Properties and Performance of the ABCDe Random Graph Model with Community Structure

Bogumił Kamiński* Tomasz Olczak† Bartosz Pankratz‡ Paweł Prałat§
François Théberge¶

September 10, 2022

Abstract

In this paper, we investigate properties and performance of synthetic random graph models with a built-in community structure. Such models are important for evaluating and tuning community detection algorithms that are unsupervised by nature. We propose **ABCDe**—a multi-threaded implementation of the **ABCD** (Artificial Benchmark for Community Detection) graph generator. We discuss the implementation details of the algorithm and compare it with both the previously available sequential version of the **ABCD** model and with the parallel implementation of the standard and extensively used **LFR** (Lancichinetti–Fortunato–Radicchi) generator. We show that **ABCDe** is more than ten times faster and scales better than the parallel implementation of **LFR** provided in `NetworKit`. Moreover, the algorithm is not only faster but random graphs generated by **ABCD** have similar properties to the ones generated by the original **LFR** algorithm, while the parallelized `NetworKit` implementation of **LFR** produces graphs that have noticeably different characteristics.

1 Introduction

The standard and extensively used method for generating artificial networks that have community structure is the **LFR** (Lancichinetti–Fortunato–Radicchi) graph generator [1]. Despite the fact that this is clearly a very good model, it is known to have some scalability limitations and it is challenging to analyze it theoretically. Moreover, the mixing parameter μ, 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, see [2] for a detailed discussion.

An alternative random graph model with community structure and power-law distribution for both degrees and community sizes is the **A**rtificial **B**enchmark for **C**ommunity **D**etection graph

*Decision Analysis and Support Unit, SGH Warsaw School of Economics, Warsaw, Poland; e-mail: `bkamins@sgh.waw.pl`

†Decision Analysis and Support Unit, SGH Warsaw School of Economics, Warsaw, Poland; e-mail: `tolczak@gmail.com`

‡Decision Analysis and Support Unit, SGH Warsaw School of Economics, Warsaw, Poland; e-mail: `bartosz.pankratz@ryerson.ca`

§Department of Mathematics, Toronto Metropolitan University, Toronto, ON, Canada; e-mail: `pralat@ryerson.ca`

¶Tutte Institute for Mathematics and Computing, Ottawa, ON, Canada; email: `theberge@ieee.org`

(**ABCD**). In [2] it is shown that the new model is fast, simple, and can be easily tuned to allow the user to make a smooth transition between the two extremes: pure (disjoint) communities and random graph with no community structure. Moreover, in [3] the modularity function of **ABCD** is theoretically analyzed and it is confirmed that its asymptotic behaviour is consistent with simulations on smaller experimental graphs. (The modularity function is, arguably, the most important graph property of networks in the context of community detection. Indeed, the modularity function is often used to measure the presence of community structure in networks. It is also used as a quality function in many community detection algorithms, including the widely used Louvain algorithm [4].) On the other hand, because of its similarity to **LFR**, **ABCD** can be expected to preserve most of its natural graph properties and parameters. We verify this claim in this paper using simulation. Hence, **ABCD** (or **ABCDe** introduced and discussed in this paper) may successfully replace the **LFR** generator when scalability becomes a bottleneck.

In this paper we introduce the implementation of the **ABCD** generator that uses multiple threads for processing, **ABCDe**. The goal of parallelizing sequential **ABCD** is to speed up computations and thus allow for handling of larger graphs (having more nodes or being more dense). We describe the challenges of parallelization of this algorithm and the approach we took to ensure both its performance and reproducibility of generated graphs. We analyze the **ABCDe** properties using simulation, taking into account two important aspects. First, we analyze the properties of the graphs that it generates and compare them to graphs generated by the **LFR** algorithm under matching parameterization. Next, we analyze the speed of **ABCDe** against the single threaded (sequential) implementation of **ABCD** and selected implementations of the **LFR** generator. The two selected implementations are: the original **LFR** algorithm [1] and its fast implementation, that optionally uses multiple threads, provided in the `NetworKit` package [5][1].

2 The Design of the ABCDe Generator

2.1 Building Blocks of the Algorithm

As a preliminary information let us start with introducing two building blocks of the **ABCD** model: configuration model and Chung-Lu model. Let $\mathbf{w} = (w_1, \ldots, w_n)$ be any vector of n non-negative integers. Our goal is to be able to build two types of random graphs on n nodes, the first one will have a given degree sequence $\mathbf{w}$ (configuration model) and the second one will only have the expected degree sequence $\mathbf{w}$ (Chung-Lu model).

A random multi-graph $\mathcal{M}(\mathbf{w})$ with a given degree sequence known as the **configuration model** (sometimes called the **pairing model**) was first introduced by Bollobás [6]. Assuming that $W := \sum_{i=1}^{n} w_i$ is even, let us consider W points partitioned into n labelled buckets $v_1, \ldots, v_n$; 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 nodes are the buckets $v_1, \ldots, v_n$, 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 $\mathcal{M}(\mathbf{w}) = G(P)$.

[1] https://networkit.github.io/index.html

In the Chung-Lu model [7] that generates graph $\mathcal{C}(\mathbf{w})$ on the node set $[n] = \{1, \ldots, n\}$, each set $e = \{i, j\}$, $i, j \in [n]$, is independently sampled as an edge with probability given by:

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

(Let us mention about one technical assumption. Note that it might happen that $\Pr(i,j)$ is greater than one and so it should really be regarded as the expected number of edges between i and j; for example, as suggested in Newman [16], one can introduce a Poisson-distributed number of edges with mean $\Pr(i,j)$ between each pair of nodes i, j. However, since typically the maximum degree Δ satisfies $\Delta^2 \leq 2|E|$ it rarely creates a problem and so we may assume that $\Pr(i,j) \leq 1$ for all pairs.)

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

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

In summary, both models are similar. The difference between them is that **configuration model** ensures that the required node degree sequence is reproduced exactly, while **Chung-Lu model** produces this degree sequence in expectation.

2.2 Structure of the ABCD Graph

In this section, we briefly discuss the **ABCD** models; details can be found in [2] or in [3]. As in **LFR**, for a given number of nodes n, we start by generating a power law distribution both for the degrees and community sizes. Those are governed by the power law exponent parameters (γ, β). We also provide additional information to the model, again as it is done in **LFR**, namely, the average and the maximum degree, and the range for the community sizes. The user may alternatively provide a specific degree distribution and/or community sizes.

For each community, we generate a random *community* subgraph on the nodes from a given community using either the **configuration model** (see [6]) which preserves the exact degree distribution, or the **Chung-Lu model** (see [7]) which preserves the expected degree distribution. On top of it, we independently generate a *background* random graph on all the nodes that is generated the same way as the community graphs. Everything is tuned properly so that the degree distribution of the union of all graphs follows the desired degree distribution (only in expectation in the case of the Chung-Lu variant). In particular, the mixing parameter ξ guides the proportion of edges which are generated via the background graph. In the two extreme cases, when $\xi = 1$ the graph has no community structure while if $\xi = 0$, then we get disjoint communities. In order to generate simple graphs, we may have to do some re-sampling or edge re-wiring, which as described in [2]. This two-step process is similar to the highly scalable **BTER** model [8]. (Similarly to **LFR** and **ABCD**, **BTER** generates graphs with a given degree distribution but the main objective is different: it aims to preserve per-degree clustering coefficients. In particular, this model does not have the same type of community structure as in **LFR** and **ABCD**.)

During this process, larger communities will additionally get some more internal edges due to the background graph. As argued in [2], this "global" variant of the model is more natural and

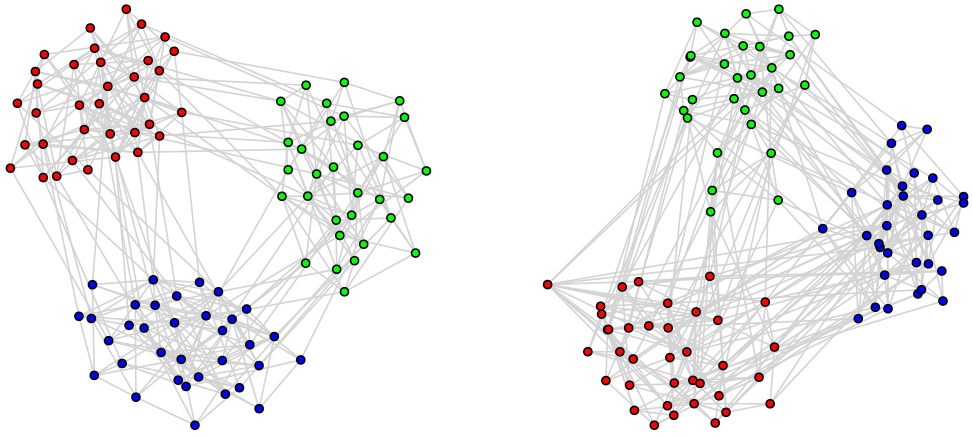

Figure 1: Two examples of **ABCD** graphs with low level of noise ($\xi = 0.2$, left) and large level of noise ($\xi = 0.4$, right).

so we recommend it. However, in order to provide a variant where the expected proportion of internal edges is exactly the same for every community (as it is done in **LFR**), we also provide a "local" variant of **ABCD** in which the mixing parameter ξ is automatically adjusted for every community. Both variants preserve the same number of edges between communities. The difference is how the degree of each node is split into internal and external degree. The **LFR** model, as well as our local variant of the **ABCD** model, keep the same fraction of neighbours to be internal neighbours for all nodes, regardless how large the community this node belongs to is. As a result, small communities become much denser than large communities. On the other hand, in the global variant of the **ABCD** model, the internal degree naturally depends on the size of the associated community.

Two examples of **ABCD** graphs on $n = 100$ nodes are presented in Figure 1. Degree distribution was generated with power law exponent $\gamma = 2.5$ with minimum and maximum values 5 and 15, respectively. Community sizes were generated with power law exponent $\beta = 1.5$ with minimum and maximum values 30 and 50, respectively; communities are shown in different colours. The global variant and the configuration model was used to generate the graphs. The left plot has the mixing parameter set $\xi = 0.2$ while the "noisier" graph on the right plot has the parameter fixed to $\xi = 0.4$.

In this paper, we compare both "global" and "local" variants of the **ABCD** model (using the configuration model to generate communities as well as the background graph) against the classical **LFR** model.

2.3 Approach to Parallelization of the ABCD Generator

The original **ABCD** model was implemented using a sequential algorithm (Algorithm 1). In [2], we discuss theoretical complexity of **ABCD** and **LFR** for this scenario.

Let us now switch to issues of parallelization of the process of generating **ABCD** graphs. The model, being a union of community graphs and the background graph, is naturally structured for concurrent processing. Since community graphs are disjoint, they can be generated inde-

pendently of each other. Typically they are numerous and their individual sizes are relatively small when compared to the whole graph. Taken together they constitute a set of sufficiently granular tasks for efficient parallel processing. Hence, in **ABCDe** (a multi-threaded version of **ABCD** generator) we distribute community graphs for parallel generation among the available CPU threads.

The main challenge for the design of parallelized **ABCDe** algorithm is posed by generation of the background graph. In the sequential version of the algorithm, the background graph is generated in the final stage of processing, after all the community graphs are generated. This order of processing enables detection of potential collisions, that is, duplicate edges in the background graph and one of the community graphs. Since the background graph shares nodes with all the community graphs such collisions may arise and must be avoided to ensure the final graph to be simple.

Preserving the same processing sequence in the parallel algorithm would have deteriorating effect on performance as often the background graph is chosen to be so large that its generation consumes a non-negligible fraction of the overall processing time. Postponing generation of the background graph only after all the community graphs are generated would significantly increase the fraction of time spend in sequential processing in overall processing time thus limiting speedup from parallel processing as governed by Amdahl's law [24].

To work around this serialization bottleneck, one could parallelize generation of the background graph itself. Unfortunately, this approach is challenging by nature of a single graph generation algorithm. Recall that we defined the **configuration model** graph by $G(P) = \mathcal{M}(\mathbf{w})$ where a pairing P is sampled uniformly at random from the family of all pairings of $W := \sum_{i=1}^{n} w_i$ points. This sampling is implemented in the following way. First, a random permutation $p := \{1, \ldots, m\} \mapsto \{1, \ldots, m\}$ where $m := W$ is generated using Fisher–Yates shuffle [25]. Then, a pairing $P_p := (p_1, \ldots, p_{\frac{m}{2}}) \mapsto (p_{\frac{m}{2}+1}, \ldots, p_m)$ is obtained and a multi-graph $G(P_p)$ is produced.

Fisher–Yates algorithm [25] is inherently sequential and its potential parallelization would involve high performance penalty due to thread synchronization thus cancelling gains from parallelization. Therefore, in **ABCDe** the second best approach was taken where, in contrast to sequential **ABCD**, the background graph is generated *in parallel* with community graphs. To enable this change, generation of the background graph was split into two phases. In the first phase, composed of tasks independent of community graph generation, the disjoint background graph is generated and any internal self-loops or parallel edges rewired to ensure the graph to be simple (steps 6 and 7 in Algorithm 2). Note that this phase is indistinguishable from generation of community graphs. In the second phase, after all the community graphs *and* the background graph were generated, individual graphs are merged to form the final **ABCDe** graph and any parallel edges resulting from the merger are rewired, if present (steps 9 and 10 in Algorithm 2).

To ensure well-balanced distribution of work among available threads and to minimize total processing time, we apply the following task allocation policy. Typically, the background graph is the largest one in the ensemble and its generation is the most time consuming. It is then the first graph picked up for processing by the first available thread so its generation could start as soon as possible. The remaining set of community graphs is placed in a random order into a FIFO queue and consumed for processing by a pool of available threads following a classical producer-consumer pattern. This generates a uniform workload in expectation. Note that this policy does not prevent unintended serialization in case of an extremely large background graph,

as it would still be processed by a single thread only. In this case benefits of parallelization could be limited. Fortunately, this happens only for ξ close to 1, which is not normally used in practical applications (usually, ξ less than 0.5 is used).

Let us also make a short remark on the optimal number of threads. **ABCDe** will use *all* the threads available for Julia process and the number of Julia threads can be specified when Julia process is started. As a rule of thumb, for optimal **ABCDe** performance one should use the number of threads equal to the number of *physical* (not logical) cores as **ABCDe** is a memory intensive algorithm. Increasing number of threads beyond this point will usually not make noticeable improvement or can even deteriorate performance as excess threads would be forced to share the same memory bus.

Finally, a note should be taken on reproducibility of results. The challenge is that generation of each graph is an independent task and both the order and thread allocation is unspecified and decided only at run-time by the operating system scheduler. This would lead to a situation where running the algorithm twice for the same parameterization could potentially produce different graphs. In order to solve this issue, each task is associated with a random number generator seed before any graph generation is started. This way the stream of pseudo-random numbers is ensured to be reproducible independently of task processing order, number of threads, and task to thread allocation.

In summary, we parallelize generation of the **ABCD** graph in the following way:

Step 1: sequentially assign nodes to communities and split their degrees into community and background graph;

Step 2: sequentially put all community graphs and background graph in a queue of tasks; assign a random number generator seed to each task;

Step 3: using parallel map algorithm, generate all community graphs and background graph independently;

Step 4: sequentially create a union of all graphs;

Step 5: sequentially resolve all conflicts between the background graph and community graphs to ensure that the resulting graph is simple.

Steps 1, 2, 4, and 5 are done sequentially. Note that steps 1, 2, and 4 are computationally cheap so that there would be no noticeable benefit of running them using a parallel algorithm. Step 5 could, in general, be potentially expensive. Fortunately, for large and sparse graphs the number of conflicts that need to be resolved in this step is small and does not significantly affect the overall run-time of the algorithm. Step 3, the most expensive part of the algorithm, is executed using multi-threading. In our experiments, we verify how increasing number of threads affects the run-time of the whole algorithm.

In general, the computational complexity of the **ABCDe** algorithm is linear in the number of edges generated, similarly to the sequential **ABCD** algorithm, as discussed in [2]. However, as it will be seen in Section 4, the scaling of the algorithm with increasing n has a growing constant factor. The reason for this behaviour is twofold. The first is theoretical—increasing n changes degre distribution and community size distribution of the generated graph. These changes lead to changes of cost of collision resolution process. The second is technical—as graph size increases, the number of operations that are performed in CPU cache decreases. For large

graphs most operations end up to be cache misses. Since reading data from CPU cache is much faster than fetching data from a new area of RAM, again the constant factor in the algorithm increases.

Input: vector of node degrees
Output: list of edges
for *node* ∈ *nodes* **do**
    assign node to a community;
    split the degree among the community and the background graph;
end
for *community* ∈ *communities* **do**
    generate a community graph;
    rewire self-loops and parallel edges to ensure the community graph is simple;
end
generate the background graph;
rewire self-loops and parallel edges to ensure the background graph is simple;
create union of community graphs and the background graph;
rewire parallel edges to ensure the output graph is simple;
return edges;

Algorithm 1: ABCD (sequential)

Input: vector of node degrees
Output: list of edges
for *node* ∈ *nodes* **do**
    assign node to a community;
    split the degree among the community and the background graph;
end
for *graph* ∈ *community graphs* ∪ {*background graph*} **do in parallel**
    generate a graph;
    rewire self-loops and parallel edges to ensure the graph is simple;
end
create union of community graphs and the background graph;
rewire parallel edges to ensure the output graph is simple;
return edges;

Algorithm 2: ABCDe (parallel)

3 Properties of the Generated Graphs

In this section, we present the results of several experiments comparing the properties of graphs generated using the sequential **ABCD** model and parallelized **ABCDe** model (both global and local variants) against the original **LFR** implementation [1] as well as its fast `NetworKit` implementation [5]. Note that the **ABCD** and **ABCDe** local variant of our algorithm was developed with the goal to reproduce the behaviour of the original **LFR** implementation. On the other hand, the **ABCD** and **ABCDe** global variant was proposed as an alternative to overcome some properties of the the original **LFR** implementation that we believe are not desirable—see [2] for more details.

3.1 Experiment Setup

Properties were tested on graph with $n = 10{,}000$ nodes. Graphs were generated with various values of the mixing parameter ξ (namely, values between 0 and 1 with a step equal to 0.05). Recall that the parameter ξ is the main parameter of the model, responsible for the level of noise. The detailed results can be found in the accompanying notebook; we present the result for a specific value of $\xi = 0.5$ below. Both variants of the **ABCD** model were tested for a given value of ξ but in the case of the two variants of the **LFR** model, the value of its parameter μ, the counterpart of ξ in the **ABCD** model, was computed using the following formula:

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

where W is the volume of G and W_ℓ is the volume of nodes that belong to ℓ'th community.

In order to increase the comparability of all algorithms, we used the following coupling. For a given set of parameters, we pre-generated the degree distributions and the community sizes. Such pre-generated sequences were used in all four models. In both cases, the distributions were sampled using the discrete power-law distribution with truncation range. The samplers included in the package with the **ABCD** algorithm were used to do this.

As with the parameter ξ, various exponents of the power-law distributions were tested (namely, $\beta \in \{1.1, 1.5, 1.9\}$ and $\gamma \in \{2.1, 2.5, 2.9\}$) and details can be found in the associated notebook. However, in the figures we present, the degree distributions were generated with the exponent $\gamma = 2.5$, the minimum degree δ equal to 5, and the maximum degree equal to $\sqrt{n}$. Community sizes were generated with the exponent $\beta = 1.5$ and the lower and the upper bounds for the sizes of communities were functions of n, namely, there were equal to $0.005n$ and $0.2n$, respectively.

In the figures we present properties of both sequential **ABCD** algorithm and the **ABCDe** algorithm run with 32 threads to show that indeed, as expected from the description of the algorithms, the properties of the produced graphs are the same.

3.2 Results

For every sweep of the parameters of the model, 30 graphs were generated and for each of them 10 representative graph properties were investigated. In the remaining of this section, we summarize the main findings based on the experiments. We will discuss each parameter independently but there are some general observations that we noticed. Given the results of the measurements of 10 different graph characteristics, one can observe that both **ABCD** variants are much more similar to the original **LFR** implementation than the `NetworKit` implementation of **LFR**. Not surprisingly, the local variant of the **ABCD** model is especially close to the original **LFR** model.

Since the evaluated graph parameters are standard and well-known, we do not formally define them. We direct the reader to any book on mining complex networks such as [9].

3.2.1 Clustering Coefficients

The global clustering coefficient of a given graph G is the ratio of three times the number of triangles to the number of pairs of adjacent edges. This parameter has a nice and important

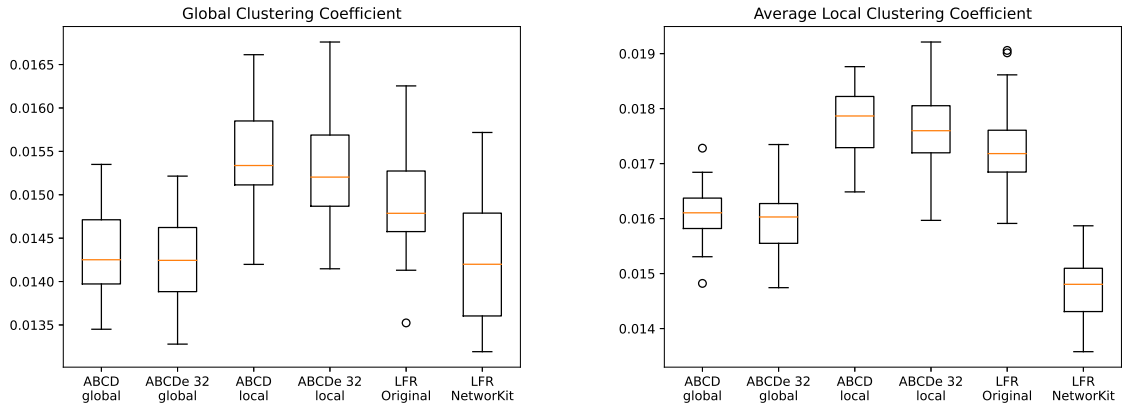

Figure 2: Comparison of the distribution of the global clustering coefficient (left) and average local clustering coefficient (right).

interpretation: given a random pair of adjacent edges, the global clustering coefficient is the probability that those three nodes form a triangle. As a result, it is often used as a measure of the presence of the so-called triadic closure, a natural mechanism present in many complex networks. For example, in a social network, strong triadic closure occurs because there is an increased opportunity for nodes x and z with common neighbour y to meet.

The local clustering coefficient is defined for each node $v \in V(G)$ as the number of triangles this node is part of, divided by the total number of distinct pairs of neighbouring nodes for v. In other words, it is the probability that two random neighbours of v are adjacent. The average local clustering coefficient of G is obtained by averaging the local clustering coefficient over all nodes $v \in V(G)$. Formulas for those coefficients can be found in Chapter 1 of [9].

In Figure 2, we see that the global clustering coefficient is similar for all four compared algorithms. On the other hand, the average local clustering coefficient is most similar between the local variant of **ABCD** and the original **LFR**, while `NetworKit` **LFR** differs the most. Havins said that, the observed differences are not substantial. Finally, let us note that the corresponding numerical values (for both variants) are rather low, comparable to what one would expect from a random chance based on the global density of the corresponding graphs. This is not surprising as none of these models aim to produce graphs with large clustering coefficients, a typical property of the so-called small-world networks. One example of a random graph with large clustering coefficient is the classical Watts and Strogatz model [10].

3.2.2 Node Centralities

An important property of a node of a graph is how central it is with respect to the entire graph. Such measures often reflect the relative importance of a node. There are several ways to measure centrality. In Figure 3, we compare the distribution of four commonly used node centrality coefficients: betweenness [11], closeness [12], PageRank [13], and degree centrality. As usual, we only provide intuition behind these coefficient and direct the reader to Chapter 3 of [9] for more details. The betweenness centrality for a given node is proportional to the number of shortest paths that pass through that node. For example, in a telecommunications network, a

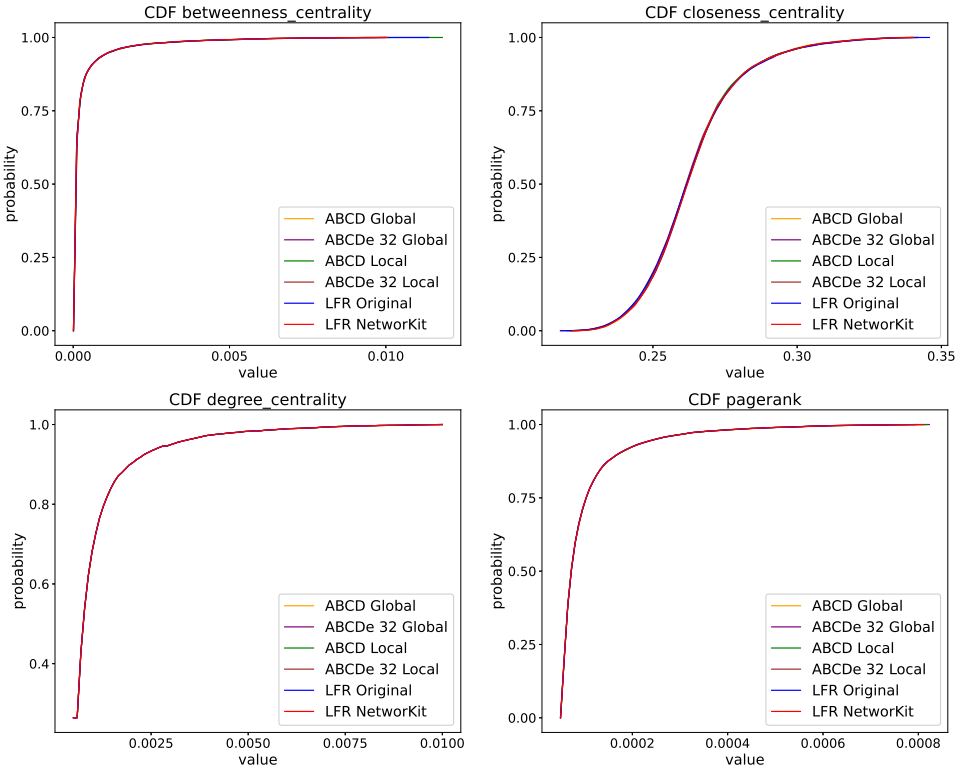

Figure 3: Comparison of the distribution of four commonly used centrality coefficients.

node with higher betweenness would have more control over the network since more information passes through that node. The closeness centrality is defined as the reciprocal of the sum of the length of the shortest paths between a given node and all other nodes in the graph (assuming the graphs is connected). As a result, nodes that are close to all other nodes are considered more central. The PageRank centrality, developed by the Google founders, measures the importance of nodes by assuming that important nodes are those that have many important neighbours, something that the degree centrality ignores as it only considers the number of neighbours to score the nodes. From those plots, we see that all measures are similar for all the benchmark considered.

3.2.3 Degree Correlation

In assortative graphs, high degree nodes tend to link to other high degree nodes, while low degree nodes are more often adjacent to low degree nodes. On the other hand, disassortative graph behave differently: high degree nodes tend to be connected to low degree nodes and vice-versa. In order to capture the preference of nodes it is often useful to investigate the degree correlation function which for a given value of d computes the average degree of neighbours of all nodes

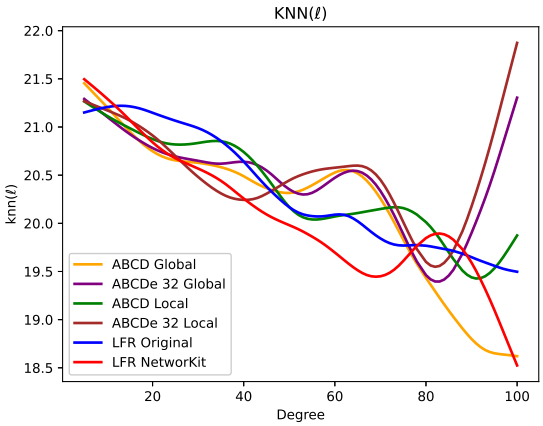

Figure 4: Comparison of the degree correlation function.

of degree d. There are two standard ways to measure the overall assortativity of a graph G, the degree correlation coefficient and the correlation exponent (see [14] and [15], or Chapter 4 in [9]). For both measures, a negative value indicates an assortative network, a value close to zero a neutral one, and a positive value a disassortative one. Many important properties of graphs, such as speed of spreading information or forming large components, are affected by these measures.

In Figure 4, we see that the shape of the degree correlation function is similar for all random graph models, with a clear negative slope. Hence, all benchmarks produce slightly disassortative graphs. In Figure 5, we see that the correlation coefficient is the largest for the original **LFR** and the smallest for `NetworKit` **LFR**; both **ABCD** variants are similar and are roughly in the middle of the range of observed values. Results for the correlation exponent are similar for both **ABCD** variants and the original **LFR**, but it is lower for `NetworKit` **LFR**.

3.2.4 Community Edges and Modularity

We say that an edge is inside a community if both of its nodes are part of the same community; we also refer to those as community edges. Participation coefficient of a given node, as defined in Chapter 5 of [9], provides more detailed information and measures the distribution of a node's neighbours among the communities of a graph. It is equal to 0 if all of its neighbours are in the same community, and it is close to 1 if its neighbours are equally divided amongst all communities. In Figure 6, we see that the proportion of edges that are inside communities is very similar for local **ABCD** and original **LFR**; it is slightly lower for global **ABCD** and much lower for `NetworKit` **LFR**. We also plot the average participation coefficient, with the same conclusions (albeit, symmetric).

The most important graph property of networks in the context of community detection is the modularity function [17]. Indeed, the modularity function is often used to measure the presence of community structure in networks. It is also used as a quality function in many community detection algorithms, including the widely used Louvain algorithm. It is defined as the difference between the proportion of edges inside communities and the expected value of this quantity over some null random graph model. Thus, large (positive) modularity indicates the presence of

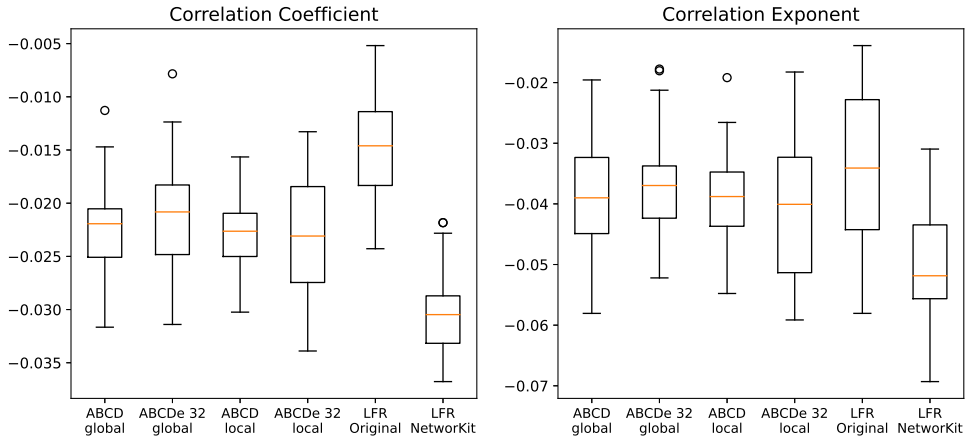

Figure 5: Comparison of distribution for the correlation coefficient (left) and the correlation exponent (right).

communities in a graph. In Figure 7, we see that the modularity function is similar for both **ABCD** variants and original **ABCD**, but is again much lower for `NetworKit` **LFR**.

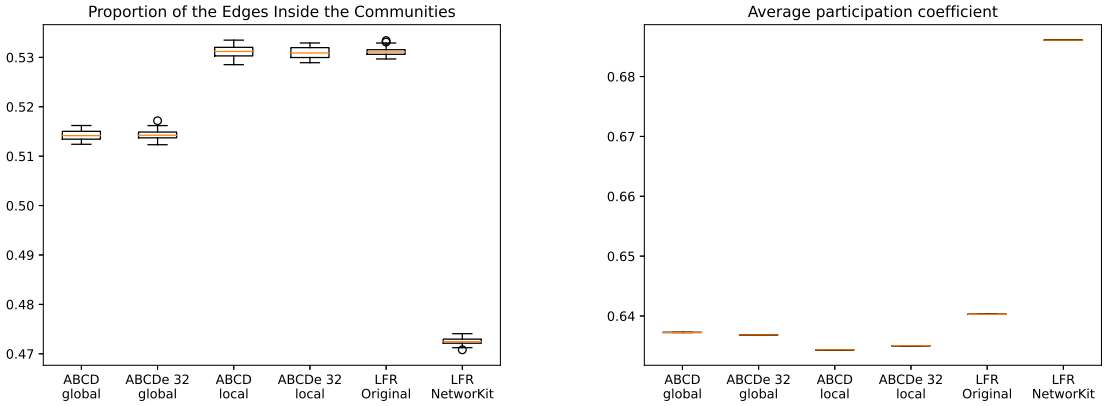

Figure 6: Comparison of proportion of edges inside communities (left) and distribution of average participation coefficient (right).

3.2.5 Shortest Paths

The shortest path between two nodes in a connected graph is the minimum number of hops to go from one node to the other. Several studies, inspired by the famous Milgram's small-world experiment, suggest that many real-world networks exhibit surprisingly small average distance between pairs of nodes. As a result, this property (often referred to as "six degrees of separation") is often investigated and expected from good random graph models. In Figure 8, we compare the average shortest path length (obtained via sampling as there are usually too

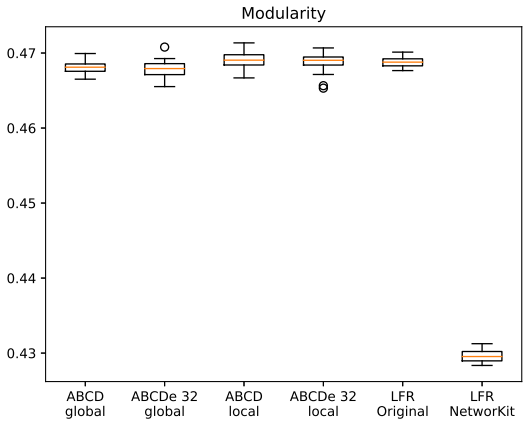

Figure 7: Comparison of distribution of modularity.

many node pairs to investigate). One can see that this quantity is similar for all benchmark algorithms we tested.

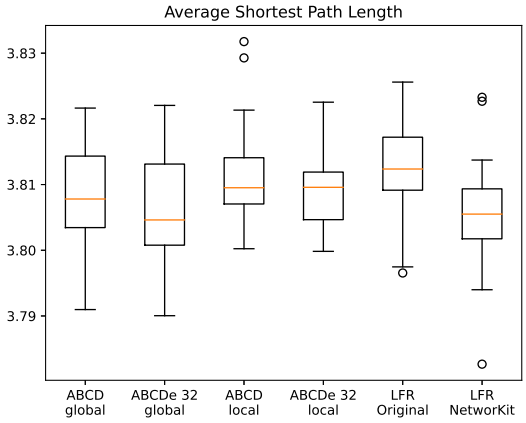

Figure 8: Comparison of distribution of average shortest path length.

4 Speed Tests of Graph Generation

In this section, we present the results of performance comparison of **ABCD**, **ABCDe**, and **LFR**. For **LFR**, we use the `NetworKit` implementation as it is approximately 10 times faster than the original **LFR** implementation. Moreover, let us mention that both **ABCDe** and `NetworKit` **LFR** implementation support multi-threading. For experiments on these benchmarks, we used 1 to 32 threads.

4.1 Experiment Setup

We generated graphs with `NetworKit` **LFR** and **ABCDe** using $k = 2^i$ threads for $0 \leq i \leq 5$. We independently tested single-threaded **ABCD**. We compare the time complexity of each method as a function of the number of nodes n in the graph, where $n = 10^i$ for $4 \leq i \leq 9$. The other parameters were set as for the experiments analyzing properties of the graph presented in the previous section, except for the sizes of the smallest and the largest communities which we vary with n, respectively $0.005n$ and $0.2n$. The range of the values used were chosen based on the objective of the experiment but also the capacity of the machines. For $n < 10^4$ the gain from using multi-threading was negligible whereas experiments with $n > 10^9$ were too computationally expensive for the machines used.

Since the process of generating the background graph is not parallelized, it is expected that the speed of generating **ABCDe** should depend on the size of the background graph which, in turn, depends on the parameter ξ. Results of experiments investigating this relationship (see Figure 10) are presented for a fixed value of $n = 10^7$. (As usual, we refer the reader to notebooks for results for other values of n.) In order to couple **LFR** and global versions of **ABCD** and **ABCDe** so that they have a comparable number of edges in the background graph, the mixing parameter μ was approximated using formula (1).

The experiments are focused on the graph generation process itself, thus the time spent on generating the degree distribution and community sizes are not included in the comparisons. Similarly to the properties test, pre-generated degree distributions and community sizes were used.

The code for execution and analysis of the experiments was written in Julia 1.6.1 programming language. It is available on GitHub repository[2], and so are Jupyter notebooks as well as the results for all combinations of parameters[3]. Julia is a high-level, high-performance, dynamic programming language that recently gains a lot of interest in scientific computing applications [18]. **ABCD** and **ABCDe** graphs were generated using the following packages written in the Julia language. `ABCDGraphGenerator v0.1.0`[4] was used for single threaded implementation (**ABCD**). `ABCDeGraphGenerator v0.2.4`[5] was used for a multi threaded implementation (**ABCDe**). **LFR** graphs were generated using Python v3.8.8 with the `NetworKit v8.1` module[6]. This implementation of the **LFR** algorithm was chosen because it is considered to be the fastest one available at the time of running of the experiments.

Experiments were performed on the machines with 32 Intel Xeon Processors (Cascadelake) 2.30 GHZ vCPUs with 160GB RAM memory, 120GB disk space and Ubuntu 20.04.1 operating system. They were run simultaneously on four machines for approximately two weeks, totalling in over 2000 vCPU hours.

4.2 Results

We first show results of experiments comparing multi-thread **ABCDe** with single-thread **LFR**. Then, we compare it with multi-thread **LFR** for varying n (the number of nodes) and ξ (the

[2] https://github.com/bartoszpankratz/ABCDe_Experiments/tree/main/speed%20test
[3] https://github.com/bartoszpankratz/ABCDe_Experiments
[4] https://github.com/bkamins/ABCDGraphGenerator.jl
[5] https://github.com/tolcz/ABCDeGraphGenerator.jl
[6] https://networkit.github.io/

ξ	n	**ABCDe1**	**ABCDe2**	**ABCDe4**	**ABCDe8**	**ABCDe16**	**ABCDe32**
0.2	10^4	32.14	45.11	57.2	**63.73**	53.98	38.58
0.2	10^5	12.96	17.05	24.22	25.71	**28.07**	26.62
0.2	10^6	16.96	25.28	35.82	36.01	39.66	**39.95**
0.2	10^7	23.52	35.47	47.21	50.73	54.19	**54.21**
0.2	10^8	15.31	23.79	32.57	36.5	**37.24**	35.77
0.2	10^9	45.84	**53.17**	50.77	47.48	46.56	40.29
0.5	10^4	19.95	24.92	29.21	**30.89**	27.51	21.99
0.5	10^5	20.73	29.09	34.03	33.8	**36.17**	35.55
0.5	10^6	30.18	43.44	51.05	51.26	50.95	**56.79**
0.5	10^7	33.89	44.04	47.34	47.89	49.9	**50.38**
0.5	10^8	27.96	39.04	42.48	43.75	**44.81**	44.17
0.5	10^9	38.19	39.96	**41.52**	37.91	37.31	35.06
0.8	10^4	29.34	33.57	**35.55**	35.26	34.22	29.81
0.8	10^5	35.76	42.93	46.36	43.89	46.71	**47.51**
0.8	10^6	43.46	57.65	56.28	57.69	61.3	**69.14**
0.8	10^7	39.78	43.1	46.88	47.83	46.31	**48.35**
0.8	10^8	38.52	44.49	46.3	48.31	**49.1**	46.57
0.8	10^9	37.69	41.57	41.14	44.24	**46.05**	34.27

Table 1: Comparison of graph generation speed in reference to `NetworKit` **LFR** (single threaded) as a function of graph size and ξ. The values reported are normalized so they reflect how many times faster is the graph generation process in comparison to the reference `NetworKit` **LFR** implementation; hence, the larger the values the better. Number **X** in the name of the model **ABCDeX** represents the number of threads used to generate the graph. The best results are shown in bold.

mixing parameter), respectively. As a general conclusion, we see 10–50 fold speed-ups using **ABCDe** instead of **LFR**. Another general observation is that using multiple threads does yield speed-ups but the difference could be small beyond 8 threads.

4.2.1 Speed-up with Multi-thread ABCDe

In Table 1, we show the speed-up of **ABCDe** algorithm over `NetworKit` **LFR** run on a single thread. In particular, even single-thread **ABCDe** algorithm is faster than `NetwrorKit` **LFR** for varying values of graph size n and ξ parameter; we observe from 13 to 45 times speed-up which is a practically significant improvement regarding that the **LFR** generation time for graphs with $n = 10^9$ nodes was roughly 60 hours in our experiments. Let us also point out that the new **ABCDe** algorithm (run in a single-thread mode) is faster than the old sequential **ABCD** implementation by around 30% (as reported in Figure 9) (the changes leading to these speedups are of code optimization nature, like reduction of volume of memory allocations, and were guided by profiling of code runtime; they did not introduce new algorithmic ideas). Additionally, one can observe that increasing the number of threads improves the speed of the **ABCDe** graph generation, with best results achieved typically when using 8 threads or more (unfortunately the individual timings were not very stable since it was impossible to maintain the same level of system load of the test machine in the cloud infrastructure because the time of running of the whole experiment was long).

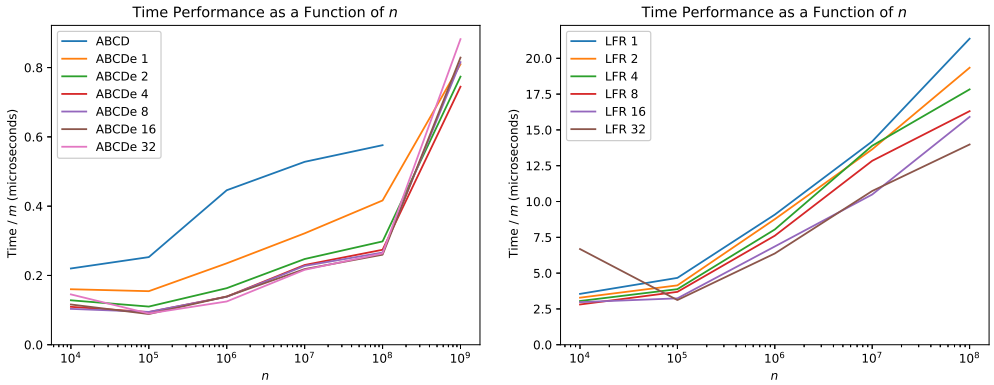

Figure 9: Comparison of time (in microseconds) to generate a single edge as a function of the graph order. In the left plot we present statistics for **ABCD** model and in the right plot for **LFR** model. Numbers in the legend are the number of threads used to generate the graphs.

4.2.2 Comparing Multi-thread ABCDe and LFR—Varying n

In Figure 9, we present a comparison of average generation time per edge as a function of n, the order of a graph, for **ABCDe** and `NetworKit` **LFR**, with varying number of threads (for sequential **ABCD** and for **LFR** the experiments were not run for $n = 10^9$ as their run-time was very long; we only run the **LFR** algorithm once to measure its baseline timing for Table 1, which was around 60 hours). The parameter ξ was fixed to $\xi = 0.5$. The results are consistent with the data presented in Table 1 with **ABCDe** over 10 times faster than `NetworKit` **LFR** for all considered scenarios. However, there are two additional characteristics that should be noted. First of all, observe that both algorithms decrease speed of edge generation as n increases. However, this speed decrease is much slower for **ABCDe**. Indeed, in the case of **ABCDe**, moving from $n = 10^4$ to $n = 10^8$, the edge generation time increases by no more than a factor of 2.5. On the other hand, for `NetworKit` **LFR**, the increase is over 5 fold and so a better scaling is observed for **ABCDe**. For $n = 10^9$ we observe a bump in processing time of **ABCDe**. This bump is due to the fact that this size of graph was at the limit of processing capability of the available infrastructure. A similar bump for **LFR** due to the same resource limits reasons is expected, which conformed in Table 1, where the relative speedup of **ABCDe** over **LFR** does not drop for $n = 10^9$. The second aspect we would like to highlight are the speed-ups obtained when increasing the number of threads. Here one can also see that **ABCDe** scales better; for example, when moving from 1 to 2 threads there is a noticeable speed-up in **ABCDe** (roughly 20%), while for `NetworKit` **LFR** it is relatively small (less than 5%). Moreover, when we compare using 1 vs. 32 threads, the speed-up for **ABCDe** is over 2-fold, while for `NetworKit` **LFR** it is less than 1.5-fold.

4.2.3 Comparing Multi-thread ABCDe and LFR—Varying ξ

For the last experiment, we fix the number of nodes to $n = 10^7$ and vary the mixing parameters ξ. The results are shown in Figure 10. The speed-up obtained with **ABCDe** is still enormous compared to **LFR**, but we also see that both algorithms behave similarly with respect to the different number of threads used. The largest speed-ups with multiple threads are observed for

low values of ξ; for **ABCDe** model, this is not too surprising since in that case most of the time is spent for generating community graphs which are completely independent tasks.

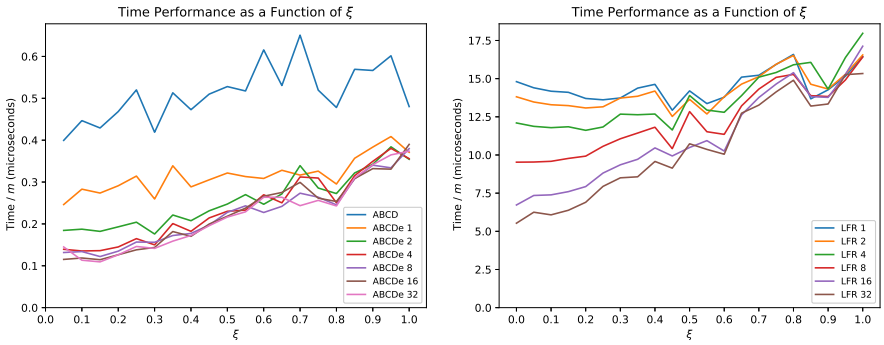

Figure 10: Comparison time (in microseconds) to generate a singe edge as a function of the parameter ξ. In the left plot we present statistics for **ABCD** model and in the right plot for **LFR** model. Numbers in the legend are the number of threads used to generate the graphs.

5 Conclusions and Future Directions

In this paper, we showed that the multi-thread **ABCDe** is over 10 times faster than **LFR** and scales better than parallel implementation of the **LFR** algorithm provided in `NetworKit`. Moreover, the algorithm is not only faster but graphs generated by **ABCDe** algorithm have similar properties to graphs generated by the original **LFR** algorithm, while the parallelized **NetworKit** implementation of **LFR** produces graphs that have noticeably different characteristics. In the paper we described the technical approach we have taken to parallelize the **ABCDe** algorithm. Additionally, the produced algorithm ensures reproducibility of the generated graph independent from task execution ordering or number of threads used.

Despite the fact that **ABCDe** is already very fast, there are some ways the generation process can be improved. In particular, the current "bottleneck" and an area for further research is to design a faster, parallelized implementation of the background graph. This would allow to make a better use of multiple threads in scenarios when the parameter ξ is large and so most of the edges are present in the background graph. Currently, as it is shown in Figure 10, the scalability with number of threads of **ABCDe** algorithm degrades as ξ increases.

Another further direction worth investigating would be to generalize the **ABCD**/**ABCDe** model to include more sophisticated, higher-order structures as well as to capture the dynamics of networks. A good starting point would be to deal with hypergraphs in which edges (called hyperedges) may contain more than two nodes. Indeed, the modularity function for graphs was recently generalized to hypergraphs [19] and a number of research groups started working on scalable algorithms [20, 21] as well as software implementations such as the `HyperNetX` package[7] but there is need for synthetic hypergraph benchmarks. One of the very first attempts include the hypergraph stochastic block model [22] but now it is time for more realistic models producing power-law degree distribution and other desired properties. Temporal graph generators that

[7] https://github.com/pnnl/HyperNetX

control both the evolution of the degree distribution as well as the distribution of community sizes are even more challenging. One of the very first such models is **RTGEN**, A **R**elative **T**emporal Graph **GEN**erator [23] that was recently introduced.

6 Acknowledgment

Hardware used for the computations was provided by the SOSCIP consortium[8]. Launched in 2012, the SOSCIP consortium is a collaboration between Ontario's research-intensive post-secondary institutions and small- and medium-sized enterprises (SMEs) across the province. Working together with the partners, SOSCIP is driving the uptake of AI and data science solutions and enabling the development of a knowledge-based and innovative economy in Ontario by supporting technical skill development and delivering high-quality outcomes. SOSCIP supports industrial-academic collaborative research projects through partnership-building services and access to leading-edge advanced computing platforms, fuelling innovation across every sector of Ontario's economy.

References

[1] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. Phys. Rev. E, 78(4), 2008.

[2] B. Kamiński, P. Prałat, and F. Théberge, Artificial Benchmark for Community Detection (ABCD)—Fast Random Graph Model with Community Structure, Network Science 9(2) (2021), 153–178.

[3] B. Kamiński, B. Pankratz, P. Prałat, and F. Théberge, Modularity of the ABCD Random Graph Model with Community Structure, arXiv:2203.01480, 2022.

[4] V.D. Blondel, J-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. Journal of statistical mechanics: theory and experiment, 2008(10):P10008, 2008.

[5] Staudt C. L., Sazonovs A., and Meyerhenke H.: NetworKit: A Tool Suite for Large-scale Complex Network Analysis, (2015)

[6] Bollobás, B.: A probabilistic proof of an asymptotic formula for the number of labelled regular graphs. European Journal of Combinatorics, 1, 311–316 (1980)

[7] Chung, F. and Lu, L.: Complex Graphs and Networks. American Mathematical Society (2006)

[8] Kolda, T. G., Pinar, A., Plantenga, T., and Seshadhri, C.: A scalable generative graph model with community structure. SIAM Journal on Scientific Computing, 36(5), C424–C452 (2014)

[9] B. Kamiński, P. Prałat and F. Théberge, Mining Complex Networks, Chapman and Hall/CRC (2021).

[8] https://www.soscip.org/

[10] Watts, Duncan J., and Steven H. Strogatz. "Collective dynamics of 'small-world'networks." nature 393, no. 6684 (1998): 440-442.

[11] L. Freeman. "A set of measures of centrality based on betweenness". *Sociometry* (1977).

[12] L. Freeman. "Centrality in social networks: conceptual clarification". *Soc. Networks* **1** (1979), 215–239.

[13] S. Brin and L. Page. "The anatomy of a large-scale hypertextual Web search engine". *Comput. Networks ISDN Systems* **30(1-7)** (1998), 107–117.

[14] R. Pastor-Satorras, A. Vazquez, A. Vespignani. "Dynamical and correlation properties of the Internet". *Phys. Rev. Lett.* **87** (2001), 258701.

[15] M.E.J. Newman. "Assortative mixing in networks". *Phys. Rev. Lett.* **89** (2002), 208701.

[16] M.E.J. Newman. Networks: An introduction. Oxford University Press (2010).

[17] M.E.J. Newman, M. Girvan. "Finding and evaluating community structure in networks". *Phys. Rev. E.* 2004; 69: 026–113.

[18] Bezanson, J., Edelman, A., Karpinski, S., and Shah, V.: Julia: A fresh approach to numerical computing. SIAM Review, 69, 65–98 (2017).

[19] B. Kamiński, V. Poulin, P. Prałat, P. Szufel, and F. Théberge, Clustering via Hypergraph Modularity, PLoS ONE 14(11): e0224307.

[20] T. Kumar, S. Vaidyanathan, H. Ananthapadmanabhan, S. Parthasarathy and B. Ravindran, A New Measure of Modularity in Hypergraphs: Theoretical Insights and Implications for Effective Clustering. In: Cherifi H., Gaito S., Mendes J., Moro E., Rocha L. (eds) Complex Networks and Their Applications VIII. COMPLEX NETWORKS 2019. Studies in Computational Intelligence, vol 881. Springer, Cham.

[21] B. Kamiński, P. Prałat and F. Théberge, Community Detection Algorithm Using Hypergraph Modularity, Proceedings of the 9th International Conference on Complex Networks and their Applications, Studies in Computational Intelligence 943, Springer, 2021, 152–163.

[22] Maria Chiara Angelini, Francesco Caltagirone, Florent Krzakala, and Lenka Zdeborová. Spectral detection on sparse hypergraphs. In 2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton), pages 66–73. IEEE, 2015.

[23] Massri, M., Miklos, Z., Raipin, P. and Meye, P., 2022, March. RTGEN: A Relative Temporal Graph GENerator. In DATAPLAT workshop at the EDBT/ICDT 2022 Joint Conference.

[24] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring)). Association for Computing Machinery, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[25] Fisher, Ronald A., and Frank Yates. Statistical tables for biological, agricultural and medical research. Oliver and Boyd Ltd, London, 1943.