Efficient Maintenance of Leiden Communities in Large Dynamic Graphs

Chunxu Lin
The Chinese University of Hong Kong, Shenzhen
chunxulin1@cuhk.edu.cn

Yixiang Fang
The Chinese University of Hong Kong, Shenzhen
fangyixiang@cuhk.edu.cn

Yumao Xie
The Chinese University of Hong Kong, Shenzhen
beckham_of_laiwu@foxmail.com

Yongming Hu
ByteDance Inc
huyongmin@bytedance.com

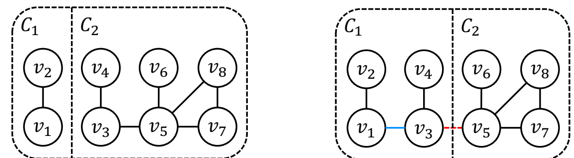
Yingqian Hu
ByteDance Inc
huyingqian@bytedance.com

Chen Cheng
ByteDance Inc
chencheng.sg@bytedance.com

ABSTRACT

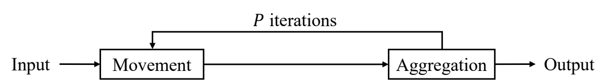
As a well-known community detection algorithm, Leiden has been widely used in various scenarios such as large language model generation (e.g., Graph-RAG), anomaly detection, and biological analysis. In these scenarios, the graphs are often large and dynamic, where vertices and edges are inserted and deleted frequently, so it is costly to obtain the updated communities by Leiden from scratch when the graph has changed. Recently, one work has attempted to study how to maintain Leiden communities in the dynamic graph, but it lacks a detailed theoretical analysis, and its algorithms are inefficient for large graphs. To address these issues, in this paper, we first theoretically show that the existing algorithms are relatively unbounded via the boundedness analysis (a powerful tool for analyzing incremental algorithms on dynamic graphs), and also analyze the memberships of vertices in communities when the graph changes. Based on theoretical analysis, we develop a novel efficient maintenance algorithm, called *Hierarchical Incremental Tree Leiden* (`HIT-Leiden`), which effectively reduces the range of affected vertices by maintaining the connected components and hierarchical community structures. Comprehensive experiments in various datasets demonstrate the superior performance of `HIT-Leiden`. In particular, it achieves speedups of up to five orders of magnitude over existing methods. Our algorithm has been deployed in ByteDance.

PVLDB Reference Format:
Chunxu Lin, Yixiang Fang, Yumao Xie, Yongming Hu, Yingqian Hu, and Chen Cheng. Efficient Maintenance of Leiden Communities in Large Dynamic Graphs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

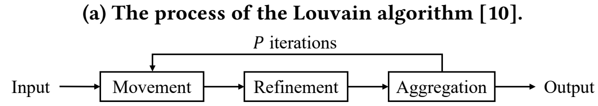


(a) A static graph G **(b) A dynamic graph G'**

Figure 1: Illustrating community maintenance, where (v_1, v_3) is a newly inserted edge and (v_3, v_5) is a newly deleted edge.



(a) The process of the Louvain algorithm [10].



(b) The process of the Leiden algorithm [80].

Figure 2: Illustrating the Louvain and Leiden algorithms.

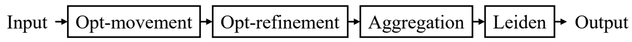
1 INTRODUCTION

As one of the fundamental measures in network science, modularity [60] effectively measures the strength of division of a network into modules (also called communities). Essentially, it captures the difference between the actual number of edges within a community and the expected number of such edges if connections were random. By maximizing the modularity of a graph, it can reveal all the communities in the graph. In Figure 1(a), for example, by maximizing the modularity of the graph, we can obtain two communities C_1 and C_2. As shown in the literature [13, 78], the graph communities have found a wide range of applications in recommendation systems, social marketing, and biological analysis.

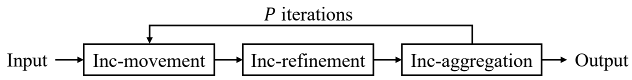
One of the most popular community detection (CD) algorithms that use modularity maximization is Louvain [10], which partitions a graph into disjoint communities. As shown in Figure 2(a), Louvain employs an iterative process with each iteration having two phases, called ***movement*** and ***aggregation***, to adjust the community structure and improve modularity. Specifically, in the movement phase, each vertex is relocated to a suitable community to maximize the modularity of the graph. In the aggregation phase, all the vertices belonging to the same community are merged into a hypervertex to form a hypergraph for the next iteration. Since a hypervertex corresponds to a set of vertices, the communities of a graph naturally form a tree-like hierarchical structure. In practice, to balance modularity gains against the running time, users often limit Louvain to P iterations, where P is a pre-defined parameter.

Despite its popularity, Louvain may produce communities that are internally disconnected. This typically occurs during the movement phase, where a vertex that serves as a bridge within a community may be moved to a different community that has stronger connections, thereby breaking the connectivity of the original community. To overcome this issue, Traag et al. [80] proposed the *Leiden*

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



(a) The process of the increment algorithms in [70].



(b) The process of our `HIT-Leiden` algorithm.

Figure 3: Algorithms for maintaining Leiden communities.

algorithm [1], which introduces an additional phase, called ***refinement***, between the movement and aggregation phases, as shown in Figure 2(b). Specifically, during the refinement phase, vertices explore merging with their neighbors within the same community to form sub-communities. By adding this additional phase, Leiden produces communities with higher quality than Louvain, since its communities well preserve the connectivity.

As shown in the literature, Leiden has recently received plenty of attention because of its applications in many areas, including large language model (LLM) generation [43, 54, 55, 63, 104], anomaly detection [27, 38, 65, 73, 82], and biological analysis [1, 8, 28, 47, 99]. For example, Microsoft has recently developed Graph-RAG [54], a retrieval-augmented generation (RAG) method that enhances prompts by searching external knowledge to improve the accuracy and trustworthiness of LLM generation, and builds a hierarchical index by using the communities detected by Leiden. As another example, Liu et al. introduced eRiskComm [48], a community-based fraud detection system that assists regulators in identifying high-risk individuals from social networks by using Louvain to partition communities, and Leiden can be naturally applied in this context.

In the aforementioned application scenarios, the graphs often evolve frequently over time, with many insertions and deletions of vertices and edges. For instance, in Wikipedia, the number of English articles increases by about 15,000 per month as of July 2024[2], making their contributors form a massive and continuously evolving collaboration graph, where nodes represent users. In these settings, changes to the underlying graph can significantly alter the communities produced by Leiden, thereby affecting downstream tasks and decision-making. However, the original Leiden algorithm is designed for static graphs, so it is very costly to recompute the communities from scratch using Leiden whenever a graph change occurs, especially for large graphs. Hence, it is strongly desirable to develop efficient algorithms for maintaining the up-to-date Leiden communities in large dynamic graphs.

Prior works. To maintain Louvain communities in dynamic graphs, several algorithms have been developed, such as DF-Louvain [69], Delta-Screening [97], DynaMo [105], and Batch [18]. However, little attention has been paid to maintaining Leiden communities. To the best of our knowledge, [70] is the only work that achieves this. It first uses some optimizations for the first iteration of `DF-Leiden`, and then invokes the original Leiden algorithm for the remaining iterations, as depicted in Figure 3(a). Following the optimized movement phase (`opt-movement`), the refinement phase in DF-Leiden separates communities affected by edge or vertex changes into multiple sub-communities, while leaving unchanged communities as single sub-communities. The aggregation phase remains identical to that of the Leiden algorithm. After constructing the aggregated graph, the standard Leiden algorithm is applied to complete the remaining CD process. The author has also developed two variants of `DF-Leiden`, called `ND-Leiden` and `DS-Leiden`, by using different optimizations for the movement phase of the first iteration. Nevertheless, there is a lack of detailed theoretical analysis for these algorithms, and they are inefficient for large graphs with few changes.

Our contributions. To address the above limitations, we first theoretically analyze the time cost of existing algorithms for maintaining Leiden communities and theoretically show that they are relatively unbounded via the boundedness analysis, which is a powerful tool for analyzing the time complexity of incremental algorithms on dynamic graphs. We further analyze the membership of vertices in communities and sub-communities when the graph edges change, and observe that the procedure for maintaining these memberships generalizes naturally to all the hypergraphs generated by Leiden. The above analysis not only lays a solid foundation for us to comprehend existing algorithms but also offers us opportunities to improve upon them.

Based on the above analyses, we develop a novel efficient maintenance algorithm, called Hierarchical Incremental Tree Leiden (HIT-Leiden), which effectively reduces the range of affected vertices by maintaining the connected components and hierarchical community structures. As depicted in Figure 3(b), `HIT-Leiden` is an iterative algorithm with each iteration having three key phases, namely incremental movement, incremental refinement, and incremental aggregation, abbreviated as `inc-movement`, `inc-refinement`, and `inc-aggregation`, respectively. More specifically, `inc-movement` extends the movement phase from [70] by incorporating hierarchical community structures [80]. Unlike prior approaches, it operates on a hypergraph where each hypervertex represents a sub-community, focusing on hierarchical dependencies between communities and their nested substructures. Inspired by the key technique of maintaining the connected components in dynamic graphs [90], `inc-refinement` maintains sub-communities by using tree-based structures to efficiently track changes in sub-communities. `Inc-aggregation` updates the hypergraph by computing structural changes based on the outputs of the previous two phases.

We have evaluated `HIT-Leiden` on several large-scale real-world dynamic graph datasets. The experimental results show that our algorithm achieves comparable community quality with the state-of-the-art algorithms for maintaining Leiden communities, while achieving up to five orders of magnitude faster than `DF-Leiden`. In addition, we have deployed our algorithm in real-world applications at ByteDance.

Outline. We first review related work in Section 2. We then formally introduce some preliminaries, including the Leiden algorithm and problem definition in Section 3, provide some theoretical analysis in Section 4, and present our proposed `HIT-Leiden` algorithm in Section 5. Finally, we present the experimental results in Section 6 and conclude in Section 7.

2 RELATED WORK

In this section, we first review the existing works of CD for both static and dynamic graphs. We simply classify these works as modularity and other metrics-based CD methods.

[1] As of July 2025, Leiden has received over 5,000 citations according to Google Scholar.
[2] https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia

• **Modularity-based CD.** Modularity-based CD methods aim to partition a graph such that communities exhibit high internal connectivity relative to a null model. Among these methods, Louvain [10] is the most popular one due to its high efficiency and scalability as shown in some comparative analyses [4, 39, 94]. Leiden [80] improves upon Louvain by resolving the problem of disconnected communities, yielding higher-quality results with comparable runtime. Other modularity heuristics [19, 56, 58] or incorporate simulated annealing [11, 37], spectral techniques [59], and evolutionary strategies [42, 49]. Further refinements explore multi-resolution [77], robust optimization [5], normalized modularity [52], and clustering cost frameworks [35]. Recent neural approaches have integrated modularity objectives into deep learning models [9, 12, 89, 93, 100], enhancing representation learning for CD.

Besides, some recent works have studied how to incrementally maintain modularity-based communities when the graph is changed. Aynaud et al. [6] proposed one of the earliest approaches by reusing previous community assignments to warm-start the Louvain algorithm. Subsequent works extended this idea to both Louvain [18, 20, 53, 62, 69, 74, 75, 97] and Leiden [70], incorporating mechanisms such as edge-based impact screening or localized modularity updates. Nevertheless, the existing algorithms of maintaining Leiden communities lack in-depth theoretical analysis, and their practical efficiency is poor. Other methods based on modularity, including extensions to spectral clustering [17], multi-step CD [7], and label propagation-based methods [61, 86–88] have been studied on dynamic graphs.

• **Other metrics-based CD.** Beyond modularity, various CD methods have been developed by using different optimization purposes, such as similarity, statistical inference, spectral clustering, and neural networks. The similarity-based methods like SCAN [23, 83, 92] identify dense regions from the graph via structural similarity. Statistical inference approaches, including stochastic block models [2, 29, 36, 64], infer communities by fitting generative probabilistic models to observed networks. Spectral clustering methods [3, 22, 57] exploit the eigenstructure of graph Laplacians to group nodes with similar structural roles. Deep learning-based methods for CD have recently gained traction. Graph convolutional networks [21, 31, 32, 40, 50, 76, 91, 101, 103], and graph attention networks [26, 34, 51, 81, 84, 96] have demonstrated strong performance in learning expressive node embeddings for CD tasks. For more details, please refer to recent survey papers of CD [13, 78].

Besides, many of the above methods have also been extended for dynamic graphs. Ruan et al. [68] and Zhang et al. [98] have studied structural graph clustering on dynamic graphs, which is based on structural similarity. Temporal spectral methods [16, 17] and dynamic stochastic block models [45, 72] enable statistical modeling of evolving community structures over time. Recent deep learning approaches also support dynamic CD through mechanisms such as temporal embeddings [102], variational inference [41], contrastive learning [15, 24, 85], and generative modeling [33]. These models capture temporal dependencies and structural evolution.

3 PRELIMINARIES

In this section, we first formally present the problem we study, and then briefly introduce the original Leiden algorithm. Table 1 summarizes the notations frequently used throughout this paper.

Table 1: Frequently used notations and their meanings.

Notation	Meaning
$G = (V, E)$	A graph with vertex set V and edge set E
$N(v)$, $N_2(v)$	The vertex v's 1- and 2-hop neighbor sets, resp.
$w(v_i, v_j)$	The weight of edge between v_i and v_j
$d(v)$	The weighted degree of vertex v
m	The total weight of all edges in G
$\mathbb{C}$	A set of communities forming a partition of G
Q	The modularity of the graph G with partition $\mathbb{C}$
$G^p = (V^p, E^p)$	The hypergraph in the p-th iteration of Leiden
$\Delta Q(v \to C', \gamma)$	Modularity gain by moving v from C to C' with γ
$f(\cdot)\colon V \to \mathbb{C}$	A mapping from vertices to communities
$f^p(\cdot)\colon V^p \to \mathbb{C}$	A mapping from hypervertices to communities
$s^p(\cdot)$: $V^p \to V^{p+1}$	A mapping from hypervertices in p-th level to hypervertices in $(p+1)$-th level (sub-communities)
ΔG	The set of changed edges in the dynamic graph

3.1 Problem definition

We consider an undirected and weighted graph $G = (V, E)$, where V and E are the sets of vertices and edges, respectively. Each vertex v's neighbor set is denoted by $N(v)$. Each edge (v_i, v_j) is associated with a positive weight $w(v_i, v_j) \geq 0$. The degree of v_i is given by $d(v_i) = \sum_{v_j \in N(v_i)} w(v_i, v_j)$. Denote by m the total weight of all edges in G, i.e., $m = \sum_{(v_i, v_j) \in E} w(v_i, v_j)$.

Given a graph $G = (V, E)$, the CD process aims to partition all the vertices of V into some disjoint sets $\mathbb{C}$, each of which is called a community, corresponding to a set of vertices that are densely connected. This process can be modeled as a mapping function $f(\cdot) : V \to \mathbb{C}$, such that each v belongs to a community $f(v)$ of the partition $\mathbb{C}$. For each vertex v, the total weight between v and a community C is denoted by $w(v, C) = \sum_{v' \in N(v) \cap C} w(v, v')$.

As a well-known CD metric, the modularity measures the difference between the actual number of edges in a community and the expected number of such edges.

Definition 1 (Modularity [10]). *Given a graph $G = (V, E)$ and a community partition $\mathbb{C}$ over V, the modularity $Q(G, \mathbb{C}, \gamma)$ of the graph G with the partition $\mathbb{C}$ is defined as:*

$$Q(G, \mathbb{C}, \gamma) = \sum_{C \in \mathbb{C}} \left(\frac{1}{2m} \sum_{v \in C} w(v, C) - \gamma \left(\frac{d(C)}{2m} \right)^2 \right), \quad (1)$$

where $d(C)$ is the total degree of all vertices in a community C, and $\gamma > 0$ is a hyperparameter.

Note that the parameter $\gamma > 0$ controls the granularity of the detected communities [67]. A higher γ favors smaller, finer-grained communities. In practice, γ is often set to 0.5, 1, 4, or 32, as shown in [46]. Besides, to guide community updates, the concept of modularity gain is often used to capture the changed modularity when a vertex is moved from one community to another.

Definition 2 (Modularity gain [10]). *Given a graph G, a partition $\mathbb{C}$, and a vertex v that belongs to a community C, the modularity gain of moving v from C to another community C' is defined as:*

$$\Delta Q(v \to C', \gamma) = \frac{w(v, C') - w(v, C)}{2m} + \frac{\gamma \cdot d(v) \cdot (d(C) - d(v) - d(C'))}{(2m)^2}. \quad (2)$$

In this paper, we focus on the dynamic graph with insertions of deletions of both vertices and edges. Since a vertex insertion (resp.

deletion) can be modeled as a sequence of edge insertions (resp. deletions), we simply focus on edge changes. Given a set of edge changes ΔG to a graph $G = (V, E)$, we obtain an updated graph $G' = (V', E')$. Since there are two types of edge updates, we let $\Delta G = \Delta G_+ \cup \Delta G_-$, where $\Delta G_+ = E' \setminus E$ and $\Delta G_- = E \setminus E'$ denote the sets of inserted and deleted edges, respectively. We use $G \oplus \Delta G$ to denote applying ΔG to G, yielding an updated graph G'.

We now formally introduce the problem studied in this paper.

Problem 1 (Maintenance of Leiden communities [70]). *Given a graph G with its Leiden communities $\mathbb{C}$, and some edge updates ΔG, return the updated Leiden communities after applying ΔG to G.*

We illustrate our problem via Example 1.

Example 1. *In Figure 1(a), the original graph G with unit edge weights contains two Leiden communities: $C_1 = \{v_1, v_2\}$ and $C_2 = \{v_3, v_4, v_5, v_6, v_7, v_8\}$. After inserting a new edge $(v_1, v_3, -1)$ and deleting an existing edge $(v_3, v_5, 1)$ into G, we obtain an updated graph G', which has two updated communities $C_1 = \{v_1, v_2, v_3, v_4\}$ and $C_2 = \{v_5, v_6, v_7, v_8\}$.*

3.2 Leiden algorithm

Algorithm 1 presents Leiden [71, 79], following the process in Figure 2(b). Given a graph G, and an initial mapping $f(\cdot)$ (w.l.o.g., $f(v) = \{v\}$), it first initializes the level-1 hypergraph G^1, lets level-1 mapping $f^1(\cdot)$ be $f(\cdot)$, and sets up the sub-community mapping $s(\cdot)$ (line 1). Next, it iterates P times, each having three phases.

(1) **Movement phase** (line 3): for each hypervertex v^p in the hypergraph G^p, it attempts to move v^p to a neighboring community that yields the maximum positive modularity gain, resulting in an updated community mapping $f^p(\cdot)$.
(2) **Refinement phase** (line 4): it splits each community into some sub-communities such that each of them corresponds to a connected component, producing a sub-community mapping $s^p(\cdot)$.
(3) **Aggregation phase** (line 6): when $p < P$, it aggregates each sub-community as a hypervertex and builds a new graph G^{p+1}.

Finally, after P iterations, we update $f(\cdot)$ and obtain the communities (lines 7-8). Note that $f(\cdot)$ is updated using $s^P(\cdot)$ rather than $f^P(\cdot)$ since sub-communities guarantee connectivity with comparable modularity. Besides, we use the terms hypervertex and sub-community interchangeably in this paper. A hyperedge is an edge between two hypervertices, and its weight is the sum of the weights of edges between their sub-communities.

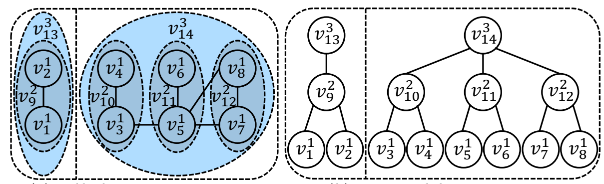
Clearly, the vertices assigned to a sub-community will be further aggregated as a hypervertex, so all the vertices and hypervertices generated naturally form a tree-like hierarchical structure. The total time complexity of Leiden is $O(P \cdot (|V| + |E|))$ [71], since each iteration costs $O(|V| + |E|)$ time.

Example 2. *Figure 4 (a) depicts the process of Leiden with P=3 for the graph in Figure 1. Denote by v_i^p the hypervertex (i.e., sub-community) in the p-th iteration of Leiden. It generates three levels of hypergraphs: G^1, G^2, and G^3, with $G^1 = G$. The vertices of these hypergraphs form a tree-like structure, as shown in Figure 4(b).*

Take the first iteration as an example depicted in Figure 5. In the movement phase, it generates three communities $C_1 = \{v_1^1, v_2^1\}$, $C_2 = \{v_5^1, v_6^1, v_7^1, v_8^1\}$ and $C_3 = \{v_3^1, v_4^1\}$. In the refinement phase, C_2 is split into two sub-communities $v_{11}^2 = \{v_5^1, v_6^1\}$ and $v_{12}^2 = \{v_7^1, v_8^1\}$, and C_1 and C_2 are unchanged. In the aggregation phase, all vertices are aggregated into hypervertices based on their sub-community memberships, resulting in G^2.

Algorithm 1: Leiden algorithm [71, 79]

Input: $G, f(\cdot), P, \gamma$
Output: Updated $f(\cdot)$
1 $G^1 \leftarrow G, f^1(\cdot) \leftarrow f(\cdot)$;
2 **for** $p = 1$ *to* P **do**
3 $\quad f^p(\cdot) \leftarrow Move(G^p, f^p(\cdot), \gamma)$;
4 $\quad s^p(\cdot) \leftarrow Refine(G^p, f^p(\cdot), \gamma)$;
5 $\quad$ **if** $p < P$ **then**
6 $\quad\quad G^{p+1}, f^{p+1}(\cdot) \leftarrow Aggregate(G^p, f^p(\cdot), s^p(\cdot))$;
7 Update $f(\cdot)$ using $s^1(\cdot), \cdots, s^P(\cdot)$;
8 **return** $f(\cdot)$;



(a) All the communities. **(b) A tree-like structure.**

Figure 4: The process of Leiden for the graph G in Figure 1(a).

4 THEORETICAL ANALYSIS OF LEIDEN

In this section, we first analyze the boundedness of existing algorithms, then study how vertex behavior impacts community structure under graph updates, and extend it for hypervertices.

4.1 Boundedness analysis

We first introduce some concepts related to boundedness.

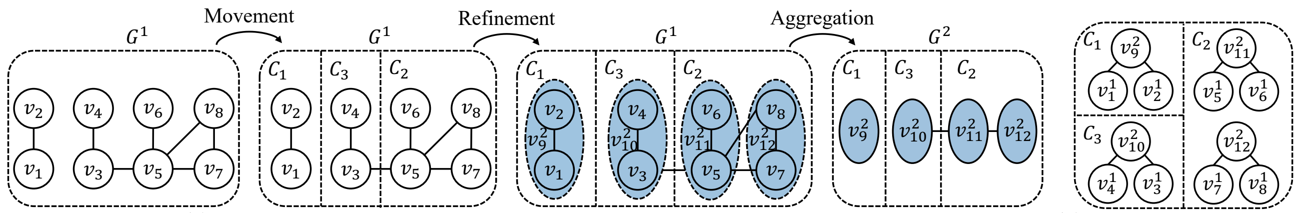
• **Notation.** Let Θ denote the CD query applied to a graph G, where $\Theta(G) = \mathbb{C}$ is the set of detected communities. The new graph is $G \oplus \Delta G$, and the updated community is $\Theta(G \oplus \Delta G)$. We denote the output difference as $\Delta\mathbb{C}$, where $\Theta(G \oplus \Delta G) = \Theta(G) \oplus \Delta\mathbb{C}$.

• **Concepts of boundedness.** The notion of boundedness [66] evaluates the effectiveness of an incremental algorithm using the metric CHANGED, defined as CHANGED $= \Delta G + \Delta\mathbb{C}$, which leads to $|$CHANGED$| = |\Delta G| + |\Delta\mathbb{C}|$.

Definition 3 (Boundedness [25, 66]). *An incremental algorithm is bounded if its computational cost can be expressed as a polynomial function of $|$CHANGED$|$ and $|\Theta|$. Otherwise, it is unbounded.*

• **Concepts of relative boundedness.** In real-world dynamic graphs, $|$CHANGED$|$ is often small, yet some unbounded algorithms can be solved in polynomial time using measures comparable to $|$CHANGED$|$, making these algorithms feasible. To assess these incremental algorithms effectively, Fan et al. [25] introduced the concept of relative boundedness, which leverages a more refined cost model called the affected region. Let AFF denote the affected part, the region of the graph actually processed by the incremental algorithm.

Definition 4 (AFF [25]). *Given a graph G, a query Θ, and the input update ΔG to G, AFF signifies the cost difference of the static algorithm between computing $\Theta(G)$ and $\Theta(G \oplus \Delta G)$.*



(a) The process of hierarchical partitions at first iteration on the graph. **(b) The tree-like structure.**

Figure 5: The process of hierarchical partitions of Figure 4 at level-1 with the Leiden algorithm.

Unlike CHANGED, AFF captures the concrete portion of the graph touched by an incremental algorithm, providing a tighter bound on its computational cost. This leads to the following definition.

Definition 5 (Relative boundedness [25]). *An incremental graph algorithm is relatively bounded to the static algorithm if its cost is polynomial in* $|\Theta|$ *and* $|\mathsf{AFF}|$.

We now analyze the boundedness of existing incremental Leiden algorithms.

Theorem 1. *When processing an edge deletion or insertion, the increment Leiden algorithms proposed in [70] all cost* $O(P\cdot(|V|+|E|))$.

Table 2: Incremental Leiden algorithms

Method	Time complexity	Relative boundedness
ST-Leiden [70]	$O(P \cdot (\|V\| + \|E\|))$	✗
DS-Leiden [70]	$O(P \cdot (\|V\| + \|E\|))$	✗
DF-Leiden [70]	$O(P \cdot (\|V\| + \|E\|))$	✗
HIT-Leiden	$O(\|N_2(\mathsf{CHANGED})\| + \|N_2(\mathsf{AFF})\|)$	✓

By Theorem 1, the existing algorithms for maintaining Leiden communities are both unbounded and relatively unbounded as shown in Table 2. They are very costly for large graphs, even with a small update. Following, we review the property of Leiden and then identify AFF of Leiden in the end.

4.2 Vertex optimality and subpartition γ-density

As shown in the literature [10, 80], if $s^P(\cdot) = f^P(\cdot)$ after P iterations in Leiden, it is guaranteed that the following two properties hold:

- **Vertex optimality:** All the vertices are vertex optimal.
- **Subpartition γ-density:** All the communities are subpartition γ-dense.

To design an efficient and effective maintenance algorithm for Leiden communities, we analyze the behaviors of vertices and communities when the graph changes as follows.

• **Analysis of vertex optimality.** We begin with a key concept.

Definition 6 (Vertex optimality [10]). *A community* $C \in \mathbb{C}$ *is called vertex optimality if for each vertex* $v \in C$ *and* $C' \in \mathbb{C}$, *the modularity gain* $\Delta Q(v \to C', \gamma) \le 0$.

Prior studies suggest that when the number of edge updates is small relative to the graph size, three heuristics hold: (1) intra-community edge deletions and inter-community edge insertions are the most likely to affect vertex-level community membership [69, 97]; (2) Inter-community edge deletions and intra-community edge insertions can be ignored [69, 97]; (3) Vertices directly involved in such edge changes are the most likely to alter their communities [69]. The heuristics are stated in Lemma 2, which can be proved based on Definition 6.

Lemma 2 ([69]). *Given an intra-community edge deletion* $(v_i, v_j, -\alpha)$ *with* $-\alpha < 0$, *the communities of both vertices* v_i *and* v_j *are likely to change. Similarly, given a cross-community edge insertion* (v_i, v_j, α) *with* $\alpha > 0$, *the communities of both vertices* v_i *and* v_j *are likely to change.*

We further derive the propagation of community changes from Lemma 2.

Lemma 3. *When a vertex* v *changes its community to* C, *then the communities of its neighbors not in* C *in the updated graph are also likely to change.*

Proof. Assuming v changes its community from C_i to C, there are three cases:

(1) For each neighbor v_i in C_i, the edge (v, v_i) is a *deleted intra-community* edge and an inserted cross-community edge;
(2) For each neighbor v_j in C, the edge (v, v_j) is a deleted cross-community edge and an inserted intra-community edge;
(3) For each other neighbor v_k, edge (v, v_k) is a deleted cross-community edge and an *inserted cross-community* edge.

Since only the first and third cases meet the conditions in Lemma 2, all the neighbors of v that are not in C are likely to change their communities. □

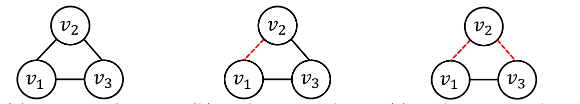
Lemma 3 implies that after inserting or deleting an edge (v_i, v_j), we need to verify the vertices that satisfy the conditions in Lemma 2: if any vertex changes its community, then we have to verify its neighbors in other communities. This motivates us to develop a novel movement phase, called inc-movement, in our maintenance algorithm HIT-Leiden, which will be introduced in Section 5.1.

• **Analysis of subpartition γ-density.** We first introduce some key concepts for defining subpartition γ-density, and analyze the subpartition γ-density in a dynamic graph.

Definition 7 (γ-order). *Given two vertex sets* X *and* Y *of a graph* G, *let* $X \otimes Y$ *represent the* γ*-merge between them, meaning that* Y *is merged into* X *such that* $2m \cdot w(X, Y) \ge \gamma \cdot d(X) \cdot d(Y)$, *where* $w(X,Y) = \sum_{v_i \in X}\sum_{v_j \in Y} w(v_i, v_j)$. *A* γ*-order of a vertex subset* $U = \{v_1, \cdots, v_x\}$ *is a sequence that represents a sequence of* γ*-merges beginning from singleton subsets* $\{v_1\}, \cdots, \{v_x\}$.

Definition 8 (γ-connectivity [80]). *Given a graph* G, *a vertex set* U *is* γ*-connected, if* U *can be generated from any* γ*-order .*

Definition 9 (Subpartition γ-density [80]). *A vertex set* $U \subseteq C \in \mathbb{C}$ *is subpartition* γ*-dense if* U *is* γ*-connected, and any intermediate vertex set* X *is locally optimized, i.e.,* $\Delta Q(X \to \emptyset, \gamma) \le 0$.



(a) A triangle. (b) Delete an edge. (c) Delete two edges.

Figure 6: An example for illustrating subpartition γ-density.

EXAMPLE 3. *The triangle in Figure 6(a) is subpartition γ-dense with $\gamma = 1$ since there are six different γ-orders. For instance, one is $\{v_3\} \otimes (\{v_1\} \otimes \{v_2\})$, which represents that v_2 is merged into $\{v_1\}$ generating subset $\{v_1, v_2\}$, and then $\{v_1, v_2\}$ merges into v_3 generating subset $\{v_1, v_2, v_3\}$. After deleting the edge (v_1, v_2), although $\{v_3\} \otimes (\{v_1\} \otimes \{v_2\})$ is not a γ-order, the update graph is still subpartition γ-dense since $\{v_1\} \otimes (\{v_2\} \otimes \{v_3\})$ is a γ-order in the update graph. After continuing to delete the edge (v_2, v_3), the updated graph is not subpartitionγ-dense since v_2 is not connected to v_1 and v_3.*

In essence, each community C (or sub-community S) of Leiden is subpartition γ-dense, since (1) any sub-community in C (or S) is locally optimized, and (2) all sub-communities are γ-connected. Notably, vertex optimality ensures the first condition by design. Hence, to preserve subpartition γ-density under dynamic updates, it suffices to maintain γ-connectivity of sub-communities.

Next, we analyze the subpartition γ-density property under three kinds of graph updates, i.e., *edge deletion*, *edge insertion*, and *vertex movement*. For lack of space, all the proofs of lemmas are shown in the appendix of the full version [44].

(1) Edge deletion. We consider the deletions of both intra-sub-community edges and cross-sub-community edges:

LEMMA 4. *Given an intra-sub-community edge deletion $(v_i, v_j, -\alpha)$ with a γ-order and $-\alpha < 0$, the later-inserted vertex in the edge is likely to leave the sub-community.*

LEMMA 5. *Given a cross-sub-community edge deletion $(v_i, v_j, -\alpha)$ with $-\alpha < 0$, the deletion is unlikely to affect the sub-community membership of any involved vertex.*

(2) Edge insertion. We consider the insertion of an edge:

LEMMA 6. *Given an edge insertion (v_i, v_j, α) with $\alpha > 0$, it is unlikely to affect the sub-community memberships.*

(3) Vertex movement. We consider the scenario that a vertex moves from its sub-community to another one:

LEMMA 7. *Given a sub-community S with a γ-order, when a vertex v moves out of S, its neighbors in the updated graph, which merge into S after v in γ-order, are likely to leave S.*

Lemma 4 implies that given a γ-order, after deleting an edge (v_i, v_j), we need to verify the vertices that satisfy the conditions in Lemma 7: if any vertex changes their sub-community, then we have to verify their neighbors which merge into S after the vertex according to γ-order. However, Leiden only offers us a γ-order from the refinement phase, and a subgraph often exists with multiple distinct γ-orders as shown in Example 3. Besides, if a vertex is a candidate affecting γ-connectivity, it is often a candidate affecting vertex optimality, e.g., the vertex v_2 in Figure 6(c). In this case, the vertex is likely to change its community before verifying whether the vertex needs to move out of its sub-community. We assume that each connected component of a sub-community is treated as a γ-connected subset in the updated graph. This motivates us to develop a novel refinement phase, called `inc-refinement`, in `HIT-Leiden`, which will be introduced in Section 5.2.

In addition, changes at the lower-level propagate upward to hyperedge changes in the higher-level hypergraph, since Leiden builds a list of hypergraphs in a bottom-up manner. This motivates us to develop an incremental aggregation phase, namely `inc-aggregation`, to compute the hyperedge changes in Section 5.3.

EXAMPLE 4. *In Figure 1, communities C_1 and C_2 are treated as hypervertices. Deleting an edge $(v_3, v_5, 1)$ and inserting an edge $(v_1, v_3, 1)$ cause v_4 and v_5 to move from C_1 to C_2. This results in the deletion of $(C_2, C_2, -2)$ and insertion of $(C_1, C_1, 2)$ in the hypergraph.*

Characterization of `AFF`. Based on these analyses, we define the hypervertices that change their communities or sub-communities as the affected area `AFF` of Leiden.

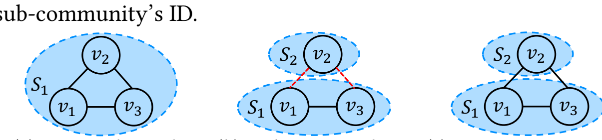
5 OUR HIT-LEIDEN ALGORITHM

In this section, we first introduce the key components, `inc-movement`, `inc-refinement`, and `inc-aggregation` of our `HIT-Leiden`. Then, we present an auxiliary procedure, called deferred update, abbreviated as `def-update`. Afterwards, we give an overview of `HIT-Leiden`, and finally analyze the boundedness of `HIT-Leiden`.

5.1 Inc-movement

The goal of `inc-movement` is to preserve vertex optimality. As analyzed in Section 4.2, the endpoints of a deleted intra-community edge or an inserted cross-community edge may affect their community memberships. If an affected vertex changes its community, its neighbors outside the target community may also be affected. Note that any vertex that changes its community has to change its sub-community, since each sub-community is a subset of its community. Hence, sub-community memberships are also considered in `inc-movement`.

We first introduce the data structures used to maintain a dynamic sub-community. Each connected component of a sub-community is treated as a γ-connected subset. When edge updates or vertex movements split a sub-community into multiple connected components, we re-assign each resulting component as a new sub-community, and the largest sub-community succeeds the original sub-community's ID.



(a) Original graph. (b) Delete two edges. (c) Move out a vertex.

Figure 7: Illustrating the process that a sub-community S_1 is split into two sub-communities S_1 and S_2.

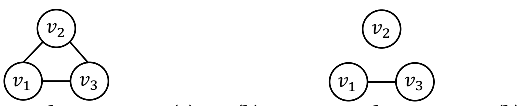
EXAMPLE 5. *Figure 7 shows the sub-community S_1 is split into two sub-communities $S_1 = \{v_1, v_3\}$ and $S_2 = \{v_2\}$. The component $\{v_1, v_3\}$ retains the original sub-community ID S_1, since it is larger than $\{v_2\}$. The separation can occur either due to the deletion of edges (v_1, v_2) and (v_2, v_3) during graph updates, as shown in Figure 7(b), or due to the removal of vertex v_2 during the movement phase, as shown in Figure 7(c).*

To preserve the structure under such changes, we leverage dynamic connected component maintenance techniques. Various index-based methods have been proposed for this purpose, such as D-Tree

Algorithm 2: `Inc-movement`

Input: $G, \Delta G, f(\cdot), s(\cdot), \Psi, \gamma$
Output: Updated $f(\cdot), \Psi, B, K$

1 $A \leftarrow \emptyset, B \leftarrow \emptyset, K \leftarrow \emptyset$;
2 **for** $(v_i, v_j, \alpha) \in \Delta G$ **do**
3   **if** $\alpha > 0$ *and* $f(v_i) \neq f(v_j)$ **then**
4     $A.add(v_i); A.add(v_j)$;
5   **if** $\alpha < 0$ *and* $f(v_i) = f(v_j)$ **then**
6     $A.add(v_i); A.add(v_j)$;
7   **if** $s(v_i) = s(v_j)$ *and* $update\_edge\left(G_\Psi, (v_i, v_j, \alpha)\right)$ **then**
8     $K.add(v_i); K.add(v_j)$;
9 **for** $A \neq \emptyset$ **do**
10   $v_i \leftarrow A.pop()$;
11   $C^* \leftarrow argmax_{C \in \mathbb{C} \cup \emptyset} \Delta Q(v_i \rightarrow C, \gamma)$;
12   **if** $\Delta Q(v_i \rightarrow C^*, \gamma) > 0$ **then**
13     $f(v_i) \leftarrow C^*; B \leftarrow v_i$;
14   **for** $v_j \in N(v_i)$ **do**
15   **if** $f(v_j) \neq C^*$ **then**
16         $A.add(v_j)$;
17   **for** $v_j \in N(v_i) \wedge s(v_i) = s(v_j)$ **do**
18   **if** $update\_edge\left(G_\Psi, (v_i, v_j, -w(v_i, v_j))\right)$ **then**
19         $K.add(v_i); K.add(v_j)$;
20 **return** $f(\cdot), \Psi, K$;



(a) Corresponds to Figure 7(a). **(b) Corresponds to Figure 7(b).**

Figure 8: An example of G_Ψ with the sub-community memberships corresponding to Figure 7.

[14], DND-Tree [90], and HDT [30]. Let Ψ denote a connected component index, abbreviated as CC-index. The graph G_Ψ stores the subgraph of G consisting only of intra-sub-community edges based on $s(\cdot)$. Figure 8 gives an example of G_Ψ in Figures 7 (a) and (b).

Algorithm 2 shows `inc-movement`. Given an updated graph G, a set of graph changes ΔG, community mappings $f(\cdot)$, sub-community mappings $s(\cdot)$, and a CC-index Ψ, it first initializes three empty sets: A, B and K (line 1). Here, A keeps the vertices whose community memberships may be changed, B keeps the vertices that have changed their community memberships, and K records the endpoints on edges whose deletion disconnects the connected component in G_Ψ. Subsequently, vertices involved in intra-community edge deletion or cross-community edge insertion are added to A, and edges in G_Ψ are updated according to intra-sub-community changes (lines 2-7). If an edge update in G_Ψ causes a connected component to split (i.e., $update\_edge(\cdot)$ returns $true$), its endpoints are added to K (line 8). It then processes vertices in A until the set is empty (line 9). For each vertex v_i, it identifies the target community C^* that yields the highest modularity gain (lines 10-11). If $\Delta Q(v_i \rightarrow C^*) > 0$, $f(v_i)$ is updated to C^*, v_i are added into B, and the neighbors of v_i not in C^* are added to A (line 12-16). Besides, the intra-sub-community edges involving v_i are deleted from G_Ψ, and the vertices involved in component splits are added to K (lines 18-19). Finally, it returns $f(\cdot)$, Ψ, and K (line 20).

Algorithm 3: `Inc-refinement`

Input: $G, f(\cdot), s(\cdot), \Psi, K, \gamma$
Output: Updated $s(\cdot), \Psi, R$,

1 $R \leftarrow \emptyset$;
2 **for** $v_i \in K$ **do**
3   **if** v_i *is not in the largest connected component of* $s(v)$ **then**
4   Map all vertices in the connected component into a new sub-community and add them into R;
5 **for** $v_i \in R$ **do**
6   **if** v_i *is in singleton sub-community* **then**
7     $\mathcal{T} \leftarrow \{s(v) | v \in N(v_i) \cap f(v_i), \Delta Q(s(v) \rightarrow \emptyset, \gamma) \leq 0\}$;
8     $S^* \leftarrow argmax_{S \in \mathcal{T}} \Delta M(v_i \rightarrow S, \gamma)$;
9   **if** $\Delta M(v_i \rightarrow S^*, \gamma) > 0$ **then**
10       $s(v_i) \leftarrow S^*$;
11   **for** $v_j \in N(v_i)$ **do**
12   **if** $s(v_i) = s(v_j)$ **then**
13           $update\_edge\left(G_\Psi, (v_i, v_j, w(v_i, v_j))\right)$;
14 **return** $s(\cdot), \Psi, R$;

5.2 Inc-refinement

As discussed in Section 4.2 and Section 5.1, we treat each connected component in G_Ψ as a sub-community which is maintained in `inc-movement`. Therefore, we design `inc-refinement` for reassigning each new connected component in G_Ψ as a sub-community. Additionally, we attempt to merge singleton sub-communities whose process is the same as the process of the refinement phase in Leiden with G_Ψ maintenance.

Algorithm 3 presents its pseudocode. Given an updated graph G, community mappings $f(\cdot)$ and sub-community mapping $s(\cdot)$, a CC-index Ψ, and a set K, it first initializes R as an empty set to track vertices which have changed their sub-communities (line 1). It then traverses K to identify split connected components in G_Ψ using breadth-first search or depth-first search. If a connected component is not the largest in its original sub-community, all its vertices are re-mapped to a new sub-community, and added to R (lines 2-4). If multiple components tie for the largest component, one of them is randomly selected to represent the original sub-community. For each vertex $v_i \in R$ that is in a singleton sub-community, `inc-refinement` uses a set $\mathcal{T}$ to store the locally optimized neighboring sub-communities of v_i within the same community (lines 5-7). Then, it attempts to re-assign v_i to a sub-community $S^* \in \mathcal{T}$, which offers the highest modularity gain to eliminate singleton sub-communities (line 8). Notably, $\Delta M(v_i \rightarrow S, \gamma)$ denotes the modularity gain of moving v_i from $s(v_i)$ to S, whose calculation follows the same formula as the standard modularity gain. If the gain is positive, $s(v_i)$ is updated to S^*, and the corresponding intra-sub-community edges are inserted into G_Ψ (lines 9-13). Finally, `inc-refinement` returns the $s(\cdot)$, Ψ, and R (line 14).

5.3 Inc-aggregation

Given an updated graph G and its edge changes ΔG, modifications to edges and sub-community memberships are reflected as changes to hyperedges and hypervertices in the hypergraph H. Let $s_{pre}(\cdot)$ (resp. $s_{cur}(\cdot)$) denotes the vertex-to-hypervertex mappings before (resp. after) `inc-refinement`. Any edge change (v_i, v_j, α) in ΔG corresponds to a hyperedge change $(s_{pre}(v_i), s_{pre}(v_j), \alpha)$ in H, since

Algorithm 4: `Inc-aggregation`

Input: G, ΔG, $s_{pre}(\cdot)$, $s_{cur}(\cdot)$, R
Output: ΔH, $s_{pre}(\cdot)$

1 $\Delta H \leftarrow \emptyset$;
2 **for** $(v_i, v_j, \alpha) \in \Delta G$ **do**
3     $r_i \leftarrow s_{pre}(v_i), r_j \leftarrow s_{pre}(v_j)$;
4     $\Delta H.add((s_i, s_j, \alpha))$;
5 **for** $v_i \in R$ **do**
6   **for** $v_j \in N(v_j)$ **do**
7   **if** $s_{cur}(v_j) = s_{pre}(v_j)$ *or* $i < j$ **then**
8             $\Delta H.add((s_{pre}(v_i), s_{pre}(v_j), -w(v_i, v_j)))$;
9             $\Delta H.add((s_{cur}(v_i), s_{cur}(v_j), w(v_i, v_j)))$;
10 **for** $v_i \in R$ **do**
11     $s_{pre}(v_i) \leftarrow s_{cur}(v_i)$;
12 $Compress(\Delta H)$;
13 **return** ΔH, $s_{pre}(\cdot)$;

the weight of a hyperedge is the sum of weights of edges between their sub-communities. Besides, a vertex v migration from $s_{\text{pre}}(v)$ to $s_{\text{cur}}(v)$ requires updating these weights. Specifically, the original sub-community $s_{pre}(v)$ must decrease the hyperedge weights corresponding to the edge incident to v, and the new sub-community $s_{cur}(v)$ must increase them under the new assignment.

Example 6. *Following Example 4, the initial hyperedge changes due to edge changes are* $(C_1, C_2, 1)$ *and* $(C_2, C_2, -1)$*. Then, vertices* v_3 *and* v_4 *move from* C_2 *to* C_1*, and there are three cases:*

(1) *C_1 gains edges to the neighbors of v_3, resulting in two updates: $(C_1, C_1, 1)$ and $(C_1, C_1, 1)$;*
(2) *C_2 loses edges to the neighbor of v_3 are $(C_1, C_2, -1)$ and $(C_2, C_2, -1)$;*
(3) *The effect of v_4 is skipped to avoid duplicate updates, since its only neighbor v_3 already changed.*

After compressing the above six hyperedge changes, we obtain the final hyperedge changes, which are $(C_1, C_1, 2)$ *and* $(C_2, C_2, -2)$.

Algorithm 4 presents `inc-aggregation`. Initially, the set of changes ΔH of H is empty (line 1). Then, it maps the edge changes ΔG to hyperedge changes using $s_{pre}(\cdot)$ (lines 2-4). Following, it updates hyperedges for vertices that switch sub-communities by removing edges from the old community and adding edges to the new one. For any vertex v_i in R, if one of its neighbors v_j in G also changes sub-community, it only updates hyperedges if $i < j$ to avoid duplicate updates (lines 5-9). Finally, it locally updates $s_{pre}(\cdot)$ to match $s_{cur}(\cdot)$ for the next time step (Lines 10-11), and compresses entries by summing the weight of identical hyperedges in ΔH (lines 12).

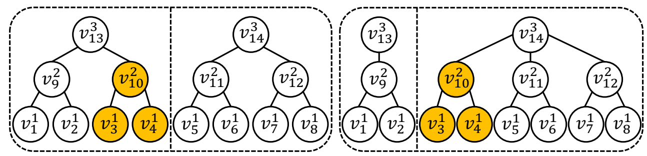
5.4 Overall HIT-Leiden algorithm

Before presenting our overall HIT-Leiden algorithm, we introduce an optimization technique to further improve the efficiency of the vertices' membership update. Specifically, when a hypervertex changes its community membership, all the lower-level hypervertices associated with it have to update their community membership. As shown in Figure 9, when v^2_{10} changes its community, v^1_3 and v^1_4 also update their community memberships to the community containing v^2_{10}. However, during the iteration process of `HIT-Leiden`, a hypervertex that changes its community does not automatically trigger updates to the community memberships of its constituent lower-level hypervertices.

Algorithm 5: `def-update`

Input: $\{f^P(\cdot)\}$, $\{s^P(\cdot)\}$, $\{B^P\}$, P
Output: Updated $\{f^P(\cdot)\}$

1 **for** p *from* P *to* 1 **do**
2   **if** $p \neq P$ **then**
3   **for** $v^p_i \in B^p$ **do**
4             $f^p(v^i_p) = f^{p+1}(s^p(v^i_p))$;
5   **if** $p \neq 1$ **then**
6   **for** $v^p_i \in B^p$ **do**
7             $B^{p-1}.add(s^{-p}(v^p_i))$;
8 **return** $\{f^P(\cdot)\}$;



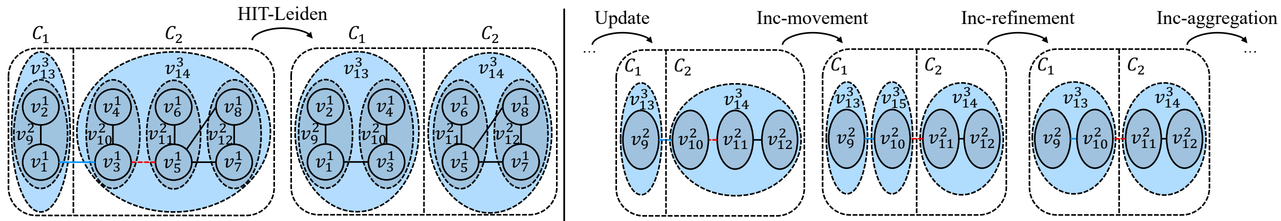
(a) Before maintenance. (b) After maintenance.

Figure 9: The hierarchical partitions changes of Figure 1.

To resolve the above inconsistency, we perform a post-processing step to synchronize the community memberships across all levels, as described in Algorithm 5. Let $\{B^P\}$ denote a sequence of P sets $\{B^1, \cdots, B^P\}$, $\{s^P(\cdot)\}$ denote a sequence of P adajcent-level hypervertex mappings $\{s^1(\cdot), \cdots, s^P(\cdot)\}$, and $\{f^P(\cdot)\}$ denote a sequence of P community mappings $\{f^1(\cdot), \cdots, f^P(\cdot)\}$. Specifically, each set B^p in $\{B^P\}$ collects hypervertices at level-p whose community memberships have changed, each mapping $s^p(\cdot)$ in $\{s^P(\cdot)\}$ maps from level-p hypervertices to their parent hypervertices at level-$(p+1)$, and each $f^p\{\cdot\}$ in $\{f^P(\cdot)\}$ maps from level-p hypervertices to their assigned communities. A hypervertex is added to B^p for one of two reasons: (1) it changes its community during `inc-movement`, or (2) its higher-level hypervertex changes community. Hence, for each level p, `def-update` updates each hypervertex in B^p by remapping its community membership of its parent using $s^p(\cdot)$ and $f^{p+1}(\cdot)$ (lines 1-4), and adds its constituent vertices to B^{p-1} for the next level updates where $s^{-p}(\cdot)$ is the inverse mapping of $s^p(\cdot)$ (lines 5-7). This algorithm also supports updating the mappings $\{g^P(\cdot)\}$ from each level hypervertex to its level-P ancestor.

• **Overall HIT-Leiden.** After introducing all the key components, we present our overall `HIT-Leiden` in Algorithm 6. The algorithm proceeds over P hierarchical levels, where each level-p operates on a corresponding hypergraph G^p. Besides the community membership $f(\cdot)$, `HIT-Leiden` also maintains hypergraphs $\{G^P\}$, community mappings $\{f^P(\cdot)\}$, sub-community mappings $\{g^P(\cdot)\}$, $\{s^P_{pre}(\cdot)\}$ and $\{s^P_{cur}(\cdot)\}$, and CC-indices $\{\Psi^P\}$ to maintain sub-community for each level. Note, $\{s^P_{pre}(\cdot)\}$ are the mappings from the previous time step, and $\{s^P_{cur}(\cdot)\}$ are the in-time mappings to track sub-community memberships as they evolve at the current time step.

Specifically, it initializes $\{s^P_{cur}(\cdot)\} = \{s^P_{pre}(\cdot)\}$. Given the graph change ΔG, it first initializes the first-level update ΔG to ΔG^1 (line



(a) Community maintain by `HIT-Leiden` **(b) The process of `HIT-Leiden` in iteration two**

Figure 10: An example of `HIT-Leiden`

Algorithm 6: `HIT-Leiden`

Input: $\{G^P\}$, ΔG, $\{f^P(\cdot)\}$, $\{g^P(\cdot)\}$,$\{s^P_{pre}(\cdot)\}$, $\{s^P_{cur}(\cdot)\}$, $\{\Psi^P\}$, P, γ

Output: $f(\cdot)$, $\{G^P\}$, $\{f^P(\cdot)\}$, $\{f^P(\cdot)\}$, $\{s^P_{pre}(\cdot)\}$, $\{s^P_{cur}(\cdot)\}$, $\{\Psi^P\}$

1 $\Delta G^1 \leftarrow \Delta G$;

2 **for** p *from* 1 *to* P **do**

3 $\quad G^p \leftarrow G^p \oplus \Delta G^p$;

4 $\quad f^p(\cdot), \Psi, B^p, K \leftarrow$ `inc-movement`$(G^p, \Delta G^p, f^p(\cdot), \Psi, \gamma)$;

5 $\quad s^p_{cur}(\cdot), \Psi, R^p \leftarrow$ `inc-refinement`$(G^p, f^p(\cdot), s^p_{cur}(\cdot), \Psi, K, \gamma)$;

6 $\quad$ **if** $p < P$ **then**

7 $\quad\quad \Delta G^{p+1}, s^p_{pre}(\cdot) \leftarrow$ `inc-aggregation`$(G^p, \Delta G^p, s^p_{pre}(\cdot), s^p_{cur}(\cdot), R^p)$;

8 `def-update`$(\{f^P(\cdot)\}, \{s^P_{cur}(\cdot)\}, \{B^P\}, P)$;

9 `def-update`$(\{g^P(\cdot)\}, \{s^P_{cur}(\cdot)\}, \{R^P\}, P)$;

10 $f(\cdot) \leftarrow g^1(\cdot)$;

11 **return** $f(\cdot)$, $\{G^P\}$, $\{f^P(\cdot)\}$, $\{g^P(\cdot)\}$, $\{s^P_{pre}(\cdot)\}$, $\{s^P_{cur}(\cdot)\}$, $\{\Psi^P\}$;

---

1). It then proceeds through P iterations, each including three phases after updating the hypergraph G^p (line 3).

(1) `Inc-movement` (line 4): it re-assigns community memberships of affected vertices to achieve vertex optimality, which yields $f^p(\cdot)$, Ψ, B^p, and K.
(2) `Inc-refinement` (line 5): it re-maps the hypervertices of split connected components in Ψ to new sub-communities, producing $s^p_{cur}(\cdot)$, Ψ, and R^p.
(3) `Inc-aggregation` (line 7): it calculates the next level's hyperedge changes ΔG^{p+1}, and synchronizes $s^p_{pre}(\cdot)$ to match $s^p_{cur}(\cdot)$.

After P iterations, `def-update` (Algorithm 5) synchronizes community mappings $\{f^P(\cdot)\}$ and sub-community mappings $\{g^P(\cdot)\}$ across levels (lines 8-9). The final output $f(\cdot)$ is set to $g^1(\cdot)$ (line 10). Besides, we also return $\{G^P\}$, $\{f^P(\cdot)\}$, $\{g^P(\cdot)\}$, $\{s^P_{pre}(\cdot)\}$ and $\{s^P_{cur}(\cdot)\}$ for the next graph evolution (line 11).

Example 7. *Consider the result in Figure 4. The graph undergoes an edge deletion $(v^1_3, v^1_5, -1)$ and an edge insertions $(v^1_1, v^1_3, 1)$. Resulting community and sub-community changes are shown in Figure 10, with hierarchical changes in Figure 9. Take the second iteration as an example. In* `inc-movement`*, the hypervertex v^2_{10} is reassigned to v^3_{15} due to disconnection, and migrates from community C_2 to C_1. In* `inc-refinement`*, v^2_{10} is merged into v^3_{13}. Then,* `inc-aggregation` *calculates hyperedge changes for level-3, including edge insertion $(v^3_{13}, v^3_{13}, 2)$ and edge deletions $(v^3_{14}, v^3_{14}, -2)$.*

• **Complexity analysis.** We now analyze the time complexity of `HIT-Leiden` over P iterations. Let Γ^p denote the set of hypervertices involved in hyperedge changes, and array Λ^p track the hypervertices that change their communities or sub-communities at level-p. Therefore, for each level-p, `inc-movement`, `inc-refinement`, and `inc-aggregation` complete in $O(|N_2(\Gamma^p)|+|N_2(\Lambda^p)|)$, $O(|N(\Gamma^p)|+|N(\Lambda^p)|)$, and $O(|N(\Gamma^p)|+|N(\Lambda^p)|)$, respectively. Besides, the time cost of `def-update` is $O\left(\sum_{p=1}^{P}|\Lambda^p|\right)$. Hence, the total time cost of `HIT-Leiden` is $O(\sum_{p=1}^{P}(|N_2(\Gamma^p)|+|N_2(\Lambda^p)|)) = O(|N_2(\mathsf{CHANGED})|+|N_2(\mathsf{AFF})|)$, as analyzed in Section 4.2. As a result, our `HIT-Leiden` is bounded relative to Leiden.

6 EXPERIMENTS

We now present our experimental results. Section 6.1 introduces the experimental setup. Section 6.2 and 6.3 evaluate the effectiveness and efficiency of `HIT-Leiden`, respectively.

6.1 Setup

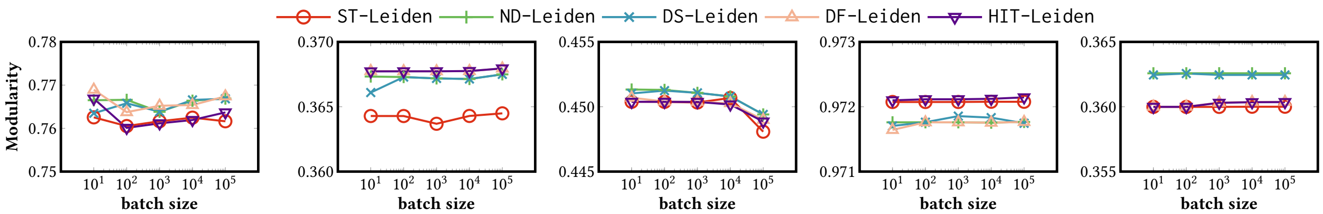
Table 3: Datasets used in our experiments.

Dataset	Abbr.	$\lvert V\rvert$	$\lvert E\rvert$	Timestamp
dblp-coauthor	DC	1.8M	29.4M	Yes
yahoo-song	YS	1.6M	256.8M	Yes
sx-stackoverflow	SS	2.6M	63.4M	Yes
it-2004	IT	41.2M	1.0B	No
risk	RS	201.0M	4.0B	Yes

Datasets. We use four real-world dynamic datasets, including *dblp-coauthor*[1] (academic collaboration), *yahoo-song*[1] (user-song interactions), *sx-stackoverflow*[2] (developer Q&A), and *risk* (financial transactions) provided by ByteDance. All these dynamic edges are associated with real timestamps. We also use one static dataset *it-2004*[3] (a large-scale web graph), but randomly insert or delete some edges to simulate a dynamic graph. All the graphs are treated as undirected graphs. For each real-world dynamic graph, we collect a sequence of batch updates by sorting the edges in ascending order of their timestamps; for *it-2004*, which lacks timestamps, we randomly shuffle its edge order. Table 3 summarizes the key statistics of the above datasets.

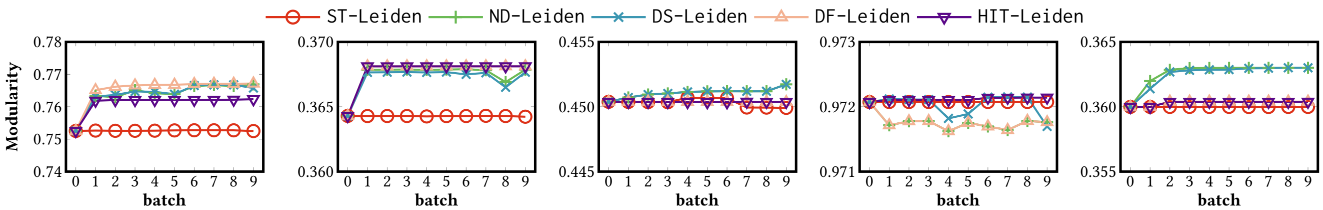
Algorithms. We test the following maintenance algorithms:

[1] http://konect.cc/networks/
[2] https://snap.stanford.edu/data/
[3] https://networkrepository.com/



(a) DC **(b)** YS **(c)** SS **(d)** IT **(e)** RS

Figure 11: Modularity values on dynamic graphs.



(a) DC **(b)** YS **(c)** SS **(d)** IT **(e)** RS

Figure 12: Modularity changes w.r.t. the number of update batches.

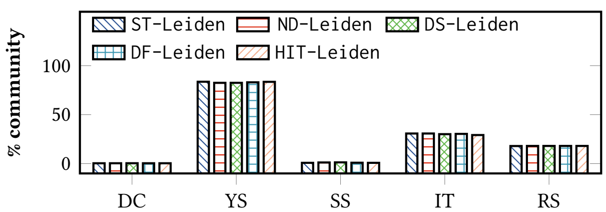


Figure 13: Percentage of badly connected communities.

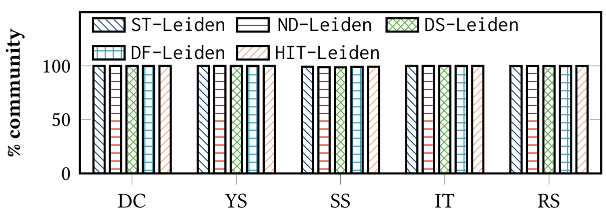


Figure 14: Percentage of subpartition γ-dense communities.

- ST-Leiden: A naive baseline that executes the static Leiden algorithm from scratch when the graph changes.
- ND-Leiden: A simple maintenance algorithm in [70], which processes all vertices during the movement phase, initialized with previous community memberships.
- DS-Leiden: A maintenance algorithm based on [70], which uses the delta-screening technique [97] to restrict the number of vertices considered in the movement phase.
- DF-Leiden: An advanced maintenance algorithm from [70], which adopts the dynamic frontier approach [69] to support localized updates.
- HIT-Leiden: Our proposed method.

Dynamic graph settings. As the temporal span varies across datasets (e.g., 62 years for *dblp-coauthor* versus 8 years for *sx-stackoverflow*), we apply a sliding edge window, avoiding reliance on fixed valid time intervals that are hard to standardize. Initially, we construct a static graph using the first 80% of edges. Then, we select a window size $b \in \{10, 10^2, 10^3, 10^4, 10^5\}$, denoting the number of updated edges in an updated batch. Next, we slide this window $r = 9$ times, so we update 9 batches of edges for each dataset. Note that by default, we set $b = 10^3$.

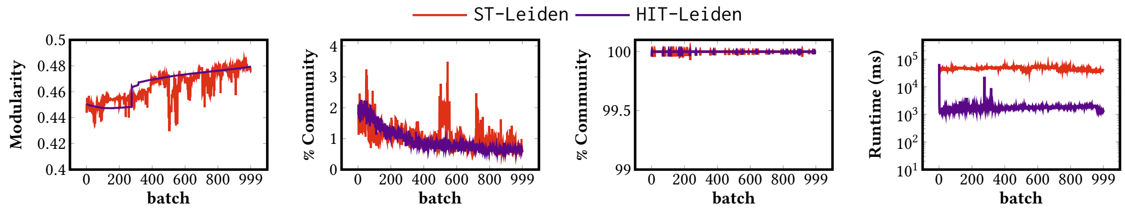
All the algorithms are implemented in C++ and compiled with the gcc 8.3.0 compiler using the -O0 optimization level. We set $\gamma = 1$ and use $P = 10$ iterations. Before running the Leiden community maintenance algorithms, we obtain the communities by running the Leiden algorithm, and HIT-Leiden requires an additional procedure to build auxiliary structures. Due to the limited number of iterations, the community structure has not fully converged, so the maintenance algorithms usually take more time in the first two batches than in other batches. Therefore, we exclude the first two batches from efficiency evaluations. Experiments are conducted on a Linux server running Debian 5.4.56, equipped with an Intel(R) Xeon(R) Platinum 8336C CPU @ 2.30GHz and 2.0 TB of RAM.

6.2 Effectiveness evaluation

To evaluate the effectiveness of different maintenance algorithms, we compare the modularity value, proportion of badly connected communities, and proportion of subpartition γ-density communities for their returned communities. We also evaluate the long-term effectiveness of community maintenance and present a case study.

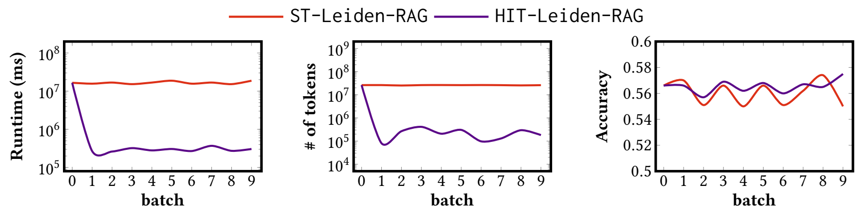
• **Modularity.** Figure 11 depicts the average modularity values of all the maintenance algorithms, where the batch size ranges from 10 to 10^5. Figure 12 depicts the modularity value across all the 9 batches, where the batch size is fixed as 1,000. Across all datasets, the expected fluctuation in modularity for ST-Leiden is around 0.02 due to its inherent randomness. These maintenance algorithms achieve equivalent quality in modularity, since the difference in their modularity values is within 0.01. Overall, our HIT-Leiden achieves comparable modularity with other methods.

• **Proportion of badly connected communities.** [80] presents a method to measure the number of badly connected communities; that is, if a community needs to be split into multiple subcommunities, then it is counted as badly connected. Figure 13 shows the percentages of badly connected communities returned by all the maintenance algorithms. The differences in the proportions of badly connected communities among these algorithms are below 0.01,



(a) Modularity (b) Bad connectivity (c) Subpartition γ-density (d) Runtime

Figure 15: Effectiveness of HIT-Leiden in 999 update batches.



(a) Runtime (b) Token cost (c) Accuracy

Figure 16: Comparing `ST-Leiden-RAG` and `HIT-Leiden-RAG`.

which falls within the expected fluctuation (around 0.02) caused by the inherent randomness of `ST-Leiden` and the measurement method. As a result, `HIT-Leiden` achieves a comparable percentage of bad connectivity with others.

• **Proportion of subpartition γ-density.** After running `HIT-Leiden`, for each returned community, we try to re-find its γ-order such that any intermediate vertex set in the γ-order is locally optimized, according to Definition 9. If we can find a valid γ-order for a community, we classify it as a subpartition γ-dense community. We report the proportion of subpartition γ-dense communities in Figure 14. The proportions of subpartition γ-density communities among these Leiden algorithms are almost 1, and they are within the expected fluctuation (around 0.0001) caused by the inherent randomness of the measure method. Thus, `HIT-Leiden` achieves a comparable percentage of subpartition γ-density with others.

• **Long-term effectiveness.** To demonstrate the long-term effectiveness of maintaining communities, we enlarge the number r of batches from 9 to 999 and set $b = 10,000$. Figure 15 presents the modularity, proportion of badly connected communities, proportion of subpartition γ-dense communities, and runtime of `ST-Leiden` and `HIT-Leiden` on the sx-stackoverflow dataset. We observe that `HIT-Leiden` exhibits higher stability than `ST-Leiden` in both modularity and number of badly connected communities since it uses previous community memberships. Besides, it is also much faster than `ST-Leiden`. Note that when updating the 300^{th} batch, `HIT-Leiden` incurs high runtime due to substantial changes in community memberships caused by edge evolution, which is also reflected in the corresponding increase in modularity.

• **A case study.** Our HIT-Leiden has been deployed at ByteDance to support several real applications. Here, we briefly introduce the application of Graph-RAG. To augment the LLM generation for answering a question, people often retrieve relevant information from an external corpus. To facilitate the retrieval, Graph-RAG builds an offline index: it first builds a graph for the corpus, then clusters the graph hierarchically using Leiden, and finally associates a summary for each community which is generated by an LLM with some token cost. In practice, since the underlying corpus often changes, the communities and their summaries need to be updated as well. Our `HIT-Leiden` can not only dynamically update the communities efficiently, but also save the token cost since we only need re-generate the summaries for the updated communities.

To do the experiment, we use the HotpotQA [95] dataset, which contains Wikipedia-based question-answer (QA) pairs. We randomly select 9,500 articles to build the initial graph, and insert 9 batches of new articles, each with 5 articles. The LLM we use is doubao-1.5-pro-32k. To support dynamic corpus, we adapt the static Graph-RAG method by updating communities using `ST-Leiden` and `HIT-Leiden`, respectively. These two RAG methods are denoted by `ST-Leiden-RAG` and `HIT-Leiden-RAG`, respectively. We report their runtime, token cost, and accuracy in Figure 16. Clearly, `HIT-Leiden-RAG` is 56.1× faster than `ST-Leiden-RAG`. Moreover, it significantly reduces the summary token cost while preserving downstream QA accuracy, since its token cost is only 0.8% of the token cost of `ST-Leiden-RAG`. Hence, `HIT-Leiden` is effective for supporting Graph-RAG on dynamic corpus.

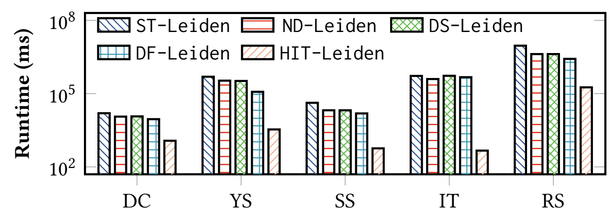
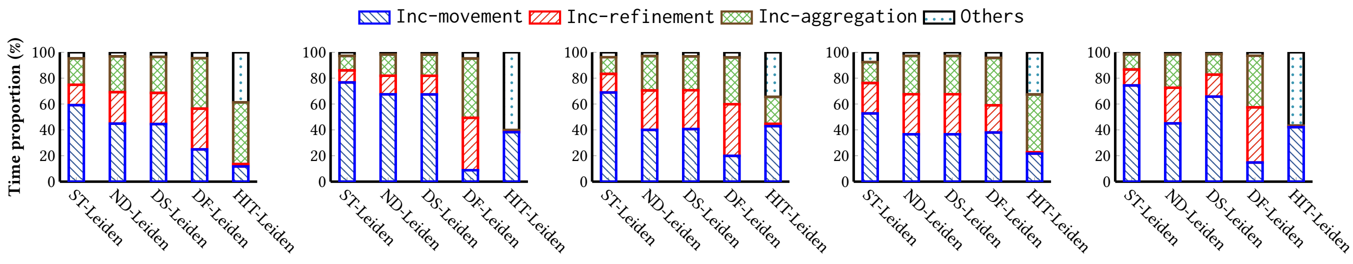


Figure 17: Efficiency of all Leiden algorithms on all datasets.

6.3 Efficiency evaluation

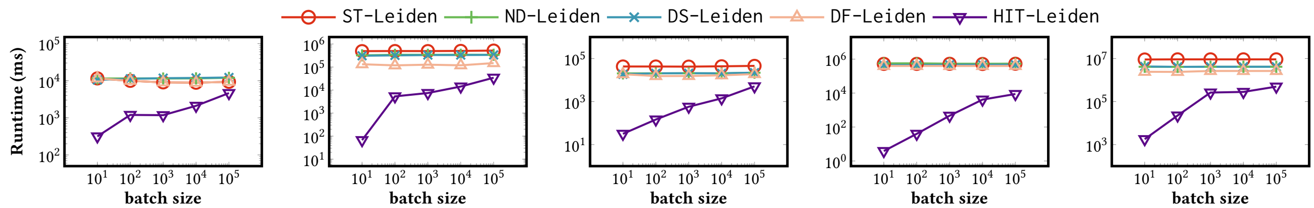
In this section, we first present the overall efficiency results, then analyze the time cost of each component, and finally evaluate the effects of some hyperparameters.

• **Overall results.** Figure 17 presents the overall efficiency results where b is set to its default value 1,000. Clearly, `HIT-Leiden` achieves the best efficiency on datasets, especially on the `it-2004` dataset, since it is up to three orders of magnitude faster than



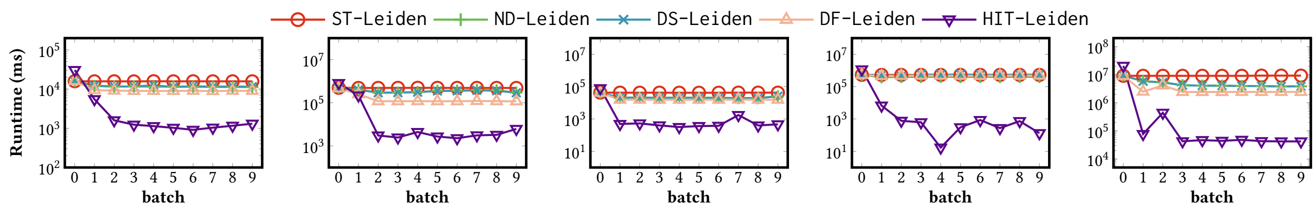
(a) DC **(b)** YS **(c)** SS **(d)** IT **(e)** RS

Figure 18: Proportion of time cost of each component for the Leiden algorithms on all datasets.



(a) DC **(b)** YS **(c)** SS **(d)** IT **(e)** RS

Figure 19: Runtime on dynamic graphs.



(a) DC **(b)** YS **(c)** SS **(d)** IT **(e)** RS

Figure 20: Runtime w.r.t. the number of update batches.

the state-of-the-art algorithms. That is mainly because the number of updated edges in `it-2004` deviates more from the total number of edges than those of `dblp-coauthor`, `yahoo-song`, and `sx-stackoverflow`.

• **Time cost of different components in** `HIT-Leiden`**.** There are three key components, i.e., `inc-movement`, `inc-refine`, and `inc-aggregation`, in `HIT-Leiden`. We evaluate the proportion of time cost for each component and present the results in Figure 18. Note that some operations (e.g., `def-update` in `HIT-Leiden`) may not be included by the above three components, so we put them into the "Others" component. Notably, in `HIT-Leiden`, the refinement phase contributes minimally to the overall runtime. Besides, the combined proportion of time spent in its movement and aggregation phase is comparable to that of other algorithms. This implies that `inc-movement`, `inc-refinement`, and `inc-aggregation` consistently outperform their counterparts in other algorithms across all datasets, achieving lower absolute runtime costs.

• **Effect of** b**.** We vary the batch size $b \in \{10, 10^2, 10^3, 10^4, 10^5\}$ and report the efficiency in Figure 19. We see that `HIT-Leiden` is up to five orders of magnitude faster than other algorithms. Also, it exhibits a notable increase as b becomes smaller because it is a relatively bounded algorithm. In contrast, `ND-Leiden`, `DS-Leiden`, and `DF-Leiden` still need to process the entire graph when processing a new batch.

• **Effect of** r**.** Recall that after fixing the batch size b, we update the graph for r batches. Figure 20 shows the efficiency, where b is fixed as 1,000, but r ranges from 1 to 9. We observe that the incremental speedup is limited in the first few batches because $P = 10$ is small, and additional iterations may slightly improve the community membership. As a result, all the maintenance algorithms often require more time for the second batch to adjust the community structure. Once high-quality community structure is established, the speedup becomes significant. In addition, `HIT-Leiden` incurs a slightly higher runtime to record more information and construct the CC-index.

7 CONCLUSIONS

In this paper, we study the problem of maintenance of Leiden communities on a dynamic graph and develop an efficient algorithm. We first theoretically analyze the boundedness of existing algorithms and how hypervertex behaviors affect community membership under graph update. Building on these analyses, we further develop a relative boundedness algorithm, called `HIT-Leiden`, which consists of three key components, i.e., `inc-movement`, `inc-refinement`, and `aggregation`. Extensive experiments on five real-world dynamic graphs show that `HIT-Leiden` not only preserves the properties of Leiden and achieves comparable modularity quality with Leiden, but also runs faster than state-of-the-art Leiden community maintenance algorithms. In future work, we plan to extend our algorithm to handle directed graphs and also evaluate our algorithm in a distributed environment.

REFERENCES

[1] 2020. A single-cell transcriptomic atlas characterizes ageing tissues in the mouse. *Nature* 583, 7817 (2020), 590–595.

[2] Edo M Airoldi, David Blei, Stephen Fienberg, and Eric Xing. 2008. Mixed membership stochastic blockmodels. *Advances in neural information processing systems* 21 (2008).

[3] Arash A Amini, Aiyou Chen, Peter J Bickel, and Elizaveta Levina. 2013. Pseudo-likelihood methods for community detection in large sparse networks. (2013).

[4] Abdelouahab Amira, Abdelouahid Derhab, Elmouatez Billah Karbab, and Omar Nouali. 2023. A survey of malware analysis using community detection algorithms. *Comput. Surveys* 56, 2 (2023), 1–29.

[5] LN Fred Ana and Anil K Jain. 2003. Robust data clustering. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, Vol. 2. IEEE, II–II.

[6] Thomas Aynaud and Jean-Loup Guillaume. 2010. Static community detection algorithms for evolving networks. In *8th international symposium on modeling and optimization in mobile, ad hoc, and wireless networks*. IEEE, 513–519.

[7] Thomas Aynaud and Jean-Loup Guillaume. 2011. Multi-step community detection and hierarchical time segmentation in evolving networks. In *Proceedings of the 5th SNA-KDD workshop*, Vol. 11.

[8] Trygve E Bakken, Nikolas L Jorstad, Qiwen Hu, Blue B Lake, Wei Tian, Brian E Kalmbach, Megan Crow, Rebecca D Hodge, Fenna M Krienen, Staci A Sorensen, et al. 2021. Comparative cellular analysis of motor cortex in human, marmoset and mouse. *Nature* 598, 7879 (2021), 111–119.

[9] Vandana Bhatia and Rinkle Rani. 2018. Dfuzzy: a deep learning-based fuzzy clustering model for large graphs. *Knowledge and Information Systems* 57 (2018), 159–181.

[10] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[11] Stefan Boettcher and Allon G Percus. 2002. Optimization with extremal dynamics. *complexity* 8, 2 (2002), 57–62.

[12] Biao Cai, Yanpeng Wang, Lina Zeng, Yanmei Hu, and Hongjun Li. 2020. Edge classification based on convolutional neural networks for community detection in complex network. *Physica A: statistical mechanics and its applications* 556 (2020), 124826.

[13] Tanmoy Chakraborty, Ayushi Dalmia, Animesh Mukherjee, and Niloy Ganguly. 2017. Metrics for community analysis: A survey. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–37.

[14] Qing Chen, Sven Helmer, Oded Lachish, and Michael Bohlen. 2022. Dynamic spanning trees for connectivity queries on fully-dynamic undirected graphs. (2022).

[15] Jiafeng Cheng, Qianqian Wang, Zhiqiang Tao, Deyan Xie, and Quanxue Gao. 2021. Multi-view attribute graph convolution networks for clustering. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence*. 2973–2979.

[16] Yun Chi, Xiaodan Song, Dengyong Zhou, Koji Hino, and Belle L Tseng. 2007. Evolutionary spectral clustering by incorporating temporal smoothness. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 153–162.

[17] Yun Chi, Xiaodan Song, Dengyong Zhou, Koji Hino, and Belle L Tseng. 2009. On evolutionary spectral clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 3, 4 (2009), 1–30.

[18] Wen Haw Chong and Loo Nin Teow. 2013. An incremental batch technique for community detection. In *Proceedings of the 16th international conference on information fusion*. IEEE, 750–757.

[19] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 70, 6 (2004), 066111.

[20] Mário Cordeiro, Rui Portocarrero Sarmento, and Joao Gama. 2016. Dynamic community detection in evolving networks using locality modularity optimization. *Social Network Analysis and Mining* 6 (2016), 1–20.

[21] Ganqu Cui, Jie Zhou, Cheng Yang, and Zhiyuan Liu. 2020. Adaptive graph encoder for attributed graph embedding. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 976–985.

[22] Siemon C de Lange, Marcel A de Reus, and Martijn P van den Heuvel. 2014. The Laplacian spectrum of neural networks. *Frontiers in computational neuroscience* 7 (2014), 189.

[23] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, Vol. 96. 226–231.

[24] Shaohua Fan, Xiao Wang, Chuan Shi, Emiao Lu, Ken Lin, and Bai Wang. 2020. One2multi graph autoencoder for multi-view graph clustering. In *proceedings of the web conference 2020*. 3070–3076.

[25] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 155–169.

[26] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. 2020. Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding. In *Proceedings of the web conference 2020*. 2331–2341.

[27] László Gadár and János Abonyi. 2024. Explainable prediction of node labels in multilayer networks: a case study of turnover prediction in organizations. *Scientific Reports* 14, 1 (2024), 9036.

[28] Michael S Haney, Róbert Pálovics, Christy Nicole Munson, Chris Long, Patrik K Johansson, Oscar Yip, Wentao Dong, Eshaan Rawat, Elizabeth West, Johannes CM Schlachetzki, et al. 2024. APOE4/4 is linked to damaging lipid droplets in Alzheimer's disease microglia. *Nature* 628, 8006 (2024), 154–161.

[29] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. 1983. Stochastic blockmodels: First steps. *Social networks* 5, 2 (1983), 109–137.

[30] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)* 48, 4 (2001), 723–760.

[31] Ruiqi Hu, Shirui Pan, Guodong Long, Qinghua Lu, Liming Zhu, and Jing Jiang. 2020. Going deep: Graph convolutional ladder-shape networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 2838–2845.

[32] Xiao Huang, Jundong Li, and Xia Hu. 2017. Accelerated attributed network embedding. In *Proceedings of the 2017 SIAM international conference on data mining*. SIAM, 633–641.

[33] Yuting Jia, Qinqin Zhang, Weinan Zhang, and Xinbing Wang. 2019. Communitygan: Community detection with generative adversarial nets. In *The world wide web conference*. 784–794.

[34] Baoyu Jing, Chanyoung Park, and Hanghang Tong. 2021. Hdmi: High-order deep multiplex infomax. In *Proceedings of the web conference 2021*. 2414–2424.

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

[36] Brian Karrer and Mark EJ Newman. 2011. Stochastic blockmodels and community structure in networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 83, 1 (2011), 016107.

[37] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[38] Sadamori Kojaku, Giacomo Livan, and Naoki Masuda. 2021. Detecting anomalous citation groups in journal networks. *Scientific Reports* 11, 1 (2021), 14524.

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

[40] Ron Levie, Federico Monti, Xavier Bresson, and Michael M Bronstein. 2018. Cayleynets: Graph convolutional neural networks with complex rational spectral filters. *IEEE Transactions on Signal Processing* 67, 1 (2018), 97–109.

[41] Bentian Li, Dechang Pi, Yunxia Lin, and Lin Cui. 2021. DNC: A deep neural network-based clustering-oriented network embedding algorithm. *Journal of Network and Computer Applications* 173 (2021), 102854.

[42] Zhangtao Li and Jing Liu. 2016. A multi-agent genetic algorithm for community detection in complex networks. *Physica A: Statistical Mechanics and its Applications* 449 (2016), 336–347.

[43] Xujian Liang and Zhaoquan Gu. 2025. Fast think-on-graph: Wider, deeper and faster reasoning of large language model on knowledge graph. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 24558–24566.

[44] Chunxu Lin, YiXiang Fang, Yumao Xie, Yongming Hu, Yingqian Hu, and Chen Cheng. 2025. Efficient Maintenance of Leiden Communities in Large Dynamic Graphs (full version). https://anonymous.4open.science/r/HIT_Leiden-2DC1.

[45] Yu-Ru Lin, Yun Chi, Shenghuo Zhu, Hari Sundaram, and Belle L Tseng. 2008. Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. In *Proceedings of the 17th international conference on World Wide Web*. 685–694.

[46] Rik GH Lindeboom, Kaylee B Worlock, Lisa M Dratva, Masahiro Yoshida, David Scobie, Helen R Wagstaffe, Laura Richardson, Anna Wilbrey-Clark, Josephine L Barnes, Lorenz Kretschmer, et al. 2024. Human SARS-CoV-2 challenge uncovers local and systemic response dynamics. *Nature* 631, 8019 (2024), 189–198.

[47] Monika Litviňuková, Carlos Talavera-López, Henrike Maatz, Daniel Reichart, Catherine L Worth, Eric L Lindberg, Masatoshi Kanda, Krzysztof Polanski, Matthias Heinig, Michael Lee, et al. 2020. Cells of the adult human heart. *Nature* 588, 7838 (2020), 466–472.

[48] Fanzhen Liu, Zhao Li, Baokun Wang, Jia Wu, Jian Yang, Jiaming Huang, Yiqing Zhang, Weiqiang Wang, Shan Xue, Surya Nepal, et al. 2022. eRiskCom: an e-commerce risky community detection platform. *The VLDB Journal* 31, 5 (2022), 1085–1101.

[49] Fanzhen Liu, Jia Wu, Chuan Zhou, and Jian Yang. 2019. Evolutionary community detection in dynamic social networks. In *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–7.

[50] Yanbei Liu, Xiao Wang, Shu Wu, and Zhitao Xiao. 2020. Independence promoted graph disentangled networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4916–4923.

[51] Linhao Luo, Yixiang Fang, Xin Cao, Xiaofeng Zhang, and Wenjie Zhang. 2021. Detecting communities from heterogeneous graphs: A context path-based graph neural network model. In *Proceedings of the 30th ACM international conference on information & knowledge management*. 1170–1180.

[52] Aaron F McDaid, Derek Greene, and Neil Hurley. 2011. Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515* (2011).
[53] Xiangfeng Meng, Yunhai Tong, Xinhai Liu, Shuai Zhao, Xianglin Yang, and Shaohua Tan. 2016. A novel dynamic community detection algorithm based on modularity optimization. In *2016 7th IEEE international conference on software engineering and service science (ICSESS)*. IEEE, 72–75.
[54] Microsoft. 2025. GraphRAG: A Structured, Hierarchical Approach to Retrieval Augmented Generation. https://microsoft.github.io/graphrag/. Accessed: 2025-03-31.
[55] Ida Momennejad, Hosein Hasanbeig, Felipe Vieira Frujeri, WA Redmond, Hiteshi Sharma, Robert Ness, Nebojsa Jojic, Hamid Palangi, and Jonathan Larson. [n.d.]. Evaluating Cognitive Maps and Planning in Large Language Models with CogEval (Supplementary Materials). ([n. d.]).
[56] Mark EJ Newman. 2004. Fast algorithm for detecting community structure in networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 69, 6 (2004), 066133.
[57] Mark EJ Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 74, 3 (2006), 036104.
[58] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 103, 23 (2006), 8577–8582.
[59] Mark EJ Newman. 2013. Spectral methods for community detection and graph partitioning. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 88, 4 (2013), 042822.
[60] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.
[61] Nam P Nguyen, Thang N Dinh, Sindhura Tokala, and My T Thai. 2011. Overlapping communities in dynamic networks: their detection and mobile applications. In *Proceedings of the 17th annual international conference on Mobile computing and networking*. 85–96.
[62] Nam P Nguyen, Thang N Dinh, Ying Xuan, and My T Thai. 2011. Adaptive algorithms for detecting community structure in dynamic social networks. In *2011 Proceedings IEEE INFOCOM*. IEEE, 2282–2290.
[63] Alexandru Oarga, Matthew Hart, Andres M Bran, Magdalena Lederbauer, and Philippe Schwaller. 2024. Scientific knowledge graph and ontology generation using open large language models. In *AI for Accelerated Materials Design-NeurIPS 2024*.
[64] Shashank Pandit, Duen Horng Chau, Samuel Wang, and Christos Faloutsos. 2007. Netprobe: a fast and scalable system for fraud detection in online auction networks. In *Proceedings of the 16th international conference on World Wide Web*. 201–210.
[65] Songtao Peng, Jiaqi Nie, Xincheng Shu, Zhongyuan Ruan, Lei Wang, Yunxuan Sheng, and Qi Xuan. 2022. A multi-view framework for BGP anomaly detection via graph attention network. *Computer Networks* 214 (2022), 109129.
[66] Ganesan Ramalingam and Thomas Reps. 1996. On the computational complexity of dynamic graph problems. *Theoretical Computer Science* 158, 1-2 (1996), 233–277.
[67] Jörg Reichardt and Stefan Bornholdt. 2006. Statistical mechanics of community detection. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 74, 1 (2006), 016110.
[68] Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2021. Dynamic structural clustering on graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 1491–1503.
[69] Subhajit Sahu. 2024. DF Louvain: Fast Incrementally Expanding Approach for Community Detection on Dynamic Graphs. *arXiv preprint arXiv:2404.19634* (2024).
[70] Subhajit Sahu. 2024. A Starting Point for Dynamic Community Detection with Leiden Algorithm. *arXiv preprint arXiv:2405.11658* (2024).
[71] Subhajit Sahu, Kishore Kothapalli, and Dip Sankar Banerjee. 2024. Fast Leiden Algorithm for Community Detection in Shared Memory Setting. In *Proceedings of the 53rd International Conference on Parallel Processing*. 11–20.
[72] Arindam Sarkar, Nikhil Mehta, and Piyush Rai. 2020. Graph representation learning via ladder gamma variational autoencoders. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5604–5611.
[73] Akrati Saxena, Yulong Pei, Jan Veldsink, Werner van Ipenburg, George Fletcher, and Mykola Pechenizkiy. 2021. The banking transactions dataset and its comparative analysis with scale-free networks. In *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 283–296.
[74] Jiaxing Shang, Lianchen Liu, Xin Li, Feng Xie, and Cheng Wu. 2016. Targeted revision: A learning-based approach for incremental community detection in dynamic networks. *Physica A: Statistical Mechanics and its Applications* 443 (2016), 70–85.
[75] Jiaxing Shang, Lianchen Liu, Feng Xie, Zhen Chen, Jiajia Miao, Xuelin Fang, and Cheng Wu. 2014. A real-time detecting algorithm for tracking community structure of dynamic networks. *arXiv preprint arXiv:1407.2683* (2014).
[76] Oleksandr Shchur and Stephan Günnemann. 2019. Overlapping community detection with graph neural networks. *arXiv preprint arXiv:1909.12201* (2019).
[77] Stanislav Sobolevsky, Riccardo Campari, Alexander Belyi, and Carlo Ratti. 2014. General optimization technique for high-quality community detection in complex networks. *Physical Review E* 90, 1 (2014), 012811.
[78] Xing Su, Shan Xue, Fanzhen Liu, Jia Wu, Jian Yang, Chuan Zhou, Wenbin Hu, Cecile Paris, Surya Nepal, Di Jin, et al. 2022. A comprehensive survey on community detection with deep learning. *IEEE transactions on neural networks and learning systems* 35, 4 (2022), 4682–4702.
[79] Tencent. 2019. *Tencent Graph Computing (TGraph) Officially Open Sourced High-Performance Graph Computing Framework: Plato*. Accessed: 2025-04-17.
[80] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. 2019. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific reports* 9, 1 (2019), 1–12.
[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[82] Lewen Wang, Haozhe Zhao, Cunguang Feng, Weiqing Liu, Congrui Huang, Marco Santoni, Manuel Cristofaro, Paola Jafrancesco, and Jiang Bian. 2023. Removing camouflage and revealing collusion: Leveraging gang-crime pattern in fraudster detection. In *Proceedings of the 29th ACM SIGKDD conference on knowledge discovery and data mining*. 5104–5115.
[83] Shu Wang, Yixiang Fang, and Wensheng Luo. 2025. Searching and Detecting Structurally Similar Communities in Large Heterogeneous Information Networks. *Proceedings of the VLDB Endowment* 18, 5 (2025), 1425–1438.
[84] Xiao Wang, Nian Liu, Hui Han, and Chuan Shi. 2021. Self-supervised heterogeneous graph neural network with co-contrastive learning. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 1726–1736.
[85] Wei Xia, Qianqian Wang, Quanxue Gao, Xiangdong Zhang, and Xinbo Gao. 2021. Self-supervised graph convolutional network for multi-view clustering. *IEEE Transactions on Multimedia* 24 (2021), 3182–3192.
[86] Jierui Xie, Mingming Chen, and Boleslaw K Szymanski. 2013. LabelrankT: Incremental community detection in dynamic networks via label propagation. In *Proceedings of the workshop on dynamic networks management and mining*. 25–32.
[87] Jierui Xie and Boleslaw K Szymanski. 2013. Labelrank: A stabilized label propagation algorithm for community detection in networks. In *2013 IEEE 2nd Network Science Workshop (NSW)*. IEEE, 138–143.
[88] Jierui Xie, Boleslaw K Szymanski, and Xiaoming Liu. 2011. Slpa: Uncovering overlapping communities in social networks via a speaker-listener interaction dynamic process. In *2011 ieee 11th international conference on data mining workshops*. IEEE, 344–349.
[89] Yu Xie, Maoguo Gong, Shanfeng Wang, and Bin Yu. 2018. Community discovery in networks with deep sparse filtering. *Pattern Recognition* 81 (2018), 50–59.
[90] Lantian Xu, Dong Wen, Lu Qin, Ronghua Li, Ying Zhang, and Xuemin Lin. 2024. Constant-time Connectivity Querying in Dynamic Graphs. *Proceedings of the ACM on Management of Data* 2, 6 (2024), 1–23.
[91] Rongbin Xu, Yan Che, Xinmei Wang, Jianxiong Hu, and Ying Xie. 2020. Stacked autoencoder-based community detection method via an ensemble clustering framework. *Information sciences* 526 (2020), 151–165.
[92] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 824–833.
[93] Liang Yang, Xiaochun Cao, Dongxiao He, Chuan Wang, Xiao Wang, and Weixiong Zhang. 2016. Modularity based community detection with deep learning.. In *IJCAI*, Vol. 16. 2252–2258.
[94] Zhao Yang, René Algesheimer, and Claudio J Tessone. 2016. A comparative analysis of community detection algorithms on artificial networks. *Scientific reports* 6, 1 (2016), 30750.
[95] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600* (2018).
[96] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. 2016. Image captioning with semantic attention. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4651–4659.
[97] Neda Zarayeneh and Ananth Kalyanaraman. 2021. Delta-screening: a fast and efficient technique to update communities in dynamic graphs. *IEEE transactions on network science and engineering* 8, 2 (2021), 1614–1629.
[98] Fangyuan Zhang and Sibo Wang. 2022. Effective indexing for dynamic structural graph clustering. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2908–2920.
[99] Meng Zhang, Xingjie Pan, Won Jung, Aaron R Halpern, Stephen W Eichhorn, Zhiyun Lei, Limor Cohen, Kimberly A Smith, Bosiljka Tasic, Zizhen Yao, et al. 2023. Molecularly defined and spatially resolved cell atlas of the whole mouse brain. *Nature* 624, 7991 (2023), 343–354.
[100] Tianqi Zhang, Yun Xiong, Jiawei Zhang, Yao Zhang, Yizhu Jiao, and Yangyong Zhu. 2020. CommDGI: community detection oriented deep graph infomax. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1843–1852.

[101] Xiaotong Zhang, Han Liu, Xiao-Ming Wu, Xianchao Zhang, and Xinyue Liu. 2021. Spectral embedding network for attributed graph clustering. *Neural Networks* 142 (2021), 388–396.

[102] Yao Zhang, Yun Xiong, Yun Ye, Tengfei Liu, Weiqiang Wang, Yangyong Zhu, and Philip S Yu. 2020. SEAL: Learning heuristics for community detection with generative adversarial networks. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 1103–1113.

[103] Han Zhao, Xu Yang, Zhenru Wang, Erkun Yang, and Cheng Deng. 2021. Graph debiased contrastive learning with joint representation clustering.. In *IJCAI*. 3434–3440.

[104] Yingli Zhou, Qingshuo Guo, Yi Yang, Yixiang Fang, Chenhao Ma, and Laks Lakshmanan. 2024. In-depth Analysis of Densest Subgraph Discovery in a Unified Framework. *arXiv preprint arXiv:2406.04738* (2024).

[105] Di Zhuang, J Morris Chang, and Mingchen Li. 2019. DynaMo: Dynamic community detection by incrementally maximizing modularity. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2019), 1934–1945.

APPENDIX

A EXTRA LEMMA

To assist in proving lemmas in Appendix, we present Lemma 8.

Lemma 8. *If a vertex set U is γ-connected, then for each vertex $v_i \in U$, there exist at least one vertex subset $U_i \subseteq U$ such that U_i is γ-connected and $U_i \setminus v_i$ remains γ-connected.*

Proof. We prove by contradiction. Assume there exists a vertex $v \in U$ such that for any γ-connected vertex subset $U_j \subseteq U$ containing v, $(U_j \setminus v) \cup v$ is not γ-connected. Under this assumption, U can not be a γ-connected, since any subset $U_j^{'}$ in U that cannot be obtained from v, implying no γ-order of U includes v. This contradicts the original assumption that U is γ-connected. Therefore, the lemma holds. □

B PROOF OF LEMMAS

B.1 Proof of Lemma 4

Proof. Given a γ-order, let U_i and U_j be two γ-connected vertex subsets in sub-community S, such that $v_i \in U_i$, $v_j \in U_j$, and both $U_i \setminus v_i$ and $U_j \setminus v_j$ are γ-connected prior to the graph update. We analyze the modularity gain $\Delta M(v_i \to \emptyset, \gamma)$, which denotes the modularity gain of moving v_i from U_i to $\emptyset$, whose calculation follows the same formula as the standard modularity gain.

Case 1: v_i was inserted into S after v_j. According to Lemma 8, there exists an intermediate γ-connected set U_i such that $v_j \in U_i$. Let $M_{old}(v_i \to \emptyset, \gamma)$ denote the modularity gain before the deletion. After the deletion, the new modularity gain $M_{new}(v_i \to \emptyset, \gamma)$ formulates:

$$\begin{aligned}\Delta M_{new}(v_i \to \emptyset, \gamma) = &-\frac{E(v_i, U_i \setminus v_i) - 2 \cdot w}{2 \cdot (m - w)} \\ &+ \frac{\gamma \cdot (d(v_i) - w) \cdot (d(U_i) - d_w(v_i) - w)}{(2 \cdot m - 2 \cdot w)^2}.\end{aligned} \tag{3}$$

In practice, the size of batch change is usually sufficiently smaller than the weight sum of edges m [69, 97]. We thus use m instead of $m + w$ in the denominators. Moreover, the value of γ is not set to a large number (e.g., 0.5, 1, 4, or 32, as shown in [46]). Since $d(U_i)$ is typically much smaller than m, we have $\gamma \cdot d(U_i) \ll m$. We thus approximate:

$$\begin{aligned}\Delta M_{new}(v_i \to \emptyset, \gamma) \approx &-\frac{E(v_i, U_i \setminus v_i) - 2 \cdot w}{2 \cdot m} \\ &+ \frac{\gamma \cdot (d(v_i) - w) \cdot (d(U_i) - d(v_i) - w)}{(2 \cdot m)^2} \\ &= \Delta M_{old}(v_i \to \emptyset, \gamma) \\ &+ \frac{w}{m} + \frac{\gamma \cdot w \cdot (d(U_i) - w)}{(2 \cdot m)^2} \\ &\approx \Delta M_{old}(v_i \to \emptyset, \gamma) + \frac{w}{m}.\end{aligned} \tag{4}$$

$M_{new}(v_i \to \emptyset, \gamma)$ could be positive because of the significant positive factor $\frac{w}{m}$. Thus, v_i has an incentive to leave U_i.

Case 2: v_i was inserted into S before v_j. In this case, we have $v_i \in U_i$, $v_j \notin U_i$, and the edge deletion does not affect intra-edges within U_i. The new modularity becomes:

$$\begin{aligned}\Delta M_{new}(v_i \to \emptyset, \gamma) = &-\frac{E(v_i, U_i \setminus v_i)}{2 \cdot (m - w)} \\ &+ \frac{\gamma \cdot (d(v_i) - w) \cdot (d(U_i) - d(v_i))}{(2 \cdot m - 2 \cdot w)^2} \\ &\approx -\frac{E(v_i, U_i / v_i)}{2 \cdot m} \\ &+ \frac{\gamma \cdot (d(v_i) - w) \cdot (d(U_i) - d(v_i))}{(2 \cdot m)^2} \\ &\approx \Delta M_{old}(v_i \to \emptyset, \gamma) \leq 0.\end{aligned} \tag{5}$$

Hence, v_i has no incentive to leave the sub-community.

The same analysis can be carried out for the vertex v_j. If the vertex v_j is inserted after the vertex v_i, v_j may be affected by the deletion within the sub-community S.

Generalization to other vertices. Let v_k (v_k not an endpoint) be a vertex within the sub-community S and v_l a vertex outside of S. There exist γ-connected subsets $U_k \subseteq S$ and $U_l \subseteq \mathbb{C} \setminus S$ such that $v_k \in U_k$, $v_l \in U_l$, and both $U_k \setminus v_k$ and $U_l \setminus v_l$ are also γ-connected prior to the update. After the edge deletion, their new modularity gains satisfy:

$$\Delta M_{new}(v_k \to \emptyset, \gamma) \approx \Delta M_{old}(v_k \to \emptyset, \gamma) \leq 0, \tag{6}$$

$$\Delta M_{new}(v_l \to \emptyset, \gamma) \approx \Delta M_{old}(v_l \to \emptyset, \gamma) \leq 0 \tag{7}$$

Thus, the intra-sub-community edge deletion has a negligible effect on the modularity of other vertices.

Conclusively, when an intra-sub-community edge (v_i, v_j) is removed, the endpoint that was added to the sub-community later is likely to be removed.

□

B.2 Proof of Lemma 5

Proof. We adopt the same notations as in the proof of Lemma 4, with the exception that v_k now denotes a vertex residing in the same sub-community as either v_i or v_j. Based on this setup, we observe the following approximations for modularity gain after the edge deletion:

$$\Delta M_{new}(v_i \to \emptyset, \gamma) \approx \Delta M_{old}(v_i \to \emptyset, \gamma) \leq 0, \tag{8}$$

$$\Delta M_{new}(v_j \to \emptyset, \gamma) \approx \Delta M_{old}(v_j \to \emptyset, \gamma) \leq 0, \tag{9}$$

$$\Delta M_{new}(v_k \to \emptyset, \gamma) \approx \Delta M_{old}(v_k \to \emptyset, \gamma) \leq 0, \tag{10}$$

$$\Delta M_{new}(v_l \rightarrow \emptyset, \gamma) \approx \Delta M_{old}(v_l \rightarrow \emptyset, \gamma) \leq 0, \tag{11}$$

In each case, the modularity gain remains non-positive, indicating no incentive for the vertices to leave their current sub-communities. Therefore, the deletion of a cross-sub-community edge is unlikely to alter the sub-community memberships of any vertex. □

B.3 Proof of Lemma 6

PROOF. We use the same notation in the proof of Lemma 4. For vertices directly involved, v_i and v_j, the new modularity gain can be approximated as:

$$\Delta M_{\text{new}}(v_i \rightarrow \emptyset, \gamma) \lesssim \Delta M_{\text{old}}(v_i \rightarrow \emptyset, \gamma) \leq 0, \tag{12}$$

$$\Delta M_{\text{new}}(v_j \rightarrow \emptyset, \gamma) \lesssim \Delta M_{\text{old}}(v_j \rightarrow \emptyset, \gamma) \leq 0. \tag{13}$$

For other vertices v_k, within the same sub-community, and v_l, in different sub-communities, the insertion does not change their immediate local structure.

$$\Delta M_{new}(v_k \rightarrow \emptyset, \gamma) \approx \Delta M_{old}(v_k \rightarrow \emptyset, \gamma) \leq 0, \tag{14}$$

$$\Delta M_{new}(v_l \rightarrow \emptyset, \gamma) \approx \Delta M_{old}(v_l \rightarrow \emptyset, \gamma) \leq 0, \tag{15}$$

Hence, the insertion is unlikely to affect sub-community memberships. □

B.4 Proof of Lemma 7

PROOF. The removal of vertex v_i from sub-community S can be interpreted as the deletion of all intra-sub-community edges $(v_i, v_{i'})$ for each neighbor $v_{i'} \in S$. According to Lemma 4, such deletions can disrupt the γ-connectivity of affected substructures. Thus, the neighbors of vertex v_i, which are inserted into the sub-community S after v_i, have an incentive to change their sub-community memberships. □