LAMFRIA Documentation

Release 1

Carlos Andres Delgado S

Oct 07, 2020

CONTENTS

1 Description **1**
- 1.1 Box Counting algorithm . . . 1
  - 1.1.1 BoxCounting algorithm: . . . 1
  - 1.1.2 Example . . . 2
- 1.2 Fixed Size Box Counting Algorithm . . . 2
  - 1.2.1 Box Counting Fixed Size module . . . 2
  - 1.2.2 Example . . . 3
- 1.3 CBBAlgorithm package . . . 3
  - 1.3.1 Submodules . . . 3
  - 1.3.2 CBBAlgorithm.CBBAlgorithm module . . . 3
  - 1.3.3 Module contents . . . 3
- 1.4 Genetic strategy . . . 3
  - 1.4.1 Genetic . . . 3
  - 1.4.2 Example . . . 6
- 1.5 SandBox Algorithm . . . 6
  - 1.5.1 SBAlgorithm . . . 6
  - 1.5.2 Example . . . 7
- 1.6 Simulated Annealing strategy . . . 7
  - 1.6.1 SimulatedAnnealing.SimulatedAnnealing module . . . 7
  - 1.6.2 Example . . . 9
- 1.7 robustness package . . . 9
  - 1.7.1 Submodules . . . 9
  - 1.7.2 robustness.robustness module . . . 9
  - 1.7.3 Module contents . . . 10
- 1.8 utils package . . . 10
  - 1.8.1 Submodules . . . 10
  - 1.8.2 utils.utils module . . . 10
  - 1.8.3 Module contents . . . 10

2 Requeriments **11**

3 Indices and tables **13**

Python Module Index **15**

Index **17**

CHAPTER

ONE

DESCRIPTION

LAMFRIA is a tool for multifractal and robustness analisys. It also includes two artificial intelligence strategies for calculate fractal dimensions.

Contents:

1.1 Box Counting algorithm

This package provides multifractal analysis with Box Counting Algorithm proposed in journal article: Fractal and multifractal properties of a family of fractal networks DOI: 10.1088/1742-5468/2014/02/P02020

1.1.1 BoxCounting algorithm:

BCAlgorithm.BCAlgorithm.**BCAlgorithm**(*g*, *minq*, *maxq*, *percentNodesT*, *centerNodes=array([]*, *dtype=float64)*)

Calculate fractal dimension with BoxCounting method

Inputs are parameters to configure algorithm behaviour.

Parameters

- **g** (*Snap PUN Graph.*) – Network.
- **minq** – Minimum value of q
- **minq** – Maximum value of q
- **percentNodesT** (*Integer*) – Number of combinations of center nodes. This value is a percent of the total nodes
- **CenterNodes** (*Numpy 1D Array*) – Calculated center. If this is null, then the centers are calculated

Returns

logR: Numpy array logarithm of r/d

Indexzero: Integer position of q=0 in Tq and Dq

Tq: Numpy array mass exponents

Dq: Numpy array fractal dimensions

lnMrq: Numpy 2D array logarithm of number of nodes in boxes by radio

1.1.2 Example

```
import sys
import lib.snap as snap
import BCAlgorithm.BCAlgorithm as BCAlgorithm
import numpy

minq = -10
maxq = 10
percentNodesT = 2 #200% of nodes
Rnd = snap.TRnd(1,0)
graph = snap.GenPrefAttach(10000, 10,Rnd)  #ScaleFree with 10 edges per node

logR, Indexzero,Tq, Dq, lnMrq = BCAlgorithm.BCAlgorithm(graph,minq,maxq,percentNodesT)
```

1.2 Fixed Size Box Counting Algorithm

This package provides multifractal analysis with Box Counting Fixed Size Algorithm proposed in journal article: Multifractal analysis of complex networks DOI: 10.1088/1674-1056/21/8/080504

1.2.1 Box Counting Fixed Size module

FSBCAlgorithm.FSBCAlgorithm.**FSBCAlgorithm**(*g*, *minq*, *maxq*, *percentNodesT*, *centerNodes=array([], dtype=float64)*)

Calculate fractal dimension with BoxCounting fixed size method

Inputs are parameters to configure algorithm behaviour.

Parameters

- **g** (*Snap PUN Graph.*) – Network.
- **minq** – Minimum value of q
- **minq** – Maximum value of q
- **percentNodesT** – Number of combinations of center nodes. This value is a percent of the total nodes
- **percentNodesT** – Center of boxes. All centers have to different and the lenght array must be equal to number of nodes

Returns

logR: Numpy array logarithm of r/d

Indexzero: Integer position of q=0 in Tq and Dq

Tq: Numpy array mass exponents

Dq: Numpy array fractal dimensions

lnMrq: Numpy 2D array logarithm of number of nodes in boxes by radio

1.2.2 Example

```
import sys
import lib.snap as snap
import FSBCAlgorithm.FSBCAlgorithm as FSBCAlgorithm
import numpy

minq = -10
maxq = 10
percentNodesT = 2 #200% of nodes
Rnd = snap.TRnd(1,0)
graph = snap.GenPrefAttach(10000, 10,Rnd)  #ScaleFree with 10 edges per node

logR, Indexzero,Tq, Dq, lnMrq = FSBCAlgorithm.FSBCAlgorithm(graph,minq,maxq,
↪percentNodesT)
```

1.3 CBBAlgorithm package

1.3.1 Submodules

1.3.2 CBBAlgorithm.CBBAlgorithm module

CBBAlgorithm.CBBAlgorithm.**CBBFractality**(*graph*)

CBBAlgorithm.CBBAlgorithm.**calculateLb**(*boxes*)

1.3.3 Module contents

1.4 Genetic strategy

Genetic algorithm for multifractal analysis

1.4.1 Genetic

Genetic.Genetic.**Genetic**(*g*, *minq*, *maxq*, *sizePopulation*, *iterations*, *percentCrossOver*, *percentMutation*, *degreeOfBoring*, *typeAlgorithm*)

The genetic algorithm

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **iterations** (*Integer*) – Number of max iterations
- **sizePopulation** (*Integer*) – Size of population
- **percentCrossOver** (*Double*) – Percent (0 to 1) of individual select to cross
- **percentMutation** (*Double*) – (0 to 1) of probability to apply mutation to an individual
- **degreeOfBoring** (*Integer*) – Number of iterations of boring
- **typeAlgorithm** (*String*) – Method for calculate fractal dimensions: 'SB' for Sandbox, 'BC' for BoxCounting, 'FSBC' for fixed size box counting

Returns

logR: Numpy array logarithm of r/d

Indexzero: Integer position of q=0 in Tq and Dq

Tq: Numpy array mass exponents

Dq: Numpy array fractal dimensions

lnMrq: Numpy 2D array logarithm of number of nodes in boxes by radio

fitNessAverage: Numpy array Average fitness across iterations

fitNessMax: Numpy array Maximum fitness across iterations

fitNessMin: Numpy array: Minimal fitness across iterations

Genetic.Genetic.**calculateCenters**(*graph*, *numNodes*, *iterations*, *sizePopulation*, *radius*, *distances*, *percentCrossOver*, *percentMutation*, *listDegree*, *maxDegree*, *degreeOfBoring*)

Calculate centers with a random size between 20% and 90% of number of nodes

The genetic algorithm:

1. Generate a poblation of ramdom nodes as centers of the boxes
2. Evaluate each set of nodes with fitness funtion
3. Categorize poblation according fitness
4. Select two indivudues into population, then create a new individual
5. Remove worst individuals
6. Repeat 2 to 5 ultil a number of iterations or a boring degree

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **numNodes** (*Integer*) – Number of nodes in the network.
- **sizePopulation** (*Integer*) – Size of population
- **radius** (*Integer*) – Diameter of the network
- **distances** (*Numpy 2D array of integers*) – Distance to other nodes
- **percentCrossOver** (*Double*) – Percent (0 to 1) of individual select to cross
- **percentMutation** (*Double*) – (0 to 1) of probability to apply mutation to an individual
- **listDegree** (*Numpy 1D Array*) – List of degree all nodes
- **maxDegree** (*Integer*) – Max degree in the network
- **degreeOfBoring** (*Integer*) – Number of iterations of boring

Returns

best: Numpy array Best individual, select centers of boxes

Indexzero: Integer position of q=0 in Tq and Dq

Tq: Numpy array mass exponents

Dq: Numpy array fractal dimensions

lnMrq: Numpy 2D array logarithm of number of nodes in boxes by radio

fitNessAverage: Numpy array Average fitness across iterations

fitNessMax: Numpy array Maximum fitness across iterations

fitNessMin: Numpy array: Minimal fitness across iterations

Genetic.Genetic.**calculateCentersFixedSize**(*graph*, *numNodes*, *iterations*, *sizePopulation*, *radius*, *distances*, *percentCrossOver*, *percentMutation*, *listDegree*, *maxDegree*, *sizeChromosome*, *degreeOfBoring*)

Calculate centers with a specified size

The genetic algorithm:

1. Generate a poblation of ramdom nodes as centers of the boxes
2. Evaluate each set of nodes with fitness funtion
3. Categorize poblation according fitness
4. Select two indivudues into population, then create a new individual
5. Remove worst individuals
6. Repeat 2 to 5 ultil a number of iterations or a boring degree

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **numNodes** (*Integer*) – Number of nodes in the network.
- **sizePopulation** (*Integer*) – Size of population
- **radius** (*Integer*) – Diameter of the network
- **distances** (*Numpy 2D array of integers*) – Distance to other nodes
- **percentCrossOver** (*Double*) – Percent (0 to 1) of individual select to cross
- **Percent** (*Double*) – (0 to 1) of probability to apply mutation to an individual
- **listDegree** (*Numpy 1D Array*) – List of degree all nodes
- **maxDegree** (*Integer*) – Max degree in the network
- **sizeChromosome** (*Integer*) – Number of centers of boxes
- **degreeOfBoring** (*Integer*) – Number of iterations of boring

Returns

best: Numpy array Best individual, select centers of boxes

Genetic.Genetic.**calculateFitness**(*graph*, *chromosome*, *radius*, *distances*, *listDegree*, *maxDegree*)

Calculate fitness from a select node centers in a network

Fitness is the average between distances of the centers and the average the degrees , the centers can be of different size

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **chromosome** (*Numpy array of integers*) – Centers
- **radius** (*Integer*) – Diameter of the network

- **distances** (*Numpy 2D array of integers*) – Distance between all nodes
- **listDegree** (*Numpy 1D Array*) – List of degree all nodes
- **maxDegree** (*Integer*) – Max degree in the network

Returns Fitness of the centers

Type Double

1.4.2 Example

```
import sys
import lib.snap as snap
import Genetic.Genetic as Genetic
import numpy

minq = -10
maxq = 10
sizePopulation = 200
percentCrossOver = 0.4
percentMutation = 0.05
degreeOfBoring = 20
Rnd = snap.TRnd(1,0)
graph = snap.GenPrefAttach(10000, 10,Rnd)  #ScaleFree with 10 edges per node

logR, Indexzero,Tq, Dq, lnMrq,fitNessAverage,fitNessMax,fitNessMin = Genetic.
↪Genetic(graph,minq,maxq,sizePopulation,iterations, percentCrossOver,␣
↪percentMutation,degreeOfBoring, 'SB')
```

1.5 SandBox Algorithm

This package provides multifractal analysis with SandBox Algorithm proposed in journal article: Determination of multifractal dimensions of complex networks by means of the sandbox algorithm DOI: 10.1063/1.4907557

1.5.1 SBAlgorithm

SBAlgorithm.SBAlgorithm.**SBAlgorithm**(*g*, *minq*, *maxq*, *percentSandBox*, *repetitions*, *centerNodes=array([], dtype=float64)*)

Calculate fractal dimension with SandBox method

Inputs are parameters to configure algorithm behaviour.

Parameters

- **g** (*Snap PUN Graph.*) – Network.
- **minq** – Minimum value of q
- **minq** – Maximum value of q
- **percentSandBox** (*Double*) – Number of combinations of center nodes. This value is a percent of the total nodes
- **repetitions** (*Integer*) – Number of repetitions of algorithm

- **CenterNodes** (*Numpy 1D Array*) – Calculated center. If this is null, then the centers are calculated

Returns

logR: Numpy array logarithm of r/d

Indexzero: Integer position of q=0 in Tq and Dq

Tq: Numpy array mass exponents

Dq: Numpy array fractal dimensions

lnMrq: Numpy 2D array logarithm of number of nodes in boxes by radio

1.5.2 Example

```
import sys
import lib.snap as snap
import SBAlgorithm.SBAlgorithm as SBAlgorithm
import numpy

minq = -10
maxq = 10
percentOfSandBoxes = 0.6
repetitionsSB = 50
Rnd = snap.TRnd(1,0)
graph = snap.GenPrefAttach(10000, 10,Rnd)  #ScaleFree with 10 edges per node

logRB, IndexzeroB,TqB, DqB, lnMrqB = SBAlgorithm.SBAlgorithm(graph,minq,maxq,
↪percentOfSandBoxes,repetitionsSB)
```

1.6 Simulated Annealing strategy

Simulated annealing algorithm for multifractal analysis

1.6.1 SimulatedAnnealing.SimulatedAnnealing module

SimulatedAnnealing.SimulatedAnnealing.**SA**(*g*, *minq*, *maxq*, *percentNodes*, *sizePopulation*, *Kmax*, *typeAlgorithm*)

The simulated annealing algorithm

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **sizePopulation** (*Integer*) – Size of population
- **Kmax** (*Integer*) – Initial temperature
- **typeAlgorithm** (*String*) – Method for calculate fractal dimensions: 'SB' for Sandbox, 'BC' for BoxCounting, 'FSBC' for fixed size box counting

Returns

logR: Numpy array logarithm of r/d

Indexzero: Integer position of q=0 in Tq and Dq

Tq: Numpy array mass exponents

Dq: Numpy array fractal dimensions

lnMrq: Numpy 2D array logarithm of number of nodes in boxes by radio

SimulatedAnnealing.SimulatedAnnealing.**calculateCenters**(*graph*, *numNodes*, *percentNodes*, *Kmax*, *d*, *distances*, *listID*, *listDegree*, *totalRemoved=0*)

Calculate centers with a specified size

The genetic algorithm:

1. Generate a poblation of ramdom nodes as centers of the boxes
2. Select a neighbor state
3. Select this state according its fitness and global temperature

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **numNodes** (*Integer*) – Number of nodes in the network.
- **percentNodes** (*Integer*) – Size of population
- **Kmax** (*Integer*) – System temperature
- **lisID** (*Numpy 2D array of integers*) – ID of nodes
- **listDegree** (*Numpy 2D array of integers*) – List of degree all nodes
- **totalRemoved** (*Integer*) – Number of nodes in solution, only if you want apply robusness analysis

Returns

currentState: Numpy array current individual, select centers of boxes

SimulatedAnnealing.SimulatedAnnealing.**calculateFitness**(*g*, *element*, *radius*, *distances*, *listID*, *listDegree*)

Calculate fitness from a select node centers in a network

Fitness is the average between distances of the centers and the average the degrees , the centers can be of different size

Parameters

- **graph** (*Snap PUN Graph.*) – Network.
- **chromosome** (*Numpy array of integers*) – Centers
- **radius** (*Integer*) – Diameter of the network
- **distances** (*Numpy 2D array of integers*) – Distance between all nodes
- **listDegree** (*Numpy 1D Array*) – List of degree all nodes
- **maxDegree** (*Integer*) – Max degree in the network

Returns Fitness of the centers

Type Double

SimulatedAnnealing.SimulatedAnnealing.**createNeighbors**(*node*, *numNodes*, *distances*)

Calculate fitness from a select node centers in a network

Fitness is the average between distances of the centers and the average the degrees , the centers can be of different size

Parameters

- **node** (*Integer.*) – ID of node in the network.
- **numNodes** (*Integer*) – Number of nodes
- **distances** (*Numpy 2D array of integers*) – Distance between all nodes

Returns neighbors. Numpy ID array

Type Double

1.6.2 Example

```
import sys
import lib.snap as snap
import SimulatedAnnealing.SimulatedAnnealing as SimulatedAnnealing
import numpy

minq = -10
maxq = 10
sizePopulation = 200
Kmax = 1500
Rnd = snap.TRnd(1,0)
graph = snap.GenPrefAttach(10000, 10,Rnd)  #ScaleFree with 10 edges per node

logRD, IndexzeroD,TqD, DqD, lnMrqD = SimulatedAnnealing.SA(graph,minq,maxq,
↪percentOfSandBoxes,sizePopulation, Kmax, 'BC')
```

1.7 robustness package

1.7.1 Submodules

1.7.2 robustness.robustness module

robustness.robustness.**robustness_analysis**(*graph*, *typeRemoval*, *minq*, *maxq*, *percentSandBox*, *repetitions*, *temperature=0*, *sizePopulation=0*, *iterationsGenetic=0*, *percentCrossOver=0*, *percentMutation=0*, *degreeOfBoring=0*, *percentOfNodes=0.1*, *initialPercent=0.1*, *finalPercent=1.0*, *iteracionPercent=0.1*, *nameFile='none'*)

1.7.3 Module contents

1.8 utils package

1.8.1 Submodules

1.8.2 utils.utils module

utils.utils.**copyGraph**(*graph*)

utils.utils.**getAdjacenceMatriz**(*distances*, *numNodes*)

utils.utils.**getAveragePathLength**(*graph*)

utils.utils.**getDistancesMatrix**(*graph*, *numNodes*, *listID*)

utils.utils.**getOrderedClosenessCentrality**(*graph*, *N*)

utils.utils.**getSizeOfGiantComponent**(*graph*)

utils.utils.**linealRegresssion**(*x*, *y*)

utils.utils.**removeNodes**(*graph*, *typeRemoval*, *p*, *numberNodesToRemove*, *ClosenessCentrality*, *listID*, *nodesToRemove=array([], dtype=float64)*)

1.8.3 Module contents

CHAPTER

TWO

REQUERIMENTS

- numpy >= 1.12.1
- snap >= 0.5
- matplotlib >= 2.0
- python = 2.7

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

BCAlgorithm.BCAlgorithm, 1

c

CBBAlgorithm, 3
CBBAlgorithm.CBBAlgorithm, 3

f

FSBCAlgorithm.FSBCAlgorithm, 2

g

Genetic.Genetic, 3

r

robustness, 10
robustness.robustness, 9

s

SBAlgorithm.SBAlgorithm, 6
SimulatedAnnealing.SimulatedAnnealing, 7

u

utils, 10
utils.utils, 10

INDEX

B

BCAlgorithm() (*in module BCAlgorithm.BCAlgorithm*), 1
BCAlgorithm.BCAlgorithm (*module*), 1

C

calculateCenters() (*in module Genetic.Genetic*), 4
calculateCenters() (*in module SimulatedAnnealing.SimulatedAnnealing*), 8
calculateCentersFixedSize() (*in module Genetic.Genetic*), 5
calculateFitness() (*in module Genetic.Genetic*), 5
calculateFitness() (*in module SimulatedAnnealing.SimulatedAnnealing*), 8
calculateLb() (*in module CBBAlgorithm.CBBAlgorithm*), 3
CBBAlgorithm (*module*), 3
CBBAlgorithm.CBBAlgorithm (*module*), 3
CBBFractality() (*in module CBBAlgorithm.CBBAlgorithm*), 3
copyGraph() (*in module utils.utils*), 10
createNeighbors() (*in module SimulatedAnnealing.SimulatedAnnealing*), 8

F

FSBCAlgorithm() (*in module FSBCAlgorithm.FSBCAlgorithm*), 2
FSBCAlgorithm.FSBCAlgorithm (*module*), 2

G

Genetic() (*in module Genetic.Genetic*), 3
Genetic.Genetic (*module*), 3
getAdjacenceMatriz() (*in module utils.utils*), 10
getAveragePathLength() (*in module utils.utils*), 10
getDistancesMatrix() (*in module utils.utils*), 10
getOrderedClosenessCentrality() (*in module utils.utils*), 10
getSizeOfGiantComponent() (*in module utils.utils*), 10

L

linealRegresssion() (*in module utils.utils*), 10

R

removeNodes() (*in module utils.utils*), 10
robustness (*module*), 10
robustness.robustness (*module*), 9
robustness_analysis() (*in module robustness.robustness*), 9

S

SA() (*in module SimulatedAnnealing.SimulatedAnnealing*), 7
SBAlgorithm() (*in module SBAlgorithm.SBAlgorithm*), 6
SBAlgorithm.SBAlgorithm (*module*), 6
SimulatedAnnealing.SimulatedAnnealing (*module*), 7

U

utils (*module*), 10
utils.utils (*module*), 10