pks

Release 1.0

lionel tabourier, julien karadayi

Mar 07, 2024

CONTENTS:

- pks (probabilistic k-swap) is a python package that provides a MCMC method to generate a uniform sample of graphs under certain constraints.

CHAPTER ONE

INSTALLATION

1.1 Requirements

- pip and setuptools. To upgrade to final version : pip install --upgrade pip setuptools wheel
- arch : pip install arch
- numpy
- argparse
- progressbar
- pytest
- joblib
- ipdb (not needed but useful for debugging)

1.2 Installation

- To install the package :

  ```
  pip install ./
  ```

- Check if package works as intended:

  ```
  pytest tests/test.py
  ```

CHAPTER

TWO

USAGE

- The **pks** package provides a program to generate random simple graphs uniformly with a given set of target constraints:
  - undirected graphs with a fixed degree sequence
  - directed graphs with a fixed degree sequence
  - bipartite graphs with a fixed degree sequence
  - directed graphs with a fixed degree sequence and a fixed number of mutual dyads
  - undirected graphs with a fixed joint degree matrix

2.1 Input Format

- The program takes as input a simple graph described as an edge list (one line corresponds to an edge, two nodes are seperated by a space or tabulation):

  ```
  1 2
  3 5
  6 5
  .
  .
  .
  ```

Note: This package does not handle loops and multiple edges. If the input graph contains either, they will be automatically removed when the graph is read.

2.2 Output

- The program generates a sample of graphs selected uniformly at random from the set of graphs with the same target constraints as the input graph.

2.3 Target Constraints

- The program generates uniformly at random **simple** graphs (no loop, no multiedge).
- Several target constraints are available:
  - fixed degree sequence (default): Can be applied on several "flavours" of simple graphs: undirected, directed, bipartite. The convergence can be evaluated either by following the assortativity (-a option) or the number of triangles (-t option).
  - fixed joint degree matrix: Can be applied to undirected, directed and bipartite graphs. The convergence can only be evaluated by following the number of triangles (-t), as the assortativity is constant.
  - fixed number of mutual dyads: Can only be applied to directed graphs. A mutual dyad occurs when the graph contain links in two directions between two nodes, this constraint fixes the total number of mutual dyads in the graph. The model convergence can be evaluated either by following the assortativity (-a option) or the number of triangles (-t option).

Note: A user can add to the code additional constraints by adding them to the MarkovChain class. The constraints should be added in the **check_swap** method, used to verify if an edge swap is valid or not. An argument to select this constraint can then be added to the list of arguments in the main.py file.

2.4 Command Line Interface

- Usage example, in the root folder of the package:

  python kedgeswap/main.py -f ./data/ucidata-zachary/out.ucidata-zachary -o ./karateclub.out -a

- list of main.py parameters:
  - Required arguments:
    - -f <path> : path to the input file.
    - -o <path> : path to the output files. Will write *N* output graphs with this prefix as filename, where *N* is fixed by the *--output_number* parameter (see Optional arguments).
    - -d : enable if the input graph is directed or bipartite.
    - -a : enable to follow the convergence of the Markov chain using the assortativity of the graph. Warning, option is not compatible with *-t* or *-jd*.
    - -t : enable to follow the convergence of the Markov chain using the number of triangles in the graph. Warning, option is not compatible with *-a*.
    - -jd : (target constraint argument) enable to generate sample of graphs with a fixed joint degree matrix. Warning: only works with *-t* option to follow convergence (assortativity is constant when joint degree matrix is fixed).
    - -md : (target constraint argument) enable to generate sample of graphs with a fixed number of mutual dyads. Warning: only works on directed graphs (*-d* option).
  - Optional arguments:
    - -v : enable to be more verbose. Adds the Markov Chain status to the logs, number of accepted/rejected swaps, DFGLS output to follow convergence.
    - -g <positive integer> : exponent of the probability law used to pick the number of edges to swap.

- -e <positive integer> : sampling gap between each generated graph. If not specified, will use a (slow) estimation method.
- --output_number <positive integer> : number *N* of uncorrelated graphs to generate once the Markov Chain has reached its convergence. Default is 1000.
- --njobs <positive integer> : number of threads on which the process is parallelized, if possible. Default is 4.
- --debug : makes some additional checks, like checking that the degree sequence hasn't changed after a swap (warning: slows down the code, only used for debugging purposes).
- --keep_record : enable to store every step (as gzip file) of the Markov chain, as well as every permutation (warning: produces a large number of files, mostly useful for debugging purposes).
- --log_dir : only useful if keep_record is enabled. Specify a path to store each step of the Markov Chain.

CHAPTER

THREE

DEVELOPMENT GUIDE

- The aim of this guide is to give a quick understanding of the structure of the package. If you want to modify or add content to the package, this page should help you!

3.1 Package Structure

- This package is made of three classes that interact with each other:
  - Graph: The Graph object reads the input graph and stores it in a data structure described below. The graph is *simple*, *without loop*, can be *undirected* or *directed*, and can be *bipartite*. This is the class to check to add new input format or to modify the data structure.
  - MarkovChain: The MarkovChain object selects the edges to swap, checks the constraints, performs the swap and measures some metrics on the graph (e.g., the assortativity value, the number of triangles). This is the class to check to add new constraints on the MarkovChain (e.g. fixed number of triangles) or to measure other values (number of a given motif).
  - Stat: The Stat object estimates the sampling gap of the Markov Chain, and check if the Markov Chain has converged. This is the class to check to adapt the sampling gap estimation, or to implement other methods to check for the Markov Chain convergence.
- You will find below some insights on the implementations of each class.

3.2 Graph

- The Graph object stores a *simple* graph. The data structure used for the graph is
  - for undirected, directed and bipartite graphs:
    - **M and N: int**
      Respectively store the number of edges and the number of nodes of the graph. Both remain constant during the swapping process.
    - **neighbors: dict(list)**
      Stores the adjacency list for each node of the graph. Given that the structure is a dictionary (hash map), getting the adjacency list of a node is in **O(1)**, amortized time. **neighbors** is used to store the neighbors of each node, and is updated at each swap.
    - **unique_edges: list**
      Stores one copy of each edge of the graph. **unique_edges** is used to uniformly pick random edges of the graph, and is updated at each swap.

        - **directed: bool**
          Indicates if the graph is directed or not.

    - for undirected graphs:

        - **edges: dict(int)**
          For each edge **(u,v)**, stores the position of **v** in the adjacency list of **u**. E.g.

```
v_idx = edges[(u,v)]
u_idx = edges[(v,u)]
neighbors[u][v_idx] == v # True
neighbors[v][u_idx] == u # True
```

    - for directed graphs:

        - **in_neighbors and out_neighbors: dict(list)**
          For each node **u** of the graph, respectively store the incoming neighbors and outgoing neighbors of **u**. This structure is used for example to check on mutual dyads and is updated at each swap.

        - **edges: dict(tuple)**
          For each edge **(u,v)**, it stores the position of **v** in **neighbors[u]**, in **out_neighbors[u]**, and the position of **u** in **neighbors[v]** and in **in_neighbors[v]**.

3.3 MarkovChain

- The main function of the MarkovChain object is the **run** method, that calls all the others depending on the input constraints. To add new constraints, update the **check_swap** method, that accepts or rejects a swap depending on the constraints.

    - **pick_k** : Chooses a **k** value following a power-law distribution with exponent **gamma**

    - **find_swap** : Uniformly chooses **k** edges to swap, and a random permutation.

    - **check_swap :**
      Checks that the chosen swap respects each constraint. The complexity depends on the constraints. For undirected graph, with fixed degree sequence constraint (the basic case), for each swap between **(u,v)** and **(x,y)**:

        - Checks that it doesn't create a loop (**u != y**) in **O(1)**.

        - Checks that swapped edge doesn't exist (**(u,y) not in graph.edges**) in **O(1)**.

        - If **k>2**, check that several permutations don't result in the same edge in **O(k)** (**len(goal_edges) == len(set(goal_edges))**, where **goal_edges** is the list of all the resulting edges if the swap is accepted).

    - **perform_swap :**
      Updates the Graph data structure depending on the swap. Each swap between two edges **(u,v)** and **(x,y)** is in amortized **O(1)** time (for undirected graphs - easily generalized to directed graphs):

        - Gets the position **v_idx = edges[(u,v)]** of **v** in **neighbors[u]** in amortized **O(1)** (edges is a hash map), and **x_idx = edges[(y,x)]**.

        - Updates the value **neighbors[u][v_idx] = y** in amortized **O(1)**.

        - Deletes **edges[(u,v)]** and **edges[(v,u)]** in amortized **O(1)**.

        - Adds **edges[(u,y)] = y** and **edges[(y,u)]=x_idx** in amortized **O(1)**.

- Other methods are implemented to measure some metrics on the graph. Each metric has an “init” and an “update” function, the “init” function computes the value for the input graph, while the “update” methods updates it after each swap without having to compute it again for edges that haven’t changed.

3.4 Stat

- The Stat class implements methods to estimate the sampling gap of the MarkovChain object and follows its convergence.
- The sampling gap is the number of required steps of the *stationary* Markov Chain between two samples to ensure that both samples are uncorrelated.
- Two methods are implemented to choose a sampling gap:
  - estimate_sampling_gap:
    - First runs a *burn-in* step to obtain a fully stationary Markov Chain.
    - Estimates the acceptation rate **A** of the Markov Chain during the burn-in.
    - Initializes the sample gap value at **eta = M/A ** where **M** is the number of edges in the graph and **A** the acceptation rate of the chain.
    - Starting from the same final step of the *burn-in*, runs **10** different chains for **500eta** steps, measuring **1** assortativity (or number of triangles) each **eta** step.
    - If the autocorellation at lag 1 of each time series of assortativities (or number of triangles) returns that at least **9** out of the **10** chains are not correlated, the sampling gap is considered valid.
      - If the sampling gap **eta** is valid, tries **eta=eta/2**.
      - If not, tries **eta=2eta**.
      - Stops as soon as the behaviour changes and returns the last valid eta value.
  - run_dfgls:
    - If no **eta** value is given in input, estimates an **eta** value.
    - While the MarkovChain has not converged:
      - Runs the MarkovChain object for **eta** steps and collects the assortativity (or number of triangles) at each step.
      - After **eta** steps, checks if the time series of assortativity values (or number of triangles) has converged using a DFGLS test.

3.5 Contribution

- To contribute to the package, you can put issues on the gitlab repository to either report a problem or to ask a question.
- Any pull request will be reviewed and integrated if the contribution is within the scope of the project.

CHAPTER

FOUR

BENCHMARK

- On this page, you will find a benchmark of *pks* on different datasets with various constraints, to give an idea of what to expect in terms of execution time and a better grasp of how to choose the sampling gap.

4.1 Datasets

- For this benchmark, we used the following datasets (we note n the number of nodes and m the number of edges):
  - karateclub: undirected simple graph: n=34, m=78, available here
  - lesmiserables: undirected simple graph: n=77, m=254, available here
  - powergrid: undirected simple graph, n=4941, m=6594, available here
  - yeast: directed simple graph, n=688, m=1079, available here
  - airtraffic: directed simple graph: n=1226, m=2615, available here
  - crime: bipartite graph, n=829+551, m=1476, available here

4.2 Protocol

- We generate samples of 1000 graphs.
- The parameter g is set to 2.
- We follow the convergence using the number of triangles of the graph except for the case of bipartite graphs, where we use the assortativity of the graph.
- An algorithm adapted from [1] estimates the sampling gap.
- The datasets are tested with different constraints:
  - Fixed degree sequence : the k-swaps do not affect the degree sequence of the graph.
  - Fixed joint degree matrix : the k-swaps do not affect the joint degree matrix of the graph. Note that it implies that the degree sequence is also fixed.
  - Fixed degree sequence and number of mutual dyads : the k-swaps do not affect the degree sequence of the graph and the number of reciprocal links of the graph (directed graphs only).

4.3 Results

dataset	constraint	eta value	eta estimation runtime (in seconds)	convergence runtime (in seconds)	total runtime (in seconds)
karateclub	degree sequence	751	377	2	1575
lesmiserables	degree sequence	1781	1203	5	4955
powergrid	degree sequence	16137	50995	52	233528
yeast	degree sequence	3394	5318	4	25997
airtraffic	degree sequence	6719	9063	19	37817
crime	degree sequence	3869	3026	38	13885
karateclub	joint degree matrix	5053	2 567	8s	4184
lesmiserables	joint degree matrix	24643	15987	43	26398
powergrid	joint degree matrix	197180	1644229	6023	2001551
yeast	degree sequence + dyads	3426	5719	103	27386
airtraffic	degree sequence + dyads	12332	123153	26	610772

4.4 References

[1] Dutta, U., Fosdick, B.K., & Clauset, A. (2021). Sampling random graphs with specified degree sequences. arXiv preprint arXiv:2105.12120.

CHAPTER

FIVE

PYTHON API DOCUMENTATION

- Python API

5.1 Graph

Contents

5.1.1 Graph Class

`class kedgeswap.Graph.Graph`(*directed=False*)

Bases: `object`

Read input graph and store graph as adjacency list

N

number of nodes

Type
int

M

number of edges

Type
int

neighbors

store adjacency list for each node

Type
dict(list)

in_neighbors

used only in directed graph, for each node store their neighbors from “in-edges”

Type
dict(list)

`out_neighbors`

used only in directed graph, for each node store their neighbors from "out-edges"

Type
dict(list)

`edges`

in undirected graph:

* for each edge (u,v), store the position | of v in the adjacency list of u | in directed graph:

* for each edge (u,v), store a quartuplet | (v_idx, u_idx, v_out_idx, u_in_idx), where: | v_idx is the position of v in u's adjacency list | u_idx is the position of u in v's adjacency list | v_out_idx is the position of v in out_neighbors[u]_ | u_in_idx is the position of u in in_neighbors[v] unique_edges: list()

used mostly for undirected graph, to store one version of each edge

Type
dict()

`directed`

enable if graph is directed

Type
bool

`copy()`

`read_ssv`(*in_file*)

Read space separated values Input format is separated with spaces or tabulations, e.g.:

0 1

3 2

2 4

.

.

.

where the first columns is the source node and the second column is the destination node. |When self.directed == True, the graph is considered directed and edges are stored as written in the file, else they are stored as (src, dest) with src < dest. Self Loop and multi-graphs are not accepted.

Parameters
`in_file` (*`str`*) – path to the input file

`read_ael`(*in_file*)

Read ael format TODO example format TODO lecteurs pour d'autres formats

`to_ael`(*output*)

`to_ssv`(*output*)

5.2 MarkovChain

Contents

5.2.1 MarkovChain Class

MarkovChain class, used to perform k-edge on a Graph object.

`class kedgeswap.MarkovChain.MarkovChain`(*graph*, *N_swap=0*, *gamma=0*, *use_jd=False*, *use_triangles=False*, *use_assortativity=False*, *use_mutualdiades=False*, *verbose=False*, *keep_record=False*, *log_dir=None*, *debug=False*)

Bases: `object`

make swaps

`pick_k`()

Pick k value using powerlaw distribution. The exponent of the powerlaw can be fixed by the gamma argument.

Returns
k (*int*) – number of edges to swap

`find_swap`(*k*)

Randomly pick k edges to swap, and randomly pick a permutation When self.force_k == True, permutation is a cyclic permutation, else it is a random permutation, with possible identity for some edges.

Parameters
k (`int`) – number of edges to swap

Returns

- **edge_to_swap** (*list(tuples)*) – list of the edges to swap
- **permutation** (*list(tuples)*) – list of the edges with which we should swap the edges in edge_to_swap
- **_edge_to_swap** (*list(int)*) – indexes in unique_edges of the edges in edge_to_swap

`check_swap`(*edge_to_swap*, *permutation*)

Verify constraints to see if swap can be accepted or not

Parameters

- **edge_to_swap** (`list(tuples)`) – list of the edges to swap
- **permutation** (`list(tuples)`) – list of the edges with which we should swap the edges in edge_to_swap

Returns
swap_accepted (*bool*) – true if swap can be accepted

`check_dyads`(*edge_to_swap*, *permutation*) `perform_swap`(*edge_to_swap*, *permutation*, *edge_to_swap_idx*)

When permutation is accepted, swap the edges in the graph data structure.

Parameters

- **edge_to_swap** (`list(tuples)`) – list of the edges to swap
- **permutation** (`list(tuples)`) – list of the edges with which we should swap the edges in edge_to_swap
- **edge_to_swap_idx** (`list(int)`) – index of the edges in graph.unique_edges (useful when undirected)

`init_assortativity`()

Compute Assortativity initial value, using the formula found in “Dutta, Fosdick et Clauset, 2022: Sampling random graphs with specified degree sequences”.

> Using the notation deg(u) for the degree of u, and Axy for the adjancency matrix value for nodes x and y, and Sk = sum_x (deg(x) ^ k), we compute the following values:
>
> > -S1, S2 and S3, -Sl= sum_xy (Axy * deg(x) * deg(y))

Using these values, the assortativity is computed as :

r = (S1 * Sl - S2 * S2) / (S1 * S3 - S2 * S2)

Since Sl is the only value to depend on the presence of each link, we store the denominator to update the assortativity value in O(1) after each swap.

`update_assortativity`(*edge_to_swap*, *permutation*)

Given a K-edge swap, update assortativy value using generalised formual from “Dutta, Fosdick et Clauset, 2022: Sampling random graphs with specified degree sequences”

`count_triangles`()

Enumerate and store all triangles found in the graph. For undirected graphs:

> we store each triangle in a set of tuplet ((u,v,w)) where | u, v and w are the node, with u < v < w, and we store each link | involved in the triangle in edges_in_triangles (pointing to the triangle tuplet) For directed graphs:
>
> we store each triangle thrice in a set of tuplet, with each node as a starting point, | e.g. for triangle (u,v,w) we store {(u,v,w), (v,w,u), (w,u,v)}. We store each link | involved in the triangle in edges_in_triangles (pointing to the triangle tuplet)

`update_triangles`(*edge_to_swap*, *permutation*)

Update the sets of triangles by looking at each edge swap:

- if the initial edge was involved in a triangle, remove triangle from sets
- if the goal edge creates a triangle, add it to the sets

`init_joint_degree`()

Initialize the joint degree matrix.

joint_degree[i - 1, j - 1] gives the number of links from nodes of degree i to nodes of the degree j. Initialise the joint degree matrix by looping over each node n, then each neighbor nn of n, and incrementing joint_degree[deg(n), deg(nn)] by 1/2. (increment by 1/2 to take into account that each edge is added twice)

update_joint_degree_old(*edge_to_swap*, *permutation*)

DEPRECATED - Only used for unit testing ! Given a permutation, compute the changed in the joint degree matrix. Compute the update by copying the joint degree matrix, looping over each edge swap, decrementing the joint degree value for the 'old' edges and incrementing the joint degree value for the 'new' edges.

Parameters: edge_to_swap : list of the edges to swap permutation : list of the edges with which we should swap the edges in edge_to_swap

Return : updated_joint_degree : np.array, the updated version of the joint degree matrix if the permutation given in input is performed.

update_joint_degree(*edge_to_swap*, *permutation*)

Given a permutation, compute the changed in the joint degree matrix. Compute the update by copying the joint degree matrix, looping over each edge swap, decrementing the joint degree value for the 'old' edges and incrementing the joint degree value for the 'new' edges.

Parameters: edge_to_swap : list of the edges to swap permutation : list of the edges with which we should swap the edges in edge_to_swap

Return : updated_joint_degree : np.array, the updated version of the joint degree matrix if the permutation given in input is performed.

run(*N_swap=None*)

K-edge swap algorithm. Start by computing assortativity initial value, then perform N_swap, each time checking the constraints and computing metrics.

5.3 Stat

Contents

5.3.1 Stat Class

class kedgeswap.Stat.**Stat**(*mc*, *eta=None*, *turbo=False*, *verbose=False*, *njobs=1*)

Bases: object

Class to compute statistics on a Markov Chain object. This class implements methods to estimate the Markov Chain's sampling gap, and to follow its convergence using the DFGLS test.

Attributes:

mc: MarkovChain object

The MarkovChain object on which we follow the convergence.

eta: float

The sampling gap used for the Markov Chain. The sampling gap gives a number of steps to make on the Markov Chain to obtain two uncorrelated graphs.

turbo: bool

Enable to make a fast but unverified estimation of the sampling gap.

verbose: bool
Enable to add information to the logs

`static CheckAutocorrLag1`(*S_T*, *alpha*)

Check the autocorrelation with lag 1 of a time serie.

Parameters

- `S_T` (*list(float)*) – List of assortativity(/number of triangles) values to test autocorrelation
- `alpha` (*float*) – Significance level of the test (usually fixed to 0.04).

`guesstimate_sampling_gap`(*graph*, *gamma*)

Sampling gap estimation is long, this function gives an empirical estimation of the sampling gap. Measure the acceptation rate A of the Markov Chain, and fix the sampling gap as 10*(1/A) * M, where M is the number of edges of the network. This estimation was fixed empirically to overestimate the sampling gap we measure using the estimation from Dutta, U. (2022).

`estimate_sampling_gap`(*graph*, *gamma*)

Estimate the sampling gap for the MCMC, following algorithm 1 (and using the same values) of Dutta, U. (2022). Sampling random graphs with specified degree sequences

`run_dfgls`(*output*)

If no sampling gap eta specified, run estimation of eta. Run Markov Chain for eta steps, retrieve list of assortativity values (or number of triangles) and estimate the convergence of this time serie, to decide if the Markov Chain is converged.

PYTHON MODULE INDEX

INDEX