Streaming Graph Challenge: Stochastic Block Partition - draft -

Edward Kao, Vijay Gadepally, Michael Hurley, Michael Jones, Jeremy Kepner, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Siddharth Samsi, William Song, Diane Staheli, Steven Smith
MIT Lincoln Laboratory, Lexington, MA

***Abstract*—An important objective for analyzing real-world graphs is to achieve scalable performance on large, streaming graphs. A challenging and relevant example is the graph partition problem. As a combinatorial problem, graph partition is NP-hard, but existing relaxation methods provide reasonable approximate solutions that can be scaled for large graphs. Competitive benchmarks and challenges have proven to be an effective means to advance state-of-the-art performance and foster community collaboration. This paper describes a graph partition challenge with a baseline partition algorithm of sub-quadratic complexity. The algorithm employs rigorous Bayesian inferential methods based on a statistical model that captures characteristics of the real-world graphs. This strong foundation enables the algorithm to address limitations of well-known graph partition approaches such as modularity maximization. This paper describes various aspects of the challenge including: (1) the data sets and streaming graph generator, (2) the baseline partition algorithm with pseudocode, (3) an argument for the correctness of parallelizing the Bayesian inference, (4) different parallel computation strategies such as node-based parallelism and matrix-based parallelism, (5) evaluation metrics for partition correctness and computational requirements, (6) preliminary timing of a Python-based demonstration code and the open source C++ code, and (7) considerations for partitioning the graph in streaming fashion. Data sets and source code for the algorithm as well as metrics, with detailed documentation are available at GraphChallenge.org.**

I. Introduction

In the era of big data, analysis and algorithms often need to scale up to large data sets for real-world applications. With the rise of social media and network data, algorithms on graphs face the same challenge. Competitive benchmarks and challenges have proven to be an effective means to advance state-of-the-art performance and foster community collaboration. Previous benchmarks such as Graph500 [1] and the Pagerank Pipeline [2] are examples of such, targeting analysis of large graphs and focusing on problems with sub-quadratic complexity, such as search, path-finding, and PageRank computation. However, some analyses on graphs with valuable applications are NP-hard. The graph partition and the graph isomorphism (i.e. matching) problems are well-known examples. Although these problems are NP-hard, existing relaxation methods provide good approximate solutions that can be scaled to large graphs [3], [4], especially with the aid of high performance computing hardware platform such as massively parallel CPUs and GPUs. For example, the 10th DIMACS Implementation Challenge [5] resulted in substantial participation in the graph partition problem, mostly with solutions based on modularity maximization. To promote algorithmic and computational advancement in these two important areas of graph analysis, our team has implemented a challenge for graph isomorphism [6] and graph partition at GraphChallenge.org. This paper describes the graph partition challenge with a recommended baseline partition algorithm of sub-quadratic complexity. Furthermore, the algorithm employs rigorous Bayesian inferential methods based on the stochstic blockmodels that capture characteristics of the real-world graphs. Participants are welcome to submit solutions based on other partition algorithms as long as knowledge on the true number of communities (i.e. blocks) is not assumed. All entries should be submitted with performance evaluation on the challenge data sets using the metrics described in Section V.

Graph partition, also known as community detection and graph clustering, is an important problem with many real-world applications. The objective of graph partition is to discover the distinct community structure of the graph, specifically the community membership for each node in the graph. The partition gives much insight to the interactions and relationships between the nodes and enables detection of nodes belonging to certain communities of interest. Much prior work has been done in the problem space of graph partition, with a comprehensive survey in [7]. The most well-known algorithm is probably the spectral method by [8] where partition is done through the eigenspectrum of the modularity matrix. Most of the existing partition algorithms work through the principle of graph modularity where the graph is partitioned into communities (i.e. modules) that have much stronger interactions within them than between them.

*This material is based upon work supported by the Defense Advanced Research Projects Agency under Air Force Contract No. FA8721-05-C-0002. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense.

Typically, partitioning is done by maximizing the graph modularity [9]. [10] extends the concept of modularity for time-dependent, multiscale, and multiplex graphs. Modularity maximization is an intuitive and convenient approach, but has inherent challenges such as resolution limit on the size of the detectable communities [11], degeneracies in the objective function, and difficulty in identifying the optimal number of communities [12].

To address these challenges, recent works perform graph partition through membership estimation based on generative statistical models. For example, [13], [14], [15], [16] estimate community memberships using the degree corrected stochastic blockmodels [17], and [18] proposes a mixed-memberships estimation procedure by applying tensor methods to the mixed-membership stochastic blockmodels [19]. The baseline partition algorithm for this challenge is based on [14], [15], [16], because of its rigorous statistical foundation and subquadratic computational requirement. Under this approach, each community is represented as a "block" in the model. Going forward, this paper will use the term "block" as the nomenclature for a community or a graph cluster.

When some nodes in the graph have known memberships a priori, these nodes can serve as "cues" in the graph partition problem. [20] is an example of such using random walks on graph. This challenge will focus on the graph partition problem where such cues are not available.

In many real-world applications, graph data arrives in streaming fashion over time or stages of sampling [21]. This challenge addresses this aspect by providing streaming graph data sets and recommending a baseline partition algorithm that is suitable for streaming graphs under the Bayesian inference paradigm.

This paper describes the graph partition challenge in detail, beginning with Section II on the data sets and streaming graph generator. Section III describes the baseline partition algorithm, including pseudocode on the core Bayesian updates. Section IV focuses on the parallel computation of the baseline algorithm, argues for the correctness of parallelizing the Bayesian updates, then proposes parallel computation strategies such as node-based parallelism and matrix-based parallelism. Section V describes the evaluation metrics for both partition correctness and computational requirements, including a preliminary timing of a Python-based demonstration code and the open source C++ code [22]. Considerations for partitioning the graph in streaming fashion are given throughout the paper.

II. Data Sets

The data sets for this challenge consist of graphs of varying sizes and characteristics. Denote a graph $\boldsymbol{G} = (\boldsymbol{\mathcal{V}}, \boldsymbol{\mathcal{E}})$, with the set $\boldsymbol{\mathcal{V}}$ of N nodes and the set $\boldsymbol{\mathcal{E}}$ of E edges. The edges, represented by a $N \times N$ adjacency matrix $\boldsymbol{A}$, can be either directed or undirected, binary or weighted. Specifically, A_{ij} is the weight of the edge from node i to node j. A undirected graph will have a symmetric adjacency matrix.

In order to evaluate the partition algorithm implementation on graphs with a wide range of realistic characteristics, graphs are generated according to a truth partition $\boldsymbol{b}^\dagger$ of $B^\dagger$ blocks (i.e. clusters), based on the degree-corrected stochastic blockmodels by Karrer and Newman in [17]. Under this generative model, each edge, A_{ij}, is drawn from a Poisson distribution of rate λ_{ij} governed by the equations below:

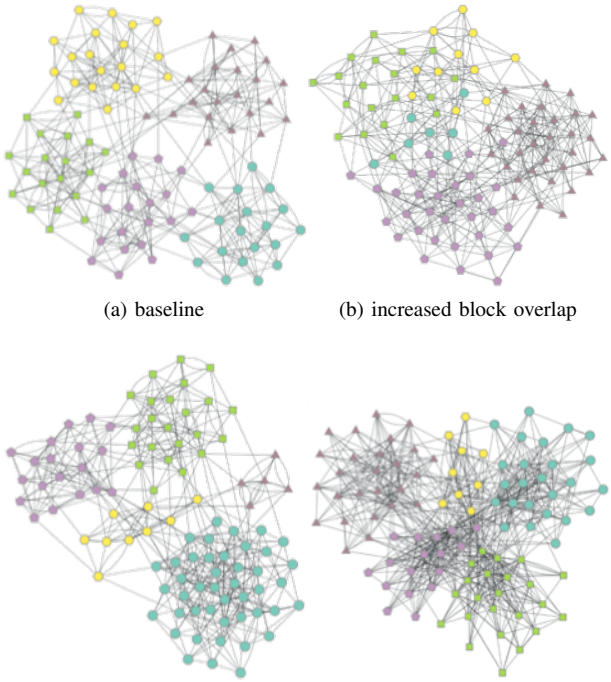
$$A_{ij} \sim \text{Poisson}(\lambda_{ij}) \tag{1}$$

$$\lambda_{ij} = \theta_i \theta_j \Omega_{b_i b_j} \tag{2}$$

where θ_i is a correction term that adjusts node i's expected degree, $\Omega_{b_i b_j}$ the strength of interaction between block b_i and b_j, and b_i the block assignment for node i. The degree-corrected stochastic blockmodels enable the generation of graphs with characteristics and variations consistent with real-world graphs. The degree correction term for each node can be drawn from a Power-Law distribution with an exponent between -3 and -2 to capture the degree distribution of realistic, scale-free graphs [23]. The block interaction matrix $\boldsymbol{\Omega}$ specifies the strength of within- and between-block (i.e. community) interactions. Stronger between-block interactions will increase the block overlap, making the block partition task more difficult. Lastly, the block assignment for each node (i.e. the truth partition $\boldsymbol{b}^\dagger$) can be drawn from a multinomial distribution with a Dirichlet prior that determines the amount of variation in size between the blocks. Figure 1 shows generated graphs of various characteristics by adjusting the parameters of the generator. These parameters server as "knobs" that can be dialed to capture a rich set of characteristics for realism and also for adjusting the difficulty of the block partition task.

Real-world graphs will also be included in the data sets. Since the truth partition is not available in most real-world graphs, generated graphs with truth will be embedded with the real-world graphs. While the entire graph will be partitioned, evaluation on the correctness of the partition will be done only on the generated part of the hybrid graph. Embedding will be done by adding edges between nodes in the real-world graph and the generated graph, with a relatively small probability proportional to the product of both node degrees.

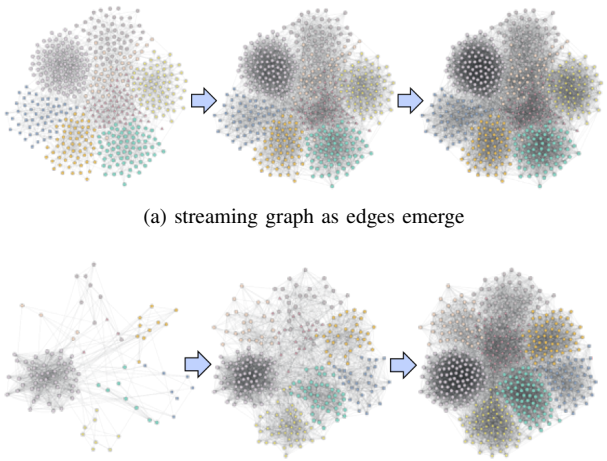
In real-world applications, graph data often arrives in streaming fashion, where parts of the input graph become available at different stages. This happens as interactions and relationships take place and are observed over time,



(a) baseline (b) increased block overlap

(c) higher block size variation (d) more high degree nodes

Fig. 1. Generated graphs with varying characteristics. Nodes are colored and shaped according to their true block assignments. Graphs are typically much larger. Small graphs are shown here for the purpose of demonstration. For simplicity and clarity, the edge directions (i.e. arrows) are not displayed.



(a) streaming graph as edges emerge

(b) streaming graph with snowball sampling

Fig. 2. Streaming graphs generated in two ways: (a) as edges emerge over time and (b) as the graph is explored from starting point(s).

or as data is collected incrementally by exploring the graph from starting points (e.g. snowball sampling) [21]. Streaming graph data sets in this challenge are generated in both ways, as demonstrate in Figure 2. The partition algorithm should process the streaming graph at each stage and ingest the next stage upon completion of the current stage. Performance evaluated using the metrics in Section V should be reported at each stage of the processing. For efficiency, it is recommended that the partition algorithm leverages partitions from the previous stage(s) to speed up processing at the current stage. The baseline partition algorithm for this challenge is a natural fit for streaming processing, as discussed in Section III.

III. Baseline Algorithm

This section described the recommended baseline partition algorithm, although participants are welcome to submit solutions based on other partition algorithms as long as knowledge on the true number of blocks is not assumed.

The baseline graph partition algorithm for this challenge, chosen for its rigorous statistical foundation and sub-quadratic, $O(E \log^2 E)$, computational requirement, is developed by Tiago Peixoto in [14], [15], [16] based on the degree-corrected stochastic blockmodels by Karrer and Newman in [17]. Given the input graph, the algorithm partitions the nodes into B blocks (i.e. clusters or communities), by updating the nodal block assignment represented by vector $\boldsymbol{b}$ of N elements where $b_i \in \{1, 2, ..., B\}$, and the inter-block and intra-block edge count matrix (typically sparse in a large graph) represented by $\boldsymbol{M}$ of size $B \times B$, where each element M_{ij} represents the number or the total weight of edges going from block i to block j. The diagonal elements represent the edge counts within each block. For conciseness, this matrix will be referred to as the inter-block edge count matrix going forward. The goal of the algorithm is to recover the truth partition $\boldsymbol{b}^\dagger$ of $B^\dagger$ blocks (i.e. clusters).

The algorithm performs a Fibonacci search (i.e. golden section search) [24] through different numbers of blocks B and attempts to find the minimum description length partition. The best overall partition $\boldsymbol{b}^*$ with the optimal number of block B^* minimize the total description length of the model and the observed graph (i.e. entropy of the fitted model). To avoid being trapped in local minima, the algorithm starts with each node in its own block (i.e. $B = N$) and the blocks are merged at each step of the Fibonacci search, followed by iterative Monte Carlo Markov Chain (MCMC) updates on the block assignment for each node to find the best partition for the current number of blocks. The block-merge moves and the nodal updates are both governed by the same underlying log posterior probability of the partition given the observed graph:

$$p(\boldsymbol{b}|G) \propto \sum_{t_1,t_2} M_{t_1 t_2} \log\left(\frac{M_{t_1 t_2}}{d_{t_1,\text{out}} d_{t_2,\text{in}}}\right) \quad (3)$$

The log posterior probability is a summation over all pairs of blocks t_1 and t_2 where $d_{t_1,\text{out}}$ is the total out-

degree for block t_1 and $d_{t_2,\text{in}}$ is the total in-degree for block t_2. Note that in computing the posterior probabilities on the block assignments, the sufficient statistics for the entire graph is only the inter-block edge counts, giving much computational advantage for this algorithm. Another nice property of the log posterior probability is that it is also the negative entropy of the fitted model. Therefore, maximizing the posterior probability of the partition also minimizes the overall entropy, fitting nicely into the minimum description length framework. The block-merge moves and the nodal block assignment updates are described in detail next, starting with the nodal updates.

A. Nodal Block Assignment Updates

The nodal updates are performed using the Monte Carlo Markov Chain (MCMC), specifically with Gibbs sampling and the Metropolis-Hastings algorithm since the partition posterior distribution in Equation 3 does not have a closed-form and is best sampled one node at a time. At each MCMC iteration, the block assignment of each node i is updated conditional on the assignments of the other nodes according to the conditional posterior distribution: $p(b_i|\boldsymbol{b}_{-i}, G)$. Specifically, the block assignment b_i for each node i is updated based on the edges to its neighbors, $\boldsymbol{A}_{i\mathcal{N}_i}$ and $\boldsymbol{A}_{\mathcal{N}_i i}$, the assignments of its neighbors, $\boldsymbol{b}_{\mathcal{N}_i}$, and the inter-block edge count, $\boldsymbol{M}$. For each node i, the update begins by proposing a new block assignment. To increase exploration, a block is randomly chosen as the proposal with some predefined probability. Otherwise, the proposal will be chosen from the block assignments of nodes nearby to i. The new proposal will be considered for acceptance according to how much it changes the log posterior probability. The acceptance probability is adjusted by the Hastings correction, which accounts for potential asymmetry in the directions of the proposal to achieve the important detailed balance condition that ensures the correct convergence of the MCMC. Algorithm 1 in Appendix A is a detailed description of the block assignment update at each node, using some additional notations: $d_{t,\text{in}} = \sum_k M_{kt}$ is the number of edges into block t, $d_{t,\text{out}} = \sum_k M_{tk}$ the number of edges out of block t, $d_t = d_{t,\text{in}} + d_{t,\text{out}}$ the number of edges into and out of block t, K_{it} the number of edges between nodes i and block t, and β is the update rate that controls the balance between exploration and exploitation. The block assignments are updated for each node iteratively until convergence when the improvement in the log posterior probability falls below a threshold.

B. Block-Merge Moves

The block-merge moves work in almost identical ways as the nodal updates described in Algorithm 1 in Appendix A, except that it takes place at the block level. Specifically, a block-merge move proposes to reassign all the nodes belonging to the current block i to a proposed block s. In other words, it is like applying Algorithm 1 on the block graph where each node represents the entire block (i.e. all the nodes belonging to that block) and each edge represents the number of edges between the two blocks. Another difference is that the block-merges are done in a greedy manner to maximize the log posterior probability, instead of through MCMC. Therefore, the Hastings correction computation step and the proposal acceptance step are not needed. Instead, the best merge move over some number of proposals is computed for each block according to the change in the log posterior probability, and the top merges are carried out to result in the number of blocks targeted by the Fibonacci search.

C. Put It All Together

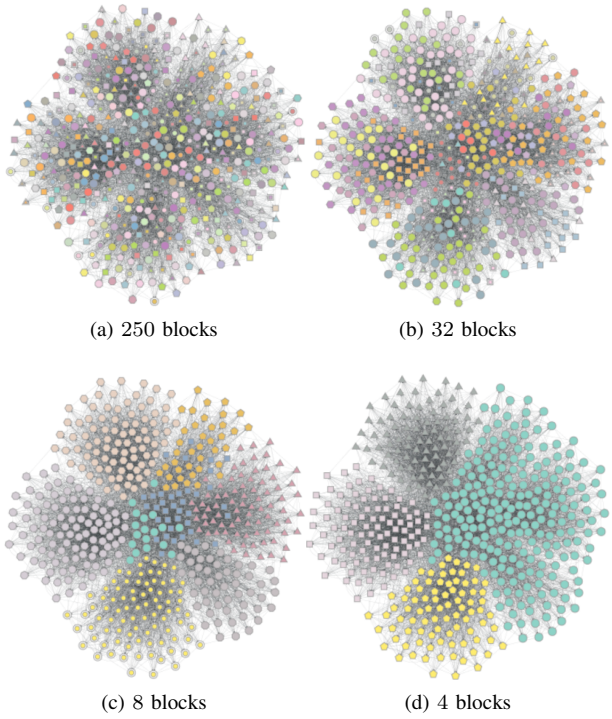
Overall, the algorithms shifts back and forth between the block-merge moves and the MCMC nodal updates, to find the optimal number of blocks B^* with the resulting partition $\boldsymbol{b}^*$. Optimality is defined as having the minimum overall description length, H, of the model and the observed graph given the model:

$$H = E\, h\left(\frac{B^2}{E}\right) + N \log B - \sum_{r,s} M_{rs} \log\left(\frac{M_{rs}}{d_{r,\text{out}} d_{s,\text{in}}}\right) \tag{4}$$

where the function $h(x) = (1+x)\log(1+x) - x\log(x)$. The number of blocks may be reduced at a fixed rated (e.g. 50%) at each block-merge phase until the Fibonacci 3-point bracket is established. At any given stage of the search for optimal number of blocks, the past partition with the closest and higher number of blocks is used to begin the block-merge moves, followed by the MCMC nodal updates, to find the best partition at the targeted number of blocks. Figure 3 shows the partition at selected stages of the algorithm on a 500 node graph:

The algorithm description in this section is for directed graphs. Very minor modifications can be applied for undirected graphs that have no impact on the computational requirement. These minor differences are documented in Peixoto's papers [14], [15], [16].

Advantageously, the baseline partition algorithm with its rigorous statistical foundation, is ideal for processing streaming graphs. Good partitions found on the graph at a previous streaming stage are samples on the posterior distribution of the partition, which can be used as starting partitions for the graph at the current stage. This has the natural Bayesian interpretation of the posterior distribution from a previous state serving as the prior distribution on the current state, as additional data on the graph arrives.



(a) 250 blocks (b) 32 blocks

(c) 8 blocks (d) 4 blocks

Fig. 3. Partitions at selected stages of the algorithm, with the nodes colored and shaped according to their block assignments. The algorithm begins with too many blocks (i.e. over partition) and performs block-merges and nodal updates as it searches for the optimal partition. The Fibonacci search eventually converges to the partition with the optimal number of blocks, which is shown in (c) with 8 blocks.

IV. Parallel Computation Strategies

Significant speed up of the baseline partition algorithm is the primary focus of this graph challenge, and is necessary for computation on large graphs. Since the same core computation, described in Algorithm 1 in Appendix A, is repeated for each block and each node, parallelizing this core computation across the blocks and nodes provides a way to speed up the computation potentially by the order of the number of processors available. This section first discusses the correctness in parallelizing the MCMC updates. It then examines some of the parallel computation schemes for the baseline algorithm, with their respective advantages and requirements.

A. Correctness of Parallel MCMC Updates

The block-merge moves are readily parallelizable, since each of the potential merge move is evaluated based on the previous partition and the best merges are carried out. However, the nodal block assignment updates are not so straight forward, since it relies on MCMC through Gibbs sampling which is by nature a sequential algorithm where each node is updated one at a time. Parallelizing MCMC updates is an area of rising interest, with the increasing demand to perform Bayesian inference on large data sets. Running the baseline partition algorithm on large graphs is a perfect example of this need. Very recently, researchers have proposed to use asynchronous Gibbs sampling as a way to parallelize MCMC updates [25], [26]. In asynchronous Gibbs sampling, the parameters are updated in parallel and asynchronous fashion without any dependency constraint. In [26], a proof is given to show that when the parameters in the MCMC sparsely influence one another (i.e. the Dobrushin's condition), asynchronous Gibbs is able to converge quickly to the correct distribution. It is difficult to show analytically that the MCMC nodal updates here satisfy the Dobrushin's condition. However, since the graph is typically quite sparse, the block assignment on each node influences one another sparsely. This gives intuition on the adequacy of parallel MCMC updates for the baseline partition algorithm. In fact, parallel MCMC updates based on one-iteration-old block assignments have shown to result in equally good partitions compared to the sequential updates, based on the quantitative metrics in Section V-A, for the preliminary tests we conducted so far.

B. Parallel Updates on Nodes and Blocks

An intuitive and straight-forward parallel computation scheme is to evaluate each block-merge and update each nodal block assignment (i.e. Algorithm 1 in Appendix A) in distributed fashion across multiple processors. The block-merge evaluation is readily parallelizable since the computation is based on the previous partition. The MCMC nodal updates can be parallelized using the one-iteration-old block assignments, essentially approximating the true conditional posterior distribution with: $p(b_i|\boldsymbol{b}^-_{-i}, G)$. The conditional block assignments, $\boldsymbol{b}^-_{-i}$, may be more "fresh" if asynchronous Gibbs sampling is used so that some newly updated assignments may become available to be used for updates on later nodes. In any case, once all the nodes have been updated in the current iteration, all the new block assignments are gathered and their modifications on the inter-block edge count matrix aggregated (this can also be done in parallel). These new block assignments and the new inter-block edge count matrix are then available for the next iteration of MCMC updates.

C. Batch Updates Using Matrix Operations

Given an efficient parallelized implementation of large-scale matrix operations, one may consider carrying out Algorithm 1 as much as possible with batch computation using matrix operations [27]. Such matrix operations in practice perform parallel computation across all nodes simultaneously.

Under this computation paradigm, the block assignments are represented as a sparse $N \times B$ binary matrix

$\mathbf{\Gamma}$, where each row $\boldsymbol{\pi}_{i\cdot}$ is an indicator vector with a value of one at the block it is assigned to and zeros everywhere else. This representation results in simple matrix products for the inter-block edge counts:

$$\boldsymbol{M} = \mathbf{\Gamma}^T \boldsymbol{A} \mathbf{\Gamma} \tag{5}$$

The contributions of node i of block assignment r to the inter-block edge count matrix row r and column r are:

$$\Delta \boldsymbol{M}_{\text{row},i\cdot} = \boldsymbol{A}_{i\cdot} \mathbf{\Gamma} \tag{6}$$

$$\Delta \boldsymbol{M}^{+}_{\text{col},i\cdot} = \boldsymbol{A}^T_{\cdot i} \mathbf{\Gamma} \tag{7}$$

These contributions are needed for computing the acceptance probabilities of the nodal block assignment proposals, which makes up a large part of the overall computation requirement.

Algorithm 2 in Appendix B is a batch implementation of the nodal updates described in Algorithm 1. The inter-block edge counts under each of the N proposal are represented using a 3D matrix $\boldsymbol{\mathcal{M}}$ of size $N \times B \times B$. For clarity, computations of the acceptance probabilities involving the inter-block edge counts and degrees are specified using tensor notation. Note that much of these computations may be avoided with clever implementations. For example:

- If the proposed block assignment for a node is the same as its previous assignment, its acceptance probability does not need to be computed.
- New proposals only change two rows and columns of the inter-block edge count matrix, corresponding to moving the counts from the old block to the new block, so most of the entries in $\boldsymbol{\mathcal{M}}$ are simply copies of $\boldsymbol{M}^{-}$.
- The inter-block edge count matrix should be sparse, especially when there is a large number of communities, since most communities do not interact with one another. This gives additional opportunity for speeding up operations on this matrix.
- Similarly, each node is likely to connect with only a few different communities (i.e. blocks). Therefore, changes by each nodal proposal on the inter-block edge count matrix will only involve a few selected rows and columns. Limiting the computation of change in log posterior, ΔS, to these rows and columns may result in significant computation speedup.

V. Metrics

An essential part of this graph challenge is a canonical set of metrics for comprehensive evaluation of the partition algorithm implementation by each participating team. The evaluation should report both the correctness of the partitions produced, as well as the computational requirements, efficiency, and complexity of the implementations. For streaming graphs, evaluation should be done at each stage of the streaming processing, for example, the length of time it took for the algorithm to finish processing the graph after the first two parts of the graph become available, and the correctness of the output partition on the available parts so far. Efficient implementations of the partition algorithm leverage partitions from previous stages of the streaming graph to "jump start" the partition at the current stage.

A. Correctness Metrics

The true partition of the graph is available in this challenge, since the graph is generated with a stochastic block structure, as described in Section II. Therefore, correctness of the output partition by the algorithm implementation can be evaluated against the true partition. On the hybrid graphs where a generated graph is embedded within a real-world graph with no available true partition, correctness is only evaluated on the generated part.

Evaluation of the output partition (i.e. clustering) against the true partition is well established in existing literature and a good overview can be found in [28]. Widely-adopted metrics fall under three general categories: unit counting, pair-wise counting, and information theoretic metrics. The challenge in this paper adopts all of them for comprehensiveness and recommends the pairwise precision-recall as the primary correctness metric for its holistic evaluation and intuitive interpretation. Computation of the correctness metrics described in this section are implemented in Python and shared as a resource for the participants at GraphChallenge.org. Table I provides a simple example to demonstrate each metric, where each cell in row i and column j is the count of nodes belonging to truth block i and reported in output block j.

TABLE I
CONTINGENCY TABLE OF TRUE VS. OUTPUT PARTITION

	Output A	Output B	Output C	Total
Truth A	30	2	0	32
Truth B	1	20	3	24
Total	31	22	3	56

In this example, the nodes are divided into two blocks in the true partition, but divided into three blocks in the output partition. Therefore, this is an example of over-clustering (i.e. too many blocks). The diagonal cells shaded in green here represent the nodes that are correctly partitioned whereas the off-diagonal cells shaded in pink represent the nodes with some kind of partition error.

1) Unit Counting Metrics: The most intuitive metric is perhaps the overall accuracy, specifically the percentage of nodes correctly partitioned. This is simply the fraction of the total count that belong to the diagonal entries of the contingency table after the truth blocks and the output blocks have been optimally associated to maximize the diagonal entries, typically using a linear assignment algorithm [29]. In this example, the overall accuracy is simply $50/66 = 89\%$. While this one single number provides an intuitive overall score, it does not account for the types and distribution of errors. For example, truth block B in Table I has three nodes incorrectly split into output block C. If instead, these three nodes were split one-by-one into output block C,D, and E, a worse case of over-clustering would have taken place. The overly simplified accuracy cannot make this differentiation.

A way to capture more details on the types and distribution of errors is to report block-wise precision-recall. Block-wise precision is the fraction of correctly identified nodes for each output block (e.g. Precision(Output A) $= 30/31$) and the block-wise recall is the fraction of correctly identified nodes for each truth block (e.g. Recall(Truth B) $= 20/24$). The block-wise precision-recall present a intuitive score for each of the truth and output blocks, and can be useful for diagnosing the block-level behavior of the implementation. However, it does not provide a global measure on correctness.

2) Pairwise Counting Metrics: Measuring the level of agreement between the truth and the output partition by considering every pair of nodes has a long history within the clustering community [30], [31]. The basic idea is simple, by considering every pair of nodes which belongs to one of the following four categories: 1.) in the same truth block and the same output block, 2.) in different truth blocks and different output blocks, 3.) in the same truth block but different output blocks, and 4.) in different truth blocks but the same output block. Category 1.) and 2.) are the cases of agreements between the truth and the output partition, whereas categories 3.) and 4.) indicate disagreements. An intuitive overall score on the level of agreement is the fraction of all pairs belonging to category 1.) and 2.), known as the Rand index [30]. [31] proposes the adjusted Rand index with a correction to account for the expected value of the index by random chance, to provide a fairer metric across different data sets. Categories 4.) and 3.) can be interpreted as type I (i.e. false positives) and type II (i.e. false negative) errors, if one considers a "positive" case to be where the pair belongs to the same block. The pairwise precision-recall metrics [32] can be computed as:

$$\text{Pairwise-precision} = \frac{\#\text{Category 1}}{\#\text{Category 1} + \#\text{Category 4}} \tag{8}$$

$$\text{Pairwise-recall} = \frac{\#\text{Category 1}}{\#\text{Category 1} + \#\text{Category 3}} \tag{9}$$

Pairwise-precision considers all the pairs reported as belonging to the same output block and measures the fraction of them being correct, whereas pairwise-recall considers all the pairs belonging to the same truth block and measures the fraction of them reported as belonging to the same output block. In the example of Table I, the pairwise-precision is about 90% and the pairwise-recall about 81%, which indicates this to be a case of over-clustering with more Type II errors. Although pairwise counting is somewhat arbitrary, it does present holistic and intuitive measures on the overall level of agreement between the output and the true partition. For the challenge, the pairwise precision-recall will serve as the primary metrics for evaluating correctness of the output partition.

3) Information Theoretic Metrics: In recent years, holistic and rigorous metrics have been proposed based on information theory, for evaluating partitions and clusterings [28], [33]. Specifically, these metrics are based on the information content of the partitions measured in Shannon entropy. Naturally, information theoretic precision-recall metrics can be computed as:

$$\text{Information-precision} = \frac{I(T;O)}{H(O)} \tag{10}$$

$$\text{Information-recall} = \frac{I(T;O)}{H(T)} \tag{11}$$

where $I(T;O)$ is the mutual information between truth partition T and the output partition O, and $H(O)$ is the entropy (i.e. information content) of the output partition. Using the information theoretic measures, precision is defined as the fraction of the output partition information that is true, and recall is defined as the fraction of the truth partition information captured by the output partition. In the example of Table I, the information theoretic precision is about 57% and recall about 71%. The precision is lower than the recall because of the extra block in the output partition introducing information content that does not correspond to the truth. The information theoretic precision-recall provide a rigorous and comprehensive measure of the correctness of the output partition. However, the information theoretic quantities may not be as intuitive to some and the metrics tend to be harsh, as even a small number of errors often lower the metrics significantly.

B. Computational Metrics

The following metrics should be reported by the challenge participants to characterize the computational requirements of their implementations.

- Total number of edges in the graph (E): This measures the amount of data processed.
- Execution time: The total amount of time taken for the implementation to complete the partition, in seconds.
- Rate: This metric measures the throughput of the implementation, in the number of edges processed over execution time (E/second). Figure 4 shows the preliminary results on this metric between four different implementations of the partition algorithm, when run on a desktop with 16-core 2.4 GHz Intel Xeon processors and 128 GB of 1066 MHz DDR3 SDRAM. The four implementations are: 1.) C++ sequential implementation, 2.) C++ parallel implementation [22], 3.) Python sequential implementation without sparse matrices, and 4.) Python sequential implementation with sparse matrices. Since the algorithm complexity is super-linear, the rate drops as the size of the graph increases, with a slope matching the change in rate according to the analytical complexity of the algorithm, $O(E \log^2 E)$.

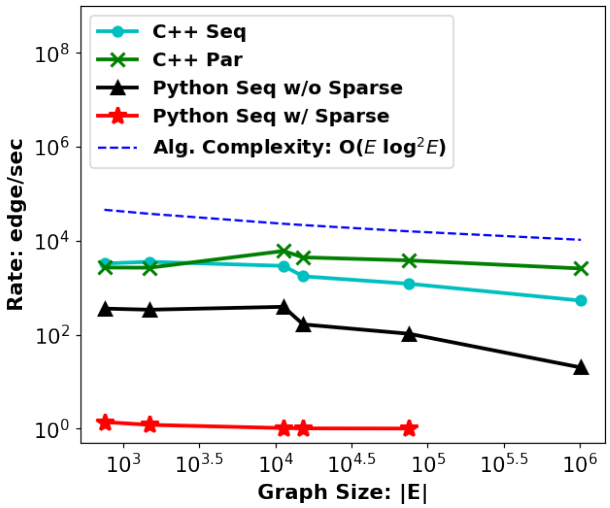


Fig. 4. Processing rate for four different implementations of the baseline algorithm across graphs of increasing size. Overall, the slope of the rates follow the complexity of the algorithm, $O(E \log^2 E)$.

The C++ implementation is about an order of magnitude faster than the Python implementation. With parallel updates, the C++ implementation gains another order of magnitude in rate when the graph is large enough. The Python implementation without sparse matrices suffers in performance on larger graphs due to the inefficiency of the dense matrix representation. The Python implementation with sparse matrices attempts to address this issue, but it runs very slowly due to the lack of a fast implementation of sparse matrices in Python. All four implementations are available at GraphChallenge.org.

- Energy consumption in watts: The total amount of energy consumption for the computation.
- Rate per energy: This metric captures the throughput achieved per unit of energy consumed, measured in E/second/Watt.
- Memory requirement: The amount of memory required to execute the implementation.
- Processor requirement: The number and type of processors used to execute the implementation.

C. Implementation Complexity Metric

- Total lines-of-code count: This measure the complexity of the implementation. SCLC [34] and CLOC [35] are open source line counters that can be used for this metric. The Python demonstration code for this challenge has a total of 569 lines. The C++ open source implementation is a part of a bigger package, so it is difficult to count the lines on just the graph partition.

VI. Summary

This paper gives a detailed description of the graph partition challenge, its statistical foundation in the stochastic blockmodels, and comprehensive metrics to evaluate the correctness, computational requirements, and complexity of the competing algorithm implementations. This paper also recommends strategies for massively parallelizing the computation of the algorithm in order to achieve scalability for large graphs. Theoretical arguments for the correctness of the parallelization are also given. Our hope is that this challenge will provide a helpful resource to advance state-of-the-art performance and foster community collaboration in the important and challenging problem of graph partition on large graphs. Data sets and source code for the algorithm as well as metrics, with detailed documentation are available at GraphChallenge.org.

VII. Acknowledgment

The authors would like thank Trung Tran, Tom Salter, David Bader, Jon Berry, Paul Burkhardt, Justin Brukardt, Chris Clarke, Kris Cook, John Feo, Peter Kogge, Chris Long, Jure Leskovec, Richard Murphy, Steve Pritchard, Michael Wolfe, Michael Wright, and the entire GraphBLAS.org community for their support and helpful suggestions. Also, the authors would like to recognize Ryan Soklaski, John Griffith, and Philip Tran for their help on the baseline algorithm implementation, as well as Benjamin Miller for his feedback on the matrix-based parallelism.

REFERENCES

[1] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 2010.

[2] Patrick Dreher, Chansup Byun, Chris Hill, Vijay Gadepally, Bradley Kuszmaul, and Jeremy Kepner. Pagerank pipeline benchmark: Proposal for a holistic system benchmark for big-data platforms. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 929–937. IEEE, 2016.

[3] Yu Jin and Joseph F Jaja. A high performance implementation of spectral clustering on cpu-gpu platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 825–834. IEEE, 2016.

[4] Hiroki Kanezashi and Toyotaro Suzumura. An incremental local-first community detection method for dynamic graphs. In *2016 IEEE International Conference on Big Data*, pages 3318–3325. IEEE, 2016.

[5] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph partitioning and graph clustering*, volume 588. American Mathematical Soc., 2013.

[6] Siddharth Samsi, Vijay Gadepally, Michael Hurley, Michael Jones, Edward Kao, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Steven Smith, William Song, Diane Staheli, and Jeremy Kepner. Subgraph isomorphism graph challenge. in prep.

[7] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3):75–174, 2010.

[8] Mark EJ Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104, 2006.

[9] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[10] Peter J Mucha, Thomas Richardson, Kevin Macon, Mason A Porter, and Jukka-Pekka Onnela. Community structure in time-dependent, multiscale, and multiplex networks. *science*, 328(5980):876–878, 2010.

[11] Andrea Lancichinetti and Santo Fortunato. Limits of modularity maximization in community detection. *Physical Rev. E*, 84(6):066122, 2011.

[12] Benjamin H Good, Yves-Alexandre de Montjoye, and Aaron Clauset. Performance of modularity maximization in practical contexts. *Physical Rev. E*, 81(4):046106, 2010.

[13] Karrer-Brian Ball, Brian and Mark E.J. Newman. An efficient and principled method for detecting communities in networks. *Physical Review E*, 84:036103, 2011.

[14] Tiago P Peixoto. Efficient monte carlo and greedy heuristic for the inference of stochastic block models. *Physical Rev. E*, 89(1):012804, 2014.

[15] Tiago P Peixoto. Parsimonious module inference in large networks. *Physical Rev. Letters*, 110(14):148701, 2013.

[16] Tiago P Peixoto. Entropy of stochastic blockmodel ensembles. *Physical Rev. E*, 85(5):056122, 2012.

[17] Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical Rev. E*, 83(1):016107, 2011.

[18] Furong Huang, UN Niranjan, M Hakeem, and Animashree Anandkumar. Fast detection of overlapping communities via online tensor methods. *arXiv preprint arXiv:1309.0787*, 2013.

[19] Edoardo M Airoldi, David M Blei, Stephen E Fienberg, and Eric P Xing. Mixed membership stochastic blockmodels. *Journal of Machine Learning Research*, 9(1981-2014):3, 2008.

[20] Steven Thomas Smith, Edward K Kao, Kenneth D Senne, Garrett Bernstein, and Scott Philips. Bayesian discovery of threat networks. *IEEE Transactions on Signal Processing*, 62(20):5324–5338, 2014.

[21] Nesreen K Ahmed, Jennifer Neville, and Ramana Kompella. Network sampling: From static to streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(2):7, 2014.

[22] Tiago P Peixoto. Graph-tool repository. https://git.skewed.de/count0/graph-tool/tree/master, 2014.

[23] Albert-László Barabási. Scale-free networks: a decade and beyond. *science*, 325(5939):412–413, 2009.

[24] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Cambridge Univ Press, 1982.

[25] Alexander Terenin, Daniel Simpson, and David Draper. Asynchronous gibbs sampling. *arXiv preprint arXiv:1509.08999*, 2015.

[26] Christopher De Sa, Kunle Olukotun, and Christopher Ré. Ensuring rapid mixing and low bias for asynchronous gibbs sampling. *arXiv preprint arXiv:1602.07415*, 2016.

[27] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[28] Marina Meilă. Comparing clusteringsan information based distance. *Journal of multivariate analysis*, 98(5):873–895, 2007.

[29] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[30] William M Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[31] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[32] Arindam Banerjee, Chase Krumpelman, Joydeep Ghosh, Sugato Basu, and Raymond J Mooney. Model-based overlapping clustering. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 532–537. ACM, 2005.

[33] Ryan S Holt, Peter A Mastromarino, Edward K Kao, and Michael B Hurley. Information theoretic approach for performance evaluation of multi-class assignment systems. In *SPIE Defense, Security, and Sensing*, pages 76970R–76970R. International Society for Optics and Photonics, 2010.

[34] Brad Appleton. Source code line counter. http://www.bradapp.com/clearperl/sclc.html.

[35] Al Danial. Count lines of code. https://github.com/AlDanial/cloc, 2017.

APPENDIX A: PARTITION ALGORITHM PSEUDOCODE

Algorithm 1: Block Assignment Update At Each Node i

input : b_i^-, $\boldsymbol{b}_{\mathcal{N}_i}^-$: current block labels for node i and its neighbors $\mathcal{N}_i$
$\boldsymbol{M}^-$: current $B \times B$ inter-block edge count matrix
$\boldsymbol{A}_{i\mathcal{N}_i}, \boldsymbol{A}_{\mathcal{N}_i i}$: edges between i and all its neighbors
output: b_i^+: the new block assignment for node i

`// propose a block assignment`
obtain the current block assignment $r = b_i^-$
draw a random edge of i which connects with a neighbor j, obtain its block assignment $u = b_j^-$
draw a uniform random variable $x_1 \sim \text{Uniform}(0, 1)$
if $x_1 \leqslant \frac{B}{d_u^- + B}$ **then**
  `// with some probability, propose randomly for exploration`
  propose $b_i^+ = s$ by drawing s randomly from $\{1, 2, ..., B\}$
else
  `// otherwise, propose by multinomial draw from neighboring blocks to` u
  propose $b_i^+ = s$ from MultinomialDraw$\left(\frac{M_{u\bullet}^- + M_{\bullet u}^-}{d_u^-}\right)$
end
`// accept or reject the proposals`
if $s = r$ **then**
  return $b_i^+ = b_i^-$ `// proposal is the same as the old assignment. done!`
else
  compute $\boldsymbol{M}^+$ under proposal (update only rows and cols r and s, on entries for blocks connected to i)
  compute proposal probabilities for the Hastings correction:
  $p_{r\to s} = \sum_{t\in\{\boldsymbol{b}_{\mathcal{N}_i}^-\}} \left[K_{it} \frac{M_{ts}^- + M_{st}^- + 1}{d_t^- + B}\right]$ and $p_{s\to r} = \sum_{t\in\{\boldsymbol{b}_{\mathcal{N}_i}^-\}} \left[K_{it} \frac{M_{tr}^+ + M_{rt}^+ + 1}{d_t^+ + B}\right]$
  compute change in log posterior (t_1 and t_2 only need to cover rows and cols r and s):
  $$\Delta S = \sum_{t_1,t_2} \left[-M_{t_1t_2}^+ \log\left(\frac{M_{t_1t_2}^+}{d_{t_1,\text{out}}^+ d_{t_2,\text{in}}^+}\right) + M_{t_1t_2}^- \log\left(\frac{M_{t_1t_2}^-}{d_{t_1,\text{out}}^- d_{t_2,\text{in}}^-}\right)\right]$$
  compute probability of acceptance:
  $$p_{\text{accept}} = \min\left[\exp(-\beta\Delta S)\frac{p_{s\to r}}{p_{r\to s}}, 1\right]$$
  draw a uniform random variable $x_3 \sim \text{Uniform}(0, 1)$
  if $x_3 \leqslant p_{accept}$ **then**
    return $b_i^+ = s$ `// accept the proposal`
  else
    return $b_i^+ = r$ `// reject the proposal`
  end
end

APPENDIX B: MATRIX-BASED BATCH UPDATE PSEUDOCODE

Algorithm 2: Batch Assignment Update for All Nodes

input : $\boldsymbol{\Gamma}^-$: current block assignment matrix for all nodes
$\boldsymbol{M}^-$: current $B \times B$ inter-block edge count matrix
$\boldsymbol{A}$: graph adjacency matrix
output: $\boldsymbol{\Gamma}^+$: new block assignments for all nodes

`// propose new block assignments`
compute node degrees: $\boldsymbol{k} = (\boldsymbol{A} + \boldsymbol{A}^T)\mathbf{1}$
compute block degrees: $\boldsymbol{d}^-_{\text{out}} = \boldsymbol{M}^-\mathbf{1}$; $\boldsymbol{d}^-_{\text{in}} = \boldsymbol{M}^{-^T}\mathbf{1}$; $\boldsymbol{d}^- = \boldsymbol{d}^-_{\text{out}} + \boldsymbol{d}^-_{\text{in}}$
compute probability for drawing each neighbor: $\boldsymbol{P}_{\text{Nbr}} = \text{RowDivide}(\boldsymbol{A} + \boldsymbol{A}^T, \boldsymbol{k})$
draw neighbors ($\boldsymbol{N}_{\text{br}}$ is a binary selection matrix): $\boldsymbol{N}_{\text{br}} = \text{MultinomialDraw}(\boldsymbol{P}_{rn})$
compute probability of uniform random proposal: $\boldsymbol{p}_{\text{UnifProp}} = \frac{B}{\boldsymbol{N}_{\text{br}}\boldsymbol{\Gamma}^-\boldsymbol{d}^- + B}$
compute probability of block transition: $\boldsymbol{P}_{\text{BlkTran}} = \text{RowDivide}(\boldsymbol{M}^- + \boldsymbol{M}^{-^T}, \boldsymbol{d}^-)$
compute probability of block transition proposal: $\boldsymbol{P}_{\text{BlkProp}} = \boldsymbol{N}_{br}\boldsymbol{\Gamma}^-\boldsymbol{P}_{\text{BlkTran}}$
propose new assignments uniformly: $\boldsymbol{\Gamma}_{\text{Unif}} = \text{UniformDraw}(B, N)$
propose new assignments from neighborhood: $\boldsymbol{\Gamma}_{\text{Nbr}} = \text{MultinomialDraw}(\boldsymbol{P}_{\text{BlkProp}})$
draw N Uniform$(0, 1)$ random variables $\boldsymbol{x}$
compute which proposal to use for each node: $\boldsymbol{I}_{\text{UnifProp}} = \boldsymbol{x} \leqslant \boldsymbol{p}_{\text{UnifProp}}$
select block assignment proposal for each node:
$\boldsymbol{\Gamma}^P = \text{RowMultiply}(\boldsymbol{\Gamma}_{\text{Unif}}, \boldsymbol{I}_{\text{UnifProp}}) + \text{RowMultiply}(\boldsymbol{\Gamma}_{\text{Nbr}}, (\mathbf{1} - \boldsymbol{I}_{\text{UnifProp}}))$
`// accept or reject the proposals`
compute change in edge counts by row and col: $\Delta\boldsymbol{M}^+_{\text{row}} = \boldsymbol{A}\boldsymbol{\Gamma}^-$; $\Delta\boldsymbol{M}^+_{\text{col}} = \boldsymbol{A}^T\boldsymbol{\Gamma}^-$
update edge count matrix for each proposal: (resulting matrix is $N \times P \times P$):
$\mathcal{M}^+_{ijk} = M^-_{jk} - \Gamma^-_{ij}\Delta M^+_{\text{row},ik} + \Gamma^P_{ij}\Delta M^+_{\text{row},ik} - \Gamma^-_{ik}\Delta M^+_{\text{col},ij} + \Gamma^P_{ik}\Delta M^+_{\text{col},ij}$
update block degrees for each proposal: (resulting matrix is $N \times P$):
$D^+_{\text{out},ij} = d^-_{\text{out,j}} - \Gamma^-_{ij}\sum_k \Delta M^+_{\text{row},ik} + \Gamma^P_{ij}\sum_k \Delta M^+_{\text{row},ik}$
$D^+_{\text{in},ij} = d^-_{\text{in},j} - \Gamma^-_{ij}\sum_k \Delta M^+_{\text{col},ik} + \Gamma^P_{ij}\sum_k \Delta M^+_{\text{col},ik}$
compute the proposal probabilities for Hastings correction ($N \times 1$ vectors):
$\boldsymbol{p}_{r\to s} = \left[(\boldsymbol{p}_{\text{Nbr}}\boldsymbol{\Gamma}^-) \circ (\boldsymbol{\Gamma}^P\boldsymbol{M}^- + \boldsymbol{\Gamma}^P\boldsymbol{M}^{-^T} + 1) \circ \text{RepMat}(\frac{1}{\boldsymbol{d}^- + B}, N)\right]\mathbf{1}$
$\boldsymbol{p}_{s\to r,i} = \left[(\boldsymbol{p}_{\text{Nbr}}\boldsymbol{\Gamma}^-) \circ (\boldsymbol{\Gamma}^-\boldsymbol{\mathcal{M}}^+_{i\bullet\bullet} + \boldsymbol{\Gamma}^-\boldsymbol{\mathcal{M}}^{+^T}_{i\bullet\bullet} + 1) \circ \frac{1}{\boldsymbol{D}^+_{\text{out}} + \boldsymbol{D}^+_{\text{in}} + B}\right]\mathbf{1}$
compute change in log posterior (only need to operate on the impacted rows and columns corresponding to r, s, and the neighboring blocks to i):
$\Delta S_i = \sum_{jk}\left[-\mathcal{M}^+_{ijk}\log\left(\frac{\mathcal{M}^+_{ijk}}{D^+_{\text{out},ij} + D^+_{\text{in},ik}}\right) + M^-_{jk}\log\left(\frac{M^-_{jk}}{d^-_{\text{out},j} + d^-_{\text{in},k}}\right)\right]$
compute probabilities of accepting the proposal ($N \times 1$ vector):
$\boldsymbol{p}_{\text{Accept}} = \min\left[\exp(-\beta\Delta\boldsymbol{S}) \circ \boldsymbol{p}_{s\to r} \circ \frac{1}{\boldsymbol{p}_{r\to s}}, \mathbf{1}\right]$
draw N Uniform$(0, 1)$ random variable $\boldsymbol{x}_{\text{Accept}}$
compute which proposals to accept: $\boldsymbol{I}_{\text{Accept}} = \boldsymbol{x}_{\text{Accept}} \leqslant \boldsymbol{p}_{\text{Accept}}$
return $\boldsymbol{\Gamma}^+ = \text{RowMultiply}(\boldsymbol{\Gamma}^P, \boldsymbol{I}_{\text{Accept}}) + \text{RowMultiply}(\boldsymbol{\Gamma}^-, (\mathbf{1} - \boldsymbol{I}_{\text{Accept}}))$

APPENDIX C: LIST OF NOTATIONS

Below is a list of notations used in this document:

N: Number of nodes in the graph

B: Number of blocks in the partition

$\boldsymbol{A}$: Adjacency matrix of size $N \times N$, where A_{ij} is the edge weight from node i to j

$\boldsymbol{k}$: Node degree vector of N elements, where k_i is the total (i.e. both in and out) degree of node i

$\boldsymbol{K}$: Node degree matrix of $N \times B$ elements, where k_{it} is the total number of edges between node i and block t

$\mathcal{N}_i$: Neighborhood of node i, which is a set containing all the neighbors of i

$^-$: Superscript that denotes any variable from the previous MCMC iteration

$^+$: Superscript that denotes any updated variable in the current MCMC iteration

$\boldsymbol{b}$: Block assignment vector of N elements where b_i is the block assignment for node i

$\boldsymbol{\Gamma}$: Block assignment matrix of $N \times B$ elements where each row $\boldsymbol{\Gamma}_{i\cdot}$ is a binary indicator vector with 1 only at the block node i is assigned to. $\boldsymbol{\Gamma}^P$ is the proposed block assignment matrix.

$\boldsymbol{M}$: Inter-block edge count matrix of size $B \times B$, where M_{ij} is the number of edges from block i to j

$\boldsymbol{\mathcal{M}}^+$: Updated inter-block edge count matrix for each proposal, of size $N \times B \times B$

$\Delta\boldsymbol{M}^+_{\text{row/col}}$: Row and column updates to the inter-block edge count matrix, for each proposal. This matrix is of size $N \times B$.

$\boldsymbol{d}_{\text{in}}$: In-degree vector of B elements, where $d_{\text{in,i}}$ is the number of edges into block i

$\boldsymbol{d}_{\text{out}}$: Out-degree count vector of B elements, where $d_{\text{out,i}}$ is the number of edges out of block i

$\boldsymbol{d}$: Total edge count vector of B elements, where d_i is the total number of edges into and out of block i. $\boldsymbol{d} = \boldsymbol{d}_{\text{in}} + \boldsymbol{d}_{\text{out}}$

$\boldsymbol{D}^+_{\text{in/out}}$: In and out edge count matrix for each block, on each proposal. It is of size $N \times B$

ΔS: The difference in log posterior between the previous block assignment and the new proposed assignment

β: Learning rate of the MCMC

$p_{r \to s}$: Probability of proposing block s on the node to be updated which currently is in block r

p_{Accept}: Probability of accepting the proposed block on the node

$\boldsymbol{P}_{\text{Nbr}}$: Matrix of $N \times N$ elements where each element $\boldsymbol{P}_{Nbr,ij}$ is the probability of selecting node j when updating node i

$\boldsymbol{N}_{\text{br}}$: Matrix of $N \times N$ elements where each row $\boldsymbol{N}_{\text{br},i\cdot}$ is a binary indicator vector with 1 only at j, indicating that j is selected when updating i

$\boldsymbol{p}_{\text{UnifProp}}$: Vector of N elements representing the probability of uniform proposal when updating each node

$\boldsymbol{P}_{\text{BlkTran}}$: Matrix of $B \times B$ elements where each element $\boldsymbol{P}_{\text{BlkTran},ij}$ is the probability of landing in block j when randomly traversing an edge from block i

$\boldsymbol{P}_{\text{BlkProp}}$: Matrix of $N \times B$ elements where each element $\boldsymbol{P}_{\text{BlkProp},ij}$ is the probability of proposing block assignment j for node i

$\boldsymbol{\Gamma}_{\text{Unif}}$: Block assignment matrix from uniform proposal across all blocks. It has $N \times B$ elements where each row $\boldsymbol{\Gamma}_{\text{Unif},i\cdot}$ is a binary indicator vector with 1 only at the block node i is assigned to

$\boldsymbol{\Gamma}_{\text{Nbr}}$: Block assignment matrix from neighborhood proposal. It has $N \times B$ elements where each row $\boldsymbol{\Gamma}_{\text{Unif},i\cdot}$ is a binary indicator vector with 1 only at the block node i is assigned to

$\boldsymbol{I}_{\text{UnifProp}}$: Binary vector of N elements with 1 at each node taking the uniform proposal and 0 at each node taking the neighborhood proposal

$\boldsymbol{I}_{\text{Accept}}$: Binary vector of N elements with 1 at each node where the proposal is accepted and 0 where the proposal is rejected

$\text{Uniform}(x, y)$: Uniform distribution with range from x to y

δ_{tk}: Dirac delta function which equals 1 if $t = k$ and 0 otherwise.

$\text{RowDivide}(\boldsymbol{A}, \boldsymbol{b})$: Matrix operator that divides each row of matrix $\boldsymbol{A}$ by the corresponding element in vector $\boldsymbol{b}$

$\text{RowMultiply}(\boldsymbol{A}, \boldsymbol{b})$: Matrix operator that multiplies each row of matrix $\boldsymbol{A}$ by the corresponding element in vector $\boldsymbol{b}$

$\text{UniformDraw}(B, N)$: Uniformly choose an element from $\{1, 2, ..., B\}$ as the block assignment N times for each node, and return a $N \times B$ matrix where each row i is a binary indicator vector with 1 only at j, indicating node i is assigned block j

$\text{MutinomialDraw}(\boldsymbol{P}_{\text{BlkProp}})$: For each row of the proposal probability matrix $\boldsymbol{P}_{\text{BlkProp},i\cdot}$, draw an block according to the multinomial probability vector $\boldsymbol{P}_{\text{BlkProp,i}\cdot}$ and return a $N \times B$ matrix where each row i is a binary indicator vector with 1 only at j, indicating node i is assigned block j