CluStRE: Streaming Graph Clustering with Multi-Stage Refinement

Shai Dorian Peretz

March 14, 2025

4222895

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Adil Chhabra

Acknowledgments

First and foremost, I want to express my profound gratitude to Prof. Dr. Christian Schulz for granting me the incredible opportunity to work on this project under his guidance. I will always be grateful for the countless opportunities he has presented me with throughout these years starting with the beginner's research internship, and now with the ability to continue this fascinating project as a student research assistant at the Algorithm Engineering Group. I would also like to extend my sincere gratitude to Adil Chhabra, whose guidance was invaluable. Knowing that I could always reach out to him at any time with my questions gave me an incredible sense of confidence and direction. His willingness to offer insights and help whenever needed made a significant difference, and I cannot thank him enough for that.
Beyond academia, I owe the deepest thanks to my family, whose unwavering love and sacrifices have shaped me into the person I am today. Lastly, I sincerely appreciate all my friends, who have stood by me through the various phases of my life, making this journey all the more memorable.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, March 14, 2025

Shai Dorian Peretz

Abstract

Clustering a graph into disjoint communities is a crucial technique in data analysis for evaluating interactions and similarities between entities within a dataset. In this work, we propose a novel streaming graph clustering algorithm, CLUSTRE, that balances computational efficiency with high-quality clustering using a multi-stage refinement scheme. CLUSTRE processes the graph in a node-streaming setting, significantly reducing overall memory consumption while leveraging re-streaming and evolutionary heuristics to improve solution quality. During streaming, CLUSTRE dynamically constructs a quotient graph, capturing key structural properties of the original graph. This method allows for efficient modularity-based optimizations for large graphs. CLUSTRE offers multiple configurations, providing trade-offs between runtime, memory consumption, and clustering quality, further highlighting its versatility. Our approach produces state-of-the-art results, improving solution quality by more than 92%, while operating 1.36× faster and requiring only 72.76% of the memory consumption compared to existing state-of-the-art streaming methods. Furthermore, our strongest mode, CLUSTRE enhances solution quality by more than 140%. Moreover, CLUSTRE achieves solution quality that is highly comparable to in-memory clustering algorithms, reaching over 97% of the solution quality produced by state-of-the-art in-memory algorithms such as LOUVAIN, effectively bridging the gap between streaming and in-memory clustering algorithms.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Graph clustering, also known as community detection, refers to the problem of identifying densely connected regions of a graph to reveal structures and relationships in data. Graph clustering has a wide range of applications, as systems and datasets with interacting or coexisting entities can often be represented as graphs. Common applications for the graph clustering problem include improving recommendation systems by clustering similar users or items together [48], detecting functional modules in Protein-Protein Interaction (PPI) networks and grouping proteins with similar functions [45], image analysis and segmentation [36], anomaly detection in cybersecurity [26], and analyzing the formation and recruitment patterns of terrorist groups [54].
Since, in practical cases, the ground-truth communities of entities are not known, clustering algorithms often assess clustering quality using various objective functions, with modularity being among the most widely used [39]. Modularity quantifies clustering quality by considering the density of the connections within communities and the connections between different communities, compared to what would be expected in a random network while preserving expected node degrees [21, 40]. Therefore, to identify densely connected regions of a graph, we optimize the clustering using modularity. However, it has been shown that modularity optimization is strongly NP-complete [10]; thus, heuristic algorithms are used in practice.
There exist many algorithms that tackle the graph clustering problem using modularity optimization, such as LOUVAIN [5] and VIECLUS [4]. These and other in-memory clustering algorithms operate by storing the entire graph in memory. On the one hand, they achieve high-quality results by leveraging complete global information, enabling high-quality decision making when assigning entities to clusters. On the other hand, they come with a major drawback: high memory consumption, as the entire graph must be stored in memory. This drawback is especially significant given the sizes of modern graphs used to represent vari-

ous networks and systems. Most modern graphs are massive, exceeding millions of nodes and billions of edges, making in-memory storage and clustering computations infeasible due to their sheer size, often requiring hundreds of gigabytes of memory. Hence, there has been a growing interest in graph clustering algorithms that scale well and require significantly less memory. Reducing the memory consumption of clustering algorithms not only addresses the challenge of computational feasibility but also enables clustering of large graphs on small, cost-effective machines.
To tackle this problem, there has been a growing interest in streaming algorithms across various fields and applications. Streaming algorithms provide a scalable alternative to in-memory graph clustering algorithms. However, they typically sacrifice solution quality. The idea behind streaming algorithms is that only a fraction of the graph is loaded into memory at a time. Once a set of nodes and edges is loaded, the algorithm assigns the nodes to specific clusters. Subsequently, the algorithm processes the next set of nodes and edges while discarding the previously assigned set. This approach significantly reduces the memory overhead compared to in-memory algorithms but often results in lower clustering quality due to the lack of global graph knowledge when making decisions. While streaming algorithms have been extensively studied for related problems such as the graph partitioning problem [11, 16], the same level of attention has not been given to the graph clustering problem. One of the few streaming graph clustering algorithms is proposed by Hollocou et al. [25]. Hollocou et al. [25] iterate over the set of edges once, loading only a single edge and its endpoints into memory before making a decision. However, research on improving clustering quality through various techniques such as re-streaming or multi-stage refinement to leverage partial global information is limited. These techniques have been shown to deliver high-quality solutions, as seen in state-of-the-art re-streaming [43] and buffered streaming [16] approaches for the similar partitioning problem, demonstrating their potential. Thus, various techniques can be utilized to develop a streaming graph clustering algorithm that reduces memory consumption while minimizing the compromise in solution quality.

1.2 Our Contribution

In this thesis, we propose a new streaming algorithm for the graph clustering problem. Our algorithm achieves state-of-the-art solution quality, comparable to in-memory algorithms, while drastically reducing memory consumption, requiring only a fraction of the memory consumed by in-memory algorithms. More specifically:

1. We propose CLUSTRE, a streaming graph **Clu**stering algorithm with multi-stage refinement using **R**e-streaming and **E**volutionary heuristics to leverage partial global information. CLUSTRE is a node-processing streaming algorithm that dynamically constructs a quotient graph, enabling refinement and modularity optimization on a global scale.

2. We provide four configurations for CLUSTRE. Each configuration incorporates different refinement algorithms, offering a trade-off between runtime, memory consumption, and solution quality.

3. Through experimental analysis, we demonstrate that the lightest mode, which consumes the least memory among the four configurations, yields 92.50% higher solution quality, runs 1.36× faster, and requires only 72.76% of the memory compared to the current state-of-the-art streaming graph clustering algorithm. Furthermore, using our strongest mode, we improve solution quality by an average of 140% over the current state-of-the-art streaming algorithm, thus setting a new benchmark.

4. We also show that CLUSTRE extracts ground-truth communities from real-world networks more efficiently than the current state-of-the-art streaming algorithm. Our lightest mode improves the Normalized Mutual Information (NMI) score by approximately 20.04%, whereas our strongest mode achieves an improvement of about 39.65% over the current state-of-the-art.

1.3 Structure

The remainder of this thesis is organized as follows: Chapter 2 introduces basic concepts and notation used throughout this thesis, including the formal definition of the graph clustering problem and concepts related to the multi-level scheme, evolutionary algorithms, and streaming approaches. Chapter 3 surveys related work, beginning with an overview of in-memory graph clustering algorithms and different approaches used to balance runtime, solution quality, and memory consumption. Additionally, we examine existing streaming algorithms for the graph clustering problem. Chapter 4 presents our newly developed algorithm, CLUSTRE, a streaming graph clustering algorithm that utilizes multi-stage refinement. We begin by outlining the overall structure of the algorithm, followed by a detailed discussion of our one-pass streaming algorithm for modularity optimization. Subsequently, we explain the first refinement phase, consisting of an evolutionary in-memory algorithm applied to the dynamically constructed quotient graph. Finally, we present our last refinement phase, which incorporates re-streaming and local search to further optimize clustering quality. In Chapter 5, we present the empirical results of CLUSTRE. We describe our experimental methodology and the dataset used, followed by a tuning experiment to evaluate the effect of various parameters on different configurations of our algorithm. We then compare CLUSTRE to other state-of-the-art clustering algorithms. Chapter 6 concludes by summarizing our work and highlighting possibilities for future work.

CHAPTER 2

Fundamentals

2.1 Basic Concepts

A *graph* $G = (V, E)$ is a tuple defined by a set of vertices V, also known as nodes, and a set of edges $E \subseteq V \times V$. A *weighted* graph is further defined by a vertex weight function $c : V \to \mathbb{R}_{\geq 0}$ and an edge weight function $\omega : E \to \mathbb{R}_{\geq 0}$. The number of vertices of a graph G is denoted by $n = |V|$, while $m = |E|$ represents the total number of edges. For an *undirected* graph, the presence of an edge $(u, v) \in E$ implies the existence of $(v, u) \in E$. The graph G is said to be *simple* if it does not contain multiple or self-loops. An edge $e = (u, v)$ is said to be *incident* on vertices u and v. The cost functions c and ω can also be extended to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. The *neighborhood* of a vertex u is defined as $N(u) = \{v \mid (u, v) \in E\}$, and its *degree* $d(u) = |N(u)|$ is the total number of neighbors. The *weighted* degree of a vertex is the sum of the weights of all its incident edges. The input graphs in this thesis are *unweighted* and *simple*, i.e., $\forall u \in V : c(u) = 1$ and $\forall e \in E : \omega(e) = 1$, with no multiple or self-loops. A graph $S = (V', E')$ is a *subgraph* of $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$.

2.2 Graph Clustering

Given an undirected graph $G = (V, E)$, the graph clustering problem seeks to cluster the set of vertices V into natural blocks, also known as *clusters*, $C_1, C_2, \ldots, C_k$ with

1. $C_1 \cup C_2 \cup \cdots \cup C_k = V$
2. $C_i \cap C_j = \emptyset \; \forall i \neq j$

such that the *intra-cluster edge density* is significantly higher than the *inter-cluster edge density*, i.e. *inter-cluster edge sparsity*. $C = (C_1, C_2, \ldots, C_k)$ is called a *clustering* of G.

A clustering is *trivial* if $k = 1$ or if each cluster C_i contains only a single element, also known as a *singleton*. The set of *intra-cluster edges* is defined as $E(C) := E \cap (\cup_i C_i \times C_i)$, while $E \backslash E(C)$ is the set of *inter-cluster edges*. $m(C) := |E(C)|$ represents the *intra-cluster edge density* and similarly, $\overline{m}(C) := |E \backslash E(C)|$ represents *inter-cluster edge density*. It is important to note that the number of clusters, k, is not provided in advance.

2.2.1 Objective Functions

To achieve the desired intra-cluster edge density and inter-cluster edge sparsity in a clustering, several objective functions have been defined. One such naturally arising objective function is *coverage* [9]. The coverage $cov(C)$ of a graph clustering C is defined as the ratio of all intra-cluster edges to the total number of edges in the graph.

$$cov(C) = \frac{m(C)}{m} \tag{2.1}$$

Ideally, the higher the coverage value, the better the quality of the clustering C. At first glance, this definition seems to capture the essence of a good clustering C quite well. However, when used as an objective function to maximize, one quickly realizes that coverage always reaches its maximum value of 1 when the clustering of a graph is the trivial clustering where all vertices are assigned to the same cluster. Such a clustering, for obvious reasons, is not useful.
To address this issue, the *modularity* equation [41] has been introduced. Modularity evaluates the quality of a clustering by comparing the coverage of the resulting clustering with the expected coverage of the same clustering applied to a graph with identical degree distribution but randomized edges. Thus, the modularity objective function is defined as:

$$mod(C) = cov(C) - \mathbb{E}[cov(C)] \tag{2.2}$$

The expected coverage $\mathbb{E}[cov(C)]$ is defined as:

$$\mathbb{E}[cov(C)] = \frac{1}{4m^2} \sum_{c \in C} (\sum_{v \in c} deg(v))^2 \tag{2.3}$$

Maximizing modularity in graph clustering has been shown to be NP-complete, without an efficient approximation algorithm [10]. Although modularity addresses the issue of the trivial clustering, by assigning it a quality score of 0, and captures the concept of intra-cluster edge density and inter-cluster edge sparsity well, it is not without limitations. One main issue that modularity faces is *resolution limit* [18]. A resolution limit is a constraint in the objective function that hinders the accurate identification of smaller communities within large graphs. The resolution limit in modularity arises, because the

expected coverage $\mathbb{E}[cov(C)]$ depends on the total number of edges in the graph. This causes smaller communities to contribute less to the modularity score in large graphs. As a result, modularity may overlook small communities and merge them into larger ones, leading to scale-dependent clustering results.
Several modifications to modularity [23] and many other objective functions such as *performance* [63], *inter-cluster conductance* [27], *surprise* [1], *CPM* [61], and *map equation* [49] have been proposed, each with its own set of advantages and disadvantages. During this thesis, we focus on the modularity objective function in equation 2.2, as it is a widely accepted quality function and has been the main objective function in many implementation challenges such as the 10th DIMACS challenge [3].

2.3 Multilevel Scheme

As previously mentioned, graph clustering via modularity maximization is NP-complete. Furthermore, most objective functions for the graph clustering problem have been shown to be NP-hard to optimize and resilient for efficient approximations [2, 10]; hence, heuristic algorithms are used in practice. One popular heuristic scheme, which has gained a lot of traction for the graph clustering problem is the *multilevel scheme*. This approach can be divided into three phases: the *coarsening* or *contraction* phase, the *initial partitioning* phase, and the *refinement* or *uncoarsening* phase. In the coarsening phase, the graph G is repeatedly reduced (contracted) to obtain smaller, i.e. coarser, graphs until no more reductions can be applied or a certain threshold size is reached, at which point the graph is referred to as the *coarsest graph*. Notably, the applied reductions preserve the structural properties of the original graph G. As the name suggests, during the initial partitioning phase, a clustering algorithm is applied to the coarsest graph to obtain an initial clustering. Since the coarsest graph is relatively small, even computationally expensive algorithms can efficiently compute a clustering. In the uncoarsening phase, the contractions of the coarsest graph are recursively reversed until the original graph is reached. At each uncoarsening level, a *local search* algorithm refines the clustering obtained at the coarsest level by optimizing an objective function.

2.4 Evolutionary Algorithms

Evolutionary or *memetic* algorithms are heuristic-based algorithms that mimic natural evolution, also known as survival of the fittest, to optimize solutions to a given problem. These algorithms have seen much success for various optimization problems [4, 20]. Evolutionary algorithms consist of five key phases, *initialization*, *selection*, *recombination*, *mutation*, and *eviction*. Each of these phases mimics an important step in natural evolution. First, in the initialization phase, the algorithm creates an initial population P, a set of candidate solutions to the given problem. Subsequently, the selection phase begins by selecting a

set of parents, typically two individuals. There exist many methods to select the parents. A widely used method is the *tournament selection* [38]. Tournament selection randomly selects a subset s from the population P to compete with each other based on their fitness values. The fittest individual from s is then chosen to be a parent. Once the desired number of parents is selected, the recombination phase begins where, similar to nature, structural attributes and features are taken from all the selected parents to create an offspring. During the creation of the offspring, there is a probability p_m [15] that a mutation occurs. In the mutation phase, the offspring experiences a mutation, as in nature. The mutation phase introduces more diversity into the population P, which is essential to obtain high-quality results and escape from local optima. Finally, after creating a new offspring with a potential mutation, the algorithm enters the eviction phase, where an individual is removed from the population P. There are many eviction methods, but the most popular one is to evict the least fit individual in terms of solution quality. This completes one round of the evolutionary algorithm. Evolutionary algorithms operate when runtime constraints are not a major concern, mimicking the natural evolution of the initial population to ultimately obtain a high-quality solution.

2.5 Streaming Models

Streaming algorithms generally follow an iterative load-compute-store approach. Most streaming algorithms adopt the one-pass model, where the vertices of the graph are processed sequentially. In this model, each vertex and its neighbors are loaded into memory one at a time. Then, a decision process permanently assigns the streamed vertex to a cluster, based on the assignments of previously processed vertices. Other variations of the one-pass streaming model exist, where instead of streaming vertices, individual edges and their endpoints are simultaneously loaded into memory [25] one at a time. In this setting, an endpoint is assigned to the cluster of the other endpoint, based on previously assigned vertices. In general, many more streaming models exist, such as the buffered streaming model [16], where instead of a single vertex, a set of vertices is loaded into memory at once. However, in this thesis, we adopt the one-pass vertex streaming model to further minimize memory consumption.

CHAPTER 3

Related Work

There has been a significant amount of research over the years on the graph clustering problem. We refer the reader to [17, 24, 34, 67] for a more thorough examination of the contributions to this field, which may include techniques and approaches that are not part of this thesis. These approaches include distributed graph clustering frameworks such as PREGEL [35] and TERAHAC [13], which utilize hierarchical agglomerative clustering techniques. Furthermore, there has been a growing interest in recent years to explore the graph clustering problem using deep learning and graph neural networks. We refer the reader to [33, 58, 66] for a comprehensive analysis.
In this chapter, we first explore well-known in-memory graph clustering algorithms such as SPECTRAL CLUSTERING [65], and the MARKOV CLUSTER PROCESS (MCL) [64]. Additionally, we examine more computationally efficient in-memory clustering algorithms, such as LABEL PROPAGATION, first introduced by Raghavan et al. [46], and local search [5]. We then explore some multilevel schemes, as outlined in Section 2.3, such as the LOUVAIN [5] method and its refinement, the LEIDEN [60] algorithm. To conclude this section, we analyze the state-of-the-art in-memory graph clustering algorithm, VIECLUS by Biedermann et al. [4], which employs an evolutionary approach, as explained in Section 2.4. Since the work in this thesis utilizes the VIECLUS algorithm, we use Section 3.1.4 to provide a detailed overview of VIECLUS.
Subsequently, we focus on existing streaming graph clustering algorithm, which is the main focus of this thesis. Streaming algorithms have the main advantage of consuming significantly less memory in comparison to in-memory algorithms due to the streaming model, explained in Section 2.5, which ensures, that only a fraction of the graph is stored in-memory at a time. The typical downside of streaming algorithms is that they yield lower-quality solutions than in-memory graph clustering algorithms, due to their lack of global graph knowledge.

3.1 In-Memory Graph Clustering

3.1.1 Global Clustering Algorithms

When tackling the in-memory graph clustering problem, two approaches can be taken. The first is to construct an algorithm, which considers the entire graph when clustering, meaning all nodes and edges are analyzed before any decisions are made. This method is commonly referred to as a *global clustering algorithm*. This approach typically yields high-quality solutions, as each decision is made with complete knowledge of the global graph structure. One prominent global clustering algorithm, where linear algebra and graph theory meet, is the SPECTRAL CLUSTERING ALGORITHM [65], specifically the technique covered by Ng et al. [42], which constitutes the construction of a symmetric normalized Laplacian matrix L_{sym}. Once built, the algorithm computes the eigenvectors and their corresponding eigenvalues of L_{sym}, which identify the total number of connected components and the overall connectivity of the graph. The computed eigenvectors then form a new $n \times k$ matrix U. With the help of the k-means algorithm [12], each node is then assigned to the cluster corresponding to its row in the matrix U.
Another popular global in-memory clustering algorithm is the MARKOV CLUSTER PROCESS (MCL), first introduced by Van Dongen [64]. The basic idea behind the MCL algorithm revolves around an exploration phase and a reward phase of a random walk moving around the graph. The MCL algorithm alternates between the two phases, allowing the random walk to first explore, also known as *expansion*, and then reward, also known as *inflation*, paths it frequently traverses. This process gradually isolates and identifies clusters by making intra-cluster paths more attractive than inter-cluster paths.
While global clustering algorithms usually yield high-quality clustering, their application is highly restrictive when dealing with large graphs. This is because in-memory algorithms often require high computational and memory cost, such as computing the eigenvectors and their corresponding eigenvalues, which has a complexity of $\mathcal{O}(n^3)$. Furthermore, global algorithms scale poorly for large graphs, to the extent that even storing the adjacency matrix can be a challenge if the graph has millions or billions of nodes.

3.1.2 Local Clustering Algorithms

The other approach to tackle the graph clustering problem is to use local graph clustering algorithms. In comparison to global graph clustering algorithms, local algorithms do not consider the entire graph when making a decision but instead require information only for a small subset of the graph. Therefore, local graph clustering algorithms require significantly less time for each decision, making them faster in practice compared to global clustering algorithms.
LABEL PROPAGATION (LPA), first introduced by Raghavan et al. [46], is a quintessential example of a linear local clustering algorithm. At first, each vertex is assigned to a singleton cluster; subsequently, the algorithm proceeds in rounds, where in each round it traverses

the nodes uniformly at random and assigns the currently traversed vertex to the cluster that is most prevalent in its neighborhood. Ties are broken uniformly at random. The algorithm terminates either upon convergence or when a cut-off variable, such as a time limit is met. Oscillations are avoided by updating nodes asynchronously, using the most recent clustering labels from the neighbors, instead of waiting for an entire iteration to complete. An interesting attribute of the LPA algorithm is that it can find a clustering solely based on node connections to various clusters, independently of any objective function.
Building on the idea of the LPA algorithm is the common local search algorithm [5]. Local search shares many similarities with the LPA, namely, it is also a linear-time algorithm, with the same random node traversal approach as the LPA algorithm, where each node initially starts in its own singleton cluster. The key difference between the LPA and local search is the criteria by which the nodes are assigned to their neighboring clusters. In the LPA, only the connectivity to the neighbors is used to assess the best movement for the currently visited vertex. In the local search algorithm, this evaluation is based on optimizing a specific objective function, with nodes moved to the neighboring cluster that yields the highest gain to the objective function. The objective function can be chosen freely, such as those introduced in Section 2.2.1, namely the modularity score [41] or the CPM objective function [61]. Different objective functions directly affect the clustering outcome; while modularity strongly favors more populated clusters, the CPM objective function can identify clusters of smaller size [61].
Although local algorithms and methods are highly computationally efficient, they typically yield relatively poor results [44]. The poor results stem from the lack of global graph information when making decisions, as they only consider the neighbors of a specific node. To partially address the poor performance of local algorithms while still maintaining computational efficiency, multi-level algorithms have been developed that build upon local graph clustering algorithms.

3.1.3 Multi-Level Clustering Algorithms

One of the most popular and widely used multi-level algorithm for the clustering problem is the LOUVAIN method by Blondel et al. [5]. The algorithm can be divided into two different phases. The main components consist of a local search and a contraction phase, which alternate until the local search algorithm converges. First, a local search algorithm is applied to the graph to find a clustering. The local search used is the same as described in Section 3.1.2, with the modularity objective function. Once the local search phase is complete, the graph is contracted. The contracted graph is induced by the clustering found in the local search phase, where each node in the contracted graph represents a cluster in the original graph.
An edge exists between two nodes u and v in the contracted graph if and only if the respective clusters are connected by at least one edge in G. The weight of an edge between two nodes u and v in the contracted graph is set to the sum of the weights of the edges that span between the corresponding clusters of nodes u and v in G. Self-loops are added

to each node in the contracted graph, where the weight of the self-loop is defined as the total weight of the intra-cluster edges in the corresponding cluster. Note that the objective function of the contracted graph is the same as the original graph G due to the added and modified edge weights. Thus, the same cluster features and structures found in the clustering of the original graph G are also found in the contracted graph. After the graph has been contracted, the local search algorithm is applied again but solely to the contracted graph. Now, moving a single node in the contracted graph indirectly moves multiple nodes of the original graph at once. This allows for significant changes in the cluster structure, resulting in potentially high jumps in quality. The local search algorithm is always executed on the coarsest graph found at each stage. Once the local search phase converges and no more contractions can be made, uncoarsening starts, where the clustering of the coarsest graph is projected back to the original graph. It has been shown that this method excels in terms of running time. For most graphs only a few iterations of local search and contraction are necessary to obtain high-quality clustering, as demonstrated by Lancichinetti et al. [29]. This is because local search is applied continuously only to the coarsest graphs, where moving a node results in moving a set of nodes of the original graph, which helps the algorithm converge faster. Furthermore, by using the local search phase, where only a single node is considered to be moved at a time, the modularity resolution limit, as explained in Section 2.2.1 is partially mitigated [5].

One challenge that the LOUVAIN algorithm faces is that the resulting clustering may be internally disconnected [60], which for some use cases is undesirable. The LEIDEN algorithm introduced by Traag et al. [60] builds on the LOUVAIN method but ensures that all nodes in a cluster are connected, by incorporating a new phase called the *refinement* phase. The refinement phase is executed between the local search and contraction phases, which work identically to those in the LOUVAIN method. In the refinement phase, the algorithm inspects each cluster to check whether every node in the cluster is part of a single connected component. If a cluster is not internally connected, then the algorithm splits it into connected components, such that each connected component corresponds to a new cluster. This guarantees that no cluster is disconnected.

While multi-level algorithms tend to deliver relatively good results in a short period of time, they often converge to local optima. This is because of the local algorithms playing a major role in the multi-level algorithms, where the optimal decision in a local setting might not be the optimal decision towards reaching the global optimum. Many different strategies exist to combat this phenomenon. One approach is to introduce controlled randomness [22], where suboptimal cluster assignments are allowed by some probability p. Furthermore, one could also execute multiple runs with different node orderings [29]. Applying these techniques to the LOUVAIN or LEIDEN algorithm can help escape local optima. When piecing these strategies together, it becomes evident that the mentioned techniques strongly resemble the phases of an evolutionary algorithm.

3.1.4 VieClus

To escape local optima, Biedermann et al. [4] developed VIECLUS. The VIECLUS algorithm is an evolutionary algorithm that optimizes modularity and utilizes methods introduced in Section 3.1.2 and Section 3.1.3. The key aspects making VIECLUS the state-of-the-art in-memory graph clustering algorithm lie in the recombination and mutation operations. The algorithm, as detailed in Section 2.4, can be divided into five phases: initialization, selection, recombination, mutation, and eviction. The VIECLUS algorithm runs through these phases until a given time limit is reached, at which point the fittest individual is the output.
VIECLUS starts by generating p clusterings, referred to in the context of evolutionary algorithms as individuals, using a modified LOUVAIN method [5]. The order in which the nodes are traversed during the LOUVAIN method is randomized in each round, ensuring some diversity in the population. To increase diversification, VIECLUS modifies the LOUVAIN method, such that for the first $\lambda \in [0, 4]$ rounds, the LOUVAIN method uses the SIZE CONSTRAINT LABEL PROPAGATION (SCLP) algorithm [37] to compute the clustering instead of the local search algorithm. The SCLP algorithm works similarly to the LPA algorithm but ensures that the size of each cluster does not exceed a certain limit. The upper bound of a cluster size is set to $U \in [\frac{n}{10}, n]$ at the beginning of the algorithm. After the initial λ rounds, the LOUVAIN method switches back to the local search algorithm to compute the clustering. Note that for $\lambda = 0$ the initialization of all clusterings is the normal LOUVAIN method. The population size is set to $p \leq 100$ to ensure sufficient competition among individuals. As observed by Eiben et al. [15], this is essential for maintaining a consistently high-quality population while preserving computational efficiency.
Once the population is initialized, the selection phase begins, where a set of individuals is selected for reproduction. VIECLUS employs the tournament-based selection approach [38], which, as explained in Section 2.4, randomly selects s individuals from the population, and from the s individuals, selects the fittest with respect to the modularity score. The tournament size is set to $s = 2$ to ensure a lower *selection pressure* [38], reducing the risk of premature convergence to a local optimum caused by repeatedly selecting the fittest individuals for reproduction.
For the recombination phase, VIECLUS defines two different approaches: the flat- and multi-level recombination operations. Both approaches rely on the notion of the *maximum overlap*, introduced by Ovelgönne et al. [44], which is applied to the two parent clusterings. The principle behind the maximum overlap method is to identify common node cluster assignments between the parent clusterings. Two nodes from the graph belong to the same cluster in the overlay clustering if and only if they are clustered together in both parents; otherwise, they belong to different clusters. The overlay clustering method enforces an agreement process, with the idea being that if both parents agree that two specific nodes should belong to the same cluster, then there is a high confidence that they should be grouped together.
The flat recombination approach includes four different techniques, which are selected at

random. The PLAIN recombination method is the most basic, where the first step is to compute the maximum overlap of both parents selected in the selection phase. The overlay graph is then contracted. The contraction scheme applied to the overlay graph mirrors that of the LOUVAIN method, ensuring that the modularity score on the contracted graph remains the same as that of the original graph. The LOUVAIN method is then applied starting from singleton clusters on the contracted overlay graph to create the offspring. Note that due to the construction of the contracted graph, nodes that are clustered together in both parents belong to the same cluster in the offspring. Solely nodes that are split by a parent's clustering are affected by the LOUVAIN method. The algorithm therefore focuses on the regions of the graph where the parents disagree. Similarly, APPLIED INPUT CLUSTERING follows the same steps as the PLAIN recombination technique. The only difference is that the overlay graph is not initialized with singleton clusters but with the clustering of the fitter of the two parents. As a result, the algorithm can expect the offspring to be less diverse, but its potential to improve on the parent is significant. The CLUSTER RECOMBINATION method differs from the others in that only one parent is selected in the selection phase. The other parent is generated dynamically using the SCLP algorithm, as described above. Once the two parents have been selected and initialized, the algorithm continues as in the APPLIED INPUT CLUSTERING method. Finally, PARTITION RECOMBINATION works identically to the CLUSTER RECOMBINATION technique. The only difference lies in the way the second parent is created; instead of using the SCLP algorithm, this method uses the graph partitioning tool KAHIP [53, 55] with a random value for the partition size $k \in [2, 64]$ and the balance constraint $\epsilon \in [0.03, 0.5]$. The partitioning problem is closely related to the clustering problem, with two key differences: the number of partitions k is predetermined, and a balance constraint ensures that the total weight of each partition does not exceed $(1 + \epsilon) * \lceil \frac{c(V)}{k} \rceil$. The objective function is typically to minimize the cut size, being the total number of edges spanning between two nodes of different partitions. A key feature of the last three flat recombination operations is that the offspring maintains at least the same solution quality as the better of the two parents, since the better clustering of the parents is applied to the overlay contracted graph, which is then refined using the LOUVAIN algorithm.

The multi-level recombination operation works by first selecting two individuals $\mathcal{C}_1$ and $\mathcal{C}_2$ from the population to be the parents, as detailed in the selection phase above. Then, both individuals are used as input to a modified LOUVAIN algorithm as follows. In the modified LOUVAIN algorithm, the graph restricts the way in which contraction takes place. Let ε be the set of inter-cluster edges of $\mathcal{C}_1$ and $\mathcal{C}_2$. The edges in ε are blocked during the LOUVAIN coarsening phase, meaning that these edges are not contracted during the multi-level scheme. The contraction phase is stopped when no more contractible edges exist, resulting in the coarsest graph. The algorithm then applies the clustering of the better of the two parents with respect to solution quality to the coarsest graph to be used as an initial clustering, after which the uncoarsening phase begins. Note that the modularity score of the coarsest graph matches that of the fitter parent. During the uncoarsening phase, local search is repeatedly applied to optimize modularity on each level of the contraction hierarchy. This

ensures that, in the end, the found clustering is at least as good as the better of the two parents with respect to the modularity score. By not contracting edges that were identified as inter-cluster edges, the algorithm combines structural properties from both parents.
An important aspect to mention is that during the recombination operation, the number of clusters can only decrease compared to the number of clusters of the parents. The mutation phase counteracts this behavior and is executed with probability p_m instead of the recombination phase, where a random subset of the clusters of the two chosen parents is split in half. The splitting operation is performed using the KAHIP [53, 55] graph partitioning framework, which splits a cluster into two balanced blocks while minimizing inter-cluster edges between the two new blocks. The two modified individuals are then used as part of the multi-level recombination operation. Initially, the two mutant individuals may have reduced scores, but using them in the multi-level recombination operation can improve their scores and introduce more diversity into the population upon insertion, as nodes are more likely to be moved out of smaller clusters by local search.
As the VIECLUS algorithm has a population size constraint, choosing which individual to evict is a crucial task in maintaining a high-quality but diverse population to prevent premature convergence of the algorithm. To achieve this, VIECLUS evicts the individual whose inter-cluster edge set most closely resembles that of the offspring, provided that its solution quality is lower. If no such individual exists, the offspring is evicted.
VIECLUS can be run asynchronously in parallel and uses a parallelization scheme that has been successful for the graph partitioning problem [52]. We refer the reader to [4] for a more detailed overview of the parallel implementation.
While VIECLUS produces high-quality solutions in a relatively short time, it is a memory-intensive algorithm for large graphs, as storing multiple clustering instances of the graph in-memory could become a challenge. Since most modern graphs are massive, with millions of nodes and billions of edges, there has been a growing interest in-memory efficient algorithms that still produce high-quality results within a short time-frame.

3.2 Streaming Graph Clustering

One approach to achieve a memory-efficient algorithm is by using a streaming algorithm. When observing the research conducted around the graph clustering problem, most of the state-of-the-art solutions are in-memory algorithms, with limited research on streaming graph clustering algorithms.
The algorithm introduced by Hollocou et al. [25], which we refer to as HOLLOCOU, is one of the few streaming graph clustering algorithms that directly optimizes modularity. We refer the reader to [25] for a full proof of this claim. The streaming model used throughout the algorithm is a one-pass edge-streaming model, as explained in Section 2.5, where only a single edge $e = (u, v) \in E$ is loaded into memory at a time.
The intuition behind the HOLLOCOU algorithm is based on the definition of a clustering, following the paradigm of intra-cluster density versus inter-cluster sparsity. Nodes tend

to be more connected within their cluster than to nodes in different clusters. With this in mind, Hollocou et al. [25] argue that if one picks a random edge, it is more likely to be an intra-cluster edge than an inter-cluster edge. Therefore, they claim that if one can assume that the edges are streamed in a random order, then it is expected that more intra-cluster edges arrive before inter-cluster edges. This assumption is the cornerstone of the algorithm, where, for each streamed edge, the two endpoints are placed in the same cluster if the edge is an *early* edge. An edge is classified as early if the current cluster volume of both endpoints does not exceed a chosen threshold v_{max}. The volume of a cluster is defined as the sum of the degrees of the nodes in the same cluster.
The algorithm starts with all nodes as singletons. For each streamed edge $e = (u, v)$, the algorithm either (a) assigns u to the cluster of v, (b) assigns v to the cluster of u, or (c) does nothing and leaves the nodes in their original clusters. The choice of action solely depends on the updated cluster volumes of the endpoints with respect to all previously streamed edges. If the cluster volumes of nodes u and v are both below the threshold v_{max}, indicating the arrival of an early edge, the node belonging to the smaller cluster is assigned to the cluster of the other endpoint. If this condition is not met, HOLLOCOU takes no action and proceeds to the next edge.
The algorithm is a linear-time algorithm with complexity $\mathcal{O}(m)$ and has a linear space complexity of $\mathcal{O}(n)$, since for each node, the algorithm only needs to store three integers: the node degree d_u, the cluster assignment c_u, and the corresponding cluster volume v_k.
On the one hand, the HOLLOCOU algorithm has shown to be a memory- and runtime-efficient clustering algorithm, able to efficiently cluster huge graphs with millions of nodes and billions of edges. On the other hand, our experiments indicate that the solution quality is low. This could be because all node assignment decisions are made while having limited global graph knowledge, which can lead to suboptimal decisions.

CHAPTER 4

CluStRE: Streaming Graph Clustering with Multi-Stage Refinement

In this chapter, we introduce, CLUSTRE, our algorithm for computing a **Clu**stering in a **S**treaming fashion using multi-stage refinement techniques including **R**e-streaming and **E**volutionary heuristics. Our primary goal with CLUSTRE is to bridge the gap between high-quality in-memory clustering algorithms and streaming graph clustering algorithms. More specifically, we aim to achieve a solution quality that is comparable to state-of-the-art in-memory algorithms while maintaining peak memory consumption comparable to that of current state-of-the-art streaming graph clustering algorithms. A key challenge for streaming algorithms is their limited access to global graph knowledge when making decisions, often resulting in suboptimal cluster assignments. The CLUSTRE algorithm overcomes this well-known limitation by incorporating multiple refinement stages, each of which leverages graph knowledge through various techniques.
We begin by providing an overview of the overall structure of CLUSTRE and its lightweight streaming approach to the graph clustering problem in Section 4.1. Next, in Section 4.2, we present our approach to modularity optimization in a streaming setting. We then introduce various optional refinement approaches to enhance solution quality and exploit partial global information. These approaches include constructing a dynamic quotient graph model, which serves as input to a memetic in-memory graph clustering algorithm, elaborated in Section 4.3, and a re-streaming phase combined with a local search, discussed in Section 4.4.

4.1 Overall Algorithm

CLUSTRE is a multi-stage streaming graph clustering algorithm that addresses the quality limitations of conventional streaming algorithms by incorporating global knowledge

Algorithm 1: Overall Structure of the CLUSTRE algorithm

Input: Graph $G = (V, E)$, Flags: *refine*G_Q, *restream*LS
Output: Clustering $\mathcal{C}$
$\mathcal{C}[v] \leftarrow v \quad \forall v \in V$; `// node cluster assignments` $\Theta(n)$ `memory`
Initialize empty quotient graph G_Q ; `//` $\mathcal{O}(|E_{G_Q}|)$ `memory`
foreach $v \in V$ *(node stream)* **do**
  $\mathcal{C}[v] \leftarrow$ COMPUTECLUSTER$(v, N(v), \mathcal{C})$; `// max modularity gain`
  if *refine*G_Q **then**
    UPDATEQUOTIENTGRAPH$(v, \mathcal{C}, G_Q)$; `// on-the-fly` G_Q `update`
if *refine*G_Q **then**
  $\mathcal{C} \leftarrow$ MEMETICREFINEMENT(G_Q) ; `// memetic clustering`
if *restream*LS **then**
  RESTREAMLOCALSEARCH$(\mathcal{C})$; `// re-streaming + local search`
return $\mathcal{C}$;

through memetic clustering, re-streaming, and local search. CLUSTRE is a node streaming model that only loads a single streamed node v and all of its neighbors $N(v)$ to memory (RAM) at a time. For each streamed node v, we compute its optimal clustering assignment C^* and assign it to v. We achieve this by considering a set of candidate clusters $C(N(v))$ and the possibility of assigning the streamed node v to a cluster of its own. The decision is based on selecting the cluster that yields the highest modularity gain in the current setting. Optionally, during the node streaming, CLUSTRE constructs a dynamic quotient graph data structure that represents the input graph in a compressed form while preserving essential structural features. Each node in the quotient graph G_Q corresponds to a cluster in G. Edges between distinct nodes in the quotient graph represent inter-cluster edges in the original graph, while self-edges in G_Q model intra-cluster edges in G. The quotient graph G_Q retains essential properties, one of which is that the modularity of any clustering computed on G_Q is equivalent to that of the corresponding clustering on G. This claim is formally proven in Theorem 2 (Section A.1) of the Appendix. After completing the streaming phase and fully constructing the quotient graph G_Q, the algorithm optionally performs further enhancement to modularity by using the compressed data structure G_Q as the input graph for a memetic clustering algorithm. The approach combines and splits clusters, significantly expanding the search space to optimize modularity. Once the memetic algorithm has finished processing, the clustering with the highest score is projected back onto G_Q. At this stage, the algorithm either outputs the clustering by projecting the clustering of G_Q back onto G, or proceeds to an additional refinement phase incorporating re-streaming and local search, iterating until a stopping criterion is met. The overall implementation of CLUSTRE is outlined in Algorithm 1, with details for each step given in the following sections.

4.2 One-Pass Streaming with Modularity Gain Scoring

When streaming the graph, CLUSTRE employs a one-pass node streaming model, loading only one node v and its neighbors $N(v)$ into memory at a time. Initially, each node is assumed to belong to its own singleton cluster. As streaming begins, our algorithm assigns the streamed node v to the cluster C^* that maximizes the modularity gain $\Delta Q_{v:C_{cur}\to C_{can}}$, where C_{cur} corresponds to the current cluster assignment and C_{can} represents a candidate cluster to which the streamed node v can be assigned. The modularity gain $\Delta Q_{v:C_{cur}\to C_{can}}$ is computed using Equation 4.1, which is also used in the LOUVAIN algorithm [5] and noted in other papers [50].

$$\Delta Q_{v:C_{cur}\to C_{can}} = \frac{1}{m}(K_{v\to C_{can}} - K_{v\to C_{cur}}) - \frac{d_w(v)}{2m^2}(d_w(v) + \text{vol}(C_{can}) - \text{vol}(C_{cur})) \quad (4.1)$$

Here we set $K_{v\to C_i}$, where C_i is a cluster, to be the total number of edges spanning from the node v to any node u such that $u \in C_i$. We denote $d_w(v)$ to be the weighted degree of a vertex v. Lastly, $\text{vol}(C_i)$ corresponds to the volume of cluster C_i being $\sum_{v\in C_i} d_w(v)$. Note that the delta modularity function in Equation 4.1 outputs the change in modularity when moving a node v from its current cluster C_{cur} to a candidate cluster.
An important consideration is that candidate clusters must consist only of nodes that have already been streamed. The volume of a cluster is unknown beforehand since the degrees of the nodes are only available when streamed. As a result, clusters containing non-streamed nodes are infeasible for the delta modularity Equation 4.1, whereas feasible clusters are those whose assigned nodes have all been streamed. Therefore, all streamed nodes are initially singletons, as their clusters were previously deemed infeasible. At first, it might appear that the candidate clusters correspond to all feasible clusters. However, calculating the modularity gain for each feasible cluster would be computationally expensive. This inefficiency can be avoided by the following observation:

Theorem 1. *Assigning a streamed singleton node to a cluster it is not adjacent to will never lead to a positive modularity gain; that is,* $\Delta Q_{v:C_{cur}\to C_{can}} \leq 0$.

Proof. Consider a streamed node v initially belonging to its own singleton cluster C_{cur} for reasons mentioned above. Since v is a singleton, it has no intra-cluster edges, only inter-cluster edges. Additionally, we know that the volume of a singleton cluster is equivalent to the weighted degree of the node v, denoted by $d_w(v)$. Therefore, we can simplify and adapt Equation 4.1 to:

$$\Delta Q_{v:C_{cur}\to C_{can}} = \frac{1}{m}\left(K_{v\to C_{can}}\right) - \frac{d_w(v)}{2m^2}\left(d_w(v) + \text{vol}(C_{can}) - d_w(v)\right), \quad (4.2)$$

Now, if node v is not adjacent to any node in C_{can}, then by definition:

$$K_{v \to C_{can}} = 0.$$

Substituting this into the equation simplifies the delta modularity gain function as follows:

$$\Delta Q_{v:C_{cur} \to C_{can}} = 0 - \frac{d_w(v)}{2m^2} \cdot \text{vol}(C_{can}) = -\frac{d_w(v) \cdot \text{vol}(C_{can})}{2m^2}. \tag{4.3}$$

Since $d_w(v) \geq 0$, $\text{vol}(C_{can}) \geq 0$, and $m \geq 0$ the resulting modularity gain $\Delta Q_{v:C_{cur} \to C_{can}}$ is non positive:

$$\Delta Q_{v:C_{cur} \to C_{can}} \leq 0.$$

Therefore, no positive modularity gain ($\Delta Q > 0$) can be achieved by assigning v to a non-adjacent cluster. ■

Theorem 1 shows that we can disregard all clusters that are not adjacent to the streamed node, since their gain will never be > 0, thereby improving computational efficiency. Hence, the candidate clusters for a streamed singleton node are always its feasible neighboring clusters $C(N(v))$. Only when all feasible neighboring clusters deem to result in a negative modularity gain do we leave the streamed node as a singleton.

$$C^* = \underset{C_{can} \in C(N(v))}{\text{argmax}} \; \Delta Q_{v:C_{cur} \to C_{can}} \tag{4.4}$$

As indicated in Equation 4.1, the delta modularity function requires determining the volume of neighboring clusters. To efficiently retrieve the volume of a feasible cluster, we maintain an initially empty vector. For each feasible cluster, we allocate memory and adjust its volume accordingly each time a node is streamed and assigned to a cluster. We do not allocate memory for infeasible clusters, as the volume of such a cluster is unknown, rendering this allocation unnecessary. Memory is allocated only after a node has been streamed and assigned to a feasible cluster. This approach saves memory allocation in cases where a streamed node yields a positive gain with one of its neighboring clusters. In such cases, we assign the neighboring cluster to the streamed node and adjust the cluster volume accordingly without allocating any additional memory in the vector for the singleton node, thereby saving crucial memory resources.
Figure 4.1 illustrates a small implementation detail in our program that allows for fast and efficient computation of the delta modularity function. To facilitate this, we update a vector $ArtNodes$ while processing the neighboring nodes, which stores the total edges to different clusters. This vector can be interpreted as a set of artificial nodes $ArtNodes$ that are initialized and continuously updated as neighboring nodes are streamed from disk. Ultimately, each artificial node corresponds to tuple of a unique cluster-ID, with the total number of edges connecting the streamed node v to the specific cluster. This vector enables efficient retrieval of the parameter $K_{v \to C_{can}}$ for each candidate cluster $C(N(v))$ in the delta modularity function. Notably, the $ArtNodes$ vector is reset for each streamed node.

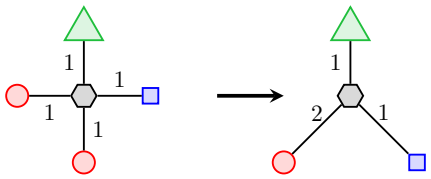


Figure 4.1: The construction of the artificial nodes. On the left, the initial representation of the streamed node (black) and its neighbors. The colors and shapes represent the cluster assignments of the nodes. On the right, the construction of the artificial nodes is shown. The black node now has only one edge with weight 2 to the red cluster, representing the two initial edges spanning between two nodes in the red cluster and the streamed node.

In cases where the streamed node v is an isolated node, meaning it does not share edges with any other nodes in the graph, we assign it its own singleton cluster. Note that any cluster assignment of v does not affect the overall modularity score. As shown in Section 2.2.1, the modularity function only considers edge density (intra-cluster density and inter-cluster sparsity) and neglects node density. This means that the modularity scores of a set of clusterings, where only the cluster assignment of v changes, all have the same modularity score. A formal proof is omitted, as this result can be extracted directly from the modularity objective function (the degree of an isolated node $deg(v) = 0$) specified in Section 2.2.1. Our decision to assign it its own cluster stems from the idea that a cluster with no neighbors has no similarities or common attributes with the rest of the graph and is therefore likely to be in its own group.
Currently, there is no restriction on the total number of clusters that can be initialized during the streaming phase. This could pose a problem when the algorithm is executed on a machine with limited resources. Imposing an upper limit on the total number of clusters enables a trade-off between solution quality and memory consumption. Once this limit is reached, the streamed node can no longer remain a singleton. Instead, it is assigned to the cluster that yields the best modularity gain among the candidate clusters $C(N(v))$ even if they all yield a negative gain, to maintain computational efficiency.
For each streamed node v we compute the clustering in $\mathcal{O}(deg(v))$ time. The overall runtime of the streaming phase is $\mathcal{O}(n * \Delta)$, where Δ is the maximum degree of a node in the graph. This efficiency is achieved through the aforementioned data structures, which allow constant-time access to the information required by the delta modularity function. The modularity maximization approach used in this streaming phase is nearly identical to the first phase of the LOUVAIN method [5], with the exception that each node is visited only once, in the streaming order provided by the input graph.

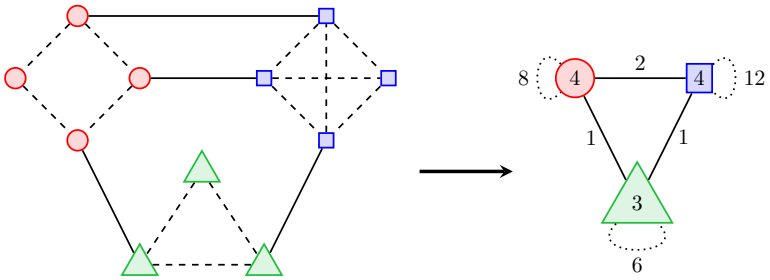


Figure 4.2: An illustration of the quotient graph construction G_Q (right) from the input graph G (left) with all unitary weights. Clusters are represented by unique colors and shapes. Thick lines illustrate inter-cluster edges, while dashed lines depict intra-cluster edges. Each cluster in the input graph G is represented by a single node in the quotient graph G_Q, with a weight corresponding to the total number of nodes assigned to the cluster in G. Weighted edges between nodes in the quotient graph represent the number of inter-cluster edges connecting the corresponding clusters in G. Intra-cluster edges in G are represented by weighted self-loops in G_Q, counted twice, once for each directed intra-cluster edge in G.

4.3 Modularity Refinement via Memetic Clustering

The first refinement option in the CLUSTRE algorithm is memetic refinement, where we enhance solution quality by integrating partial global information and evolutionary heuristics. To achieve this, CLUSTRE constructs a dynamic quotient graph G_Q during the streaming phase, which serves as the input to the state-of-the-art in-memory memetic graph clustering algorithm VIECLUS [4]. This approach closely resembles the LOUVAIN method, with the key difference being that the quotient graph is computed on-the-fly rather than after all nodes have been processed. Computing the quotient graph on-the-fly is crucial because a node's edges become inaccessible after processing, making post-streaming construction infeasible.

The quotient graph G_Q captures the structural properties of the input graph in a compressed form by introducing edge and node weights, as illustrated in Figure 4.2. Each initialized cluster C_i of the input graph G is represented as a single *supernode* v'_i. Each supernode v'_i is assigned a weight, which corresponds to the total number of nodes belonging to cluster C_i in G. Edges in the quotient graph are derived from the input graph with the following rules: an edge between two supernodes v'_i and v'_j exists if and only if at least one inter-cluster edge exists between clusters C_i and C_j. Each edge between two supernodes is weighted and corresponds to the total number of inter-cluster edges between the respective clusters in the input graph G. A supernode v'_i may also contain self-loops. A self-loop on v'_i in G_Q is inserted if and only if cluster C_i in G has at least one intra-cluster edge. This self-loop is also weighted and corresponds to the total number of intra-cluster edges in C_i. Each undirected intra-cluster edge is counted twice, once for each directed intra-cluster edge, to

Algorithm 2: UPDATEQUOTIENTGRAPH $\big(v, G_Q, \mathcal{C}\big)$: On-the-Fly G_Q update

Input: v (streamed node), G_Q representing quotient graph edges (where $G_Q[(C_i, C_j)]$ stores edge weights), Array $\mathcal{C}$ of cluster IDs for all nodes
Output: Updated hashmap for G_Q

```
C_i ← C[v] ;                          // cluster ID of current node v
foreach u ∈ ArtNodes do
    C_j ← u_cluster ;   // cluster ID of neighboring cluster u_cluster
    w ← u_weight ;        // edge weight from v to neighbor u_cluster
    if C_i = C_j then
        w ← 2 * w ;          // double the weight for self-loops
    if (C_i, C_j) ∉ G_Q then
        G_Q[(C_i, C_j)] ← w ;       // Insert new edge (C_i, C_j) with w
    else
        G_Q[(C_i, C_j)] ← G_Q[(C_i, C_j)] + w ;       // Update edge weight
return G_Q;
```

correctly set $w(v'_i, v'_i) = K_{C_i \to C_i}$.

By constructing the quotient graph as detailed above, we ensure that key properties of G are preserved. A major feature is that the modularity score of any clustering computed in the quotient graph G_Q remains equivalent to that of the input graph G when the cluster assignments of G_Q are projected back onto G. The claim is formally proven in Theorem 2 of the Appendix. This result demonstrates that the quotient graph serves as a low-memory representation of the input graph G and its clustering. G_Q enables efficient access to partial global information for modularity optimization using in-memory clustering algorithms, such as VIECLUS in our case. Clustering computations on the quotient graph are more efficient because moving a single node in G_Q effectively relocates an entire set of nodes in G, considering node and edge weights. This results in larger and faster modularity score improvements.

To construct the quotient graph G_Q on-the-fly, we update G_Q after a cluster assignment to the stream node is made. The quotient graph data structure is implemented as a hash map, where keys are pairs representing the quotient graph edges E_{G_Q}, allowing an average time complexity of $\mathcal{O}(1)$ for insert and update operations. By utilizing the artificial nodes constructed during the streaming stage, we can efficiently update the quotient graph data structure. However, constructing the quotient graph requires additional $\mathcal{O}(|E_{G_Q}|)$ memory and takes linear time in the size of E_{G_Q}. Algorithm 2 outlines the update operations for the quotient graph. Note that we iterate over the artificial nodes to optimize runtime.

As previously mentioned, after the streaming phase has ended and the quotient graph has been constructed, G_Q is the input to the state-of-the-art in-memory clustering algorithm VIECLUS. As elaborated earlier in Section 3.1.4, this heuristic consists of five stages:

the initialization phase, selection phase, recombination phase, mutation phase, and eviction phase. Upon completion, VIECLUS outputs the clustering with the highest modularity score. Notably, the algorithm optimizes modularity while considering node and edge weights. Thus, the modularity score of any clustering computed on the quotient graph G_Q remains equivalent to that of the input graph G.
An important aspect to highlight is that the evolutionary heuristic executes until a predefined time limit is reached. The time limit is a tunable parameter provided as input to VIECLUS. Optimally setting this tunable parameter is part of the tuning study. We expect this parameter to serve as a trade-off between runtime, solution quality, and peak memory consumption. Particularly, the more time provided to the memetic refinement, the higher we expect the solution quality to be, while simultaneously also increasing the peak memory consumption, as the more time VIECLUS is given, the larger the explored solution space becomes, and the more instances are generated. Both factors contribute to increased solution quality and peak memory.

4.4 Modularity Refinement via Re-Streaming with Local Search

The second refinement strategy used in the multi-stage CLUSTRE algorithm is re-streaming with local search. More specifically, this stage consists of two phases: a re-streaming phase and a local search phase. This refinement can be applied either directly after the initial streaming phase or after the memetic refinement stage.
In the re-streaming phase, the graph is processed again, similar to the initial streaming phase. The main benefit of re-streaming is that, from the first re-streaming iteration, all nodes have already been assigned to feasible clusters. This means that modularity computation is more informed, as the set of candidate clusters always corresponds to the whole set of neighboring clusters. This improves the modularity computation, enabling higher-quality decisions. During the re-streaming phase, we use the same approach to compute the delta modularity gain as described in Equation 4.1, with the exception that we do not consider the possibility of assigning the streamed node a new cluster to become a singleton. Instead, only the neighboring clusters are considered to determine whether moving the node results in a modularity gain.
It is important to note that the algorithm allows for multiple re-streams of the graph, which is a tunable parameter. This parameter is assessed and evaluated in the tuning experiments and serves as a trade-off between solution quality and runtime. We expect to improve solution quality by increasing the number of re-streams. However, the key question is by how much and whether multiple re-streams are worth it, given the high I/O and computational costs of each additional re-stream iteration.
While the re-streaming phase processes the entire graph as a whole, the local search phase uses an optimization technique that significantly reduces the runtime by minimizing I/O

Algorithm 3: RESTREAMLOCALSEARCH$(\mathcal{C})$

Input: *ActiveNodes*, $\Delta Q_{curr} \leftarrow \infty$, $startTime \leftarrow$ CURRTIME, $ls_{\text{cutoff}} \in [0, 1]$, $ls_{\text{time\_frac}}$, ls_{time_limit}

while *ActiveNodes* $\neq \emptyset$ ***and*** $\Delta Q_{total} \geq ls_{cutoff} * Q_{curr}$ ***and*** CURRTIME $-$ *startTime* $<$ $ls_{\text{time\_frac}}$ ***and*** CURRTIME $-$ *startTime* $< ls_{time\_limit}$ **do**

  NextActiveNodes $\leftarrow \emptyset$, $\Delta Q_{curr \leftarrow 0}$; // ΔQ_{curr} = mod current round

  foreach $v \in$ *ActiveNodes* **do**

    $C^* \leftarrow$ COMPUTECLUSTER$(v, N(v), \mathcal{C})$;

    if $C^* \neq \mathcal{C}[v]$ **then**

      $\mathcal{C}[v] \leftarrow C^*$;

      foreach $u \in N(v)$ **do**

        Add u to NextActiveNodes;

      $\Delta Q_{curr} \leftarrow \Delta Q_{curr} + \Delta Q_v$; // ΔQ_v = mod gain by moving v

  ActiveNodes $\leftarrow$ NextActiveNodes;

  $Q_{\text{total}} \leftarrow Q_{\text{total}} + \Delta Q_{curr}$; // ΔQ_{total} = overall mod score of $\mathcal{C}$

return $\mathcal{C}$;

operations. During the final re-streaming iteration, we track a set of nodes, which we call *active nodes*. A node is labeled as an *active node* if and only if at least one of its neighboring nodes has changed its cluster assignment. In the local search phase, we load only individual *active nodes* and their neighbors from disk, significantly reducing I/O operations. Then, we iteratively evaluate each active node's optimal cluster assignment using the delta modularity gain function in Equation 4.1. For each local search iteration, we keep track of a new set of active nodes, which serve as the set for the next local search iteration to further optimize the clustering. We restrict the cluster reassignment evaluation to active nodes, due to the following observation: only nodes whose neighbors have been reassigned exhibit different values for the first component of the modularity gain function in Equation 4.1. We refer to the first component of the modularity gain function as:

$$\frac{1}{m}\left(K_{v \to C_{can}} - K_{v \to C_{cur}}\right) \tag{4.5}$$

More specifically, when a node's neighborhood is reassigned, it alters the total edge weight between the node and its neighbors, which are assigned to the same cluster, relative to the edge weight between the node and the candidate cluster. Conversely, if a node's neighborhood assignments remain unchanged, the value of the first component in Equation 4.1 also remains unchanged across consecutive re-streams. The relative total edge weights remain the same across all candidate clusters. This indicates that the delta modularity gain of nodes whose neighborhoods have not been reassigned does not change significantly. Only

the second component, which accounts for the volume of the candidate clusters, may yield a different value. By focusing only on active nodes, we strike a balance between optimizing I/O efficiency and preserving nodes that could significantly impact the clustering due to substantial changes in their delta modularity scores.
To efficiently extract the neighbors of active nodes, we store the positions of nodes in the input graph in a vector during the final re-streaming iteration of the entire graph. The vector allows quick access to the neighbors of each node. To maintain memory efficiency, we only store the position of every 10th node. Therefore, in practice, we compute the position (using the *mod* function) of the nearest stored node and then iterate up to nine consecutive times. This actively saves many I/O operations, thereby decreasing the overall runtime. Note, however, that at least one re-stream phase is required before the local search phase can take place to initialize the active node set.
The local search phase executes until one of the following conditions is met: (a) the local search reaches convergence, meaning no active nodes exist; (b) the modularity improvement of the current round falls below a certain threshold, ls_{cutoff}, where ls_{cutoff} is a tunable parameter; or (c) a certain time limit is reached, controlled by two tunable parameters: $ls_{\text{time\_frac}}$, which defines a relative time value to spend in the local search phase, and $ls_{\text{time\_limit}}$, which sets an absolute time limit for the local search phase.

CHAPTER 5

Experimental Evaluation

After introducing our streaming graph clustering algorithm, CLUSTRE, in Chapter 4, we now provide an experimental evaluation of CLUSTRE. We begin by detailing the hardware used for the experiments in Section 5.1, followed by an explanation of our methodology in Section 5.2, and a description of the datasets in Section 5.3. Then, we showcase, in Section 5.4, several tuning experiments to identify the best parameter values for CLUSTRE. Subsequently, we introduce, in Section 5.5, various configurations of our algorithm and compare them, highlighting their respective use cases. Finally, in Section 5.6, we compare CLUSTRE against state-of-the-art in-memory and streaming algorithms to evaluate its performance.

5.1 Hardware

Each experimental instance was executed on a single core of a machine equipped with an Intel Xeon Silver 4216 CPU, 93 GB of RAM, 16 MB of L2 cache, and 22 MB of L3 cache. The CPU has 16 physical cores, each supporting two threads. The machine operates at a base clock speed of 2.1 GHz, with dynamic scaling between 0.8 GHz and 3.2 GHz. The system runs Ubuntu 20.04.01 LTS with a Linux kernel version 5.4.0-152-generic, running on an x86_64 architecture.

5.2 Methodology

The experiments are structured into two distinct phases: the tuning phase and the test phase. In the tuning phase, we conduct a series of experiments to identify the best configuration for our algorithm. We begin with the base configuration of CLUSTRE, where no upper limit is set on the total number of clusters, and the local search refinement phase runs until convergence. The duration of the memetic refinement is initially unfixed, as it is the first

Research Question	Description
RQ1	What is the impact of evolutionary 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? Can CLUSTRE bridge the gap between existing streaming and in-memory clustering algorithms?
RQ3	How does CLUSTRE balance runtime and memory consumption with solution quality, compared to other streaming and in-memory clustering algorithms?
RQ4	How well does CLUSTRE retrieve ground-truth communities compared to existing streaming algorithms?

Table 5.1: Research questions investigated throughout this experimental evaluation

parameter we tune. Once we determine the best configuration, we examine our research questions, listed in Table 5.1.

The first aspect we are interested in is evaluating the impact of evolutionary clustering and re-streaming on the solution quality of CLUSTRE, serving as our first research question **RQ1**. In the test phase, we compare our algorithm with other state-of-the-art clustering algorithms. Specifically, we analyze how the solution quality of CLUSTRE compares with state-of-the-art streaming clustering algorithms and in-memory clustering techniques. In particular, we examine whether we can bridge the solution quality gap between streaming and in-memory graph clustering algorithms, which is addressed in **RQ2**. With **RQ3**, we aim to evaluate how effectively CLUSTRE balances runtime, memory consumption, and solution quality compared to state-of-the-art streaming and in-memory clustering algorithms. Finally, in **RQ4**, we compare CLUSTRE's ability to retrieve ground-truth communities against existing state-of-the-art streaming clustering algorithms. The state-of-the-art clustering algorithms used for comparison with CLUSTRE include the multi-level LOUVAIN algorithm [5], the evolutionary VIECLUS algorithm [4], and the streaming graph clustering algorithm by Hollocou et al. [25].
It is important to note that we obtained the C++ implementation of the HOLLOCOU algorithm from its official GitHub repository. We observed that the implementation does not support streaming from disk. Instead, it loads the entire edge set into memory before individually processing each edge. To ensure a fair comparison, particularly regarding memory consumption, we modified the implementation to support edge streaming from disk. As mentioned in Chapter 3, the HOLLOCOU algorithm also requires the parameter $v_{\max}$, which defines the maximum volume constraint for each cluster and must be specified in advance. However, the original authors provide no guidance on how to choose an appropriate value for this parameter, nor do they specify the value used in their experiments. To ensure fairness, we also conduct a tuning study on the $v_{\max}$ parameter in Appendix A.2.

All algorithms listed for the experiments are implemented in C++. While all competitor algorithms are implemented with C++11, our algorithm is implemented with C++20. Additionally, all algorithms are compiled with g++ version 11.4.0 using the full optimization flag -O3. When incorporating the memetic graph clustering algorithm, we run each instance with three different seeds and compute their geometric mean. In these experiments, we measure running time, including the I/O time for all algorithms, solution quality, measured by the modularity score, and the memory consumption. To measure the memory consumption, we extract the maximum resident set size given in kilobytes. When running our experiments, we use the *GNU parallel* tool [59] to run eight instances independently in parallel on our machine.
To visually benchmark our algorithms we often plot the results in the form of a *performance profile*, introduced by Dolan and Moré [14]. We use performance profiles to compare different objectives, such as peak memory consumption, runtime or solution quality. The plot is a 2D graph where the x-axis represents the performance ratio τ, which always begins at one and increases for minimization problems and decreases for maximization problems. The y-axis represents the fraction of all tests, ranging from zero to one, being the entire test set. For each algorithm A, we plot a separate line on the graph. Every point $(frac, \tau)$ on the line of the algorithm represents the fraction $frac$ of all instances of A where the current measured objective is less than or equal to τ times the measured objective of the best algorithm for the same instance. As instances may have different algorithms on which they perform best, a point $(frac, \tau)$ indicates that the algorithm A is for a fraction $frac$ of the instances never more than τ times worse than the best algorithm on a per-instance level. The advantages in performance profiles are that they are less sensitive to outliers. A single bad performance for one instance does not disproportionately affect the overall comparison of the algorithm. Additionally, we use box plots to illustrate the distribution of runtime and memory consumption. A box plot displays the five-number summary of a dataset. The five-number summary consists of the minimum, first quartile (Q1), median, third quartile (Q3), and maximum values. In a box plot, the box spans from the first quartile (Q1) to the third quartile (Q3). A vertical line inside the box indicates the median value of the dataset. Furthermore, let a measured objective result be denoted by σ_A for some clustering output found by algorithm A. We compare this result with other results found by different algorithms using the following tools: *improvement* over an algorithm B, computed as a percentage with the formula $(\frac{\sigma_A}{\sigma_B} - 1) * 100$ and the *relative* value over an algorithm B, calculated as $\frac{\sigma_A}{\sigma_B}$. When the average of all instances is required, we use the geometric mean, which minimizes the influence of outliers by giving each instance the same influence on the final score. To assess and compare the performance of our algorithm in retrieving ground-truth communities we utilize the Normalized Mutual Information [30] metric to evaluate the similarity between two clusterings of the same instance, which is widely recognized as reliable [32].

5.3 Dataset

The graphs used in our experiments are listed in Table 5.2. We define two different datasets. The first set of ten graphs is for our tuning phase to obtain the appropriate configuration for our algorithm, while the second set of 20 graphs is used to evaluate the performance of CLUSTRE and test it against state-of-the-art clustering algorithms. To test for robustness and ensure that our algorithm performs well across different graph types and structures, we selected graphs from various benchmark datasets. We sourced the social network graphs from the SNAP dataset [31], as well as from the Laboratory for Web Algorithmics [7, 8, 6], the Koblenz Network Collection [28], and Facebook friendship networks [62]. Web, road, and citation graphs were sourced from the Network Repository [47] and the 10th DIMACS Graph Clustering Implementation Challenge [3]. To conduct our experiments, we converted all graphs to an undirected METIS node-stream format, removing any parallel or self-loops, while assigning unit weights to all nodes and edges.
In addition, our dataset includes a 2D random geometric graph (RGG2D), as well as a 2D random hyperbolic graph (RHG2D) created using KAGEN [19], which model real-world scale-free networks. The RGG2D is formed by randomly distributing nodes within a unit square in Euclidean space. An edge is created between two nodes if the distance between them is less than or equal to a given radius r, which is approximated by Newton's method. An RHG2D graph extends the concept of an RGG2D graph by placing the vertices in hyperbolic space rather than Euclidean space.
An important aspect to mention is that the ground-truth communities provided in the SNAP [31] dataset are not suitable for our ground-truth retrieval performance evaluation, as nodes can belong to multiple clusters. Ground-truth communities represent the true clustering of entities in a dataset. In our algorithm and definition of a clustering, we restrict the assignment of nodes to exactly one cluster. Therefore, we incorporate citation [68] and co-purchase networks [56] with feasible ground-truth communities from PyTorch Geometric. In the citation networks, each node represents a scientific paper, the edges correspond to citations, and the communities correspond to different research topics. For the co-purchase network, each node represents a product, and an edge between two products indicates a frequent co-purchase relationship, with the communities being the product categories.

5.4 Tuning Study

In this section, we conduct experiments to tune parameters used in CLUSTRE, presented in Table 5.3, while also exploring its performance under various configurations. When performing the tuning experiments, we begin with the base configuration of our CLUSTRE algorithm, with no maximum cluster bound, and running local search until convergence. In each tuning study, we determine the best configuration for a specific parameter, which then serves as the default value for the next set of tuning experiments. The nodes are streamed in their natural order as they appear in the input graphs.

Graph	n	m	Type
Tuning Graphs			
Maryland58	20,871	744,862	Social Network
Texas84	36,371	1,590,655	Social Network
coAuthorsDBLP	299,067	977,676	Citations
web-Google	356,648	2,093,324	Web
amazon-2008	735,323	3,523,472	Web
rhg1m10m	1,000,000	10,047,330	Rand. Geo.
in-2004	1,382,908	13,591,473	Web
netherlands	2,216,688	2,441,238	Roads
ljournal-2008	5,363,260	49,514,271	Social Network
germany	11,548,845	12,369,181	Roads
Test Graphs			
Penn94	41,554	1,362,229	Social Network
libimseti	220,970	17,233,144	Social Network
wiki-Talk	232,314	1,458,806	Web
citationCiteseer	268,495	1,156,647	Citations
com-amazon	334,863	925,872	Social Network
coPapersDBLP	540,486	15,245,729	Citations
eu-2005	862,664	16,138,468	Web
hollywood-2011	2,180,759	114,492,816	Social Network
enwiki-2013	4,206,785	91,939,728	Social Network
italy	6,686,493	7,013,978	Roads
great-britain	7,733,822	8,156,517	Roads
arabic-2005	22,744,080	553,903,073	Web
it-2004	41,291,594	1,027,474,947	Web
sk-2005	50,636,154	1,810,063,330	Web
europe	50,912,018	54,054,660	Roads
com-friendster	65,608,366	1,806,067,135	Social Network
rgg_n26	67,108,864	574,553,645	Rand. Geo.
rhg2b	100,000,000	1,999,544,833	Rand Hyp.
uk-2007-05	105,896,555	3,301,876,564	Web
webbase-2001	118,142,155	854,809,761	web

Table 5.2. Graphs for the different phases of the experiments with their node and edge size and type.

Parameter	Symbol	Description
Memetic Refinement Duration	D	Time limit for memetic refinement
Number of Re-streams	R	Number of re-streams before performing local search
Local Search Cutoff	ls_{cutoff}	Local search cutoff based on solution quality
Local Search Time Constraint	$ls_{\text{frac\_time}}$	Local search constraint based on runtime
Max Clusters	$cluster\_frac$	Relative limit on the number of clusters

Table 5.3. Tunable Parameters of CLUSTRE

5.4.1 Memetic Refinement Time Limit

Our tuning experiments begin with the selection of the appropriate duration D in seconds for the memetic refinement phase using the VIECLUS algorithm. To perform this experiment, we exclude the re-streaming and local search refinement from the base configuration of CLUSTRE; thus, it only undergoes the streaming phase, where the dynamic quotient graph is constructed, as explained in Chapter 4, followed by the memetic refinement phase. Other phases are excluded to accurately distinguish the improvement provided solely by the VIECLUS algorithm. Note that after the streaming phase, the solution quality of the dynamic quotient graph remains consistent under different proposed memetic duration configurations. This is because our streaming phase does not involve randomness, and the streaming order of the nodes remains consistent, as it is determined by the input graph. In particular, we aim to evaluate our claim regarding the trade-off between memory consumption and solution quality, as discussed in Chapter 4. This claim posits that spending more time in the memetic refinement phase leads to better solution quality but also results in higher memory consumption. To evaluate this claim, we propose five different durations for the VIECLUS algorithm: $D \in \{15, 30, 60, 300, 600\}$ seconds.
It is important to note that the time limit specified for the VIECLUS algorithm serves only as an estimate and is not treated as a strict cut-off parameter. Many tested instances exceeded this limit; however, the relative additional time spent remained consistent across all instances and durations.
First of all, figure 5.1 illustrates the impact that an evolutionary algorithm has compared to the streaming phase alone (D = 0). From Figure 5.1, it is evident that the memetic refinement algorithm significantly improves solution quality. Even for the shortest duration, $D = 15$, the algorithm improves solution quality by 25.56% compared to no memetic refinement ($D = 0$), but requires 6.37× more memory and is 29.52× slower on average across all tuning instances. The high memory consumption stems from constructing the quotient graph and the evolutionary scheme, while the slower runtime is due to instances like `Texas84`, which require only 0.1 seconds without memetic refinement. In such cases, an additional 15 seconds results in a larger relative runtime increase.
Figure 5.2 presents the results using a performance profile analysis. Runtime analysis is omitted from this figure, as the runtime comparison between different algorithms in our proposed configuration is straightforward and thus provided in Appendix A.3. The runtime

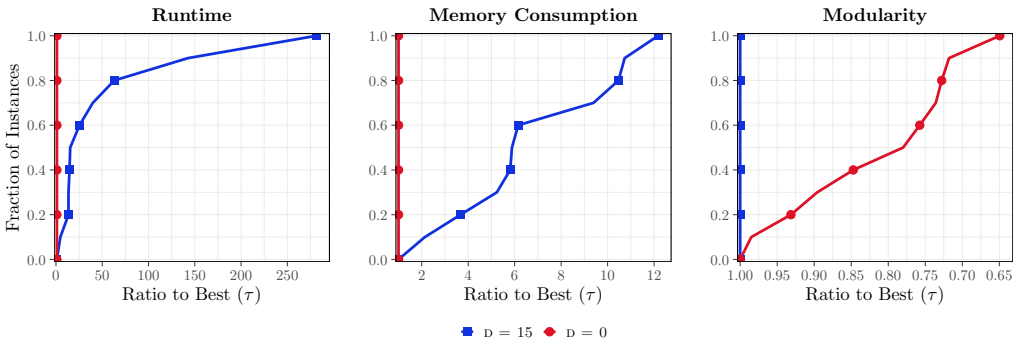


Figure 5.1. Tuning Experiment: Memetic Refinement. Performance profiles for runtime, peak memory consumption, and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with no restriction on the maximum number of clusters. The algorithms underwent a streaming phase followed by a memetic refinement phase with varying duration values (D). Note that $D = 0$ indicates no memetic refinement was applied.

of our proposed configuration consists of the streaming phase plus the memetic refinement duration. Since the streaming phase duration remains similar for each instance across all proposed configurations, the only module that affects the runtime is the memetic refinement phase, which we explicitly vary for each configuration.

Figure 5.2 demonstrates that our expectation from Chapter 4 holds true. When analyzing the performance profile showing memory consumption, we observe that the algorithm running the memetic refinement for the shortest possible time, 15 seconds, also exhibits the lowest memory peak. On average, across all tuning instances, the algorithm with $D = 15$ requires only 62.21% of the memory used by $D = 600$ and 90.47% of the memory required by the second-lowest memory consumption configuration with $D = 30$. The performance profile reveals a correlation between memory consumption and runtime: the longer the runtime, the higher the memory peak. This is mainly because the more time the memetic refinement algorithm is given, the more instances of the quotient graph are initialized, resulting in higher memory consumption. It is interesting to note that the curves for the $D = 300$ and $D = 600$ configurations in the memory peak performance profile are very similar, even though the latter configuration is given twice the time. On average across all instances, the algorithm with $D = 300$ consumes 98.68% of the memory used by the algorithm with $D = 600$, which is practically the same. One reason for this behavior could be that the total number of initialized quotient graph instances has reached its limit in both configurations for most instances. As shown in our graph dataset (Table 5.2), some algorithms operate on graphs with fewer than 50,000 total nodes. This results in a quotient graph with an even smaller number of nodes, which, in turn, makes initializing multiple instances of the quotient graph highly time-efficient.
The analysis of the solution quality performance profile confirms our claim, showing that

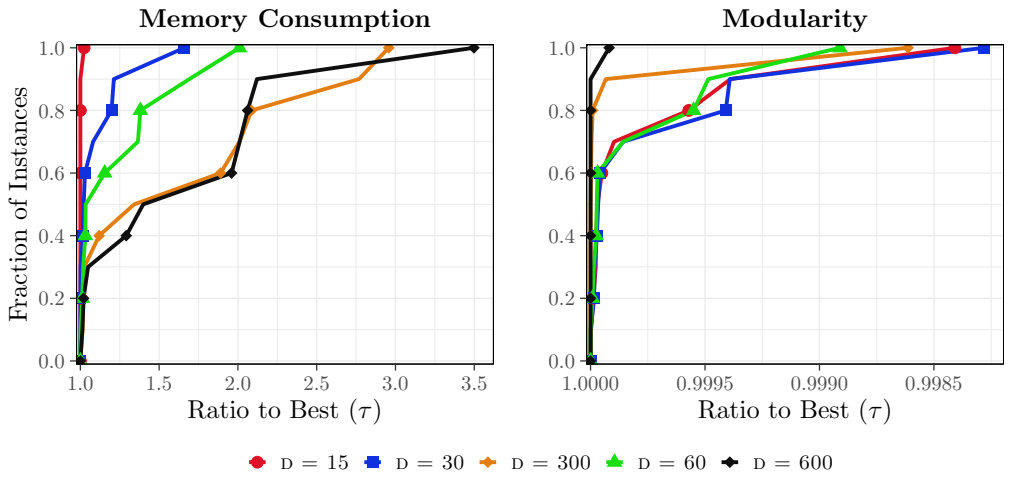


Figure 5.2. Tuning Experiment: Memetic Refinement. Performance profiles for peak memory consumption and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with no restriction on the maximum number of clusters. The algorithms underwent a streaming phase followed by a memetic refinement phase with varying duration values (D).

the algorithm with the highest memetic refinement duration, $D = 600$, yields the best solution quality in about 90% of the instances. This is followed by the algorithm with the second-highest duration, $D = 300$, which yields the best solution for 80% of the instances. However, the three configurations with the shortest memetic refinement time, that is, $D \in \{15, 30, 60\}$, all produce almost identical solution quality results, differing by only 0.01% on average across all tuning instances. This phenomenon could be due to a low number of evolutionary cycles performed in the refinement phase, resulting from the limited time provided, which hinders the algorithm's ability to explore new solutions. This argument could also explain why the algorithm with the $D = 600$ configuration produces, on average across all instances, only about 0.03% better solution quality than the other configurations, suggesting that even ten minutes may not be sufficient for the evolutionary algorithm to thoroughly explore the solution space.

Observation 1. The configuration that offers the best trade-off between solution quality and memory consumption is $D = 15$, the lowest examined value. Not only does $D = 15$ achieve nearly identical results in terms of solution quality, just 0.03% lower than the five-minute and ten-minute memetic refinement configurations, but it also consumes only 62.21% of the memory required by $D = 600$ and 90.47% of that used by $D = 30$ on average across all instances.

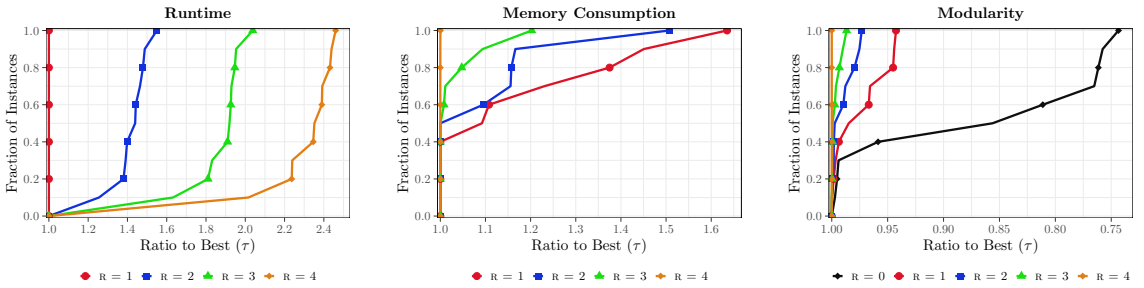


Figure 5.3. Tuning Experiment: Total Re-streams. No Memetic Refinement. Performance profiles for runtime, memory consumption, and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with no memetic refinement, no restriction on the maximum number of clusters, and with varying number of total graph re-streaming indicated by the value of R.

5.4.2 Number of Re-streams

The next tuning parameter we want to investigate is the total number of re-streaming iterations, denoted as R, applied to our algorithm after the initial streaming phase and before the local search refinement phase. In this experiment, we use two different configurations of our algorithm to accurately assess the impact of re-streaming. The first configuration consists of the initial streaming phase, followed by a re-streaming phase in which the graph is fully re-streamed R times to improve solution quality. The second configuration includes a memetic refinement phase between the initial streaming phase and the re-streaming phase. The purpose of these two different configurations is to accurately evaluate how re-streaming alone improves the quality of the initial solutions and evaluate its impact across different configurations. The memetic refinement phase is set to the previously determined 15-second configuration. Local search refinement is excluded in both configurations, as it would distort the effects of re-streaming on both solution quality and runtime. These experiments evaluate the trade-off between solution quality and runtime. We expect that by increasing the total number of re-streams, to improve the solution quality, but at the same time, increase the runtime of our algorithm. Moreover, considering that the active node set is initialized in the last re-streaming iteration, we want to evaluate how varying the number of graph re-streaming iterations affects the total number of active nodes. We expect that the more re-streaming iterations we perform, the fewer active nodes remain.
The parameter values of R selected for this experiment are $R \in \{1, 2, 3, 4\}$. For reference, we include the re-streaming value $R = 0$ in the solution quality performance profile to establish a baseline for comparison. However, the re-streaming configuration with $R = 0$ is not feasible, as the graph must be re-streamed at least once to detect active nodes for the subsequent local search refinement.
Figure 5.3 shows that an increase in the number of re-streams leads to a higher runtime of our algorithm. On average, across all instances without memetic refinement before re-

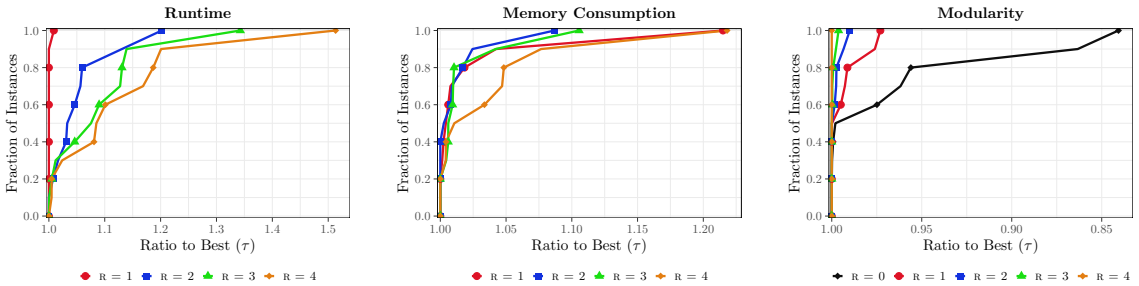


Figure 5.4. Tuning Experiment: Total Re-streams. With Memetic Refinement. Performance profiles for runtime, memory consumption, and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with memetic refinement, no restriction on the maximum number of clusters, and with varying number of total graph re-streaming indicated by the value of R.

streaming, the algorithm with $R = 1$ runs $1.43\times$ faster than the second-fastest algorithm with $R = 2$ and $2.33\times$ faster than the slowest algorithm with $R = 4$. Since the memetic refinement phase adds at least 15 seconds to the total runtime, the relative difference in runtime between algorithms with the memetic configuration is much smaller than without it, although the same trend remains observable, as seen in Figure 5.4. On average, across all instances with the second configuration, where memetic refinement is performed before re-streaming, the algorithm with $R = 1$ runs approximately $1.06\times$ faster than the second-fastest algorithm with $R = 2$ and $1.13\times$ faster than the slowest algorithm with $R = 4$. This occurs because each additional iteration requires re-streaming the entire graph, increasing both computation time and I/O operations.
As expected, when analyzing the solution quality performance profiles of the two configurations, we can see, that increasing the number of re-streaming iterations leads to higher solution quality. An interesting observation is that for both configurations, the quality improvement between one re-streaming iteration and no re-streaming iterations is significant, while the improvement from one to four re-streaming iterations is not as substantial. For the first configuration, the algorithm with $R = 1$ improves solution quality across all instances by an average of 13.49% compared to no re-streaming iterations, while achieving solution quality that is only 2.65% lower than the algorithm with $R = 4$. This can be attributed to the technique used to stream the graph in the initial streaming phase. During the initial streaming phase, the clusters of unvisited neighboring nodes are unknown and considered infeasible (see Section 4.2). These unvisited neighboring nodes are ignored when assigning nodes to clusters, resulting in suboptimal assignments for the first streamed nodes. However, starting from the first re-streaming phase, all nodes are assigned to a feasible cluster, allowing each streamed node to consider all neighboring clusters when making decisions, leading to higher-quality assignments. A similar pattern can be observed in the solution quality performance profile for the memetic refinement configuration, although the effect is not as pronounced, since the memetic refinement phase already significantly improves

Configuration	$R = 1$	$R = 2$	$R = 3$	$R = 4$
With memetic Refinement	32,378.79	9,798.74	3,489.32	1,666.42
Without memetic Refinement	210,828.29	82,481.39	41,436.74	23,746.67

Table 5.4. Geometric mean of the total number of active nodes for all tuning instances for each configuration, with varying number of re-streams indicated by the value of R.

clustering quality. On average, across all tuning instances in the second configuration with $R = 1$, solution quality improves by 3.66%, while achieving a solution quality that is only 0.73% lower than the algorithm with $R = 4$.
Another significant effect of re-streaming on our algorithm is the total number of active nodes found after the last re-streaming phase. As shown in Table 5.4, the more re-streaming operations are performed, the fewer active nodes are found in the final phase. This is due to the continuous movement of nodes in each re-streaming iteration, where each change in a node's cluster assignment directs the node closer to its locally optimal assignment. For the configuration with no memetic refinement (Figure 5.3), the performance profile indicates that a higher number of active nodes corresponds to a higher peak memory usage. However, in the second configuration (Figure 5.4), we recognize that a greater number of active nodes does not significantly affect memory consumption, suggesting that the additional memory usage remains relatively small in absolute terms.

Observation 2. A single round of re-streaming ($R = 1$) offers the best trade-off between runtime, memory consumption, and solution quality. We achieve a 13.49% increase in solution quality compared to no re-streaming iterations. Compared to four re-stream iterations, a single re-stream achieves only 2.65% worse solution quality while delivering 2.33× faster results. With memetic refinement, the difference in solution quality is even lower between all number of re-stream iterations. A single re-stream achieves only 0.73% lower solution quality than four re-streams when memetic refinement is used before re-streaming. Since multiple re-streaming iterations yield only marginal improvements in solution quality, a single re-stream iteration enables a direct transition to the local search refinement phase. This transition further enhances efficiency in both runtime and solution quality by significantly reducing I/O operations and visiting fewer nodes, while still improving clustering quality.

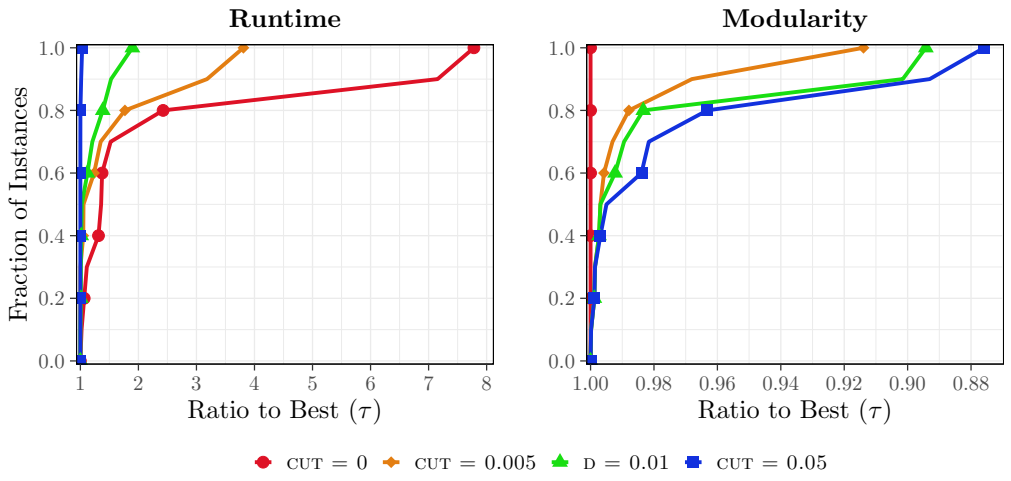


Figure 5.5. Tuning Experiment: Local Search Cut-off Constraint. No Memetic Refinement. Performance profiles for runtime and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with no memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search threshold value ls_{cutoff} varies as indicated in the legend (CUT).

5.4.3 Local Search Limit

After determining the default value for the number of re-streams, we now explore restrictions for the local search refinement phase to provide greater control over our algorithm. To achieve this, we introduce two user-selectable parameters. The first constraint for the local search phase is a cut-off parameter (ls_{cutoff}) that defines the minimum relative improvement in solution quality required in each iteration, relative to the overall computed solution quality. The second constraint is a time fraction ($ls_{\text{frac\_time}}$) parameter, which determines the fraction of the total runtime prior to the local search phase to allocate for the local search phase. This study seeks to find the best configurations for ls_{cutoff} and $ls_{\text{frac\_time}}$ that best balance the runtime-solution quality trade-off in the local search phase.
To conduct these experiments, we extend the two configurations used in the previous tuning study. First, we apply the established baseline value of one re-streaming iteration before adding the local search phase to both configurations. For the threshold parameter, we use the following values: $ls_{\text{cutoff}} \in \{0.05, 0.01, 0.005, 0\}$. Note that $ls_{\text{cutoff}} = 0$ allows the local search refinement to run until convergence, while for the $ls_{\text{frac\_time}}$ parameter we use the values $ls_{\text{frac\_time}} \in \{2, 1, 0.5\}$.
Note that the overall quality of the computed clustering preceding the local search refinement is only an estimate. This is because, when assigning nodes to clusters in the first streaming phase, the modularity gain does not consider the entire cluster volume, but only of the nodes that have been visited. Additionally, when the memetic refinement is used, we cannot incorporate its impact on the modularity score into our overall computed estimate,

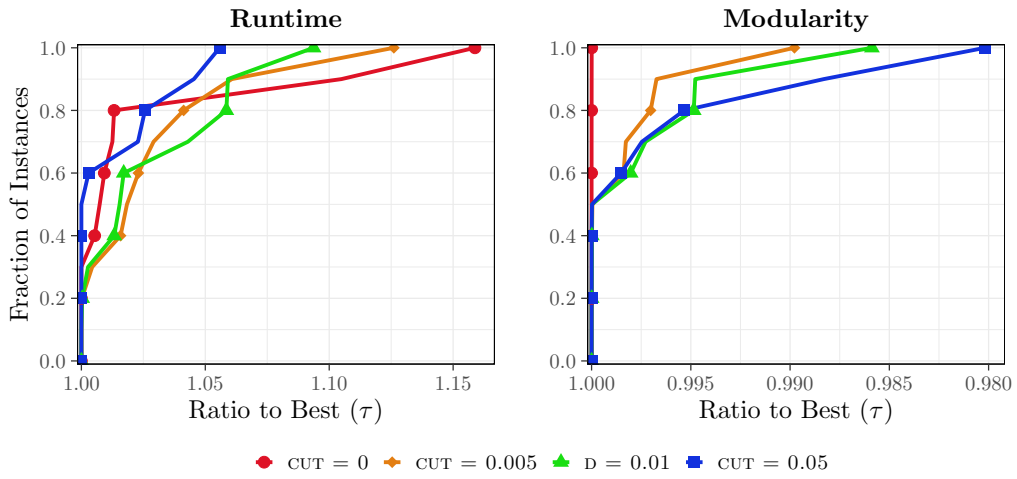


Figure 5.6. Tuning Experiment: Local Search Cut-off Constraint. With Memetic Refinement. Performance profiles for runtime and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search threshold value ls_{cutoff} varies as indicated in the legend (CUT).

since only the cluster assignments are provided as output. This is the main reason we introduced both parameters: to allow for greater flexibility in the algorithm's computation. We add the memory peak performance profiles to the appendix in Figures A.4 A.5 A.6 A.7, since, during the local search phase, only the active node vector changes in size, an effect already examined in the previous study.

We first analyze the performance profiles for the cut-off variable with the configuration excluding memetic refinement, as shown in Figure 5.5. Based on these performance profiles, we infer that reducing the cut-off constraint significantly increases the runtime of our algorithm while only marginally improving solution quality. With $ls_{\text{cutoff}} = 0.05$, solution quality is, on average across all instances, 3.23% lower than the algorithm that runs until convergence ($ls_{\text{cutoff}} = 0$). However, $ls_{\text{cutoff}} = 0$ has a significantly higher runtime, on average, $1.89\times$ longer than $ls_{\text{cutoff}} = 0.05$. The same trend is observed for the constraint values $ls_{\text{cutoff}} = 0.01$ and $ls_{\text{cutoff}} = 0.005$, where a slight improvement in solution quality is achieved compared to $ls_{\text{cutoff}} = 0.05$, but at the cost of a significantly higher runtime. Across all instances, $ls_{\text{cutoff}} = 0.005$ decreases on average solution quality by 1.53% compared to running with no constraints, which is only 1.7% better than $ls_{\text{cutoff}} = 0.05$, but runs $1.44\times$ slower compared to $ls_{\text{cutoff}} = 0.05$. Additionally, across all instances, the algorithm that refines until convergence takes up to more than 7 $\times$ the runtime compared to $ls_{\text{cutoff}} = 0.05$. Furthermore, in Figure 5.6 we observe that the algorithms with memetic refinement before the local search yield only minor improvements in modularity across the different values of ls_{cutoff}, with an average improvement of less than 1% across all instances.

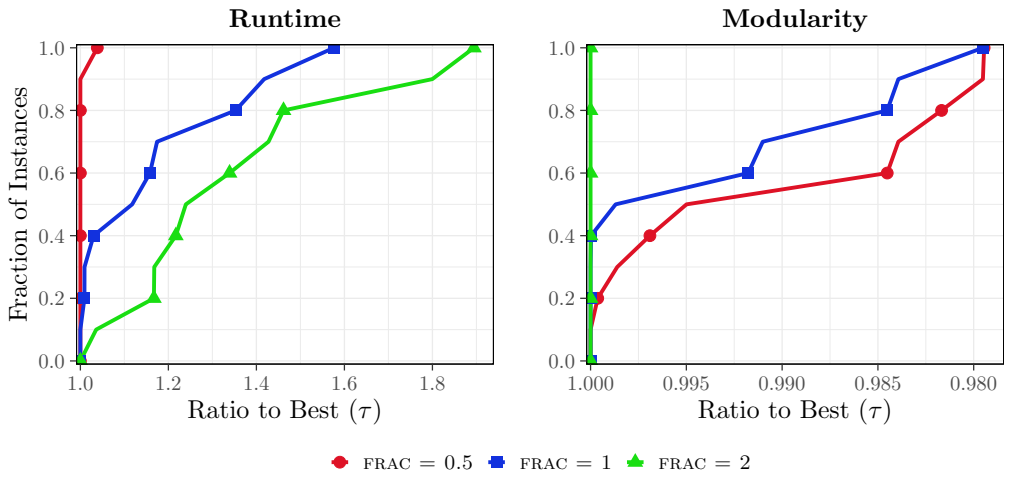


Figure 5.7. Tuning Experiment: Local Search Time Fraction Constraint. No Memetic Refinement. Performance profiles for runtime and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with no memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search time fraction $ls_{\text{frac\_time}}$ varies as indicated in the legend (FRAC).

This is because after the memetic refinement, most nodes are already in their locally optimal assignment. Thus, fewer nodes move during local search, as shown in Table 5.4, which compares active node counts between memetic and non-memetic configurations.

Figure 5.7 shows a similar expected behavior when evaluating the time constraint $ls_{\text{frac\_time}}$, without memetic refinement before the local search. As expected, increasing $ls_{\text{frac\_time}}$ increases runtime; however, the improvement in solution quality remains minimal. When analyzing $ls_{\text{frac\_time}} = 0.5$, we find that, on average across all instances, the solution quality is 1.01% lower while achieving up to $1.35\times$ faster results compared to the best-performing algorithm in terms of modularity score, being the algorithm with $ls_{\text{frac\_time}} = 2$. Furthermore, this solution quality gap is even smaller when using memetic refinement before local search, as seen in Figure 5.8, for the reasons mentioned earlier. Across all instances, the algorithm with $ls_{\text{frac\_time}} = 0.5$ decreases the modularity score by less than 1%. The algorithm with $ls_{\text{frac\_time}} = 1$ yields better results than $ls_{\text{frac\_time}} = 0.5$ but, as expected, is significantly slower, on average across all instances, it is approximately $1.17\times$ slower.
In the local search phase, we provide the option of specifying a value $ls_{\text{time\_limit}}$, which sets an upper bound on the total time, in seconds, to be spent in the local search refinement phase. $ls_{\text{time\_limit}}$ is particularly useful for large graphs, where each iteration takes a significant amount of time, and the modularity improvement per iteration is marginal, thus providing additional control over the algorithm's behavior.
In conclusion, the two tuning parameters emphasize different performance metrics. If solution quality is a priority, tuning the ls_{cutoff} parameter allows direct control over the im-

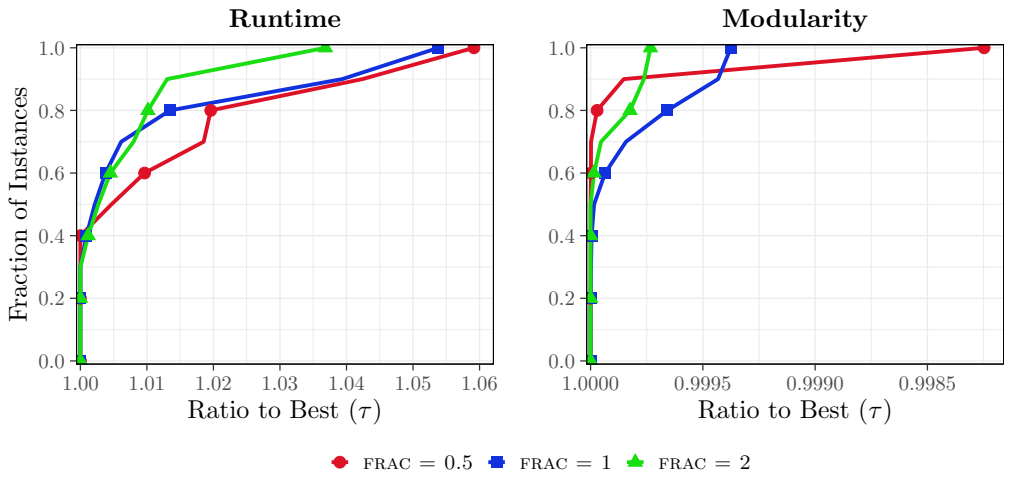


Figure 5.8. Tuning Experiment: Local Search Time Fraction Constraint. With Memetic Refinement. Performance profiles for runtime and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search time fraction $ls_{\text{frac\_time}}$ varies as indicated in the legend (FRAC).

provements achieved by the local search algorithm, regardless of the number of iterations or the time required. On the other hand, if runtime is of high concern, tuning $ls_{\text{time\_limit}}$ directly affects the runtime of the local search phase, independently of the current modularity improvement. Since this thesis primarily emphasizes solution quality, our baseline algorithm incorporates the ls_{cutoff} parameter and not the $ls_{\text{frac\_time}}$ parameter.

Observation 3. The constraint ls_{cutoff} specifies the minimum relative modularity improvement required per local search iteration, while $ls_{\text{frac\_time}}$ is the fraction of total pre-local search runtime to allocate for the local search. The configurations that best balance the runtime-solution quality trade-off in the local search are $ls_{\text{cutoff}} = 0.05$ and $ls_{\text{frac\_time}} = 0.5$. $ls_{\text{cutoff}} = 0.05$ results in only a 3.23% decrease in solution quality on average while running $1.89\times$ faster than no constraint configuration. Compared to $ls_{\text{cutoff}} = 0.005$, $ls_{\text{cutoff}} = 0.05$ yields only 1.7% lower solution quality while running $1.44\times$ faster. On average, $ls_{\text{frac\_time}} = 0.5$ decreases solution quality only by 1.01% while delivering $1.35\times$ faster results compared to no restrictions. With memetic refinement before local search, the solution quality gap is even smaller between different constraint values, remaining less than 1% compared to the best performing configuration in terms of solution quality.

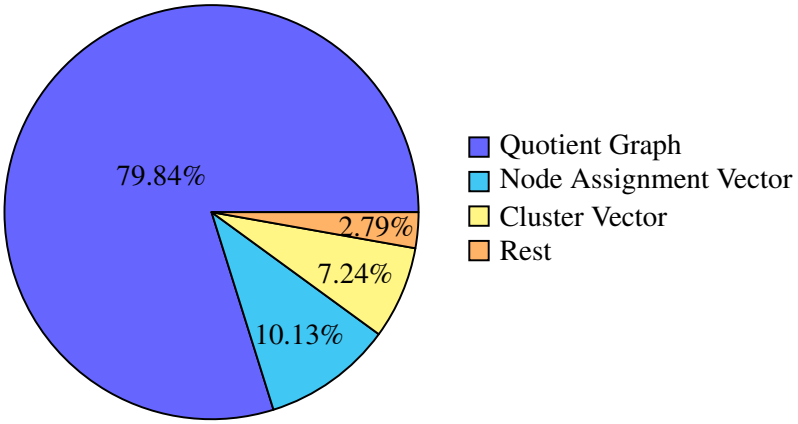


Figure 5.9. Memory distribution during quotient graph construction for the memetic refinement phase. The pie chart is created using the geometric mean over all tuning graphs, where the algorithms ran using the baseline CLUSTRE configuration with only the streaming and memetic phase.

5.4.4 Maximum Number of Clusters

Finally, we assess the last tuning parameter: the total number of initialized clusters. As seen in Figure 5.9, the memory consumption bottleneck of CLUSTRE lies in the size of the quotient graph. Thus, we evaluate techniques to mitigate this bottleneck. By reducing the quotient graph size we also significantly reduce the memory consumption of the memetic refinement phase, as each instance in the population then requires less memory. One approach is to set an upper limit to the total number of clusters initialized, which reduces the size of the quotient graph and, in turn, saves memory. However, imposing an upper limit may result in suboptimal cluster assignments. This study evaluates the trade-off between solution quality and peak memory usage when applying an upper-bound variable. To execute these experiments, we use the tuning parameter $cluster\_frac \in \{1, 0.1, 0.05, 0.01, 0.005\}$. The $cluster\_frac$ value corresponds to the total number of clusters that can be initialized relative to the total number of nodes. $cluster\_frac = 0.01$ implies that at most 1% of the total number of nodes can be initialized as clusters. The value $cluster\_frac = 1$ indicates that no upper limit is set, meaning each node can become its own cluster. Note that the quotient graph is only constructed when the memetic refinement is used. Hence, the algorithm configuration chosen for this study consists of the required streaming phase, followed by a memetic phase using the previously determined baseline duration of 15 seconds. Limiting the total number of clusters does not impact runtime, as all previously visited neighbors of a streamed node are still evaluated. The only difference is that when the upper limit is reached, the node is assigned to the neighboring cluster that yields the highest gain, even if that gain is negative. Since all algorithms receive the same estimated amount of time in the memetic phase, the runtime performance profile is presented in the Appendix A.8.

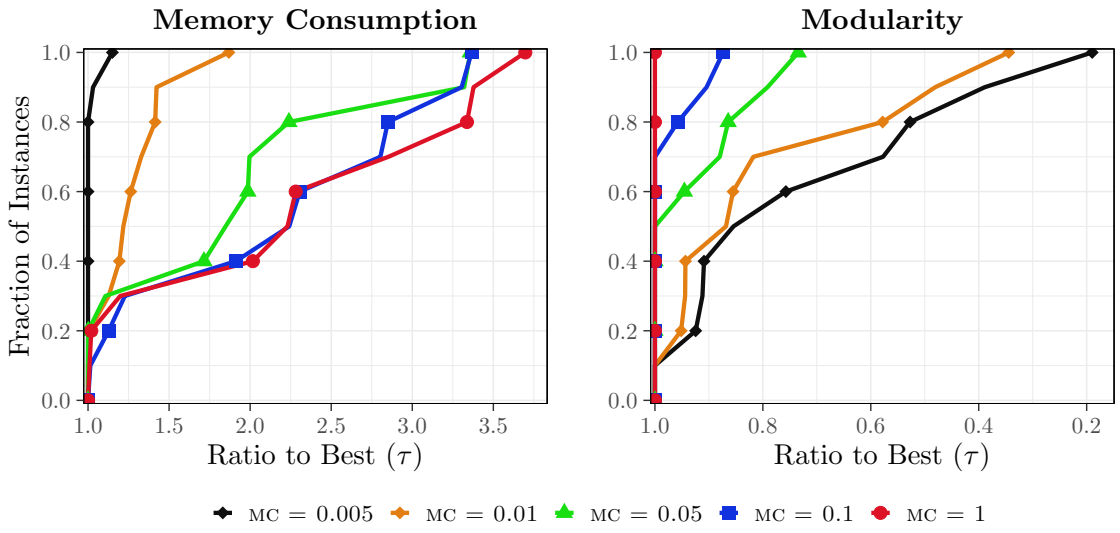


Figure 5.10. Tuning Experiment: Maximum Number of Clusters. Performance profiles for memory consumption and modularity score. For all algorithms, we use the baseline CLUSTRE configuration with varying values for the maximum cluster fraction parameter (MC). Each algorithm underwent a streaming phase followed by a 15-second memetic phase.

Figure 5.10 presents the results, confirming the expected trend: a lower $cluster\_frac$ reduces memory usage but significantly decreases modularity, by up to 80% compared to the best solution found. The algorithm with the lowest $cluster\_frac = 0.005$ yields on average, the lowest memory peak, requiring only 80.50% of the memory compared to $cluster\_frac = 0.01$, and just 48.94% compared to the algorithm with $cluster\_frac = 1$. However, this algorithm performs the worst overall in terms of solution quality. For all instances, the algorithm with the lowest $cluster\_frac$ produces, on average, a solution quality 36.22% worse than the unrestricted algorithm, rendering this parameter value infeasible due to its significantly lower modularity performance. The same applies to the second-lowest $cluster\_frac = 0.01$, where the algorithm delivers slightly better solution quality at the expense of slightly higher memory consumption. However, the improvement in solution quality remains negligible. Therefore, $cluster\_frac = 0.01$ is equally infeasible due to its low modularity score. The highest two values of $cluster\_frac$ result in similar memory consumption. In 60% of the instances, the peak memory usage is up to 2.25× higher for both algorithms. A similar pattern emerges on average across all instances: the algorithm with $cluster\_frac = 1$ consumes approximately 3.7× the memory peak of the best result found, while the algorithm with $cluster\_frac = 0.1$ consumes about 3.35× the memory peak. Across all instances, the unrestricted algorithm achieves the best solution quality, while the algorithm with $cluster\_frac = 0.1$ results in up to an 11% lower solution quality.

Note that the algorithm allows an absolute value $MaxCluster$ to be specified to set

Parameter	Symbol	Baseline Value
Memetic Refinement Duration	D	15 seconds
Number of Re-streams	R	1 re-streaming iteration
Local Search Cutoff	ls_{cutoff}	0.05
Local Search Time Constraint	$ls_{\text{frac\_time}}$	0.5
Max Clusters	$cluster\_frac$	no cluster upper-limit

Table 5.5. Baseline Configuration of CLUSTRE

an upper bound on the total number of initialized clusters. This feature is particularly useful in scenarios where only a limited amount of memory is available. Setting $MaxCluster$ ensures that the algorithm's memory usage does not exceed a predefined limit, while allowing smaller instances, where this limit is never reached, to use the optimal local assignment in the streaming phase.

Observation 4. $cluster\_frac$ defines the total number of clusters that can be initialized relative to the number of nodes in the graph. With $cluster\_frac = 0.005$ and $cluster\_frac = 0.01$, memory consumption is minimized to 48.94% and 60.80% of the unrestricted value, respectively. However, they significantly decrease solution quality by approximately 36.22% and 26.00% respectively, making them infeasible. The unrestricted value, $cluster\_frac = 1$, offers the best balance between memory consumption and solution quality. It achieves the best results while consuming only slightly more memory than the other feasible configurations, $cluster\_frac \in \{0.1, 0.05\}$, requiring just $1.02\times$ and $1.16\times$ more memory, respectively.

5.5 CluStRE Performance Evaluation

Now that we have established the baseline configuration for our CLUSTRE algorithm, summarized in Table 5.5, we now aim to answer our first research question, which examines the impact of multi-stage refinement, including memetic algorithms, re-streaming, and local search, on solution quality, peak memory consumption, and runtime.
To evaluate this question, we introduce four selectable modes in our algorithm: STRONG, EVO, LIGHT+, and LIGHT, also listed in Table 5.6. The LIGHT mode consists only of the streaming phase, without any multi-stage refinement. The LIGHT+ mode builds upon the LIGHT mode, adding a re-streaming and local search phase. The EVO mode, similar to the LIGHT+ mode, extends the LIGHT mode, but instead of re-streaming and local search refinement, the algorithm undergoes the memetic refinement phase after the streaming phase. Finally, the STRONG mode integrates all multi-stage refinements, starting with memetic refinement, followed by re-streaming, and concluding with a local search phase.
We conduct this analysis using the Test dataset (Table 5.2) and the baseline configurations

Mode	Initial Streaming	Evolutionary Refinement	Re-Streaming and Local Search
LIGHT	✓	-	-
LIGHT+	✓	-	✓
EVO	✓	✓	-
STRONG	✓	✓	✓

Table 5.6. Outline of the different configurations of CLUSTRE.

for all refinement phases determined in the tuning study, summarized in Table 5.5. We apply the cut-off constraint parameter in the local search phase, to ensure we continue making improvements if the rounds result in a significant increase in modularity. Additionally, we set $ls_{\text{time\_limit}}$ to ten minutes to prevent our algorithm from running for an excessive amount of time. Furthermore, we set the $MaxCluster$ to five million clusters, ensuring that all experiments can run on our machine.
Based on the tuning experiments, we expect the CLUSTRE-LIGHT mode to be the fastest algorithm with the lowest peak memory consumption but with the lowest solution quality. This is because it only applies the streaming phase, without constructing the quotient graph, identified as the memory bottleneck in Figure 5.9, or utilizing additional refinement phases to improve solution quality. We expect the CLUSTRE-LIGHT+ and CLUSTRE-EVO modes to outperform the CLUSTRE-LIGHT mode, but at the cost of additional memory and runtime, since both modes go through an additional refinement phase after the streaming phase, where either the quotient graph is constructed or the active node vector is initialized. Finally, the CLUSTRE-STRONG mode is expected to yield the best solution quality, but at the expense of significantly higher memory consumption and runtime, as it incorporates all refinement phases of the multi-stage algorithm.
The comparative analysis of the four modes in Figure 5.11 highlights their unique performance and potential applications, considering computational limitations and solution quality requirements. As expected, the CLUSTRE-LIGHT mode is the fastest and has the lowest memory peak across all instances, but at the same time, it also yields the lowest modularity score. Next, we observe that the re-streaming and local search refinements improve solution quality at the cost of slightly higher memory consumption and runtime. The CLUSTRE-LIGHT+ mode improves solution quality by 14.93% over CLUSTRE-LIGHT, while running 2.22× slower and consuming 1.74× more memory on average across all instances. As seen in the performance of the CLUSTRE-EVO mode, the memetic refinement phase alone also improves solution quality. On average, the CLUSTRE-EVO mode yields 19.87% better solution quality, though it has a 4.70× higher memory consumption and runs 9.97× slower than the CLUSTRE-LIGHT mode. As expected, the CLUSTRE-STRONG mode achieves the best quality among the four modes but also results in the highest memory consumption and the slowest runtime. CLUSTRE-STRONG Improves solution quality by 24.85% while running 10.35× slower and using 4.94× more memory on average across all instances. At first, it may appear that the CLUSTRE-STRONG mode requires

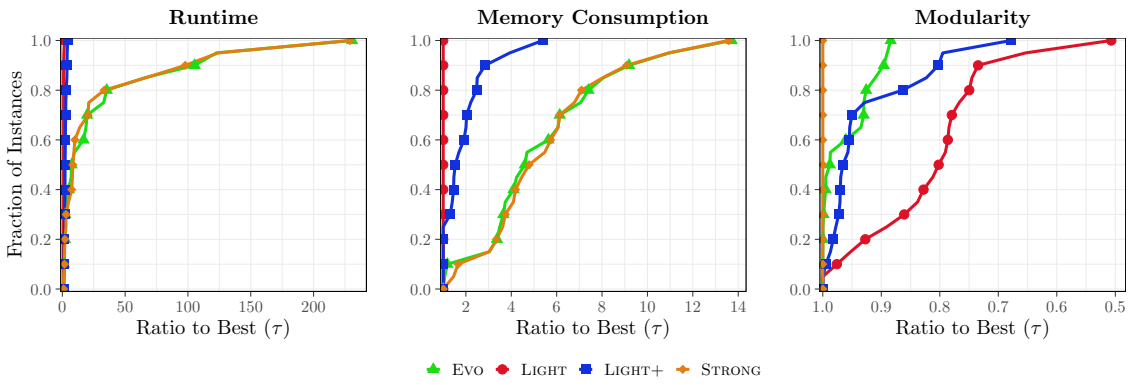


Figure 5.11. CLUSTRE Performance Evaluation: Performance profiles for runtime, peak memory consumption, and modularity. For all algorithms, we use the baseline CLUSTRE configuration, with different modes indicated in the legend.

large amounts of memory; however, when considering absolute values, the memory consumption of the CLUSTRE-STRONG mode remains reasonable. For our largest instance, `uk-2007-05`, with 105 million nodes and 3.3 billion edges, CLUSTRE-STRONG requires only 2.26GB of peak memory, making it manageable for almost all computational devices.

Observation 5. The different modes of CLUSTRE (Table 5.6) emphasize different criteria and can be used for different applications. CLUSTRE-LIGHT is the lightest and most memory-efficient but yields the lowest quality. CLUSTRE-STRONG achieves the highest quality, improving it by 24.85% but requires the longest runtime and has the highest memory consumption. CLUSTRE-LIGHT+ strikes a balance between memory consumption, runtime, and solution quality, improving solution quality by 14.93% over CLUSTRE-LIGHT while running 2.22× slower and consuming 1.74× more memory on average. To answer **RQ1**, we can clearly state that the re-streaming and memetic refinement stages significantly enhance the solution quality of the clustering. However, this comes at the expected cost of higher memory consumption and a longer runtime.

5.6 Comparison against State-of-the-Art

We now provide experiments to evaluate the performance of CLUSTRE against current state-of-the-art clustering algorithms. For our CLUSTRE algorithm, we test all different modes with the baseline configuration determined in the tuning study, with their values listed in Table 5.5. We first compare CLUSTRE to the current state-of-the-art streaming graph clustering algorithm by Hollocou et al. [25]. For this algorithm, we apply the best configuration we found for $v_{\max}$ determined in the Appendix A.2, which is set to a rel-

ative value of 1% of the total edges for each graph. We are also interested in how our solution quality results compare to the current state-of-the-art in-memory graph clustering algorithms. Specifically, we compare CLUSTRE to the well-known LOUVAIN algorithm introduced by Blondel et al. [5] and to the state-of-the-art in-memory graph clustering algorithm VIECLUS, which builds on the Louvain algorithm and uses a memetic approach introduced by Biedermann et al. [4]. To ensure a fair runtime comparison, we allocate five minutes for the VIECLUS evolutionary algorithm.

Goal. The main goal of these experiments is to tackle the two research questions we posed at the beginning: whether CLUSTRE can bridge the solution quality gap when using streaming algorithms compared to in-memory algorithms, as stated in **RQ2**. Furthermore, with the different modes introduced by CLUSTRE, we investigate in **RQ3** how well CLUSTRE balances runtime, peak memory consumption, and solution quality compared to other state-of-the-art in-memory and streaming graph clustering algorithms.

Expectations. Before analyzing our experimental results, we expect in-memory algorithms to yield the highest solution quality, as they have access to more global graph information than streaming algorithms when making decisions. In particular, we expect the VIECLUS algorithm to outperform the LOUVAIN algorithm, due to its evolutionary nature, which builds upon the LOUVAIN algorithm and explores a significantly larger solution space. When examining the streaming graph clustering algorithms, we expect all modes undergoing any refinement stage to yield higher solution quality than the HOLLOCOU algorithm and the CLUSTRE-LIGHT mode, as previously observed in Section 5.5. A key question then arises as to whether the CLUSTRE-LIGHT algorithm outperforms the HOLLOCOU algorithm, as both operate solely in a streaming fashion. In terms of memory and runtime comparison, we expect the in-memory graph clustering algorithms to require significantly more memory, as the entire graph must be stored in-memory. For all the streaming algorithms, we expect significantly lower memory consumption. However, for the modes that require the construction of the quotient graph, we anticipate higher memory consumption than modes that do not. We expect a similar behavior regarding runtime. As the evolutionary algorithm undergoes multiple rounds, we anticipate VIECLUS to require the longest runtime. Additionally, we expect CLUSTRE-EVO, CLUSTRE-STRONG, and LOUVAIN to have significantly longer runtimes, due to the memetic refinement phase in the former two modes and the multiple rounds conducted by the LOUVAIN algorithm. We expect the HOLLOCOU algorithm and the CLUSTRE-LIGHT mode to be the fastest algorithms, as they are both purely streaming-based and do not incorporate any refinement or additional processing phases.

Solution Quality Evaluation. The solution quality results of the experiments are depicted in Table 5.7. From Table 5.7, we observe that all CLUSTRE modes outperform the current state-of-the-art streaming algorithm in terms of solution quality based on the

Graph	Light	Light+	Evo	Strong	Hollocou	Louvain	VieClus
arabic-2005	0.7407	0.9541	0.9850	**0.9880**	0.6579	0.9897	0.9899
citationCiteseer	0.4130	0.5525	0.7828	**0.8148**	0.2647	0.8184	0.8247
com-amazon	0.7637	0.7967	0.9106	**0.9226**	0.6238	0.9316	0.9344
com-friendster	0.5229	0.5838	0.5459	**0.5870**	0.0879	-	-
enwiki-2013	0.5121	0.6243	0.5719	**0.6385**	0.0901	0.6611	0.6627
eu-2005	0.7364	0.8894	0.9204	**0.9316**	0.4223	0.9402	0.9408
great-britain	0.9248	0.9262	0.9972	**0.9973**	0.9344	0.9976	0.9977
hollywood-2011	0.5352	0.6924	0.6777	**0.7286**	0.2234	0.7531	0.7558
it-2004	0.7433	0.9403	0.9646	**0.9693**	0.6699	0.9763	0.9762
italy	0.9500	0.9517	0.9976	**0.9976**	0.9554	0.9980	0.9981
libimseti	0.3310	0.3829	0.3657	**0.3949**	0.0304	0.3945	0.3974
Penn94	0.3181	0.4006	0.4304	**0.4869**	0.0257	0.4896	0.4986
rgg_n26	0.9671	0.9733	0.9897	**0.9909**	0.9513	0.9956	0.9956
rhg2b	0.9920	0.9920	0.9921	**0.9921**	0.9446	-	-
roadNet-CA	0.7802	0.7962	0.9919	**0.9922**	0.8388	0.9929	0.9935
roadNet-PA	0.7712	0.7864	0.9890	**0.9894**	0.8339	0.9901	0.9910
sk-2005	0.7237	0.9428	0.9664	**0.9714**	0.5015	-	-
uk-2007-05	0.7135	0.8675	0.8216	**0.8789**	0.2315	-	-
webbase-2001	0.6341	0.7865	0.7195	**0.8085**	0.7005	0.9824	0.9824
wiki-Talk	0.3156	0.3509	0.3338	**0.3667**	0.1068	0.3601	0.3845
GeoMean	0.6313	0.7256	0.7567	**0.7882**	0.3280	N/A	N/A

Table 5.7. Test Experiment: Modularity Scores Comparison. Modularity scores achieved by our proposed algorithms compared to competing in-memory and streaming clustering approaches across the test instances (Table 5.2). HOLLOCOU serves as the streaming competitor, while LOUVAIN and VIECLUS represent state-of-the-art in-memory clustering algorithms. Missing instances ("−") indicate failed runs due to exceeding the available amount of memory on the machine.

geometric mean. On average, the CLUSTRE-LIGHT mode achieves 92.50% higher solution quality than the HOLLOCOU algorithm across all test instances, while the CLUSTRE-STRONG algorithm improves solution quality by 140.33%, thus setting a new benchmark for streaming graph clustering algorithms. Furthermore, we observe that for the `Penn94` instance, the CLUSTRE-LIGHT mode achieves $12.37\times$ higher solution quality than the HOLLOCOU algorithm, while CLUSTRE-STRONG attains $18.95\times$ the solution quality. One prominent reason for the low solution quality of the HOLLOCOU algorithm is that its performance strongly depends on the ordering of the streamed edges. This is because it relies on the assumption that edges arriving early are more likely to be intra-cluster edges than inter-cluster edges and therefore clusters incident nodes together early on. However, this condition cannot be guaranteed. Moreover, in their official implementation, they first

stored all the edges in-memory, randomized their order, and then processed them to further strengthen their assumption. However, we modified the HOLLOCOU algorithm to function as a proper streaming algorithm, which reads the graph from disk, meaning that the order in which the edges appear depends solely on the order provided by the graph. Hence, we ensure that every algorithm reads the graph in the same order, allowing for a fair comparison. Notably, even with the original implementation, where all edges are stored in-memory and randomized before processing, the solution quality of the HOLLOCOU algorithm does not significantly improve but requires approximately $7.32\times$ more memory on average across all instances compared to the modified streaming version. Additionally, the HOLLOCOU algorithm relies heavily on the $v_{\max}$ parameter, which we attempted to set as best as possible during the tuning study in Appendix A.2. Nevertheless, since the authors do not provide any guidance on how to select this parameter, it is likely that the parameter does not suit all instances equally well, as observed with the `Penn94` instance.
Analyzing the solution quality results confirms our expectations: the in-memory clustering algorithms deliver the highest quality for every single instance, while the VIECLUS algorithm provides slightly higher results than LOUVAIN due to its evolutionary nature. When comparing the different modes of our CLUSTRE algorithms with the state-of-the-art in-memory clustering algorithms, we observe a significant leap in solution quality. On average across all instances, our lightest and fastest mode, CLUSTRE-LIGHT, attains 77.36% of LOUVAIN's solution quality and 76.83% of VIECLUS' solution quality. We observe that with each additional refinement phase, the solution quality approaches that of the in-memory algorithms. Specifically, with CLUSTRE-LIGHT+, we achieve 88.83% of LOUVAIN's and 88.22% of VIECLUS' solution quality, and with CLUSTRE-EVO, we achieve 94.19% of LOUVAIN's and 93.55% of VIECLUS' solution quality. In our best-performing mode, which incorporates all refinement stages, CLUSTRE-STRONG reaches 98.21% of LOUVAIN's solution quality and 97.54% of VIECLUS' solution quality on average across all instances where the VIECLUS and LOUVAIN algorithms did not fail. As displayed in Figure 5.7, the instances `com-friendster`, `rhg2b`, `sk-2005`, and `uk-2007-05`, failed for the in-memory algorithms VIECLUS and LOUVAIN. This occurred because their memory consumption exceeded the available memory on our 93 GB machine, further highlighting the necessity of a memory-efficient algorithm. These results clearly demonstrate that the multi-stage refinements of CLUSTRE are highly effective at leveraging partial global information through the quotient graph and the local search phase.

Observation 6. CLUSTRE outperforms HOLLOCOU in all configurations, improving solution quality by an average of 92.50% with CLUSTRE-LIGHT and 140.33% with CLUSTRE-STRONG. To answer **RQ2**, we conclude that CLUSTRE effectively bridges the solution quality gap between streaming and in-memory clustering algorithms. CLUSTRE-STRONG establishes a new state-of-the-art standard in modularity optimization, achieving over 97% of the solution quality of the state-of-the-art in-memory clustering algorithms, such as LOUVAIN and VIECLUS.

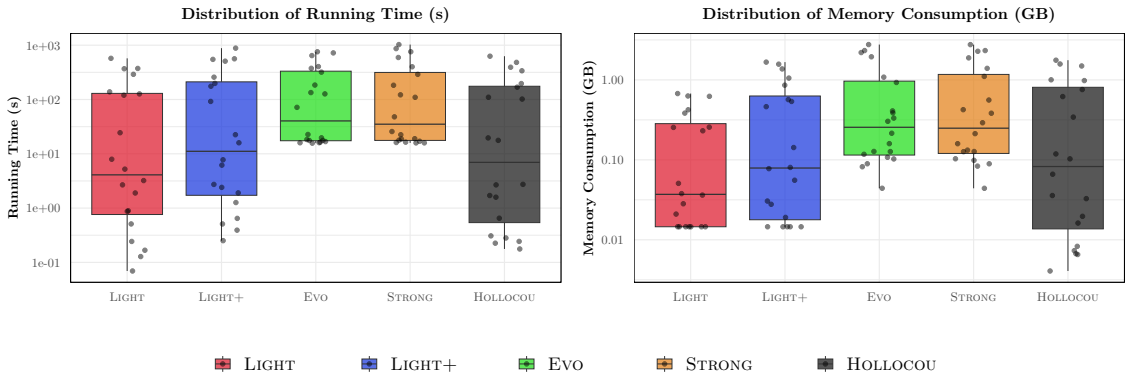


Figure 5.12. Comparison Against State-of-the-Art: Runtime and Peak Memory Consumption of CLUSTRE Modes vs. HOLLOCOU. Box plots illustrate the runtime distribution (left) and the peak memory consumption (right) across all test instances. Note the logarithmic scale on the y-axis.

Runtime and Memory Consumption Evaluation. To answer **RQ3**, we assess the computational efficiency of our CLUSTRE algorithm compared to state-of-the-art in-memory and streaming algorithms. Figure 5.12 and Figure 5.13 illustrate the distribution of runtime and peak memory consumption as box plots for the different algorithms. Figure 5.12 benchmarks streaming algorithms across all instances, whereas Figure 5.13 illustrates the distribution across all algorithms, excluding instances where VIECLUS and LOUVAIN failed due to the 93 GB memory constraint imposed by the machine. As expected, the CLUSTRE-LIGHT mode is the fastest and most memory-efficient among all streaming algorithms, even compared to the HOLLOCOU algorithm, with both the median runtime and peak memory usage being lower. One possible reason is that HOLLOCOU allocates a vector of size n at the start, solely to store the community volumes of singleton clusters. This step is omitted in our CLUSTRE algorithm, as the community volume of a singleton cluster corresponds to the degree of the node. Therefore, CLUSTRE does not directly allocate memory in the cluster volumes vector for every streamed node, as we already know its community volume. Thus, in cases where assigning the streamed node to an existing cluster results in higher modularity gain, we save one allocation in the cluster volume vector since we only need to update the existing cluster volume. Furthermore, Our approach is faster due to our streaming technique: only edges connected to already streamed nodes are considered in our streaming algorithms, resulting in many edges being left out of the initial streaming process. On average, across all instances, our CLUSTRE-LIGHT mode runs $1.36\times$ faster while requiring only 72.76% of the memory consumption compared to HOLLOCOU. As a result, our CLUSTRE-LIGHT algorithm outperforms the current state-of-the-art streaming approach across all key metrics, namely solution quality, peak memory consumption, and runtime. When comparing HOLLOCOU against the modes that use refinements, we achieve, higher modularity score at the cost of increased

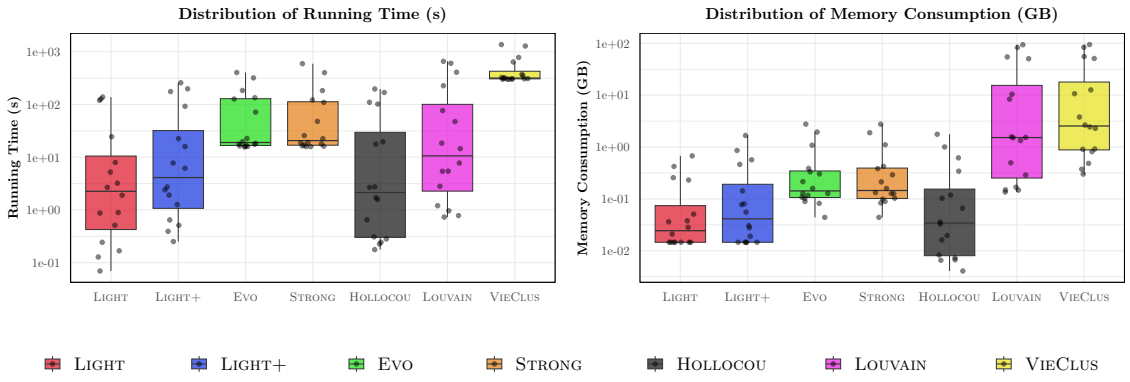


Figure 5.13. Comparison Against State-of-the-Art: Runtime and Peak Memory Consumption of CLUSTRE Modes vs. HOLLOCOU, LOUVAIN, and VIECLUS. Box plots illustrate the runtime distribution (left) and the memory consumption (right) across all test instances. VIECLUS is given five minutes. Instances where the LOUVAIN and VIECLUS algorithm fail ("−") are excluded from the plots.

computational resources. Our strongest mode, CLUSTRE-STRONG, improves the solution quality of HOLLOCOU by an average of 140.33% across all instances, as seen in Table 5.7, while requiring nearly $7.59\times$ more runtime and $3.59\times$ more memory. CLUSTRE-LIGHT+ offers a balanced alternative between solution quality and computational efficiency. On average across all instances, CLUSTRE-LIGHT+ improves solution quality by 121.25% while requiring only $1.63\times$ longer and $1.27\times$ more memory.

Figure 5.13 confirms our expectations that in-memory algorithms require significantly more memory than all the streaming graph clustering algorithms. Furthermore, we observe that for CLUSTRE configurations where the quotient graph was constructed, specifically, the CLUSTRE-EVO and CLUSTRE-STRONG modes, the memory distribution is nearly identical. As shown in Figure 5.9, constructing the quotient graph accounts for nearly 80% of the peak memory consumption. However, all CLUSTRE modes consume significantly less memory than the LOUVAIN and VIECLUS algorithms. Our strongest mode, CLUSTRE-STRONG, requires, on average, for all instances where VIECLUS and LOUVAIN did not fail, only 10% of LOUVAIN's and 5.64% of VIECLUS' memory consumption, while attaining over 97% of the solution quality, as shown in Table 5.7. By excluding refinement stages, the median runtime and memory distribution significantly decrease, especially for the modes where the quotient graph is not constructed, such as CLUSTRE-LIGHT+ and CLUSTRE-LIGHT. Again, we observe that CLUSTRE-LIGHT+ optimizes computational costs while retaining most of the solution quality benefits. On average across all feasible instances, CLUSTRE-LIGHT+ attains 88.83% of LOUVAIN's and 88.22% of VIECLUS' solution quality, while requiring only 2.93% of LOUVAIN's and 1.65% of VIECLUS' peak memory consumption. Additionally, it runs $2.48\times$ faster than LOUVAIN and an impressive $65.84\times$ faster than VIECLUS.

Observation 7. To address **RQ3**, we conclude that CLUSTRE effectively balances runtime, memory consumption, and solution quality by providing multiple modes tailored for specific applications. CLUSTRE-LIGHT runs $1.36\times$ faster and consumes only 72.76% of the memory consumption of HOLLOCOU while achieving 92.50% higher solution quality. Furthermore, CLUSTRE-STRONG attains 97% of the solution quality of VIECLUS while requiring only 5.64% of the memory consumption and running $9.85\times$ faster than VIECLUS. The CLUSTRE-LIGHT+ mode offers a balance between the two modes mentioned above while still achieving more than double the solution quality compared to HOLLOCOU.

5.6.1 Ground-Truth Communities Performance

Finally we investigate **RQ4**, which questions the ability of CLUSTRE to find ground-truth communities compared to the current state-of-the-art streaming clustering algorithm HOLLOCOU. To evaluate our ability to uncover ground-truth communities we use the ground-truth dataset and the Normalized Mutual Information (NMI) as the comparison metric [30]. NMI quantifies the similarity between two clusterings by measuring the mutual information they share, normalized by their individual uncertainties, irrespective of label permutations. For this experiment we apply the baseline configuration for all CLUSTRE algorithms and use the determined best value for the $v_{\max}$ parameter being the relative value of 1% of the total edges for HOLLOCOU.

Before analyzing the results, we speculate CLUSTRE-STRONG to perform best and yield the highest NMI score, while the HOLLOCOU algorithm to yield lower scores. This is because, as observed in Section 5.6, CLUSTRE-STRONG yields the highest modularity score, while the HOLLOCOU algorithm delivers the lowest. Note that the modularity objective function assesses clusterings based on intra-cluster density and inter-cluster sparsity. Therefore, we can assume that for a clustering of high modularity, the clusters are likely well-defined and reflect real groupings in the graph, thereby increasing the likelihood of these clustering to match the true labels, which, in turn, could result in a higher NMI score.

Table 5.8 presents the results of our experiment. First of all, we observe that our speculations are confirmed. The NMI score of our CLUSTRE-LIGHT mode is higher than the HOLLOCOU algorithm for every single instance and yields a 20.04% better NMI score on average across all instances. Furthermore, we observe an increase in NMI with each additional refinement stage. With LIGHT+ and EVO modes, we improve the ground-truth recovery ability by approximately 26.42% and 34.76% on average across all instances, respectively. This highlights the significant impact of the memetic refinement stage alone. By incorporating all multi-stage refinements, with CLUSTRE-STRONG, we attain the highest NMI score out of all algorithms. In this case, the ground-truth recovery ability of CLUSTRE-STRONG is about 39.65% higher than that of HOLLOCOU.

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**	0.5054	0.36
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**	0.1193	0.3404
PubMed	0.6388	0.1658	0.6775	0.1692	0.7301	0.1871	**0.7552**	**0.1917**	0.3258	0.1518
AmazonPH	0.5448	0.4698	0.6547	0.5291	0.7122	0.6188	**0.7318**	**0.6452**	0.2725	0.3174
GEOMEAN	0.6444	0.3399	0.6965	0.3538	0.7334	0.3763	**0.7542**	**0.3885**	0.3212	0.2883

Table 5.8. Test Experiment: Ground-Truth Comparisons. This experiment evaluates modularity and Normalized Mutual Information (NMI) scores by comparing them against ground-truth community structures in various benchmark graphs. Higher scores indicate a stronger alignment with the known community structures.

Observation 8. To answer **RQ4**, we confidently conclude that CLUSTRE improves the ability to recover ground-truth communities. Starting with CLUSTRE-LIGHT, we outperform the HOLLOCOU algorithm by 20.04%, whereas CLUSTRE-STRONG further enhances this ability, increasing the NMI score by 39.65%, demonstrating the effectiveness of multi-stage refinement in ground-truth retrieval.

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we propose CLUSTRE, a new graph **Clu**stering algorithm that uses a **St**reaming setting with multi-stage refinement, incorporating **R**e-streaming and **E**volutionary heuristics. CLUSTRE is a node-streaming algorithm that dynamically constructs a quotient graph data structure that portrays key structural properties and interactions between clusters in the original graph. The quotient graph serves as input to a state-of-the-art in-memory evolutionary graph clustering algorithm, VIECLUS, to further optimize clustering quality. Furthermore, our algorithm includes a re-streaming and local search stage, which also optimizes the solution quality by leveraging partial global information. Overall, CLUSTRE bridges the solution quality gap between in-memory and streaming graph clustering algorithms by achieving approximately 97% of the in-memory solution quality, while only requiring about 5% of the memory consumption and operating 10 $\times$ faster compared to state-of-the-art in-memory clustering algorithms LOUVAIN and VIECLUS. The experimental evaluation highlights the superiority of CLUSTRE compared to other state-of-the-art streaming graph clustering algorithms, such as HOLLOCOU, in all key metrics, including solution quality, memory consumption, runtime, and ground-truth community retrieval. Even our lightest mode, CLUSTRE-LIGHT, significantly outperforms HOLLOCOU, improving solution quality by 92.50%, while requiring only 72.76% of the total memory and running 1.36 $\times$ faster on average for all instances. Additionally, our strongest mode, CLUSTRE-STRONG, improves solution quality by about 140%, setting a new benchmark for streaming graph clustering algorithms. Most importantly, our CLUSTRE algorithm improves the retrieval of ground-truth communities compared to HOLLOCOU by up to 39.65%, making it the current best-performing streaming algorithm for accurately identifying ground-truth communities. These results underline the versatility of CLUSTRE, positioning it as a promising tool for high-quality clustering computations, even in resource-constrained settings.

6.2 Future Work

This work opens up several avenues for future research to further improve CLUSTRE with respect to the key metrics mentioned above. First of all, as previously seen in Figure 5.9, the quotient graph is the primary memory consumption bottleneck. One potential approach to improve CLUSTRE's peak memory efficiency is to explore alternative dynamic and efficient data structures for storing the quotient graph representation. A promising direction is to explore compression algorithms, such as LIGRA+ [57]. This concept can also be extended to VIECLUS in the memetic refinement phase. For the memetic refinement algorithm, only a static compression algorithm is required, as the graph for each generated instance remains unchanged; we simply create and discard instances. Applying compression algorithms is expected to significantly reduce overall memory consumption. This would not only optimize memory usage during quotient graph construction but also decrease memory required for each generated instance in the memetic refinement phase.
Moreover, to improve solution quality and reduce runtime, we could integrate other in-memory graph clustering algorithms for refining the quotient graph. For instance, we could leverage in-memory algorithms discussed in Chapter 3, such as the LEIDEN or LOUVAIN algorithm. In this case, we can expect a reduction in runtime and memory consumption, but also a reduction in solution quality, as the evolutionary scheme is omitted, eliminating the need to generate multiple instances and undergo the five phases of an evolutionary scheme.
Another promising approach is to explore parallelization in the local search phase of CLUSTRE. We could apply a technique similar to the *Two-Phase Label Propagation* introduced by the TERAPART [51] algorithm for the graph partitioning problem. The Two-Phase Label Propagation iterates over the set of nodes in parallel and in some cases over the neighbors of nodes as well. This approach can be integrated into our local search phase, as detailed in Section 4.4. We would iterate over the active node set in parallel, using the same iteration scheme as in the Two-Phase Label Propagation algorithm, except that, instead of moving nodes to the clusters with the strongest connection, we move nodes to the cluster yielding the highest modularity gain. We expect this modification to improve the algorithm's runtime.
Finally, to improve solution quality when a maximum cluster limit is set, we suggest modifying the gain function in Section 4.2. As shown in Section 5.4.4 of the tuning experiments, setting an upper bound significantly reduces solution quality, even when using the memetic refinement phase. One suggestion to mitigate this phenomenon is to modify the gain function to account for the current number of initialized clusters, assigning nodes to already existing clusters if the modularity decrease is only minimal. This approach would later allow nodes to be assigned to a new cluster if their neighborhood assignment drastically decreases the modularity score. This modification could help counteract the rapid drop in solution quality once the cluster boundary is reached.

APPENDIX A

Appendix

A.1 Modularity Equivalence Proof

Theorem 2. *Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to \mathbb{R}_{\geq 0}$, 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 defined as:*

$$mod_G(\mathcal{C}) = \frac{1}{m} \sum_{C_i \in C} \big(K_{C_i \to C_i} - \frac{vol(C_i)^2}{2m} \big), \tag{A.1}$$

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,*
- $\mathrm{vol}(C_i) = \sum_{v \in C_i} d_w(v)$ *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 $G_Q = \big(V_Q, E_Q\big)$, where each cluster C_i in G is contracted to a supernode v'_i in G_Q, with edge weights:

$$w'(v'_i, v'_j) = K_{C_i \to C_j} = \sum_{u \in C_i,\, v \in C_j, (u,v)\in E} w(u,v). \tag{A.2}$$

and self-loop weights $w'(v'_i, v'_i) = K_{C_i \to C_i}$. Then:

- *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 $mod_{G_Q}(\mathcal{C}') = mod_G(\mathcal{C})$.*

- *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 $mod_G(\hat{\mathcal{C}}) = mod_{G_Q}(\mathcal{C}')$*

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

$$m' = \sum_{(v'_i,v'_j)\in E_Q} w'(v'_i, v'_j) \; = \sum_{(u,v)\in E} w(u,v) = m. \tag{A.3}$$

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:

$$mod_{G_Q}(\mathcal{C}') = \frac{1}{m} \sum_{v'_i \in V_Q} \left(w'(v'_i, v'_i) \; - \; \frac{\text{vol}(v'_i)^2}{2m} \right). \tag{A.4}$$

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

$$mod_{G_Q}(\mathcal{C}') = \frac{1}{m} \sum_{C_i \in \mathcal{C}} \left(K_{C_i \to C_i} \; - \; \frac{\text{vol}(C_i)^2}{2m} \right) = mod_G(\mathcal{C}). \tag{A.5}$$

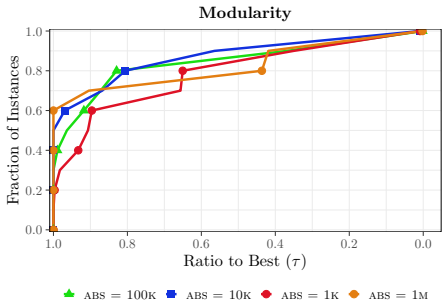
Modularity is invariant under contraction.
For the reverse direction, consider a clustering $\mathcal{C}'$ in G_Q. The expansion $\mathcal{C}'$ into G defines a clustering $\hat{\mathcal{C}}$, where each supernode $v'_i \in C'_j$ is replaced by its original cluster C_i, forming $\hat{\mathcal{C}}_j =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

$$mod_G(\hat{\mathcal{C}}) = \frac{1}{m} \sum_{\hat{C}_j \in \hat{\mathcal{C}}} \left(K_{\hat{C}_j \to \hat{C}_j} \; - \; \frac{\text{vol}(\hat{C}_j)^2}{2m} \right). \tag{A.6}$$

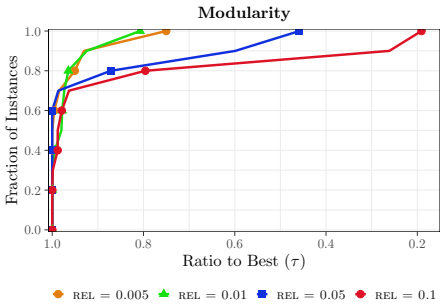
By construction, $K_{\hat{C}_j \to \hat{C}_j} = K_{C'_j \to C'_j}$ and $\text{vol}(\hat{C}_j) = \text{vol}(C'_j) \implies mod_G(\hat{\mathcal{C}}) = mod_{G_Q}(\mathcal{C}')$. Therefore, modularity is invariant under expansion. ■

A.2 $v_{\max}$ Tuning for Hollocou

As detailed above, the HOLLOCOU algorithm introduced by Hollocou et al. [25] requires a predefined parameter, $v_{\max}$. However, the authors provide no guidance on selecting $v_{\max}$ and do not specify the value used in their experiments. Thus, we conduct this study to determine an appropriate value for $v_{\max}$, ensuring a fair comparison between our algorithm and HOLLOCOU.
For this study, we take two approaches. The first approach involves defining an absolute value, $v_{\text{abs}} \in \{10k, 100k, 500k, 1m\}$, and the second involves using a relative parameter, $v_{\text{rel}} \in \{0.1, 0.05, 0.01, 0.005\}$. The parameter v_{rel} represents the fraction of the total edges used as $v_{\max}$. This means that if $v_{\text{rel}} = 0.1$, then the $v_{\max}$ is set to 10% of the total edges.



(a) Modularity Score - Absolute Values



(b) Modularity Score - Relative Values

Figure A.1. Tuning Experiment: $v_{\max}$. Performance profiles for Modularity scores. For all experiments, we use the HOLLOCOU algorithm with varying $v_{\max}$ values, as indicated in the legend.

Absolute values set a fixed upper bound for each cluster, independent of graph size, while the relative parameter dynamically adjusts the maximum cluster size, preventing any single cluster from dominating the graph.
Note that memory consumption is independent of $v_{\max}$. This is because the algorithm initializes all vectors to size n, including the cluster volumes vector, with all nodes starting as singletons, regardless of $v_{\max}$. Furthermore, runtime is independent of $v_{\max}$ since it only determines whether to merge the current edge's endpoints. Thus, we omit performance profiles comparing runtime and memory consumption, as $v_{\max}$ does not affect them.

Figure A.1 illustrates the performance of the different $v_{\max}$ values. Comparing the absolute values, we observe that the algorithm with $v_{\text{abs}} = 10\text{k}$ performs best, achieving an average modularity score 26.94% higher than the worst-performing algorithm, $v_{\text{abs}} = 1$ million, across all instances. Additionally, it attains 13.45% higher solution quality on average across all instances compared to the second-best algorithm, $v_{\max} = 1\text{k}$. Among the algorithms with varying relative $v_{\max}$ values, $v_{\text{rel}} = 0.01$ achieves an average modularity score 33.74% higher than the worst-performing algorithm, $v_{\text{rel}} = 0.1$, while also outperforming the second-best algorithm, $v_{\text{rel}} = 0.005$, by 0.27%

Figure A.2 depicts the comparison of the two best-performing configurations of the relative and absolute values, namely $v_{\text{abs}} = 10\text{k}$, $v_{\text{abs}} = 1\text{k}$, and $v_{\text{rel}} = 0.01$, $v_{\text{rel}} = 0.005$. There we we observe that, the relative parameter values result in higher solution quality. Specifically, $v_{\text{rel}} = 0.01$ increases modularity score on average by 6.48% across all tuning instances compared to the best-performing absolute value, $v_{\text{abs}} = 10\text{k}$.
For the reasons mentioned above, the most suitable parameter to use as a baseline for HOLLOCOU is the best-performing configuration of a relative value for $v_{\max}$, namely $v_{\text{rel}} = 0.01$.

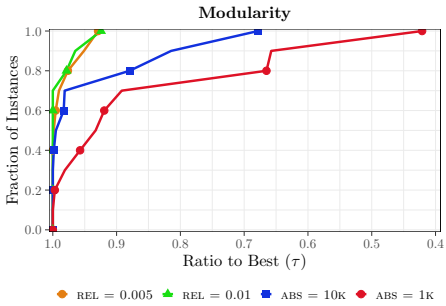


Figure A.2. Tuning Experiment: $v_{\max}$. Performance profiles for Modularity scores. For all experiments, we use the HOLLOCOU algorithm with varying relative and absolute values for $v_{\max}$, as indicated in the legend.

A.3 Further Results

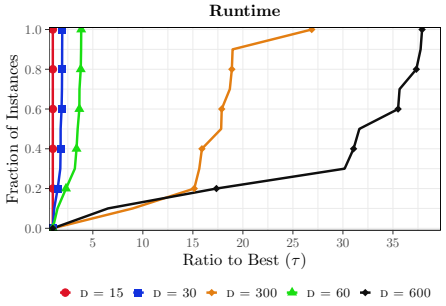


Figure A.3. Tuning Experiment: Memetic Refinement. Performance profile for runtime. For all algorithms, we use the baseline CLUSTRE configuration with no restriction on the maximum number of clusters. The algorithms underwent a streaming phase followed by a memetic refinement phase with varying duration values (D).

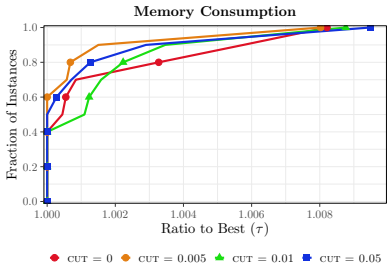


Figure A.4. Tuning Experiment: Local Search Cut-off Constraint. No Memetic Refinement. Performance profile for peak memory consumption. For all algorithms, we use the baseline CLUSTRE configuration with no memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search threshold value ls_{cutoff} varies as indicated in the legend (CUT).

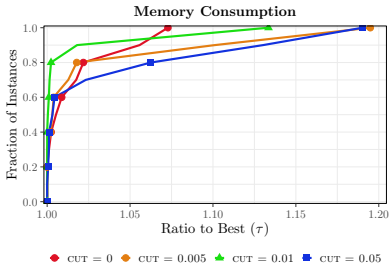


Figure A.5. Tuning Experiment: Local Search Cut-off Constraint. With Memetic Refinement. Performance profiles for peak memory consumption. For all algorithms, we use the baseline CLUSTRE configuration with memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search threshold value ls_{cutoff} varies as indicated in the legend (CUT).

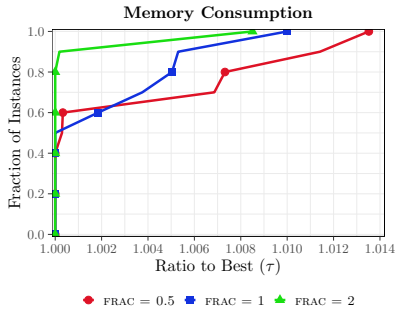


Figure A.6. Tuning Experiment: Local Search Time Fraction Constraint. No Memetic Refinement. Performance profiles for peak memory consumption. For all algorithms, we use the baseline CLUSTRE configuration with no memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search time fraction $ls_{\text{frac\_time}}$ varies as indicated in the legend (FRAC).

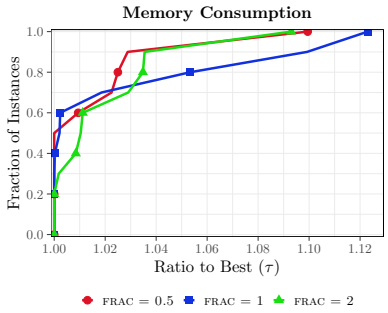


Figure A.7. Tuning Experiment: Local Search Cut-off Constraint. With Memetic Refinement. Performance profiles for peak memory consumption. For all algorithms, we use the baseline CLUSTRE configuration with memetic refinement, no restriction on the maximum number of clusters, and one re-stream iteration. The local search time fraction $ls_{\text{frac\_time}}$ varies as indicated in the legend (FRAC).

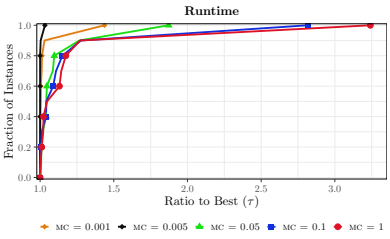


Figure A.8. Tuning Experiment: Max Cluster. Performance profile for runtime. For all algorithms, we use the baseline CLUSTRE configuration with varying values for the maximum cluster fraction parameter (MC). Each algorithm underwent a streaming phase followed by a 15-second memetic phase.

Zusammenfassung

Das Clustern eines Graphen in dijunkte Gemeinschaften ist eine zentrale Technik der Datenanalyse, um Interaktionen und ähnlichkeiten zwischen Entitäten innerhalb eines Datensatzes zu bewerten. In dieser Arbeit präsentieren wir den neuartigen Streaming-Graph-Clustering-Algorithmus CLUSTRE, der mithilfe eines mehrstufigen Verfeinerungsschemas ein ausgewogenes Verhältnis zwischen rechnerischer Effizienz und hochwertigem Clustering erreicht. CLUSTRE verarbeitet den Graphen in einer Streaming-Umgebung, wodurch der Gesamtspeicherverbrauch erheblich reduziert wird. Gleichzeitig nutzt er Re-Streaming und evolutionäre Heuristiken, um die Lösungsqualität weiter zu verbessern. Während des Streamings erzeugt CLUSTRE dynamisch einen Quotientengraphen, der die wesentlichen strukturellen Eigenschaften des ursprünglichen Graphen bewahrt. Diese Methode ermöglicht effiziente, modularitätsbasierte Optimierungen für große Graphen. CLUSTRE bietet verschiedene Konfigurationsoptionen, die unterschiedliche Kompromisse zwischen Laufzeit, Speicherverbrauch und Clustering-Qualität erlauben und so seine Vielseitigkeit unterstreichen. Unser Ansatz erzielt eine Lösungsqualität, die bestehende Streaming-Clustering-Algorithmen um mehr als 92 % übertrifft, während er 1,36× schneller arbeitet und nur 72,76 % des Speicherverbrauchs moderner Streaming-Methoden benötigt. Darüber hinaus steigert CLUSTRE in seiner leistungsstärksten Konfiguration die Lösungsqualität um mehr als 140 %. Diese Ergebnisse zeigen, dass CLUSTRE eine Lösungsqualität erreicht, die mit In-Memory-Clustering-Algorithmen weitgehend vergleichbar ist. Er erzielt über 97 % der Qualität hochmoderner In-Memory-Algorithmen wie LOUVAIN und überbrückt damit effektiv die Lücke zwischen Streaming- und In-Memory-Clustering.

Bibliography

[1] Vicente Arnau, Sergio Mars, and Ignacio Marín. Iterative cluster analysis of protein interaction data. *Bioinformatics*, 21(3):364–378, 2005.

[2] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

[3] 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.

[4] Sonja Biedermann, Monika Henzinger, Christian Schulz, and Bernhard Schuster. Memetic Graph Clustering. In *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA'18)*, LIPIcs. Dagstuhl, 2018. Technical Report, arXiv:1802.07034.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, October 2008.

[6] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: Massive crawling for the masses. *ACM Transactions on the Web (TWEB)*, 12(2):1–26, 2018.

[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*, pages 587–596, 2011.

[8] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602, 2004.

[9] Ulrik Brandes. *Network analysis: methodological foundations*, volume 3418. Springer Science & Business Media, 2005.

[10] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE transactions on knowledge and data engineering*, 20(2):172–188, 2007.

[11] Adil Chhabra, Marcelo Fonseca Faraj, Christian Schulz, and Daniel Seemaier. Buffered streaming edge partitioning. *arXiv preprint arXiv:2402.11980*, 2024.

[12] Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis. *A unified view of kernel k-means, spectral clustering and graph cuts*. Citeseer, 2004.

[13] Laxman Dhulipala, Jakub Lkacki, Jason Lee, and Vahab Mirrokni. Terahac: Hierarchical agglomerative clustering of trillion-edge graphs. *Proceedings of the ACM on Management of Data*, 1(3):1–27, 2023.

[14] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91:201–213, 2002.

[15] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2015.

[16] Marcelo Fonseca Faraj and Christian Schulz. Buffered streaming graph partitioning. *ACM Journal of Experimental Algorithmics*, 27:1–26, 2022.

[17] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[18] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the national academy of sciences*, 104(1):36–41, 2007.

[19] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *Journal of Parallel and Distributed Computing*, 131:200–217, 2019.

[20] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3:379–397, 1999.

[21] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[22] Roger Guimera and Luís A Nunes Amaral. Functional cartography of complex metabolic networks. *nature*, 433(7028):895–900, 2005.

[23] Jiahao Guo, Pramesh Singh, and Kevin E Bassler. Resolution limit revisited: community detection using generalized modularity density. *Journal of Physics: Complexity*, 4(2):025001, 2023.

[24] Michael Hamann, Ben Strasser, Dorothea Wagner, and Tim Zeitz. Distributed graph clustering using modularity and map equation. In *European Conference on Parallel Processing*, pages 688–702. Springer, 2018.

[25] Alexandre Hollocou, Julien Maudet, Thomas Bonald, and Marc Lelarge. A streaming algorithm for graph clustering. *arXiv preprint arXiv:1712.04337*, 2017.

[26] Rongqi Jing, Zhengwei Jiang, Qiuyun Wang, Shuwei Wang, Hao Li, and Xiao Chen. From fine-grained to refined: Apt malware knowledge graph construction and attribution analysis driven by multi-stage graph computation. In *International Conference on Computational Science*, pages 78–93. Springer, 2024.

[27] Ravi Kannan, Santosh Vempala, and Adrian Vetta. On clusterings: Good, bad and spectral. *Journal of the ACM (JACM)*, 51(3):497–515, 2004.

[28] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13 Companion, pages 1343–1350, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: a comparative analysis. *Physical Review E-Statistical, Nonlinear, and Soft Matter Physics*, 80(5):056117, 2009.

[30] Andrea Lancichinetti, Santo Fortunato, and János Kertész. Detecting the overlapping and hierarchical community structure in complex networks. *New journal of physics*, 11(3):033015, 2009.

[31] Jure Leskovec and R Sosič. Snap: Stanford network analysis platform, 2013.

[32] Jiakang Li, Songning Lai, Zhihao Shuai, Yuan Tan, Yifan Jia, Mianyang Yu, Zichen Song, Xiaokang Peng, Ziyang Xu, Yongxin Ni, et al. A comprehensive review of community detection in graphs. *arXiv preprint arXiv:2309.11798*, 2023.

[33] Fanzhen Liu, Shan Xue, Jia Wu, Chuan Zhou, Wenbin Hu, Cecile Paris, Surya Nepal, Jian Yang, and Philip S Yu. Deep learning for community detection: progress, challenges and opportunities. *arXiv preprint arXiv:2005.08225*, 2020.

[34] Yue Liu, Jun Xia, Sihang Zhou, Xihong Yang, Ke Liang, Chenchen Fan, Yan Zhuang, Stan Z Li, Xinwang Liu, and Kunlun He. A survey of deep graph clustering: Taxonomy, challenge, application, and open resource. *arXiv preprint arXiv:2211.12875*, 2022.

[35] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[36] Noeleene Mallia-Parfitt and Georgios Giasemidis. Graph clustering and variational image segmentation for automated firearm detection in x-ray images. *IET Image Processing*, 13(7):1105–1114, 2019.

[37] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering, 2014.

[38] Brad L Miller and David E Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary computation*, 4(2):113–131, 1996.

[39] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical Review E-Statistical, Nonlinear, and Soft Matter Physics*, 69(6):066133, 2004.

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

[41] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[42] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[43] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1106–1114, 2013.

[44] Michael Ovelgönne and Andreas Geyer-Schulz. An ensemble learning strategy for graph clustering. *Graph partitioning and graph clustering*, 588:187, 2012.

[45] Jose B Pereira-Leal, Anton J Enright, and Christos A Ouzounis. Detection of functional modules from protein interaction networks. *Proteins: Structure, Function, and Bioinformatics*, 54(1):49–57, 2004.

[46] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[47] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

[48] Mehrdad Rostami, Mourad Oussalah, and Vahid Farrahi. A novel time-aware food recommender-system based on deep learning and graph clustering. *Ieee Access*, 10:52508–52524, 2022.

[49] Martin Rosvall, Daniel Axelsson, and Carl T Bergstrom. The map equation. *The European Physical Journal Special Topics*, 178(1):13–23, 2009.

[50] Subhajit Sahu. Heuristic-based dynamic leiden algorithm for efficient tracking of communities on evolving graphs. *arXiv preprint arXiv:2410.15451*, 2024.

[51] Daniel Salwasser, Daniel Seemaier, Lars Gottesbüren, and Peter Sanders. Tera-scale multilevel graph partitioning. *arXiv preprint arXiv:2410.19119*, 2024.

[52] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *2012 Proceedings of the fourteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 16–29. SIAM, 2012.

[53] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *International Symposium on Experimental Algorithms*, pages 164–175. Springer, 2013.

[54] Satu Elisa Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[55] Christian Schulz. *High quality graph partitioning*. Citeseer, 2013.

[56] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[57] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference*, pages 403–412. IEEE, 2015.

[58] Xing Su, Shan Xue, Fanzhen Liu, Jia Wu, Jian Yang, Chuan Zhou, Wenbin Hu, Cecile Paris, Surya Nepal, Di Jin, et al. A comprehensive survey on community detection with deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[59] Ole Tange. Gnu parallel-the command-line power tool. *Usenix Mag*, 36(1):42, 2011.

[60] V. A. Traag, L. Waltman, and N. J. van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1), March 2019.

[61] Vincent A Traag, Paul Van Dooren, and Yurii Nesterov. Narrow scope for resolution-limit-free community detection. *Physical Review E-Statistical, Nonlinear, and Soft Matter Physics*, 84(1):016114, 2011.

[62] Amanda L. Traud, Peter J. Mucha, and Mason A. Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, August 2012.

[63] Stijn Van Dongen. Graph clustering by flow simulation. *PhD thesis, University of Utrecht*, 2000.

[64] Stijn Van Dongen. Graph clustering via a discrete uncoupling process. *SIAM Journal on Matrix Analysis and Applications*, 30(1):121–141, 2008.

[65] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17:395–416, 2007.

[66] Shiping Wang, Jinbin Yang, Jie Yao, Yang Bai, and William Zhu. An overview of advanced deep graph node clustering. *IEEE Transactions on Computational Social Systems*, 11(1):1302–1314, 2023.

[67] Timothé Watteau, Aubin Bonnefoy, Simon Illouz-Laurent, Joaquim Jusseau, and Serge Iovleff. Advanced graph clustering methods: A comprehensive and in-depth analysis. *arXiv preprint arXiv:2407.09055*, 2024.

[68] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, pages 40–48. PMLR, 2016.