classes.graph.Graph*, *array: bool = False*)
Computes the triangle coefficients for all nodes in a graph.

Parameters

- **G** (Graph) – the graph
- **array** (bool) – set array to True if you want the result as a list instead of a dict

See *sma.triangleCoefficient()* for details.

sma.**connectedOpposingSystem**(*G: networkx.classes.graph.Graph*, *node*, *othertype=None*) → set
Returns the set of nodes which are connected to the given node but have the opposite social-ecological type.

Return type set

Parameters

- **G** (Graph) – the graph

- **node** – the node
- **othertype** – the `sesType` of the opposite nodes or `None` if all nodes with `sesType` different from the `sesType` of the specified node shall be considered. A value must be specified when used with multi-level networks in order to avoid unexpected behaviour.

Returns set of nodes which share edges with the given node and are of the other social-ecological type.

`sma.`**`overlappingCoefficients`**(*G: networkx.classes.graph.Graph, subsystem: int = 0*) → numpy.ndarray

Returns a matrix of overlapping coefficients for a given subsystem of nodes. For two nodes x, y of the same type the overlapping coefficient is defined as

$$O(x,y) = \begin{cases} \frac{|S(x) \cap S(y)|}{|S(x) \cup S(y)|}, & \text{if } S(x) \cup S(y) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

where $O(x,y)$ is the overlapping coefficient of x and y and $S(i)$ is the connected opposing system of i, cf. *sma.connectedOpposingSystem()*, i.e. the set of nodes which are connected to i but are of the opposite social ecological type.

Return type `ndarray`

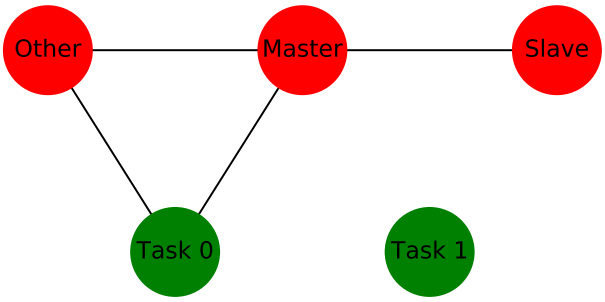
Parameters

- **G** (`Graph`) – the graph
- **subsystem** (`int`) – type of the nodes to compute the coefficients for, default is *sma.NODE_TYPE_ECO*.

Returns symmetric matrix containing the coefficients, default is *sma.NODE_TYPE_ECO* for computing the overlapping coefficients of the ecological subsystem

6.5 Co-occurrences

Co-occurrence is a way of describing the relations among motifs. Two motifs co-occur whenever they share a common vertex. The number of these co-occurring vertices can then be determined and is an indicator for meso-level properties of the SEN.



In the above example two 3E-motifs co-occur. The II.C motif on the left and the II.A motif on the right share a common vertex labelled as "master" in this example. Hence the contribution of the above configuration to the II.A-II.C entry in the result of *sma.cooccurrenceTableFull()* would be 1.

`sma.`**`cooccurrenceTable`**(*G: networkx.classes.graph.Graph, *motifs, to_array: bool = False, to_dict: bool = False*)

Returns the co-occurrence table for a given set of motif generators. This table contains one row for each node in the given SEN G. The i-th entry for the j-th row represents the number of occurrences of the j-th node in motifs yielded by the i-th motif generator.

For example, one could want to count the number of occurrences of nodes in closed and open triangles:

```
import sma
# let G be some SEN
result = sma.cooccurrenceTable(G,
                               sma.ThreeEMotifs(G) & sma.is3Class('I.C'),
                               sma.ThreeEMotifs(G) & sma.is3Class('II.C'))
```

Then the first entry for each node contains the number of open triangles (class I.C motif) this node is involved in and the second entry the number of closed triangles (class II.C motif). Note, that in this example the involvement of a node at any position (not only at the distinct position) counts, cf. *sma.triangleCoefficient()*.

See also *sma.cooccurrenceTableFull()*.

Parameters

- **G** (Graph) – the SEN
- **motifs** – a list of motif generators, cf. *sma.MotifIterator*
- **to_array** (bool) – Boolean indicating whether the output should be a numpy array, entries for nodes a given in the order returned by the graph
- **to_dict** (bool) – Boolean indicating whether the output should be a two-level dict. Values can be accessed by `result[node][motif]` where motif is the string representation of the motif

Returns co-occurrence table as described above

sma.**cooccurrenceTableFull**(*G: networkx.classes.graph.Graph, iterator: sma.iterate.MotifIterator, classificator: sma.classify.MotifClassificator, to_array: bool = False*)

This function returns a dict/array similar to the result of *sma.cooccurrenceTable()*. This function should be used when the values for several classes of motifs taken from the same source iterator are of interest. For example, if all 3-motifs shall be fully classified, *sma.cooccurrenceTableFull()* should be preferred over *sma.cooccurrenceTable()* since it does not incur any redundant costs.

If the parameter `to_array` is set to `False` (default), the result is a dictionary which maps vertices to sub-dictionaries mapping motif classes (taken from the classificator) to the integer representing the number of occasions this vertex occurs in a motif of this class. If `to_array` is flipped, the result is converted to a matrix with one row for each vertex and one column for each motif class.

The sum of all entries equals the total number of motifs in the iterator multiplied by the arity of the classificator.

Parameters

- **G** (Graph) – a SEN
- **iterator** (MotifIterator) – a *sma.MotifIterator* as source of motifs
- **classificator** (MotifClassificator) – a *sma.MotifClassificator* for classifying the motifs
- **to_array** (bool) – whether the result should be an array or a dict.

Returns co-occurrences table featuring values for all classes of motifs

sma.**cooccurrenceEdgeTableFull**(*G: networkx.classes.graph.Graph, iterator: sma.iterate.MotifIterator, classificator: sma.classify.MotifClassificator, dyad, levels: tuple, processes: int = 0, chunksize: int = 10000*)

Computes a co-occurrence table on edge level. Given an edge (v_1, v_2) of a motif, v_1 in level i, v_2 in level j, this table consists of $|V_i| \times |V_j| \times M$ entries where V_i, V_j denotes the set of notes from level i, resp. j in the SEN and M denotes the number of motif classes as classified by the given classificator. The (a, b, c) occurs in a motif of class c.

The edge is specified using the `dyad` parameter. This parameter must contain a tuple of two integers specifying the index of the nodes spanning the edge in the motifs provided by the iterator. For example, a typical 3E-motif looks like (e, s_1, s_2) where s_1 and s_2 are social and e is ecological. In this case, `dyad = (1,2)` would imply that the co-occurrence matrix for the edge (s_1, s_2) would be computed. For technical reasons, a parameter `levels` must be provided. `levels` must be a tuple of length two specifying the `sesType` of the nodes in edge specified by `dyad`. In the example, `levels` would be (*sma.NODE_TYPE_SOC*,:py:const:*sma.NODE_TYPE_SOC*).

Multiprocessing is supported. Use parameters `processes` and `types`.

Example Compute the co-occurrence table for 3S-motifs

```
sma.cooccurrenceEdgeTableFull(G,
                              sma.ThreeSMotifs(G),
                              sma.ThreeMotifClassificator(G),
                              (1,2), # dyad, social nodes
                              (0,0)) # levels, both social
```

Parameters

- **G** (`Graph`) – the SEN
- **iterator** (`MotifIterator`) – source of motifs
- **classificator** (`MotifClassificator`) – classificator for the motifs
- **dyad** – specification of the esge for which the co-occurrence table is computed (tuple of length 2)
- **levels** (`tuple`) – `sesTypes` of the nodes in `dyad` (tuple of length 2)
- **processes** (`int`) – number of processes for multiprocessing
- **chunksize** (`int`) – chunksize for multiprocessing

Returns three values: the co-occurrence table, list of nodes as index for the rows, list of nodes as index for the columns

6.6 Motif Graphs

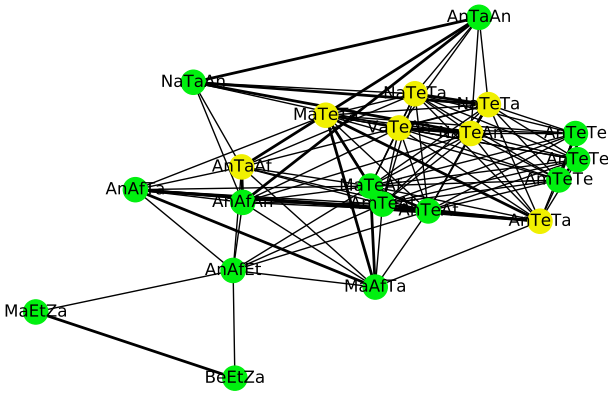
6.6.1 Motif Multigraph

The motif multigraph contains one vertex for each of the original graph's motifs. Contrarily, the motif class graph contains one vertex for each class of vertices. Since any multigraph can be translated naturally to a weighted simple graph, both graphs can be represented as such weighted simple graphs.

The following example demonstrates how *sma.motifMultigraph()* is supposed to be used:

```
import sma
# Let G be some SEN (here Madagascar dataset)
motifs = [sma.ThreeEMotifs(G) & sma.is3Class('I.C'),
          sma.ThreeEMotifs(G) & sma.is3Class('II.C')]
M = sma.motifWeightedGraph(G, *motifs)
# some manipulations for plotting
```

The result is shown below. The multigraph contains one node for each 3E-motif of class I.C (yellow) and II.C (green). Two motifs are connected with an edge if they share a vertex. This edge is bold if not only one but two vertices are shared by the two motifs.



sma.**motifMultigraph**(*G: networkx.classes.graph.Graph, *motifs, attr='motif'*) → networkx.classes.multigraph.MultiGraph

Returns a multigraph of motifs. In this multigraph every node represents motif. For example, when the second parameter is `sma.FourMotif(G)`, then every node in the returned graph represents one 4-motif. Each edge connecting two nodes m_1, m_2 represents one node in the original graph that is shared by two motifs that m_1 and m_2 represent.

Note that the motif multigraph does not contain any loops since each motifs shares trivially all its vertices with itself.

A list of motif iterators must be given, cf. *sma.MotifIterator*. In this way different classes of motifs can be incorporated in one multigraph. For example, one could be interested in the adjacence of open triangles with distinct social node (social motif I.C) and closed triangles with distinct ecological node (ecological motif II.C). Then these two motif iterators can be provided as input, cf. *sma.ThreeEMotifs*, *sma.ThreeSMotifs* and *sma.is3Class*.

The multigraph can be converted to a weighted simple graph with edge weights corresponding to edge multiplicities in the multigraph using *sma.multiToWeightedGraph()*. See also *sma.motifWeightedGraph()*.

The class of the motifs, i.e. a string representation of the given motif iterators, is stored as nodal attribute.

Return type `MultiGraph`

Parameters

- **G** (`Graph`) – a SEN
- **motifs** – one or several instances of *sma.MotifIterator*
- **attr** (`str`) – attribute key for the nodal attribute representing the class of the motifs, default `motif`

Returns motif multigraph

sma.**motifWeightedGraph**(*G: networkx.classes.graph.Graph, *motifs, attr='weight'*) → networkx.classes.graph.Graph

Returns a weighted graph corresponding to the motif multigraph generated by *sma.motifMultigraph()*. That two motifs share vertices is not encoded by multiple edges but by edge weights.

See *sma.multiToWeightedGraph()*, the function that takes care of the translation.

Return type `Graph`

Parameters

- **G** (`Graph`) – the SEN
- **motifs** – one or several instances of *sma.MotifIterator*
- **attr** (`str`) – the attribute key for the weight attribute, default is `weight`.

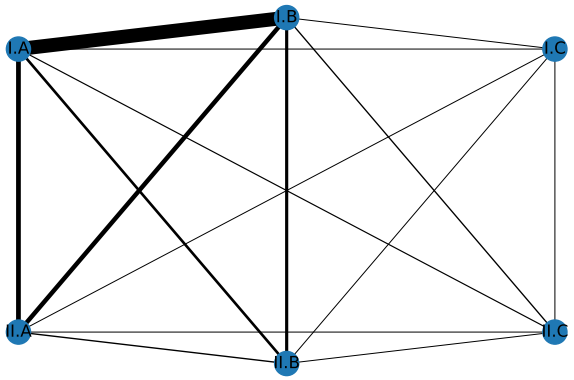
6.6.2 Motif Class Graph

The motif class graph contains one vertex for each class of motifs. The edges are weighted by the sum of the numbers of shared vertices of distinct motifs of the respective classes.

The following example demonstrates the usage of *sma.cooccurrenceTableFull()* which lays the groundwork for *sma.motifClassMatrix()* and *sma.motifClassGraph()*.

```
import sma
# Let G be some SEN, here Madagascar dataset

mat1 = sma.cooccurrenceTableFull(G,
                                 sma.ThreeEMotifs(G),
                                 sma.ThreeMotifClassificator(G),
                                 to_array=True)
# one row for each vertex, one column for each motif class
#   array([[13, 51,  6,  6, 16,  6],
#          [44, 37,  3, 10,  2,  2],
#          ...,
#          [10, 11,  0,  5,  1,  1]])

mat2 = sma.motifClassMatrix(G,
                            sma.ThreeEMotifs(G),
                            sma.ThreeMotifClassificator(G))
# upper triangular matrix, one column/row for each motif class
#   array([[ 6556, 11563,   465,  3938,  2202,   986],
#          [    0,  5744,   641,  3726,  2612,  1131],
#          ...
#          [    0,     0,     0,     0,     0,    51]])

G2 = sma.motifClassGraph(G, sma.ThreeEMotifs(G), sma.ThreeMotifClassificator(G))
```

The resulting graph `G2` is shown below. Thicker edges carry heavier weight. Loops are hidden.



sma.**motifClassMatrix**(*G: networkx.classes.graph.Graph*, *iterator: sma.iterate.MotifIterator*, *classificator: sma.classify.MotifClassificator*, *as_symmetric: bool = False*) → numpy.ndarray

In the returned quadratic matrix every column and every row represents a motif class, i.e. a class of motif as recorgnized by the given *sma.MotifClassificator*. For example, in case of 3-motifs the first column and the first row represent class 'I.A' motifs whereas the last row / column represents 'II.C' motifs. In total, the matrix would be of dimension 6×6 since there are six 3-motifs.

The (i, j)-th entry represents the number of vertices in the given SEN shared by motifs of class i and j. If a vertex is contained in several class i and class j motifs, it is counted multiply.

Let (C_{ij}) denote the returned matrix. Then for the sum of all upper triangular entries the following corre-

spondence holds:

$$\sum_{i \geq j} C_{ij} = \frac{1}{2} \sum_{M_1} \sum_{M_2 \neq M_1} |M_1 \cap M_2|$$

where M_1 and M_2 are motifs taken from the iterator.

This function computes the desired matrix based on the result of *sma.cooccurrenceTableFull()*. For every vertex its occurrence in each of the possible classes of motifs is counted. Let red, blue and orange be three distinct motif classes. If a vertex occurs n times in red motifs, m_1 times in blue motifs and m_2 times in orange motifs, then it establishes $\binom{n}{2}$ connections between red motifs. Hence it contributes this number to the red diagonal entry. This is the number of edges in the complete graph K_n. Moreover, the vertex establishes $m_1 \cdot m_2$ connections between blue and orange motifs. This is the number of edges in the complete bipartite graph K_{m_1,m_2} and the vertex' contribution to the blue-orange off-diagonal entry.

Return type ndarray

Parameters

- **G** (Graph) – the SEN
- **iterator** (MotifIterator) – a *sma.MotifIterator* as a source of motifs
- **classificiator** – a *sma.MotifClassificator* for classifying the motifs. Note that it must match with the given iterator.
- **as_symmetric** (bool) – per default the returned matrix is upper triangular. If this switch is set to True the returned matrix will be symmetrical, with the upper entries copied to the matrix' lower half.

sma.**motifClassGraph**(*G: networkx.classes.graph.Graph*, *iterator: sma.iterate.MotifIterator*, *classificator: sma.classify.MotifClassificator*) → networkx.classes.graph.Graph

Returns an networkx.Graph with one node for every motif class as described by the given *sma.MotifClassificator*. The edges are weighted by the entries of *sma.motifClassMatrix()*. The graph contains loops.

Return type Graph

Parameters

- **G** (Graph) – the SEN
- **iterator** (MotifIterator) – a *sma.MotifIterator* as a source of motifs
- **classificiator** – a *sma.MotifClassificator* for classifying the motifs. Note that it must match with the given iterator.

6.7 Gaps

Flipping a dyad results in a changed number of motifs. When fixing a dyad on a specific level, motifs might be grouped into pairs of open and closed motifs based on the presence of the fixed dyad. Flipping the dyad turns a open motif into a closed motif and vice versa. One might ask which dyads contribute the most to the amount of a specific motif when being flipped. Functions in this section analyse the contribution of these *gaps* to the number of motifs.

sma.**identifyGaps**(*G: networkx.classes.graph.Graph*, *motif_identifier: str*, *level: int = - 1*) → list

Computes a list of gaps (specific dyads) together with their contribution to the number of motifs of a specific class.

The contribution of a gap is the number of motifs of a fixed class which would be created by replacing the gap with an edge. This function computes this contribution for every gap on the specified level.

The behaviour above applies when the specified motif contains the edge on the given level (i.e. the motif is a *closed motif*). If the edge is not present in the specified motif (the motif is *open*), the contribution is defined the number of motifs of the specified class are created by removing an edge from the network.

This function does not work for directed graphs.

Return type list

Parameters

- **G** (Graph) – the SEN
- **motif_identifier** (str) – motif identifier string of the motif
- **level** (int) – dyads of which level (sesType) are to be considered

Returns

list of tuples ordered by the second entry in descending order consisting of

- a tuple specifying an edge
- the contribution of this edge to the number of motifs of the given type when being flipped

sma.**isClosedMotif**(*motif_identifier: str*, *level: int* = - *1*) → bool

Queries the motif database and determines whether a motif is closed or open with respect to a given level.

See *sma.identifyGaps()*.

This function does not work for directed graphs.

Return type bool

Parameters

- **motif_identifier** (str) – motif identifier string
- **level** (int) – level (sesType) for the gap

Returns true/false whether the motif is closed.

Raises **NotImplementedError** – if the motif database does not know..

6.8 Distribution of motifs in Erdős-Rényi random graphs

The Erdős-Rényi model provides a straightforward idea for modelling random graphs. Its basic assumption is that the edges in a random graph are chosen uniformly and independently, i.e. an edge is present in a graph with fixed probability p.

The Erdős-Rényi model fails to model more complicated processes that govern the creation of edges in most real world graphs. For example, it cannot be used to model triadic closure.

Nevertheless, the model provides a starting point for a probabilistic analysis of a SEN. Questions like "Are the numbers of 4-motifs in a given real world network higher/lower than expected?" can be answered assuming that the set of all graphs the real world network is compared with is only restricted by the density of the graph. In fact, the question should be reformulated as "Are the numbers of 4-motifs in a given real world network higher/lower than expected in a graph with same expected density?".

For our purposes we consider a refined version of the Erdős-Rényi model (*sma.MODEL_ERDOS_RENYI*). Instead of fixing one probability we fix probabilities p_{ij} for all pairs $i \leq j \in \{0, \ldots, n-1\}$ where $0, \ldots, n-1$ refer to the levels in the SEN (e.g. 0 for ecological nodes, 1 for social nodes in a two level SEN). Then the assumption is that edges linking levels i and j are chosen independently with probability p_{ij}. Hence the values p_{ij} are the densities of the respective subsystem.

Under these assumptions we are able to compute the expected numbers of motifs of a given class analytically, i.e. without computing thousands of random graphs. The functions described in this sections make use of the following observation:

Mathematical Background Fix a set of ecological nodes V_0 and a set of social nodes V_1 and let $\mathscr{G}$ denote the set of all graphs with vertex set $V_0 \cup V_1$. Turn $\mathscr{G}$ into a probability space by taking a (refined) Erdős-Rényi model on two levels with probabilities p_{00}, p_{01}, p_{11} where 0 refers to the ecological level and 1 to the social

level. We want to compute the expected number of 3E-motifs of class I.C (open triangle). Let $X : \mathscr{G} \to \mathbb{N}$ be a random variable mapping a graph to the number of I.C motifs that it contains. We observe that $X(G) = \sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} Y(v_0, v_1, v_2; G)$ where $Y(v_0, v_1, v_2; G)$ is one whenever the given triple forms a I.C motif in G and zero otherwise. Then by linearity of the expectation the following holds,

$$\begin{aligned}
\mathbb{E}\left[X(G)\right] &= \mathbb{E}\left[\sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} Y(v_0, v_1, v_2; G)\right] \\
&= \sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} \mathbb{E}\left[Y(v_0, v_1, v_2; G)\right] \\
&= \sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} \mathbb{P}(\text{edge } v_0, v_1)\mathbb{P}(\text{edge } v_0, v_2)\mathbb{P}(\text{no edge } v_1, v_2) \\
&= \sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} p_{01}^2(1 - p_{11}) \\
&= |V_0|\binom{|V_1|}{2} p_{01}^2(1 - p_{11}).
\end{aligned}$$

Note that $|V_0|\binom{|V_1|}{2}$ represents the total number of 3E-motifs in the graph.

The functions in this section can either compute densities, i.e. number of motifs divided by the total number of motifs, e.g. $p_{01}^2(1 - p_{11})$ in this example, or the expected number of motifs as above.

It is also possible to compute the variance $\mathbb{V}[X]$ in this way. In particular, after having computed the expectation as above it remains to compute the second moment $\mathbb{E}[X^2]$ since the variance satisfies $\mathbb{V}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$. Again, by linearity of the expectation,

$$\begin{aligned}
\mathbb{E}\left[X(G)^2\right] &= \mathbb{E}\left[\left(\sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} Y(v_0, v_1, v_2; G)\right)^2\right] \\
&= \mathbb{E}\left[\sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} \sum_{v_0' \in V_0} \sum_{v_1' \neq v_2' \in V_1} Y(v_0, v_1, v_2; G) Y(v_0', v_1', v_2'; G)\right] \\
&= \sum_{v_0 \in V_0} \sum_{v_1 \neq v_2 \in V_1} \sum_{v_0' \in V_0} \sum_{v_1' \neq v_2' \in V_1} \mathbb{E}\left[Y(v_0, v_1, v_2; G) Y(v_0', v_1', v_2'; G)\right].
\end{aligned}$$

At this point, are more subtle argument is requirement because the expectation is not multiplicative. The fourfold sum is over $\left(|V_0|\binom{|V_1|}{2}\right)^2$ choices for $v_0, v_1, v_2, v_0', v_1', v_2'$. We split the sum according to $\{v_0, v_1, v_2\} \cap \{v_0', v_1', v_2'\}$ and compute the individual probabilities. In this way, we obtain the desired second moment, see *Appendix: Probabilities in Erdős-Rényi random graphs* for details.

Comparison with SEN generators Functions like `sma.randomSENs()` and `sma.randomMultiSENs()` can be used to compute random graphs with same number of edges (per subsystems) as a given graph. For the purpose of this package this model is called *Fixed Densities Model* (`sma.MODEL_FIXED_DENSITIES`).

Note that this approach and the (refined) Erdős-Rényi model are slightly different. `sma.randomSENs()` computes only graphs with exactly the same amount of edges (per subsytems) assuming a uniform distribution on the set of all these graphs. In Erdős-Rényi the graphs have only in expectation the same amount of edges. This relaxation leads the combinatorial argument above. Using Erdős-Rényi is therefore roughly equivalent to the ERGM model `ergm(G ~ nodemix(each combination of levels in the network))` in R, cf. *Integration in R*.

The Actor's Choice Model Besides the general Erdős-Rényi model which is outlined above, a more refined baseline model, the *Actor's Choice Model* is supported. In this model, it is assumed that all but one subsystem (the actor subsystem) are fixed. The edges in the actor subsystem are chosen independently and uniformly with a certain fixed probability. In other words, only the edges between actors are assumed to be random. Edges linking actors with other levels of the network and dyads in these other levels are assumed to be fixed. This allows for an analysis which better models decision making processes in the actor level.

Characteristics of Actor's Choice distribution can be computed using *sma.distributionMotifsActorsChoice()* and the front-end function *sma.distributionMotifsAuto()*. The mathematical justification for these methods is as follows: Let for an edge e in the actor level $C(e)$ denote the contribution of an edge to a pair of open/closed motifs, cf. *sma.edgeContributionList()*. That is, the number of times this edge occurs in a (fixed) motif M or in the motif $\bar{M}$ which differs from the fixed motif only in the edge on the actor level. Suppose that M contains an edge on the actor level (i.e., it is *closed*) and that $\bar{M}$ does not contain any edge between actors (it is *open*). Examples for such pairs of motifs are the open and closed ecological triangle (motifs I.C and II.C). Furthermore, let X denote the random variable giving the number of closed motifs of class M. Let as above Y denote the random variable on the set of edges in the actor level giving 1 if the edge occurs and 0 otherwise. Then,

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_e Y(e)C(e)\right] = \sum_e \mathbb{E}[Y(e)C(e)] = \sum_e pC(e) = p\sum_e C(e),$$

where p denotes the density of the actor level. Similarly,

$$\begin{aligned}
\mathbb{V}[X] &= \mathbb{E}\left(\left[\sum_e Y(e)C(e)\right]\right)^2 - \left(\mathbb{E}\left[\sum_e Y(e)C(e)\right]\right)^2 \\
&= \sum_{e,e'} \mathbb{E}[Y(e)C(e)Y(e')C(e')] - p^2 \sum_{e,e'} C(e)C(e') \\
&= p^2 \sum_{e\neq e'} C(e)C(e') + p\sum_e C(e)^2 - p^2 \sum_{e,e'} C(e)C(e') \\
&= p\sum_e C(e)^2 - p^2 \sum_e C(e)^2 \\
&= p(1-p)\sum_e C(e)^2.
\end{aligned}$$

References See [Ryd18].

Example The functions in this section can replace expensive simulations of random graphs:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import sma
import time, multiprocessing, itertools, numpy

# load graph
G = sma.loadSEN('data/dummy_adj.csv', 'data/dummy_attr.csv', delimiter=',')

n = 5000 # samples
processes = 6 # cores

def mapper(G):
     return sma.count3EMotifs(G, array = True)

start = time.perf_counter()
with multiprocessing.Pool(processes = processes) as p:
    res = p.imap_unordered(mapper,
                           itertools.islice(sma.randomSimilarSENs(G), n),
                           chunksize = n // processes // 4)
    p.close()
    p.join()

end = time.perf_counter()
print('Calculated %d samples in %f sec' % (n, end-start))

results = numpy.array(list(res))

means = numpy.mean(results, axis = 0)
```

(continues on next page)

(continued from previous page)

```
print('Simulating %d random networks gave the following means/expectations:' % n)
print(means)
print("Now lets analytically determine the expections:")
means_er = sma.expected3EMotifs(G, array = True)
print(means_er)
```

6.8.1 Front-end functions

sma.**expectedMultiMotifs**(*G: networkx.classes.graph.Graph*, **arities*, *roles=[]*, ***kwargs*)

Front-end function for several functions for computing motif expectations. It supports the same parameters for defining the motifs (arities, roles) as other multi-level (counting) functions. Check the documentations of the respective functions. The supported functions are:

- *sma.expected3Motifs()*,
- *sma.expected4Motifs()*,
- *sma.expected121Motifs()*,
- *sma.expected221Motifs()*,
- *sma.expected222Motifs()*.

See the documentation for an exposition of the mathematical background.

Parameters

- **G** (Graph) – the SEN
- **arities** – arities of the motifs, cf. *sma.MultiMotifClassificator*.
- **roles** – roles of the levels, cf. *sma.MultiMotifClassificator*.
- **kwargs** – additional parameters for the functions, see above.

sma.**varMultiMotifs**(*G: networkx.classes.graph.Graph*, **arities*, *roles=[]*, ***kwargs*)

Front-end function for several functions for computing motif variances. It supports the same parameters for defining the motifs (arities, roles) as other multi-level (counting) functions. Check the documentations of the respective functions. The supported functions are:

- *sma.var3Motifs()*,
- *sma.var4Motifs()*.

See the documentation for an exposition of the mathematical background.

Parameters

- **G** (Graph) – the SEN
- **arities** – arities of the motifs, cf. *sma.MultiMotifClassificator*.
- **roles** – roles of the levels, cf. *sma.MultiMotifClassificator*.
- **kwargs** – additional parameters for the functions, see above.

sma.**distributionMotifsAuto**(*G: networkx.classes.graph.Graph*, **motifs: str*, *model: str = 'erdos_renyi'*, *level: int = - 1*)

Front-end function for functions in sma which provide insight into the distribution of motif counts in various baseline models.

This function returns a tuple consisting of two dicts:

1. a dict of partial results as requested using the motif identifiers. This dict maps motif identifiers to tuples containing firstly the expectation and secondly the variance of this motif.

2. a dict of the results of all motifs with same arities/roles as the motif specified using the motif identifiers. This dict maps the heads of the motif identifiers to dicts mapping motif classes to tuples of expectations and variances as above.

Values which are not available in the selected model are given as `None`.

The following models (specify using parameter `model`) are supported:

- `erdos_renyi` (default) general Erdős-Rényi random graphs with fixed edge probabilities for each subsystem equalling the densities of the given SEN, see documentation, cf. *sma.MODEL_ERDOS_RENYI*.
- `actors_choice` edges in one specific level are chosen at random. The remainder of the network is fixed. The edge probability in the variable level is extracted from the given SEN. Use parameter `level` to specify the `sesType` of the variable level. If -1 (default), then the first entry equalling 2 in the list of arities is selected, cd. *sma.MODEL_ACTORS_CHOICE*.

Parameters

- **G** (`Graph`) – the SEN
- **motifs** (`str`) – motif identifier strings
- **model** (`str`) – model to be used
- **level** (`int`) – additional parameter for model `actors_choice`

Returns tuple of two dicts as above

Raises

- **ValueError** – when the motif identifiers cannot be parsed
- **NotImplementedError** – if the specified model is not supported

6.8.2 Erdős-Rényi expectations

sma.**expected3Motifs**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, *array=False*, *densities: bool = False*)

Computes the expected number of 3-motifs in a SEN. See the documentation for an exposition of the mathematical background.

Use functions *sma.expected3EMotifs()* and *sma.expected3SMotifs()* for regular two level graphs.

If `densities = True`, the returned values sum up to one. Otherwise their sum is total number of 3-motifs, see *sma.total3EMotifs()* etc.

Parameters

- **G** (`Graph`) – the SEN
- **level0** – `sesType` of the distinct node, e.g. in 3E-motifs *sma.NODE_TYPE_ECO*
- **level1** – `sesType` of the other nodes, e.g. in 3E-motifs *sma.NODE_TYPE_SOC*
- **array** (`bool`) – whether the result shall be an array or a dict
- **densities** (`bool`) – whether densities or absolute numbers of motifs shall be returned

sma.**expected3EMotifs**(*G: networkx.classes.graph.Graph*, ***kwargs*)

Computes the expected number of 3E-motifs in a SEN. See the documentation for an exposition of the mathematical background. See also *sma.expected3Motifs()*.

Parameters

- **G** (`Graph`) – the SEN

- **kwargs** – additional parameters for *sma.expected3Motifs()*.

sma.**expected3SMotifs**(*G: networkx.classes.graph.Graph*, ***kwargs*)
Computes the expected number of 3S-motifs in a SEN. See the documentation for an exposition of the mathematical background. See also *sma.expected3Motifs()*.

Parameters

- **G** (Graph) – the SEN
- **kwargs** – additional parameters for *sma.expected3Motifs()*.

sma.**expected4Motifs**(*G: networkx.classes.graph.Graph*, *level0: int = 1*, *level1: int = 0*, *array: bool = False*, *densities: bool = False*)
Computes the expected number of 4-motifs in a SEN. See the documentation for an exposition of the mathematical background.

If `densities = True`, the returned values sum up to one. Otherwise their sum is total number of 4-motifs, see *sma.total4Motifs()* etc.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the upper nodes, standard *sma.NODE_TYPE_SOC*
- **level1** (int) – sesType of the lower nodes, e.g. in 3E-motifs *sma.NODE_TYPE_ECO*
- **array** (bool) – whether the result shall be an array or a dict
- **densities** (bool) – whether densities or absolute numbers of motifs shall be returned

sma.**expected121Motifs**(*G: networkx.classes.graph.Graph*, *level0: int*, *level1: int*, *level2: int*, *array: bool = False*, *density: bool = False*)
Computes the expected number of multi level motifs with arities 1, 2, 1. See the documentation for an exposition of the mathematical background.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the upper node
- **level1** (int) – sesType of the middle nodes
- **level2** (int) – sesType of the lower node
- **array** (bool) – whether the result shall be an array or a dict
- **densities** – whether densities or absolute numbers of motifs shall be returned

sma.**expected221Motifs**(*G: networkx.classes.graph.Graph*, *level0: int*, *level1: int*, *level2: int*, *array: bool = False*, *density: bool = False*)
Computes the expected number of multi level motifs with arities 2, 2, 1. See the documentation for an exposition of the mathematical background.

Uses *sma.expected4Motifs()* for computing the expectations of the upper part.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the upper node
- **level1** (int) – sesType of the middle nodes
- **level2** (int) – sesType of the lower node
- **array** (bool) – whether the result shall be an array or a dict
- **densities** – whether densities or absolute numbers of motifs shall be returned

sma.**expected222Motifs**(*G: networkx.classes.graph.Graph*, *level0: int*, *level1: int*, *level2: int*, *array: bool = False*, *density: bool = False*)

Computes the expected number of multi level motifs with arities 2, 2, 2. See the documentation for an exposition of the mathematical background.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the upper node
- **level1** (int) – sesType of the middle nodes
- **level2** (int) – sesType of the lower node
- **array** (bool) – whether the result shall be an array or a dict
- **densities** – whether densities or absolute numbers of motifs shall be returned

sma.**expected111Motifs**(*G: networkx.classes.graph.Graph*, *level0: int*, *level1: int*, *level2: int*, *array: bool = False*, *density: bool = False*)

Computes the expected number of multi level motifs with arities 2, 2, 2. See the documentation for an exposition of the mathematical background.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the upper node
- **level1** (int) – sesType of the middle nodes
- **level2** (int) – sesType of the lower node
- **array** (bool) – whether the result shall be an array or a dict
- **densities** – whether densities or absolute numbers of motifs shall be returned

6.8.3 Erdős-Rényi variances

sma.**var3Motifs**(*G: networkx.classes.graph.Graph*, *level0: int*, *level1: int*, *array: bool = False*, *second_moment: bool = False*)

Computes variances of the number for 3-motifs in a SEN. See documentation for the mathematical background.

This function uses the following formula where X denotes the random variable number of 3-motifs

$$\mathbb{V}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$$

and computes the expectation using *sma.expected3Motifs()*. Set `second_moment = True` to compute $\mathbb{E}[X^2]$ instead of the variance. In this case *sma.expected3Motifs()* is not called.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the distinct node, e.g. in 3E-motifs *sma.NODE_TYPE_ECO*
- **level1** (int) – sesType of the other nodes, e.g. in 3E-motifs *sma.NODE_TYPE_SOC*
- **array** (bool) – whether the result should be an array instead of a dict
- **second_moment** (bool) – switch on for computing the second moment instead of the variance

sma.**var3EMotifs**(*G: networkx.classes.graph.Graph*, ***kwargs*)

Computes the variance of the number of 3E-motifs in a SEN. See the documentation for a exposition of the mathematical background. See also *sma.expected3Motifs()*.

Parameters

- **G** (Graph) – the SEN
- **kwargs** – additional parameters for *sma.var3Motifs()*.

sma.**var3SMotifs**(*G: networkx.classes.graph.Graph*, ***kwargs*)
Computes the variance of the number of 3S-motifs in a SEN. See the documentation for a exposition of the mathematical background. See also *sma.expected3Motifs()*.

Parameters

- **G** (Graph) – the SEN
- **kwargs** – additional parameters for *sma.var3Motifs()*.

sma.**var4Motifs**(*G: networkx.classes.graph.Graph*, *level0: int = 1*, *level1: int = 0*, *array: bool = False*, *second_moment: bool = False*)
Variances for the number of 4-motifs. See the documentation for a exposition of the mathematical background.

Parameters

- **G** (Graph) – the SEN
- **level0** (int) – sesType of the upper nodes, standard *sma.NODE_TYPE_SOC*
- **level1** (int) – sesType of the lower nodes, e.g. in 3E-motifs *sma.NODE_TYPE_ECO*
- **array** (bool) – whether the result shall be an array or a dict
- **second_moment** (bool) – switch on for computing the second moment instead of the variance

6.8.4 Actor's Choice Model

sma.**edgeContributionList**(*G: networkx.classes.graph.Graph*, **arities*, *roles=[]*, *level: int*) → numpy.ndarray
Computes edge contribution lists for a certain family of motifs. This list contains one entry for every possible edge in the level of the network specified by level. The entry associated with a dyad is number of times this dyad occurs in a certain family of motifs. The dyad itself is not taken into account here. In this way, open and closed motifs (i.e. motifs with a dyad in level level and otherwise equivalent motifs without this dyad) are grouped together. The value measures the contribution of a dyad in level level to open/closed motifs of similar class.

This only makes sense if the motif specified by arities (and roles) contains two nodes in level level.

For example, when working with 3-motifs (and level equalling the sesType of the non-distinct nodes), motifs I.A and II.A, I.B and II.B and I.C and II.C are grouped together.

This function does not work for directed graphs.

Return type ndarray

Parameters

- **G** (Graph) – the SEN
- **arities** – arities of the (multi-level) motif
- **roles** – optional roles for position matching, cf. *sma.matchPositions()*
- **level** (int) – level as described above

Returns

tuple of:

- two-dimensional array with one row for every group of motif (grouped by open/closed) and one column for every dyad on level `level`
- iterator giving the edges which index the columns of the array, call `list()` on it to obtain a list of indices

Raises **NotImplementedError** – if the motif signature is not supported

sma.**distributionMotifsActorsChoice**(*G: networkx.classes.graph.Graph*, **arities*, *roles=[]*, *level: int = - 1*, *array: bool = False*)

Computes expectation and variances for Actor's Choice distribution of motifs. See the documentation for a description of this model.

If `level = -1`, then the first entry equalling two in the signature of the motif is taken as level.

Parameters

- **G** (`Graph`) – the SEN
- **arities** – arities of the (multi-level) motif
- **level** (`int`) – level which is assumed to be variable in the Actor's Choice Model
- **array** (`bool`) – whether the output is a tuple `(e,v)` where `e` is a vector of expectations and `v` a vector of variances, or dict mapping motif names to tuples of expectation and variance for this motif

Returns properties of the distribution as tuple or dict (see `array`)

6.8.5 Auxiliary functions

sma.**markovBound**(*value: numpy.ndarray*, *expectation: numpy.ndarray*) → numpy.ndarray

Uses Markov's inequality to compute bounds for the probability that a random variable with given expectation takes values higher than the given values.

$$\mathbb{P}[X \geq a] \leq \frac{\mathbb{E}[X]}{a}$$

where a is given by `values` and $\mathbb{E}[X]$ is given by `expectation`.

See Markov's inequality on Wikipedia.

Return type `ndarray`

Parameters

- **value** (`ndarray`) – the value
- **expectation** (`ndarray`) – the expectation

Returns bounds on the probabilities

sma.**cantelliBound**(*value: numpy.ndarray*, *expectation: numpy.ndarray*, *variance: numpy.ndarray*) → numpy.ndarray

Uses Cantelli's inequality to compute bounds for the probability that a random variable with given expectation takes values higher/lower than the given values.

$$\mathbb{P}[X - \mathbb{E}[X] \geq a] \leq \begin{cases} \frac{\sigma^2}{\sigma^2 + a^2}, & a > 0 \\ 1 - \frac{\sigma^2}{\sigma^2 + a^2}, & a < 0 \end{cases}.$$

where a is given by `value` minus `expectation` and σ^2 is given by `variance`.

See Cantelli's inequality on Wikipedia.

Return type `ndarray`

Parameters

- **value** (`ndarray`) – the value

- **expectation** (ndarray) – the expectation

Returns bounds on the probabilities

6.9 Simulate baselines

sma.**simulateBaselineAuto**(*G: networkx.classes.graph.Graph*, **motifs*, *n: int = 100*, *processes: int = 0*, *chunksize: int = 100*, *assume_sparse: bool = False*, *model: str = 'erdos_renyi'*, *level: int = - 1*) → dict

Simulates a random baseline of graphs similar to a given SEN and counts motifs in these randomly generated graphs.

See *sma.randomSimilarMultiSENs()* for a documentation of the various baseline models.

Return type dict

Parameters

- **G** (Graph) – the SEN
- **motifs** – motif identifier strings of motifs that shall be counted
- **n** (int) – number of iterations
- **processes** (int) – number of processes, default zero (no multiprocessing)
- **chunksize** (int) – chunksize for multiprocessing
- **assume_sparse** (bool) – whether the random graphs shall be assumed to be sparse. Used to find an ideal counter, cf. *sma.findIdealCounter()*.
- **model** (str) – model to be used, default *sma.MODEL_ERDOS_RENYI*.
- **level** (int) – sesType for Actor's Choice model, see *sma.randomMultiSENsActorsChoice()*.

Returns dict mapping motif identifiers to list of counts

6.10 Other simple network properties

sma.**density**(*G: networkx.classes.graph.Graph*) → float

Returns the density of a given graph G = (V, E) defined as

$$\frac{2|E|}{|V|(|V|-1)}.$$

See also *sma.densityMatrix()*.

Return type float

Parameters **G** (Graph) – the SEN

sma.**densityMatrix**(*G: networkx.classes.graph.Graph*) → numpy.ndarray

Returns an upper triangular matrix with edge densities for every subsystem, i.e. the entry-wise quotient of *sma.edgesCountMatrix()* and *sma.maxEdgeCountMatrix()*.

See also *sma.density()*.

Return type ndarray

Parameters **G** (Graph) – the SEN

sma.**nodesCount**(*G: networkx.classes.graph.Graph*, *array: bool = False*, *levels: list = None*) → numpy.ndarray

Returns either a dict or an array with the number of nodes in each sesType.

Return type ndarray

Parameters

- **G** (Graph) – the SEN
- **array** (bool) – whether the result is an array or a dict
- **levels** (list) – a list of levels to provide an empty result if the graph is empty

Raises **ValueError** – if the graph is empty and levels is None or if one or more nodes have no nodal attribute sesType.

sma.**edgesCount**(*G: networkx.classes.graph.Graph*, *array: bool = False*) → dict

Classifies all edges in a SEN according to the following classes.

- *sma.EDGE_TYPE_SOC_SOC* for edges linking two social nodes
- *sma.EDGE_TYPE_ECO_ECO* for edges linking two ecological nodes
- *sma.EDGE_TYPE_ECO_SOC* for edges linking a social and an ecological node

Since edges are undirected no other types occur.

See *sma.edgesCountMatrix()* for multi-level support.

Return type dict

Parameters

- **G** (Graph) – SEN as networkx graph
- **array** (bool) – whether the result is an array or a dict

Returns dict containing type-count pairs according to the list of types above

sma.**edgesCountMatrix**(*G: networkx.classes.graph.Graph*, *nTypes: int = None*) → numpy.ndarray

This function provides an analogous result as *sma.edgesCount()* for multi-level graphs. For a SEN with n types of nodes the returned matrix is upper-triangular and of dimension $n \times n$. The entry (i, j) contains the number of edges in the given graph linking nodes of type i and j.

The sum over all entries equals the total number of edges in the SEN.

The returned matrix can be used as a parameter for sma.randomMultiSENs().

Return type ndarray

Parameters

- **G** (Graph) – a SEN, possibly with more than two levels
- **nTypes** (int) – the number of types in the SEN. An ordinary social-ecological network has two levels. If None, the number of levels is determined automatically using *sma.sesTypes()*.

sma.**degreeDistribution**(*G: networkx.classes.graph.Graph*) → dict

Returns a dict mapping each node of the given network to a dict as returned by *sma.nodesCount()* giving the number of neighbours on the various levels.

Return type dict

Parameters **G** (Graph) – the SEN

Returns dict of dicts

sma.**nodesByType**(*G: networkx.classes.graph.Graph*) → dict

Returns a dict mapping the levels of the network to list of nodes on the respective level.

Return type dict

Parameters **G** (Graph) – the SEN

Returns dict of node names by level

sma.**untypedNodes**(*G: networkx.classes.graph.Graph*) → map
Returns a map object yielding all nodes which do not have a `sesType`-attribute. Most functions used in analyses require all nodes to have such an attribute.

Example: List all untyped nodes:

```
# Let G be a SEN
print(list(sma.untypedNodes(G)))
```

Return type map

Parameters **G** (Graph) – the SEN

Returns map object yielding all untyped nodes

sma.**sesTypes**(*G: networkx.classes.graph.Graph*) → set
Returns the set of all `sesTypes` occurring in the given graph.

Return type set

Parameters **G** (Graph) – the SEN

Returns set of all `sesTypes`, e.g. $\{0, 1\}$ in standard two-level SENs

sma.**adjacentEdgesCount**(*G: networkx.classes.graph.Graph*, *node*, *nTypes=2*) → numpy.ndarray
The i-th entry in the returned matrix row is the number of neighbours the given node has of the type i. For example, in a ordinary two-level SEN with social and ecological nodes the first entry contains the number of adjacent ecological nodes (type 0) whereas the second entry contains the number of social nodes (type 1). Multi-level SENs are supported.

Return type ndarray

Parameters

- **G** (Graph) – the SEN
- **node** – the node
- **nTypes** (int) – number of types in the SEN, usually 2 (social and ecological)

Returns matrix row with numbers of adjacent vertices by type

sma.**total4Motifs**(*G: networkx.classes.graph.Graph*) → int
Returns the total number of 4-motifs in a graph, i.e.

$$\binom{n_s}{2}\binom{n_e}{2} = \frac{n_s(n_s-1)n_e(n_e-1)}{4}$$

where n_s and n_e are respectively the numbers of social and ecological nodes.

See also *sma.totalMultiMotifs()*.

Return type int

Parameters **G** (Graph) – the graph

Returns total number of 4-motifs

sma.**total3EMotifs**(*G: networkx.classes.graph.Graph*) → int
Returns the total number of 3E-motifs in a graph, i.e.

$$\binom{n_s}{2}n_e = \frac{n_s(n_s-1)n_e}{2}$$

where n_s and n_e are respectively the numbers of social and ecological nodes.

See also *sma.totalMultiMotifs()*.

Return type int

Parameters **G** (Graph) – the graph

Returns total number of 3E-motifs

sma.**total3SMotifs**(*G: networkx.classes.graph.Graph*) → int
Returns the total number of 3S-motifs in a graph, i.e.

$$\binom{n_e}{2} n_s = \frac{n_e(n_e-1)n_s}{2}$$

where n_s and n_e are respectively the numbers of social and ecological nodes.

See also *sma.totalMultiMotifs()*.

Return type int

Parameters **G** (Graph) – the graph

Returns total number of 3S-motifs

sma.**totalMultiMotifs**(*G: networkx.classes.graph.Graph*, **arities: int*) → int
Returns the total number of multilevel motifs of given arities (number of nodes taken from each level) for the given graph. If the given arities are $a_1, \ldots, a_n$, then the returned value is

$$\prod_{i=1}^{n} \binom{N_i}{a_i}$$

where $N_1, \ldots, N_n$ are the amounts of vertices in the different levels.

The following function calls are equivalent, cf. *sma.total4Motifs()* but also *sma.total3EMotifs()* and *sma.total3SMotifs()*:

```
sma.total4Motifs(G, multi = True)
# and
sma.totalMultiMotifs(G, 2, 2)
```

Return type int

Parameters

- **G** (Graph) – the SEN
- **arities** (int) – the arities, i.e. number of nodes taken from each level

CHAPTER

SEVEN

ADVANCED ITERATION

Motif iterators are a sophisticated way for modelling sets of motifs. They can be used for 3- and 4-motifs and even for plain sets of nodes (1-motifs). These sets of motifs consist of a source (where the motifs are taken from) and conditions (which refine the source). Several sources and conditions are provided in this package. Users might extend them. See *sma.MotifIterator* for an introduction to the concept and to the technical background.

Fig. 1: Inheritance diagram of the motif iterators implemented in this module. Note, that all classes can be accessed by simply importing sma. Importing sma.iterate is not necessary as it is imported implicitely.

Example:

```
import sma

# some random graph
G = next(sma.randomSENs(10,5, socName = 'City'))

# how many 4-motifs of class II.C exist with node 'City 0'?
it = sma.FourMotifs(G) & sma.is4Class('II.C') & sma.hasNode('City 0')
print(len(list(it)))
```

These iteratators can be piped into the motif classification methods *sma.count4Motifs()* or *sma.count3Motifs()*:

```
import sma

# some random graph
```

(continues on next page)

(continued from previous page)

```
G = next(sma.randomSENs(10,5, socName = 'City'))

# get the number of occurences of the motif classes which contain
# the node 'City 0':
it = sma.FourMotifs(G) & sma.hasNode('City 0')
print(sma.count4Motifs(G, it))
```

7.1 Abstract classes

class sma.**MotifIterator**(*it*)

Motif iterators aim at describing abstact sets of motifs. The concept covers 4- and 3-motifs as well as sets of nodes (1-motifs). Basically, a set of motifs is defined by stating a "source" and "conditions".

A source is an instance of *sma.SourceMotifIterator*. It is an unconditioned iterator which yields all motifs in a network, i.e. all 4-motifs or all 3E-motifs.

A condition is a filter which describes a rule for the motifs in the set, i.e. "contains node X" or "contains a node with property Y". Conditions are modeled as instances of *sma.ConditionMotifIterator* or of its subclasses.

Conditions can be combined using the binary operators & (and) or | (or). Conditions and composed conditions can be combined with a source. For example:

"All 4-motifs" & "contains node X" & "contains a node with property Y > Z"

Note, that such a complex term must only contain one source but can contain several conditions. Linking a source with conditions by & or | has the same effect.

Technically, all classes related to motif iteration are Python iterators. Only sources are tangible Python iterators, whereas conditions are filters to these iterators.

Since in R binary operators are not available for complex types, this module provides synonymous functions for combining iterators which are callable in R as well as in Python, cf. *sma.motifSet()*, *sma.countAnyMotifs()*.

Motif iterators can be piped into this module's motif counting and motif classifying facilities, cf. *sma.count4Motifs()*, *sma.count3Motifs()*, *sma.countMotifs()*.

Parameters **it** – an iterator which provides motifs

class sma.**ComplexMotifIterator**(*G: networkx.classes.graph.Graph*, *source*, *condition=None*, *infostr='ComplexIterator'*)

Motif iterators can be combined to form complex motif iterators describing complex sets of motifs with several linked conditions. This class models any combination of motif iterators, i.e. two conditions (condition 1 AND condition B) or a source and a condition (all motifs from source which fulfill the condition).

See *sma.MotifIterator* for details.

Parameters

- **G** – the graph where the motifs are taken from. This parameter is needed by some conditions which rely i.e. on information about the edges.
- **source** – an iterator which provides the motifs
- **condition** – a function which maps a motif to either True or False, stating whether a motif is contained in the set. If None, no condition is used (the condition returns always true).
- **infostr** – a string represantation, to be returned by `__str__()`

class sma.**ConditionMotifIterator**(*condition*)

Super class of all motif iterators representing a condition (usually without providing the source). Note, that an instance of this class cannot be iterated, because no source is provided.

Example "contains node X", cf. *sma.hasNode*.

Parameters **condition** – function mapping a motif to True or False, stating whether the motif is contained by the set. Note, that the internal format of the motifs depends on the source, cf. the subclasses of *sma.SourceMotifIterator*.

class sma.**SourceMotifIterator**(*G: networkx.classes.graph.Graph*, *source: sma.iterate.MotifIterator*)

Representation for a motif source, i.e. a raw, unconditioned set of motifs. Examples are the set of all 4-motifs, cf. *sma.FourMotifs* and the set of all 3-motifs, cf. *sma.ThreeEMotifs* for 3E-motif.

SourceMotifIterators not only provide an iterator but also a graph object, which is needed by some *ConditionMotifIterator* for evaluating for example if certain edges exist or not.

This class might not be used in practice. See its subclass for implementations.

Parameters

- **G** – the graph which this iterator takes motifs from
- **source** – an iterator providing motifs from the graph

7.2 Motif sources

class sma.**MultiMotifs**(*G: networkx.classes.graph.Graph*, **arities: int*, *flatten: bool = False*, *subsystems: list = None*)

Instances of this class represent a sets of multi-motif. A multi-motif is a motif spanning across one or several levels of a SEN. In this way it is a generalization of the motifs, cf. *sma.FourMotifs*, *sma.ThreeEMotifs*, etc. A multi-motif is a tuple of tuples. Every of its entries represents a tuple of nodes drawn from the associated level of the SEN.

The parameters are used to specify how many nodes shall be taken from the levels of the SEN. The levels are numbered according to their `sesType`. Hence the first given integer states the number of nodes which shall be taken from level 0, the second from level 1 and so on. For example, in SEN with default `sesType` (0: ecological, 1: social) the following two lines are almost equivalent:

```
# Let G be some SEN
motifs1 = sma.MultiMotifs(G, 1, 2)
list_motifs1 = list(motifs1) # [((123,), (1, 2)), ...]

motifs2 = sma.ThreeEMotifs(G)
list_motifs2 = list(motifs2) # [(123, 1, 2), ...]
```

The fundamental difference between this class and *sma.ThreeEMotifs* is that the former yields tuples of tuples while the latter produces flat tuples of integers. This causes some compatibility problems. You will need to use *sma.MultiMotifClassificator* for classifying/counting multi-motifs instead of e.g. *sma.ThreeMotifClassificator*. Set `flatten = True` to make this iterator yield tuples of nodes (not of tuples).

With default parameters this iterator scans through `arities` and yields as many nodes as specified from the various subystems. For example, if `arities = [2, 1, 0]`, then the tuples will contain two nodes from level 0, one node from level 1 and zero nodes from level 2. However, if `subsystems` contains an array, the behaviour is different. Instead of taking a_i nodes from level i where $(a_0, \ldots, a_n)$ denotes the array in `arities`, this iterator yields a_i nodes from level s_i where $(s_0, \ldots, s_n)$ denotes the array in `subsystems`.

See also *sma.MultiMotifsNormalized*.

Parameters

- **G** – the SEN
- **arities** – a list of integers representing the numbers of nodes which shall be taken from the different levels of the SEN.

- **flatten** – whether the tuples shall be flattened, i.e. whether `((123,), (1, 2))` shall be transformed into `(123, 1, 2)`.
- **subsystems** – specify the subsystems from which the nodes shall be taken, see above

class sma.**MultiMotifsNormalized**(*G: networkx.classes.graph.Graph*, **arities: int*, *roles=[]*, *motif_info=None*)

Source for multi-motifs with entries in the order specified by the signature of the motif. For example, 3-motifs are always given as tuples (A, B, B) regardless of the `sesType`'s of the levels that A and B come from.

Position matching is used to identify the relavent levels. Extends *sma.MultiMotifs*.

Parameters

- **G** – the SEN
- **arities** – list of arities
- **roles** – roles, optionally
- **motif_info** – a subclass of *sma.MotifInfo*, optionally

class sma.**FourMotifs**(*G: networkx.classes.graph.Graph*)

Set of all 4-motifs in a graph, cf. *sma.iterate4Motifs()*.

class sma.**ThreeEMotifs**(*G: networkx.classes.graph.Graph*)

Set of all 3E-motifs in a graph, cf. *sma.iterate3EMotifs()*.

class sma.**ThreeSMotifs**(*G: networkx.classes.graph.Graph*)

Set of all 3S-motifs in a graph, cf. *sma.iterate3SMotifs()*.

class sma.**OneMotifs**(*G: networkx.classes.graph.Graph*)

Set of all 1-motifs, i.e. nodes, in a graph. This iterator can be used for counting nodes with certain properties:

class sma.**TwoMotifs**(*G: networkx.classes.graph.Graph*)

Set of all tuples (s, e) where s is a social node and e is an ecological node. Note, that this set is only a superset of the set of all social-ecological edges, since not all social and ecological nodes must be connected.

class sma.**DenseMulti221Motifs**(*G: networkx.classes.graph.Graph*, *level0*, *level1*, *level2*, *optimize_top_adjacent: bool = False*)

Set of all tuples (a_1, a_2, b_1, b_2, c) such that:

- a_1 and a_2 are from the upper level (`level0`),
- b_1 and b_2 are from the middle level (`level1`),
- c is from the lower level (`level2`),
- b_1 and b_2 are adjacent to c,

and if `optimize_top_adjacent = True` additionally:

- a_1 and a_2 are adjacent to c.

In other words, this motif source contains all *CLASS*.2 and *CLASS*.3 (when used with default parameters) or when `optimize_top_adjacent = True` all *CLASS*.3 motifs.

This motif source is used by *sma.count221MotifsSparse()*.

Parameters

- **G** – the SEN
- **level0** – `sesType` of the upper level
- **level1** – `sesType` of the middle level
- **level2** – `sesType` of the lower level
- **optimize_top_adjacent** – whether the additional condition above should be imposed.

7.3 Conditions

class sma.**hasProperty**(*prop*, *rule*)
Condition for motifs modelling that at least one of the nodes in the motif has nodal attribute prop and that rule returns True for this value.

Since it is apparently not possible to transfer lambda/anonymous functions from R to Python using reticulate, some variants of this condition are implemented, cf. *sma.hasPropertyGreaterThan* and sma.hasPropertyMatch.

This condition works for any motif set.

Parameters

- **prop** – the name of the nodal property to look for
- **rule** – a function mapping the value of this attribute to either True or False, stating whether the motif should be contained in the set

Example `hasProperty('EXP', lambda x : x > 2)` gives all motifs with the property that at least one of the nodes has the attribute 'EXP' with value greater than 2.

class sma.**hasPropertyGreaterThan**(*prop*, *threshold*)
Condition for motifs modelling that at least one of the nodes in the motif has nodal attribute prop and that its value is greater than threshould.

This condition is equivalent to hasProperty(prop, lambda x : x > threshold).

This condition works for any motif set.

Parameters

- **prop** – the name of the nodal property to look for
- **threshold** – lower boundary for the property's values

class sma.**hasPropertyMatching**(*prop*, *value*)
Condition for motifs modelling that at least one of the nodes in the motif has nodal attribute prop and that its value matches value.

This condition is equivalent to hasProperty(prop, lambda x : x == value).

This condition works for any motif set.

Parameters

- **prop** – the name of the nodal property to look for
- **value** – value the nodal property must have

class sma.**hasNode**(*node*)
Condition for motifs modelling that the motif contains a certain node.

This condition works for any motif set.

Parameters **node** – the node to look for

class sma.**isClass**(*classificator: sma.classify.MotifClassificator*, **typ*)
Condition for any sort of motif modelling that a motif is of a certain class. Provide a *sma.MotifClassificator* as first parameter. The classification generated by this classificator will be used to match the given classes.

Parameters

- **classificator** – a *sma.MotifClassificator* for classifying the motifs
- **typ** – one or several classes

class sma.**is4Class**(**typ*)
Condition for 4-motifs modelling that the motif is of certain classes, cf. *sma.classify4Motif()*.

This condition works only for 4-motifs.

Parameters **typ** – classes of the motifs to look for, e.g. 'II.C'.

class sma.**is3Class**(**typ*)
Condition for 3-motifs modelling that the motif is of certain classes, cf. *sma.classify3Motif()*.

This condition works only for 3-motifs.

Parameters **typ** – classes of the motifs to look for, e.g. 'II.C'.

class sma.**matchesPropertyPattern4Motif**(*ruleS1*, *ruleS2*, *ruleE1*, *ruleE2*)
Condition for 4-motifs modelling that the nodes in the motif have certain properties.

In contrary to *sma.hasProperty* this condition takes distinct nodes into account. Thus, rules like "one ecological node has property X and *the other* has property Y" can be formulated. If this distinction is not necessary *sma.hasProperty* would be the better choice.

The parameters are function mapping a dict of nodal attributes to True or False, stating whether the node matches the criterion. All rules have to be matched for a motif to be in the set.

This condition may have to complex parameters. See *sma.matchesPropertyPattern4MotifGreaterThan* for a more straightforward version.

This condition works only for 4-motifs.

Parameters

- **ruleS1** – rule for the first social node
- **ruleS2** – rule for the second social node
- **ruleS3** – rule for the first ecological node
- **ruleS4** – rule for the second ecological node

class sma.**matchesPropertyPattern4MotifGreaterThan**(*propS1=None*, *propS2=None*, *propE1=None*, *propE2=None*)
Condition for 4-motifs modelling that the nodes in the motif have certain properties with values greater than a certain threshold.

In contrary to *sma.hasPropertyGreaterThan* this condition takes distinct nodes into account. Thus, rules like "one ecological node has property X and *the other* has property Y" can be formulated. If this distinction is not necessary, *sma.hasPropertyGreaterThan* would be the better choice.

All parameters are tuples (k, t) where k is the name of the property to check for and t is the threshold.

Example Possible parameters could be `propS1 = ('EXP', 2)` for getting all motifs with social nodes with nodal property EXP greater than 2.

Parameters

- **propS1** – tuple (k, t) as described above for the first social node
- **propS2** – tuple (k, t) as described above for the second social node
- **propE1** – tuple (k, t) as described above for the first ecological node
- **propE2** – tuple (k, t) as described above for the second ecological node

class sma.**matchesPropertyPattern3Motif**(*ruleA*, *ruleB1*, *ruleB2*)
Condition for 3-motifs modelling that the nodes in the motif have certain properties.

In contrary to *sma.hasProperty* this condition takes distinct nodes into account. Thus, rules like "one ecological node has property X and *the other* has property Y" can be formulated. If this distinction is not necessary *sma.hasProperty* would be the better choice.

The parameters are functions mapping a dict of nodal attributes to True or False, stating whether the node matches the criterion. All rules have to be matched for a motif to be in the set.

This condition may have to complex parameters. See *sma.matchesPropertyPattern3MotifGreaterThan* for a more straightforward version.

This condition works only for 3-motifs.

Parameters

- **ruleA** – rule for the distinct node (the ecological node in 3E-motifs or the social node in 3S-motifs)
- **ruleB1** – rule for the first other node
- **ruleB2** – rule for the second other node

class sma.**matchesPropertyPattern3MotifGreaterThan**(*propA=None*, *propB1=None*, *propB2=None*)

Condition for 3-motifs modelling that the nodes in the motif have certain properties with values greater than a certain threshold.

In contrary to *sma.hasPropertyGreaterThan* this condition takes distinct nodes into account. Thus, rules like "one ecological node has property X and *the other* has property Y" can be formulated. If this distinction is not necessary *sma.hasPropertyGreaterThan* would be the better choice.

All parameters are tuples (k, t) where k is the name of the property to check for and t is the threshold.

Example Possible parameters could be `propA = ('EXP', 2)` for getting all motifs with distinct nodes with nodal property EXP greater than 2.

Parameters

- **propA** – tuple (k, t) as described above for the the distinct node (the ecological node in 3E-motifs or the social node in 3S-motifs)
- **propB1** – tuple (k, t) as described above for the first other node
- **propB2** – tuple (k, t) as described above for the second other node

CHAPTER

EIGHT

INTEGRATION IN R

Many functions of this package are made available in R by the R package motifr. See its documentation for a user-friendly introduction. This section describes the technicalities of the underlying interface.

All functions provided by this module can be used in R. On the R side we use the `statnet` package for modelling networks. The `reticulate` package is used for making the Python functions accessible in R.

Note, that this is written in Python 3 and makes use of its language features. Running it with Python 2 is not possible. Thus, the user is requested to assure that Python 3 is installed and accessible.

The following code is needed for setting up the bridge between R and Python. You may need to adjust the paths ll. 4, 6.

```
1  library(statnet)
2
3  # set-up python interface
4  Sys.setenv(RETICULATE_PYTHON = "/usr/bin/python3") # ensure Python version 3
5  library("reticulate")
6  sma <<- import_from_path('sma', path = '.')
7
8  toPyGraph <- function(g, typeAttr = 'sesType') {
9          # function for translating a statnet network object into a Python
10         # compatible networkx object
11         adjacencyMatrix = as.matrix(g)
12         attributeNames  = list.vertex.attributes(g)
13         attributeValues = lapply(list.vertex.attributes(g),function(x) get.vertex.
   ↪attribute(g,x))
14
15         sma$translateGraph(adjacencyMatrix, attributeNames, attributeValues,␣
   ↪typeAttr)
16 }
```

Since SENs – and not arbitrary networks – are considered, the social-ecological type of the nodes is of crucial importance. Therefore, the user needs to specify the nodal attribute which shall be interpreted as this type as second parameter of the function `toPyGraph`. In the last line *`sma.translateGraph()`* is called. See its documentation for technical background information.

Note, that the social-ecological type must be modelled in accordance with the constants *`sma.NODE_TYPE_ECO`* and *`sma.NODE_TYPE_SOC`*.

Example (cont.), cf. *`sma.count4Motifs()`* and *`sma.triangleCoefficients()`*:

```
# load graph
load("firefighters.RData")
g = firefighters # statnet network object from R data file

# get python equivalent
pyGraph = toPyGraph(g)

# perform analysis
```

(continues on next page)

(continued from previous page)

```
sma$count4Motifs(pyGraph, array=TRUE) # count 4-motifs
sma$triangleCoefficients(pyGraph, array=TRUE) # compute triangle coefficients
```

Some Python specific features used by the advanced iteration functions are not available in R. The functions listed in the following section can be used as substitute for these features.

Example:

```
set = sma$motifSet(sma$FourMotifs(pyGraph), sma$is4Class('II.C'), sma$hasNode(
↪'City 0'))
# count 4-motifs of class II.C with node 'City 0'
sma$countAnyMotifs(set)
```

Note, that a set object, as returned by *sma.motifSet()*, can only be used once. Functions like *sma.count4Motifs()* or *sma.listMotifs()* empty the iterators when they are executed.

8.1 Auxiliary functions

sma.**translateGraph**(*adjacency*, *attributeNames*, *attributeValues*, *typeAttr: str = None*, *directed: bool = False*) → networkx.classes.graph.Graph

Returns a `networkx.Graph` from a adjacency matrix, a list of nodal attribute names and a matrix of nodal attribute values. This function is designed to make networkx objects compatible with R's statnet network objects.

Return type `Graph`

Parameters

- **adjacency** – adjacency matrix provided as a numerical matrix
- **attributeNames** – list of attribute names
- **attributeValues** – matrix of attribute values in accordance to *attributeNames*.
- **typeAttr** (`str`) – name of an attribute with specifies whether a node is social or ecological. The values must be integers, matching *sma.NODE_TYPE_SOC* and *sma.NODE_TYPE_ECO* in two-level networks. If None, attributeNames must contain 'sesType'.
- **directed** (`bool`) – whether the graph is directed

Returns networkx graph with the given properties

Raises **ValueError** – in case of invalid parameters

sma.**countAnyMotifs**(*it: sma.iterate.ComplexMotifIterator*) → int

Counts the number of elements in the given iterator.

Function for compatability with R.

sma.**motifSet**(*source: sma.iterate.SourceMotifIterator*, **conditions: sma.iterate.ConditionMotifIterator*) → sma.iterate.ComplexMotifIterator

Returns an motif interator representing a set of motifs with certain properties.

Function for compatability with R.

Return type `ComplexMotifIterator`

Parameters

- **source** (`SourceMotifIterator`) – *sma.SourceMotifIterator*, the basal set to take the motifs from

- **conditions** (ConditionMotifIterator) – one or several *sma.ConditionMotifIterator* representing conditions which the motifs must fulfill to be part of the set

sma.**listMotifs**(*iterator: sma.iterate.ComplexMotifIterator*) → list
Returns a list of all motifs in a motif iterator.

Function for compatability with R.

Return type list

Parameters **iterator** (ComplexMotifIterator) – motif iterator

sma.**countMotifsAutoR**(*G: networkx.classes.graph.Graph*, *motifs: list*, *omit_total_result: bool = False*, ***kwargs*) → pandas.core.frame.DataFrame
Wrapper function for *sma.countMotifsAuto()*. Optimized for the reticulate R interface.

Return type DataFrame

Parameters

- **G** (Graph) – the SEN
- **motifs** (list) – the list of motifs to be counted
- **omit_total_result** (bool) – whether only the partial result shall be returned
- **kwargs** – further parameters for *sma.countMotifsAuto()*

Returns pandas dataframe with either partial or total counts

sma.**distributionMotifsAutoR**(*G: networkx.classes.graph.Graph*, *motifs: list*, *model: str = 'erdos_renyi'*, *level: int = - 1*, *omit_total_result: bool = False*) → pandas.core.frame.DataFrame
Wrapper function for *sma.distributionMotifsAuto()*. Optimized for the reticulate R interface.

Return type DataFrame

Parameters

- **G** (Graph) – the SEN
- **motifs** (list) – the list of motifs whose distributions are to be described
- **omit_total_result** (bool) – whether only the partial result shall be returned

Returns pandas dataframe with either partial or total distribution information

sma.**simulateBaselineAutoR**(*G: networkx.classes.graph.Graph*, *motifs: list*, ***kwargs*) → pandas.core.frame.DataFrame
Wrapper function for *sma.simulateBaselineAuto()*. Optimized for the reticulate R interface.

Return type DataFrame

Parameters

- **G** (Graph) – the SEN
- **motifs** (list) – the list of motifs whose distributions are to be described
- **kwargs** – further parameters for *sma.simulateBaselineAutoR()*.

Returns pandas dataframe with one column for every motif and one row for every random SEN

sma.**identifyGapsR**(*G: networkx.classes.graph.Graph*, *motif_identifier: str*, *level: int = - 1*) → pandas.core.frame.DataFrame
Wrapper function for *sma.identifyGaps()*. Optimized for the reticulate R interface.

Return type DataFrame

Parameters

- **G** (Graph) – the SEN
- **motif_identifier** (str) – a motif identifier

- **level** (int) – level

Returns pandas dataframe with three columns: two for the edges, one for their contribution

sma.**isClosedMotifR**(*motif_identifier: str*, *level: int = - 1*) → bool

Wrapper function for *sma.isClosedMotif()*. Optimized for the reticulate R interface.

Return type bool

Parameters

- **motif_identifier** (str) – motif identifier
- **level** (int) – level

Returns whether the motif is closed with respect to the specified level

Raises **NotImplementedError** – if open/closed relations are not implemented for this motif and level

CHAPTER

NINE

AUXILIARY FUNCTIONS

9.1 Identifying motifs

Functions in this section are used to identify multi-level motifs, for example in *sma.MultiMotifClassificator*, and for interpreting motif identifier strings as defined in *Terminology*. Structural information about motifs are kept in subclasses of *sma.MotifInfo* and can be accessed via functions described in this section.

9.1.1 Motif database

sma.**motifInfo**(*signature: list*, *directed: bool*)

Returns a subclass of *sma.MotifInfo* correponding to the given signature. The order of the signature is irrelevant.

Parameters

- **signature** (list) – the signature
- **directed** (bool) – whether the motif is a directed graph

Returns subclass of *sma.MotifInfo*

Raises **NotImplementedError** – if no subclass of *sma.MotifInfo* is available for the given signature

sma.**supportedSignatures**() → iter

Returns an iterator yielding all tuples of supported signatures together with a boolean value indicated whether the corresponding motifs are directed or not.

```
print(list(sma.supportedSignatures()))
```

Returns iterator yielding all supported signatures.

class sma.**MotifInfo**

Parent class of all MotifInfo classes. Subclasses contain information about motifs, their classificators and counters. The following properties are provided:

Variables

- **signature** – signature of the motifs
- **classes** – list of names of all motif classes
- **classes_type** – type of the class names, e.g. str or int
- **directed** – whether the motifs are considered as directed graphs
- **classificator** – subclass of *sma.MotifClassificator* for classifying motifs

- **counter** – counting function with signature

  `counter(G, *levels, **kwargs)`

- **sparse_counter** – counting function for sparse graphs with signature

  `sparse_counter(G, *levels, **kwargs)`

- **expectations** – function for computing motif expectations in a Erdős-Rényi model

  `expectations(G, *levels, **kwargs)`

- **variances** – function for computing motif variances in a Erdős-Rényi model

  `variances(G, *levels, **kwargs)`

- **projections** – dict containing information about closed/open pairs of motifs, i.e. which motif classes belong together when non considering dyads on a specific level of the motif. The dict is indexed by abstract levels (indexes of levels in `signature`). Every entry is a list of tuples of motif class names. Each tuple contains first the open and secondly the closed motif. See *sma.Motif3Info* as an example.

If a certain feature is not implemented, the corresponding value is set to `None`.

The following families of motifs are supported. Visit the motif zoo in *Appendix: The Motif Zoo* for details.

One level motifs

class sma.**Motif3pInfo**
3p-motifs have signature (3). So they consist of three nodes on the same level.

class sma.**Motif1Info**
1-motifs are most boring. They contain just a single node. See also `sma.DiMotif1Info`.

class sma.**Motif2Info**
2-motifs are similar to (1,1)-motifs (cf. *sma.Motif11Info*). They have signature (2), so consist of two nodes on the same level.

class sma.**DiMotif2Info**
Directed motifs with signature (2). See also *sma.DiMotif11Info*.

Two levels motifs

class sma.**Motif11Info**
(1,1)-motifs are similar to 2-motifs (cf. *sma.Motif2Info*). They have signature (1,1), so consist of two nodes on different levels.

class sma.**Motif3Info**
3-motifs have signature (1,2). In two-level networks, they are often called 3E- or 3S-motifs depending on the `sesType` of the distinct node.

class sma.**Motif4Info**
4-motifs have signature (2,2). In two-level networks, the first level is usually the social level (*sma.NODE_TYPE_SOC*) while the second level is the ecological level (*sma.NODE_TYPE_ECO*).

class sma.**DiMotif11Info**
Directed motifs with signature (1, 1), i.e. with two distinct nodes. See also *sma.DiMotif2Info*.

class sma.**DiMotif12Info**
Directed motifs with signature (1,2).

class sma.**DiMotif22Info**
Directed motifs with signature (2,2).

Three levels motifs

class sma.**Motif111Info**
Signature (1,1,1).

class sma.**Motif121Info**
Signature (1,2,1). Not all motifs with this signature are classified yet. Visit the motif zoo for details.

class sma.**Motif221Info**
Signature (2,2,1). Not all motifs with this signature are classified yet. Visit the motif zoo for details.

class sma.**Motif222Info**
Signature (2,2,2). Not all motifs with this signature are classified yet. Visit the motif zoo for details.

9.1.2 Working with Motif Identifier Strings

sma.**multiSignature**(*arities: list*) → list
Returns the signature of a list of arities. The signature is the list of non-zero entries in ascending order. It determines for example the counter function that can be used to count motifs with these arities.

Return type list

Parameters **arities** (list) – list of arities

Returns signature

sma.**splitMotifIdentifier**(*identifier: str*) → tuple
Splits a motif identifier into the first part (encoding the pattern and roles of the motif) and the second part (encoding the type of the motif).

For example, `1:2,2:1[I.C]` is mapped to the tuple consisting of `1:2,2:1` and `I.C`.

Return type tuple

Parameters **identifier** (str) – motif identifier string

Raises **ValueError** – in case of parsing problems

Returns tuple consisting of first and second part

sma.**parseMotifIdentifier**(*identifier: str*) → tuple
Parser for motif identifier strings. This function extracts a patterns, roles and motif from a motif identifier string, cf. *sma.MultiMotifClassificator*.

A simple motif identifier, e.g. `1,2[I.C]`, consist of a comma-separated list of integers specifying the pattern of the motif (here, 1 node from one level, 2 nodes from the other), and of an optional motif name enclosed in brackets.

Motif identifiers can also be used to specify the roles of the levels. An example is `1:1,2:0[I.C]`. The values behind the colons are the roles of the levels.

Return type tuple

Parameters **identifier** (str) – motif identifier string

Returns

triple consisting of:

- an arities array
- a roles array, or `[]` if no roles are specified
- a motif, or `None` if no motif is specified

Raises **ValueError** – in case of parsing problems

sma.**groupMotifIdentifiers**(**motifs: str*) → dict

Groups motif identifiers by their heads in a dictionary. For example, the list `1,2[I.A], 2,2[II.B], 1,2[II.B]` is mapped to `{'1,2': ['I.A', 'II.B'], '2,2': ['II.B']}`. The returned dict is indexed by the heads of the identifiers, the values are lists of motif classes.

Return type `dict`

Parameters **motifs** (`str`) – list of motif identifiers

Returns dict of grouped motif identifiers

9.1.3 Position matching

See also Position matching.

sma.**matchPositions**(*signature: list*, *arities: list*, *roles: list = []*) → list

Position matching is the process in which a signature (e.g. `1, 2`) is mapped to list of positions, i.e. `sesType` values in the network (e.g., *`sma.NODE_TYPE_ECO`*, *`sma.NODE_TYPE_SOC`* in case of ecological triangles). Optionally, a list of roles can be provides. This overwrites the position matching.

Return type `list`

Parameters

- **pattern** – the signature
- **arities** (`list`) – the arities given
- **roles** (`list`) – optional list of rolls

Returns positions

Raises **AssertionError** – in case of mismatching signature/arities/roles

sma.**exemplifyMotif**(*G: networkx.classes.graph.Graph*, *identifier: str*) → tuple

Returns an example for a motif with given identifier in a SEN.

This function works for directed and undirected graphs. The directed case is distinguished by calling `networkx.is_directed()` on the given graph.

Return type `tuple`

Parameters

- **G** (`Graph`) – the SEN
- **identifier** (`str`) – motif identifier string

Returns motif tuple in normalised form (ordered as in signature) or None if there is no such motif in the network

Raises **ValueError** – if the given motif identifier string does not specify a motif class, i.e. if only `1,2` is given and not `1,2[I.A]`.

sma.**sortPositions**(*levels*, *arities*)

Maps a list of levels and arities to a list of arities ordered according to the positions.

For example `levels = [0, 2, 1]` and `arities = [3, 2, 1]` is mapped to `[3, 1, 2]`. There are 3 vertices on level 0, 2 vertices on level 2 and 1 vertex on level 1.

Parameters

- **levels** – list of levels
- **arities** – list of arities

Returns list of arities sorted according to positions

9.2 Manipulating SENs

sma.**addRandomEdges**(*G: networkx.classes.graph.Graph*, *ecoNodes: list*, *socNodes: list*, *edgesCounts: dict*) → networkx.classes.graph.Graph

Add a specified number of random edges to a given SEN.

Return type Graph

Parameters

- **G** (Graph) – the graph
- **ecoNodes** (list) – a list of the ecological nodes in the SEN
- **socNodes** (list) – a list of social nodes in the SEN
- **edgesCounts** (dict) – dict containing the amounts of social-social, ecological-ecological and social-ecological edges, labelled in accordance with *sma.edgesCount()*.

sma.**multiToWeightedGraph**(*M: networkx.classes.multigraph.MultiGraph*, *attr='weight'*) → networkx.classes.graph.Graph

Helper function for converting a multigraph into a weighted simple graph with edge weights corresponding to the multiplicities of the edges in the multigraph.

Nodal attributes are preserved while edge attributes will be lost. Use the second parameter to set the name of the weight attribute.

Return type Graph

Parameters

- **M** (MultiGraph) – a multigraph
- **attr** (str) – name of the weight attribute

Returns a simple graph with weighted edges

9.3 Constants

sma.**NODE_TYPE_SOC**

sma.**NODE_TYPE_ECO**

sma.**EDGE_TYPE_SOC_SOC**

sma.**EDGE_TYPE_ECO_ECO**

sma.**EDGE_TYPE_ECO_SOC**

sma.**MOTIF4_NAMES**

sma.**MOTIF3_NAMES**

sma.**MOTIF3_AUT**

sma.**COLORS_TYPES**

sma.**MULTI_DEFAULT_NAMES**

sma.**MOTIF3_EDGES**

sma.**MOTIF4_EDGES**

sma.**MOTIF4_SYMMETRIES**

sma.**MOTIF4_AUT**

sma.**MODEL_ERDOS_RENYI**

sma.**MODEL_ACTORS_CHOICE**

sma.**MODEL_FIXED_DENSITIES**

9.4 Motif classificators

These are internal helper functions. See *Motif classificators* for more practical and less technical implementations.

sma.**binaryCodeToClass4Motifs**(*digits: int*) → str
Classifies 4-motifs according to Ö. Bodin, M. Tengö: Disentangling intangible social–ecological systems Global Environmental Change 22 (2012) 430–439 http://dx.doi.org/10.1016/j.gloenvcha.2012.01.005

Return type str

Parameters **digits** (int) – binary code of the motif as described in the paper

Returns string code 'I.A' to 'VII.D'

Raises **ValueError** – if the given digits are not in the required range

sma.**binaryCodeToClass3Motifs**(*digits: int*) → str
Classifies 3-motifs according to the following scheme: A 3-motif consists of either one social and two ecological nodes or one ecological and two social nodes. Thus, one node can be considered distinct from the two other nodes. Let's call this node a and the other two nodes b_1 and b_2. The following classes occur (listing undirected edges):

class I: without edge (b_1, b_2)**:**

- I.A: no edges,
- I.B: either (a, b_1) or (a, b_2),
- I.C: both (a, b_1) and (a, b_2).

class II: with edge (b_1, b_2)**:**

- II.A: (b_1, b_2),
- II.B: (b_1, b_2) and either (a, b_1) or (a, b_2),
- II.C: (b_1, b_2) and both (a, b_1) and (a, b_2).

Return type str

Parameters **digits** (int) – binary code of the motif as described above

Returns string code 'I.A' to 'VII.D'

Raises **ValueError** – if the given digits are not in the required range

9.5 Drawing SENs

Most example graphs in this documentation were drawn using the following functions.

sma.**drawSEN**(*G: networkx.classes.graph.Graph*, *color_map={0: 'green', 1: 'red', 2: 'c', 3: 'm', 4: 'y', 5: 'k', 6: 'w'}*, ***kwargs*)
Draws a social-ecological network using networkx.draw_kamada_kawai(). The default drawing behaviour can be overwritten by providing the parameter pos.

Parameters

- **G** (Graph) – the graph
- **color_map** – a dict mapping the sesTypes of the nodes to colors, default is *sma.COLORS_TYPES*.

- **kwargs** – parameters for `networkx.draw_kamada_kawai()`

sma.**drawGeoSEN**(*G: networkx.classes.graph.Graph*, *longAttribute: str = 'long'*, *latAttribute: str = 'lat'*, ***kwargs*)

Draws a SEN whose nodes have a location property. `draw_networkx()` is used as backend.

Parameters

- **G** (`Graph`) – the graph
- **longAttribute** (`str`) – the name of the nodal attribute which contains the longitude of the nodes
- **latAttribute** (`str`) – the name of the nodal attribute which contains the latitude of the nodes
- **kwargs** – parameters for `draw_networkx()`

sma.**drawMotif**(*G: networkx.classes.graph.Graph*, *motif: tuple*, ***kwargs*)

Draws a specific motif in a SEN.

Example:

```
# let G be some SEN
motif = sma.exemplifyMotif(G, '1,2[II.C]')
sma.drawMotif(G, motif)
```

Parameters

- **G** (`Graph`) – the SEN
- **motif** (`tuple`) – a motif (tuple of nodes)
- **kwargs** – more parameters for *sma.drawSEN()* and `networkx.draw_networkx()`.

sma.**layer_layout**(*G: networkx.classes.graph.Graph*, *scale: float = 5*)

Computes a layout for a SEN which places the nodes of one type on one horizontal line. Use it together with *sma.drawSEN()*.

```
import sma
# Let G be some SEN
sma.drawSEN(G, pos=sma.layer_layout(G))
```

Parameters

- **G** (`Graph`) – the SEN
- **scale** (`float`) – scaling factor

Returns layout for *sma.drawSEN()*

sma.**advanced_layer_layout**(*G: networkx.classes.graph.Graph*, *space: float = 1*)

Computes a layout for a SEN which places the nodes of one type close to each other. Use it together with *sma.drawSEN()*.

Parameters

- **G** (`Graph`) – the SEN
- **space** (`float`) – space between the layers

Returns layout for *sma.drawSEN()*

9.6 Other Functions

sma.**maxEdgeCountMatrix**(*nVertices: numpy.ndarray*) → numpy.ndarray
Returns a matrix containing the maximal possible number of edges in the subsystems of a multigraph. The entries of this matrix constitute the upper bounds of the entries of the `nEdges` parameter of `sma.randomMultiSENs()`.

Let $v_1, \ldots, v_n$ be the entries of `nVertices`. Then the entries of the returned matrix are

$$a_{ij} = \begin{cases} \frac{1}{2} v_i (v_i - 1), & i = j \\ v_i v_j, & i < j \end{cases}.$$

See also *sma.motifClassMatrix()*.

Return type ndarray

Parameters **nVertices** (ndarray) – numbers of vertices per subsystem

sma.**randomEdgeCountMatrix**(*nVertices: numpy.ndarray*) → numpy.ndarray
This function takes the result of *sma.maxEdgeCountMatrix()* and a matrix with random integer entries less or equal this matrix. It can be used to generate a random multilevel SEN with `sma.randomMultiSENs()`.

Return type ndarray

Parameters **nVertices** (ndarray) – numbers of vertices per subsystem

sma.**progress_indicator**(*iterable*, *total: int*, *width: int = 75*, *msg: str = 'Classifying {:,} motifs'*)
Wraps an iterator and shows a progress indicator.

Parameters

- **iterable** – the iterator
- **total** (int) – total number of items to be processed
- **width** (int) – width of the progress indicator
- **msg** (str) – message to print above progress indicator. Use `{:,}` as placeholder for number of items.

CHAPTER

TEN

APPENDIX: THE MOTIF ZOO

Motifs are identified using *motif identifiers*, e.g. `1,2[I.C]`, the open triangle with one node on level 0 and two nodes on level 1. They describe the position of the motif in the network (*positions*) and the structure of the motif itself (*signature* and *motif class*). See *Terminology* for a complete definition.

This section lists the motif classes for the signatures supported by this software. The motifs are grouped by the number of levels they span across.

10.1 Undirected motifs

Motifs in this section occur in undirected networks.

10.1.1 One level motifs

Motifs in this section contain nodes from only one level.

1-motifs

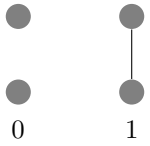
- **Signature:** `1`
- **Motif Info Object**: *sma.Motif1Info*

A 1-motif consists of just one node. Therefore there is also only one motif class called `1`.

Plain 2-motifs

- **Signature:** `2`
- **Motif Info Object**: *sma.Motif2Info*

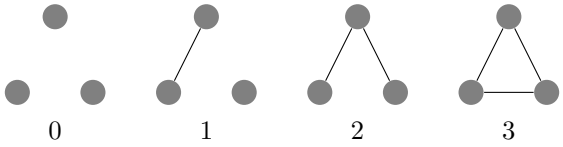
Plain 2-motifs contain two nodes from the same level.



Plain 3-motifs

- **Signature:** `3`
- **Motif Info Object**: *sma.Motif3pInfo*

Plain 3-motifs contain three nodes from the same level.



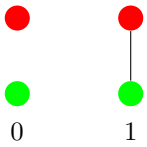
10.1.2 Two level motifs

Motifs in this section contain nodes from two distinct levels.

2-motifs

- **Signature:** `1,1`
- **Motif Info Object**: *sma.Motif11Info*

2-motifs contain two nodes from distinct levels.

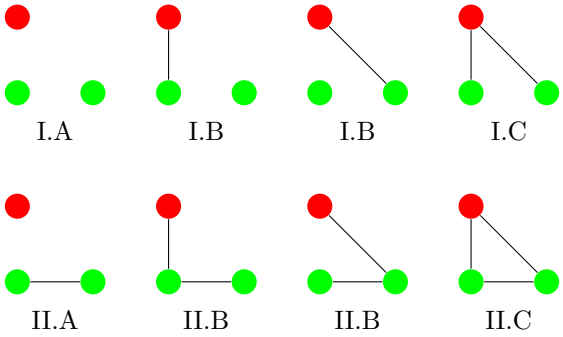


3-motifs

- **Signature:** `1,2`
- **Motif Info Object**: *sma.Motif3Info*

3-motifs can be classified in 6 classes which will be called I.A to II.C. The following figure shows the configurations for the classes of 3S-motifs, i.e. the distinct node is social (colouring ecological nodes green and social nodes red). 3E-motifs are classified analogously. In the figure only the colours would have to be swapped.

See also the classification for the corresponding directed motifs in *(1,2)-motifs*.

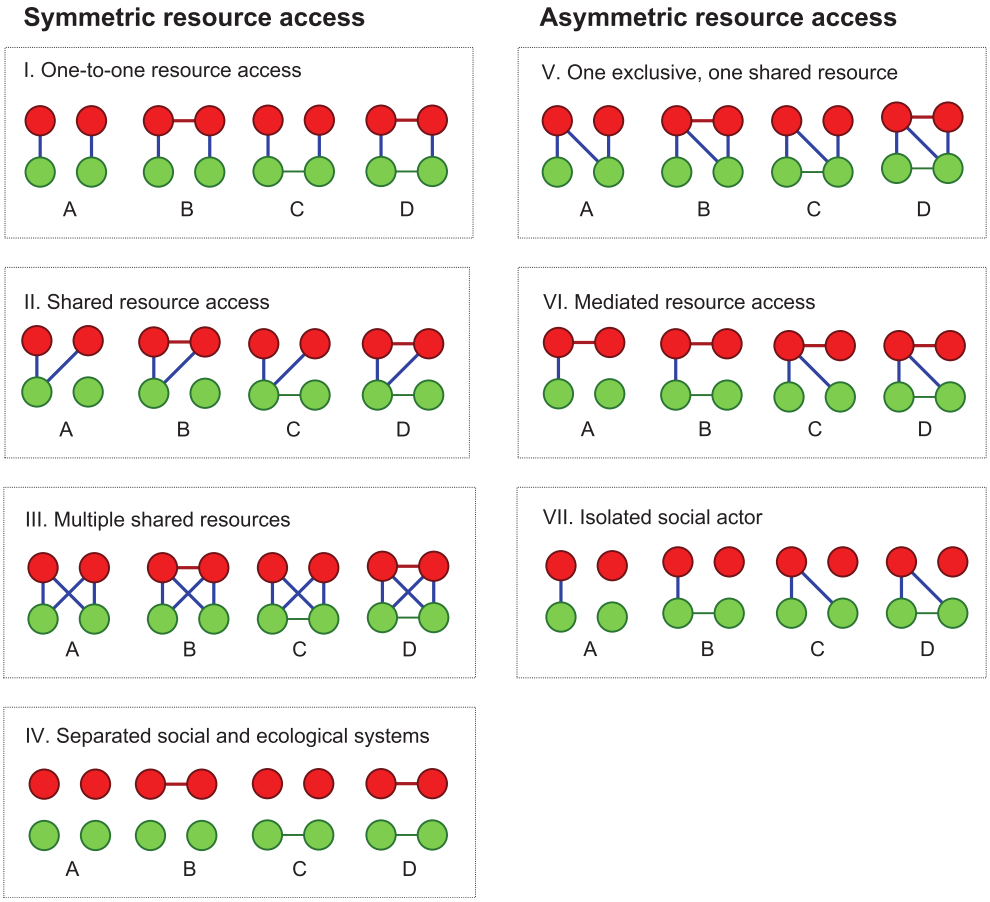


4-motifs

- **Signature:** `2,2`
- **Motif Info Object**: *`sma.Motif4Info`*

The classification for 4-motifs used in this software was proposed by Ö. Bodin and M. Tengö. The following illustration is taken from their original paper.

Reference Ö. Bodin, M. Tengö: Disentangling intangible social–ecological systems in Global Environmental Change 22 (2012) 430–439 http://dx.doi.org/10.1016/j.gloenvcha.2012.01.005



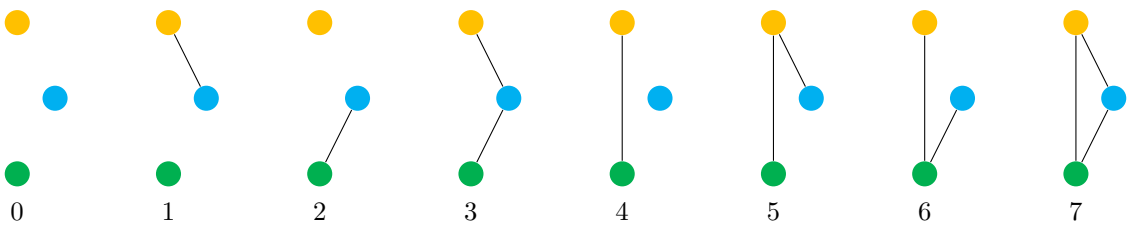
10.1.3 Three level motif

Motifs in this section contain nodes from three distinct levels.

(1,1,1)-motifs

- **Signature:** `1,1,1`
- **Motif Info Object**: *`sma.Motif111Info`*

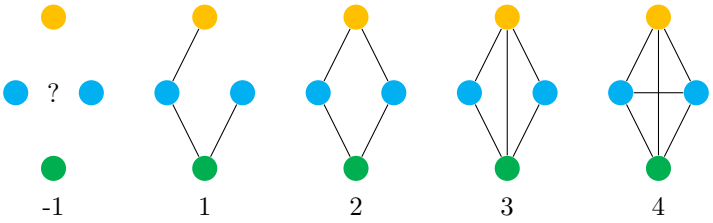
The levels are numbered from the top to the bottom.



(1,2,1)-motifs

- **Signature:** `1,2,1`
- **Motif Info Object**: *sma.Motif121Info*

The levels are numbered from the top to the bottom. -1 represents all other motifs, i.e. the motifs that do not fall into the classes specified by $1, \ldots, 4$.

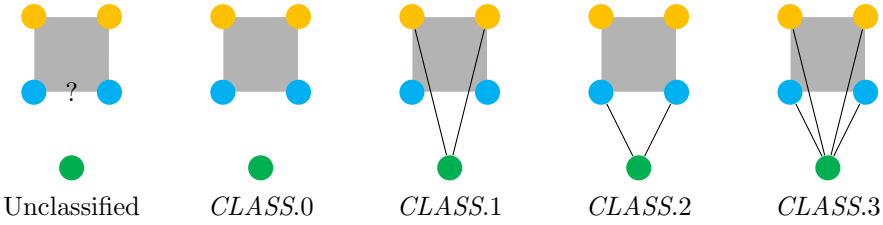


(2,2,1)-motifs

- **Signature:** `2,2,1`
- **Motif Info Object**: *sma.Motif221Info*

The levels are numbered from the top to the bottom. The grey box is a placeholder for a 4-motif (see *4-motifs*) consisting for nodes of two top levels. If for example the gray box contains a III.D motif (complete graph), then the (2,2,1)-motif is either of class III.D.0 (no edges to the bottom level), III.D.1 (only both top-bottom edges), III.D.2 (only both middle-bottom edges), III.D.3 (all top-bottom and middle-bottom edges) or Unclassified (if it does not fall into one of the preceding classes).

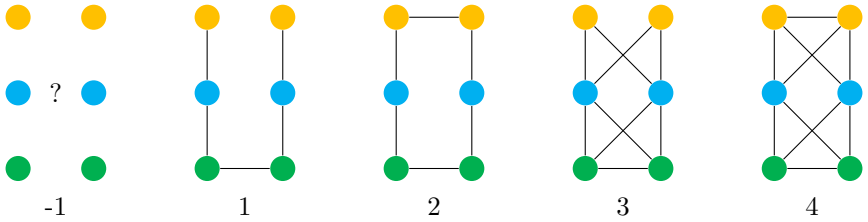
Unclassified represents all other motifs, i.e. the motifs that do not fall into the other classes.



(2,2,2)-motifs

- **Signature:** `2,2,2`
- **Motif Info Object**: *sma.Motif222Info*

The levels are numbered from the top to the bottom. -1 represents all other motifs, i.e. the motifs that do not fall into the classes specified by $1, \ldots, 4$.



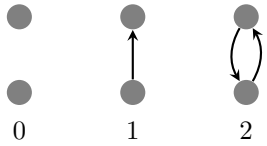
10.2 Directed motifs

Motifs in this section occur in directed networks.

10.2.1 One level motifs

Plain 2-motifs

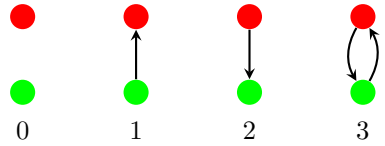
- **Signature:** `2`
- **Motif Info Object**: *`sma.DiMotif2Info`*



10.2.2 Two level motifs

2-motifs

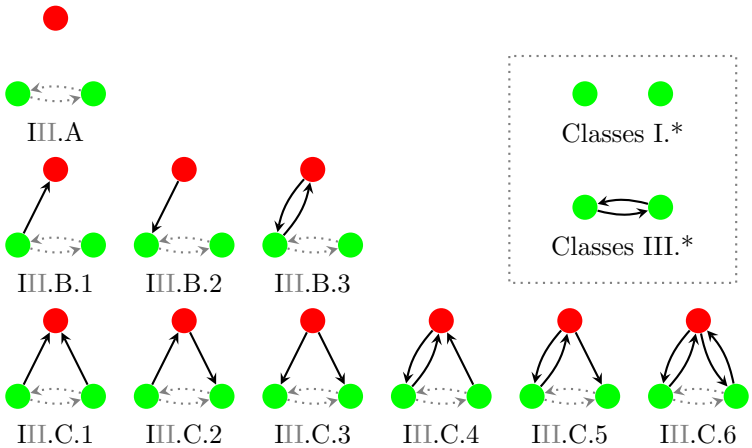
- **Signature:** `1,1`
- **Motif Info Object**: *`sma.DiMotif11Info`*



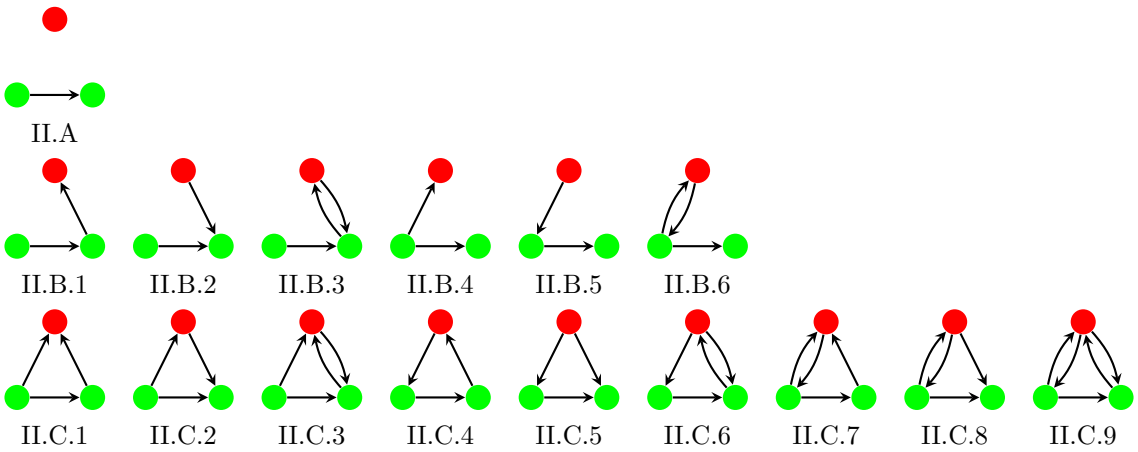
(1,2)-motifs

- **Signature:** `1,2`
- **Motif Info Object**: *`sma.DiMotif12Info`*

Directed (1,2)-motifs consist of three nodes, of which two are located on the same level. The $2^6 = 64$ possible directed graphs on three nodes fall into 36 classes which are listed below. The nomenclature follows the naming scheme for undirected *3-motifs*. When considering the underlying undirected graphs, class I.A corresponds to I.A, classes I.B.* to I.B, I.C.* to I.C, II.A to II.A, II.B.* to II.B, II.C.* to II.C and III.A to II.A, III.B.* to II.B, III.C.* to II.C.



The first figure shows the motifs of classes I.* (i.e., I.A to I.C.6) and III.* (i.e., III.A to III.C.6). Motifs of classes I.* have no edges between the two green vertices on the non-distinct level. Motifs of classes III.* have both possible vertices on this level.



The second figure illustrates the remaining motifs of classes II.*. These motifs have only one of the two possible edges between the two green vertices.

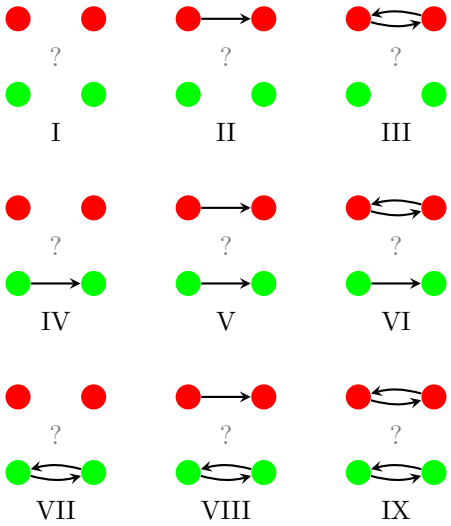
Note that the motifs of classes II.* occur exactly twice among the 64 possible directed graphs on three nodes while in I.* and III.* only the motifs of classes I.B.1, I.B.2, I.B.3, I.C.2, I.C.4, I.C.5 (and respectively III.*) occur twice. All other motifs have exactly one representative.

(2,2)-motifs

- **Signature:** `2,2`
- **Motif Info Object**: *`sma.DiMotif22Info`*

Directed (2,2)-motifs consist of four nodes, of which two are located on the upper level and two are located on the lower level. The $2^{12} = 4069$ possible directed graphs with such vertices mostly admit only few symmetries. This results in a rather complicated classification comprising 960 classes of motifs.

At a first glance, there are nine main groups of such motifs. These are illustrated below:

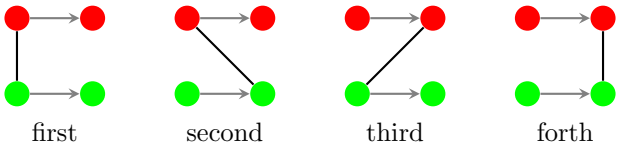


The question marks indicate that within the nine groups every possible configuration of edges spanning across the two levels occurs. Each main group comes with a different classification for its motifs. The class names are of the form *MAIN.SUB*. Where *MAIN* is one of the main groups I, II, III, IV, V, VI, VII, VIII, IX. The possible values of *SUB* depend on the main groups and are stated in the following subsections.

In the following figures the edges which are determined by the main group are drawn in gray. The black edges are the ones which affect the classification.

Asymmetric group V

In this group the edges on the top and on the bottom level induce a canonical order of the nodes (and hence of the edges). There are $2^8 = 256$ motif classes in this group. The (underlying undirected) inter-level edges are numbered as follows:



The names of the classes are the 256 length four strings on letters *n*, *d*, *u* and *b*. Where *n* stands for "none", *d* for "down", *u* for "up" and *b* for "both". According to the order of the edges given above, each edge is assigned one of the letters (*n* if its is not present, *d* if it is oriented downwards, *u* if upwards and *b* if both edges, upwards and downwards, occur in the motif).

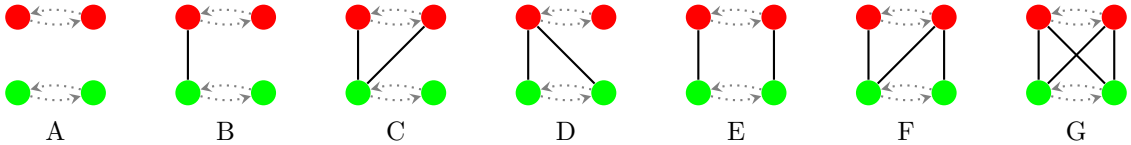
Simply-symmetric groups II, IV, VI and VIII

In these groups a single edge on one of the levels induces an order of its end nodes (tail and head). For the purpose of this section this level is called the yellow level while the other level is called the blue level. The nodes on the blue level are not ordered. It remains to classify edges in two stars, the tail star and the head star (see below). The tail (head resp.) star consists of the edges between the tail (head resp.) and the nodes on the blue level.

The names of the classes follow the nomenclature for directed *(1,2)-motifs*. The two stars are classified separately as (1,2)-motifs of classes I.* or III.* without considering the edges on the opposing (blue) level. This yields class names A to C.6 (forgetting about the I or III since these stem from the edges on the blue level). The class of the motif is then the concatenation of the class of the tail star and the class of the head star omitting all full stops. This results in 100 classes AA, AB1, ..., AC5, AC6, B1A, ..., B1C6, ..., C6A, ..., C6C6.

Double-symmetric groups I, III, VII and IX

In these groups the edges on the top and on the bottom level do not induce an orientation. Each group is subdivided into the following subgroups according to the underlying undirected bipartite graphs of the motifs. Class names are composed of the name of the subgroup (see figure) and of specifiers depending on the subgroup. These specifiers are defined below (cf. second component of nomenclature for *4-motifs*).

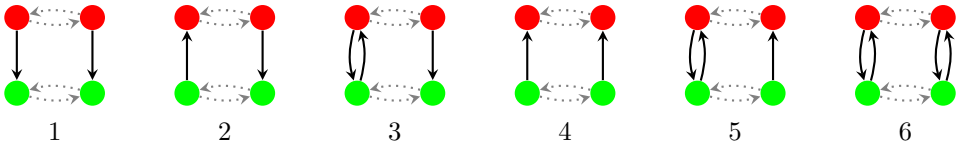


In **subcase A**, the classification is complete. This yields a single motif class.

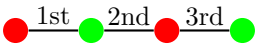
In **subcase B**, the classification is completed by invoking the classificator for directed *2-motifs*. This yields 3 motif classes (1, 2, 3).

In **subcases C and D**, the classification is completed by classifying the directed star as in *Simply-symmetric groups II, IV, VI and VIII*. Note that the classes A to B3 do not occur. This yields for each subcase 6 motif classes (1 to 6 abbreviating C1 to C6).

In **subcase E**, the following classification is used. There are 6 motif classes called 1, . . . , 6.



In **subcase F**, the task is to classify a path (i.e. a path in the underlying undirected graph) starting in one of the upper nodes and ending after 3 links in one of the lower nodes. When unfolded, this path looks as follows. The possible edges inherit canonical indices from the order of the nodes on the path. For the purpose of this classification the direction from left most red node to the right most green node is called *downwards*.

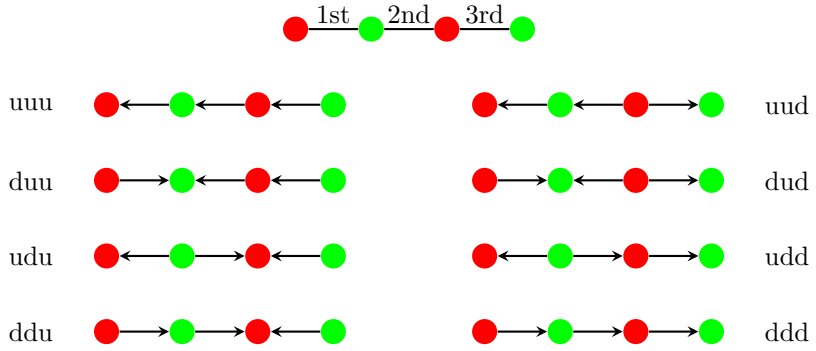


The class names are obtained by assigning to each of the nodes in the given order one of the letters *d*, *u* or *b*; *d* when the edge is oriented downwards, *u* when it is oriented upwards or *b* when both edges exist in the motif. This yields classes ddd to bbb (27 in total), cf. the following figure which depicts a similar classification.

Be aware that the directions used here might be confusing. When the first edge is oriented downwards it indeed links an upper node to a lower node. But when the second is oriented downwards it links a lower node to an upper node.

In **subcase G**, the underlying undirected bipartite graph is complete. Considering another undirected bipartite graph, this subcase is divided into 7 further subcases. More precisely, the classification for undirected bipartite graph in classes A to G which is illustrated at the beginning of this section is applied to the underlying undirected graph (the *double graph*) of the graph obtained from the motif by removing all directed edges which do not occur together with the edge in inverse direction. These subsubclasses are called Ga to Gg where the lower case letters are stemming from the classification of the double graph.

- In **subsubcase Ga**, the bipartite subgraph of the motif is a tournament, i.e. no edge occurs together with its counterpart in inverse direction. Here the classification of undirected bipartite graphs from the beginning of the section is applied again to the *downward graph*, i.e. the undirected bipartite graph obtained from the motif by considering only the edges in downward direction (and forgetting about their direction). This yields classes A to G (7 in total).
- In **subsubcase Gb**, the following classification (cf. subcase F) is applied to the directed path obtained from the motif by removing the two edges that correspond to the unique edge in the double graph. It remains to consider the directions of the three edges on the path. To each of the edges one of the letters *u* (upwards) or *d* (downwards) is assigned. This yields 8 possible classes called uuu to ddd.



- In **subsubcases Gc and Gd**, the directed star (cf. *Simply-symmetric groups II, IV, VI and VIII*) which remains when removing the four edges that correspond to the two edges in the double graph has to be classified. This is done as above. The only occurring classes are C1, C2 and C3, which are abbreviated as 1, 2 and 3 (3 classes in total).
- In **subsubcase Ge**, the pair of two parallel directed edges remaining when removing the four edges that correspond to the two edges in the double graph has to be classified. This is done as in subcase E. Note that the only possible classes are 1, 2 and 4 (3 classes in total).

- In **subsubcase Gf**, the edge classification (cf. subcase B, *2-motifs*) is applied to the remaining edges when removing the six edges corresponding to the three edges in the double graph. This yields classes 1, 2 (2 in total).
- In **subsubcase Gg**, the classification is complete (1 class in total). Nothing is added to the class name.

In total, case G comprises 27 classes. The class names from the subclasses are concatenated to the specifiers Ga to Gg separated by a full stop.

In total, the double-symmetric groups I, III, VII, IX each comprise 76 classes.

Remarks

As indicated above, the $2^{12} = 4069$ directed graphs on four vertices on two levels fall into 984 motif classes. As a sanity check, the number of representatives in each class can be counted:

- The *Asymmetric group V* comprises 256 motif classes each with 4 representatives depending on the orientation of the edge on the upper and the edge on the lower level. In total there are hence 1024 representatives.
- The four *Simply-symmetric groups II, IV, VI and VIII* each comprise 100 motif classes. There are 16 stars (falling into 9 classes) and hence $2 \cdot 16^2 = 512$ representatives in each group where the factor 2 stems from the orientation of the distinct intra-level edge. In total, 2048 representatives fall into these groups.
- The four *Double-symmetric groups I, III, VII and IX* each comprise 76 motif classes.
  - In subcase A, there is only 1 representative.
  - In subcase B, there are $4 \cdot 3 = 12$ representatives.
  - In subcase C, there are $2 \cdot 9 = 18$ representatives.
  - In subcase D, there are $2 \cdot 9 = 18$ representatives.
  - In subcase E, there are $2 \cdot 9 = 18$ representatives.
  - In subcase F, there are $4 \cdot 27 = 108$ representatives.
  - In subcase G, there are 81 representatives subdivided as follows:
    - In subsubcase Ga, there are $2^4 = 16$ representatives.
    - In subsubcase Gb, there are $4 \cdot 8 = 32$ representatives.
    - In subsubcase Gc, there are $2 \cdot 4 = 8$ representatives.
    - In subsubcase Gd, there are $2 \cdot 4 = 8$ representatives.
    - In subsubcase Ge, there are $2 \cdot 4 = 8$ representatives.
    - In subsubcase Gf, there are $2 \cdot 4 = 8$ representatives.
    - In subsubcase Gg, there is only 1 representatives.

  In total, there are 1024 representatives in these groups.

CHAPTER

ELEVEN

APPENDIX: PROBABILITIES IN ERDŐS-RÉNYI RANDOM GRAPHS

See also Distribution of motifs in Erdős-Rényi random graphs.

11.1 Expected densities

11.1.1 3-motifs

The following table lists the expected densities for 3-motifs. The distinct node lies in V_0, the two other nodes in V_1. Here, p_{11} denotes the probability that an edge between two nodes from V_1 exists. p_{01} is the probability that an edge between the two levels exists. Multiply all values by $|V_0|\binom{|V_1|}{2}$ to obtain the expected number of motifs. The function `sma.expected3Motifs()` uses the formulas in this table. It can be verified that the values sum to one.

Motif	Expected density
I.A	$(1-p_{11})(1-p_{01})^2$
I.B	$2(1-p_{11})p_{01}(1-p_{01})$
I.C	$(1-p_{11})p_{01}^2$
II.A	$p_{11}(1-p_{01})^2$
II.B	$2p_{11}p_{01}(1-p_{01})$
II.C	$p_{11}p_{01}^2$

11.1.2 4-motifs

The following table lists the expected densities for 4-motifs. The upper nodes lie in V_0, the other nodes in V_1. Here, p_{11} denotes the probability that an edge between two nodes from V_1 exists. p_{01} is the probability that an edge between the two levels exists, etc. Multiply all values by $\binom{|V_0|}{2}\binom{|V_1|}{2}$ to obtain the expected number of motifs. The function `sma.expected4Motifs()` uses the formulas in this table. It can be verified that the values sum to one.

Motif	Expected density
I.A	$2(1-p_{00})p_{01}^2(1-p_{01})^2(1-p_{11})$
I.B	$2p_{00}p_{01}^2(1-p_{01})^2(1-p_{11})$
I.C	$2(1-p_{00})p_{01}^2(1-p_{01})^2p_{11}$
I.D	$2p_{00}p_{01}^2(1-p_{01})^2p_{11}$
II.A	$2(1-p_{00})p_{01}^2(1-p_{01})^2(1-p_{11})$
II.B	$2p_{00}p_{01}^2(1-p_{01})^2(1-p_{11})$
II.C	$2(1-p_{00})p_{01}^2(1-p_{01})^2p_{11}$
II.D	$2p_{00}p_{01}^2(1-p_{01})^2p_{11}$
III.A	$(1-p_{00})p_{01}^4(1-p_{11})$
III.B	$p_{00}p_{01}^4(1-p_{11})$
III.C	$(1-p_{00})p_{01}^4p_{11}$
III.D	$p_{00}p_{01}^4p_{11}$
IV.A	$(1-p_{00})(1-p_{01})^4(1-p_{11})$
IV.B	$p_{00}(1-p_{01})^4(1-p_{11})$
IV.C	$(1-p_{00})(1-p_{01})^4p_{11}$
IV.D	$p_{00}(1-p_{01})^4p_{11}$
V.A	$4(1-p_{00})p_{01}^3(1-p_{01})(1-p_{11})$
V.B	$4p_{00}p_{01}^3(1-p_{01})(1-p_{11})$
V.C	$4(1-p_{00})p_{01}^3(1-p_{01})p_{11}$
V.D	$4p_{00}p_{01}^3(1-p_{01})p_{11}$
VI.A	$4p_{00}p_{01}(1-p_{01})^3(1-p_{11})$
VI.B	$4p_{00}p_{01}(1-p_{01})^3p_{11}$
VI.C	$2p_{00}p_{01}^2(1-p_{01})^2(1-p_{11})$
VI.D	$2p_{00}p_{01}^2(1-p_{01})^2p_{11}$
VII.A	$4(1-p_{00})p_{01}(1-p_{01})^3(1-p_{11})$
VII.B	$4(1-p_{00})p_{01}(1-p_{01})^3p_{11}$
VII.C	$2(1-p_{00})p_{01}^2(1-p_{01})^2(1-p_{11})$
VII.D	$2(1-p_{00})p_{01}^2(1-p_{01})^2p_{11}$

11.1.3 (1,1,1)-motifs

Let the three levels of a multi motif with arities 2, 2, 2 be indexed 0, 1, 2. Let p_{ij} for $0 \leq i \leq j \leq 2$ denote the probability that an edge linking levels i and j exists. Then the expected densities are:

Motif	Expected density
0	$(1-p_{01})(1-p_{12})(1-p_{02})$
1	$p_{01}(1-p_{12})(1-p_{02})$
2	$(1-p_{01})p_{12}(1-p_{02})$
3	$p_{01}p_{12}(1-p_{02})$
4	$(1-p_{01})(1-p_{12})p_{02}$
5	$p_{01}(1-p_{12})p_{02}$
6	$(1-p_{01})p_{12}p_{02}$
7	$p_{01}p_{12}p_{02}$

11.1.4 (1,2,1)-motifs

Let the three levels of a multi motif with arities 1, 2, 1 be indexed 0, 1, 2. Let p_{ij} for $0 \leq i \leq j \leq 2$ denote the probability that an edge linking levels i and j exists. Then the expected densities are:

Motif	Expected density
1	$2p_{01}(1-p_{01})(1-p_{11})p_{12}^2(1-p_{02})$
2	$p_{01}^2(1-p_{11})p_{12}^2(1-p_{02})$
3	$p_{01}^2(1-p_{11})p_{12}^2p_{02}$
4	$p_{01}^2p_{11}p_{12}^2p_{02}$
-1	remainder

The expected density for the unclassified motifs (class -1) equals one minus the sum of the other densities.

11.1.5 (2,2,1)-motifs

Let the three levels of a multi motif with arities 2, 2, 1 be indexed 0, 1, 2. Let p_{ij} for $0 \leq i \leq j \leq 2$ denote the probability that an edge linking levels i and j exists. Such a motif consists of a 4-motif in the upper part extended by a lower node. The classification of such motifs relies on the classification of 4-motifs. For this reason, the following table only provides factors. For obtaining expected densities one has to multiply them with the expected densities for 4-motifs listed in one of the preceding sections.

Motif	Factor
CLASS.0	$(1-p_{02})^2(1-p_{12})^2$
CLASS.1	$p_{02}^2(1-p_{12})^2$
CLASS.2	$(1-p_{02})^2p_{12}^2$
CLASS.3	$p_{02}^2p_{12}^2$
Unclassified	remainder

For example, the expected density for motifs of class III.A.3 is then $p_{02}^2p_{12}^2 \cdot (1-p_{00})p_{01}^4(1-p_{11})$. The expected density for the unclassified motifs equals one minus the sum of the other densities.

11.1.6 (2,2,2)-motifs

Let the three levels of a multi motif with arities 2, 2, 2 be indexed 0, 1, 2. Let p_{ij} for $0 \leq i \leq j \leq 2$ denote the probability that an edge linking levels i and j exists. Then the expected densities are:

Motif	Expected density
1	$2(1-p_{00})(1-p_{01})^2p_{01}^2(1-p_{12})^2(1-p_{11})p_{12}^2p_{22}(1-p_{02})^4$
2	$2p_{00}(1-p_{01})^2p_{01}^2(1-p_{12})^2(1-p_{11})p_{12}^2p_{22}(1-p_{02})^4$
3	$(1-p_{00})p_{01}^4p_{02}^4(1-p_{11})p_{12}^4p_{22}$
4	$p_{00}p_{01}^4p_{02}^4(1-p_{11})p_{12}^4p_{22}$
-1	remainder

The expected density for the unclassified motifs (class -1) equals one minus the sum of the other densities.

11.2 Second moments

Variances are computed by determining second moments and expectation. Therefore this section focuses on second moments. As described in *Distribution of motifs in Erdős-Rényi random graphs*, we need to compute a sum over all choices of two motifs. These motifs can overlap in different ways giving different probabilities. The following sections describe how the sums are split and which probabilities the different cases give.

11.2.1 3-motifs

Overlap	all nodes	one from each level	two non-distinct nodes
Factor	$\lvert V_0\rvert\binom{\lvert V_1\rvert}{2}$	$\lvert V_0\rvert(\lvert V_0\rvert-1)\lvert V_1\rvert(\lvert V_1\rvert-1)$	$\binom{\lvert V_1\rvert}{2}\lvert V_0\rvert(\lvert V_0\rvert-1)$
I.A	$(1-p_{01})^2(1-p_{11})$	$(1-p_{01})^3(1-p_{11})^2$	$(1-p_{01})^4(1-p_{11})$
I.B	$2(1-p_{01})(1-p_{11})p_{01}$	$(1-p_{11})^2((1-p_{01})^2p_{01}+p_{01}^2(1-p_{01}))$	$4(1-p_{01})^2p_{01}^2(1-p_{11})$
I.C	$p_{01}^2(1-p_{11})$	$p_{01}^3(1-p_{11})^2$	$p_{01}^4(1-p_{11})$
II.A	$(1-p_{01})^2p_{11}$	$(1-p_{01})^3p_{11}^2$	$(1-p_{01})^4p_{11}$
II.B	$2(1-p_{01})p_{11}p_{01}$	$p_{11}^2((1-p_{01})^2p_{01}+p_{01}^2(1-p_{01}))$	$4(1-p_{01})^2p_{01}^2p_{11}$
II.C	$p_{01}^2p_{11}$	$p_{01}^3p_{11}^2$	$p_{01}^4p_{11}$

Overlap	one non-distinct node	one distinct node	none
Factor	$\lvert V_0\rvert(\lvert V_0\rvert-1)\lvert V_1\rvert(\lvert V_1\rvert-1)(\lvert V_1\rvert-2)$	$\lvert V_0\rvert\binom{\lvert V_1\rvert}{2}\binom{\lvert V_1\rvert-2}{2}$	$\lvert V_0\rvert(\lvert V_0\rvert-1)\binom{\lvert V_1\rvert}{2}\binom{\lvert V_1\rvert-2}{2}$
I.A	$(1-p_{01})^4(1-p_{11})^2$	$(1-p_{01})^4(1-p_{11})^2$	$(1-p_{01})^4(1-p_{11})^2$
I.B	$4(1-p_{01})^2p_{01}^2(1-p_{11})^2$	$4(1-p_{01})^2p_{01}^2(1-p_{11})^2$	$4(1-p_{01})^2p_{01}^2(1-p_{11})^2$
I.C	$p_{01}^4(1-p_{11})^2$	$p_{01}^4(1-p_{11})^2$	$p_{01}^4(1-p_{11})^2$
II.A	$(1-p_{01})^4p_{11}^2$	$(1-p_{01})^4p_{11}^2$	$(1-p_{01})^4p_{11}^2$
II.B	$4(1-p_{01})^2p_{01}^2p_{11}^2$	$4(1-p_{01})^2p_{01}^2p_{11}^2$	$4(1-p_{01})^2p_{01}^2p_{11}^2$
II.C	$p_{01}^4p_{11}^2$	$p_{01}^4p_{11}^2$	$p_{01}^4p_{11}^2$

The second moments for a class of motifs can be computed by summing the products of the factors and the corresponding probabilities.

The formulas have been computed with great care and can be reproduced by considering all possibilities for the overlap of two 3-motifs. The following combinatorial observations can be made given this table:

- The factors sum to $\lvert V_0\rvert^2\binom{\lvert V_1\rvert}{2}^2$, the square of the total number of 3-motifs.

- The first columns and the columns in the bottom row are closely related to the formulas for the expectation of 3-motifs, cf. *Expected densities*. The first column equals the formula for the expectation while the columns in the bottom row equal their squares. This is because in these scenarios the overlapping subgraph does not contain any edges.
- The first column sums to one.
- The second column sums to $(2p_{01}^2 - 2p_{01} + 1)(2p_{11}^2 - 2p_{11} + 1)$, an expression symmetrical in p_{01} and p_{11} and minimal if $p_{01} = p_{11} = 1/2$. The expression $2p^2 - 2p + 1 = (1-p)^2 + p^2$ arises as the probability of two randomly with probability p chosen edges to be both present or both absent.
- The third column sums to $6p_{01}^4 - 12p_{01}^3 + 10p_{01}^2 - 4p_{01} + 1$ which equals $(1-p_{01})^4 + 4p_{01}^2(1-p_{01})^2 + p_{01}^4$. This is the probability that two 3-motifs overlapping in two non-distinct nodes are isomorphic. Node that this quantity does not depend on p_{11}.

11.2.2 4-motifs

Two 4-motifs can overlap in the following ways:

(a) all nodes, occurring $\binom{|V_0|}{2}\binom{|V_1|}{2}$ times,

(b) two upper nodes and one bottom node, occurring $\binom{|V_0|}{2}|V_1|(|V_1|-1)(|V_1|-2)$ times,

(c) one upper node and two bottom nodes, occurring $\binom{|V_1|}{2}|V_0|(|V_0|-1)(|V_0|-2)$ times,

(d) one upper node and one bottom node, occurring $|V_0||V_1|(|V_0|-1)(|V_1|-1)(|V_0|-2)(|V_1|-2)$ times,

(e) two upper nodes, occurring $\binom{|V_0|}{2}\binom{|V_1|-2}{2}\binom{|V_1|}{2}$ times,

(f) two bottom nodes, occurring $\binom{|V_1|}{2}\binom{|V_0|-2}{2}\binom{|V_0|}{2}$ times,

(g) one upper node, occurring $\binom{|V_1|}{2}\binom{|V_0|-2}{2}\binom{|V_0|}{2}$ times,

(h) one lower node, occurring $|V_1|\binom{|V_0|}{2}\binom{|V_0|-2}{2}(|V_1|-1)(|V_1|-2)$ times,

(i) no overlap, occurring $\binom{|V_0|}{2}\binom{|V_0|-2}{2}\binom{|V_1|}{2}\binom{|V_1|-2}{2}$ times.

These numbers sum to $\left(\binom{|V_0|}{2}\binom{|V_1|}{2}\right)^2$. The probabilities for the 28 possible 4-motifs can mostly be derived from their expectations, cf. *Expected densities*. In particular, the probabilities for (a) are the same as the expected densities. In (g), (h) and (i) none of the edges overlap, so the probabilities equal the squares of the expected densities. The probabilities for (e) and (f) can be computed by dividing the squares of the expected densities by the probability of the single edge that is counted twice. Hence, it remains to compute the probabilities for (b), (c) and (d). They are given in the following tables:

The values for overlap (b) and (c) are computed by multiplying the entries in the respective columns with the corresponding value in the column "Factor". This is because the only difference between (b) and (c) is that the

number of nodes and hence relevant edges on the top and the bottom level are interchanged.

Motif	Factor	Overlap (b)	Overlap (c)
I.A	$2(1-p_{01})^3 p_{01}^3$	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
I.B	$2(1-p_{01})^3 p_{01}^3$	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
I.C	$2(1-p_{01})^3 p_{01}^3$	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$
I.D	$2(1-p_{01})^3 p_{01}^3$	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
II.A	$(p_{01}^4(1-p_{01})^2 + p_{01}^2(1-p_{01})^4)$	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
II.B	$(p_{01}^4(1-p_{01})^2 + p_{01}^2(1-p_{01})^4)$	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
II.C	$(p_{01}^4(1-p_{01})^2 + p_{01}^2(1-p_{01})^4)$	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$
II.D	$(p_{01}^4(1-p_{01})^2 + p_{01}^2(1-p_{01})^4)$	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
III.A	p_{01}^6	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
III.B	p_{01}^6	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
III.C	p_{01}^6	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$
III.D	p_{01}^6	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
IV.A	$(1-p_{01})^6$	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
IV.B	$(1-p_{01})^6$	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
IV.C	$(1-p_{01})^6$	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$
IV.D	$(1-p_{01})^6$	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
V.A	$(2p_{01}^5(1-p_{01}) + 4p_{01}^4(1-p_{01})^2)$	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
V.B	$(2p_{01}^5(1-p_{01}) + 4p_{01}^4(1-p_{01})^2)$	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
V.C	$(2p_{01}^5(1-p_{01}) + 4p_{01}^4(1-p_{01})^2)$	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$
V.D	$(2p_{01}^5(1-p_{01}) + 4p_{01}^4(1-p_{01})^2)$	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
VI.A	$(4p_{01}^2(1-p_{01})^4 + 2p_{01}(1-p_{01})^5)$	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
VI.B	$(4p_{01}^2(1-p_{01})^4 + 2p_{01}(1-p_{01})^5)$	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
VI.C	$2p_{01}^3(1-p_{01})^3$	$p_{00}(1-p_{11})^2$	$p_{00}^2(1-p_{11})$
VI.D	$2p_{01}^3(1-p_{01})^3$	$p_{00}p_{11}^2$	$p_{00}^2 p_{11}$
VII.A	$(4p_{01}^2(1-p_{01})^4 + 2p_{01}(1-p_{01})^5)$	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
VII.B	$(4p_{01}^2(1-p_{01})^4 + 2p_{01}(1-p_{01})^5)$	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$
VII.C	$2p_{01}^3(1-p_{01})^3$	$(1-p_{00})(1-p_{11})^2$	$(1-p_{00})^2(1-p_{11})$
VII.D	$2p_{01}^3(1-p_{01})^3$	$(1-p_{00})p_{11}^2$	$(1-p_{00})^2 p_{11}$

Motif	Overlap (d)
I.A	$(1-p_{00})^2(1-p_{11})^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
I.B	$p_{00}^2(1-p_{11})^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
I.C	$(1-p_{00})^2p_{11}^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
I.D	$p_{00}^2p_{11}^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
II.A	$(1-p_{00})^2(1-p_{11})^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
II.B	$p_{00}^2(1-p_{11})^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
II.C	$(1-p_{00})^2p_{11}^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
II.D	$p_{00}^2p_{11}^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
III.A	$(1-p_{00})^2(1-p_{11})^2p_{01}^7$
III.B	$p_{00}^2(1-p_{11})^2p_{01}^7$
III.C	$(1-p_{00})^2p_{11}^2p_{01}^7$
III.D	$p_{00}^2p_{11}^2p_{01}^7$
IV.A	$(1-p_{00})^2(1-p_{11})^2(1-p_{01})^7$
IV.B	$p_{00}^2(1-p_{11})^2(1-p_{01})^7$
IV.C	$(1-p_{00})^2p_{11}^2(1-p_{01})^7$
IV.D	$p_{00}^2p_{11}^2(1-p_{01})^7$
V.A	$(1-p_{00})^2(1-p_{11})^2(9p_{01}^5(1-p_{01})^2+p_{01}^6(1-p_{01}))$
V.B	$p_{00}^2(1-p_{11})^2(9p_{01}^5(1-p_{01})^2+p_{01}^6(1-p_{01}))$
V.C	$(1-p_{00})^2p_{11}^2(9p_{01}^5(1-p_{01})^2+p_{01}^6(1-p_{01}))$
V.D	$p_{00}^2p_{11}^2(9p_{01}^5(1-p_{01})^2+p_{01}^6(1-p_{01}))$
VI.A	$p_{00}^2(1-p_{11})^2(9p_{01}^2(1-p_{01})^5+p_{01}(1-p_{01})^6)$
VI.B	$p_{00}^2p_{11}^2(9p_{01}^2(1-p_{01})^5+p_{01}(1-p_{01})^6)$
VI.C	$p_{00}^2(1-p_{11})^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
VI.D	$p_{00}^2p_{11}^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
VII.A	$(1-p_{00})^2(1-p_{11})^2(9p_{01}^2(1-p_{01})^5+p_{01}(1-p_{01})^6)$
VII.B	$(1-p_{00})^2p_{11}^2(9p_{01}^2(1-p_{01})^5+p_{01}(1-p_{01})^6)$
VII.C	$(1-p_{00})^2(1-p_{11})^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$
VII.D	$(1-p_{00})^2p_{11}^2(p_{01}^3(1-p_{01})^4+p_{01}^4(1-p_{01})^3)$

BIBLIOGRAPHY

[BN16] Ö. Bodin and D. Nohrstedt. Formation and performance of collaborative disaster management networks: Evidence from a Swedish wildfire response. *Global Environmental Change*, 41:183–194, November 2016. URL: https://linkinghub.elsevier.com/retrieve/pii/S0959378016303806, doi:10.1016/j.gloenvcha.2016.10.004.

[BT12] Örjan Bodin and Maria Tengö. Disentangling intangible social–ecological systems. *Global Environmental Change*, 22(2):430–439, May 2012. URL: https://linkinghub.elsevier.com/retrieve/pii/S0959378012000179, doi:10.1016/j.gloenvcha.2012.01.005.

[Ryd18] Robin Ryder. Distribution and Variance of Count of Triangles in Random Graph. April 2018. _eprint: https://stats.stackexchange.com/q/339767 Published: Cross Validated. URL: https://stats.stackexchange.com/q/339767.

[WRP09] Peng Wang, Garry Robins, and Philippa Pattison. PNet: program for the simulation and estimation of exponential random graph models. 2009. URL: https://www.melnet.org.au/pnet/.

INDEX

G

H

I

L

M

N

O

P

R

S

T

U

V