CluStRE: Streaming Graph Clustering with Multi-Stage Refinement

Adil Chhabra ✉ ⓘ
Heidelberg University, Germany

Shai Dorian Peretz ✉ ⓘ
Heidelberg University, Germany

Christian Schulz ✉ ⓘ
Heidelberg University, Germany

───── **Abstract** ─────

We present CLUSTRE, a novel streaming graph clustering algorithm that balances computational efficiency with high-quality clustering using multi-stage refinement. Unlike traditional in-memory clustering approaches, CLUSTRE processes graphs in a streaming setting, significantly reducing memory overhead while leveraging re-streaming and evolutionary heuristics to improve solution quality. Our method dynamically constructs a quotient graph, enabling modularity-based optimization while efficiently handling large-scale graphs. We introduce multiple configurations of CLUSTRE to provide trade-offs between speed, memory consumption, and clustering quality. Experimental evaluations demonstrate that CLUSTRE improves solution quality by 89.8%, operates 2.6× faster, and uses less than two-thirds of the memory required by the state-of-the-art streaming clustering algorithm on average. Moreover, our strongest mode enhances solution quality by up to 150% on average. With this, CLUSTRE achieves comparable solution quality to in-memory algorithms, i.e. over 96% of the quality of clustering approaches, including LOUVAIN, effectively bridging the gap between streaming and traditional clustering methods.

1 Introduction

Graph clustering or community detection is the problem of identifying densely connected regions of a graph. The potential applications of graph clustering are vast, as nearly all systems with interacting or coexisting entities can be represented as graphs. Common applications include gaining insights into voter behavior, the emergence of trends, terrorist group formation and recruitment [50] or the natural partitioning of data records onto pages [18], as well as analyzing protein interactions [46], gene expression networks [60], fraud detection [1], program optimization [16, 38], and epidemic spread [41].

As ground-truth communities are not known in real-world networks, clustering algorithms often model the quality of graph clustering using *modularity* as an objective function. Modularity measures the density of links inside communities as compared to links between

communities, while accounting for random chance [24, 43]. Intuitively, we optimize for modularity to identify densely connected regions of the network. Modularity is one of the most widely used objective functions for graph clustering and has been shown to be an effective objective function to optimize for [42]. However, modularity optimization is *strongly* NP-complete [12] and thus approximation and heuristic algorithms are used in practice.

State-of-the-art clustering algorithms, such as LOUVAIN [8] and VIECLUS [7], use heuristics to optimize for modularity in their objective function. These as well as other in-memory algorithms operate by loading the entire graph in memory. While these *in-memory* approaches achieve high solution quality by leveraging complete global information to compute clusters, they come with significant memory overhead due to storing the entire edge set in memory. However, there is a growing need to process massive graphs with limited computational resources, particularly in real-time applications like online social network monitoring, fraud detection in financial networks, and dynamic routing in transportation systems [14]. Reducing the memory requirements of graph clustering not only addresses these challenges but also significantly lowers monetary costs, enabling large-scale clustering on small, cost-effective machines.

Streaming algorithms offer a scalable alternative to *in-memory* graph clustering but often sacrifice solution quality. Instead of loading the entire edge set, these algorithms process nodes or edges sequentially, and assign nodes to clusters immediately. This approach significantly reduces memory overhead compared to *in-memory* clustering algorithms but results in lower solution quality due to the lack of global graph knowledge. While streaming algorithms have been extensively studied for the related problem of graph partitioning [13, 20, 54], their potential for graph clustering remains under-explored. Hollocou et al. [27] propose a one-pass streaming clustering algorithm, but research on improving solution quality through techniques like *re-streaming* or leveraging partial global information is limited. State-of-the-art re-streaming [45] and buffered streaming [20, 25] graph partitioners demonstrate the potential of these methods to enhance solution quality in streaming scenarios. Therefore, there is considerable scope to develop a streaming clustering algorithm that reduces memory overhead while limiting the compromise in solution quality.

Contributions

- To address the need for a high-quality graph clustering algorithm with low memory overhead, we propose CLUSTRE, a graph **Clu**stering algorithm in a **St**reaming setting with multi-stage refinement using **R**e-streaming and **E**volutionary heuristics to incorporate partial global information. CLUSTRE processes the graph in a node stream, dynamically constructing a quotient graph to refine clustering and optimize modularity using global information.
- We provide four configurations of CLUSTRE, which offer solution quality and resource consumption trade-offs.
- Through experimental analysis, we demonstrate that our lightest mode achieves 89.8% higher solution quality, runs 2.6× faster, and uses only 58.8% of the memory required by the current state-of-the-art streaming graph clustering algorithm. Our strongest mode improves solution quality by up to 150% on average over the state-of-the-art, setting a new benchmark.
- We show the effect of CLUSTRE's modularity optimization in predicting ground truth communities of real world networks. CLUSTRE improves the NMI score of ground-truth community recovery by 17% on average over the state-of-the-art streaming graph clustering algorithm.

2 Preliminaries

2.1 Basic Concepts

Graphs. Let $G = (V = \{0, \ldots, n-1\}, E)$ be an *undirected graph* with no multiple or self-edges, such that $n = |V|$, $m = |E|$. Let $c : V \rightarrow \mathbb{R}_{\geq 0}$ be a node-weight function, and let $\omega : E \rightarrow \mathbb{R}_{>0}$ be an edge-weight function. We generalize c and ω functions to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. An edge $e = (u, v)$ is said to be *incident* on nodes u and v. Let $N(v) = \{u : (v, u) \in E\}$ denote the neighbors of v. Let $d(v)$ be the degree of node v and $d_w(v)$ be the weighted degree of a node $d_w(v) = \sum_{u \in N(v)} w(v, u)$. A graph $S = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$.

Graph Clustering. The problem addressed in this paper is the *graph clustering problem*. A clustering $\mathcal{C}$ is defined as a partition of the node set V into a set of disjoint *blocks/clusters* $C_1, \ldots, C_k$ such that $C_1 \cup \ldots \cup C_k = V$, as depicted in Figure 1. Importantly, the number of clusters k is typically not predetermined. A clustering is considered *trivial* if it consists of only a single cluster containing all nodes or if all clusters are singletons (i.e., contain only one node). Each cluster C_i can be identified with the subgraph of G induced by its nodes. The set of **intra-cluster edges** is denoted as $E(C) := E \cap \bigcup_i (C_i \times C_i)$. These are edges with both endpoints in the same cluster C_i. The **inter-cluster edges** are given by $E \setminus E(C)$ which is the set of edges whose endpoints are in different clusters. Various objective functions for graph clustering are proposed in the literature, which are discussed in the following paragraph.

Objective Functions. Several measures assess clustering quality, including *coverage* [23], *performance* [23], *inter-cluster conductance* [29], *surprise* [2], *map equation* [48], and *modularity* [44]. We refer the reader to the given references for further insight into these objective functions. Of these, *coverage*, the fraction of intra-cluster edges, is a simple but flawed metric as it favors trivial solutions. *Modularity* mitigates this by comparing observed coverage to its expected value under a random model. In this work, we focus on the modularity objective function as it is a widely accepted clustering quality function [5, 34]. It is formally defined as:

$$Q(C) = \frac{1}{m} \sum_{C_i \in C} \left(K_{C_i \rightarrow C_i} - \frac{\text{vol}(C_i)^2}{2m} \right) \tag{1}$$

where $K_{C_i \rightarrow C_i}$ is the total weight of intra-cluster edges in C_i, and $\text{vol}(C_i) = \sum_{v \in C_i} d_w(v)$ is the weighted volume of C_i. This formulation aligns with our modularity gain computation in Section 3.2, where modularity optimization is performed iteratively using streaming updates.

Streaming Computation Model.   Streaming algorithms typically follow a *load-compute-store* logic. The most widely used streaming model is the one-pass model, which loads nodes along with their neighborhoods (in node streams) or edges (in edge streams) and immediately assigns nodes to clusters. Initially, each node is assumed to be assigned to a singleton cluster. When the node arrives in the data stream, a scoring function computes the change in score when the node is moved from its current cluster to each of the clusters to which its neighbors are assigned. The scoring function indicates how well a node is connected to a cluster by examining cluster assignments of neighbors of the current node. The node is then assigned to the cluster that maximizes the gain in score. One configuration of our algorithm uses this one-pass streaming model to permanently assign nodes, however, our multi-stage refinement approach enables the re-assignment of nodes into clusters after the initial streaming, based on updates made to optimize the clustering. The current state-of-the-art streaming clustering algorithm by Hollocou et al. [27] also reassigns clusters during its edge streaming process.

Memetic Algorithms.   *Evolutionary* or *memetic* algorithms use population-based heuristics to mimic natural evolution to optimize solutions. We utilize the state-of-the-art memetic graph clustering algorithm VieClus [7] within our approach. We chose VieClus because it outperforms the previous benchmark result of the 10th DIMACS graph clustering challenge in 98 out of 115 runs of the algorithm [7]. Memetic algorithms like VieClus identify or generate possible solutions to a problem, and then recombine them to produce new and diverse solutions with higher quality, while also introducing random solutions to explore search space and escape local optima. This process is likened to natural evolution, where the initial solutions are termed *individuals* within a *population*, which are then recombined to form the improved solutions, termed *offspring*, with random solutions generated through mutation. Recombination exploits characteristics of previous solutions to enable improvement in solution quality, and mutation fosters exploration by introducing random variability and preventing premature convergence. In memetic clustering, additional clustering solutions are generated through recombination and mutation to select the clustering that optimizes solution quality.

2.2   Related Work

There has been a significant amount of research on graph clustering; we refer the reader to [21, 26, 59] for thorough reviews of contributions in this field. Here, we survey some graph clustering algorithms relevant to our contribution.

Several successful algorithms have been developed for prominent applications of graph clustering. The Louvain method, introduced by Blondel et al. [8], is a multi-level clustering algorithm that optimizes modularity as its objective function. The Leiden algorithm [53] provides adaptations to the Louvain method. VieClus also optimizes modularity as its objective function, but uses the heuristics of evolutionary algorithms to tackle the graph clustering problem [7]. Spectral clustering [56] partitions graph data into clusters using the eigenvalues and eigenvectors of a similarity matrix and, along with k-means/k-median/k-center methods [28, 57], is particularly effective when the desired number of clusters is predefined. Stochastic Block Models (SBM) [28, 32] are probabilistic models that analyze graphs with underlying (latent) structures, where nodes are grouped into blocks, and edge density between blocks is determined by probability distribution. Markov Clustering (MCL) [55] efficiently clusters graphs by leveraging random walks and Markov chain properties to identify densely connected portions of a graph. DBSCAN [49] and correlation clustering [6] excel at detecting anomalously dense clusters. Additionally, several

distributed graph clustering algorithms have been developed, including frameworks like PREGEL/GIRAPH [37, 49] and MAPREDUCE [15]. TERAHAC [17] is a distributed algorithm for hierarchical clustering that addresses the runtime bottlenecks of other hierarchical clustering algorithms. Some recent approaches to graph clustering draw on deep learning and graph neural networks [35, 36, 52, 58].

While there are several successful approaches to the graph clustering problem, most state-of-the-art clustering algorithms are in-memory algorithms, with limited research undertaken on streaming graph clustering. Hollocou et al. [27] introduce a streaming graph clustering algorithm that reads edge streams and assigns nodes to clusters on-the-fly. When streaming an edge $e = (u, v)$, they either (a) assign u to the cluster of v, (b) assign v to the cluster of u or (c) leave both in their respective cluster, attempting to optimize for modularity. Although this algorithm requires very few computational resources compared to state-of-the-art in-memory algorithms, its solution quality is lower due to the absence of global knowledge of the graph and further impacted by sub-optimal cluster assignment decisions. Of related interest, Assadi et al. [3] present a theoretical proof for the runtime and memory complexity of a streaming algorithm for the distinct hierarchical clustering problem.

3 CluStRE: Streaming Graph Clustering with Memetic Refinement and Re-Streaming Local Search

In this section, we present our algorithm CLUSTRE. First, we provide an overview of CLUSTRE 's light-weight streaming approach to graph clustering. Subsequently, we detail the steps of our approach, namely optimizing for modularity in a streaming setting (Section 3.2) and the optional refinements: on-the-fly construction of a dynamic quotient graph model for memetic graph clustering (Section 3.3), and re-streaming with local search (Section 3.4).

3.1 Overall Algorithm

The CLUSTRE algorithm addresses the quality limitations of one-pass streaming by incorporating global knowledge through memetic clustering of a quotient graph model and re-streaming with local search to optimize modularity. The algorithm processes the input graph G node by node, loading the neighborhood $N(v)$ of only a single node v into memory at a time. For each node v, we compute the cluster C^*, accounting for the clusters assigned to $N(v)$, that yields the highest modularity gain, and assign v to that cluster. Optionally, as each node is assigned, we dynamically construct a quotient graph G_Q, where each node represents a cluster in G, and edges model inter-cluster edges in G, supplemented by self-loops to represent intra-cluster edges. This quotient graph G_Q is designed such that the modularity of any clustering of G_Q is equivalent to the modularity of that clustering of the original graph G (proven in Appendix Theorem 1). To further improve the quality of the clustering, a memetic graph clustering algorithm is run on the computed quotient graph G_Q. This allows us to split and merge clusters and drastically increases the search space to optimize modularity. Once memetic clustering on the quotient graph is finished, the set of clusters with the highest modularity computed on G_Q is used to update the clustering of the original graph. At this stage, the algorithm either outputs the resulting clusters or re-streams the graph to perform additional local search operations, iterating until modularity gain converges to the local optimum. The overall structure of CLUSTRE is outlined in Algorithm 1, with detailed explanations provided in the subsequent sections.

■ **Algorithm 1** CLUSTRE: Overall Approach for Streaming Graph Clustering.

1: **Input:** Graph $G = (V, E)$, Flags: $refineG_Q$, $restreamLS$
2: **Output:** Clustering of nodes $\mathcal{C}$
3: Init array $\mathcal{C}[v] = v \; \forall v \in V$ ▷ singleton clustering $\Theta(n)$ memory
4: Init empty quotient graph G_Q ▷ $\mathcal{O}(|E_{G_Q}|)$ memory
5: **for each** node $v \in V$ (node stream) **do**
6:      $\mathcal{C}[v] = C^* \leftarrow$ COMPUTECLUSTER$(v, \mathcal{C})$ ▷ cluster maximizing modularity gain
7:      **if** $refineG_Q$ **then**
8:          UPDATEQUOTIENTGRAPH$(v, \mathcal{C}, G_Q)$ ▷ on-the-fly G_Q construction
9: **if** $refineG_Q$ **then**
10:      REFINECLUSTERING$(\mathcal{C}, G_Q)$ ▷ memetic graph clustering refinement
11: **if** $restreamLS$ **then**
12:      RESTREAMLOCALSEARCH$(\mathcal{C})$
13: **return** $\mathcal{C}$

3.2 One-Pass Streaming with Modularity Gain Scoring

Our streaming algorithm processes nodes sequentially along with their neighborhoods, assigning them to clusters on-the-fly. At first, we assume that each node is part of its own singleton cluster. As nodes stream in, each node v is assigned to the cluster C^* that maximizes modularity gain, $\Delta Q_{v:C_{cur} \rightarrow C_{can}}$, computed using Equation 3 (also used in the LOUVAIN method [8]).

$$C^* = \underset{C_{can} \in C(N(v))}{\text{argmax}} \Delta Q_{v:C_{cur} \rightarrow C_{can}} \quad (2)$$

The delta modularity function quantifies the change in modularity when node v moves from its current cluster C_{cur} to a candidate cluster C_{can}. For a given node v, we identify a set of candidate clusters $C(N(v))$ as the clusters to which neighbors of v have been assigned. If all candidate clusters decrease modularity, we do not move the node and instead retain the singleton cluster for v. Formally, we compute delta modularity with Equation 3.

$$\Delta Q_{v:C_{cur} \rightarrow C_{can}} = \frac{1}{m}(K_{v \rightarrow C_{can}} - K_{v \rightarrow C_{cur}}) - \frac{d_w(v)}{2m^2}(d_w(v) + \text{vol}(C_{can}) - \text{vol}(C_{cur})) \quad (3)$$

where we define $K_{v \rightarrow C}$ as the total weight of edges between node v and its neighbors in cluster C and $\text{vol}(C)$ to be the weighted volume of a cluster $\sum_{v \in C_i} d_w(v)$. We compute the best cluster for each node in $O(\deg(v))$ time, resulting in an overall streaming pass time complexity of $O(n + m)$. This is achieved by maintaining an array of cluster assignments for each node of size $\Theta(n)$ and an array of cluster volumes of size $|C|$, i.e., total number of clusters. These structures are updated incrementally as nodes are processed. The modularity maximization approach is reminiscent of the first phase of the LOUVAIN method for graph clustering; however, unlike the LOUVAIN method, we visit each node only once and in the node stream order.

3.3 Modularity Refinement via Memetic Clustering

CLUSTRE optionally applies memetic graph clustering after streaming to enhance solution quality by integrating partial global information and evolutionary heuristics. This process maintains a dynamic quotient graph G_Q during streaming, which then serves as the input for memetic clustering. This approach is similar to the LOUVAIN method, which also contracts the
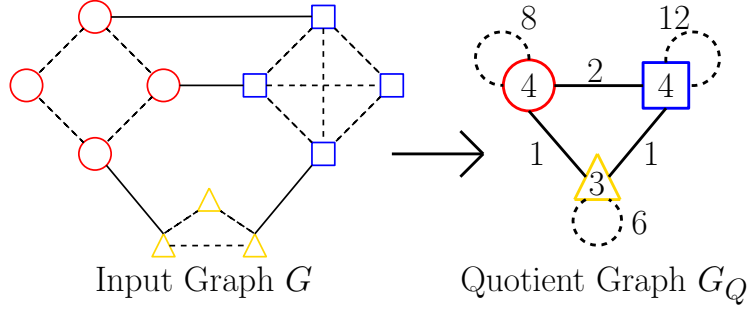
Figure 2 A quotient graph G_Q constructed from an undirected, unweighted toy graph G (all edges have unitary weight). Here, clusters are represented by unique colors and shapes, thick lines show inter-cluster edges and dashed lines show intra-cluster edges. In the quotient graph, each cluster is contracted into a *supernode* with weight equal to number of nodes in that cluster in G. Edges between the nodes of the quotient graph represent inter-cluster edges in G, with weight equal to the sum of the weight of the corresponding inter-cluster edges in G. Intra-cluster edges in G are represented by weighted self-loops in the quotient graph, counted twice - once for each directed intra-cluster edge.

initial clustering into a quotient graph; however, unlike the Louvain method we construct the quotient graph on the fly since we do not store all edges of the graph in-memory but stream them. Moreover, we only contract the clustering once before continuing with the memetic approach. Here, we apply the state-of-the-art memetic graph clustering algorithm used in VieClus [7]. We detail the quotient graph construction and give a brief overview of the techniques used in the memetic clustering approach of VieClus in order to be self-contained. In the quotient graph G_Q, each cluster C_i of the input graph G is represented as a *supernode* v_i', weighted by the number of assigned nodes in C_i. Edges in G_Q are derived from the input graph: an edge between quotient nodes v_i' and v_j' corresponds to inter-cluster edges between C_i and C_j in G, weighted by the sum of inter-cluster edge weights between C_i and C_j; self-loops from v_i' to itself reflect intra-cluster connectivity, weighted by the sum of weights of intra-cluster edges in C_i. We count each undirected intra-cluster edge twice to set $w(v_i', v_i') = K_{C_i \to C_i}$ in the quotient graph. With this construction, the modularity computed on clusterings of the coarser quotient graph is equivalent to the modularity of corresponding clusterings of the input graph, as proven in Appendix Theorem 1. Thus, the quotient graph provides a low-memory representation of the input graph and its clustering, allowing efficient access to partial global information to optimize modularity. Given modularity equivalence, by optimizing modularity of a clustering in G_Q, we obtain an updated clustering of G with higher modularity. To construct G_Q on the fly, we maintain a hash map of quotient graph edges E_{G_Q}, dynamically updated as nodes in G are streamed and assigned to clusters (Algorithm 2). This construction requires $O(E_{G_Q})$ additional memory and takes linear time in the size of E_{G_Q}.

We apply the state-of-the-art memetic graph clustering algorithm used in VieClus [7] on the quotient graph G_Q to iteratively refine clustering through evolutionary heuristics. The process in VieClus begins by generating a diverse population of clustering solutions for G_Q using an adapted Louvain method with label propagation [39]. This initial solution space is used to generate new solutions through iterative rounds of recombination and mutation.

Each round of recombination starts by selecting two sets of clusters with high modularity using a tournament selection rule [40] from the initialized solution space to serve as input sets to generate a new clustering solution. The input or parent sets are combined using overlay

■ **Algorithm 2** updateQuotientGraph($v, Q, \mathcal{C}$): On-the-Fly Quotient Graph Construction.

1: **Input:** Node v (the node to process), Hashmap Q representing quotient graph edges, where $Q[(C_i, C_j)]$ stores edge weights, Array $\mathcal{C}$ of cluster IDs for all nodes
2: **Output:** Updated hashmap Q
3: $C_i \leftarrow \mathcal{C}[v]$ ▷ cluster ID of current node v
4: **for each** neighbor u of v with $v < u$ **do**
5:     $C_j \leftarrow \mathcal{C}[u]$ ▷ cluster ID of neighboring node u
6:     $w \leftarrow w(u, v)$
7:     **if** $C_i == C_j$ **then**
8:         $w \leftarrow 2 \cdot w$ ▷ double the weight for self-loops
9:     **if** $(C_i, C_j) \notin Q$ **or** $(C_j, C_i) \notin Q$ **then**
10:         $Q[(C_i, C_j)] \leftarrow w$ ▷ insert new edge with key (C_i, C_j) and value w
11:     **else**
12:         $Q[(C_i, C_j)] \leftarrow Q[(C_i, C_j)] + w$ ▷ increment edge weight (value)
13: **return** Q

---

clustering: two nodes from G_Q belong to the same cluster in the overlay clustering if and only if they are clustered together in both input sets. Intuitively, overlay clustering enforces a stricter agreement between input sets; if both inputs independently classify two nodes as belonging to the same cluster, then there is high confidence that they should be grouped together. VieClus employs two recombination strategies: flat and multilevel recombination. The former contracts the overlay clusters into a new quotient graph and refines them using Louvain, and the latter uses multi-level local search to refine the solution produced by overlay clustering. Both methods use strategies to ensure the quality of the recombined solution is at least as good as that of its input sets [7].

Mutation is incorporated to counteract the reduction in solution diversity caused by recombination, ensuring a broader search space and avoiding local optima. To achieve this, two input sets are selected from the initialized population, and their clusters are split into two balanced blocks using graph partitioning. The modified solutions, now with additional clusters, serve as inputs for multi-level recombination.

After computing the offspring set, we evaluate it against the existing clustering solutions in the initial population. If it has higher or equal modularity than any existing solution, it replaces the most similar of those solutions, i.e., the set with the smallest symmetric difference between their sets of inter-cluster edges, thus maintaining diversity and preventing premature convergence. Otherwise, the offspring set is discarded. The process then repeats, selecting new parent solutions for recombination or mutation. This iterative process continues until a time cutoff is reached, after which the highest-modularity clustering in the population is applied to update the clustering in G.

3.4 Modularity Refinement via Re-Streaming with Local Search

Our algorithm allows for further modularity optimization via re-streaming with local search. The re-streaming step is applicable after an initial clustering is obtained from the first round of streaming, irrespective of whether we build and optimize the quotient graph. In our re-streaming approach, we first process the node stream again, and identify modularity gain movements for each node by using Equation 3. In this re-stream, the change in modularity computation is more informed since higher-quality clusters have already been identified for

Algorithm 3 RestreamLocalSearch($\mathcal{C}, X \in [0, 1], T_{\text{cutoff}}$).

1: **Init:** $ActiveNodes \leftarrow V$, $\Delta Q \leftarrow \infty$, $startTime \leftarrow$ CURRTIME
2: **while** $ActiveNodes \neq \emptyset$ **and** $\Delta Q \geq X \cdot Q_{\text{total}}$ **and** CURRTIME $- startTime < T_{\text{cutoff}}$ **do**
3:     $NextActiveNodes \leftarrow \emptyset$, $\Delta Q \leftarrow 0$
4:     **for** each $v \in ActiveNodes$ **(node stream) do**
5:         $C^* \leftarrow$ COMPUTECLUSTER$(v, \mathcal{C})$
6:         **if** $C^* \neq \mathcal{C}(v)$ **then**
7:             Update $\mathcal{C}[v] \leftarrow C^*$
8:             Add neighbors of v to $NextActiveNodes$
9:             $\Delta Q \leftarrow \Delta Q + \Delta Q_v$ $\triangleright \Delta Q_v$ = modularity gain from moving node v
10:     $ActiveNodes \leftarrow NextActiveNodes$
11:     $Q_{\text{total}} \leftarrow Q_{\text{total}} + \Delta Q$

---

the nodes in the first pass (instead of assuming they are all in singleton clusters). Here, unlike in the first round of streaming, we don't create new clusters, but merge and split the existing ones for modularity gain.

While the first re-stream requires processing the entire node stream, subsequent re-streams employ an optimization that significantly reduces I/O time. Specifically, after the first re-stream, we track the neighborhoods of nodes that changed clusters during the process – these neighborhoods define the *active nodes* for the next iteration. In subsequent re-streams, instead of reading the entire node stream, we only load the *active nodes* from disk, significantly reducing I/O overhead. The set of *active nodes* is updated dynamically in each re-stream, consisting of the neighborhoods of nodes that were reassigned in the previous iteration. This process continues until (a) convergence, i.e., when no further node movements improve modularity and no active nodes remain, or (b) improvement in the round falls below $X\%$ of the modularity score computed so far, where X is an input parameter, or (c) a specified time cut-off for the local search phase is exceeded. A pseudocode of the re-stream algorithm is shown in Algorithm 3.

To prioritize reducing I/O time, we restrict cluster reassignments to those that alter the first component of the modularity gain function (Equation 3). The key intuition is that only the reassignment of nodes whose neighborhoods changed in the previous re-stream can impact this component. When a node's neighbor is reassigned, it alters the total edge weight between the node and its neighbors in its current cluster, relative to the edge weight between the node and its neighbors in the candidate cluster – this difference is captured in the first component of the modularity gain function. Conversely, if a node's neighborhood remains unchanged, reassigning the node has no effect on this component, as the relative total edge weights remain the same. By focusing re-streaming on nodes with changed neighborhoods, we strike a balance between optimizing I/O efficiency and preserving significant updates to the modularity gain score.

4 Experimental Evaluation

Our experimental analysis investigates the following research questions:

- **RQ1:** What is the impact of memetic clustering and re-streaming on the clustering quality, runtime, and memory consumption of CLUSTRE?
- **RQ2:** How does CLUSTRE's solution quality compare to the state-of-the-art streaming clustering algorithm and in-memory clustering methods? Specifically, how much closer does CLUSTRE get to in-memory clustering than existing streaming algorithms?

- **RQ3:** How does CLUSTRE balance runtime and memory consumption with solution quality, compared to other streaming and in-memory clustering approaches?
- **RQ4:** How well does CLUSTRE recover ground-truth communities compared to existing streaming algorithms?

Datasets.   Our graph instances, listed in Table 1, are sourced from various benchmark datasets [4, 9, 10, 11, 22, 33, 47]. To ensure diversity in size and domain, we include road networks and social networks from the SNAP dataset [33], and web graphs from the 10th DIMACS Graph Clustering Implementation Challenge [4]. Additional graphs are sourced from the Network Repository [47] and the Laboratory for Web Algorithmics [9, 10, 11]. We also include synthetic graphs, such as `RGG` (Random Geometric Graphs) and `RHG` (Random Hyperbolic Graphs), which model real-world scale-free networks [22]. The ground truth communities in the SNAP dataset cannot be used reliably for evaluation in our study as their communities overlap, i.e., one node can be in multiple ground truth clusters; thus we do not use these datasets to evaluate ground truth recovery. Instead, to answer **RQ4**, we incorporate citation and co-purchase networks with ground truth communities from PyTorch Geometric: `CORA`, `CiteSeer`, and `PubMed` [61] are citation networks where nodes represent academic papers and edges denote citations, with communities corresponding to research topics; `AmazonCoPurchase`[51] represents product co-purchase data, where nodes are products and edges indicate frequently bought-together relationships, with communities corresponding to product categories. For our experiments, all graphs were converted to the METIS node-stream format, with parallel edges, self-loops, and directions removed [30]. We assigned unit weights to all nodes and edges for consistency in evaluation.

Baselines.   We compare CLUSTRE against the state-of-the-art streaming graph clustering algorithm by Hollocou et al. [27], referred to as HOLLOCOU. We obtained its C++ implementation from the official GitHub repository but found that it does not support streaming from disk; instead, it first loads the entire edge set into memory before processing edges. To ensure a fair comparison, particularly in terms of memory consumption, we modified their implementation to stream edges directly from disk. HOLLOCOU requires a parameter, v_{max}, which limits the maximum volume of any cluster. The authors provide no guidance on selecting this value or the one used in their experiments. We tested multiple settings, including $v_{max} = n$, and found $v_{max} = 10,000$ to yield the best clustering quality in our setup. Thus, all reported experiments are with $v_{max} = 10,000$. Additionally, we compare CLUSTRE with the in-memory algorithms LOUVAIN, and VIECLUS, both obtained from VIECLUS's official GitHub repository. We allocate VIECLUS five minutes for evolutionary rounds.

Experimental Setup and Reproducibility.   All experiments were performed on a single core of a machine with a sixteen-core Intel Xeon Silver 4216 processor running at 2.1 GHz, 100 GB of main memory, 16 MB of L2-Cache, and 22 MB of L3-Cache running Ubuntu 20.04.1. We implemented CLUSTRE in C++ and compiled it using gcc 13.2.0 with full optimization enabled (-O3 flag). In the study presented here, we configure CLUSTRE by setting a time limit of 15 seconds for the evolutionary rounds in memetic clustering. Additionally, for local search, we set a time limit of 10 minutes and set $X = 0.05$, that is, we stop local search when improvement falls below 5%.

Methodology.   We measure running time, solution quality, i.e., modularity scores, and memory consumption, i.e., the maximum resident set size for the executed process. When averaging over all instances, we use the geometric mean to give every instance the same

a) Running Time   b) Memory Consumption   c) Modularity

Evo — Light — Light+ — Strong
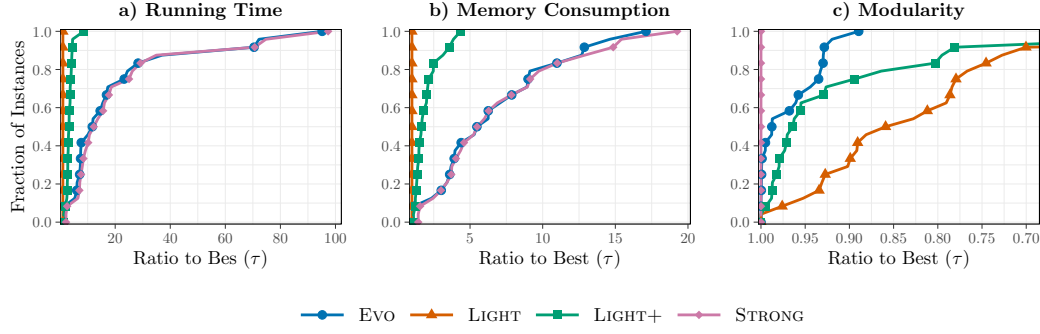
■ **Figure 3** Comparison of CluStRE modes using performance profiles. Each plot shows the fraction of instances (y-axis) for which an algorithm achieves a given ratio (τ) to the best-performing method (x-axis). Note: in plot (c), the x-axis is decreasing since higher modularity is better.

influence on the final score. Let the runtime, modularity score, or memory consumption be denoted by the score σ_A for some clustering generated by an algorithm A. We express this score relative to others using the following tools: *improvement* over an algorithm B, computed as a percentage $(\frac{\sigma_A}{\sigma_B} - 1) * 100\%$ and *relative* value over an algorithm B, computed as $\frac{\sigma_A}{\sigma_B}$. Additionally, we present performance profiles by Dolan and Moré [19] to benchmark our algorithms. These profiles relate the running time (resp. solution quality, memory) of the slower (resp. worse) algorithms to the fastest (resp. best) one on a per-instance basis. Their x-axis shows a factor τ while their y-axis shows the fraction of instances for which an algorithm has up to τ times the running time (resp. solution quality, memory) of the fastest (resp. best) algorithm. For graphs without ground truth communities, we use modularity to measure quality, and on graphs with ground truth communities, we use Normalized Mutual Information (NMI) [31] to measure similarity with predicted clusters. Both metrics are widely used and regarded as reliable metrics for clustering quality [5, 34].

4.1 Impact of Re-Streaming and Memetic Clustering

To answer **RQ1**, we evaluate the effect of our multi-stage refinement by analyzing four configurations of CluStRE, which users of our code can toggle between:

> **CluStRE-Light:** One-pass Streaming.
> **CluStRE-Light+:** Stream + Re-Streaming Local Search (LS).
> **CluStRE-Evo:** Stream + Memetic Quotient Graph Clustering.
> **CluStRE-Strong:** Stream + Memetic Quotient Graph Clustering + Re-Streaming LS.

Experimental comparison of our four modes highlights their distinct applications based on computational constraints and solution quality priorities. Their trade-offs in solution quality and resource usage are illustrated in the performance profiles in Figure 3. CluStRE-Light is the fastest and most lightweight variant across all instances but achieves the lowest solution quality, as shown in Figure 3, making it suitable for efficiency-prioritized applications. We observe that re-streaming significantly improves clustering quality, at the cost of minimal runtime and memory. On average across all instances, CluStRE-Light+ increases solution quality by 15.7% over CluStRE-Light while being 3.3× slower and using 1.9× more memory. Memetic clustering also enhances solution quality, with CluStRE-Evo achieving, on average

across all instances, a 27.3% improvement over CluStRE-Light, though at an increased computational cost, being 12.5× slower and using 5.5× more memory. CluStRE-Strong, which integrates both refinement strategies, delivers the highest solution quality on all instances (Figure 3(c)), offering 31.5% improvement over CluStRE-Light, while taking 14.2× more time and using 5.8× more memory on average across all instances. In absolute terms, CluStRE-Strong still has reasonable memory consumption; on our largest test instance, `uk-2007-05`, which has 105 million nodes and 3.3 billion edges, CluStRE-Strong requires only 2.18GB peak memory. Although our experimental results show that memetic clustering incurs a high running time cost, its duration is tunable and can be adjusted to reduce running time. CluStRE-Strong is thus ideal for applications where quality takes precedence over efficiency. CluStRE-Light+ provides a balanced alternative: as shown in Figure 3, it achieves solution quality within about 10% of CluStRE-Strong in approximately 65% of instances while maintaining a significantly lower computational cost.

> **Observation 1.** CluStRE-Light is the most efficient in runtime and memory but has the lowest quality. CluStRE-Light+ balances performance, providing 15.7% quality improvement over CluStRE-Light while being 3.3× slower and using 1.9× more memory on average. CluStRE-Strong achieves the highest quality, improving by 31.5% over CluStRE-Light on average, at the highest computational cost.

4.2 Solution Quality Compared to Existing Approaches

In Table 1, we present our experimental results for benchmarking CluStRE against Hollocou, louvain and VieClus. Our results confirm that all CluStRE modes outperform Hollocou, demonstrating its superior solution quality: on average across all instances, our lightest mode CluStRE-Light produces 89.8% better solution quality than Hollocou, while CluStRE-Strong achieves a 149.5% improvement over Hollocou, setting a new benchmark for streaming clustering. Notably, on the `com-friendster` network, CluStRE-Strong achieves a modularity score more than 16× better than that of Hollocou. Hollocou underperforms in our study because its clustering decisions are highly sensitive to the order in which edges are streamed – solution quality improves when intra-cluster edges are streamed early, a condition that cannot be guaranteed. In the official implementation, they load the entire edge set into memory, randomize the order of edges to try to get intra-cluster edges to arrive earlier, and then process them sequentially. We modified Hollocou to read/stream edges from the disk in the order in which they appear in the input data set (to ensure consistency across all algorithms) and process them in a streaming setting. Note, however, that in our experiments, the unmodified implementation – which stores all edges in memory – does not improve solution quality. However, it consumes, on average, 7× more memory than our modified streaming version. For example, on `com-friendster`, the official implementation required 27GB of memory while our streaming one used 0.98GB with comparable solution quality. Additionally, Hollocou's reliance on the critical parameter $v_{\max}$ poses a tuning challenge, as there is no principled method for selecting an optimal value for a given graph.

Compared to in-memory algorithms louvain and VieClus, CluStRE-Strong achieves 96.8% of Louvain's solution quality and 96.5% of VieClus' solution quality on average, demonstrating its capability to leverage partial global information through the quotient graph model, and the effectiveness of re-streaming with local search.

■ **Table 1 Modularity Comparison.** We compare modularity scores achieved by our proposed algorithms against competing approaches across various graph instances. The table includes graph type and size (nodes, edges) to contextualize performance variations. Hollocou is a direct competitor streaming algorithm, while Louvain and VieClus are in-memory clustering algorithms. Missing instances ("-") indicate failed runs due to exceeding the maximum amount of memory on the machine. Results demonstrate that all our proposed configurations outperform Hollocou.

Graph	Type	Size: N,M	Light	Light+	Evo	Strong	Hollocou	Louvain	VieClus
circuit5m	Circuit	5.56M, 26.98M	0.1926	0.2920	0.8208	**0.8214**	0.3707	0.8163	0.8211
cit-Patents	Citation	3.77M, 16.52M	0.1839	0.4704	0.8054	**0.8320**	0.1828	0.8330	0.8334
coAuthorsCitseer	Citation	434K, 16.04M	0.7991	0.8208	0.8751	**0.8860**	0.6694	0.9019	0.9060
com-amazon	Social	334K, 925K	0.7637	0.7971	0.9109	**0.9227**	0.6420	0.9315	0.9345
com-dblp	Social	317K, 1.04M	0.7249	0.7600	0.7455	**0.7760**	0.5488	0.8282	0.8387
com-friendster	Social	65.6M, 1.81B	0.5229	0.5839	0.5459	**0.5871**	0.0334	-	-
com-lj	Social	3.99M, 34.68M	0.6364	0.6933	0.6572	**0.7079**	0.2056	0.7639	0.7676
com-youtube	Social	1.13M, 2.98M	0.6426	0.6815	0.6610	**0.6902**	0.2427	0.7233	0.7331
com-orkut	Social	3.07M, 117M	0.5572	0.6248	0.5971	**0.6324**	0.0792	0.6698	0.6737
eu-2005	Web	862K, 16.13M	0.7364	0.8895	0.9204	**0.9316**	0.2683	0.9398	0.9414
fullchip	Circuit	2.98M, 11.81M	0.4020	0.4482	0.5279	**0.5740**	0.3569	0.5967	0.5994
it-2004	Web	41.29M, 1.02B	0.7433	0.9405	0.9646	**0.9693**	0.2387	0.9762	0.9762
italy-osm	Road	6.68M, 7.01M	0.9500	0.9601	**0.9976**	**0.9976**	0.9553	0.9980	0.9980
rgg_n26	Rand. Geo.	67.1M, 574.55M	0.9671	0.9733	0.9897	**0.9909**	0.8044	0.9956	0.9956
rhg2b	Rand. Hyp.	100M, 1.99B	0.9920	0.9921	**0.9921**	**0.9921**	0.6757	-	-
great-britain	Road	7.73M, 8.15M	0.9248	0.9264	0.9972	**0.9973**	0.9344	0.9976	0.9977
roadnet-ca	Road	1.96M, 2.76M	0.7802	0.7967	0.9919	**0.9923**	0.8376	0.9928	0.9935
roadnet-pa	Road	1.08M, 1.54M	0.7712	0.7869	0.9889	**0.9894**	0.8333	0.9901	0.9910
sk-2005	Web	50.63M, 1.81B	0.7241	0.9433	0.9646	**0.9714**	0.1450	-	-
soc-flixster	Social	2.52M, 7.92M	0.4944	0.5145	0.5368	**0.5755**	0.1539	0.6111	0.6191
soc-lj	Social	4.84M, 42.85M	0.6416	0.6937	0.6689	**0.7196**	0.2074	0.7626	0.7660
in-2004	Web	1.38M, 13.59M	0.7118	0.9351	0.9780	**0.9787**	0.4468	0.9803	0.9805
uk-2007-05	Web	105.89M, 3.30B	0.7135	0.8675	0.8216	**0.8789**	0.1838	-	-
webbase-2001	Web	118.14M, 854.80M	0.6341	0.7872	0.7194	**0.8088**	0.5217	0.9822	0.9824
GeoMean	-	-	0.6302	0.7292	0.8019	**0.8285**	0.3321	N/A	N/A

> **Observation 2.** CluStRE surpasses Hollocou in solution quality in all configurations, offering improvements between 89.8%(CluStRE-Light) to 149.5%(CluStRE-Strong) on average. To answer **RQ2**, CluStRE achieves new state-of-the-art modularity optimization in streaming graph clustering while achieving over 96% of the solution quality of in-memory clustering algorithms Louvain and VieClus.

4.3 Resource Consumption Compared to Existing Approaches

Beyond solution quality, we evaluate CluStRE's computational efficiency to answer **RQ3**. In Figure 4 and Figure 5, we present box plots showing the distribution of running time and memory consumption for the various algorithms. Figure 4 benchmarks the streaming algorithms across all graph instances; Figure 5 depicts all baseline algorithms, excluding, for all algorithms, the graph instances for which VieClus and Louvain failed due to their inability to process larger networks without exceeding memory available on the machine. We observe that CluStRE-Light is both faster and uses less memory than Hollocou while delivering higher solution quality (Table 1), as seen in Figure 4, which shows that the
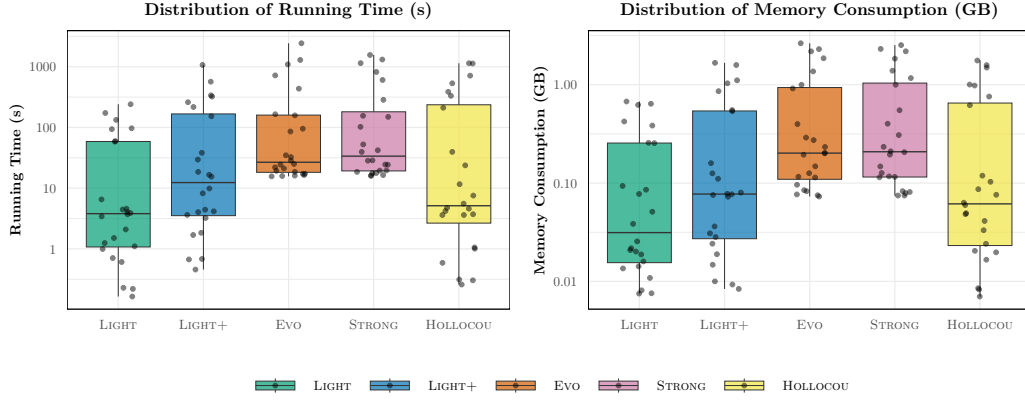
Figure 4 Comparison of CluStRE modes with Hollocou in terms of running time and memory consumption. Box plots depict the distribution of running time (left) and memory consumption (right) across all instances. Note the logarithmic scale on the y-axis. All test instances are included.

median running time and memory consumption across all instances for CluStRE-Light is lower than that of Hollocou. As a result, our algorithm outperforms the state-of-the-art streaming approach across all key metrics of efficiency and quality. On average across all instances, Hollocou is $2.6\times$ slower and consumes $1.7\times$ more memory than CluStRE-Light. CluStRE-Light+ strikes a strong balance, improving solution quality over Hollocou by 119.6% (shown in Table 1), with comparable memory consumption ($1.1\times$ more) and running times ($1.3\times$ slower), on average across all instances.

Compared to Louvain and VieClus, all modes of our algorithm consume less memory, as depicted in Figure 5. CluStRE-Strong requires, on average, only 18.3% of Louvain's memory consumption and only 10.8% of VieClus's, while achieving over 96% of the solution quality of both in-memory algorithms, on average. CluStRE-Light+ further optimizes this trade-off, retaining much of CluStRE-Strong's quality benefits at a significantly reduced computational cost: on average, it is $55.8\times$ faster than VieClus, uses only 3.0% of VieClus's memory, and retains 81.1% of its solution quality; relative to Louvain, it runs $3.06\times$ faster, consumes 5.0% of Louvain's memory, and still achieves 83.2% of its quality, on average.

> **Observation 3.** CluStRE-Light is $2.6\times$ faster and uses two-thirds of Hollocou's memory requirement while offering 89.8% better quality. CluStRE-Light+ maintains memory and runtime efficiency while more than doubling solution quality over Hollocou. CluStRE-Strong utilizes less than a fifth of the memory required by Louvain and VieClus with comparable solution quality.

4.4   Ground-Truth Community Recovery

To further assess the effectiveness of our algorithm in practical settings and answer **RQ4**, we analyzed its performance on graphs with ground-truth communities using Normalized Mutual Information (NMI) [31] as a measure of agreement between the predicted clusters and the ground-truth clusters. Our results confirm that CluStRE-Light outperforms Hollocou in ground-truth recovery, achieving a 7.0% improvement over Hollocou, on average across all instances. CluStRE-Strong further enhances this accuracy, achieving a 16.8% improvement over Hollocou, on average across all instances.
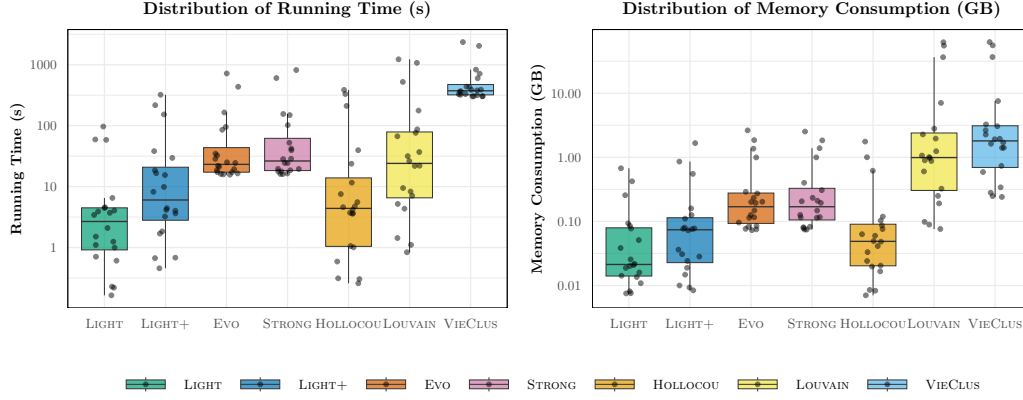
Figure 5 Comparison of CluStRE modes with Hollocou, Louvain and VieClus in terms of running time and memory consumption. Box plots depict the distribution of running time (left) and memory consumption (right). Note the logarithmic scale on the y-axis. Runtime for VieClus is primarily influenced by the tunable preset for evolutionary rounds, which is set to five minutes in our experiments. Instances where Louvain and VieClus failed are excluded for all algorithms.

Table 2 Ground Truth Comparisons. We compare modularity and Normalized Mutual Information (NMI) scores against ground truth community structures across various benchmark graphs. Higher values indicate better alignment with known community structures.

Graph	Light		Light+		Evo		Strong		Hollocou	
	Modularity	NMI	Modularity	NMI	Modularity	NMI	Modularity	NMI	Modularity	NMI
Cora	0.7158	0.3993	0.7434	0.3993	0.7820	0.4274	**0.7991**	**0.4419**	0.5372	0.3867
Citeseer	0.7926	0.3318	0.8108	0.3314	0.8781	0.3382	**0.8867**	**0.3384**	0.6730	0.3309
AmazonCP	0.5627	0.4397	0.6133	0.4679	0.5944	0.4507	**0.6231**	**0.4784**	0.0744	0.3404
PubMed	0.6388	0.1658	0.6775	0.1692	0.7301	0.1871	**0.7552**	**0.1917**	0.3283	0.1691
GEOMEAN	0.6720	0.3135	0.7074	0.3199	0.7388	0.3323	**0.7599**	**0.3422**	0.3065	0.2930

5 Conclusion

To address the need for a high-quality graph clustering algorithm with low memory overhead, we propose CluStRE, a graph **Clu**stering algorithm in a **St**reaming setting with multi-stage refinement using **R**e-streaming and **E**volutionary heuristics. CluStRE processes the graph in a node stream, dynamically constructing a quotient graph to efficiently refine clustering and optimize modularity using global information. Through memetic optimization and re-streaming with local search, the algorithm iteratively adjusts the clustering until modularity converges to the local optimum. Our results establish the superiority of CluStRE over the state-of-the-art streaming clustering approach, Hollocou, in solution quality, ground-truth community recovery, runtime and memory efficiency. CluStRE-Strong sets a new benchmark, delivering a 150% improvement in solution quality over Hollocou on average across all instances, making it the best-performing streaming clustering method. Even our lightest configuration, CluStRE-Light significantly outperforms Hollocou, improving solution quality by 90% while being faster and using less memory. Additionally, our algorithm bridges the solution quality gap between streaming and in-memory clustering algorithms, achieving comparable solution quality to in-memory algorithms, while requiring less than a

fifth of their memory consumption, on average. Notably, CluStRE improves ground-truth recovery over Hollocou by up to 17% on average, making it the best-performing streaming algorithm for accurately recovering known community structures. These outcomes highlight the considerable potential of our algorithm, as well as the versatility of our configurations, positioning it as a dynamic and promising tool for high-quality clustering even in resource-constrained settings.

───── **References** ─────

1 L. Akoglu, H. Tong, and D. Koutra. Graph Based Anomaly Detection and Description: A Survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, May 2015. `doi:10.1007/s10618-014-0365-y`.

2 V. Arnau, S. Mars, and I. Marín. Iterative Cluster Analysis of Protein Interaction Data. *Bioinformatics*, 21(3):364–378, September 2004. `doi:10.1093/bioinformatics/bti021`.

3 Sepehr Assadi, Vaggos Chatziafratis, Jakub Lacki, Vahab Mirrokni, and Chen Wang. Hierarchical Clustering in Graph Streams: Single-pass Algorithms and Space Lower Bounds. In Po-Ling Loh and Maxim Raginsky, editors, *Conference on Learning Theory, 2-5 July 2022, London, UK*, volume 178 of *Proceedings of Machine Learning Research*, pages 4643–4702, 2022. URL: `https://proceedings.mlr.press/v178/assadi22a.html`.

4 D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. 2014. `doi:10.1007/978-1-4614-6170-8_23`.

5 David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013. `doi:10.1090/conm/588`.

6 N. Bansal, A. Blum, and S. Chawla. Correlation Clustering. *Machine Learning*, 56(1):89–113, July 2004. `doi:10.1023/B:MACH.0000033116.57574.95`.

7 S. Biedermann, M. Henzinger, C. Schulz, and B. Schuster. Memetic Graph Clustering. In *17th Intl. Symposium on Experimental Algorithms*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:15, 2018. `doi:10.4230/LIPIcs.SEA.2018.3`.

8 Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast Unfolding of Community Hierarchies in Large Networks. *CoRR*, abs/0803.0476, 2008. `arXiv:0803.0476`.

9 P. Boldi, A. Marino, M. Santini, and S. Vigna. BUbiNG: Massive Crawling for the Masses. *ACM Trans. Web*, 12(2), 2018. `doi:10.1145/3160017`.

10 Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A Multiresolution Coordinate-free Ordering For Compressing Social Networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 587–596, 2011. `doi:10.1145/1963405.1963488`.

11 Paolo Boldi and Sebastiano Vigna. The Webgraph Framework I: Compression Techniques. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 595–602, 2004. `doi:10.1145/988672.988752`.

12 U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikoloski, and D. Wagner. On Modularity Clustering. *IEEE Trans. Knowl. Data Eng.*, 20(2):172–188, 2008. `doi:10.1109/TKDE.2007.190689`.

13 A. Chhabra, M. F. Faraj, C. Schulz, and D. Seemaier. Buffered Streaming Edge Partitioning. In *22nd Intl. Symposium on Experimental Algorithms*, volume 301 of *LIPIcs*, pages 5:1–5:21, 2024. `doi:10.4230/LIPIcs.SEA.2024.5`.

14 Adil Chhabra, Florian Kurpicz, Christian Schulz, Dominik Schweisgut, and Daniel Seemaier. Partitioning Trillion Edge Graphs on Edge Devices. *CoRR*, abs/2410.07732, 2024. `doi:10.48550/arXiv.2410.07732`.

15 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008. `doi:10.1145/1327452.1327492`.

16 J. Demme and S. Sethumadhavan. Approximate Graph Clustering for Program Characterization. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012. `doi:10.1145/2086696.2086700`.

17 L. Dhulipala, J. Łącki, J. Lee, and V. Mirrokni. TeraHAC: Hierarchical Agglomerative Clustering of Trillion-edge Graphs. *Proc. ACM Manag. Data*, 1(3), November 2023. `doi:10.1145/3617341`.

18 Ajit A. Diwan, Sanjeeva Rane, S. Seshadri, and S. Sudarshan. Clustering Techniques for Minimizing External Path Length. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 342–353, 1996. URL: `http://www.vldb.org/conf/1996/P342.PDF`.

19 E. D. Dolan and J. J. Moré. Benchmarking Optimization Software With Performance Profiles. *Math. Program.*, 91(2):201–213, 2002. `doi:10.1007/s101070100263`.

20 Marcelo Fonseca Faraj and Christian Schulz. Buffered Streaming Graph Partitioning. *ACM J. Exp. Algorithmics*, 27:1.10:1–1.10:26, 2022. `doi:10.1145/3546911`.

21 Santo Fortunato. Community Detection in Graphs. *CoRR*, abs/0906.0612, 2009. `arXiv:0906.0612`.

22 D. Funke, S. Lamm, U. Meyer, M. Penschuck, P. Sanders, C. Schulz, D. Strash, and M. von Looz. Communication-free Massively Distributed Graph Generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. `doi:10.1016/j.jpdc.2019.03.011`.

23 Marco Gaertler. Clustering. In Ulrik Brandes and Thomas Erlebach, editors, *Network Analysis: Methodological Foundations [outcome of a Dagstuhl seminar, 13-16 April 2004]*, volume 3418 of *Lecture Notes in Computer Science*, pages 178–215, 2004. `doi:10.1007/978-3-540-31955-9_8`.

24 M. Girvan and M. E. J. Newman. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences*, 99(12):7821–7826, June 2002.

25 Milad Rezaei Hajidehi, Sraavan Sridhar, and Margo I. Seltzer. CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics. *Proc. VLDB Endow.*, 18(1):14–27, 2024. URL: `https://www.vldb.org/pvldb/vol18/p14-hajidehi.pdf`.

26 Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. Distributed Graph Clustering Using Modularity and Map Equation. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, volume 11014 of *Lecture Notes in Computer Science*, pages 688–702, 2018. `doi:10.1007/978-3-319-96983-1_49`.

27 A. Hollocou, J. Maudet, T. Bonald, and M. Lelarge. A Streaming Algorithm for Graph Clustering. *CoRR*, abs/1712.04337, 2017. `doi:10.48550/arXiv.1712.04337`.

28 R. Jiang, S. Han, Y. Yu, and W. Ding. An Access Control Model for Medical Big Data Based on Clustering and Risk. *Information Sciences*, 621:691–707, 2023. `doi:10.1016/j.ins.2022.11.102`.

29 Ravi Kannan, Santosh S. Vempala, and Adrian Vetta. On Clusterings: Good, Bad and Spectral. *J. ACM*, 51(3):497–515, 2004. `doi:10.1145/990308.990313`.

30 G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. `doi:10.1137/S1064827595287997`.

31   A. Lancichinetti, S. Fortunato, and J. Kertész. Detecting the Overlapping and Hierarchical Community Structure in Complex Networks. *New J. of Physics*, 11(3):033015, March 2009.

32   C. Lee and D. J. Wilkinson. A Review of Stochastic Block Models and Extensions for Graph Clustering. *Applied Network Science*, 4(1):122, December 2019. `doi:10.1007/s41109-019-0232-2`.

33   Jure Leskovec and Rok Sosic. SNAP: A General-purpose Network Analysis and Graph-mining Library. *ACM Trans. Intell. Syst. Technol.*, 8(1):1:1–1:20, 2016. `doi:10.1145/2898361`.

34   J. Li, S. Lai, Z. Shuai, Y. Tan, Y. Jia, M. Yu, Z. Song, X. Peng, Z. Xu, Y. Ni, H. Qiu, J. Yang, Y. Liu, and Y. Lu. A Comprehensive Review of Community Detection in Graphs. *Neurocomputing*, 600:128169, 2024. `doi:10.1016/j.neucom.2024.128169`.

35   Fanzhen Liu, Shan Xue, Jia Wu, Chuan Zhou, Wenbin Hu, Cécile Paris, Surya Nepal, Jian Yang, and Philip S. Yu. Deep Learning for Community Detection: Progress, Challenges and Opportunities. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4981–4987, 2020. `doi:10.24963/IJCAI.2020/693`.

36   Yue Liu, Jun Xia, Sihang Zhou, Siwei Wang, Xifeng Guo, Xihong Yang, Ke Liang, Wenxuan Tu, Stan Z. Li, and Xinwang Liu. A Survey of Deep Graph Clustering: Taxonomy, Challenge, and Application. *CoRR*, abs/2211.12875, 2022. `doi:10.48550/arXiv.2211.12875`.

37   Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146, 2010. `doi:10.1145/1807167.1807184`.

38   Scott McFarling. Program Optimization for Instruction Caches. In Joel S. Emer and John L. Hennessy, editors, *ASPLOS-III Proceedings - Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, USA, April 3-6, 1989*, pages 183–191, 1989. `doi:10.1145/70082.68200`.

39   Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning Complex Networks via Size-constrained Clustering. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 351–363, 2014. `doi:10.1007/978-3-319-07959-2_30`.

40   B. L. Miller and D. E. Goldberg. Genetic Algorithms, Selection Schemes, and the Varying Effects of Noise. *Evolutionary Computation*, 4(2):113–131, June 1996. `doi:10.1162/evco.1996.4.2.113`.

41   M. E. J. Newman. Properties of Highly Clustered Networks. *Physical Review E*, 68(2), August 2003.

42   M. E. J. Newman. Fast Algorithm for Detecting Community Structure in Networks. *Physical Review E*, 69(6), June 2004.

43   M. E. J. Newman. Modularity and Community Structure in Networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, June 2006.

44   M. E. J. Newman and M. Girvan. Finding and Evaluating Community Structure in Networks. *Physical Review E*, 69(2), February 2004.

45   Joel Nishimura and Johan Ugander. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 1106–1114, 2013. `doi:10.1145/2487575.2487696`.

46   J. B. Pereira-Leal, A. J. Enright, and C. A. Ouzounis. Detection of Functional Modules From Protein Interaction Networks. *Proteins: Structure, Function, and Bioinformatics*, 54(1):49–57, 2004.

47 Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4292–4293, 2015. `doi:10.1609/AAAI.V29I1.9277`.

48 M. Rosvall, D. Axelsson, and C. T. Bergstrom. The Map Equation. *The European Physical J. Special Topics*, 178(1):13–23, November 2009.

49 Sherif Sakr, Faisal Moeen Orakzai, Ibrahim Abdelaziz, and Zuhair Khayyat. *Large-Scale Graph Processing Using Apache Giraph*. Springer, 2016. `doi:10.1007/978-3-319-47431-1`.

50 S. E. Schaeffer. Graph Clustering. *Computer Science Review*, 1(1):27–64, 2007. `doi:10.1016/j.cosrev.2007.05.001`.

51 Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of Graph Neural Network Evaluation. *CoRR*, abs/1811.05868, 2018. `arXiv:1811.05868`.

52 X. Su, S. Xue, F. Liu, J. Wu, J. Yang, C. Zhou, W. Hu, C. Paris, S. Nepal, D. Jin, Q. Z. Sheng, and P. S. Yu. A Comprehensive Survey on Community Detection With Deep Learning. *IEEE Transactions on Neural Networks and Learning Systems*, 35(4):4682–4702, 2024. `doi:10.1109/TNNLS.2021.3137396`.

53 Vincent A. Traag, Ludo Waltman, and Nees Jan van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *CoRR*, abs/1810.08473, 2018. `arXiv:1810.08473`.

54 Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler, editors, *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342, 2014. `doi:10.1145/2556195.2556213`.

55 S. Van Dongen. Graph Clustering Via a Discrete Uncoupling Process. *SIAM J. on Matrix Analysis and Applications*, 30(1):121–141, 2008. `doi:10.1137/040608635`.

56 U. von Luxburg. A Tutorial on Spectral Clustering. *Statistics and Computing*, 17(4):395–416, December 2007. `doi:10.1007/s11222-007-9033-z`.

57 Zhiqiang Wan, Yazhou Zhang, and Haibo He. Variational Autoencoder Based Synthetic Data Generation for Imbalanced Learning. In *2017 IEEE Symposium Series on Computational Intelligence, SSCI 2017, Honolulu, HI, USA, November 27 - Dec. 1, 2017*, pages 1–7, 2017. `doi:10.1109/SSCI.2017.8285168`.

58 S. Wang, J. Yang, J. Yao, Y. Bai, and W. Zhu. An Overview of Advanced Deep Graph Node Clustering. *IEEE Transactions on Computational Social Systems*, 11(1):1302–1314, 2024. `doi:10.1109/TCSS.2023.3242145`.

59 Timothé Watteau, Aubin Bonnefoy, Simon Illouz-Laurent, Joaquim Jusseau, and Serge Iovleff. Advanced Graph Clustering Methods: A Comprehensive and In-depth Analysis. *CoRR*, abs/2407.09055, 2024. `doi:10.48550/arXiv.2407.09055`.

60 Y. Xu, V. Olman, and D. Xu. Clustering Gene Expression Data Using a Graph-theoretic Approach: An Application of Minimum Spanning Trees. *Bioinformatics*, 18(4):536–545, April 2002. `doi:10.1093/bioinformatics/18.4.536`.

61 Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting Semi-supervised Learning with Graph Embeddings. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 40–48, 2016. URL: `http://proceedings.mlr.press/v48/yanga16.html`.

A   Appendix: Modularity Equivalence of Quotient Graph

▶ **Theorem 1.** *Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to \mathbb{R}_{>0}$, and let $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ be a partition of V into clusters. The modularity of $\mathcal{C}$ in G is:*

$$Q_G(\mathcal{C}) = \frac{1}{m} \sum_{C_i \in \mathcal{C}} \left(K_{C_i \to C_i} - \frac{vol(C_i)^2}{2m} \right), \tag{4}$$

where:
- $m = \sum_{(u,v) \in E} w(u, v)$ *is the total edge weight,*
- $K_{C_i \to C_i} = \sum_{u,v \in C_i, (u,v) \in E} w(u, v)$ *is the total intra-cluster edge weight,*
- $vol(C_i) = \sum_{v \in C_i} d_w(v)$ *is the volume of C_i, where $d_w(v) = \sum_{u \in N(v)} w(v, u)$ is the weighted degree.*

Consider the quotient graph $G_Q = (V_Q, E_Q)$, where each cluster C_i in G is contracted into a supernode v'_i, with edge weights:

$$w'(v'_i, v'_j) = K_{C_i \to C_j} = \sum_{u \in C_i, v \in C_j, (u,v) \in E} w(u, v), \tag{5}$$

and self-loop weights $w'(v'_i, v'_i) = K_{C_i \to C_i}$. Then:
1. *Define the clustering $\mathcal{C}' = \{\{v'_i\} \mid C_i \in \mathcal{C}\}$ in G_Q. The modularity of the clustering $\mathcal{C}'$ of the quotient graph G_Q satisfies $Q_{G_Q}(\mathcal{C}') = Q_G(\mathcal{C})$.*
2. *Given any clustering $\mathcal{C}'$ in G_Q, its modularity is preserved when expanded to G, i.e., if $\hat{\mathcal{C}}$ is the corresponding clustering in G, then $Q_G(\hat{\mathcal{C}}) = Q_{G_Q}(\mathcal{C}')$.*

Proof. The total edge weight in G_Q remains equivalent by construction:

$$m' = \sum_{(v'_i, v'_j) \in E_Q} w'(v'_i, v'_j) = \sum_{(u,v) \in E} w(u, v) = m. \tag{6}$$

Since we defined $\mathcal{C}' = \{\{v'_i\} \mid C_i \in \mathcal{C}\}$ in G_Q, we get that the modularity of $\mathcal{C}'$ in G_Q is:

$$Q_{G_Q}(\mathcal{C}') = \frac{1}{m} \sum_{v'_i \in V_Q} \left(w'(v'_i, v'_i) - \frac{vol(v'_i)^2}{2m} \right). \tag{7}$$

where, by construction, $w'(v'_i, v'_i) = K_{C_i \to C_i}$ and $vol(v'_i) = vol(C_i)$ due to the weighted self-loops. Thus, we get:

$$Q_{G_Q}(\mathcal{C}') = \frac{1}{m} \sum_{C_i \in \mathcal{C}} \left(K_{C_i \to C_i} - \frac{vol(C_i)^2}{2m} \right) = Q_G(\mathcal{C}). \tag{8}$$

∴ Modularity is invariant under contraction.

For the reverse direction, consider a clustering $\mathcal{C}'$ in G_Q. The expansion of $\mathcal{C}'$ into G defines a clustering $\hat{\mathcal{C}}$, where each supernode $v'_i \in C'_j$ is replaced by its original cluster C_i, forming $\hat{\mathcal{C}}_j = \bigcup_{v'_i \in C'_j} C_i$. The modularity of $\hat{\mathcal{C}}$ in G is:

$$Q_G(\hat{\mathcal{C}}) = \frac{1}{m} \sum_{\hat{C}_j \in \hat{\mathcal{C}}} \left(K_{\hat{C}_j \to \hat{C}_j} - \frac{vol(\hat{C}_j)^2}{2m} \right). \tag{9}$$

By construction, $K_{\hat{C}_j \to \hat{C}_j} = K_{C'_j \to C'_j}$ and $vol(\hat{C}_j) = vol(C'_j) \implies Q_G(\hat{\mathcal{C}}) = Q_{G_Q}(\mathcal{C}')$.

∴ Modularity is invariant under expansion.   ◀