

ORIGINAL ARTICLE

# Artificial Benchmark for Community Detection (ABCD)—Fast random graph model with community structure

Bogumił Kamiński<sup>1</sup> , Paweł Prałat<sup>2,\*</sup> , and François Thériberge<sup>3</sup> 

<sup>1</sup>Decision Analysis and Support Unit, SGH Warsaw School of Economics, Warsaw, Poland (e-mail: [bkamins@sgh.waw.pl](mailto:bkamins@sgh.waw.pl)),

<sup>2</sup>Department of Mathematics, Ryerson University, Toronto, Ontario, Canada and <sup>3</sup>Tutte Institute for Mathematics and Computing, Ottawa, Ontario, Canada (e-mail: [theberge@ieee.org](mailto:theberge@ieee.org))

\*Corresponding author. Email: [pralat@ryerson.ca](mailto:pralat@ryerson.ca)

Action Editor: Ulrik Brandes

## Abstract

Most of the current complex networks that are of interest to practitioners possess a certain community structure that plays an important role in understanding the properties of these networks. For instance, a closely connected social communities exhibit faster rate of transmission of information in comparison to loosely connected communities. Moreover, many machine learning algorithms and tools that are developed for complex networks try to take advantage of the existence of communities to improve their performance or speed. As a result, there are many competing algorithms for detecting communities in large networks. Unfortunately, these algorithms are often quite sensitive and so they cannot be fine-tuned for a given, but a constantly changing, real-world network at hand. It is therefore important to test these algorithms for various scenarios that can only be done using synthetic graphs that have built-in community structure, power law degree distribution, and other typical properties observed in complex networks. The standard and extensively used method for generating artificial networks is the LFR graph generator. Unfortunately, this model has some scalability limitations and it is challenging to analyze it theoretically. Finally, the mixing parameter  $\mu$ , the main parameter of the model guiding the strength of the communities, has a non-obvious interpretation and so can lead to unnaturally defined networks. In this paper, we provide an alternative random graph model with community structure and power law distribution for both degrees and community sizes, the Artificial Benchmark for Community Detection (ABCD graph). The model generates graphs with similar properties as the LFR one, and its main parameter  $\xi$  can be tuned to mimic its counterpart in the LFR model, the mixing parameter  $\mu$ . We show that the new model solves the three issues identified above and more. In particular, we test the speed of our algorithm and do a number of experiments comparing basic properties of both ABCD and LFR. The conclusion is that these models produce graphs with comparable properties but ABCD is fast, simple, and can be easily tuned to allow the user to make a smooth transition between the two extremes: pure (independent) communities and random graph with no community structure.

**Keywords:** community detection algorithms; synthetic models; random graphs

## 1. Introduction

An important property of complex networks is their community structure, that is, the organization of vertices in clusters, with many edges joining vertices of the same cluster and comparatively few edges joining vertices of different clusters (Fortunato, 2010; Girvan & Newman, 2002). In social networks, communities may represent groups by interest (practical applications include

© The Author(s), 2021. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

collaborative tagging), in citation networks they correspond to related papers, similarly in the web communities are formed by pages on related topics, etc. Being able to identify communities in a network helps to exploit it more effectively. For example, clusters in citation graphs may help to find similar scientific papers, discovering users with similar interests is important for targeted advertisement, and clustering can also be used for network compression and visualization. Finally, many machine learning algorithms and tools use clustering as an unsupervised pre-processing step and then try to take advantage of the community structure to improve their performance or speed.

The goal of community detection algorithms is to partition the vertex set of a graph into subsets of vertices called communities such that there are more edges present within communities in comparison to the global density of the graph. The key ingredient for many clustering algorithms is modularity. Modularity for graphs was introduced by Newman and Girvan (2004) and it is based on the comparison between the actual density of edges inside a community and the density one would expect to have if the vertices of the graph were attached at random regardless of community structure, while respecting the vertices degree on average. There are many variants allowing, in particular, overlapping or hierarchical communities. Moreover, it is also possible to generalize modularity for hypergraphs (Kamiński et al., 2019, 2021).

Unfortunately, detecting communities in networks is a challenging problem. Many algorithms and methods have been developed over the last few years—see, for example, Dao et al. (2020) for a recent review. It is important to point out that these algorithms are often quite sensitive and so they cannot be fine-tuned for a given family of networks we want these algorithms to work on. Some algorithms perform well on networks with strong communities but perform poorly on graphs with weak communities. The degree distribution and other properties of networks may also drastically affect the performance of these algorithms in terms of accuracy or computational complexity. Because of that, it is important to test these algorithms for various scenarios that can only be done using synthetic graphs that have built-in community structure, power law degree distribution, and other typical properties observed in complex networks.

In order to compare algorithms, one can use some quality measure, for example, the above-mentioned modularity (Newman & Girvan, 2004). Indeed, modularity is not only a global criterion to define communities and a way to measure the presence of community structure in a network but, at the same time, it is often used as a quality function of community detection algorithms. However, it is not a fair benchmark, especially for comparing algorithms (such as Louvain and Ensemble Clustering) that find communities by trying to optimize the very same modularity function! In order to evaluate algorithms in a fair and rigorous way, one should compare algorithm solutions to a synthetic network with an engineered ground truth.

The standard and extensively used method for generating artificial networks is the LFR (Lancichinetti, Fortunato, Radicchi) graph generator (Lancichinetti et al., 2008; Lancichinetti & Fortunato, 2009). This algorithm generates benchmark networks (i.e., artificial networks that resemble real-world networks) with communities. The main advantage of this benchmark over other methods is that it allows for the heterogeneity in the distributions of both vertex degrees and of community sizes. As a result, in the past decade, the LFR benchmark has become a standard benchmark for experimental studies, both for disjoint and for overlapping communities (Emmons et al., 2016). Some other benchmarks, including BTER and ReCoN—another well-known models, are discussed in the next section.

In order to generate a random graph following a given, previously computed, degree sequence, the LFR benchmark uses the fixed degree sequence model (also known as edge switching Markov chain algorithm) to obtain the desired community structure once the stationary distribution is reached. Unfortunately, the convergence process can be slow and so this model has some scalability limitations. Despite the need for experiments on large networks, the standard LFR implementation<sup>1</sup> can only be used to generate medium size networks (e.g., Figure 3 shows that

the graph on 500,000 nodes already takes several minutes to be generated). Moreover, due to its complexity and the fact that many fast implementations stop the switching before the stationary distribution is reached, it is challenging to analyze the model theoretically. Finally, the mixing parameter  $\mu$ , the main parameter of the model guiding the strength of the communities, has a non-obvious interpretation and so can lead to unnaturally defined networks. We discuss these issues at length in the next section.

In this paper, we provide an alternative random graph model with community structure and power law distribution for both degrees and community sizes, the Artificial Benchmark for Community Detection (ABCD graph). We show that the new model solves the three issues identified above. In particular, we test the speed of our algorithm and do a number of experiments comparing basic properties of both ABCD and LFR. The conclusion is that these models produce graphs with comparable properties but ABCD is fast, simple, and can be easily tuned between the two extremes: random graph with no community structure and independent communities. The Julia package providing an application programming interface (API) for the generation of ABCD graphs can be accessed at GitHub repository.<sup>2</sup> The repository also provides instructions how to set up R and Python to use the package directly from these environments. (For reference purposes, if requested, we can also provide a Python implementation of ABCD.) Moreover, a command line interface to the library is provided that allows users to generate ABCD graphs without using an API. Finally, let us mention that we currently work on parallel implementation of the model, ABCDe (enhanced implementation) that should be available at GitHub repository soon.

The paper is structured as follows. In the next section, Section 2, we justify the need for a new benchmark network model. Section 3 provides a detailed description of the model. In order to compare ABCD and LFR, one needs to tune the two mixing parameters to make the corresponding graphs comparable. We explain this process in Section 4. Section 5 presents experiments for comparison of the two models (both the speed and basic properties). Brief conclusion and directions for future work are presented in Section 6. Finally, pseudo-codes of our ABCD generator are presented in the Appendix.

## 2. Motivation

In the introduction, we already highlighted a few issues with the existing LFR benchmark. In this section, we provide more detailed justification for the need of a new benchmark model. This is not to say that we dislike the LFR model and propose an alternative that is substantially different. In fact, our ABCD model may be easily tuned such that its properties mimic the one of LFR but is faster than its competitor (Subsection 2.1). Hence, it seems that ABCD is a natural alternative for practitioners that already use and like the LFR benchmark. On the other hand, ABCD is easier to analyze theoretically (Subsection 2.2); research in that direction might be beneficial for a better understanding of networks with community structure and algorithms that are performed on them. ABCD has, arguably, more natural main parameter which prevents the user from generating graphs with “anti-communities” (Subsection 2.3). Finally, we challenge the “local” property in the LFR model that is insisted to be satisfied by communities and propose a “global” counterpart that is, arguably, more natural (Subsection 2.4).

### 2.1 Problem with scalability

In the big data era, there are many massive networks that need to be mined and analyzed. Since such networks cannot be handled in the memory of a single computer, new clustering methods have been introduced for advanced models of computation (Buzun et al., 2014; Zeng & Yu, 2016). These algorithms use hierarchical input representations which implies that the experiments performed on small or medium size benchmark graphs cannot be used to predict the performance on

much larger instances (Emmons et al., 2016). Unfortunately, many graph clustering benchmark generators currently available are not able to generate the graphs of necessary size (Bae & Howe, 2015; Buzun et al., 2014).

Let us briefly discuss the reason for the leading benchmark not to be scalable. Switching edges in LFR can be viewed as a transition in an irreducible, symmetric, and aperiodic Markov chain. As a result, it converges to the uniform (stationary) distribution. More importantly, if the maximum degree is not too large compared to the number of edges, then it converges in polynomial time (Greenhill & Sfragara, 2018). However, despite the fact that these bounds on the mixing time are of theoretical importance, they are not practical even for small graphs. The convergence process is inherently slow and so the model has clear scalability limitations that are known to both academics and practitioners. The fastest published variant of the model that is able to generate large graphs is the external memory algorithm proposed by Hamann et al. (2018).

In order to generate huge graphs, practitioners typically use computationally inexpensive random graph models such as R-MAT (Chakrabarti et al., 2004) or the generator of Funke et al. (2018). These models might create communities. In fact, it is known that many random graph models naturally create community structure, especially the ones that are geometric in nature (Prokhorenkova et al., 2017). However, they are not suitable for benchmarking purposes as there is no ground truth community structure to compare against. Hence, it is difficult to use them to evaluate clustering algorithms.

Another alternative, based on the scalable Block Two-Level Erdős–Rényi (BTER) graph generator (Seshadhri et al., 2012), was recently proposed by Slota et al. (2019). The original model takes into account the desired degree distribution and per-degree clustering coefficient. Since it does not explicitly aim to create communities, its edge generation process is more direct, simpler, and as a result faster than LFR's. Indeed, the scalability of BTER is impressive. The model aims to preserve a given degree distribution (similarly to LFR and ABCD that generate graphs with a given power law degree distribution) and a given clustering coefficient per degree. The latter objective is different than the one in LFR and ABCD; in these two models, the internal degree of community members can vary a lot. Hence, BTER generates graphs that are quite different from LFR or ABCD. The authors of Slota et al. (2019) try to twist the original model to create a graph that resembles the fLFR benchmark. However, due to inherent properties of BTER, they were unable to generate graphs that perfectly match the desired community structure. On the other hand, similarly as it is done in BTER, ABCD independently generates graphs induced by communities and the global graph but it generates “LFR-like” graphs. That is the main reason why both BTER and ABCD can be generated fast.

Finally, let us mention about the ReCoN (Replication of Complex Networks) model that was recently proposed in Staudt et al. (2017). This interesting model is very different than other benchmarks, including the ones we focus on in this paper. It uses a small reference graph to seed the process of generating large graph. Its main idea behind construction of the large output graph is similar to the LFR algorithm, and the performance of its implementation using NetworKit is about five times faster than that in the corresponding NetworKit implementation of LFR.

The proposed ABCD model is fast. The experiments we performed imply that ABCD is 40–100 times faster than the reference C++ implementation of LFR and over 10 times faster than the NetworKit implementation (see Subsection 5.2 for more details). In particular, a graph on 10 million vertices with an average vertex degree of 25 can be generated on a standard desktop computer in several minutes (see Table 1 for example timing reports; LFR algorithm implementation we used would take several hours to generate graphs of similar size). In this paper, we concentrate on single-threaded ABCD and LFR implementations in order to focus on the theoretical concepts behind ABCD. However, as an outlook for further work, it is possible to design a distributed out-of-core implementation of ABCD to generate huge graphs having billions of vertices, similarly like it is done in Hamann et al. (2018) for LFR. Indeed, for example, generation of

edges within communities can be performed using perfectly parallel approach, as each community is processed independently (see Section 3 for details).

## 2.2 Many variants and lack of theoretical foundations

The most computationally expensive part of the LFR benchmark is edge switching. In each step of this part of the algorithm, two edges are chosen uniformly at random and two of the endpoints are swapped if it removes the loop or parallel edge without introducing new ones. As already mentioned, the process converges to the stationary distribution but it does not converge fast enough for large graphs to be produced. Experimental results on the occurrence of certain motifs in networks (Milo et al., 2003), the average and maximum path length and link load (Gkantsidis et al., 2003) suggest that  $\Theta(m)$  swaps are enough to get close to the desired distribution, where  $m$  is the number of edges in the graph. (See also Ray et al., 2012 for further theoretical arguments and experiments.) The constant hidden in the asymptotic notation varies from experiment to experiment and is between 2 and 100. There are also some other heuristic arguments that justify more simplifications of the original algorithm.

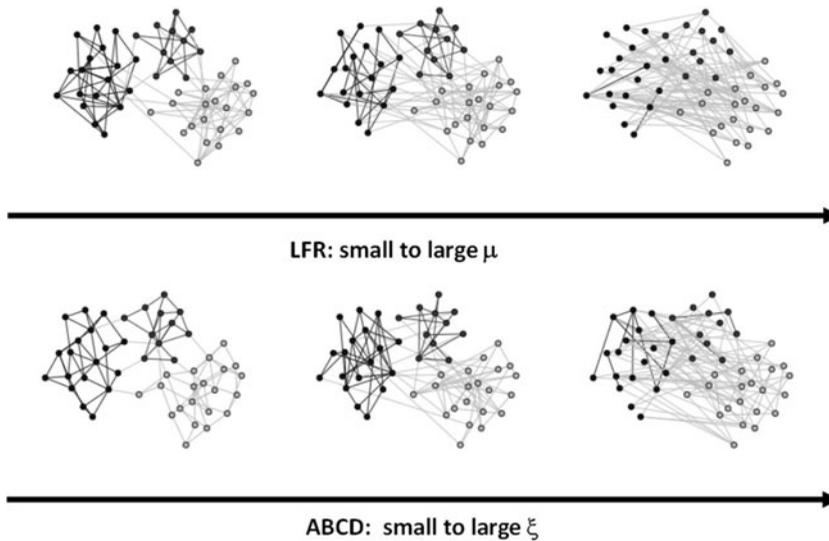
There are at least two negative implications of this situation. First of all, there are many variants of this benchmark model and various implementations further modify some steps, either as an attempt to simplify the algorithm or to gain on speed. As a result, one can only create “LFR-type” graphs, and graphs generated by different implementations can have different properties. In fact, even the original formulation of the model leaves some ingredients not rigorously defined. This is certainly not desired for benchmark graphs that should provide a rigorous and fair comparison. Moreover, it creates challenges with reproducing experiments, something that is expected, if not required, when reporting scientific results.

The lack of a simple and clear description of the algorithm has another negative aspect. Despite the fact that the initial work on Erdős–Rényi model did not aim to realistically model real-world networks, the number of papers on random graphs and their applications to model complex networks is currently exploding. Indeed, in the period after 1999, due to the fact that data sets of real-world networks became abundantly available, their structure has attracted enormous attention in mathematics as well as various applied domains. For example, one of the first articles of Albert and Barabási (1999) in the field is cited more than 35,000 times. There are many papers investigating models of complex networks starting with a natural generalization of the Erdős–Rényi model to a random graph with a given expected degree distribution (Chung & Lu, 2006) to more challenging models such as random hyperbolic graphs (Krioukov et al., 2010) or spatial preferential attachment graphs (Aiello et al., 2008). These results are not only interesting from theoretical point of view; they help us better understand the properties and the dynamics of these models by investigating local mechanisms that shape global statistics of the produced network. Despite this fact, there are very few results on theoretical properties of the LFR graphs. It is unfortunate, as more research on models with community structure might shed light on how communities are formed and help us design better and faster clustering algorithms.

As described in Section 3, the proposed ABCD model can be seen as a union of independent random graphs. As a result, its asymptotic behavior can be studied with the existing tools in random graph theory. Moreover, ABCD model is natural, relatively easy and straightforward to implement that limits a problem with reproducibility. Nevertheless, for those that look for “out-of-the-box” tool, we made it available as GitHub repository with a reference implementation.

## 2.3 Communities are unnaturally defined for large mixing parameters

One of the parameters of the LFR benchmark is the mixing parameter  $\mu \in [0, 1]$  which controls the desired “community tightness.” The goal is to keep the fraction of inter-community edges to



**Figure 1.** Examples of graphs generated by the LFR algorithm (top) and by the ABCD algorithm (bottom). All graphs have the same degree distribution and community sizes. The three LFR graphs correspond to values of the mixing parameter  $\mu \in \{0.1, 0.3, 0.95\}$ , whereas for the ABCD graphs the plots correspond to  $\xi \in \{0.1, 0.3, 0.95\}$ . Edges that fall between vertices in the same community are colored accordingly. We see strong communities for the leftmost plots, and noisy yet still coherent communities for the middle plots. The rightmost plots, where  $\mu = \xi = 0.95$ , illustrate our point regarding one of the main differences between LFR and ABCD. For LFR, in the top right plot, we see almost no edges within each community so the model generates “anti-communities.” With ABCD, we see a random looking graph, where the number of edges within each “community” is proportional to the number of vertices that belong to it, as expected in a random graph.

be approximately  $\mu$ . In one of the two extremes, when  $\mu = 0$ , all edges are within communities. On the other hand, when  $\mu = 1$ , LFR generates pure “anti-communities” with no edge present in any of the communities. We believe that this is undesired and leads to unnaturally defined communities. The threshold value of  $\mu$  that produces pure random graphs that are community agnostic is “hidden” somewhere in the interval  $[0, 1]$ . It is possible to compute this threshold value (see Section 3.4 where we actually do it), but the formula is quite involved and not widely known. Indeed, many different values are reported in the literature (e.g.,  $\mu = 0.7$  is mentioned in Slota et al., 2019) and so many experiments are performed on unnaturally defined networks and might lead to false conclusions. The influence of the parameter  $\mu$  on the LFR graph is presented in Figure 1 (top).

In contrast, the parameter  $\xi \in [0, 1]$  in the ABCD model (counterpart of  $\mu$  in LFR introduced in Section 3) has a natural and important interpretation. As in LFR, if  $\xi = 0$ , then all edges are generated exclusively within communities. More importantly,  $\xi = 1$  yields pure random graph in which communities do not affect the process of generating edges. Values of  $\xi \in (0, 1)$  produce graphs with additional signal coming from communities; the smaller the parameter, the more pronounced the communities are. One can easily move between  $\xi$  and  $\mu$  (again, see Section 3.4 for more details), but there is no risk to create unnaturally defined benchmark networks with “anti-communities.” The influence of the parameter  $\xi$  on the ABCD graph is presented in Figure 1 (bottom).

Finally, let us mention that here we only claim that large values  $\mu$  generate LFR graphs in which communities locally induce sparser graphs in comparison to global density, something that is not expected to happen in networks with community structure. Of course, there are many other properties that might be desired and, indeed, generating realistic networks is one of the major issues in most of the existing methods. In order to validate whether the model produces a realistic network, one needs to compare various properties measured on the real networks as it is done, for example, in ReCoN (Staudt et al., 2017) that we already discussed in Section 2.1.

## 2.4 Densities of communities

The LFR model aims to generate a graph in which  $(1 - \mu)$  fraction of edges adjacent to a given vertex stays within the community of that vertex. This property should hold for all vertices regardless whether this vertex belongs to large or small community. As a result, small communities will become much denser comparing to large ones. It is not clear that this property is desirable, especially in the case of unbalanced community sizes which the model is aiming for. Indeed, it seems to us that larger clusters should capture a proportionally larger fraction of edges—see Subsection 4.3 for a detailed discussion.

The approach used in LFR (which we call a local variant) seems to be inherited from the definition of the community in the classical book of Barabási (2016). We challenge it and propose another approach (that we call a global variant), although we do respect this point of view. For those researchers and practitioners who prefer the original approach, we describe two variants of the ABCD model, one for each approach, and both variants are available on GitHub repository.

## 3. Proposed model

### 3.1 Parameters of the model

We assume that the following parameters are provided as the input for the algorithm (for each input, we specify a general approach and a proposed default specification):

- (1) The number of vertices,  $n$ .  
(Notation: We label vertices with numbers from  $V = [n] := \{1, \dots, n\}$ .)
- (2) The exact (or expected) degree distribution  $\mathbf{w} = (w_1, \dots, w_n)$ . The user can decide if the degree distribution has to follow a given distribution exactly (the configuration model will be used in this case) or only to follow it in expectation (the Chung-Lu model will be used instead).  
(Remark: Note that the user does not have to provide vector  $\mathbf{w}$  explicitly; it could be generated at random. In particular, it could follow a power law distribution with parameter  $\gamma$  and extreme values  $w_{\min}$ ,  $w_{\max}$ . Alternatively, the average value  $\bar{w}$  can be supplied instead of  $w_{\min}$ , which can then be computed, as it is done in the original LFR model.)
- (3) The number of clusters,  $k$ , and the sequence of cluster sizes  $\mathbf{s} = (s_1, \dots, s_k)$  satisfying  $\sum_{i=1}^k s_i = n$ .  
(Remark: In particular,  $\mathbf{s}$  could be a random sequence following the power law distribution with parameter  $\beta$  and extremes  $s_{\min}$  and  $s_{\max}$ , as it is done in LFR. If not specified, by default LFR sets  $s_{\min}$  and  $s_{\max}$  to the minimum and the maximum degree, respectively.)  
(Notation: We label clusters with numbers from  $[k]$ . We will use  $f(\sigma(i)) \in [k]$  to denote the cluster of vertex  $i \in [n]$ , see Subsection 3.5 for a formal definition of this mapping.)
- (4) The mixing parameter  $\xi$ .  
(Remark: As already mentioned, at one extreme case when  $\xi = 0$ , all links are within clusters. On the other hand, if  $\xi = 1$ , then communities do not influence distribution of edges. Moreover, to add more flexibility, one may introduce different parameters  $\xi$  for each cluster—see Subsection 4.3 for more on that.)

### 3.2 Sampling $\mathbf{w}$ and $\mathbf{s}$

At the very beginning, we sample the exact/expected degree distribution  $\mathbf{w}$ , the number of clusters, and the cluster sizes  $\mathbf{s}$  (unless they are given as deterministic parameters of the model). The algorithms used to generate them are presented in Appendix. Let us stress the fact that if  $\mathbf{w}$  and  $\mathbf{s}$  are sampled, then they are random variables. However, for fair comparison purposes, the same values of  $\mathbf{w}$  and  $\mathbf{s}$  are used when experiments on LFR and ABCD models are performed in Section 5.

### 3.3 Background and cluster graphs

Our model can be viewed as a union of  $k + 1$  independent random graphs  $G_i$  ( $i \in [k] \cup \{0\}$ )—one for each cluster and one for the whole graph. As a result, one can view it as a generalization of the *double round exposure method* (also known in the literature as “*sprinkling*”). We start with the background graph  $G_0$  and “sprinkle” additional edges within communities that come from graphs  $G_i$  ( $i \in [k]$ ); the smaller the value of  $\xi$ , the stronger the ties between members of the same cluster are. As these graphs are generated independently, one can alternatively start with the cluster graphs and then “sprinkle” the background graph on top of it that can be seen as adding the “noise”; the larger the value of  $\xi$ , the larger the level of noise is.

First, we need to split the weight vector  $\mathbf{w}$  into  $\mathbf{y}$  and  $\mathbf{z}$ ;  $\mathbf{z}$  will be responsible for the background graph and  $\mathbf{y}$  will affect additional edges within communities. The process of splitting the weight is discussed in Section 3.4. Then, for a given cluster  $i \in [k]$ , we restrict ourselves to  $V_i \subseteq V = [n]$ , the set of vertices that belong to cluster  $i$ . We discuss the process of assigning vertices into clusters in Section 3.5. Let  $\mathbf{y}_i$  be the subsequence of  $\mathbf{y}$  restricted to terms corresponding to vertices from  $V_i$ . Let  $G_i = (V_i, E_i)$  be a random graph  $\mathbf{G}(\mathbf{y}_i)$  guided by the sequence  $\mathbf{y}_i$ —the exact model will be specified in Section 3.6. Finally, let  $G_0 = (V, E_0)$  be a random graph  $\mathbf{G}(\mathbf{z})$  guided by the sequence  $\mathbf{z}$ . We call graph  $G_0$  the background graph and the remaining graphs  $G_i$  (for  $i \in [k]$ ) are called the cluster graphs. The model is defined as the union of these graphs, that is,  $\mathbf{G} = (V, E)$ , where  $E = \bigcup_{i=0}^k E_i$ .

Note that  $\mathbf{G}$  allows loops and multiple edges. Indeed, they can occur both in any of the generated graphs  $G_i$  ( $i \in [k] \cup \{0\}$ ) or after taking a union of their edge sets. In general, however, there will not be very many of them, especially for sparse graphs. If one wants to study this random graph theoretically, one option is to work with multi-graphs or condition on  $\mathbf{G}$  to be simple. From a practical point of view, one can still work with multi-graphs or do some minor adjustments to the graph such as rewiring, re-sampling, or simply delete parallel edges. We will come back to this practical issue and provide a specific solution in Section 3.6. However, note that the proposed model of  $\mathbf{G}$  is important as it can be rigorously analyzed theoretically as all its components are well studied in graph theory literature and we take a union of independent graphs—see Subsection 2.2 for motivation for theoretical results.

### 3.4 Distribution of weights

Parameter  $\xi \in [0, 1]$  controls the fraction of edges that are between communities, that is, it reflects the amount of noise in the network. Its role is similar to the role of the mixing parameter  $\mu$  in the original LFR model. We split weights  $\mathbf{w}$  into  $\mathbf{y}$  and  $\mathbf{z}$  as follows, keeping the same value of  $\xi$  for each vertex (recall that  $\mathbf{y}$  will be associated with clusters and  $\mathbf{z}$  will be associated with the noise):

$$\begin{aligned}\mathbf{y} = (y_1, \dots, y_n) &= (1 - \xi)\mathbf{w} = ((1 - \xi) \cdot w_1, \dots, (1 - \xi) \cdot w_n) \\ \mathbf{z} = (z_1, \dots, z_n) &= \xi\mathbf{w} = (\xi \cdot w_1, \dots, \xi \cdot w_n)\end{aligned}$$

However, in order to add more flexibility, one may allow different coefficients  $\xi$  for each cluster. Let  $(\xi_1, \dots, \xi_k) \in [0, 1]^k$ . In the next subsection, vertices will be assigned into clusters: vertex  $i \in [n]$  will be assigned to cluster  $f(\sigma(i)) \in [k]$ . Then,

$$\begin{aligned}\mathbf{y} = (y_1, \dots, y_n) &= ((1 - \xi_{f(\sigma(1))}) \cdot w_1, \dots, (1 - \xi_{f(\sigma(n))}) \cdot w_n) \\ \mathbf{z} = (z_1, \dots, z_n) &= (\xi_{f(\sigma(1))} \cdot w_1, \dots, \xi_{f(\sigma(n))} \cdot w_n)\end{aligned}$$

This variant is important if one wants to mimic the original LFR model as closely as possible, that is, to try to keep the fraction of internal edges for each cluster equal; otherwise, using the same  $\xi$  for all vertices suffice—see Subsection 4.3 for more discussion.

### 3.5 Assigning vertices into clusters

Our task now is to assign vertices into clusters, that is, to define the mapping  $f:[n] \rightarrow [k]$  from vertices to clusters. Our goal is to design the fast algorithm that produces an assignment selected uniformly at random from some natural class of admissible assignments (formal definition is provided below).

The main problem is that vertices of large degree cannot be assigned to small clusters, as we aim to generate simple graphs for some applications of the proposed model. Recall that the weight vector  $\mathbf{w}$  will be split into  $\mathbf{y}$  and  $\mathbf{z}$  that will guide the process of generating cluster graphs and, respectively, the background graph. All edges within cluster graphs will end up between vertices of the same community. On the other hand, only some fraction of the background edges will be present within communities as an effect of the random sampling. Unfortunately, the number of such edges depends on the mapping  $f$  we are about to create, and so it is not known at this point. So how can we decide which vertex can be assigned to a given cluster leaving enough room for not only edges from the cluster graph but also for additional edges coming from the background graph? Fortunately, this “chicken and egg” problem can be solved as there exists a universal upper bound  $x_i$  for  $y_i$  that leaves enough room for the edges coming from the background graphs that works for all  $i \in [n]$ , namely,

$$x_i := \left\lceil (1 - \xi\phi)w_i \right\rceil, \tag{1}$$

where  $\phi := 1 - \sum_{\ell \in [k]} (s_\ell/n)^2$ . The reason for this choice of  $x_i$  is as follows. In Subsection 4.1, we will show that the expected number of edges between communities is equal to  $\xi\mu_0$ , where  $\mu_0 = 1 - \sum_{\ell \in [k]} (W_\ell/W)^2$  ( $W_\ell$  is the volume of cluster  $\ell$  and  $W$  is the volume of the whole graph). If vertices are assigned to clusters randomly, then the expected value of  $W_\ell$  is  $s_\ell W$ . It follows that  $\phi$  is a good approximation of  $\mu_0$  that is not known at this point. In any case, since  $\phi < 1$ , we observe that  $x_i \geq (1 - \xi\phi)w_i \geq (1 - \xi)w_i = y_i$  and so there is definitely room for edges of the cluster graphs.

Let us call an assignment of vertices into clusters admissible if each vertex  $i \in [n]$  is assigned to cluster  $j \in [k]$  with  $x_i \leq s_j - 1$ . Recall that our goal is to select one admissible assignment uniformly at random. This condition is a necessary condition for the existence of a simple graph that this cluster induces. Note that it is only a necessary condition; in fact, the corresponding degree sequence has to be graphic. (A graphic sequence is a sequence of numbers which can be the degree sequence of some graph; see, e.g., West, 2001 or any other textbook on graph theory for more.) We use this slightly weaker condition because it is much simpler and more convenient to use which gives us an easier framework to work with. Finally, let us stress that for the final graph  $\mathbf{G}$  to be able to be simple, we get some additional constraints on admissible assignments of vertices into clusters. Not only  $\mathbf{z}$  and all  $\mathbf{y}_i$ 's must be graphic, but the union of all graphs needs to be simple as well. However, in practice, this causes no issue as we usually deal with sparse graphs that leave a lot of room for graphs to be fit.

Indeed, in practice, the probability that a non-graphic degree sequence is obtained for some cluster graph is extremely low. However, in order to deal with such potential problematic situations, the algorithm tries to assign as many edges as possible to stay within the cluster graph and move the remaining ones to the background graphs. See the end of Subsection 3.6 for more details.

A formal definition is slightly technical. Suppose that vertices are sorted according to their bounds on the expected/exact internal degree, that is,  $x_1 \geq x_2 \geq \dots \geq x_n$ . Similarly, suppose that cluster sizes are sorted, that is,  $s_1 \geq s_2 \geq \dots \geq s_k$ . In order to define the assignment of vertices into clusters, we need the following auxiliary sequence  $s_{\leq \ell}$ . For each  $\ell \in [k] \cup \{0\}$ , let

$$s_{\leq \ell} := \sum_{i=1}^{\ell} s_i$$

In particular,  $s_{\leq 0} = 0$  and  $s_{\leq k} = n$ . Function  $f: [n] \rightarrow [k]$ , that we informally introduced earlier, is defined as follows. For each  $i \in [n]$  and  $j \in [k]$ , we fix

$$f(i) = j \quad \text{if and only if} \quad s_{\leq j-1} < i \leq s_{\leq j}$$

The assignment now can be viewed as a permutation  $\sigma: [n] \rightarrow [n]$ —vertex  $i \in [n]$  is assigned to cluster  $f(\sigma(i)) \in [k]$ . Such assignments guarantee that the right number of vertices is assigned to each cluster, but vertices of large degree could be assigned to small clusters. Let  $\mathcal{A}$  be the set of admissible assignments defined as follows:

$$\mathcal{A} = \left\{ \sigma: [n] \rightarrow [n] : x_i \leq s_{f(\sigma(i))} - 1 \text{ for all } i \in [n] \right\}$$

In other words, no vertex in an admissible assignment gets assigned to a cluster of size smaller than or equal to its expected/exact degree. Our goal is to select one member of the family  $\mathcal{A}$  uniformly at random.

Sampling with uniform distribution is often a difficult task. Of course, generating one permutation with uniform distribution on the set of *all* permutations is easy and can be done in many different ways. If such permutation falls into  $\mathcal{A}$ , then we could accept it; otherwise, we repeat the process until we get one that does it. Unfortunately, the size of  $\mathcal{A}$  comparing to  $n!$ , the number of all possible permutations, can be very small so this rejection sampling process is not feasible from a practical point of view. However, this point of view does have theoretical implications and might be useful in the future for analyzing the model.

Fortunately, sampling uniformly from  $\mathcal{A}$  turns out to be relatively easy. To that end, we will use the following natural algorithm. (See also a pseudo-code in the Appendix.) Recall that vertices are sorted according to their bounds on internal degrees, that is, sequence  $\mathbf{x} = (x_1, \dots, x_n)$  is non-increasing. Consider vertices, one by one, starting with vertices that are associated with large values of  $x_i$  and assign them randomly to a cluster that has size larger than the corresponding bound and still has some “free spots”; that is, a cluster of size  $s_j$  is considered for a vertex of degree  $x_i$  if  $x_i \leq s_j - 1$  and the number of vertices already assigned to it is less than  $s_j$ . The probability that a given vertex is assigned to a given cluster is proportional to the number of “free spots” that remain in that cluster.

The reason why this algorithm produces an admissible assignment uniformly at random comes from the fact that clusters that are assigned to earlier vertices could also be assigned to vertices considered later. In other words, it is *not* the case that vertices considered earlier could make decisions that create more (or less) choices for vertices considered later. They need to be assigned somewhere and, regardless of where they get assigned, the number of choices left for future vertices is not affected. In particular, the algorithm always terminates, *unless*  $\mathcal{A} = \emptyset$ .

To see a formal argument, let

$$t_i = \max\{s_{\leq \ell} : x_i \leq s_{\ell} - 1\}$$

It is straightforward to see that  $\sigma \in \mathcal{A}$  if and only if  $\sigma(i) \in [t_i]$  for all  $i \in [n]$ . Note that, for any given admissible permutation  $\sigma \in \mathcal{A}$ , our algorithm produces it with probability  $p$  that is only a function of  $t_i$  but does not depend on  $\sigma$ . Indeed, it is easy to see that:

$$p = \prod_{i=1}^n \frac{1}{t_i - i + 1}$$

as there are  $t_i - (i - 1)$  available “free spots” for a vertex  $i \in [n]$ . Clearly, the algorithm does not produce any permutation that is not admissible. Hence, indeed, the algorithm generates a permutation from  $\mathcal{A}$  uniformly at random. As mentioned above, the algorithm fails only if  $\mathcal{A} = \emptyset$ .

**3.6 Exact versus expected degree distribution—Two variants of the model**

We will consider two variants of the model: the first one generates graphs with the expected degree distribution  $\mathbf{w}$  (related to the well-known Chung-Lu model) and the second one with the exact degree distribution  $\mathbf{w}$  (related to another well-known model, the configuration model). We will start with the description of the first variant, as it is slightly easier. However, it is presumably the case that the practitioners prefer the second variant. (In particular, a potential appearance of isolated vertices in sparse Chung-Lu models might not be desirable for practical purposes.)

Recall that at this point, we have vertices assigned to clusters: vertex  $i \in [n]$  belongs to cluster  $f(\sigma(i))$ . Moreover, the weight vector  $\mathbf{w}$  is split into two vectors  $\mathbf{y}$  and  $\mathbf{z}$  that will guide the creation of cluster graphs  $G_i$  ( $i \in [k]$ ) and, respectively, the background graph  $G_0$ . We need to specify how we actually do it and how we deal with potential problems after taking the union of these graphs.

**3.6.1 The expected degree distribution**

In this variant of the model, we use the Chung-Lu model that produces a random graph with expected degree sequence following a given sequence.

*Chung-Lu model*

Let  $\mathbf{w} = (w_1, \dots, w_n)$  be any vector of  $n$  real numbers and let  $W = \sum_{i=1}^n w_i$ . We define  $\mathcal{C}(\mathbf{w}) = ([n], E)$  to be the probability distribution of graphs on the vertex set  $[n]$  following the well-known Chung-Lu model (Chung & Lu, 2006; Seshadhri et al., 2012; Kolda et al., 2014; Winlaw et al., 2015). In this model, each set  $e = \{i, j\}$ ,  $i, j \in [n]$ , is independently sampled as an edge with probability given by:

$$\mathbb{P}(i, j) = \begin{cases} \frac{w_i w_j}{W}, & i \neq j \\ \frac{(w_i)^2}{2W}, & i = j \end{cases}$$

(Let us mention about one technical assumption. Note that it might happen that  $\mathbb{P}(i, j)$  is greater than one and so it should really be regarded as the expected number of edges between  $i$  and  $j$ , e.g., as suggested in Newman, 2010, one can introduce a Poisson-distributed number of edges with mean  $\mathbb{P}(i, j)$  between each pair of vertices  $i, j$ . However, since typically the maximum degree  $\Delta$  satisfies  $\Delta^2 \leq 2|E|$ , it rarely creates a problem and so we may assume that  $\mathbb{P}(i, j) \leq 1$  for all pairs.)

One desired property of this random model is that it yields a distribution that preserves the expected degree for each vertex, namely: for any  $i \in [n]$ ,

$$\mathbb{E}[\deg(i)] = \sum_{j \in [n] \setminus \{i\}} \frac{w_i w_j}{W} + 2 \cdot \frac{(w_i)^2}{2W} = \frac{w_i}{W} \sum_{j \in [n]} w_j = w_i$$

*Theoretical approach*

The original Chung-Lu model is a multi-graph, so it is natural and convenient to stay with multi-graphs in our model too. We simply take  $G_i = \mathbf{G}(\mathbf{y}_i) = \mathcal{C}(\mathbf{y}_i)$  for each  $i \in [k]$ , and  $G_0 = \mathbf{G}(\mathbf{z}) = \mathcal{C}(\mathbf{z})$ .

*Practical approach—Insisting on simple graphs*

From practical point of view, it is desired to generate a simple graph and use the fast algorithm that does it. In order to achieve both things, we use a version of the (fast) Chung-Lu model that produces the graph with a given number of edges. As a result, we need to round some numbers to

integers. We use the following randomized way that is also used in the original LFR model. For a given integer  $k \in \mathbb{Z}$  and real number  $\ell \in [0, 1)$ , let

$$\lceil k + \ell \rceil = \begin{cases} k & \text{with probability } 1 - \ell \\ k + 1 & \text{with probability } \ell \end{cases}$$

(Note that the expected value of random variable  $\lceil k + \ell \rceil$  is equal to  $k + \ell$ .)

We independently generate cluster graphs  $G_i$  ( $i \in [k]$ ) as follows. Note that  $\sum_{v \in V_i} y_v/2$  is the expected number of edges in  $\mathcal{C}(y_i)$ . We fix

$$e_i := \left\lfloor \frac{1}{2} \sum_{v \in V_i} y_v \right\rfloor \in \mathbb{N} \cup \{0\}$$

and then we generate the Chung-Lu graph  $\mathcal{C}(y_i)$  conditioning on not having parallel edges or loops, and having exactly  $e_i$  edges. This can be done in a fast way. We independently sample two vertices  $i$  and  $j$  with probabilities proportional to their weights. If  $i \neq j$  and adding an edge  $\{i, j\}$  does not create a parallel edge, then we accept it. We continue this process until  $e_i$  edges are created.

Once all cluster graphs are created, we move to the background graph. In order to keep the total number of edges as desired, we fix

$$e := \frac{1}{2} \sum_{v \in V} w_v - \sum_{i=1}^k e_i$$

Note that  $\sum_{v \in V} w_v$  is usually an even integer (since vector  $\mathbf{w}$  corresponds to the degree sequence) so  $e \in \mathbb{N} \cup \{0\}$ . (If not, we may replace  $\sum_{v \in V} w_v/2$  with  $\lfloor \sum_{v \in V} w_v/2 \rfloor$ .) Note also that the expected value of  $e$  is equal to  $\sum_{v \in V} z_v/2$ , the expected number of edges in  $\mathcal{C}(z)$ . We generate the Chung-Lu graph  $\mathcal{C}(z)$  conditioning on not having loops, not creating parallel edges (in the union of all cluster graphs and the background edges created so far!), and having exactly  $e$  edges. To that end, we use the same fast algorithm as before. Note that, as long as the whole graph is sparse (which is typically the case), the second step is fast since not too many collisions occur, even if some of the cluster graphs  $G_i$  ( $i \in [k]$ ) are dense.

### 3.6.2 The exact degree distribution

This variant of the model uses the configuration model (instead of the Chung-Lu model) that produces a random graph with a given degree sequence. However, this change brings a few small issues that need to be dealt with.

#### Configuration model

Let  $\mathbf{w} = (w_1, \dots, w_n)$  be any vector of  $n$  nonnegative integers such that  $W := \sum_{i=1}^n w_i$  is even. We define a random multi-graph  $\mathcal{M}(\mathbf{w})$  with a given degree sequence known as the configuration model (sometimes called the pairing model), which was first introduced by Bollobás (1980). (See Bender & Canfield, 1978; Wormald, 1984 for related models and results.)

Let us consider  $W$  configuration points partitioned into  $n$  labeled buckets  $v_1, \dots, v_n$ ; bucket  $v_i$  consists of  $w_i$  points. A pairing of these points is a perfect matching into  $W/2$  pairs. (There are  $W!/((W/2)!2^W)$  such pairings.) Given a pairing  $P$ , we may construct a multi-graph  $G(P)$ , with loops and parallel edges allowed, as follows: the vertices are the buckets  $v_1, \dots, v_n$ , and a pair  $\{x, y\}$  in  $P$  corresponds to an edge  $\{v_i, v_j\}$  in  $G(P)$ , if  $x$  and  $y$  are contained in the buckets  $v_i$  and  $v_j$ , respectively. We take a pairing  $P$  uniformly at random from the family of all pairings of  $W$  points and set  $\mathcal{M}(\mathbf{w}) = G(P)$ .

It is an easy but a fundamental fact that the probability of a random pairing corresponding to a given simple graph  $G$  is independent of the graph. Indeed, an easy calculation shows that every

simple graph corresponds to exactly  $\prod_{i=1}^n w_i!$  pairings. Hence, the restriction of the probability space of random pairings to simple graphs is precisely  $\mathcal{S}(\mathbf{w})$ , the uniform probability space of all *simple* graphs with a given degree sequence. Moreover, it is well known that if

$$\sum_{i=1}^n w_i = \Theta(n) \quad \text{and} \quad \sum_{i=1}^n w_i^2 = O(n)$$

then the expected number of loops and multiple edges that are present in  $\mathcal{M}(\mathbf{w})$  is  $O(1)$  and so the probability that  $\mathcal{M}(\mathbf{w})$  is simple tends to  $\delta = \delta(\mathbf{w}) > 0$  which depends on  $\mathbf{w}$  but is always separated from zero. As a result, event holding a.a.s. (i.e., with probability tending to 1 as  $n \rightarrow \infty$ ) over the probability space  $\mathcal{M}(\mathbf{w})$  also holds a.a.s. over the corresponding space  $\mathcal{S}(\mathbf{w})$ . For this reason, asymptotic results over random pairings immediately transfer to  $\mathcal{S}(\mathbf{w})$ . One of the advantages of using this model is that the pairs may be chosen sequentially, so that the next pair is chosen uniformly at random over the remaining (unchosen) points.

*Distribution of weights*

We assume that integer-valued vector  $\mathbf{w}$  is such that  $\sum_i w_i$  is even, so that a given degree sequence is feasible. (As mentioned earlier, it is only a trivial, necessary condition—in fact,  $\mathbf{w}$  should be a graphic sequence.) Recall that the weight, vector  $\mathbf{w}$ , is split into real-valued vectors  $\mathbf{y}$  and  $\mathbf{z}$ . However, since we deal with exact degree sequences not expected ones, this time we have two additional constraints that we need to satisfy, namely, that (a) all involved weights are integers, and (b) for each of the  $k$  clusters, the corresponding sum of weights is even. Note that once these conditions are satisfied for all cluster graphs, the background graph immediately has them too—all degrees are integers and the sum of weights is even.

We split  $\mathbf{w}$  into integer-valued vectors  $\hat{\mathbf{y}} = (\hat{y}_1, \dots, \hat{y}_n)$  and  $\hat{\mathbf{z}} = (\hat{z}_1, \dots, \hat{z}_n)$  as follows. For each cluster  $i \in [k]$ , we identify the leader, vertex of the largest weight in cluster  $i$ . (If more than one vertex has the largest weight, we select one of them to be the leader, arbitrarily.) In order to deal with non-integer values, for all vertices  $i \in [n]$  that are *not* leaders, we set  $\hat{y}_i = \lfloor y_i \rfloor$ . For the remaining  $k$  vertices, the leaders, we round  $y_i$  up or down so that the sum of weights in each cluster is even. (If some leader has the weight  $y_i \in n$  and the sum of weights in its cluster is odd, then we randomly make a decision whether subtract or add 1 to make the sum to be even.)

*Theoretical approach*

We take  $G_i = \mathbf{G}(\hat{\mathbf{y}}_i) = \mathcal{M}(\hat{\mathbf{y}}_i)$  for each  $i \in [k]$ , and  $G_0 = \mathbf{G}(\hat{\mathbf{z}}) = \mathcal{M}(\hat{\mathbf{z}})$ . Some of the involved graphs might not be simple but the expected number of loops and parallel edges is small, especially for sparse graphs. We have a few options how to deal with them. The first option is the easiest: we could do nothing and work with multi-graphs. Alternatively, we could condition on all  $G_i$  ( $i \in [k] \cup \{0\}$ ) to be simple. From theoretical point of view, this model is equally easy to analyze, provided that for each  $G_i$ , the probability of getting a simple graph tends to a constant as  $n \rightarrow \infty$  (does not matter how small it is and could be different for each  $i \in [k] \cup \{0\}$ ). It is known that under some mild assumptions, this is the case (in particular, the order of each cluster graph should tend to infinity with  $n$ , etc.)—see above for the discussion around the configuration model. Let us remark that even though all  $G_i$ 's are simple, it is not guaranteed that the final graph,  $\mathbf{G}$ , is simple as edges from  $G_0$  can overlap with edges of  $G_i$  for some  $i \in [k]$ . Hence, we could condition on  $\mathbf{G}$  to be simple. Unfortunately, this model might be more challenging to analyze (as it introduces some dependencies between the background graph and the cluster graphs), but this is certainly worth investigating in the future work.

*Practical approach—Insisting on simple graphs*

Before we discuss how we apply these observations to our problem, let us discuss a general approach and some theoretical, asymptotic results. Let us generate a random graph with a given degree sequence using the configuration model. If it happens that it is a simple graph, it is a

uniformly distributed random graph from the family of simple graphs with this degree sequence. Suppose then that it is not simple. It is known that after performing some kind of “switching,” we get a random graph that is very close to the uniform distribution and we should solve all problems in  $O(1)$  time. Indeed, in Janson (2020), it is proved that, assuming essentially a bounded second moment of the degree distribution, the configuration model with the simplest types of switchings yields a simple random graph with an almost uniform distribution, in the sense that the total variation distance is  $o(1)$ . For each parallel edge  $uv$ , one needs to choose a random edge  $xy$ , remove  $uv$  and  $xy$ , and with probability  $1/2$  add  $ux$  and  $vy$ ; otherwise, add  $uy$  and  $vx$ .

Let us now explain how we actually apply switchings to our problem. We start with the configuration model to generate cluster multi-graphs  $G_i$  ( $i \in [k]$ ). We then apply switchings to get a family of simple graphs. After that, we use the configuration model to generate the background graph  $G_0$  and use switchings to remove loops and parallel edges. After taking the union, more parallel edges could be created. As usual, we use switchings to remove them. However, this time we restrict ourselves to edges in the background graph and switch only those. This can be done since all graphs  $G_i$  are simple at this point and so collisions must involve at least one edge from the background graph. During switching, more collisions can be created but each collision again involves at least one edge from the background graph (after switching the resulting edges are kept in the background graph). We do this to preserve the number of internal edges within cluster; the cluster graphs are not affected by this final round of switchings.

In order for our algorithm to be fast in all potential situations, we have implemented a procedure that controls the process of fixing multiple edges and self-loops, so that it is not extremely slow (and, in particular, to be robust against a mentioned earlier rare possibility of obtaining a non-graphic degree sequence). If some cluster graph  $G_i$  is extremely dense, it might be computationally expensive (or simply impossible for non-graphic degree sequence) to sample a correct replacement that maintains all the desired constraints. This situation is extremely rare but if it happens, then we retry it only for a limited number of times. This creates a small bias for the number of edges captured within that community, but we have empirically found that it happens less than 1 per 1,000,000 edges for typical tight configurations of the model so the bias should not be noticeable in practice. It is possible to resolve all conflicts exactly (unless, of course, a non-graphic degree sequence is obtained, which is rare), so this is simply a trade-off between the speed and the quality of the implementation.

In summary, the conflict resolution algorithm we use for each cluster graph  $G_i$  works as follows:

1. perform a standard configuration model on  $G_i$  but put all self-loops and multiple edges in a *recycle* list assigned to this graph;
2. iteratively, remove one edge from the *recycle* list and try to rewire it with randomly selected edge from  $G_i$  including those from the *recycle* list; this process is tried as many times as the target number of edges in  $G_i$  (so, in expectation, each edge is tried for rewiring once); if we successfully do the switching, then we move forward; otherwise, we return the chosen edge back to the *recycle* list;
3. the whole process is repeated as long as we are able to find a good rewiring for an edge in *recycle* until *recycle* becomes empty or the number of times we were unable to reduce the *recycle* list size is equal to the size of *recycle*, that is, we unsuccessfully tried to recycle all edges in *recycle*; in such a case, we give up and move the remaining degrees of the vertices forming those unmatched edges from  $G_i$  to the global graph, so that the final degree of all vertices in the union graph follows  $\mathbf{w}$ —as noted above, this action is extremely rare—approximately less frequent than once per 1,000,000 edges.

For the background graph, we follow the same procedure. However, we do not “give up” recycling and follow the process until all required edges are created. As the background graph is sparse, this process is very fast in practice.

### 4. Comparing ABCD and LFR

The role of parameter  $\xi$  in ABCD is similar to the one of the parameters  $\mu$  in LFR; however, they are not the same! If  $\xi = \mu = 0$ , then in both models all edges are within communities, but if  $\xi = \mu = 1$ , then ABCD is a random graph and so is substantially different than LFR which produces “anti-communities.” As a result, in order to compare the two models, one needs to tune the parameters such that the corresponding densities of communities are comparable.

There are two natural ways of distributing the weight  $w$ —the first one preserves the densities globally, whereas the second one preserves it locally for each vertex in the graph. We will independently consider both approaches. After that, we will discuss the difference between the two and their implications. However, before that let us recall one subtle caveat we already discussed in Section 3.5 when we assigned vertices into clusters.

If one creates a pure ABCD graph, then  $\xi$  is known upfront and there is no issue. Now, we try to find  $\xi$  for ABCD that matches given  $\mu$  for LFR. The problem is that we cannot compute  $\xi$  before vertices are assigned to clusters. On the other hand, to do the assignment, we need to bound the number of neighbors of each vertex that belong to its own cluster graph that is a function of  $\xi$ —recall Equation (1). To overcome this “chicken and egg” problem, we apply some universal upper bound  $x_i$  for  $y_i$ , namely  $x_i := \lceil (1 - \mu)w_i \rceil$  to do the assignment, and then compute  $\xi$ . Hence, in what follows, we may assume that the assignment is given to us and we simply tune  $\xi$  to match given  $\mu$ .

#### 4.1 Recovering the mixing parameter $\mu$ (Globally)

In this scenario, we start with a fixed  $\xi$  that will be applied for all vertices regardless to which cluster they belong to. Recall that  $V_\ell = \{t \in V : f(\sigma(t)) = \ell\}$  ( $\ell \in [k]$ ) is the set of vertices assigned to cluster  $\ell$ . Let  $W = \sum_{t \in V} w_t$  be the volume of  $G$  and let  $W_\ell = \sum_{t \in V_\ell} w_t$  be the expected/exact volume of vertices of cluster  $\ell$ . Clearly,  $W = \sum_{\ell \in [k]} W_\ell$ .

There are two models (namely, Chung-Lu and configuration model) used to generate multi-graphs  $G_i$  ( $i \in [k] \cup \{0\}$ ), but both of them have the property that edges occur with probability proportional to the product of the weights of the two endpoints. Consider two vertices  $i$  and  $j$  with weights  $w_i$  and  $w_j$ , respectively. If they are in different clusters ( $f(\sigma(i)) \neq f(\sigma(j))$ ), then the probability that they are adjacent is equal to

$$\frac{z_i z_j}{\sum_{t \in V} z_t} = \frac{\xi w_i \cdot \xi w_j}{\sum_{t \in V} \xi w_t} = \xi \frac{w_i w_j}{\sum_{t \in V} w_t} = \xi \frac{w_i w_j}{W}$$

(In fact, for multi-graphs, it is the expected number of edges as the value above could potentially exceed 1. However, it is a rare situation in practice.) It follows that the fraction of edges that are between communities is equal to:

$$\begin{aligned} \frac{1}{W} \sum_{i \in V} \sum_{j \in V \setminus V_{f(\sigma(i))}} \xi \frac{w_i w_j}{W} &= \frac{\xi}{W^2} \sum_{i \in V} w_i \sum_{j \in V \setminus V_{f(\sigma(i))}} w_j = \frac{\xi}{W^2} \sum_{i \in V} w_i (W - W_{f(\sigma(i))}) \\ &= \frac{\xi}{W^2} \sum_{\ell \in [k]} W_\ell (W - W_\ell) = \frac{\xi}{W^2} \left( W^2 - \sum_{\ell \in [k]} W_\ell^2 \right) \\ &= \xi \left( 1 - \sum_{\ell \in [k]} (W_\ell / W)^2 \right) = \xi \mu_0 \end{aligned}$$

where  $\mu_0 = 1 - \sum_{\ell \in [k]} (W_\ell/W)^2$ . Hence, in order to mimic the structure of the LFR graph, one should consider

$$\xi = \frac{\mu}{\mu_0} = \mu \left( 1 - \sum_{\ell \in [k]} (W_\ell/W)^2 \right)^{-1} \tag{2}$$

On the other hand, if vertices  $i$  and  $j$  are in the same cluster  $\ell$ , the probability is equal to:

$$\begin{aligned} \frac{z_i z_j}{\sum_{t \in V} z_t} + \frac{y_i y_j}{\sum_{t \in V_\ell} y_t} &= \xi \frac{w_i w_j}{\sum_{t \in V} w_t} + (1 - \xi) \frac{w_i w_j}{\sum_{t \in V_\ell} w_t} \\ &= \frac{\xi w_i w_j}{W} + \frac{(1 - \xi) w_i w_j}{W_\ell} = \frac{w_i w_j}{W} + (1 - \xi) w_i w_j \left( \frac{W - W_\ell}{W \cdot W_\ell} \right) \end{aligned}$$

The expected number of neighbors of  $i$  that are in cluster  $\ell$  then equal to:

$$\sum_{j \in V_\ell} \left( \frac{\xi w_i w_j}{W} + \frac{(1 - \xi) w_i w_j}{W_\ell} \right) = w_i \left( \xi \frac{W_\ell}{W} + (1 - \xi) \right) = w_i \left( \frac{W_\ell}{W} + (1 - \xi) \frac{W - W_\ell}{W} \right). \tag{3}$$

Let us make one remark. Note that if  $\mu > \mu_0$ , then the corresponding value of  $\xi$  is greater than 1. As a result, we cannot generate our random graph. One can see it as a potential problem but, in fact, it is the opposite. Such values of  $\mu$  correspond to models in which the density between clusters is larger than the internal density. As discussed in Subsection 2.3, we should not be ever concerned with such networks with “anti-communities.”

#### 4.2 Recovering the mixing parameter $\mu$ for each vertex (locally)

In this scenario, we consider a sequence of parameters  $\xi_i$  ( $i \in [k]$ ), one per each cluster. In the original LFR model, once the degree sequence  $\mathbf{w} = (w_1, \dots, w_n)$  is fixed, the algorithm tries to rewire the edges such that for each vertex  $i$ , the internal degree is close to  $(1 - \mu)w_i$ . There is some variability in the final “local” mixing parameters but mainly due to the presence of low degree vertices which clearly must deviate from the desired ratio.

It is not clear if matching local parameters is what we want (see the discussion in the introduction and in Subsection 4.3) but here is a possible way to modify the approach presented above in order to have local mixing parameters close to  $\mu$ . Instead of using the same ratio  $\xi$  for splitting weights into background and cluster portions, one can carefully tune it and use different values of  $\xi$  for different clusters. Consider vertex  $i$  with degree  $w_i$  that belongs to a cluster with the total weight equal to  $W_{f(\sigma(i))}$ . For the background graph  $G_0 = \mathbf{G}(\mathbf{z})$ , let  $z_i = \xi_{f(\sigma(i))} \cdot w_i$  be such that:

$$z_i \left( \frac{W - W_{f(\sigma(i))}}{W} \right) = w_i \cdot \mu$$

Indeed, this is desired as only the  $(W - W_{f(\sigma(i))})/W$  fraction of the background edges are expected to be present between the communities. It follows that the ratio for cluster  $\ell \in [k]$  should be defined as follows:

$$\xi_\ell = \mu \left( \frac{W}{W - W_\ell} \right) \tag{4}$$

As a result, for the cluster graph  $G_{f(\sigma(i))} = \mathbf{G}(y_{f(\sigma(i))})$  corresponding to the cluster of vertex  $i$ , we let  $y_i = w_i - z_i = (1 - \xi_{f(\sigma(i))}) \cdot w_i$ .

As before, there exists a threshold  $\mu_1$  such that if  $\mu > \mu_1$ , then some value of  $\xi_\ell$  is greater than 1 and so the model cannot be applied. This time

$$\mu_1 = \min_{\ell \in [k]} \frac{W - W_\ell}{W} = 1 - \frac{\max_{\ell \in [k]} W_\ell}{W}$$

### 4.3 The comparison between the two variants (global vs. local)

Let us summarize the difference between the two approaches discussed above. Both of them preserve the same number of edges between clusters:  $\mu$  fraction of all edges are of this type (global property). The difference is how we split the degree of each vertex into internal degree and external one (local property). The original LFR model insists on each vertex keeping the same fraction of internal neighbors and the local version of our model (with  $k$  parameters  $\xi_i$ ) does this too. As a result, small clusters will be much denser than large clusters. Is it what we expect to happen in complex networks?

Suppose that two researchers have the same number of friends (say, 100) but belong to different communities. The first one, Bob, belongs to a small community (say, he is a mathematician doing some esoteric part of mathematics), the second one, Alice, is part of a large community (say, she is a data scientist). Suppose that 30% of friends of Alice do data science. Should we expect 30% of friends of Bob to be in his field? We believe the answer is no. It might be the case that there are less than 30 people around the world working on this subject! Coming back to the model, it seems that it makes more sense for the number of internal neighbors of a given vertex to be a function of the size of the cluster this vertex belongs to. As long as the probability that a given vertex is connected to another member of its cluster is larger than the probability of being adjacent to a random vertex in the whole graph, this vertex is a legit member of this cluster. This is what we propose in our first variant, the global version of our model (with only one parameter  $\xi$ ).

## 5. Experimental results

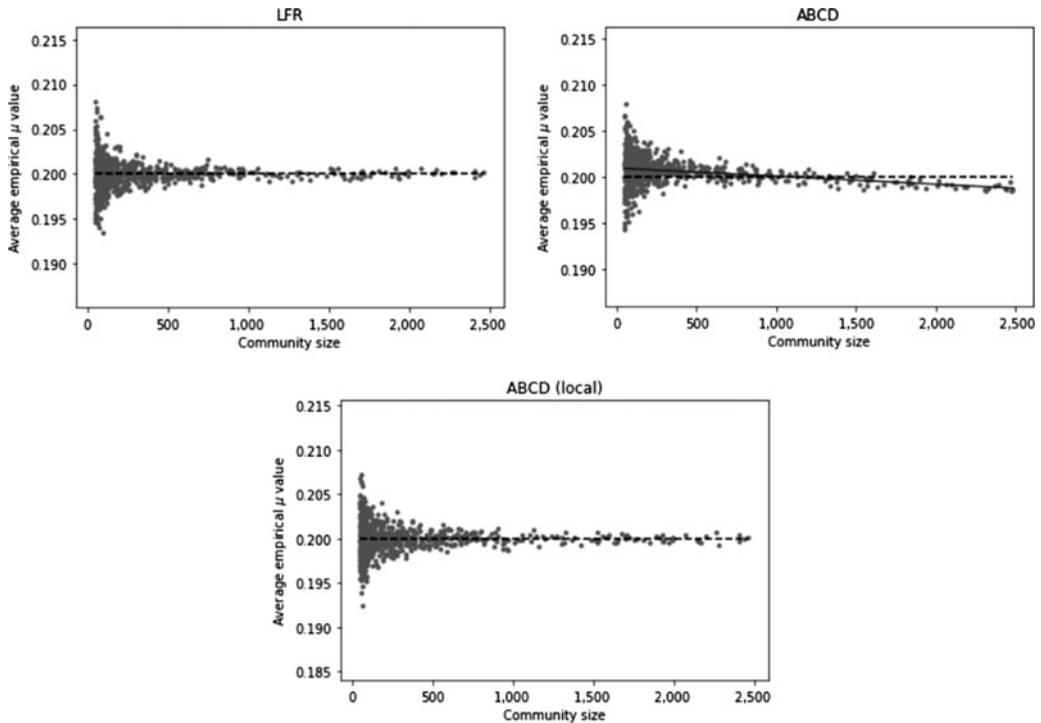
In this section, we compare ABCD and LFR benchmarks with respect to their respective mixing parameters (Subsection 5.1), the efficiency of the algorithms (Subsection 5.2), and properties of the graphs they generate (Subsection 5.3). In order to perform fair comparisons, we fix the LFR mixing parameter  $\mu$  and then derive the corresponding parameters for ABCD:  $\xi$  via Equation (2) for the global model, or  $\xi_i$ 's via Equation (4) for the local model.

Instructions how to reproduce Figures 2 and 4 can be found online.<sup>3</sup> We do not make the codes for producing the exact results given in Section 5.2 public, as they required some technical changes in comparison to publicly available implementations of the algorithms; in particular, we wanted to measure only the graph generation time without saving it to disk. Using the instructions presented on GitHub, that are based on end user versions of codes, allow to reproduce the presented results with high accuracy while minimizing complexity of execution of the experiments. Moreover, for Figures 2 and 4, we performed a slightly more exact comparison than presented on GitHub; that is, the same vertex degrees and community sizes are provided to all algorithms rather than generating them independently each time to make sure that the corresponding graph generation processes are compared on exactly the same data. However, the results are very similar to what is obtained with the simplified approach available online. The modified implementations that were used to generate figures in this paper can be made available upon request.

### 5.1 Global vs. local mixing parameters

We showed above that in the ABCD model (global variant), we expect a larger proportion of internal edges for larger communities. This implies a negative correlation between the community-wise mixing parameters  $\mu_i$  (i.e., the proportion of external edges for a given community) and the community sizes  $s_i$ . This is slightly different from the LFR model which tries to preserve the same community-wise  $\mu_i$  for each community. We showed that the ABCD model can be easily modified to mimic this property by defining community-wise parameters  $\xi_i$ , which we refer to as the local variant of the model.

In Figure 2, we illustrate this behavior for graphs with  $n = 250,000$  vertices and the same degree and community sizes distributions. We plot the mixing parameter  $\mu_i$  for each community as a



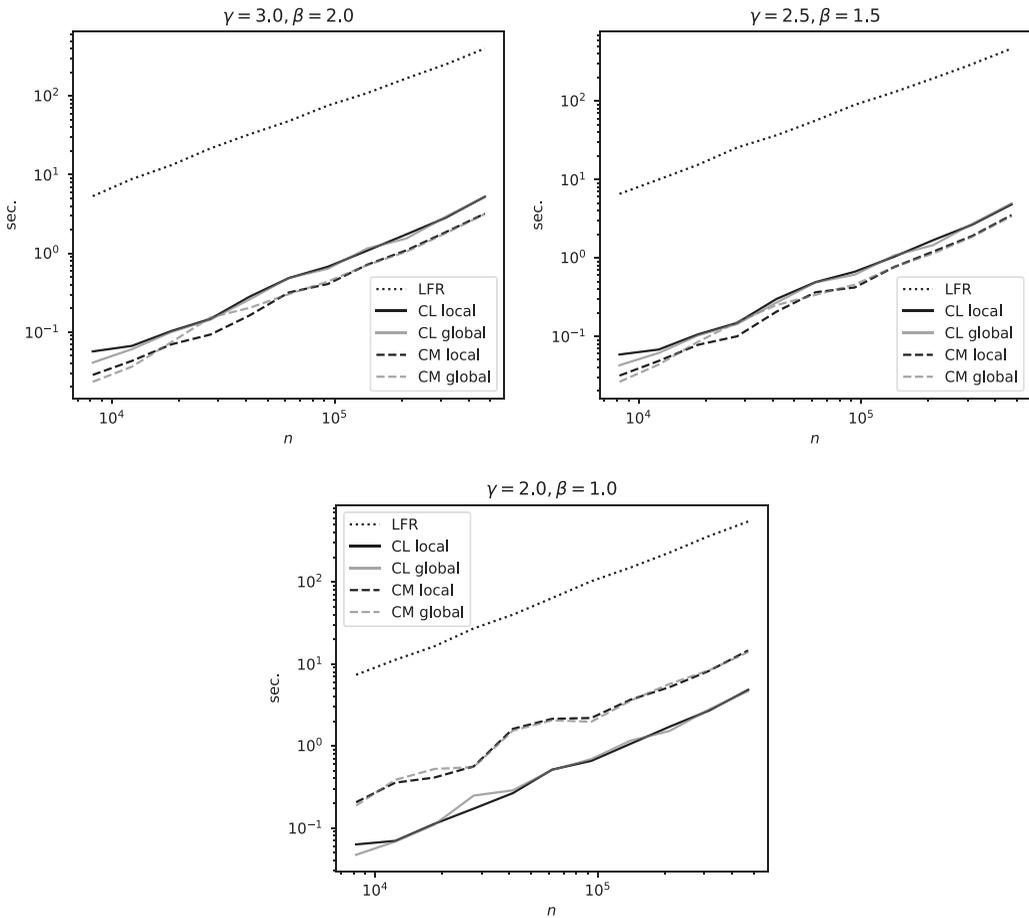
**Figure 2.** Comparing the behavior of graphs with  $n = 250,000$  vertices generated from three models: LFR, ABCD (with the configuration model), and its local variant. We used the same degree and community sizes distributions obtained with parameters:  $\bar{w} = 25$ ,  $w_{\max} = 1,500$ , and  $\gamma = 2.5$  for the degrees, and  $c_{\min} = 50$ ,  $c_{\max} = 2,500$ , and  $\beta = 1.5$  for the community sizes. We see that with LFR and ABCD (local variant), the expected community-wise mixing parameter  $\mu_i$  is constant while for the ABCD model, it decreases as a function of the community size.

function of its size. For comparison purpose, the dashed black horizontal line corresponds to the constant value  $\mu = 0.2$ . For the global variant of the ABCD model, we also display the regression line obtained by fitting the expected values for the  $\mu_i$  using the formula (3). In each case, due to rounding issue, we see more variability for small communities, as expected. For LFR, we see that the average value stays close to  $\mu = 0.2$ , while with the global variant of the ABCD, the value decreases with the community size matching the expected behavior quite well. Using the local version of ABCD, we see that we get similar behavior to the LFR model.

## 5.2 Efficiency comparison

In this subsection, we compare efficiency of the generating algorithms. All the results were obtained on a single thread of Intel Core i7-8550U CPU @ 1.80GHz, run under Microsoft Windows 10 Pro, and performing all computations in RAM. The computations for LFR were performed using the C++ language implementation<sup>4</sup> for smaller graphs (as it is a reference) and NetworKit<sup>5</sup> for larger graphs (as it is faster). For ABCD, the Julia 1.3 language implementation was used (Bezanson et al., 2017) in order to ensure high performance of graph generation, while keeping the size of the code base small. We tested all four combinations of the ABCD model (Chung-Lu vs. configuration model, and global vs. local variant).

In order for comparison to be fair, we first generated the degree distribution and the distribution of cluster sizes and then used it for all five algorithms tested (LFR and four combinations of ABCD). Only, the times to generate the corresponding graphs (single threaded, in memory,



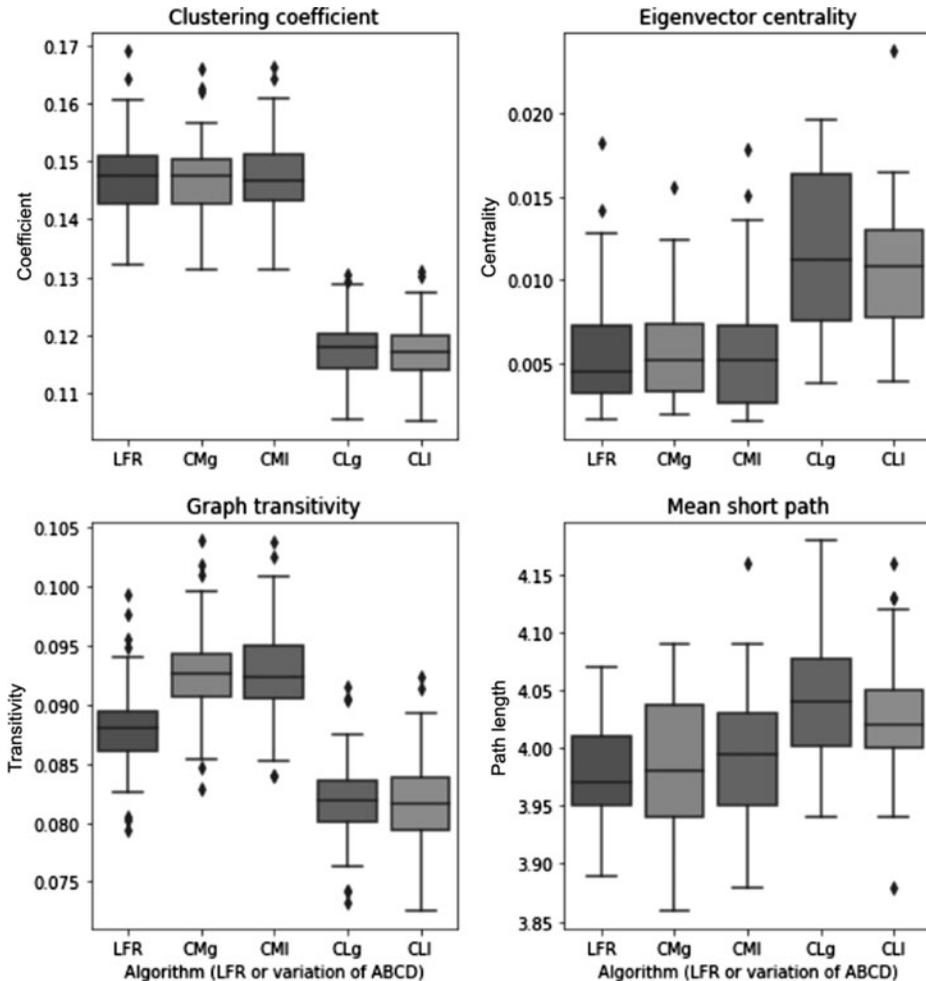
**Figure 3.** Generation times in seconds of the C++ LFR implementation and the ABCD models; CL indicates the Chung-Lu model and CM indicates the configuration model.

without storing the outcome on a hard drive) were measured, assuming the degree distribution and the distribution of cluster sizes are given—these steps are fast anyway.

The models were generated for  $\mu = 0.2$  (and its counterparts for  $\xi$ 's for ABCD—see Equations (2) and (4)), vertex average degree of 25 with maximum of 500, and community sizes varying between 50 and 1,000. Three different configurations of  $(\gamma, \beta)$  guiding the degree distribution and the distribution of cluster sizes are presented in Figure 3 ( $(\gamma, \beta) \in \{(3, 2), (2.5, 1.5), (2, 1)\}$ ). The number of vertices,  $n$ , spans from 8,192 to 472,392 and the timings are presented on the log-log scale. We present the results for one run of the reference C++ LFR model, whereas averages over five runs of the ABCD model are reported. The reason for running the ABCD generator more than once was that in most cases one run took less than a second, and so there was some non-negligible variability between run times due to external noise when performing the computing.

The conclusion is that the reference C++ LFR algorithm is of the order of 100 times slower than the one for the ABCD model; the largest ABCD was generated in a similar time to the smallest LFR. The worst scenario for ABCD is when the configuration model is used with low exponents of the two distributions (namely,  $\gamma = 2$  and  $\beta = 1$ ); in this case, ABCD is roughly 40 times faster.

In order to test an influence of various distributions of  $(\gamma, \beta)$  on the generation times of ABCD for larger networks, we performed benchmark tests for 10,000,000 vertices and switched the LFR



**Figure 4.** Comparing some properties for graphs generated with the LFR and ABCD benchmarks, using the same degree and community size distributions.

generator used to NetworkKit implementation. As before, the mixing parameter is fixed to  $\mu = 0.2$ , vertex average degree is 25 with maximum of 500, and community sizes vary between 500 and 10,000 (we increased the community sizes in comparison to the earlier test, as we now consider much larger number of vertices).

The time to generate the graphs using ABCD is of order of several minutes—see Table 1 where we vary parameters  $\gamma$  and  $\beta$ . In general, configuration model variant of ABCD is faster when local communities are not very dense. (See the rightmost plot in Figure 3 where we presented the case of very dense communities where Chung-Lu-based generator is faster.) Also we note that an increase in parameter  $\beta$  leads to longer run times. This is associated with the fact that small values of  $\beta$  produce several very large communities that attract heavy vertices. In such scenarios, the generators do not have to resolve too many collisions (multiple edges or self-loops), and so the algorithm terminates quickly. Each row in Table 1 is produced for the same of vectors  $\mathbf{w}$  and  $\mathbf{s}$  (but they vary across rows). The high variability of the results between rows indicates that the run time is quite sensitive to specific sampled values of  $\mathbf{w}$  and  $\mathbf{s}$ . Specifically, we have checked that the longest run times are to be expected, if there is a lot of heavy vertices sampled in  $\mathbf{w}$  and at the same time not many large clusters sampled in  $\mathbf{s}$ . Based on the results reported in Table 1, we also observe that for larger graphs, the NetworkKit implementation of the LFR generator is faster than

**Table 1.** Generation times in seconds of the ABCD model—four variants with  $n = 10,000,000$  vertices; CL indicates the Chung-Lu model and CM indicates the configuration model. Generation time of comparable graphs with LFR is presented using NetworKit package

$(\gamma, \beta)$	CL local	CL global	CM local	CM global	LFR NetworKit
(3.0, 2.0)	170	169	86	94	926
(3.0, 1.5)	141	184	81	74	922
(3.0, 1.0)	143	155	85	83	930
(2.5, 2.0)	228	203	105	118	1,072
(2.5, 1.5)	153	132	74	73	1,013
(2.5, 1.0)	116	116	67	67	1,099
(2.0, 2.0)	167	160	91	91	1,130
(2.0, 1.5)	132	132	79	77	1,114
(2.0, 1.0)	129	125	72	71	1,198

the reference C++ implementation of LFR but still over 10 times slower than the ABCD generator based on the configuration model.

In all the tests that we report in this paper, we concentrated on a single-threaded implementation of all the generators that run in RAM. We made this choice as our objective was to get a fair comparison of the time complexities of graph generation processes, unaffected by potential approaches to their parallelizations. Indeed, there are many architectures that could be used here, and the approach taken significantly affects timing (the three major options are multi-threading on a single machine, out of core distributed computing, and moving the graph generation to Graphical Processing Unit (GPU)/Tensor Processing Unit (TPU)). However, we would like to highlight that parallelization of ABCD generator is conceptually relatively straightforward. The major steps are the following. Cluster graphs  $G_i$  ( $i \in [k]$ ) can be generated completely independently, so their generation can be distributed with a large degree of flexibility to a given number of processors. This can be done using a dynamic load balancing of assigning jobs to workers. Generation of the background graph  $G_0$  can be achieved using standard procedures for parallel generation of Chung-Lu or configuration model graphs, as described, for example, in Section 6.1 and Section 6.2, respectively, in Penschuck et al. (2020). We currently work on various implementations of parallelization options that should be available at GitHub repository.<sup>6</sup> These are relatively straightforward adjustments, as the Julia language provides a native multi-threading and distributed computing support, and the code can be compiled to a GPU/TPU target.

### 5.3 Comparing graph properties

In this subsection, we compare graphs generated with the LFR and the ABCD benchmarks via some topology-based measures. We investigate the following graph statistics: clustering coefficient (the average vertex transitivity), eigenvector centrality, the global transitivity, and the average shortest paths length (approximated via sampling).

We generated graphs with 100,000 vertices, average degree 25, maximum degree 500, and power law exponent  $\gamma = 2.5$ ; for the community sizes, we used power law exponent  $\beta = 1.5$  with sizes between 50 and 2,000. The mixing parameter for LFR is set to  $\mu = 0.2$  and, in order to compare similar graphs, for the ABCD algorithm, we derive  $\xi$  from (2) and the  $\xi_i$ 's from (4) (for the local model). In Figure 4, we report the distribution of the graph properties obtained by generating 30 graphs each using LFR as well as 4 variations of ABCD, namely:

- CMg: configuration model with global  $\xi$ ,
- CMl: configuration model with local  $\xi_i$ 's,
- CLg: Chung-Lu model with global  $\xi$ ,
- CLl: Chung-Lu model with local  $\xi_i$ 's.

The results of these experiments show high similarity of graphs generated with LFR and ABCD, in particular, when the configuration model is used. Indeed, some graph parameters that are sensitive with respect to the degree distribution (such as clustering coefficient) are not well preserved for the Chung-Lu variant of the model, which is natural and should be expected. Having said that, all graph parameters we evaluated are relatively well aligned.

## 6. Conclusion and future work

The paper has two interrelated angles, theoretical and practical. We tried to define the model in as easy and natural way as possible. As a result, from the theoretical point of view, using abundant tools from the theory of random graphs, we plan to investigate an asymptotic behavior of the ABCD model. As explained in Subsection 2.2, this is not only interesting from pure math point of view but also might be important for practitioners. Finally, we plan to generalize the model and add geometry into the model. This would allow, for example, for overlapping and hierarchical communities.

From practical point of view, the implementation we propose in this paper is single-threaded which we believe is sufficient for generating small to medium size graphs. Indeed, it usually takes under 1 minute to generate a graph consisting of several millions of vertices; in fact, the timing of the process of generating an ABCD graph is of comparable magnitude as the time needed to save it to the hard drive later (on a typical server). However, in order to deal with enormous graphs containing billions of vertices, users might need out-of-core distributed implementation of the ABCD algorithm. In Section 2.1, we have commented on how this could be achieved in future work. Independently, it would be interesting to perform more extensive experiments with ABCD (and, in particular, compare it to LFR) when the generated graphs are used to test algorithms that require knowledge of ground truth community structure (such as clustering algorithms). We think that performing such experimental comparison is an important follow-up to this theoretical paper.

**Financial support.** The paper is financed by the Polish National Agency for Academic Exchange, grant “International Academic Partnerships”, agreement no. PPI/APM/2018/1/00037.

**Conflict of interest.** None.

## Notes

- 1 [https://github.com/eXascaleInfolab/LFR-Benchmark\\_UndirWeight0vp/](https://github.com/eXascaleInfolab/LFR-Benchmark_UndirWeight0vp/).
- 2 <https://github.com/bkamins/ABCDGraphGenerator.jl/>.
- 3 <https://github.com/bkamins/ABCDGraphGenerator.jl/tree/master/instructions>.
- 4 [https://github.com/eXascaleInfolab/LFR-Benchmark\\_UndirWeight0vp/](https://github.com/eXascaleInfolab/LFR-Benchmark_UndirWeight0vp/).
- 5 <https://networkit.github.io/>.
- 6 <https://github.com/bkamins/ABCDGraphGenerator.jl>.

## References

- Aiello, W., Bonato, A., Cooper, C., Janssen, J. C. M., & Prałat, P. (2008). A spatial web graph model with local influence regions. *Internet Mathematics*, 5(1), 175–196.
- Albert-László Barabási, R. A. (1999). Emergence of scaling in random networks. *Science*, 286(5439), 509–512.
- Bae, S.-H., & Howe, B. (2015). Gossipmap: A distributed community detection algorithm for billion-edge directed graphs. In *SC'15*, vol. 27 (pp. 1–12). ACM.
- Barabási, A.-L. (2016). *Network science*. Cambridge: Cambridge University Press.
- Bender, E. A. & Canfield, E. R. (1978). The asymptotic number of labeled graphs with given degree sequences. *Journal of Combinatorial Theory, Series A*, 24(3), 296–307.

- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 69, 65–98.
- Bollobás, B. (1980). A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. *European Journal of Combinatorics*, 1, 311–316.
- Buzun, N., Korshunov, A., Avanesov, V., Filonenko, I., Kozlov, I., Turdakov, D., & Kim, H. (2014). Egolp: Fast and distributed community detection in billion-node social networks. In *IEEE ICDM mining workshop* (pp. 533–540).
- Chakrabarti, D., Zhan, Y., & Faloutsos, C. (2004). R-mat: A recursive model for graph mining. In *proceedings of the 2004 SIAM international conference on data mining* (pp. 442–446). SIAM.
- Chung, F., & Lu, L. (2006). *Complex graphs and networks*. Providence, RI: American Mathematical Society.
- Dao, V. L., Bothorel, C., & Lenca, P. (2020). Community structure: A comparative evaluation of community detection methods. *Network Science*, 8(1) 1–41.
- Emmons, S., Kobourov, S. G., Gallant, M., & Börner, K. (2016). Analysis of network clustering algorithms and cluster quality metrics at scale. *Plos One*, 11, 1–18.
- Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486, 75–174.
- Funke, D., Lamm, S., Sanders, P., Schultz, C., Strash, D., & von Looz, M. (2018). Communication-free massively distributed graph generation. In *international parallel and distributed processing symposium (IPDPS)*.
- Girvan, M., & Newman, M. E. J. (2002). Community structure in social and biological networks. *Proceedings of the National Academy of Sciences*, 99, 7821–7826.
- Gkantsidis, C., Mihail, M., & Zegura, E. W. (2003). The markov chain simulation method for generating connected power law random graphs. In *ALENEX'03* (pp. 16–25). SIAM.
- Greenhill, C., & Sfragara, M. (2018). The switch Markov chain for sampling irregular graphs and digraphs. *Theoretical Computer Science*, 719, 1–20.
- Hamann, M., Meyer, U., Penschuck, M., Tran, H., & Wagner, D. (2018). I/o-efficient generation of massive graphs following the lfr benchmark. *Journal of Experimental Algorithmics*, 23, 2.5:1–2.5:33.
- Janson, S. (2020). Random graphs with given vertex degrees and switchings. *Random Structures Algorithms*, 57, 3–31.
- Kaminski, B., Poulin, V., Pralat, P., Szufel, P., & Theberge, F. (2019). Clustering via hypergraph modularity. *Plos One*, 14, e0224307.
- Kaminski, B., Pralat, P., & Theberge, F. (2021). Community detection algorithm using hypergraph modularity. In *Proceedings of the 9th international conference on complex networks and their applications, Studies in Computational Intelligence 943* (pp. 152–163). Springer.
- Krioukov, D. V., Papadopoulos, F., Kitsak, M., Vahdat, A., & Boguñá, M. (2010). Hyperbolic geometry of complex networks. *Physical Review E*, 82(036106).
- Kolda, T. G., Pinar, A., Plantenga, T., & Seshadhri, C. (2014). A scalable generative graph model with community structure. *SIAM Journal on Scientific Computing*, 36, C424–C452.
- Lancichinetti, A., & Fortunato, S. (2009). Benchmark graphs for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Physical Review E*, 80, 016118.
- Lancichinetti, A., Fortunato, S., & Radicchi, F. (2008). Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78.
- Milo, R., Kashtan, N., Itzkovitz, S., Newman, M. E. J., & Alon, U. (2003). On the uniform generation of random graphs with prescribed degree sequences. arxiv:cond-mat/0312028.
- Newman, M. (2010). *Networks: An introduction*. Oxford: Oxford University Press.
- Newman, M. E. J. & Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E*, 69, 26–113.
- Penschuck, M., Brandes, U., Hamann, M., Lamm, S., Meyer, U., ... Schulz, C. (2020). *Recent advances in scalable network generation*. Tech. rept. <https://arxiv.org/abs/2003.00736>.
- Prokhorenkova, L. O., Pralat, P., & Raigorodskii, A. (2017). Modularity of complex networks models. *Internet Mathematics*. doi:10.24166/im.12.2017
- Ray, J., Pinar, A., & Seshadhri, C. (2012). Are we there yet? when to stop a markov chain while generating random graphs. In *WAW'12. Lecture Notes in Computer Science*. Springer (pp. 153–164).
- Seshadhri, C. K., Tamara, G., & Pinar, A. (2012). Community structure and scale-free collections of erdős-rényi graphs. *Physical Review E*, 85, 056109.
- Slota, G. M., Berry, J., Hammond, S. D., Olivier, S., Phillips, C., & Rajamanickam, S. (2019). Scalable generation of graphs for benchmarking HPC community-detection algorithms. In *IEEE international conference for high performance computing, networking, storage and analysis (SC)*.
- Staudt, C. L., Hamann, M., Gutfraind, A., Safro, I., & Meyerhenke, H. (2017). Generating realistic scaled complex networks. *Applied Network Science*, 2(36), 1–29.
- Viger, F., & Latapy, M. (2005). Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In L. Wang (Eds.), *Computing and combinatorics* (pp. 440–449). Berlin, Heidelberg: Springer.

- West, D. B. (2001). *Introduction to graph theory* (2nd ed.). Upper Saddle River: Prentice Hall.
- Winlaw, M., DeSterck, H., & Sanders, G. (2015). *An in-depth analysis of the chung-lu model*. Tech. rept. LLNL-TR-678729. Lawrence Livermore Technical Report, doi: 10.2172/1239211.
- Wormald, N. C. (1984). Generating random regular graphs. *Journal of Algorithms*, 5(2), 247–280.
- Zeng, J., & Yu, H. (2016). A study of graph partitioning schemes for parallel graph community detection. *Parallel Computing*, 58, 131–139.

## Appendix: Algorithm Pseudo-Code

---

### Algorithm 1: Generation of the community sizes

---

```

1 INPUT:  $n$ : number of nodes,  $\beta$ : community sizes power law exponent,  $c_{\min}$ : min community
  size and  $c_{\max}$ : max community size;  $I_{\max}$  (optional, default to 100);
2 let  $s_{best} := \infty$  and  $I := 0$ ;
3 initialize empty list  $S_{best}$ ;
4 repeat
5   check if it is possible to generate the required cluster sizes; throw an error if it is not
  possible;
6   let  $s := 0$ ;
7   initialize empty list  $X$ ;
8   repeat
9     Sample value  $x$  from truncated discrete power law distribution with parameter  $\beta$ ,
    restricted to the interval  $[c_{\min}, c_{\max}]$  and store in  $x$  in  $X$ ;
10    let  $s := s + x$ ;
11  until  $s \geq n$ ;
12  if  $s = n$  then
13    OUTPUT: list of community sizes  $X$ ;
14    exit;
15  else
16    if  $s < s_{best}$  then
17      let  $s_{best} := s$  and  $S_{best} := X$ ;
18    end
19  end
20   $I = I + 1$ 
21 until  $I > I_{\max}$ ;
22 Truncate  $S_{best}$  and update  $s_{best}$  accordingly if needed (it might be impossible to find
  corrections that produce admissible community sizes in corner cases; this may lead to
   $s_{best} < n$  case).;
23 repeat
24   In random order cyclically process elements of  $S_{best}$ ;
25   If  $s_{best} > n$  decrease values sequentially by 1 unless some element is  $c_{\min}$ ; decrease  $s_{best}$  by
  one.;
26   If  $s_{best} < n$  increase values sequentially by 1 unless some element is  $c_{\max}$ ; increase  $s_{best}$  by
  one.;
27 until  $s_{best} = n$ ;
28 OUTPUT: list of community sizes  $S_{best}$ ;

```

---

**Algorithm 2:** Generation of the degree sequence

---

```

1 INPUT:  $n$ : number of nodes,  $w_{\min}$ : min degree,  $w_{\max}$ : max degree,  $\gamma$ : degree power law
  exponent;  $I_{\max}$  (optional, default to 100);
2 initialize empty list  $W$ ;
3 let  $I := 0$ ;
4 repeat
5   repeat
6     sample value  $w$  from truncated discrete power law distribution with parameter  $\gamma$ ,
     restricted to the interval  $[w_{\min}, w_{\max}]$  and add  $w$  to  $W$ ;
7   until  $|W| = n$ ;
8   if sum of degrees in  $W$  is even then
9     OUTPUT: list of degrees  $W$ 
10  end
11  let  $I := I + 1$ 
12 until  $I > I_{\max}$ ;
13 decrease the largest value in  $W$  by 1 to make the sum of degrees even;
14 OUTPUT: list of degrees  $W$ 

```

---

**Algorithm 3:** Assign nodes with degree sequence  $W$  to communities with sizes  $S$ . Algorithm given for global ABCD. For local version of ABCD, use cluster-local  $\xi_i$ 's instead of  $\xi$ 


---

```

1 INPUT: Degree sequence  $W$  on  $n$  nodes, community sizes  $S$  with  $|S| = k$  and parameter  $\xi$ 
  (LFR-style  $\mu$  can be supplied instead);
2 sort nodes from largest to smallest degrees in  $W$ :  $w_1 \geq \dots \geq w_n$ ;
3 sort communities from largest to smallest sizes in  $S$ :  $s_1 \geq \dots \geq s_k$ ;
4 initialize number of free spots in each community:  $f_i := s_i$ ,  $1 \leq i \leq k$ ;
5 initialize empty lists  $S_1, \dots, S_k$ ;
6 for  $1 \leq i \leq n$  do
7   find max value in  $1 \leq t \leq k$  s.t.  $w_i < (1 - \xi\phi)s_t$  where  $\phi$  is defined in (1)a;
8   pick random  $1 \leq j \leq t$  proportional to  $f_1, \dots, f_t$ ;
9   assign vertex  $i$  to community  $j$  by adding it to  $S_j$ ;
10  let  $f_j := f_j - 1$ ;
11 end
12 OUTPUT: community assignment of vertices:  $S_1, \dots, S_k$ 

```

---

<sup>a</sup>if  $\mu$  is specified instead of  $(1 - \xi\phi)s_t$ , we use  $(1 - \mu)s_t$ .

---

**Algorithm 4:** ABCD with Chung-Lu Model. Algorithm given for global ABCD. For local version of ABCD, use cluster-local  $\xi_i$ 's instead of  $\xi$

---

- 1 INPUT: Community assignment  $S_1, \dots, S_l$  from Algorithm 3 for  $n$  vertices, degree sequence  $W$  from Algorithm 2, and parameter  $\xi$  (LFR-style  $\mu$  can be supplied instead);
  - 2 if  $\mu$  was given, compute  $\xi$ ;
  - 3 **for**  $1 \leq i \leq k$  **do**
  - 4 let  $W_i$ , the sum of the degrees of all vertices in  $S_i$ ;
  - 5 randomly sample  $\lfloor (1 - \xi)W_i/2 \rfloor$  edges within  $S_i$  where each vertex is selected proportionally to its internal degree; duplicate edges and self-loops are skipped;
  - 6 **end**
  - 7 let  $s := (\text{sum}(W) - \text{sum}(\forall i: W_i))/2$ ; sample  $s$  edges randomly where each vertex is selected proportionally to its external degree; duplicate edges and self-loops are skipped;
  - 8 OUTPUT: ABCD graph (list of edges generated);
- 

---

**Algorithm 5:** ABCD with configuration model. Algorithm given for global ABCD. For local version of ABCD, use cluster-local  $\xi_i$ 's instead of  $\xi$

---

- 1 INPUT: Community assignment  $S_1, \dots, S_l$  from Algorithm 3 for  $n$  vertices, degree sequence  $W$  from Algorithm 2, and parameter  $\xi$  (LFR-style  $\mu$  can be supplied instead);
  - 2 if  $\mu$  was given, compute  $\xi$ ;
  - 3 **for**  $1 \leq i \leq k$  **do**
  - 4 for each vertex in  $S_i$ , given its degree  $w$ , assign internal  $w_{int} := \lfloor (1 - \xi) \cdot w \rfloor$ ;
  - 5 if the sum of all  $w_{int}$  is odd, adjust highest degree node randomly to make it even;
  - 6 wire the vertices in  $S_i$  randomly according to their values  $w_{int}^a$ ;
  - 7 re-wire duplicated edges and self-loops;
  - 8 if re-wiring fails update the  $w_{int}$  to achieved values
  - 9 **end**
  - 10 compute the external degree for each vertex as  $w_{ext} := w - w_{int}$ ;
  - 11 wire the vertices randomly according to their values  $w_{ext}$  (global model);
  - 12 re-wire duplicated edges and self-loops only considering edges in the global model;
  - 13 OUTPUT: ABCD graph (list of edges generated);
- 

<sup>a</sup>other methods can be used here; for example, high degree nodes can be wired first to limit the collisions, or algorithms such as Viger & Latapy Viger & Latapy (2005) which yields simple graphs can be used