Documentation for the boxing package

October 15, 2020

This documentation aims to introduce the basic functionality and notions of the `boxing` class. This was developed for comparing box covering algorithms that may characterize fractal properties of networks.

1 The network class

This class is used to host the network, alongside the node-to-node shortest path data. It is built upon the networkx package (for the graph).

1.1 Notable data members

- **self.shortest_paths**: returns node1:node2:shortest_path12 type dictionary
- **self.graph**: networkx.classes.graph.Graph
- **self.diameter**: nx.diameter(self.graph). Unfortunately, when the graph is unconnected, it cannot be computed this way. In that case, when calling **self.get_dist_dict**, the maximal geodesic length inside any component is computed and stored in **diameter**.
- **self.distance_dict**: the inverted version of self.shortest_paths, the output of **self.get_dist_dict**

1.2 important member functions

- **self.init**
  Constructor that awaits a networkx graph (networkx.classes.graph.Graph) as argument. Node labels are converted to integers starting from 0.
- **get_dist_dict**
  Computes the node-to-node shortest path. Awaits no arguments and returns a `{node:{distance:set_of_appropite_nodes}}` type dictionary, where the innermost value is a set object. Also, sets **self.shortest_path**.
- **invert_dict_list**
  The similar functionality as get_dist_dict but instead of sets. we have lists at the innermost. Awaits a dictionary {*key:value*} and inverts it: {*value:[list_of_keys]*}.
  static method
- **dual_network**
  Creates the dual network (connections between the ones that do not fit in one box). Takes l_b as an argument, returns the dual graph as a networkx *graph.*

  The convention used for l_B is different from usual (one box: l_B at most.)

  If **self.distance_dict** is not present, computes it.
- **ball_of_seed**
  Takes *(seed,r_B)* and determines nodes that are separated from seed r_B at most. Returns them in a set. If **self.distance_dict** is not present, computes it.

2 Miscellaneous functions

2.1 The io module

This module is built on top of the implemented algorithms, purpose is to manage boxing with calling the appropriate algorithm and logging results into files.

In addition to this, reading such files is supported too.

- **run_boxing(names,time_offset,network,box_sizes,algorithm,merge_alg=False,**kwargs)**

  This function is a generic boxing method. It performs boxing with the passed *algorithm*, on the given *network* (instance of the network class) with passed *box_sizes* (iterable).

  It is assumed that the necessary preprocessing (computing shortest path data) is already performed, its time is passed in *time_offset*. The *names* dictionary contains info for naming and filling the logfile, in the format of {*'path':'~/results/','net':'ecoli','alg':'greedy'*}. Using this, the file will be saved as *names['path']+names['net']+'_'+names['alg']+'.txt'*.

  Running the algorithms goes following way: besides the network and box sizes, all miscellaneous parameters are passed as keyword arguments. For the *merge* algorithm, *only the maximal box size* (l_b) should be passed and it is expected that *measure_time* is set False. For all other algorithms, boxing is performed for all passed *box_sizes*. An important stopping criterion is that if N_b=1, tha process terminates. After running boxing, the garbage collector is called. And the runtime of boxing is logged.

  After all of this, results are saved into a .txt file as specified above.

- **benchmark(names,time_offset,network,box_sizes,algorithm,n,merge_alg=False,**kwargs)**

  Works similarly as *run_boxing* but performs boxing *n* times for every box size. (Obviously, this function is meant for benchmarking.) Logs results into *names['path']+names['net']+'_'+names['alg']+ '_benchmark.txt'*

  Dumps the arrays of N_b-s for a given box size with default numpy method and ',' separator.

- **read_logfile(path)**

  Reading back of files logged by *run_boxing*. Returns the execution time and a list of (l_b,N_b) tuples.

- **read_logfile_bench(path)**

  This works exactly the same manner but for benchmark logs.

2.2 The load module

This module consists of:

- **read_from_edgelist(edge_file,header=0)**

  Reads unweighted, undirected graph from edgelist. Skips *header* rows from the input file and then awaits the (integer) ID of nodes that are connected, separated by any whitespace chars.

  Returns *networkx graph*.

- **read_max_connected_component(path,header_length=0)**

  Reads unweighted and undirected graph from file with *header_length* long header rows (skipped). Returns the maximal connected component as *networkx* graph.

2.3 The boxing module

This module only contains the **boxing_(network, net, alg_dict, box_sizes, path, preprocessing='distance_dict', benchmark_n=-1)** function.

As the name implies, this is the ultimate wrapper function for boxing, with pretty straightforward (usual) arguments. Note that *preprocessing* may take values 'distance_dict', 'shortest_paths' or anything else if no preprocessing is needed.

The random seed is set before *benchmark* or *run_boxing*

3 Investigated networks

In the course of the investigations, we both tested algorithms on network models and real-world networks. They are introduced in few lines each.

3.1 (u,v) flower

The (u,v) flower is a graph parametrized by {u,v,n} [1]. It is best understood considering its creation: we start with a circular graph of u+v nodes as the first (g=1) generation. In every generation, we replace all existing edges, with a u and v long path each.[1] This process is repeated until generation n (g=n).

Arguments: *(u, v, n)*.

3.2 SHM network

Introduced by Song, Havlin and Makse, the growth of this network is controlled by a continous e parameter. The idea is to reverse the renormalization process in some sense. In every generation, $k \cdot m$ new nodes are linked to each previously existing one with k being the respective degree. Between any two linked 'parent' nodes, the edge is removed with *1-e* probability and x new edges are formed between their 'children' nodes. The process is repeated until the desired generation number is reached. Generation #0 starts with two connected nodes.

Arguments: *(generation, m, x, e)*.
Evolution starts from generation #0 and after this, *generation* steps are performed.

3.3 HADGM

This network model is the slight modification of the SHM model [2]. There are two points of modification.
First, the authors argue that instead of implementing repulsion between all the parent nodes in the growth step, it may be beneficial to partition them into two groups: hubs and the others. In the repulsion step, different e_i probabilities apply: hub-hub links have a higher chance of survival than others. More precisely, a parent is hub if satisfies: $k_i(t-1)/k_{max}(t-1) > T$, where *(t-1)* signals degrees before the child assignment and k_{max} is the maximal degree of any node before the child assignment. So then hub-hub links die out with probability (1-a) and others with (1-b) with a,b,T pre-defined parameters.[2] Another modification is that after deleting a parent link, only one new is built (instead of x).
Secondly, after completing the repulsion stage, links inside the boxes (so between children of the same parent) are added. The number of new edges in each box equals $k_i(t-1)$. Citing the paper: *'We randomly pick one offspring node in each box and link it to other* $\tilde{k}(t-1)$ *number of offspring nodes.'* (In the description, we denoted $\tilde{k}(t-1)$ by $k_i(t-1)$.)

Arguments:*(generations, m, a, b, t_cutoff)*.
Evolution starts from two connected nodes as generation #0 and then *generation* new generations follow. In the documentation, T stands for t_{cutoff}.

4 Implemented algorithms

The *boxes.algortihms* list contains the callable boxing algorithms.

4.1 Random sequential

Original idea from [3]. The algorithm uses the notion of burning: in every step, an unburned node is randomly chosen[3] and we sort unburned nodes to this center that are not farther than r_B.

[1] So then u+v-2 new nodes are added per edge.

[2] To be consistent with the above arguments, $a > b$ is chosen.

[3] Note that in the original paper, all the nodes participated in the random selection! The authors argue that in some cases, that was necessary to obtain the desired behavior.

Arguments: *(network,rb,boxing=False)*
Here, *network* is an instance of the **network** class, *rb* is the radius and *boxing* determines if the box number or the boxes are returned.
If *network.distance_dict* not computed previously, the network.get_dist_dict function is ran (gives notification).

4.2 Bounding random sequential

This algorithm is based on [4]. The algorithm is almost the same as the random sequential but here we burn clusters of radius l_B. This guarantees that the number of clusters at the end is not greater than the desired N_B.[4]

To run this algorithm, the we call the random sequential with argument $r_B = l_B$.

4.3 Greedy coloring

This family of algorithms uses the conceptual trick of mapping the boxing problem to the famous graph coloring. So then, the actual coloring algorithm is implemented in the **networkx** package. Besides that, we only do some postprocessing.

Arguments: *(network,lb,boxing=False,pso_position=False,strategy='random_sequential')*
Here, *network* is an instance of the **network** class, *lb* is the box size, *boxing* determines if the box number or the boxes are returned, *pso_position* is a flag for computing the node-color(=box) list. The coloring is done on the dual network of *network*, produced by the *dual_network* member function of the class. Only one option is returned, in the pso, boxes priority order.

For coloring strategies, see the **networkx** docs.

4.4 Compact Box Burning (CBB)

This algorithm uses the concept of burning nodes. The original idea was presented in [5]. The main point is to grow boxes such that we pick new nodes randomly from a set that contains all nodes that are not farther than l_B from any node already in the box. This guarantees that the box is compact[5] in the sense of the authors.

Arguments: *(network,lb,boxing=False)*
Here, *network* is an instance of the **network** class, *lb* is the box size and *boxing* determines if the box number or the boxes are returned.
If *network.distance_dict* not computed previously, the network.get_dist_dict function is ran (gives notification).

4.5 MEMB

This algorithm was also presented in [5]. Instead of the conventional box definition, this uses the notion of centered boxes: every box has a special node, a center. Boxes are constructed such that every node is in the r_B ball of its center. The implementation guarantees that all the boxes are connected[6]. Without diving into details, the algorithm tries to avoid 'hub-failures' by selecting first centers and then assigning the remaining nodes to them in a wise manner.

Arguments: *(network,rb,boxing=False)*
Here, *network* is an instance of the **network** class, *rb* is the ball radius and *boxing* determines if the box number or the boxes are returned.
If *network.distance_dict* not computed previously, the network.get_dist_dict function is ran (gives notification). (If *network.shortest_path* is absent, it is computed too.)

Remark: the code slightly differs from the one presented in the original paper. Again, the distance of r_B from the center is allowed in the package but not in the paper.

[4] Any pair of centers is farther than l_B so cannot be in the same box anyway.

[5] Roughly speaking we cannot add any more nodes.

[6] By the implementation, every node is connected to its center.

4.6 REMCC

Idea based on [6]. The authors think this algorithm to be a better version of MEMB. The paper is quite obscure on why this algorithm would be better than any other.[7] The algorithm greedily covers nodes. In every step, a new center is chosen from uncovered nodes such that the choice maximizes the 'f_point'. This is the novel metric in the paper, which is the excluded mass times the average shortest path to all other nodes. All nodes in the r_B ball of the center are covered then.

In the current implementation, no boxing scheme is introduced, only the centers are determined. The method can be extended to weighted graphs

Arguments: *(network,rb,centres=False)*
Here, *network* is an instance of the **network** class, *rb* is the ball radius and *centres* determines if the centers or the box number is returned.

If **self.distance_dict** is not present, computes it.

4.7 MCWR

This algorithm comes from [7]. In fact, it is a combination of MEMB and the random sequential (RS) algorithm. In addition, a new way of keeping track of excluded mass is proposed.

The core concept is that we modify MEMB such that before choosing a new center, we toss a biased coin so that we perform the usual MEMB steps with p probability and choose a random center in the remaining cases.[8] After finding a new center, the excluded masses are updated: only nodes inside the newly covered's r_B ball are affected. (This is the 'novel scheme' for excluded mass.)

The boxing of non-center nodes is organized as in MEMB.

Arguments: *(network,rb,p=1,boxing=False)*
Here, *network* is an instance of the **network** class, *rb* is the ball radius, p is the coin parameter[9] and *boxing* determines if the boxes or the box number is returned.

If *network.distance_dict* not computed previously, the network.get_dist_dict function is ran (gives notification). (If *network.shortest_path* is absent, it is computed too.)

```
   remark: p is a hyperparameter that may need tuning!
remark2: wouldn't it be wiser to do the random choice at specific parts (end)?
```

4.8 Merge algorithm

Idea from [8]. For a given box size, we try to merge clusters inherited from a smaller box size. We do this until no more clusters can be merged.[10] The number of clusters is $N_B(l_B)$. Then, box size is incremented and the whole procedure is repeated. The code uses the **merge_if_possible** module.

Arguments: *(network, lb_max, return_for_sa=False, boxing=False,measure_time=False)*
Here, *network* is an instance of the **network** class, *lb_max* is the maximal l_B,[11], *return_for_sa, boxing* are two mutually exclusive ways the algorithm can return values: the SA option gives C (after the maximal l_B is reached) whereas boxing gives the numbers of boxes for every intermediate l_B. If *measure_time* is true, boxing also returns the runtime for every l_B. If neither true, a list of final clusters (represented as lists) is returned.[12]

Note that the returned list's first element corresponds to $l_B = 0$!

4.8.1 Merge_if_possible

Auxiliary function for the merge- and simulated annealing algoriths. For a given C list, it returns list D, which is the merged version of C, containing clusters as sets after the merging procedure. (First,

[7] It is said that other methods 'break network connectivity', but the MEMB run in the example is not carried out carefully. Personally, I do not find the paper credible.

[8] Here, the choice is made such that covered nodes but the ones with $m_{ex} = 0$ are allowed

[9] The default setting correspond to MEMB.

[10] After merging, both participants are 'removed' from the considered clusters

[11] l_B starts from 1

[12] list(map(list, c))

a random member, c_k is chosen, then all mergeable c_js are accumulated from whose a random one is merged with c_k. The new cluster is stored in D while the parents are removed from C. Cycle goes on until C is empty.)

```
   note that the merged ones are thrown away, may not appear with the same lB
so then if max_lB << log N then the results may be practically unusable![13]
```

```
   remark: in the current implementation, choosing the to-be-merged pairs is
 rather clumsy: almost the whole C' is thrown away
```

Arguments: *(c,lb,distance_dict)*
Here, c is C (as list), l_B the box size and *distance_dict* contains the shortest path data in the {*node1:*{*node2:distance12*}} format. Returns D as a list.

4.9 Simualted annealng

The idea from [8]. The algorithm is built upon the above **merging algorithms**. Basically, we try to improve the merged output by (i) moving single nodes, (ii) creating new clusters and (iii) merging clusters in the new cluster partitioning. After performing these steps in the above sequence, we test if the number of boxes decreased. When it did (or remained the same), we keep the new configuration. If increased, we keep the new configuraton with $exp\{-\frac{\Delta N_B}{T}\}$ probability , that is decreased in every iteration: $T' = T \cdot cc$.

Arguments: *(network, lb, k1=20, k2=2, k3=15, temp=0.6, cc=0.995).*
Here, *network* is an instance of the **network** class, *lb* is l_B, k_1 gives the approximate number of nodes moved in (i),[14] k_2 is the number of maximally created new clusters (made up of one node),[15] k_3 is the number of outer cycles in every iteration, the temperature is decreased as specified by *cc.*

4.10 Differential evolution

The idea of differential evolution was taken from [9]. It belongs to the family of evolutional optimization methods, originally proposed over a D dimensional space. This idea was used in [10]. Their proposal is to box the graph with sequential greedy coloring such that the coloring sequence is represented by an N dimensional vector.[16] Because the sequential greedy coloring algorithm is deterministic, finding the optimal box covering may be thought of as optimization over this N dimensional space. The optimization is done by the differential evolution approach.

Arguments: *(network, lb, num_p=15, big_f=0.9, cr=0.85, gn=15, boxing=False, dual_new=False).*
Here, *network* is an instance of the **network** class, *lb* is l_B, *num_p* gives the number of vectors in one generation, *big_f, cr* are parameters (see paper or code), *gn* gives the number of generations, *boxing* determines if the actual boxes or their number is returned and *dual_new* sets which greedy coloring function to use.[17]

4.11 PSO

The idea of Particle Swarm Optimization came from [11]. Basically, we define a set of 'paricles', representing a valid boxing of the network each. In every step, each particle may be updated depending on its and the whole flock's best boxing and a bunch of hyperparameters and random variables. In the end, the best overall best boxing is the outcome.

Particles are initialized with the greedy-coloring algorithm, velocities (see paper) start from zero.

Arguments: *pso(network, lb, gmax=5, pop=5, c1=1.494, c2=1.494, boxing=False).* Here, *network* is an instance of the **network** class, *lb* is l_B, *gmax* gives the number of generations, *pop* the population of the particle swarm, *c1, c2* are hyperparameters with the recommended value from the

[13] It may be that further merge operations could be performed but the elimination of once merged ones spoils the procedure so we reach the maximal l_B with 'unsaturated' boxes but the algorithm terminates.

[14] In fact, we try maximally $2k_1$ times to move nodes and actually do it k_1 times at most.

[15] We make maximally k_2 random choices of boxes from which a node is tried to be removed

[16] The ordering of components gives the ordering of nodes. Seems like a bit of an overkill...

[17] There are two, from which the new is thought to be much faster.

paper, *boxing* determines if the box number or the boxes themselves (list of lists) are returned.

```
   remark: smaller node IDs have priority in updates!
remark2: in the implementation XOR means not equal - makes more sense than bitwise XOR
```

4.12 OBCA

Original paper: [12]. The authors suggest that instead of burning boxes on the fly, one should only mark possible boxes while processing data. In a proposed box, all nodes are included whose distance from the seed of the box is at most l_B After this, redundant boxes - ones that only contain nodes that are at least in another proposed box - are deleted. After the iteration is over, only non-redundant boxes survive.

When creating proposals, nodes are iterated over in an ascending order wrt. degree. (The authors maintain that it is a good idea to choose seeds from the least-possible-degree nodes.)

Arguments: *overlapping_box_covering(network, lb, boxing=False).*

Here, *network* is an instance of the **network** class, *lb* is l_B and *boxing* determines if the box number or the boxes themselves are returned.

4.13 Fuzzy algorithm

Idea from [13]. The authors propose a novel scheme for estimating the fractal dimension of a network. Instead of assigning boxes, they introduce a measure to estimate what fraction of the network one box covers on average.

For each node, a box of radius l_B is constructed and the included neighbour nodes' contributions are summed. This contribution is a so called 'membership-function' that exponentially decays with the distance from the central node.

After aggregating and normalizing these contributions, we get an estimation of what proportion an average box covers. Taking its inverse gives the approximating box number.

Arguments: *fuzzy(network, lb, boxing=True).*

Here, *network* is an instance of the **network** class, *lb* is l_B and *boxing* should always set to **True**, since we do not have actual boxes.

References

[1] Hernán D Rozenfeld, Shlomo Havlin, and Daniel Ben-Avraham. Frac- tal and transfractal recursive scale-free nets. New Journal of Physics, 9(6):175, 2007.

[2] Kuang, Li & Zheng, Bojin & Li, Deyi & Li, Yuanxiang & Sun, Yu. (2013). A Fractal and Scale-free Model of Complex Networks with Hub Attraction Behaviors. Science China Information Sciences. 58. 10.1007/s11432-014-5115-7.

[3] Kim, Js & Goh, K-I & Kahng, B. & Kim, Doochul. (2007). A box-covering algorithm for fractal scaling in scale-free networks. Chaos (Woodbury, N.Y.). 17. 026116. 10.1063/1.2737827.

[4] C. Yuan, Z. Zhao, R. Li, M. Li and H. Zhang, "The Emergence of Scaling Law, Fractal Patterns and Small-World in Wireless Networks," in IEEE Access, vol. 5, pp. 3121-3130, 2017, doi: 10.1109/ACCESS.2017.2674021.

[5] Chaoming Song et alJ. Stat. Mech. (2007) P03006

[6] Zheng, Wei & Pan, Qian & Chen, Sun & Deng, Yu-Fan & Zhao, Xiao-Kang & Kang, Zhao. (2016). Fractal Analysis of Mobile Social Networks. Chinese Physics Letters. 33. 038901. 10.1088/0256-307X/33/3/038901.

[7] Liao, Hao & Wu, Xingtong & Wang, Bing & Wu, Xiangyang & Zhou, Mingyang. (2019). Solving the speed and accuracy of box-covering problem in complex networks. Physica A: Statistical Mechanics and its Applications. 523. 10.1016/j.physa.2019.04.242.

[8] Locci, Mario & Concas, Giulio & Tonelli, Roberto & Turnu, Ivana. (2010). Three Algorithms for Analyzing Fractal Software Networks. WSEAS Transactions on Information Science and Applications. 7.

[9] Storn, R., Price, K. Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces. Journal of Global Optimization 11, 341–359 (1997). https://doi.org/10.1023/A:1008202821328

[10] Kuang, Li. (2014). A differential evolution box-covering algorithm for fractal dimension on complex networks. 10.1109/CEC.2014.6900383.

[11] L. Kuang, F. Wang, Y. Li, H. Mao, M. Lin and F. Yu, "A discrete particle swarm optimization box-covering algorithm for fractal dimension on complex networks," 2015 IEEE Congress on Evolutionary Computation (CEC), Sendai, 2015, pp. 1396-1403, doi: 10.1109/CEC.2015.7257051.

[12] Sun, YuanYuan & Zhao, Yujie. (2014). Overlapping-box-covering method for the fractal dimension of complex networks. Physical Review E. 89. 10.1103/PhysRevE.89.042809.

[13] Haixin Zhang, Yong Hu, Xin Lan, Sankaran Mahadevan, Yong Deng, Fuzzy fractal dimension of complex networks, Applied Soft Computing, Volume 25, 2014, Pages 514-518, ISSN 1568-4946, https://doi.org/10.1016/j.asoc.2014.08.019.