# Large Scale Community-Aware Network Generation

Vikram Ramavarapu[1*], João Alfredo Cardoso Lamy[2], Mohammad Dindoost[3], David A. Bader[3]

[1*]Computer Science, University of Illinois at Urbana-Champaign, Urbana, 61801, IL, USA.
[2]Computer Science, Insper Institute, São Paulo, SP, Brazil.
[3]Data Science, New Jersey Institute of Technology, Newark, 07102, NJ, USA.

*Corresponding author(s). E-mail(s): vikramr2@illinois.edu;
Contributing authors: joaoacl@al.insper.edu.br;

**Abstract**

Community detection, or network clustering, is used to identify latent community structure in networks. Due to the scarcity of labeled ground truth in real-world networks, evaluating these algorithms poses significant challenges. To address this, researchers use synthetic network generators that produce networks with ground-truth community labels. RECCS is one such algorithm that takes a network and its clustering as input and generates a synthetic network through a modular pipeline. Each generated ground truth cluster preserves key characteristics of the corresponding input cluster, including connectivity, minimum degree, and degree sequence distribution. The output consists of a synthetically generated network, and disjoint ground truth cluster labels for all nodes. In this paper, we present two enhanced versions: RECCS+ and RECCS++. RECCS+ maintains algorithmic fidelity to the original RECCS while introducing parallelization through an orchestrator that coordinates algorithmic components across multiple processes and employs multithreading. RECCS++ builds upon this foundation with additional algorithmic optimizations to achieve further speedup. Our experimental results demonstrate that RECCS+ and RECCS++ achieve speedups of up to 49× and 139× respectively on our benchmark datasets, with RECCS++'s additional performance gains involving a modest accuracy tradeoff. With this newfound performance, RECCS++ can now scale to networks with over 100 million nodes and nearly 2 billion edges.

# 1 Introduction

Community detection, or network clustering, is the process of partitioning a network into modular, meso-scale subgraphs, and has broad applications across many domains [1–3]. Community detection algorithms typically operate with respect to an optimality metric that defines a "good" community. For instance, algorithms like Louvain [4] optimize modularity [5], which favors higher edge density within clusters and lower edge density between clusters. The Leiden algorithm [6] can optimize both Modularity the Constant Potts Model score [7], which emphasizes more clique-like cluster structure. Connectivity, or minimum cut value (mincut), measures the robustness of a community—the minimum number of edges whose removal would disconnect the cluster. Methods such as the Connectivity Modifier [8] directly optimize this metric. Probabilistic approaches like Stochastic Block Models (SBM) [9] detect communities through likelihood maximization over block assignments and inter-block edge probability matrices.

Evaluating community detection algorithms poses significant challenges due to the scarcity of ground truth labels in real-world networks. To address this, researchers use synthetic network generators for benchmarking. When a network and its clustering are provided as input to a synthetic network generator, it produces multiple similar networks with ground truth community labels. These generated networks allow clustering methods to be evaluated by comparing their predictions against the known ground truth. A general evaluation pipeline utilizing a synthetic network generator is illustrated in Figure 1. The Lancichinetti-Fortunato-Radicchi (LFR) [10] benchmark generates networks by drawing node degrees and community sizes from power-law distributions, then wiring edges according to a mixing parameter $\mu$ that controls the fraction of each node's edges connecting outside its community. While widely used, LFR generates networks from distributional parameters rather than replicating the specific structural properties of a reference network. The Stochastic Block Model (SBM) offers an alternative approach to synthetic network generation, taking as input block assignments and inter-block edge probability matrices—parameters that are otherwise inferred during community detection—to generate networks with community structure. A degree-corrected variant [11] additionally incorporates a degree sequence to better model real-world networks. However, SBM-generated networks often contain disconnected or weakly connected clusters, and SBM does not explicitly model important structural properties such as connectivity and minimum degree [12]. Edge-Connected SBM (EC-SBM) [13] addresses this by first generating k-edge-connected subnetworks for each cluster to ensure desired connectivity, then using SBM to generate remaining edges between clusters. For the purposes of this paper, we focus on **RECCS** [14], which takes as input a network and its clustering and generates a synthetic network that preserves key characteristics of the input clustering, including connectivity (mincut), minimum degree, and degree sequence distribution.

The RECCS pipeline works as follows: the reference network is divided into clustered and singleton subnetworks. A clustered node is a node that belongs to a cluster of more than one node. The clustered subnetwork, $G_c$ is the induced subgraph of all clustered nodes. The singleton subnetwork $G_s$ is the edge-based complement of $G_c$. In

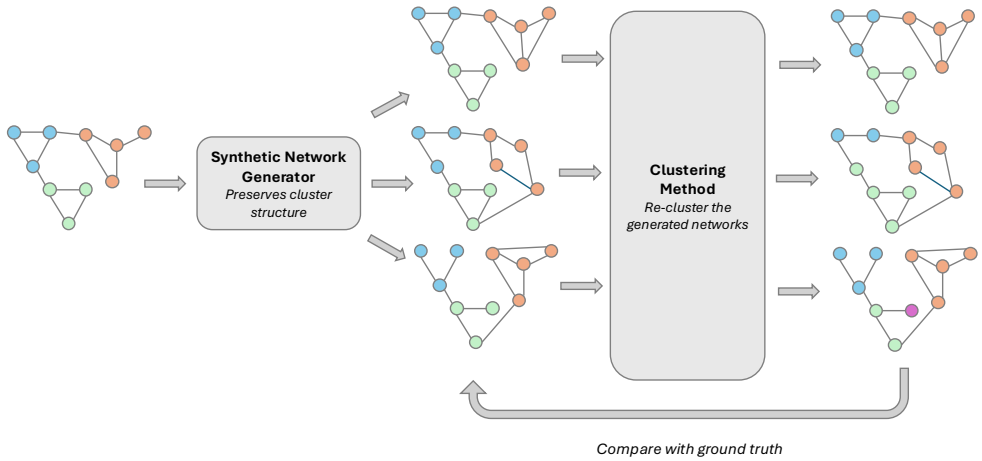


**Fig. 1 A general network clustering evaluation pipeline.** A reference network and clustering is taken in as input to a synthetic network generator, which generates multiple synthetic networks with ground truth communities. When a clustering method is run on these generated networks, they can be compared with the ground truth community labels for accuracy.

other words, the edge sets of $G_c$ and $G_s$ are disjoint and union to the edge set of the entire network.

Then, a degree-corrected SBM is run on the clustered subnetwork and the singleton subnetwork. The original cluster assignments, sizes, and probabilities are used for the SBM on the clustered subnetwork. For the singleton subnetwork, singletons are treated as their own cluster. Since the singleton subnetwork is an edge set complement of the clustered subnetwork, as opposed to the node set complement, there are still clustered nodes in the singleton subnetwork. As such, clustered nodes in $G_s$ are assigned to their original clusters, while singletons are assigned to their own clusters. This cluster assignment is used to generate a SBM of the singleton subnetwork.

The SBM-generated clustered subnetwork is then processed by the **RECCS module**, which consists of four stages. In the first stage, edges are added to ensure that each node satisfies a minimum degree requirement per cluster, determined by the minimum cut value of the corresponding reference cluster. Then, in the second stage, edges are added to disconnected clusters to stitch connected components together. In the third stage, edges are added between minimum cut partitions so that the cluster meets the minimum cut requirement. In stage four, edges are added to match degree sequences in the generated clusters to the reference clusters.

In the original implementation of RECCS, stage four is done with two different strategies. The first version uses a max heap prioritizing nodes with the highest degree deficit. The second version computes a degree-corrected SBM using leftover degrees that are appended to the cluster. For the scope of this paper, we focus on improving the performance of the first version of RECCS, and leave the second version for future work.

Once the RECCS module has processed the clustered subnetwork, it is merged with the singleton subnetwork to produce a final graph. The merge is a simple edge set concatenation.

While RECCS is effective in generating synthetic networks that preserve key properties of the input clustering, it suffers from long runtimes, especially on large networks with many clusters. This limits its applicability for large-scale network generation tasks. To address this limitation, we first present **RECCS+**, which introduces parallelization strategies to the original RECCS pipeline while maintaining algorithmic fidelity. In order to achieve further speedups, we then find that an algorithmic redesign is necessary, leading to the development of **RECCS++**. RECCS++ incorporates additional algorithmic optimizations alongside parallelization to achieve significant performance improvements, albeit with a modest tradeoff in accuracy.

In the following sections, we first describe the data used for evaluation in Section 2. Then, we describe our methods in Section 3, including the parallelization strategies used in RECCS+ and the algorithmic optimizations in RECCS++. We present our results in Section 4, including speedup results for RECCS+ and RECCS++ as well as accuracy comparisons to demonstrate algorithmic fidelity. In other words, we show that RECCS+ produce networks with near identical properties to those generated by the original RECCS. Since RECCS++ introduces algorithmic changes, we also analyze the differences in network properties between RECCS++ and the original RECCS. In Section 5, we present further algorithmic analysis, demonstrating RECCS, RECCS+, and RECCS++'s use and behavior as a synthetic network generation software by running reclustering evaluation on our generated networks. In Section 6, we present a abstract generalization of RECCS as an algorithmic framework that opens up possibilities for future work. Finally, we conclude with a summary of our findings and future directions in Section 7.

## 2 Data

In our preliminary analysis, we also use the Cit-HepPh dataset from the SNAP dataset [15], with 34,546 nodes and 421,578 edges. We also use the Curated Exosome Network (CEN), which is a citation network of publications in exosome biology [16]. In the preliminary analysis, we cluster both networks using Leiden with resolution 0.01.

For algorithmic fidelity evaluation, we use 73 of the 74 networks from the EC-SBM paper's network list, all of which are from the Netzschleuder network catalogue [17], as our benchmark datasets for algorithmic fidelity and an initial speedup evaluation. Each network is clustered using SBM-WCC [18]. We exclude the `marvel_universe` network because our clustering contains only outlier nodes, meaning RECCS would effectively reduce to a degree-corrected Erdos-Renyi generator [19]. These networks span a variety of domains, including social, biological, and information networks, ranging from 906 to 1,402,673 nodes and 2,408 to 17,233,144 edges. Each network size is shown in Figure 2.

We also use four larger networks to evaluate the scalability of RECCS+ and RECCS++: Orkut [15, 20], the Curated Exosome Network (CEN) mentioned earlier, Open Citations (OC) [21, 22], and Open Citations V2 (OCv2) [22, 23]. Each of these

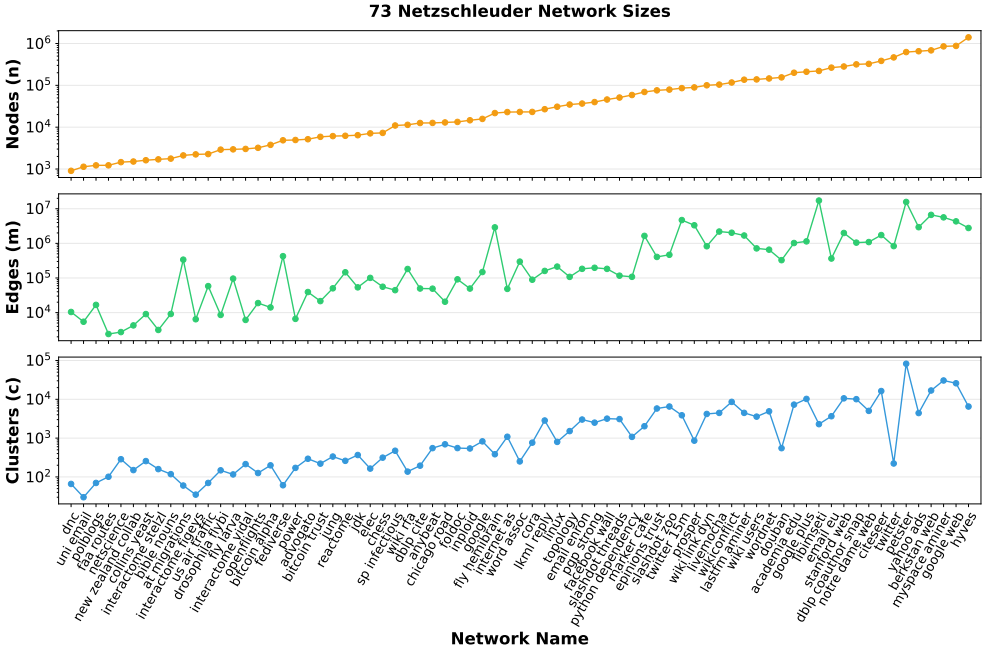


**Fig. 2 Network sizes and number of clusters in the 73 Netzschleuder set.** The top and middle panels show the number of nodes ($n$) and edges ($m$) respectively, while the bottom panel shows the number of clusters ($c$) in each network.

networks were clustered using Leiden with resolution 0.01, both with and without Connectivity Modifier (CM) treatment. We ran CM using our CM++ pipeline software [24]. The cluster size distribution across the large networks are shown in Figure 3.

Finally, we use seven networks from the "Testing 2" dataset in the original RECCS paper for further reclustering accuracy analysis. This includes the following networks: gemsec-Facebook (artist-edges), com-LiveJournal (com-lj), com-Youtube, gemsec-Deezer (HR-edges), twitch-gamers, musae-github and ego-Twitter. All of these networks are from the SNAP catalogue.

All networks used in this study are listed in Table 1.

# 3 Materials and Methods

## 3.1 Algorithmic Details

Both RECCS+ and RECCS++ employ a multi-process orchestrator that executes tasks in parallel following the pipeline shown in Figure 4. The pipeline takes as input a reference network and its clustering, replicating both the network topology and cluster statistics in the generated output.

The pipeline begins by splitting the input into two induced subgraphs: $G_c$ contains all nodes belonging to clusters of size greater than one, while $G_s$ contains its edge

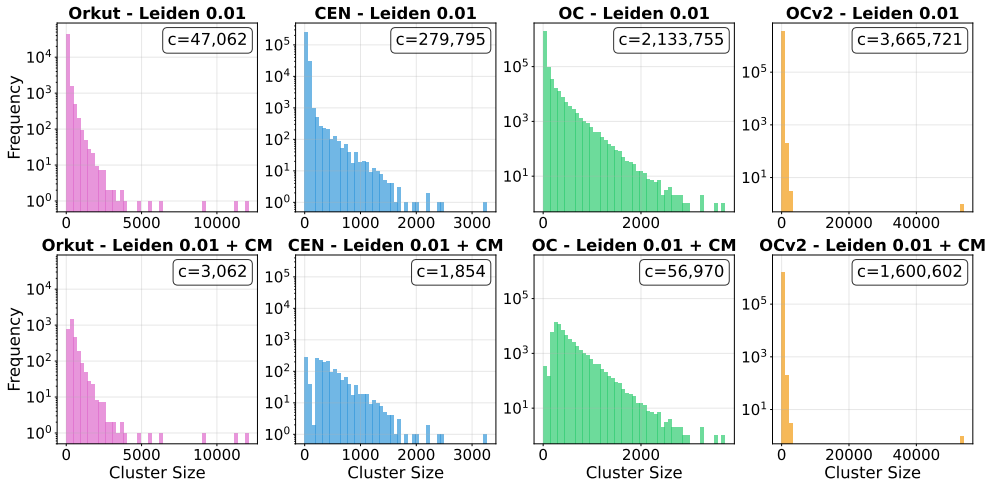


**Fig. 3 Cluster size distribution across all large networks used for scalability experiments.** We use the Orkut, CEN, OC, and OCv2 networks for scalability testing. The networks are clustered with Leiden 0.01, both with and with out Connectivity Modifier (CM) treatment.

**Table 1 Network datasets used in this study.** Network sizes are given in number of nodes and edges. The experiments column indicates which experiments each network was used for: Preliminary analysis, Algorithmic Fidelity, Scalability, and Reclustering.

| Network(s) | Nodes | Edges | Experiment(s) | Citation(s) |
|---|---|---|---|---|
| Cit-HepPh | 34,546 | 421,578 | Preliminary | [15, 25, 26] |
| Netzschleuder Set | 906–1,402,673 | 2,408–17,233,144 | Fidelity | [13, 17] |
| Orkut | 3,072,441 | 117,185,083 | Scalability | [15, 20] |
| CEN | 13,989,436 | 92,051,051 | Preliminary, Scalability | [16] |
| OC | 75,025,194 | 1,363,303,678 | Preliminary, Scalability | [21, 22] |
| OCv2 | 107,054,145 | 1,962,840,983 | Scalability | [22, 23] |
| artist-edges | 50,515 | 819,306 | Reclustering | [15, 27] |
| com-lj | 3,997,962 | 34,681,189 | Reclustering | [15, 20] |
| com-Youtube | 1,134,890 | 2,987,624 | Reclustering | [15, 20] |
| HR-edges | 54,573 | 498,202 | Reclustering | [15, 27] |
| twitch-gamers | 168,114 | 6,797,557 | Reclustering | [15, 28] |
| musae-github | 37,700 | 289,003 | Reclustering | [15, 29] |
| ego-Twitter | 81,306 | 1,768,149 | Reclustering | [15, 30] |

complement. Both subgraphs inherit their cluster assignments from the input clustering $C$, yielding $C_c$ and $C_s$ respectively. Concurrently, extracts cluster-level statistics (number of nodes and edges and mincut size) from the input to guide the RECCS module. Once the splitting completes, SBMs are run on both $(G_s, C_s)$ and $(G_c, C_c)$ in parallel, generating $G_{s\_SBM}$ and $G_{c\_SBM}$. The clustered SBM, $G_{c\_SBM}$ is processed by the RECCS module, after which the processed result is concatenated with $G_{s\_SBM}$ to produce the final synthetic network $G^*$.

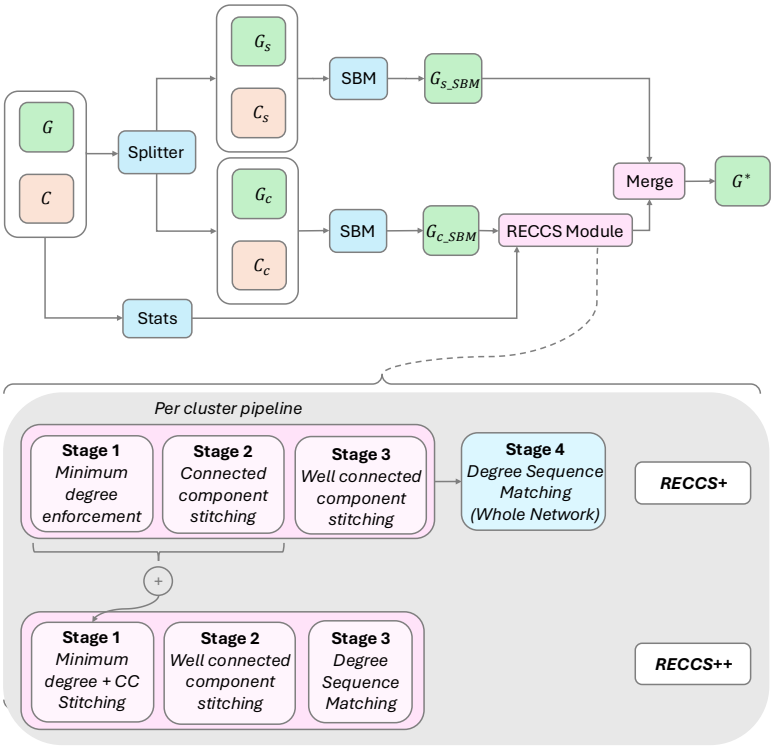


**Fig. 4 The full pipeline of RECCS+ and RECCS++.** First, statistics are collected for the input clustering. In the meantime, a splitter outputs the clustered and singleton sunetworks. When the splitter is finished, SBMs are run on both subnetworks in parallel. When the clustered SBM and the stats are both done, the RECCS module is run. When the RECCS module and the singleton SBM are finished, they are merged into a final output graph. All C++ processes are labeled in pink, while all Python processes are blue.

The RECCS module behavior differs between variants. In RECCS+, the module is algorithmically consistent with the original, with parallelism implemented. Minimum degree enforcement, connected component stitching, and well-connected component stitching operate per cluster via a multithreaded task queue, followed by global degree sequence matching after all cluster-level tasks complete. In RECCS++, all operations—including degree sequence matching—are performed per cluster within the task queue across three stages: (1) combined minimum degree enforcement and connected component stitching, (2) well-connected component stitching, and (3) per-cluster degree sequence matching. This per-cluster approach in RECCS++ enables greater parallelization compared to the global degree sequence matching in RECCS+, but results in slightly different network properties of $G^*$.

Minimum degree enforcement and connected component stitching are combined in RECCS++ by prioritizing edge additions that satisfy both requirements. When a cluster is disconnected, stitching edges between components connect the lowest-degree node from each component, after which minimum degree enforcement proceeds as normal.

## 3.2 System Details

For preliminary and reclustering experiments in Sections 3.3 and 4.3, we used the University of Illinois Campus Cluster `zgdrasil` partition, using Intel Xeon Platinum 8568Y+ 2.30 GHz processors (96 cores) and 128 GB of RAM. For the algorithmic fidelity and scalability experiments in Sections 4.1 and 4.2, we used the NJIT Wulver Cluster, with AMD EPYC 7753 2.45GHz CPUs (64 cores). For algorithmic fidelity experiments, we used the `general` partition with 512GB of RAM, and for scalability experiments, the `bigmem` partition with 2TB of RAM.

## 3.3 Implementation Details

In our implementation of RECCS, we primarily use C++ for the RECCS module. We use the `graph-tool` Python module [31] to compute SBMs. Since the inputs for the SBMs are two-column edgelists and cluster assignments (node, cluster) of nodes in $G_c$ and $G_s$, splitting the graph does not need any network loading and can be reduced to a simple tabular split that can be performed in Python using Pandas. The stats computation is done in a Python script borrowed from the CM++ Pipeline software, modified to only collect cluster IDs, cluster sizes (number of nodes and edges) and minimum cut values.

RECCS in its original form utilizes `graph-tool` for SBM generation. It first takes a network and clustering as input and computes an edge count matrix between clusters based on the observed inter-cluster and intra-cluster edge patterns in the input network. Using this matrix along with the cluster assignments and degree sequence from the original network, `graph-tool` generates a synthetic network via a degree-corrected SBM. The edge count matrix is first computed as a dictionary-of-keys (DOK) sparse matrix format, and is converted to a compressed sparse row (CSR) matrix. RECCS+ and RECCS++ improve SBM performance by using a coordinate (COO) matrix instead of a DOK. While DOK outperforms COO for incremental modifications (changing individual elements one at a time), COO is better for bulk modifications (changing chunks at a time). Therefore, edges are partitioned into chunks to modify the COO-form edge count matrix. Moreover, chunks are handled in parallel via multithreading to further improve matrix construction performance.

We verify that these changes made to the SBM generation stage indeed yields significant speed-ups through a preliminary small-scale experiment. For this experiment, we use both Cit-HepPh and CEN, clustered using Leiden CPM with resolution 0.01. As shown in Figure 5, these changes yield a 9.7x speedup on Cit-HepPh and a 3.9x speedup on CEN.

Within the RECCS module, we represent graphs using adjacency matrices in CSR format to improve loading speed and preserve memory using a compressed format. We verify that graph loading is indeed more performant with CSR adjacency representations through a preliminary experiment. We used the OpenCitations network and compared our C++ graph loader runtime with iGraph [32, 33] and Networkit [34] baselines. Figure 6 shows that our CSR loader yields a 3.48x performance improvement over iGraph and a 3.70x performance improvement over Networkit.

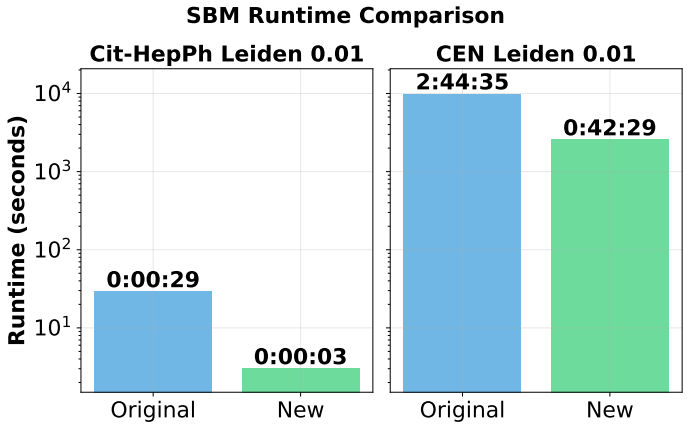


**Fig. 5 Speedup of the SBM stage after optimization.** Algorithmic, data structure, and parallelization optimizations were used to speed up SBM generation. New SBM runs utilize 64 threads. Speedup: Cit-HepPh: 9.7x, CEN: 3.9x.

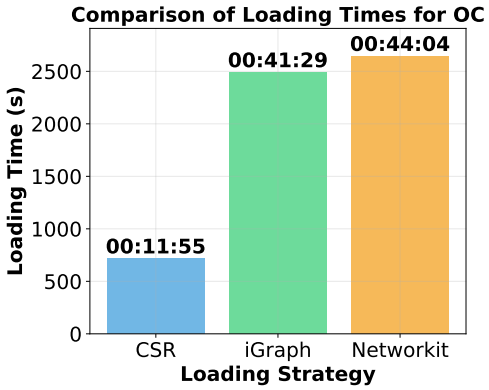


**Fig. 6 CSR graph loader runtime compared to iGraph and Networkit baselines.** All runtimes are single-threaded. CSR Speedup with respect to iGraph: 3.48x, Networkit: 3.70x.

Moreover, the RECCS module implements shared-memory, single-node parallelism using OpenMP for multithreaded queue operations. With respect to multiprocessing, the process DAG shown in Figure 4—comprising the splitter, stats, and SBM processes—is coordinated through an orchestrator that manages process forks. The splitter process divides the input graph into clustered and unclustered components, and SBMs are run in parallel on both components. During the splitter's runtime, stats are computed on the input network and clustering. The stats script is reused from the CM++ pipeline. In RECCS+, when the multithreaded queue is completed, the degree sequence matching is done through a forked Python process. The degree sequence matching script is reused from the EC-SBM source code[1].

[1] https://github.com/illinois-or-research-analytics/ec-sbm/blob/main/ec-sbm/correct_degree.py

**Table 2 Properties of the real and synthetic networks and the metrics used to evaluate them.**

| Stat | Metric Used |
|---|---|
| **Degree Sequence** | RMSE |
| **Minimum Edge Cuts Sequence** | RMSE |
| **Diameter** | Relative |
| **Edges Between Outliers & Clustered Nodes** | Relative |
| **Outlier Degree Sequence** | RMSE |
| **Mean Local Clustering Coefficients** | Absolute |
| **Global Clustering Coefficients** | Absolute |
| **Mixing Parameter** | Absolute |

The code for this software is made available on GitHub[2]

# 4 Results

## 4.1 Algorithmic Fidelity

A comparison of network statistics across the 73 selected EC-SBM networks are shown in figure 7. We use relative difference to evaluate diameter and the number of edges between the clustered and outlier subnetworks. We use absolute difference to evaluate mixing parameter and clustering coefficients. We use root mean-squared error (RMSE) for evaluation of the degree sequence, both across the graph and in the outlier subnetwork, and the minimum edge cut sequence. The absolute and relative differences are used when the real and synthetic network statistics are represented by scalars $s$ and $s'$. RMSE is used when the real and synthetic network statistics are represented by sequences $s$ and $s'$ where the $i^{\text{th}}$ element of both sequences are represented as $s_i$ and $s'_i$. The equations for each metric are represented in equations 1, 2, and 3.

$$\text{absolute difference} = s - s' \tag{1}$$

$$\text{relative difference} = (s - s')/s \tag{2}$$

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(s_i - s'_i)^2} \tag{3}$$

As the purpose of this experiment is to evaluate algorithmic fidelity, all RECCS variants use the same SBMs to avoid differences that may come from randomization. As shown in the figure, there is high algorithmic fidelity between RECCS+ and RECCSv1. However, RECCS++ shows some tradeoff in accuracy. Clusters of RECCS++ exhibit higher connectivity than the original network, more so than the RECCS and RECCS+ networks. Since the degree sequence of RECCS++ exhibit higher RMSE than the other two versions, this is likely due to RECCS++ being constrained to add edges within a cluster to fit degree sequence, therefore leading to higher

[2] https://github.com/illinois-or-research-analytics/reccs

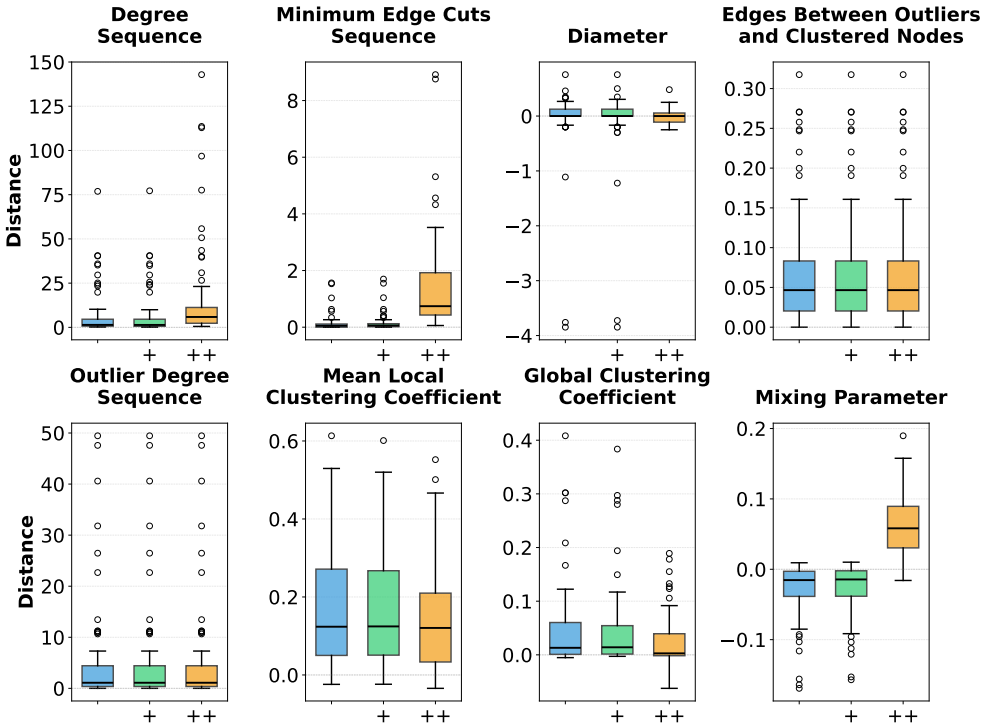


**Fig. 7 Accuracy of network statistics across 73 Netzschleuder networks.** Statistics of the networks generated by RECCS, RECCS+, and RECCS++ (indicated by '', '+', and '++' respectively), with reference to the 73 Netzschleuder networks. The y-axis shows the error metric for each statistic, as described in Table 2. Lower values indicate better fidelity.

minimum cut in the clusters of RECCS++ networks than the original. This constraint may also lead to lower mixing parameters in RECCS++ networks as well. While these discrepancies exist, there are some tradeoffs. RECCS++ networks are closer in diameter to the original network than RECCS and RECCS+. RECCS++ also exhibits slightly better accuracy in the clustering coefficients than the other two versions.

## 4.2 Performance and Scalability

Speedup curves of RECCS+ and RECCS++ on the Leiden CPM clustering of CEN with resolution 0.01, treated with CM are shown in Figure 8. The left axis shows runtime in seconds, and the right axis shows speedup relative to the original RECCS runtime on the same network-clustering pair. At 1 core, RECCS+ and RECCS++ achieve 2.9× and 6.8× speedups respectively, increasing to 5.8× and 11.5× at 64 cores. Since parallelism contributes approximately 2–3× speedup from 1 to 64 cores, the majority of the performance gain is attributable to other optimizations, such as the multiprocessed DAG formulation and CSR graph representation.

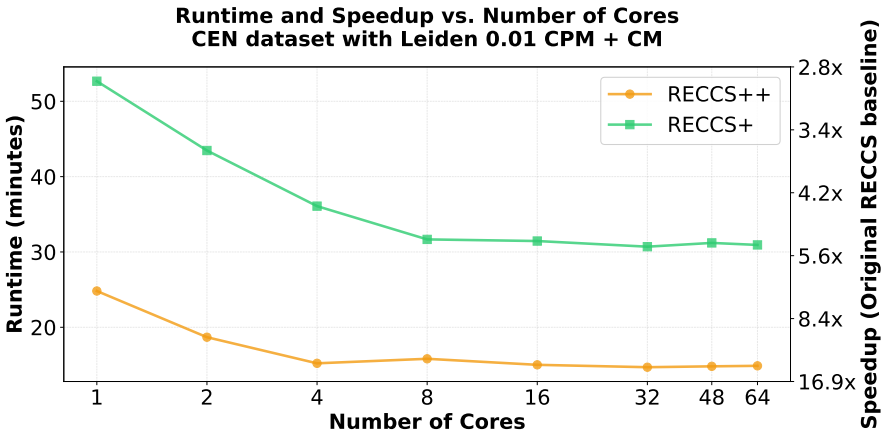


**Fig. 8 End-to-end runtime and speedup of RECCS+ and RECCS++ with respect to RECCS** on CEN, clustered with Leiden 0.01 + CM. Runtime is marked on the left y-axis, while speedup is marked on the right y-axis.

To further understand the details of the RECCS+ and RECCS++ runtimes, we used WandB[3] to profile the CPU and RAM usage of the two RECCS variants. Figure 9 shows the profiling results of RECCS+ and RECCS++ runtimes on the CEN network with Leiden 0.01 clustering, both with and without CM treatment. The pink shaded region shows the duration of the RECCS C++ module runtime. Both the lowest points in memory usage, and the highest points in CPU usage occur during the actual RECCS module runtime. The lower memory usage trend is likely due to the CSR graph representation taking less memory than the more general purpose `graph-tool` alternative. Higher CPU usage is due to the highest multithreading engagement being within the RECCS module, more so than during the SBM computation.

Moreover, there is a large portion of RECCS+ runtime after the RECCS module that is not present in RECCS++. This is due to the degree sequence matching stage in RECCS+ being run as a separate Python process after the RECCS module, while in RECCS++ the degree sequence matching is integrated into the RECCS module itself. This stage can take more than half of the RECCS+ runtime, as seen in Figure 9(a) and 9(c), and nearly half of the RECCS+ runtime in Figure 9(e) and 9(g). Therefore, much of the speedup of RECCS++ over RECCS+ can be attributed to this integration.

Table 3 shows the scaling results on the larger networks: Orkut, CEN, OC, and OCv2. RECCS+ and RECCS++ achieve their best performance improvement on CEN clustered with Leiden 0.01, with RECCS+ achieving a 49× speedup over RECCS and RECCS++ achieving a 139× speedup. Notably, on the two largest networks (OC and OCv2), both RECCS and RECCS+ failed to complete within the allocated time window, while RECCS++ successfully completed both networks in under 12 hours. This demonstrates RECCS++'s ability to scale to previously intractable network sizes, processing networks with over 100 million nodes and nearly 2 billion edges.

[3] https://wandb.ai

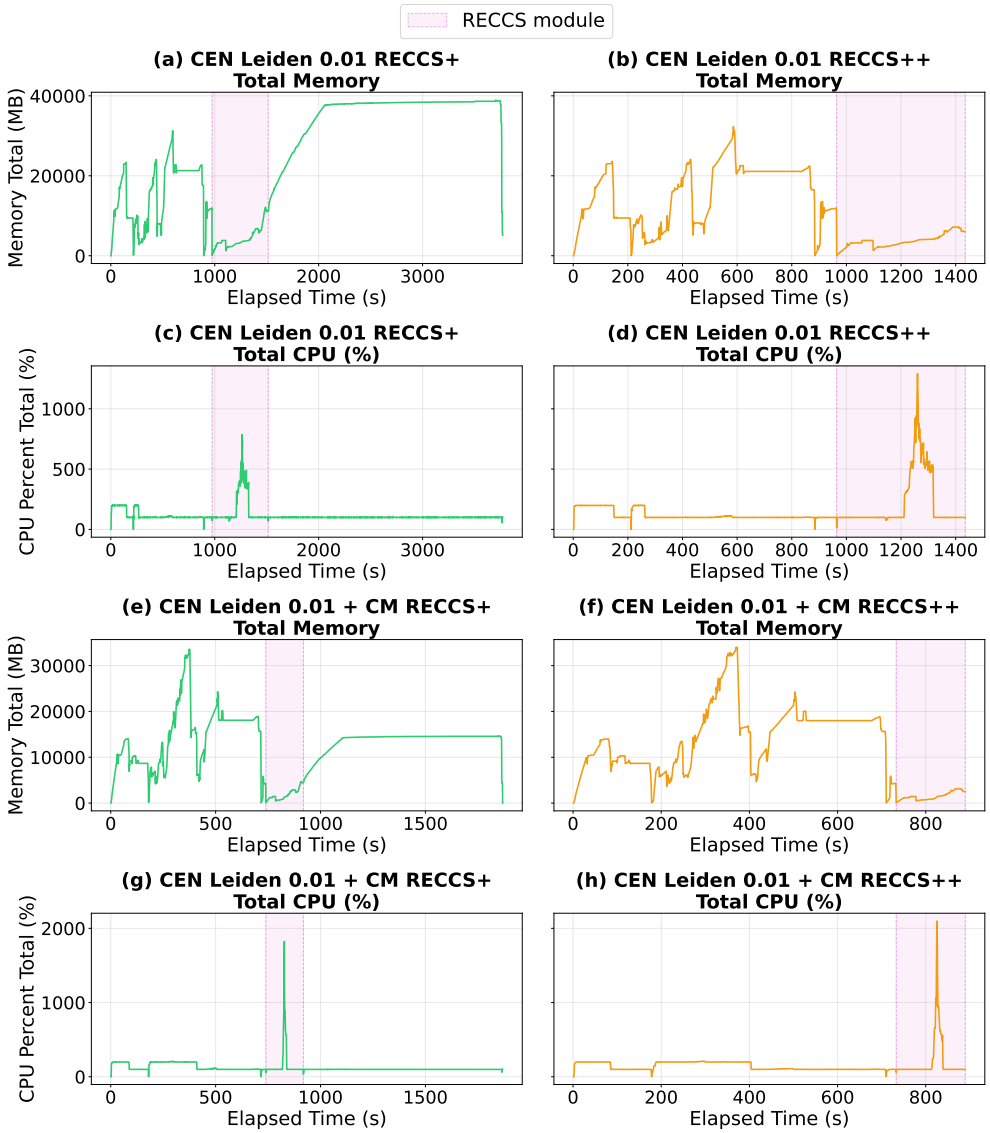


**Fig. 9 Profiling of RAM and CPU usage during the RECCS+ and RECCS++ runtimes** on CEN with a Leiden 0.01 clustering, both with and without CM treatment. The pink shaded region indicates the duration of the RECCS C++ module runtime. CEN Leiden 0.01 RECCS+ (a, c) uses a peak of 38.95GB overall and 12.13 GB during the RECCS module, and a peak of 786.3% CPU both overall and during the RECCS module. CEN Leiden 0.01 RECCS++ (b, d) uses a peak of 32.28GB overall and 7.2215GB during the RECCS module, and a peak of 1291.3% CPU both overall and during the RECCS module. CEN Leiden 0.01 + CM RECCS+ (e, g) uses a peak of 33.56GB overall and 4.73GB during the RECCS module, and a peak of 1824% CPU both overall and during the RECCS module. CEN Leiden 0.01 + CM RECCS++ (f, h) uses a peak of 33.99GB overall and 3.11GB during the RECCS module, and a peak of 2098.7% CPU both overall and during the RECCS module.

**Table 3 Table of runtimes per RECCS version across all large network datasets.** Times are in d-hh:mm:ss format. DNC means 'Did Not Complete'.

| Network | Clustering | RECCS | RECCS+ | RECCS++ |
|---|---|---|---|---|
| **Orkut** | Leiden $r = 0.01$ + CM | 6:47:11 | 2:18:08 | 22:44 |
| | Leiden $r = 0.01$ | 8:33:41 | 2:29:13 | 23:51 |
| **CEN** | Leiden $r = 0.01$ + CM | 2:48:48 | 30:53 | 15:02 |
| | Leiden $r = 0.01$ | 2-5:23:59 | 1:04:17 | 23:35 |
| **OC** | Leiden $r = 0.01$ + CM | DNC | DNC | 6:08:46 |
| | Leiden $r = 0.01$ | DNC | DNC | 7:02:58 |
| **OCv2** | Leiden $r = 0.01$ + CM | DNC | DNC | 7:59:06 |
| | Leiden $r = 0.01$ | DNC | DNC | 11:16:12 |

## 4.3 Reclustering Evaluation

In this experiment, we use synthetic network generation with the exact purpose outlined in the beginning of this paper: to evaluate community detection algorithms with respect to a generated ground truth. As such we follow a reclustering evaluation pipeline like the one shown in Figure 1. We take the seven networks from Table 1 designated for the reclustering experiment. These networks are clustered with Leiden CPM with resolution 0.01, Leiden optimizing modularity, and Iterative K-Core clustering (IKC) [16], both with and without CM treatment. RECCS, RECCS+, and RECCS++ networks are generated for each network, clustering and treatment. Then each generated network is reclustered with one of the three methods. If the original clustering used CM treatment, the reclustering will use CM treatment as well. For each clustering reclustering pair, both the normalized mutual information (NMI) and the adjusted rand index (ARI) are computed.

The NMI and ARI metrics are then averaged across the networks, creating an mean value per clustering-reclustering pair. From this, we derive 12 heatmaps: one per combination of version (3), treatment (2), and metric (2). Each heatmap is of size 3x3 as each row is a clustering method and each column is a reclustering method.

Then, we use the Frobenius norm to derive four 3x3 heatmaps to show the difference across RECCS versions across NMI, ARI, NMI with CM treatment, and ARI with CM treatment. The Frobenius norm is a normalized matrix difference, where we take two heatmaps as input:

$$\|A - B\|_F = \sqrt{\sum_{i=1}^{3}\sum_{j=1}^{3}(A_{ij} - B_{ij})^2}$$

Where $A$ and $B$ are two confusion matrices of the same metric across any of the three RECCS versions. The final heatmap grid is shown in Figure 10. All heatmaps in blue show a per-version NMI heatmap comparing base clusters to their reclusterings. All heatmaps in red show ARI comparisons. Frobenius norm heatmaps comparing RECCS are shown in green in the bottom row of the grid.

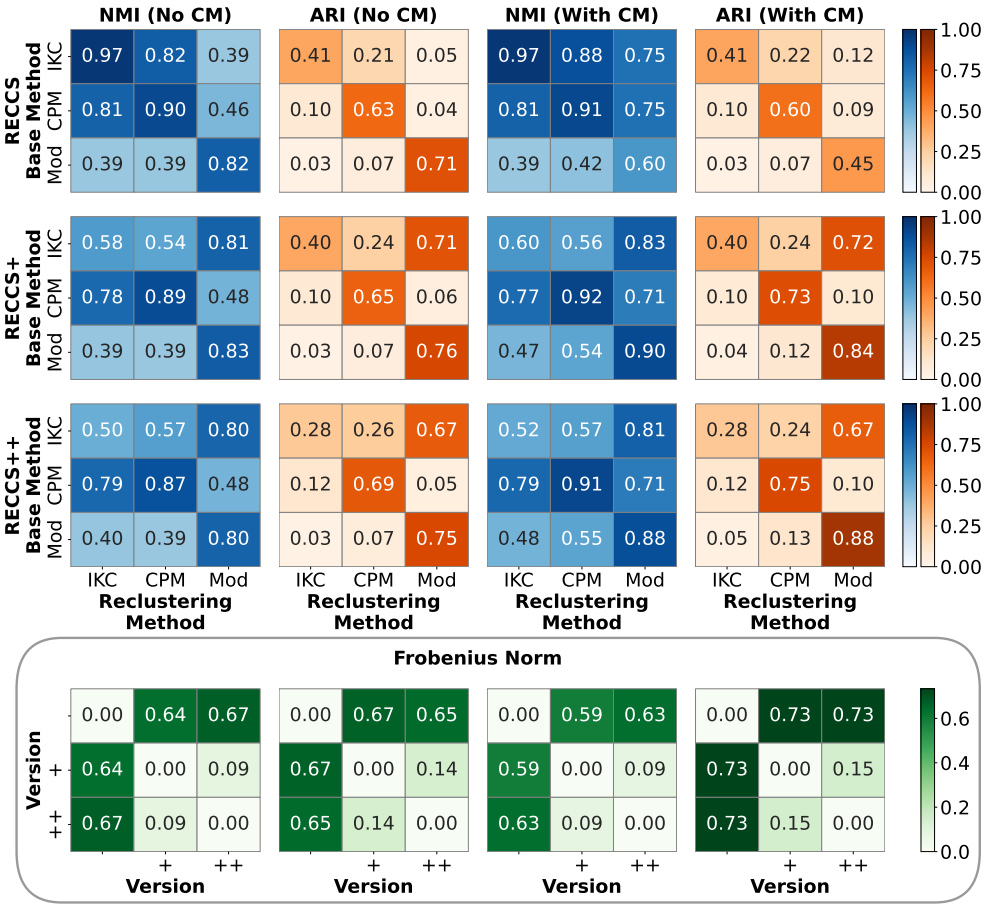


**Fig. 10 Reclustering accuracy comparison across RECCS, RECCS+, and RECCS++.** Rows 1–3 show NMI (blue) and ARI (red) confusion matrices for RECCS, RECCS+, and RECCS++ respectively, comparing original clusterings (IKC, Leiden CPM 0.01, Leiden Modularity) against reclustering methods. Columns 1–2 show results without CM treatment, columns 3–4 with CM treatment. Bottom row (green) shows pairwise Frobenius norm differences between RECCS variants for each metric and treatment condition.

The Frobenius norm matrices show that the confusion matrices of RECCS+ and RECCS++ show more similarity with each other than with the original RECCS. However, the top three rows show that the deviation in matrix similarity mainly comes from the IKC clustering results. This can be due in part to either the generator itself or the clustering method. However, since both Leiden CPM and Leiden Modularity show much smaller Frobenius norm differences across RECCS versions, it is likely that the discrepancy is mainly due to the sensitivity of IKC to network structure changes, as it relies on k-core decomposition which can be significantly affected by even small perturbations in edge structure. In contrast, Leiden CPM and Leiden Modularity appear

more robust to the structural variations introduced by different RECCS versions, resulting in more consistent clustering outcomes across synthetic networks.

Despite preserving key network statistics (as shown in Section 4), RECCS+ and RECCS++ produce different reclustering patterns than RECCS, particularly for IKC. This demonstrates that reclustering evaluation can reveal clustering algorithm sensitivities to structural variations beyond standard network metrics. The consistency between RECCS+ and RECCS++ results, combined with their computational efficiency and ability to scale to larger networks, validates their use for this type of robustness analysis in reclustering evaluation, and opens the door for such analysis on larger-scale networks that were previously infeasible with RECCS.

# 5 Discussion: Scalability Enables New Directions

The scalability achievements demonstrated in the previous sections, particularly RECCS++'s ability to handle networks with over 100 million nodes, fundamentally expand the potential applications of the RECCS methodology beyond its original evaluation-focused design. While the following discussion applies conceptually to the RECCS approach as a whole, RECCS++'s superior scalability makes it the most suitable implementation for these applications. In this section, we explore how viewing the RECCS methodology through an encoder-decoder lens reveals opportunities for broader impact in network generation tasks.

## 5.1 The Encoder-Decoder Abstraction

Given that the RECCS methodology has achieved scalability through RECCS+ and RECCS++, there are new doors that have been opened with respect to its generalization. The RECCS approach can be conceptualized as an encoder-decoder architecture, analogous to models widely used in image generation [35, 36] and increasingly adopted in graph machine learning [37, 38]. In traditional encoder-decoder models, an encoder embeds training data into a latent space, while a decoder generates new data from these latent embeddings. The RECCS methodology follows a similar paradigm as a simplified, non-deep encoder-decoder framework for network generation. The encoding phase comprises the statistics and cluster probability matrix computations that precede the RECCS module and SBMs, respectively, as these steps extract network and cluster properties that serve as latent features. The RECCS module and SBMs then function as the decoder, generating synthetic networks conditioned on these latent representations.

## 5.2 Extensions and Adaptations

The RECCS methodology has a limitation in that it is a single-reference network generator designed to mimic the original network's characteristics. Without constraints on perturbation, the optimal synthetic network that the RECCS approach could generate would simply be the original reference network itself, which limits its utility as a benchmark for evaluating community detection algorithms on diverse network structures. However, this encoder-decoder abstraction enables several natural extensions

of the RECCS methodology beyond its current single-network formulation, and can open up possibilities to perturbations in the network for more robust evaluation. First, the RECCS approach can be adapted to generate scaled versions of input networks through normalization in the encoding phase. Rather than storing absolute network properties, the encoder would extract normalized or spectral properties such as spectral mincut values instead of raw mincut counts, and density metrics instead of explicit node and edge counts. The decoder could then rescale these normalized features by a dilation or shrinkage factor to pass into the RECCS module to generate networks of varying sizes while preserving structural characteristics.

Second, the RECCS methodology could be extended to learn from distributions of networks rather than individual instances. In this formulation, the encoder would aggregate statistics across multiple training networks to infer distributions over key structural characteristics: cluster size distributions, inter-cluster edge probability matrices, and per-cluster mincut distributions. The decoder would then sample values from these learned distributions and pass them into the RECCS module to generate diverse synthetic networks that capture the variability present in the training data. This extension would address a key limitation of the current approach, which requires a single input network and may therefore impact the robustness and generalizability of synthetic network generation for evaluation purposes. While deep generative models can learn from network distributions, the demonstrated scalability of the RECCS module suggests this approach could offer improved performance over existing deep learning methods, particularly for large network generation tasks.

Lastly, the encoder-decoder abstraction of the RECCS methodology naturally positions RECCS++ as a complementary component to deep learning methods for network generation. While deep learning models excel at learning complex distributions from training data, they face computational challenges when generating large-scale networks directly. The RECCS module and SBM components of RECCS++ address this limitation by providing a scalable decoder that can efficiently generate networks from learned parameters. In this hybrid architecture, the deep learning component would serve as the encoder, learning to infer these distributional parameters from a corpus of training networks. The RECCS module and SBMs from RECCS++ would then serve as the decoder, taking these learned parameters to generate networks efficiently. This division of labor exploits the complementary strengths of both approaches: deep learning's capacity for representation learning and pattern recognition, paired with RECCS++'s demonstrated efficiency and scalability for large-scale generation. Such a hybrid framework could enable generation of synthetic networks at scales currently intractable for purely deep learning-based methods, extending network generation capabilities to networks with hundreds of millions of nodes. Moreover, the deterministic nature of RECCS++ as a decoder provides interpretability advantages over end-to-end deep generative models, as the network generation process remains transparent and the learned parameters directly correspond to structural properties of interest.

### 5.3 Challenges and Future Directions

However, implementing such distribution-based generation presents several technical challenges. The primary difficulty lies in the interdependencies among network properties: cluster sizes, mincut values, and edge densities cannot be sampled independently, as they are structurally constrained by one another. For instance, the dimensionality of the mincut distribution depends on the number of clusters, which itself varies with network size. This creates a hierarchy of coupled distributions—a "distribution of distributions"—where sampling decisions at one level constrain the valid range of values at another. Additionally, when scaling networks through dilation or shrinkage, the number of clusters typically changes non-linearly with network size, requiring variable-length distributions that adapt to the target scale. Addressing these challenges would require careful modeling of the joint distribution over network properties, potentially through hierarchical generative models or constraint-aware sampling procedures. Despite these complications, the encoder-decoder abstraction provides a principled framework for such extensions, making the RECCS methodology readily adaptable to more sophisticated generation scenarios as it matures.

## 6 Limitations and Future Work

In the previous section, several limitations were touched upon, but they all were inherent to the nature of RECCS, RECCS+ and RECCS++ being single reference network synthetic network generators. As such, we proposed potential future work in developing an abstraction of the RECCS methodology to adapt it to problems that avoid these limitations.

As shown in Figure 9 and discussed in Section 4.2, memory profiling reveals that the RECCS module's portion of the end-to-end runtime exhibits the lowest memory usage, likely due to its efficient CSR graph representation. In contrast, peak memory consumption occurs during the degree sequence fitting stage in the case of RECCS+ on CEN clustered with Leiden with resolution 0.01, and during the SBM stages for all other configurations. Both high-memory scenarios involve graph-tool's general-purpose graph representation, which trades higher memory overhead for broader functionality and faster execution of graph operations. Future work could address overhead and memory consumption caused by Python process calls from C++, by reimplementing the Python processed in C++ and integrating them directly into the main RECCS++ runtime. This would eliminate the cost of spawning Python processes from C++ while enabling the use of more memory-efficient graph representations throughout the pipeline.

A more concrete limitation is that while we scaled up RECCSv1's explicit degree sequence fitting approach, we did not extend the same optimizations to RECCSv2, which uses SBMs for degree sequence fitting. Future work should investigate whether the parallelization strategies developed for RECCS+ can be effectively applied to SBM-based degree fitting, and whether the tradeoffs between accuracy and speed differ at scale.

# 7 Conclusion

In this paper, we have attained scalability through the use of shared memory parallelism and algorithmic reformulation of RECCS, resulting in RECCS+ and RECCS++. In networks such as OC and OCv2 where RECCS and RECCS+ could not complete, RECCS++ was able to complete within a twelve hour window, demonstrating its capability to handle previously intractable network scales.

We find that RECCS+ exhibits algorithmic fidelity with RECCS, and while RECCS++ exhibits reduced algorithmic fidelity, it maintains comparable network quality, showing several tradeoffs in metric accuracy measures. The reclustering accuracy confusion matrices reveal differences between the original RECCS networks and the RECCS+ and RECCS++ networks, particularly when IKC is used as the clustering or reclustering method. However, these differences reflect the fundamental tension between exact replication and practical scalability. The magnitude and nature of these differences also depend on the sensitivity of specific clustering algorithms to network perturbations—some methods like IKC may be more sensitive to minor structural variations than others.

Most importantly, RECCS+ and RECCS++ enable researchers to evaluate algorithms on network scales that were previously inaccessible, making it possible to assess algorithmic behavior in realistic, large-scale settings rather than being limited to smaller toy examples. Furthermore, the newfound scalability opens doors for new adaptations and applications of our synthetic network generator beyond the current scope of this work.


# Acknowledgements

We thank the Insper-Illinois partnership for providing support for this project. We thank George Chacko, Tandy Warnow, The-Anh Vu Le, and Ian Wei Chen from the Siebel School of Computing and Data Science for helpful discussions and advice. This research was funded in part by NSF grant numbers CCF-2109988, OAC-2402560, and CCF-2453324.


# Declarations

Claude Sonnet 4.5, Claude Code, and GitHub Copilot, all AI assistants and agents created by Anthropic and GitHub, were used to assist in developing this software.

# References


[1] Fortunato, S., Newman, M.E.J.: 20 years of network community detection. Nat. Phys. **18**(8), 848–850 (2022) https://doi.org/10.1038/s41567-022-01716-7

[2] Karataş, A., Şahin, S.: Application areas of community detection: A review. In: 2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT), pp. 65–70 (2018). https://doi.org/10.1109/IBIGDELFT.2018.8625349

[3] Waltman, L., Eck, N.J.: A new methodology for constructing a publication-level classification system of science. J. Am. Soc. Inf. Sci. Technol. **63**(12), 2378–2392 (2012) https://doi.org/10.1002/asi.22748

[4] Blondel, V.D., Guillaume, J.-L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment **2008**(10), 10008 (2008) https://doi.org/10.1088/1742-5468/2008/10/p10008

[5] Newman, M.E.J.: Modularity and community structure in networks. Proc. Natl. Acad. Sci. U. S. A. **103**(23), 8577–8582 (2006) https://doi.org/10.1073/pnas.0601602103

[6] Traag, V.A., Waltman, L., Eck, N.J.: From louvain to leiden: guaranteeing well-connected communities. Sci. Rep. **9**(1), 5233 (2019) https://doi.org/10.1038/s41598-019-41695-z

[7] Traag, V.A., Van Dooren, P., Nesterov, Y.: Narrow scope for resolution-limit-free community detection. Phys. Rev. E Stat. Nonlin. Soft Matter Phys. **84**(1 Pt 2), 016114 (2011) https://doi.org/10.1103/PhysRevE.84.016114

[8] Park, M., Tabatabaee, Y., Ramavarapu, V., Liu, B., Pailodi, V.K., Ramachandran, R., Korobskiy, D., Ayres, F., Chacko, G., Warnow, T.: Well-connectedness and community detection. PLOS Complex Syst **1**(3), 0000009 (2024) https://doi.org/10.1371/journal.pcsy.0000009

[9] Peixoto, T.P.: Bayesian Stochastic Blockmodeling. Wiley (2019). https://doi.org/10.1002/9781119483298.ch11

[10] Lancichinetti, A., Fortunato, S., Radicchi, F.: Benchmark graphs for testing community detection algorithms. Physical Review E **78**(4) (2008) https://doi.org/10.1103/physreve.78.046110

[11] Karrer, B., Newman, M.E.J.: Stochastic blockmodels and community structure in networks. Phys. Rev. E **83**, 016107 (2011) https://doi.org/10.1103/PhysRevE.83.016107

[12] Anne, L., Vu-Le, T.-A., Park, M., Warnow, T., Chacko, G.: Synthetic Networks That Preserve Edge Connectivity (2024). https://arxiv.org/abs/2408.13647

[13] The-Anh Vu-Le, Anne, L., Chacko, G., Warnow, T.: EC-SBM synthetic network generator. Appl. Netw. Sci. **10**(1) (2025) https://doi.org/10.1007/s41109-025-00701-2

[14] Anne, L., Vu-Le, T.-A., Park, M., Warnow, T., Chacko, G.: RECCS: Realistic Cluster Connectivity Simulator for Synthetic Network Generation (2025). https://arxiv.org/abs/2502.02050

[15] Leskovec, J., Sosič, R.: Snap: A general-purpose network analysis and graph-mining library. ACM Trans. Intell. Syst. Technol. **8**(1) (2016) https://doi.org/10.1145/2898361

[16] Wedell, E., Park, M., Korobskiy, D., Warnow, T., Chacko, G.: Center–periphery structure in research communities. Quantitative Science Studies **3**(1), 289–314 (2022) https://doi.org/10.1162/qss_a_00184

[17] Peixoto, T.P.: The Netzschleuder network catalogue and repository. Zenodo (2020). https://doi.org/10.5281/zenodo.7839981

[18] Vu-Le, T.-A., Park, M., Chen, I., Chacko, G., Warnow, T.: Using Stochastic Block Models for Community Detection: The issue of edge-connectivity (2025). https://arxiv.org/abs/2508.03843

[19] Erdős, P., Rényi, A.: On random graphs. I. Publ. Math. Debrecen **6**(3-4), 290–297 (2022) https://doi.org/10.5486/pmd.1959.6.3-4.12

[20] Yang, J., Leskovec, J.: Defining and Evaluating Network Communities based on Ground-truth (2012). https://arxiv.org/abs/1205.6233

[21] Korobskiy, D., Chacko, G.: Curated Open Citations Dataset. https://doi.org/10.13012/B2IDB-6389862_V1

[22] Peroni, S., Shotton, D.: Opencitations, an infrastructure organization for open scholarship. Quantitative Science Studies **1**(1), 428–444 (2020) https://doi.org/10.1162/qss_a_00023

[23] Mohasel Arjomandi, H., Korobskiy, D., Chacko, G.: Parsed Open Citations and PubMed Data. https://doi.org/10.13012/B2IDB-5216575_V1

[24] Ramavarapu, V., Ayres, F.J., Park, M., Pailodi, V.K., Lamy, J.A.C., Warnow, T., Chacko, G.: CM++ - a meta-method for well-connected community detection. J. Open Source Softw. **9**(93), 6073 (2024) https://doi.org/10.21105/joss.06073

[25] Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining. KDD '05, pp. 177–187. Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1081870.1081893

[26] Gehrke, J., Ginsparg, P., Kleinberg, J.: Overview of the 2003 kdd cup. SIGKDD Explor. Newsl. **5**(2), 149–151 (2003) https://doi.org/10.1145/980972.980992

[27] Rozemberczki, B., Davies, R., Sarkar, R., Sutton, C.: Gemsec: Graph embedding with self clustering. In: Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2019, pp. 65–72

(2019). https://doi.org/10.48550/arXiv.1802.03997

[28] Rozemberczki, B., Sarkar, R.: Twitch Gamers: a Dataset for Evaluating Proximity Preserving and Structural Role-based Node Embeddings (2021). https://doi.org/10.48550/arXiv.2101.03091

[29] Rozemberczki, B., Allen, C., Sarkar, R.: Multi-scale Attributed Node Embedding (2019). https://doi.org/10.48550/arXiv.1909.13021

[30] Leskovec, J., McAuley, J.: Learning to discover social circles in ego networks. In: Pereira, F., Burges, C.J., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems, vol. 25. Curran Associates, Inc., NY (2012). https://doi.org/10.48550/arXiv.1210.8182

[31] Peixoto, T.P.: The graph-tool python library. figshare (2017). https://doi.org/10.6084/m9.figshare.1164194.v14

[32] Csárdi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal **Complex Systems**, 1695 (2006)

[33] Antonov, M., Csárdi, G., Horvát, S., Müller, K., Nepusz, T., Noom, D., Salmon, M., Traag, V., Welles, B.F., Zanini, F.: igraph enables fast and robust network analysis across programming languages. arXiv preprint arXiv:2311.10260 (2023) https://doi.org/10.48550/arXiv.2311.10260

[34] Staudt, C.L., Sazonovs, A., Meyerhenke, H.: NetworKit: A Tool Suite for Large-scale Complex Network Analysis (2015). https://arxiv.org/abs/1403.3005

[35] Ho, J., Jain, A., Abbeel, P.: Denoising Diffusion Probabilistic Models (2020). https://arxiv.org/abs/2006.11239

[36] Kingma, D.P., Welling, M.: An introduction to variational autoencoders. Foundations and Trends® in Machine Learning **12**(4), 307–392 (2019) https://doi.org/10.1561/2200000056

[37] Kipf, T.N., Welling, M.: Variational Graph Auto-Encoders (2016). https://arxiv.org/abs/1611.07308

[38] Liu, C., Fan, W., Liu, Y., Li, J., Li, H., Liu, H., Tang, J., Li, Q.: Generative Diffusion Models on Graphs: Methods and Applications (2023). https://arxiv.org/abs/2302.02591