Dynamic Structural Clustering Unleashed: Flexible Similarities, Versatile Updates and for All Parameters

Anonymous Author(s)

Abstract

In this paper, we study structural clustering on graphs in *dynamic* scenarios, where the graphs can be updated by *arbitrary* insertions and deletions of edges. The goal is to support *efficient* structural clustering computation for any clustering parameters ε and μ given *on the fly*. A recent state-of-the-art (SOTA) method achieves an *expected* per-update time of $O(\log^2 n)$, ensuring a ρ-absolute-approximation with high success probability for a graph with n vertices. However, this method suffers from several limitations: (i) it supports Jaccard similarity measurement *only*, while the Cosine and Dice similarities are also commonly used in structural clustering; (ii) it makes an assumption that the edge insertions and deletions occur uniformly at random within the neighborhoods of the edges' end-vertices. We propose an algorithm, named *VD-STAR*, which not only removes all the aforementioned limitations of SOTA but also improves the per-update time bound, by shaving a logarithmic factor, to $O(\log n)$ *amortized in expectation* while achieving the same approximation guarantee. We further design two variants of our algorithm to enhance the empirical performance. Experimental results show that our algorithms consistently outperform the SOTA competitors by up to 9,315 times in update time across nine real datasets.

ACM Reference Format:
Anonymous Author(s). 2025. Dynamic Structural Clustering Unleashed: Flexible Similarities, Versatile Updates and for All Parameters. In *Proceedings of 2025 International Conference on Management of Data (SIGMOD '25).* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Graph clustering is a fundamental problem that aims to group similar vertices of a graph into clusters. Different clustering schemes have been proposed, such as modularity-based clustering [15] and spectral clustering [22]. In this paper, we focus on a popular scheme named *Structural Clustering* [23] that groups the vertices based on their *structural similarity* in the graph. An important feature of Structural Clustering is that it identifies not only clusters of the vertices but also *different roles* for the vertices (i.e., cores, hubs, and outliers) in the clustering result.

Structural Clustering. Given an undirected graph $G = \langle V, E \rangle$, a specified *similarity measurement* σ between the neighborhoods of vertices, a similarity threshold parameter $0 < \varepsilon \leq 1$ and an integer parameter $\mu \geq 1$, the process of structural clustering starts with identifying a set of special vertices known as the *core vertices*. A core vertex is a vertex $u \in V$ with at least μ *similar neighbors*, and a similar neighbor is a neighbor vertex of u with a similarity score $\geq \varepsilon$. The core vertices, along with the edges to their similar core neighbors, collectively form connected components (CCs), where each CC serves as a *primitive cluster*. The non-core vertices are then assigned to the corresponding primitive clusters of their similar core neighbors. For each vertex $u \in V$ which *does not* belong to any cluster, if u connects to neighbors from two or more clusters, then u is categorized as a *hub*, in a sense that u serves as a bridge connecting multiple clusters. Otherwise, u is considered as an *outlier*.

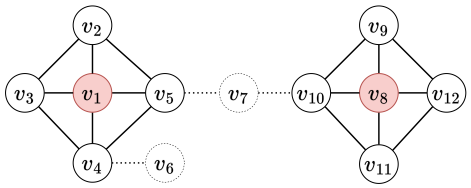


Figure 1: A Structural Clustering Example ($\varepsilon = 0.5$ and $\mu = 5$)

Figure 1 shows an example of structural clustering results with $\varepsilon = 0.5$, $\mu = 4$, using Jaccard similarity as the similarity measurement between vertices. Vertices v_1 and v_8 are core vertices (solid lines indicate edges connecting similar neighbors) and hence, each of them itself forms a primitive cluster. And vertices $\{v_1, v_2, v_3, v_4, v_5\}$ is a cluster, while $\{v_8, v_9, v_{10}, v_{11}, v_{12}\}$ is another. Both v_7 and v_6 belong to none of the clusters. However, v_7 is a hub as it connects to neighbors belonging to two different clusters, and v_6 is an outlier.

Applications. Structural clustering is widely applied in genomics and biomarker discovery [5], and over other biology data such as clinical data [13] and protein-protein interaction networks [14]. Another important application is community detection over social networks [6, 12, 18], where community structures can be unveiled, offering insights into the relationships and interactions among social network users. Similarly, in the context of web data [16, 17], structural clustering contributes to the identification and analysis of interconnected communities, enhancing the comprehension of web structures and user behaviors.

Related Work and Limitations. Structural clustering was first proposed by Xu *et al.* [23]. Since then, it has opened a line of follow-up. Among them, pSCAN [2] is the SOTA *exact* algorithm for *static* graphs G, where no updates to G are allowed. The running time complexity of pSCAN is bounded by $O(m^{1.5})$, where m is the number of edges. As shown by Chang *et al.* [2], this $O(m^{1.5})$ bound is indeed worst-case optimal.

When the graphs are *dynamic*, where the graphs can be updated by insertions or deletions of edges, structural clustering becomes even more challenging. GS*-Index [21] is the SOTA for *Dynamic Structural Clustering*, which can return the exact clustering result

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '25, June 22–27, 2025, Berlin, Germany
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Table 1: Comparison with SOTA Methods, where n is the number of vertices, m is the number of edges, $d_{\max}$ is the maximum degree, and m_{cr} is the number of edges in the clustering result graph

Algorithm Features		**Our *VD-STAR***	BOTBIN [24]	DynELM [19]	GS*-Index [21]
Update support	Update running time	$O(\log n)$ amortized expected	$O(\log^2 n)$ expected	$O(\log^2 n)$ amortized	$O(d_{\max}^2 \log n)$
	Support arbitrary updates	✓	assuming random updates	✓	✓
Similarity measurement	Jaccard similarity	✓	✓	✓	✓
	Cosine similarity	✓			✓
	Dice similarity	✓			✓
Query running time for ε and μ given on the fly		$O(m_{cr})$	$O(m_{cr})$	$O(m \log^2 n)$ running from scratch	$O(m_{cr})$

with respect to parameters ε and μ given on the fly for each query. However, it takes $O(d_{\max}^2 \cdot \log n)$ worst-case time to process each update, where $d_{\max}$ is the maximum degree and n is the number of vertices in the current graph.

DynELM [19] and BOTBIN [24] are two SOTA *approximate* algorithms, yet both of them can *only* work for Jaccard similarity. DynELM can process each update in $O(\log^2 n)$ amortized time for pre-specified parameters ε and μ. In contrast, BOTBIN supports queries with ε and μ given on the fly, and can process each update in $O(\log^2 n)$ time *in expectation* under an assumption that the updates are uniformly at random within each vertex's neighborhood. This assumption, however, may not always hold for real-world applications, e.g., people tend to follow big names in social networks.

Challenges. It is important yet remains challenging to solve approximate Dynamic Structural Clustering *beyond* Jaccard similarity, and for *arbitrary* updates (with *no* assumptions), particularly given that Cosine similarity is actually the preferred measurement by Xu *et al.* [23] when structural clustering was proposed and adopted as default in these follow-up work [2, 20, 21]. In particular, Xu *et al.* demonstrated that clustering results with Cosine similarity outperformed other methods, such as FastModularity [3].

Our Contributions. In this paper, we made these contributions.

- We propose a novel algorithm, called *VD-STAR*, which addresses all the aforementioned challenges, and hence, overcomes all the limitations of the SOTA algorithms, and most importantly, is even more efficient in processing updates! As shown in Table 1, our *VD-STAR*
  - supports all the three similarity measurements (Jaccard, Cosine, and Dice) suggested by Xu *et al.* [23];
  - returns a valid ρ-absolute-approximate clustering result with respect to the parameters ε and μ given on the fly, with high probability, for each query as efficiently as BOTBIN does;
  - processes each update in $O(\log n)$ amortized expected time without making any assumption on the update patterns; in other words, this not only improves the $O(\log^2 n)$ expected update bound of BOTBIN, but also supports *arbitrary* updates;
  - consumes space bounded by $O(n+m)$, i.e., linear to the size of the current graph, *at all times*.
- While the theoretical analysis is technical, our *VD-STAR* is surprisingly simple; it just needs to maintain and scan a number of sorted lists and hash tables. As a result, our *VD-STAR* can be easily implemented in practice.
- As a side product, we propose a unified framework (Algorithm 1) for GS*-Index, BOTBIN and our *VD-STAR*, with which all users need is just to implement the specified interfaces to obtain the algorithms. This also provides flexibility for users to swap the implementations between different algorithms to make their own "new" solutions. As we will see in Section 5, this is actually what we did to design the two variants of *VD-STAR*.
- We conduct extensive experiments on nine real-world graph datasets with up to 117 million edges to compare our algorithms with GS*-Index and BOTBIN, in terms of update efficiency (with varying update distributions), clustering quality (under different similarity measurements), and query efficiency. The experimental results show that our proposed algorithms outperform SOTA algorithms by up to 9,315× regarding update efficiency.

2 Preliminaries

2.1 Problem Formulation

Consider an undirected graph $G = \langle V, E \rangle$, where V is a set of n vertices and E is a set of m edges. Vertices $u \in V$ and $v \in V$ are *neighbors* if and only if there exists an edge $(u, v) \in E$. The *neighborhood* of u, denoted by $N(u)$, is the set of all u's neighbors, namely, $N(u) = \{v \in V | (u, v) \in E\}$, and the *degree* of u is defined to be $d_u = |N(u)|$. Moreover, we use $N[u] = N(u) \cup \{u\}$ to denote the *inclusive neighborhood* of u and let $n_u = |N[u]|$.

Similarity Measurement. The similarity between vertices u and v is denoted by $\sigma(u, v)$. Specifically, $\sigma(u, v) = 0$ if there is *no* edge between u and v; otherwise, depending on the application needs, $\sigma(u, v)$ is calculated as one of the following three popular similarity measurements, where $I(u, v) = |N[u] \cap N[v]|$ and $U(u, v) = |N[u] \cup N[v]| = n_u + n_v - I(u, v)$:

- *Jaccard similarity*: $\sigma(u, v) = \frac{I(u,v)}{U(u,v)} = \frac{I(u,v)}{n_u + n_v - I(u,v)}$, or
- *Cosine similarity*: $\sigma(u, v) = \frac{I(u,v)}{\sqrt{n_u \cdot n_v}}$, or
- *Dice similarity*: $\sigma(u, v) = \frac{I(u,v)}{(n_u + n_v)/2}$.

And $\sigma(u, v)$ can be computed in $O(1)$ time with $I(u, v)$, n_u and n_v.

Similar Neighbors, Edge Labels and Core Vertices. Given a *similarity threshold* $0 < \varepsilon < 1$, vertices u and v are *ε-similar neighbors* if $\sigma(u, v) \geq \varepsilon$. An edge (u, v) is labelled as an *ε-similar edge* if u and v are ε-similar neighbors; otherwise, it is labeled as a *ε-dissimilar edge*. A vertex u is a *(ε, μ)-core vertex* if u has *at least* μ ε-similar neighbors; otherwise, u is a *non-core vertex* with respect to ε and μ.

In the rest of this paper, when the context of the parameters ε and μ is clear, we use *sim-edges* to refer to ε-similar edges, and *core vertices* to refer to (ε, μ)-core vertices, respectively.

Core Sim-Graph. An edge (u, v) is a *core sim-edge* if (u, v) is a sim-edge and both u and v are core vertices. The *core sim-graph* of G is defined as $G_{\text{core}} = \langle V_{\text{core}}, E_{\text{core}} \rangle$, where V_{core} is the set of all the core vertices and E_{core} is the set of all core sim-edges.

Structural Clusters and the Clustering Result. Each connected component (CC) of G_{core} is defined as a *primitive (structural) cluster*.

Unpublished working draft. Not for distribution.

And each primitive cluster C, along with the set of all the *non-core* vertices that are similar neighbors of some core vertex in C, is defined as a *structural cluster* ("cluster" for short). The collection of all these clusters represents the *Structural Clustering Result* ("clustering result" for short) on G with respect to the parameters ε and μ.

Clustering Result Graph. Given ε and μ, let E_{cr} be the set of all the sim-edges that are incident on at least one *core* vertex. Denote by $G_{cr} = \langle V_{cr}, E_{cr} \rangle$ the induced sub-graph of G by E_{cr}, where V_{cr} is the set of all the end-vertices of the edges in E_{cr}. G_{cr} is called the *Clustering Result Graph* of G with respect to ε and μ. Moreover, we define $n_{cr} = |V_{cr}|$ and $m_{cr} = |E_{cr}|$.

Observation 1. *Given the clustering result graph G_{cr} with respect to the given parameters ε and μ, the structural clustering result on G can be computed in $O(m_{cr})$ time.*

Proof. By scanning G_{cr}, the core sim-graph G_{core} can be obtained, as G_{core} is a sub-graph of G_{cr}. As a result, all the primitive clusters (i.e., the connected components of G_{core}) can be computed in $O(|V_{\text{core}}| + |E_{\text{core}}|)$ time. Finally, for each edge $(u, v) \in E_{cr}$ that is incident on a non-core vertex v, assign v to the cluster of the core vertex u. The overall running time is bounded by $O(m_{cr})$. □

Problem Definition. We consider structural clustering on graph G which *evolves over time*. The problem is defined as follows.

Definition 2.1. Consider a pre-specified similarity measurement $\sigma(\cdot, \cdot)$ (either Jaccard, Cosine or Dice); given an undirected graph $G = \langle V, E \rangle$ that can be updated by *arbitrary* insertions or deletions of edges, the ***Dynamic Structural Clustering for All Parameters*** (**DynStrClu-AllPara**) problem asks to:

- (i) support updates *efficiently*, and
- (ii) return the clustering result upon request in $O(m_{cr})$ time with respect to the parameters $\varepsilon \in (0, 1)$ and $\mu \geq 1$ *given on the fly*.

Affecting Updates and Affected Edges. Observe that an update (either an insertion or a deletion) of an edge (u, v) changes the degrees of both u and v, and hence, the similarities of all the edges incident on u or v are *affected*. These edges incident on u or v are called *affected edges* of the update (u, v), and this update (u, v) is an *affecting update* to these edges. A main challenge in *DynStrClu-AllPara* is that for an udpate (u, v), there can be $O(d_u + d_v) \subseteq O(n)$ affected edges. Thus, maintaining the similarities of these affected edges can be expensive. As we shall see, how to overcome this technical difficulty is the main distinction among the state-of-the-art solutions and our proposed algorithms.

ρ-Absolute-Approximation. We exploit the notion of ρ-absolute approximation for *DynStrClu-AllPara*.

Definition 2.2 (ρ-Absolute-Approximation). Given a *constant* parameter $\rho \in (0, 1)$ and the similarity threshold parameter ε, the label of an edge (u, v) is decided as follows:

(1) if $\sigma(u, v) > \varepsilon + \rho$, (u, v) must be considered as similar;
(2) if $\sigma(u, v) < \varepsilon - \rho$, (u, v) must be considered as dissimilar;
(3) otherwise, i.e., $\varepsilon - \rho \leq \sigma(u, v) \leq \varepsilon + \rho$, (u, v) can be considered as either similar or dissimilar.

Once the edge labels are decided according to the above ρ-absolute-approximation, all the other definitions introduced in this section, immediately follow. Moreover, it is shown [24] that clustering result under the notion of ρ-absolute-approximation provides a "sandwich" guarantee on the result quality compared to the exact clustering result with the same parameters ε and μ.

Fact 1 ([19, 24]). *Given parameters ε, μ, and ρ, let $C_{\varepsilon,\mu}$ denote the exact clustering result, and let $C^{\rho}_{\varepsilon,\mu}$ denote the clustering result satisfying the ρ-absolute-approximation. We have the following properties:*

- *for every cluster $C_+ \in C_{\varepsilon+\rho,\mu}$, there is a cluster $\tilde{C} \in C^{\rho}_{\varepsilon,\mu}$ such that $C_+ \subseteq \tilde{C}$;*
- *for every cluster $\tilde{C} \in C^{\rho}_{\varepsilon,\mu}$, there is a cluster $C_- \in C_{\varepsilon-\rho,\mu}$ such that $\tilde{C} \subseteq C_-$.*

Proof. Let $G_{cr+} = \langle V_{cr+}, E_{cr+} \rangle$, $G_{cr-} = \langle V_{cr-}, E_{cr-} \rangle$, and $G_{cr_\rho} = \langle V_{cr_\rho}, E_{cr_\rho} \rangle$ be the clustering results graphs $C_{\varepsilon+\rho,\mu}$, $C_{\varepsilon-\rho,\mu}$, and $C^{\rho}_{\varepsilon,\mu}$, respectively for given μ, ε, and ρ. For an edge $(u, v) \in E_{cr+}$, we have $\sigma(u, v) \geq \varepsilon + \rho$, hence u and v are considered similar under the ρ-absolute-approximation. Additionally, as $(u, v) \in E_{cr+}$, at least one of u and v must be a core vertex in G_{cr+}. Without loss of generality, we assume u to be a core vertex in G_{cr+}. Then, u is also a core vertex in G_{cr_ρ} based on the ρ-absolute-approximation. This is because u can only have more similar neighbors under a more relaxed threshold. Therefore, (u, v) must be in E_{cr_ρ}, and hence $E_{cr+} \subseteq E_{cr_\rho}$. Symmetrically, for an edge $(u, v) \in E_{cr_\rho}$, it has $\sigma(u, v) \geq \varepsilon - \rho$, hence u and v are considered similar in G_{cr-}. Similarly, a core vertex in E_{cr_ρ} must be a core vertex in G_{cr-}, and we have $E_{cr_\rho} \subseteq E_{cr-}$.

Let V_+ and E_+ be the vertex and edge sets of $C_+ \in C_{\varepsilon+\rho,\mu}$, respectively. We have $E_+ \subseteq E_{cr+} \subseteq E_{cr_\rho}$ as verified above. Therefore, there is a cluster $\tilde{C} \in C^{\rho}_{\varepsilon,\mu}$ that contains all the vertex in V_+, and hence $C_+ \subseteq \tilde{C}$. This proves the first bullet point. For the second bullet point, let V_ρ and E_ρ be the vertex and edge sets of $\tilde{C} \in C^{\rho}_{\varepsilon,\mu}$, respectively. We have $E_\rho \subseteq E_{cr_\rho} \subseteq E_{cr-}$. Therefore, there exists a cluster $C_- \in C_{\varepsilon-\rho,\mu}$ that contains all the vertex in V_ρ, and hence $\tilde{C} \subseteq C_-$.

□

2.2 A Unified Algorithm Framework

For ease of presentation, we introduce a *unified algorithm framework* for solving *DynStrClu-AllPara*, which is shown in Algorithm 1. The SOTA exact and approximate algorithms discussed in this paper, as well as our solutions, can work under this framework. At a high level, these algorithms implement the following data structures:

- ***Sorted Neighbor Lists***: for each vertex $u \in V$, a *non-increasing sorted* list of u's neighbors by their similarities to u; with a slight abuse of notation, we simply use $N(u)$ to refer to this sorted list, and each neighbor v of u in this list is stored along with its similarity to u.
- ***EdgeSimStr***: a data structure for maintaining the (approximate) similarities for all the edges; there are five functions:
  - update((u, v), op): given an update of edge (u, v), where op indicates whether this is an insertion or a deletion, update the information maintained in *EdgeSimStr* accordingly;
  - insert((x, y)): insert an edge (x, y) to *EdgeSimStr*;
  - delete((x, y)): delete an edge (x, y) from *EdgeSimStr*;

Unpublished working draft. Not for distribution.

- find((u, v), op): return a set F of all the affected edges whose similarities are considered "invalid" and thus need to be re-computed;
- cal-sim((x, y)): given an edge (x, y), return $\sigma(x, y)$;

- ***CoreFindStr***: a data structure for finding core vertices; it has two functions:
  - update(u): given a vertex $u \in V$, update *CoreFindStr*;
  - find-core(ε, μ): return the set V_{core} of all the core vertices with respect to the given parameters ε and μ;

These modules aim to support The goal of maintaining these data structures is to answer each query with parameters ε and μ by outputting the corresponding clustering result in $O(|E_{cr}|)$ time.

All the algorithms discussed in this paper differ only in the implementations of *EdgeSimStr* and *CoreFindStr*, and hence, achieve different running time bounds on handling each update and query.

Running Time Analysis. Let $cost_{\text{EU}}$, $cost_{\text{EI}}$, $cost_{\text{ED}}$, $cost_{\text{EF}}$ and $cost_{\text{EC}}$ denote the running time cost of each invocation of the functions *update*, *insert*, *delete*, *find* and *cal-sim* in *EdgeSimStr*, respectively; and $cost_{\text{CF}}$ and $cost_{\text{CU}}$ denote the running time cost of the functions *find-core* and *update* in *CoreFindStr*, respectively.

Query Running Time. By Observation 1, the running time cost for each query is bounded by $O(cost_{\text{CF}} + m_{cr})$.

Per-Update Running Time. In the Update Procedure in Algorithm 1, Line 2 takes $O(cost_{\text{EU}})$ time and Lines 3-9 takes $O(cost_{\text{EC}} + cost_{\text{EI}} + cost_{\text{ED}} + \log n)$ time, where $O(\log n)$ is the maintenance cost for the sorted neighbor lists. Furthermore, the running time cost of Line 10 is bounded $O(cost_{\text{EF}})$ while that of Lines 11-15 is bounded by $O(|F| \cdot (cost_{\text{EC}} + cost_{\text{EI}} + cost_{\text{ED}} + \log n))$. Finally, Lines 13 - 14 can be performed in $O(|F| \cdot cost_{\text{CU}})$ because $|S| \in O(|F|)$. Summing all these up, the overall running time of each update is bounded by

$$O\left(cost_{\text{EU}} + cost_{\text{EF}} + (|F| + 1) \cdot (cost_{\text{EC}} + cost_{\text{EI}} + cost_{\text{ED}} + \log n + cost_{\text{CU}})\right).$$

With this algorithm framework, one can just focus on the implementations for *EdgeSimStr* and *CoreFindStr* of different algorithms. Substituting the corresponding costs to the above analysis, the query and per-update running time bounds follow.

2.3 A SOTA Exact Algorithm

The *GS*-Index* [21] is a state-of-the-art exact algorithm for *DynStrClu-AllPara*. It implements *EdgeSimStr* and *CoreFindStr* as follows.

The Implementation of *EdgeSimStr*. For each vertex $u \in V$, the *EdgeSimStr*, maintain d_u, the degree of u, and $I(u, x)$, the *intersection size* of $N[u]$ and $N[x]$, for each neighbor $x \in N(u)$. Clearly, the simialrity $\sigma(u, v)$ can be computed with $I(u, v)$, d_u and d_v in $O(1)$ time. And the functions are implemented as follows:

- update((u, v), op): maintain the counters d_u and $I(u, x)$ for each $x \in N(u)$ according to the given update. Perform the same maintenance symmetrically for the end-vertex v. Therefore, $cost_{\text{EU}} \in O(d_u + d_v) \subseteq O(d_{\max})$, where $d_{\max}$ is the largest degree of G.
- find((u, v), op): return the set of all the affected edges of the update (u, v). Hence, $|F| \in O(d_u + d_v)$ and hence, $cost_{\text{EF}} \in O(d_{\max})$.
- cal-sim((x, y)): compute $\sigma(u, v)$ with $I(u, v)$, d_u and d_v. Thus, $cost_{\text{EC}} \in O(1)$.
- neither insert((x, y)) nor delete((x, y)) is used in GS*-Index; hence, $cost_{\text{EI}} = 0$ and $cost_{\text{ED}} = 0$.

Algorithm 1: A Unified Algorithm Framework

```
1  Update Procedure:
       Input: an update of (u, v) flagged by op ∈ {ins, del}
2      EdgeSimStr.update((u, v), op);
3      if op == ins then
4          EdgeSimStr.cal-sim((u, v));
5          EdgeSimStr.insert((u, v));
6          insert (u, v) to E and maintain N(u) and N(v);
7      else
8          EdgeSimStr.delete((u, v));
9          remove (u, v) from E and maintain N(u) and N(v);
       // identify all the "invalid" edges
10     F ← EdgeSimStr.find((u, v), op);
11     for each (x, y) ∈ F do
12         EdgeSimStr.delete((x, y));
13         EdgeSimStr.cal-sim((x, y));
14         EdgeSimStr.insert((x, y));
15         maintain N(x) and N(y);
16     S ← {end-vertices of all the edges in F} ∪ {u, v};
17     for each x ∈ S do
18         CoreFindStr.update(x);
19 Query Procedure:
       Input: parameters 0 < ε < 1 and μ ≥ 1
       Output: a clustering result with respect to ε and μ
20     V_core ← ∅, E_cr ← ∅;
21     V_core ← CoreFindStr.find-core(ε, μ);
22     for each u ∈ V_core do
23         for each v ∈ N(u) do
24             if σ(u, v) ≥ ε, add (u, v) to E_cr; otherwise, break;
25     return the clustering result from G_cr induced by E_cr;
```

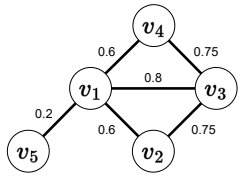
The Implementation of *CoreFindStr*. The GS*-Index implements the *CoreFindStr* as a *μ-Table*, denoted by T, which is essentially an array of $d_{\max}$ sorted lists. Specifically, for integer $1 \le i \le d_{\max}$, $T[i]$ stores a *sorted* list of all the vertices u (with degrees $d_u \ge i$) in a *non-increasing* order by the i^{th} largest similarity of u to its neighbors, denoted by $\varepsilon_{u,i}$.

- find-core(ε, μ): retrieve all the core vertices by scanning the vertices in the sorted list $T[\mu]$ until the first vertex x such that $\varepsilon_{x,\mu} < \varepsilon$ is met or the entire list has been retrieved. $cost_{\text{CF}} \in O(|V_{\text{core}}| + 1)$.
- update(x): maintain the sorted lists $T[1], \cdots, T[d_x]$ for x accordingly. This takes $cost_{\text{CU}} \in O(d_{\max} \cdot \log n)$ time, because the maintenance on each of these sorted lists takes $O(\log n)$ time.

A Running Example. Figure 2b presents a running example of a μ-Table. Consider $T[3]$ as an example, where $\varepsilon_{v_3,3} = 0.75$ indicating that v_3 has at least three similar neighbors for any $\varepsilon \le 0.75$. When querying with $\mu = 3$ and $\varepsilon = 0.7$, one can quickly scan $T[3]$ and determine that v_3 is the only core vertex since $\varepsilon_{v_3,3} > 0.7$.

Fact 2. *The GS*-Index can answer each query in $O(m_{cr})$ time and can handle each update in $O(d_{\max}^2 \cdot \log n)$ time with space consumption bounded by $O(n + m)$ at all times.*

Unpublished working draft. Not for distribution.



(a) A graph example

$T[1] : \varepsilon_{v_1,1} = 1, \varepsilon_{v_2,1} = 1, \varepsilon_{v_3,1} = 1, \varepsilon_{v_4,1} = 1, \epsilon_{v_5,1} = 1$
$T[2] : \varepsilon_{v_1,2} = 0.8, \varepsilon_{v_3,2} = 0.8, \varepsilon_{v_2,2} = 0.75, \varepsilon_{v_4,2} = 0.75, \varepsilon_{v_5,2} = 0.2$
$T[3] : \varepsilon_{v_3,3} = 0.75, \varepsilon_{v_1,3} = 0.6, \varepsilon_{v_2,3} = 0.6$
$T[4] : \varepsilon_{v_3,4} = 0.75, \varepsilon_{v_1,4} = 0.6$
$T[5] : \varepsilon_{v_1,5} = 0.2$

(b) GS*-Index example

$\Delta = 0.2$
$[0.0, 0.2) : \mu_{v_1,0} = 5, \mu_{v_3,0} = 4, \mu_{v_4,0} = 3, \mu_{v_2,0} = 3, \mu_{v_5,0} = 2$
$[0.2, 0.4) : \mu_{v_1,1} = 5, \mu_{v_3,1} = 4, \mu_{v_4,1} = 3, \mu_{v_2,1} = 3, \mu_{v_5,1} = 2$
$[0.4, 0.6) : \mu_{v_1,2} = 4, \mu_{v_3,2} = 4, \mu_{v_4,2} = 3, \mu_{v_2,2} = 3$
$[0.6, 0.8) : \mu_{v_1,3} = 4, \mu_{v_3,3} = 4, \mu_{v_4,3} = 3, \mu_{v_2,3} = 3$
$[0.8, 1.0) : \mu_{v_1,4} = 2, \mu_{v_3,4} = 2$

(c) BOTBIN example

Figure 2: Index schema examples

2.4 A SOTA Approximate Algorithm

While the space consumption and the query time complexity of GS*-Index are good, Unfortunately, the $O(d_{\max}^2 \cdot \log n)$ per-update time of GS*-Index is prohibitive as $d_{\max}$ can be as large as n. To remedy this, BOTBIN [24] adopts the notion of *ρ-absolute-approximation* for *DynStrClu-AllPara*. It improves the per-update cost from $O(d_{\max}^2 \cdot \log n)$ to roughly $O(\log^2 n)$ *in expectation* assuming that the updates are uniformly at random within the neighborhood of each vertex and for Jaccard similarity *only*. However, BOTBIN only works for Jaccard similarity measurement and this expected update bound only holds under an assumption that the updates on G are *uniformly at random*.

The Implementation of *EdgeSimStr*. BOTBIN maintains a *bottom-k signature*, denoted by $s(u)$, for each vertex $u \in V$ for some integer parameter k. An ρ-absolute-approximate Jaccard similarity, denoted by $\tilde{\sigma}(u, v)$, between any two vertices u and v can be computed with their signatures $s(u)$ and $s(v)$. Specifically, it first generates and stores a fixed random permutation π of V. For each vertex $u \in V$, if $d_u \geq k$, then the *signature* of u, denoted by $s(u)$, is the set of the k smallest neighbors in $N[u]$ according to the permutation order π; otherwise, $s(u) = N[u]$. With the bottom-k signatures, an ρ-absolute-approximate Jaccard similarity, denoted by $\tilde{\sigma}(u, v)$, between any two vertices u and v can be computed with $s(u)$ and $s(v)$. is computed as $\tilde{\sigma}(u, v) = \frac{|s(u) \cap s(v) \cap s(\{u,v\})|}{k}$, where $s(\{u, v\})$ is the k smallest vertices in $s(u) \cup s(v)$ according to the permutation π; if $|s(u) \cup s(v)| < k$, then $s(\{u, v\}) = s(u) \cup s(v)$.

- update((u, v), op): update the signatures $s(u)$ and $s(v)$ with respect to the update of (u, v) accordingly; it is known that this can be achieved in $O(\log n)$ time. Thus, $cost_{\text{EU}} \in O(\log n)$.
- find((u, v), op): if the signature $s(u)$ is changed due to this given update of (u, v), add all the edges incident on u to F; and perform the same symmetrically for v. As a result, either $s(u)$ or $s(v)$ changes, then $cost_{\text{EF}} \in O(d_{\max})$; otherwise, $cost_{\text{EF}} \in O(1)$.
- cal-sim((x, y)): return $\tilde{\sigma}(x, y)$ as the similarity of x and y. This can be done in $O(k)$ time.
- neither insert((x, y)) or delete((x, y)) is used in BOTBIN; hence, $cost_{\text{EI}} = 0$ and $cost_{\text{ED}} = 0$.

The Implementation of *CoreFindStr*. BOTBIN implements *CoreFindStr* as an array, called *Δ-Table* and denoted by T_Δ, of $\lceil \frac{1}{\Delta} \rceil$ sorted list of vertices, where $0 < \Delta < 1$ is a *constant*. Specifically, the parameter Δ partitions value range of ε into $\lceil \frac{1}{\Delta} \rceil$ intervals, where the i^{th} interval is $[i\Delta, (i+1)\Delta)$ for $i = 0, \cdots, \lceil \frac{1}{\Delta} \rceil - 1$. $T_\Delta[i]$ is a sorted list of all vertices $u \in V$ in *non-increasing* order by $\mu_{u,i}$, where $\mu_{u,i}$ is the number of neighbors of u have similarities to u at least $i\Delta$.

- find-core(ε, μ): identify $i^* = \lfloor \varepsilon / \Delta \rfloor$; retrieve all the core vertices by scanning and reporting the vertices in the sorted list $T_\Delta[i^*]$ until the first vertex x such that $\mu_{x,i^*} > \mu$ or the entire list has been retrieved. Thus, $cost_{\text{CF}} \in O(|V_{\text{core}}| + 1)$. Note that Δ-Table introduces an additive Δ error to the overall approximation.
- update(x): maintain the $\lceil \frac{1}{\Delta} \rceil \in O(1)$ sorted lists in T_Δ for x. This takes $cost_{\text{CU}} \in O(\lceil \frac{1}{\Delta} \rceil \cdot \log n) = O(\log n)$ time.

A Running Example. Figure 2c shows a running example of Δ-Table with $\Delta = 0.2$. Take $[0.2, 0.4)$ for instance, where $\mu_{v_1,1} = 5$ meaning that v_1 has 5 similar neighbors when $0.2 \leq \varepsilon < 0.4$. Given a query with $\varepsilon = 0.3$ and $\mu = 4$, we know that v_1 and v_3 are core vertices as only $\mu_{v_1,1}$ and $\mu_{v_3,1}$ are greater than or equal to 4 in the corresponding sorted linked list.

Theoretical Analysis. Zhang and Wang [24] showed that by setting $k \in O(\frac{1}{\rho^2} \cdot \log(n \cdot M)) = O(\log(n \cdot M))$ and $\Delta = \frac{1}{2}\rho$, BOTBIN guarantees to return a valid ρ-absolute-approximate clustering result with high probability, specially at least $1 - \frac{1}{n}$, for every query. This guarantee holds for up to M updates.

Query Running Time. Since $cost_{\text{CF}}$ in bounded by $O(|V_{\text{core}}| + 1)$, the running time of each query is bounded by $O(m_{cr})$.

Per-Update Running Time. Substitute the running time cost of each function in the above implementation to Expression (2.2), the per-update running time of BOTBIN is thus bounded by $O(|F| \cdot \log(n \cdot M)) \subseteq O(d_{\max} \cdot \log(n \cdot M))$. While this per-update bound is still prohibitive, Zhang and Wang [24] proved that, as long as the M updates happen *uniformly at random* in the neighborhood of each vertex, then the signature of a vertex u changes with probability $\frac{k}{d_u}$. Therefore, the expected size of the "invalid" affected edge set F is bounded by $O(\frac{k}{d_u} \cdot d_u) = O(\log(n \cdot M))$. As a result, the expected per-update cost is bounded by $O(\log^2(n \cdot M))$.

Space Consumption. It can be verified that the space consumption of BOTBIN is bounded by $O(n + m)$ at all times.

Fact 3. *BOTBIN can return a valid ρ-absolute-approximate clustering result (under Jaccard similarity* only*) in $O(m_{cr})$ time with high probability, at least $1 - \frac{1}{n}$, for each query, and this holds for up to M updates. Furthermore, it can handle each update in $O(d_{\max} \cdot \log(n \cdot M))$ time and the space consumption is bounded by $O(n + m)$ at all times.*

When the M updates occur uniformly at random in the neighborhood of each vertex, then the per-update time is bounded by $O(\log^2(n \cdot M))$ in expectation.

Remark. As discussed earlier, BOTBIN has two main limitations:

- BOTBIN can handle Jaccard similarity only.
- The $O(\log^2(n \cdot M))$ per-update expected running time bound of BOTBIN holds only for random updates. As a result, for repeated

Unpublished working draft. Not for distribution.

insertion and deletion of a *critical* edge which changes the signatures of its two end-vertices, BOTBIN has to pay $O(d_{\max} \cdot \log(n \cdot M))$ cost for each such update.

3 Our Versatile DynStrClu Algorithm

Next, we introduce our solution, called *Versatile Dynamic Structural Clustering* (*VD-STAR*), which not only overcomes *all* the aforementioned limitations of BOTBIN, but also improves the per-update running time cost to $O(\log n + \log M)$ amortized in expectation.

Let n_0 and m_0 be the number of vertices and edges in the graph at the current moment. Without loss of generality, we assume that the number of updates $M \leq n_0^2$ since now, because, otherwise, when $M = n_0^2$, we can rebuild everything from scratch in $O((n_0 + m_0 + M) \cdot \log n) = O(M \cdot \log n)$ expected time. Hence, each of such M updates is charged a $O(\log n)$ amortized expected cost which does not affect the per-update running time bound. Moreover, it is worth mentioning that the randomness in the running time of *VD-STAR* only comes from the use of hash tables. In this and the next section, we prove this theorem:

Theorem 3.1. *Our* VD-STAR *algorithm supports all three similarity measurements (Jaccard, Cosine, Dice). It can return an ρ-absolute-approximate clustering result with high probability, at least $1 - \frac{1}{n}$, for each query, and can handle each update in $O(\log n)$ amortized expected time. The space consumption of* VD-STAR *is bounded by $O(n + m)$ at all times.*

Our *VD-STAR* also works under the unified algorithm framework (Algorithm 1). Specifically, the implementation of *VD-STAR* for *CoreFindStr* follows that of BOTBIN, i.e., the Δ-Table. Therefore, we will focus on our implementation for *EdgeSimStr*.

3.1 Update Affordability and Background

We adopt the notion of ρ-absolute-approximation. Thanks to the approximation, *VD-STAR* is allowed to just maintain *approximate* rather than exact similarities. It thus creates room for efficiency improvements. First, the similarity can now be estimated (within an ρ-absolute error) via certain sampling techniques efficiently. Second, each edge can now afford a certain number of affecting updates before its estimated similarity exceeds the ρ-absolute-error range from the last estimation. Such a number of affecting updates is called the *update affordability* of the edge.

Definition 3.2 (Update Affordability). For any edge (u, v), consider the moment when an $\frac{1}{2}\rho$-absolute-approximate similarity $\tilde{\sigma}(u, v)$ is just computed; the *update affordability* of (u, v) is a *lower bound* on the number of affecting updates, denoted by $\tau(u, v)$, such that $\tilde{\sigma}(u, v)$ remains a *valid* ρ-absolute approximation to the exact similarity, in the sense that $|\tilde{\sigma}(u, v) - \sigma(u, v)| \leq \rho$, at any moment within $\tau(u, v)$ affecting updates.

The concept of update affordability was first proposed and exploited in [19] for solving the *DynStrClu* problem for Jaccard similarity with *pre-specified* parameters ε and μ. We borrow this concept and extend it to *DynStrClu-AllPara* for *versatile* similarity measurements (Jaccard, Cosine, and Dice). As we show in Section 4.4, the extension of the concept of update affordability from Jaccard to Cosine similarity is challenging and non-trivial, mainly because of the *non-linear* denominator $\sqrt{n_u \cdot n_v}$ in Cosine similarity. To avoid distraction, we defer the proof of the following claim to Section 4.4.

Claim 1. *For any edge (u, v) with $n_u \leq n_v$, the update affordability $\tau(u, v) \geq \frac{1}{4}\rho^2 n_v \in \Omega(d_{\max}(u, v))$ holds for any of Jaccard, Cosine and Dice similarity measurements, where $d_{\max}(u, v) = \max\{d_u, d_v\}$.*

By the definition of update affordability, when an approximate similarity $\tilde{\sigma}(u, v)$ of an edge (u, v) is just computed, $\tilde{\sigma}(u, v)$ will remain valid for the next at least $\tau(u, v) - 1$ affecting updates. And hence, in order to ensure a valid ρ-absolute-approximate similarity for every edge, one may need to *re-compute* $\tilde{\sigma}(u, v)$ *no later* than the arrival of the $\tau(u, v)^{\text{th}}$ affecting update for each $(u, v) \in E$.

However, observe that (i) the update affordability can be different for different edges, and (ii) an update of edge (u, v) would "consume" one affordability for each of its $d_u + d_v \in O(d_{\max})$ affected edges. Therefore, simply tracking the "remaining" update affordability for each affected edge can be as expensive as $\Omega(d_{\max})$. It is challenging to identify the set F of all invalid edges when an update arrives, without touching each of the affected edges.

Ruan *et al.* [19] overcome this technical challenge for Jaccard similarity by adopting the *Distributed Tracking* technique [4, 8, 10] to track the *exact moment* when the $\tau(u, v)^{\text{th}}$ affecting update for each edge occurs. They proved that their algorithm can achieve an $O(\log^2 n)$ amortized time for processing each update. Next, we show a *simpler yet more efficient* solution to identify the invalid edge set F just in $O(1)$ amortized expected time for each update.

3.2 Our Implementation of *EdgeSimStr*

Rationale of Our Algorithm. The basic idea of our solution for identifying invalid edges is as follows. For each edge $(u, v) \in E$, once $\tilde{\sigma}(u, v)$ is just computed, we compute its update affordability $\tau(u, v)$. Instead of tracking the exact moment when the $\tau(u, v)^{\text{th}}$ affecting update arrives, our algorithm aims to just identify an *arbitrary* moment when there have been at least $\frac{1}{4}\lfloor\tau(u, v)\rfloor_2$ affecting updates, where $\lfloor\tau(u, v)\rfloor_2$ is the *largest power-of-two* integer that is no more than $\tau(u, v)$, namely, $\lfloor\tau(u, v)\rfloor_2 = 2^{\lfloor\log_2 \tau(u,v)\rfloor}$. And such a moment is called a *checkpoint moment* of edge (u, v). Clearly, $\lfloor\tau(u, v)\rfloor_2 \geq \frac{1}{2}\tau(u, v)$. When a checkpoint moment of (u, v) is identified, there must have been at least $\frac{1}{8}\tau(u, v) \in \Omega(\tau(u, v))$ affecting updates, which is already "good enough" for our theoretical analysis.

To capture the checkpoint moments, our algorithm, for each edge (u, v), allocates an *affordability quota*, denoted by $q(u, v) = \frac{1}{4}\lfloor\tau(u, v)\rfloor_2$, to the vertices u and v. Once an arbitrary moment when at least $q(u, v)$ affecting updates incident on either u or v are "observed" since the quota is allocated, (u, v) is then reported as an invalid edge. The challenge is how to capture a checkpoint moment for each edge (u, v) before its ρ-absolute-approximate $\tilde{\sigma}(u, v)$ becomes invalid, without touching each of the affected edges for every affecting update.

The Data Structure for *EdgeSimStr*. For each $u \in V$, *VD-STAR* maintains the following information for *EdgeSimStr*:

- a counter c_u that records the number of affecting updates incident on u up to date; initially, $c_u \leftarrow 0$;
- a *sorted bucket linked list* $\mathcal{B}(u)$, where:
  - each bucket B_i has a *unique index* i (for $0 \leq i \leq \lceil\log_2 n\rceil$);

Unpublished working draft. Not for distribution.

Algorithm 2: Our Implementation of *EdgeSimStr*.insert

Input: an insertion of edge (u, v) to *EdgeSimStr*

1 $\tau(u, v) \leftarrow \frac{1}{4}\rho^2 \max\{n_u, n_v\}$;
2 $q(u, v) \leftarrow \frac{1}{4} \cdot \lfloor \tau(u, v) \rfloor_2$;
3 $i \leftarrow \log_2(q(u, v))$;
4 **if** B_i *does not exist in* $\mathcal{B}(u)$ **then**
5 create B_i and insert B_i to $\mathcal{B}(u)$;
6 set $\bar{c}_u(B_i) \leftarrow c_u$;
7 insert v to B_i;
8 perform the above steps for v symmetrically;

Algorithm 3: Our Implementation of *EdgeSimStr*.delete

Input: a deletion of edge (u, v) from *EdgeSimStr*

1 remove v from its corresponding bucket B_i;
2 **if** B_i *becomes empty* **then**
3 remove B_i from $\mathcal{B}(u)$;
4 perform the above steps for v symmetrically;

- bucket B_i stores all the neighbors $w \in N(u)$ such that the affordability quota $q(u, w) = 2^i$;
- all the *non-empty* buckets B_i (which contain at least one neighbor $w \in N(u)$) are materialized in the sorted linked list $\mathcal{B}(u)$ in an *increasing order* by their indices i.
- each non-empty bucket $B_i \in \mathcal{B}(u)$ maintains a counter $\bar{c}_u(B_i)$ that records the counter value c_u of u when B_i is last visited; initially, $\bar{c}_u(B_i)$ is set as the value of c_u when B_i is materialized and added to $\mathcal{B}(u)$;

Implementation of *EdgeSimStr*.update. In our *VD-STAR*, the update function of *EdgeSimStr* just increases the counters c_u and c_v by one, respectively, i.e., $c_u \leftarrow c_u + 1$ and $c_v \leftarrow c_v + 1$, recording that there is one more affecting update on them.

Implementation of *EdgeSimStr*.insert. The detailed implementation is shown in Algorithm 2. To insert an edge (u, v) to *EdgeSimStr*, our algorithm first computes the affordability quota $q(u, v)$, and then inserts v (resp., u) into the corresponding bucket in $\mathcal{B}(u)$ (resp., $\mathcal{B}(v)$). If the bucket does not exist, then a bucket is created and inserted into the sorted bucket linked list accordingly.

Implementation of *EdgeSimStr*.delete. This function removes v (resp., u) from its corresponding bucket in $\mathcal{B}(u)$ (resp., $\mathcal{B}(v)$). If the bucket becomes empty, then the bucket is removed from the bucket list. The pseudo code is shown in Algorithm 3.

Implementation of *EdgeSimStr*.find. Algorithm 4 gives implementation details. Observe that all the neighbors w of u are stored in a sorted list of *power-of-two* buckets of their corresponding affordability quotas. When the counter c_u is increased by one, it suffices to scan the sorted bucket list to check all the non-empty buckets B_i such that the current c_u has passed across their corresponding power-of-two values 2^i, because they were last visited when $c_u = \bar{c}_u(B_i)$ (see Line 4 in Algorithm 4). For each of such buckets B_i, our algorithm reports and adds the edge (u, w) to F for each $w \in B_i$ such that w is visited in B_i for the *second time*. The same process is performed for v symmetrically. The correctness of this implementation is proved in Section 4.1.

Algorithm 4: Our Implementation of *EdgeSimStr*.find

Input: either an insertion or a deletion of edge (u, v)
Output: a set F of potentially invalid edges

1 $F \leftarrow \emptyset$;
2 $B_i \leftarrow$ the first bucket in $\mathcal{B}(u)$, where i is the index of B;
3 **while** B_i *is not NULL* **do**
4 **if** $\lfloor \frac{c_u}{2^i} \rfloor > \lfloor \frac{\bar{c}_u(B_i)}{2^i} \rfloor$ **then**
  // check this bucket B_i
5 **for** *each* $w \in B_i$ **do**
6 **if** w *is visited in* B_i *for the second time* **then**
7 add (u, w) to F;
8 **else**
9 flag w as it has been visited for once;
10 $\bar{c}_u(B_i) \leftarrow c_u$; $B_i \leftarrow B_i$.next;
11 **else**
12 stop the scan of $\mathcal{B}(u)$ and break;
13 perform the steps from Line 2 for v symmetrically;
14 **return** F *as the set of invalid edges*

Algorithm 5: Our Implementation of *EdgeSimStr*.cal-sim

Input: an edge (x, y)
Output: an $\frac{1}{2}\rho$-absolute-approximate similarity $\tilde{\sigma}(x, y)$

1 **if** $n_x \le \frac{1}{4}\rho^2 n_y$ *or* $n_y \le \frac{1}{4}\rho^2 n_x$ **then**
2 **return** $\tilde{\sigma}(x, y) = 0$;
3 $L \leftarrow$ the number of samples as required by Lemma 4.2;
4 $X \leftarrow 0$;
5 **for** $i = 1, 2, \cdots, L$ **do**
6 flip a coin z such that $\Pr[z = 1] = \frac{n_x}{n_x + n_y}$ and $\Pr[z = 0] = \frac{n_x}{n_x + n_y}$;
7 **if** $z = 1$ **then**
8 uniformly at random pick a vertex $w \in N[x]$;
9 **else**
10 uniformly at random pick a vertex $w \in N[y]$;
11 **if** $w \in N[x] \cap N[y]$ **then**
12 $X \leftarrow X + 1$;
13 $\bar{X} \leftarrow X/L$;
14 **return**

$$\tilde{\sigma}(x, y) = \begin{cases} \frac{\bar{X}}{2-\bar{X}} & \text{for Jaccard similarity} \\ \frac{n_x + n_y}{2\sqrt{n_x \cdot n_y}} \cdot \bar{X} & \text{for Cosine similarity} \\ \bar{X} & \text{for Dice similarity} \end{cases}$$

Implementation of *EdgeSimStr*.cal-sim. See Algorithm 5.

A Running Example. Figure 3 shows a running example of the maintenance of our implementation for *EdgeSimStr*. At the current status, the affecting update counter of u, $c_u = 15$ and there are three non-empty buckets B_2, B_3 and B_5 in the sorted bucket list $\mathcal{B}(u)$, where $\bar{c}_u(B_2) = 12$ indicates that when B_2 was last visited, the value of the counter c_u was 12. Moreover, there are two neighbors w_1 and w_2 in B_2, where w_1 has not yet been visited while w_2 has been visited for once. When an update incident on u occurs, c_u is increased by

Unpublished working draft. Not for distribution.

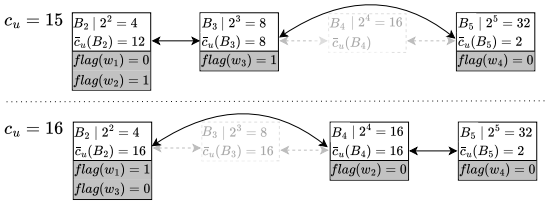


Figure 3: A Running Example of Our *EdgeSimStr*

one to $c_u = 16$ and then *EdgeSimStr*.find (Algorithm 4) is invoked and it scans $\mathcal{B}(u)$ from the first bucket B_2. Since $\lfloor \frac{16}{4} \rfloor > \lfloor \frac{12}{4} \rfloor$ (Line 4), the contents of B_2 are checked, where the flag of w_1 is set to 1 indicating that now w_1 has been visited once, while the edge (u, w_2) is added to F because w_2 is visited for the second time now. Finally, $\bar{c}_u(B_2) \leftarrow 16$ recording that the "time" when B_2 was last visited. This completes the process for B_2. As $\lfloor \frac{16}{8} \rfloor > \lfloor \frac{8}{8} \rfloor$, similarly, (u, w_3) is added to F and $\bar{c}_u(B_3) \leftarrow 16$. The algorithm stops the scanning at B_5 because $\lfloor \frac{16}{32} \rfloor = \lfloor \frac{2}{32} \rfloor$. Next, *EdgeSimStr*.delete (Algorithm 3) is invoked for (u, w_2) and (u, w_3) in F, and it removes w_1 and w_2 from B_2 and B_3, respectively (Line 1). As B_3 becomes empty, it is then removed from $\mathcal{B}(u)$ (Lines 2-3). After the re-calculation of the similarities for (u, w_2) and (u, w_3) with *EdgeSimStr*.cal-sim (Algorithm 5), *EdgeSimStr*.insert (Algorithm 2) is invoked to insert w_2 and w_3 to buckets B_4 and B_2 in $\mathcal{B}(u)$, respectively.

4 Theoretical Analysis

In this section, we prove the correctness of *VD-STAR*, analyze the amortized per-update running time and the space consumption. Putting these results together constitutes a proof for Theorem 3.1.

4.1 Correctness

Theorem 4.1. *Before and after any update,* VD-STAR *maintains a* proper *ρ-absolute-approximate similarity $\tilde{\sigma}(u, v)$ for every edge $(u, v) \in E$ with high probability at least $1 - \frac{1}{n}$.*

To prove Theorem 4.1, it suffices to show these two lemmas:

Lemma 4.2. *By setting $L = \frac{1}{2r^2} \ln(4n^4)$, where $r = \frac{1}{4}\rho$ for Jaccard, $r = \frac{1}{4}\rho^2$ for Cosine and $r = \frac{1}{2}\rho$ for Dice similarity, the approximate similarity $\tilde{\sigma}(u, v)$ returned by Algorithm 5 satisfies $|\tilde{\sigma}(u, v) - \sigma(u, v)| \leq \frac{1}{2}\rho$ with probability at least $1 - \frac{1}{2n^4}$.*

Lemma 4.3. *For any $(u, v) \in E$, its approximate similarity $\tilde{\sigma}(u, v)$ must be recomputed by Algorithm 5 before its update affordability $\tau(u, v)$ is fully consumed, that is, before the arrival of its $\tau(u, v)^{th}$ affecting update, since $\tilde{\sigma}(u, v)$ was last computed.*

Proof of Theorem 4.1. Suppose Lemmas 4.2 and 4.3 hold; by the definition of update affordability, we have that $\tilde{\sigma}(u, v)$ is a correct ρ-absolute-approximation of $\sigma(u, v)$ before and after any update for all edges $(u, v) \in E$. To see the success probability, as each update can affect at most $2n$ edges, it can trigger at most $2n$ invocations of Algorithm 5. Moreover, there are at most $M \leq n^2$ updates. Therefore, Algorithm 5 is invoked for at most $2n^3$ times. According to Lemma 4.2, each invocation fails with probability at most $\frac{1}{2n^4}$. Thus, the whole process succeeds with probability at least $1 - \frac{1}{n}$. □

Proof of Lemma 4.2. Consider an edge $(u, v) \in E$; without loss of generality, we assume that $n_u \leq n_v$.

Observation 2. *For any edge (u, v) with $n_u = \beta \cdot n_v$, where $0 < \beta \leq 1$, we have:*

- $Jaccard(u, v) = \frac{I(u,v)}{n_u + n_v - I(u,v)} \leq \frac{\beta \cdot n_v}{n_v} = \beta$;
- $Cosine(u, v) = \frac{I(u,v)}{\sqrt{n_u \cdot n_v}} \leq \frac{n_u}{\sqrt{1/\beta \cdot n_u}} = \sqrt{\beta}$;
- $Dice(u, v) = \frac{I(u,v)}{(n_u + n_v)/2} \leq \frac{\beta \cdot n_v}{n_v/2} = 2\beta$.

Substituting $\beta = \frac{1}{4}\rho^2$ to Observation 2, that is, $n_u \leq \frac{1}{4}\rho^2 n_v$, then the Jaccard, Cosine and Dice similarities of (u, v) are all $\leq \frac{1}{2}\rho$ for any constant $0 \leq \rho \leq 1$. Therefore, for any of the above similarity measurements, $\tilde{\sigma}(u, v) = 0$ is a correct $\frac{1}{2}\rho$-absolute-approximate similarity, thus, Lines 1-2 in Algorithm 5 are correct.

Next, we consider the case that $n_v \geq n_u > \frac{1}{4}\rho^2 n_v$ holds. In fact, a proof of this lemma for Jaccard similarity is given in [19] by Ruan *et al.*. We extend their proof to Cosine and Dice similarity. For completeness, we prove all of them in the following.

Let $X_i \in \{0, 1\}$ be a random variable for the i^{th} iteration in the for-loop in Lines 5 - 12 in Algorithm 5. Specifically, $X_i = 1$ if X is increased by one in Line 12; otherwise, $X_i = 0$. Therefore, $\Pr[X_i = 1] = \Pr[X_i = 1 \wedge z = 1] + \Pr[X_i = 0 \wedge z = 0] = \frac{n_u}{n_u + n_v} \cdot \frac{I(u,v)}{n_u} + \frac{n_v}{n_u + n_v} \cdot \frac{I(u,v)}{n_v} = \frac{2I(u,v)}{n_u + n_v}$. Furthermore, since $\bar{X} = \frac{X}{L} = \frac{\sum_{i=1}^{L} X_i}{L}$, we have the expectation $E[\bar{X}] = E[X_i] = \frac{2 \cdot I(u,v)}{n_u + n_v}$. Thus, we have the following for each of the similarity measurements.

For Jaccard similarity, we have $Jaccard(u, v) = \frac{E[\bar{X}]}{2 - E[\bar{X}]}$, and by Line 15 , $\tilde{\sigma}(u, v) = \frac{\bar{X}}{2 - \bar{X}}$. Thus, $\Pr[|\tilde{\sigma}(u, v) - Jaccard(u, v)| > \frac{1}{2}\rho] = \Pr[\frac{2 \cdot |\bar{X} - E[\bar{X}]|}{(2 - \bar{X})(2 - E[\bar{X}])} > \frac{1}{2}\rho] \leq \Pr[|\bar{X} - E[\bar{X}]| > \frac{1}{4}\rho]$, where the last inequality is by both $\bar{X}$ and $E[\bar{X}]$ are values in $[0, 1]$.

For Cosine similarity, we have $Cosine(u, v) = \frac{n_u + n_v}{2\sqrt{n_u \cdot n_v}} \cdot E[\bar{X}]$, and by Line 16, $\tilde{\sigma}(u, v) = \frac{n_u + n_v}{2\sqrt{n_u \cdot n_v}} \cdot \bar{X}$. Thus, $\Pr[|\tilde{\sigma}(u, v) - Cosine(u, v)| > \frac{1}{2}\rho] = \Pr[\frac{n_u + n_v}{2\sqrt{n_u \cdot n_v}} \cdot |\bar{X} - E[\bar{X}]| > \frac{1}{2}\rho] = \Pr[|\bar{X} - E[\bar{X}]| > \frac{2\sqrt{n_u \cdot n_v}}{n_u + n_v} \cdot \frac{1}{2}\rho] \leq \Pr[|\bar{X} - E[\bar{X}]| > \frac{2\sqrt{1/4\rho^2 n_v \cdot n_v}}{2 \cdot n_v} \cdot \frac{1}{2}\rho] = \Pr[|\bar{X} - E[\bar{X}]| > \frac{1}{4}\rho^2]$, where the last inequality is by the fact that $n_v \geq n_u > \frac{1}{4}\rho^2 n_v$.

For Dice similarity, we have $Dice(u, v) = E[\bar{X}]$ and by Line 17, $\tilde{\sigma}(u, v) = \bar{X}$. Therefore, $\Pr[|\tilde{\sigma}(u, v) - Dice(u, v)| > \frac{1}{2}\rho] = \Pr[|\bar{X} - E[\bar{X}]| > \frac{1}{2}\rho]$.

According to the Hoeffding Bound [7], by setting $L = \frac{1}{2 \cdot r^2} \ln \frac{2}{\delta}$, we have $\Pr[|\bar{X} - E[\bar{X}] > r] \leq \delta$. As a result, by setting $\delta = \frac{1}{2n^4}$, $r_j = \frac{1}{4}\rho$, $r_c = \frac{1}{4}\rho^2$, and $r_d = \frac{1}{2}\rho$, respectively for Jaccard, Cosine and Dice similarities, we can get the corresponding number of samples L to achieve $\tilde{\sigma}(u, v)$ being a correct $\frac{1}{2}\rho$-absolute approximation to $\sigma(u, v)$ with high probability at least $1 - \frac{1}{2n^4}$. □

Proof of Lemma 4.3. Recall that for each edge (u, v) right after $\tilde{\sigma}(u, v)$ is computed, (u, v) allocates an affordability quota $q(u, v) = \frac{1}{4}\lfloor \tau(u, v) \rfloor_2$ to an entry in a bucket B_i with index $i = \log_2 q(u, v)$ in both the sorted linked bucket lists $\mathcal{B}(u)$ and $\mathcal{B}(v)$. According to Algorithm 4, (u, v) is reported as an invalid edge in F when the entry in either the bucket in $\mathcal{B}(u)$ or $\mathcal{B}(v)$ is visited for the second time. As a result, the entry of edge (u, v) can be checked for at most three times in total, because at that time, the entry in either bucket must be checked for twice. Moreover, since each checking of the

Unpublished working draft. Not for distribution.

bucket B_i is triggered by at most $q(u,v)$ affecting updates, there can be at most $3 \cdot q(u,v) + q(u,v) - 1 < \lfloor \tau(u,v) \rfloor_2 \leq \tau(u,v)$ affecting updates happened. Lemma 4.3 thus follows. □

By Theorem 4.1 and the fact that *VD-STAR* adopts the Δ-Table for *CoreFindStr*, the following theorem immediately follows, which completes the correctness proof for *VD-STAR*.

Theorem 4.4. VD-STAR *returns a* $(\rho + \Delta)$*-absolute-approximate clustering result, with high probability at least* $1 - \frac{1}{n}$*, for any query with respect to the given parameters* ε *and* μ*.*

Remark. Given any constant *target overall approximation* parameter ρ^*, by setting $\rho = \Delta = \frac{1}{2}\rho^*$, *VD-STAR* can achieve ρ^*-absolute approximation without affecting its theoretical bounds.

4.2 Running Time Analysis

Query Running Time. As our *VD-STAR* adopts the Δ-Table technique for *CoreFindStr*, the query running time bound follows immediately from the analysis in Section 2. Thus, we have:

Lemma 4.5. VD-STAR *can answer each query in* $O(m_{cr})$ *time.*

The Maintenance Cost of an Edge. We first analyze the *maintenance cost* of each edge (u,v), denoted by $\ell(u,v)$, between two *consecutive* approximation similarity calculations for (u,v). Consider the moment when the similarity of an edge (u,v) needs to be computed; according to Algorithm 1, the maintenance for (u,v) involves the following operations:

- a similarity calculation (Algorithm 5) which takes $cost_{\text{EC}}$;
- an invocation of Algorithm 3 to remove the "old" quota entries of (u,v) from the buckets in $\mathcal{B}(u)$ and $\mathcal{B}(v)$; this takes $cost_{\text{ED}}$;
- an invocation of Algorithm 2 to insert the "updated" quota entries of (u,v) to buckets in $\mathcal{B}(u)$ and $\mathcal{B}(v)$; this takes $cost_{\text{EI}}$;
- the maintenance of the sorted neighbor lists of u and v due to the change of $\tilde{\sigma}(u,v)$; this maintenance takes $O(\log n)$ time;
- the maintenance of Δ-Table for u and v due to the change of their sorted neighbor lists; as discussed in Section 2, this cost is bounded by $O(\frac{1}{\Delta} \cdot \log n) = O(\log n)$ since Δ is a constant;
- at most three times of visits of the entries of (u,v) in the corresponding buckets before getting reported as an invalid edge; this cost is just $O(1)$.

Summing these costs up, the maintenance cost of (u,v) is:

$$\ell(u,v) \in O(cost_{\text{EC}} + cost_{\text{ED}} + cost_{\text{EI}} + \log n)\,. \tag{1}$$

Next, we analyse $cost_{\text{EC}}$, $cost_{\text{ED}}$ and $cost_{\text{EI}}$, respectively.

For the cost of similarity calculation, $cost_{\text{EC}}$, by Algorithm 5, by Lemma 4.2, we know that $L \in O(\log n)$ samples suffice. Each sample needs to check if a neighbor w is in $N[u] \cap N[v]$. By maintaining a hash table of $N[u]$ and $N[v]$, each of this checking can be performed in $O(1)$ expected time. Therefore, $cost_{\text{EC}}$ is bounded by $O(L) = O(\log n)$ in expectation.

To bound the costs $cost_{\text{EI}}$ and $cost_{\text{ED}}$ of Algorithms 2 and 3, observe that inserting and removing an entry from a bucket can be done in $O(1)$ time. This can be achieved simply by recording the locations (e.g., the indices in arrays) of the entries in the corresponding buckets. The remaining cost are from the operations on the sorted bucket lists $\mathcal{B}(u)$ and $\mathcal{B}(v)$ which include: (i) checking if a bucket exists or not, (ii) inserting a new bucket, and (iii) removing an existing bucket. According to the following Fact 4, each of this operation can be performed in $O(1)$ expected time. And therefore, $cost_{\text{EI}} + cost_{\text{ED}}$ is bounded by $O(1)$ in expectation.

Fact 4 ([25]). *The sorted linked list* $\mathcal{B}(u)$ *can be maintained with* $O(|\mathcal{B}(u)|)$ *space and support the following in* $O(1)$ *expected time:*

- *an insertion or deletion of a bucket to or from* $\mathcal{B}(u)$*, and*
- *return the pointer of the largest bucket* $B_i \in \mathcal{B}(u)$ *with index* $i \leq j$*, for any given integer index* $0 \leq j \leq \lceil \log_2 n \rceil$*.*

Putting all the above cost bounds to Expression (1), we thus have:

Lemma 4.6. *The maintenance cost of each edge* (u,v) *between two consecutive similarity calculations of it,* $\ell(u,v)$*, is bounded by* $O(\log n)$ *in expectation.*

Amortized Per-Update Cost. Next, we analyze the amortized running time for each update. Observe that, for an update of edge (u,v), according to Algorithm 1, the running time cost of processing this update consists of:

- a maintenance cost of $\ell(u,v)$ for the update;
- a cost of Algorithm 4, $cost_{\text{EF}}$, to find a set F of invalid edges;
- a maintenance cost of $\ell(x,y)$, for each edge $(x,y) \in F$.

By Lemma 4.6, the update cost of an edge (u,v) is bounded by $O(\log n + cost_{\text{EF}} + |F| \cdot \log n)$ in expectation. As in the worst case, the number of invalid edges, $|F|$, can be as large as $O(n)$, and the update cost can be as expensive as $O(n \log n + cost_{\text{EF}})$ in expectation.

Fortunately, by update affordability, there must have been a certain number of affecting updates to trigger an edge(x,y) being reported as invalid. Therefore, we can *charge* the costs of $O(cost_{\text{EF}})$ and $O(|F| \cdot \log n)$ respectively to those updates which had contributed to them. The key question is how to make the *charging argument* for these costs, specifically, which update is charged at what cost.

For simplicity, for the current update of edge (u,v), we only analyze the part of u, because the analysis for the part of v is symmetric.

Amortize cost_{EF} *to Updates.* According to Algorithm 4, we know that $cost_{\text{EF}}$ consists of two parts: (i) the bucket scanning cost, and (ii) the invalid edges reporting cost which is bounded by $O(|F|)$. Let K be the number of buckets that are checked (satisfying the if-condition in Line 4 of Algorithm 4). Clearly, the scanning cost is $O(K+1)$, where the "+1" term comes from the last bucket which does not satisfy the if-condition. We thus charge this "+1" cost to the current update (u,v). Since, for each of the K checked buckets, it must have at least one neighbor $w \in N(u)$ visited. If w is visited for the first time, this bucket checking cost can be charged to the maintenance cost $\ell(u,w)$. Otherwise, if w is visited for the second time, this bucket checking cost can be charged to the reporting cost $O(|F|)$, which, in turn, can also be further charged to the maintenance cost of the edges in F, as we analyze next.

Amortize the Maintenance Cost of an Edge to Updates. For each edge (u,w) reported from a bucket B_i in $\mathcal{B}(u)$, according to Line 6 in Algorithm 4, w is visited for the second time in B_i. Hence, there must have been at least $q(u,w)$ affecting updates incident on u since w was inserted to bucket B_i. Therefore, $\ell(u,w)$, the maintenance cost of edge (u,w), can be charged to those at least $q(u,w)$ affecting updates, each of which is charged by a cost at most $\frac{\ell(u,w)}{q(u,w)}$.

Unpublished working draft. Not for distribution.

Consider the current moment when an update of edge (u, v) arrives; this update (u, v) is then charged (from the part of u) by at most $\sum_{w \in N(u)} \frac{\ell(u,w)}{q(u,w)}$.

Let $q(u, w^*)$ be the update affordability quota value in the *smallest* non-empty bucket B^* in $\mathcal{B}(u)$ at the current moment, and $w^* \in N(u)$. Consider the *retrospective degree* of u, denoted by d_u^{ret}, when w^* was inserted to B^*, that is, when $q(u, w^*)$ was allocated. The degree, d_u, of u at the current moment satisfies: $d_u \leq d_u^{\text{ret}} + 2 \cdot q(u, w^*)$. This is because, otherwise, w^* must have been visited twice in B^*, and hence, the edge (u, w^*) must have been reported as invalid. This is contradictory to the fact that w^* is still in B^*, and that (u, w^*) is still considered as valid since $q(u, w^*)$ was allocated.

Furthermore, since $q(u, w^*) \geq \frac{1}{8}\tau(u, w^*)$ (see Line 2 in Algorithm 2) and by Claim 1, we have $q(u, w^*) \in \Omega(\max\{d_u^{\text{ret}}, d_w^{\text{ret}}\})$. Thus, $d_u^{\text{ret}} \in O(q(u, w^*))$; and it turns out that: $d_u \leq d_u^{\text{ret}} + 2 \cdot q(u, w^*) \in O(q(u, w^*))$. Therefore, the current update (u, v) is charged by at most

$$\sum_{w \in N(u)} \frac{\ell(u, w)}{q(u, w)} \leq \frac{d_u}{q(u, w^*)} \cdot O(\log n) = O(\log n) \text{ in expectation.}$$

Putting the above-charged costs and the maintenance cost of the update (u, v) itself together, we have:

Lemma 4.7. *The amortized cost of each update is bounded by $O(\log n)$ in expectation.*

4.3 Space Consumption

For each $u \in V$, the space consumption of (i) the date structures in *EdgeSimStr* and *CoreFindStr* with respect to u, (ii) the hash table of $N[u]$ for similarity calculation, and (iii) the auxiliary data structure for maintaining $\mathcal{B}(u)$ are all bounded by $O(n_u)$. Hence, we have:

Lemma 4.8. *The overall space consumption of* VD-STAR *is bounded by $O(n + m)$ at all times.*

4.4 Proof of the Last Missing Piece: Claim 1

Next, we give proof for Claim 1 to complete our theoretical analysis. To show this claim, it suffices to prove that the update affordability satisfies $\tau(u, v) \geq t = \frac{1}{4}\rho^2 n_v \in \Omega(d_{\max}(u, v))$, for any edge (u, v) with $n_u \leq n_v$. More specifically, in the following, we prove that $\tilde{\sigma}(u, v)$ remains a valid ρ-absolute approximation to the exact similarity $\sigma(u, v)$ at any moment within t *arbitrary* affecting updates since the last moment when $\tilde{\sigma}(u, v)$ was computed.

In fact, Ruan *et al.* [19] give proof for a lemma similar to our Claim 1 for Jaccard similarity only. Unfortunately, their proof is not immediately applicable to Cosine similarity. As we show below, overcoming this technical difficulty of proving Claim 1 for Cosine similarity requires a more sophisticated analysis.

First, we identify the cases when the similarity has the largest increment or decrement on the exact similarity for an affected update. Consider an update of edge (u, w) and an affected edge (u, v), there are four cases for each similarity measurement:

For Jaccard similarity,

- (u, w) is an insertion,
  - if $w \in N(v)$, $\sigma(u, v)$ is increased to $\frac{I(u,v)+1}{n_u+n_v-I(u,v)}$
  - if $w \notin N(v)$, $\sigma(u, v)$ is decreased to $\frac{I(u,v)}{n_u+n_v-I(u,v)+1}$
- (u, w) is a deletion,
  - if $w \in N(v)$, $\sigma(u, v)$ is decreased to $\frac{I(u,v)-1}{n_u+n_v-I(u,v)}$
  - if $w \notin N(v)$, $\sigma(u, v)$ is increased to $\frac{I(u,v)}{n_u+n_v-I(u,v)-1}$

For Cosine similarity,

- (u, w) is an insertion,
  - if $w \in N(v)$, $\sigma(u, v)$ is increased to $\frac{I(u,v)+1}{\sqrt{(n_u+1)\cdot n_v}}$
  - if $w \notin N(v)$, $\sigma(u, v)$ is decreased to $\frac{I(u,v)}{\sqrt{(n_u+1)\cdot n_v}}$
- (u, w) is a deletion,
  - if $w \in N(v)$, $\sigma(u, v)$ is decreased to $\frac{I(u,v)-1}{\sqrt{(n_u-1)\cdot n_v}}$
  - if $w \notin N(v)$, $\sigma(u, v)$ is increased to $\frac{I(u,v)}{\sqrt{(n_u-1)\cdot n_v}}$

For Dice similarity,

- (u, w) is an insertion,
  - if $w \in N(v)$, $\sigma(u, v)$ is increased to $\frac{I(u,v)+1}{(n_u+n_v+1)/2}$
  - if $w \notin N(v)$, $\sigma(u, v)$ is decreased to $\frac{I(u,v)}{(n_u+n_v+1)/2}$
- (u, w) is a deletion,
  - if $w \in N(v)$, $\sigma(u, v)$ is decreased to $\frac{I(u,v)-1}{(n_u+n_v-1)/2}$
  - if $w \notin N(v)$, $\sigma(u, v)$ is increased to $\frac{I(u,v)}{(n_u+n_v-1)/2}$

Through factorization, it is not difficult to prove that the first case has the largest increment and the third case has the largest decrement for all three similarity measurements. With this, we now prove Claim 1 for the following two cases separately.

Case 1: $n_u \leq \frac{1}{4}\rho^2 n_v$**.** According to Algorithm 5, we set $\tilde{\sigma}(u, v) = 0$ for all three similarity measurements in this case. By Observation 2 in the proof of Lemma 4.2, we know that the exact similarities can be upper bounded by a function of $\beta = \frac{n_u}{n_v}$ for the three measurements. Specifically, $Jaccard(u, v) \leq \beta$, $Cosine(u, v) \leq \sqrt{\beta}$ and $Dice(u, v) \leq 2\beta$. At the moment when $\tilde{\sigma}(u, v)$ is set to 0, we know that the value of $\beta \leq \frac{1}{4}\rho^2$. Next, we show that after $t = \frac{1}{4}\rho^2 n_v$ arbitrary affecting updates, the value of β cannot be greater than $\frac{1}{2}\rho^2$. And thus, by Observation 2, the exact similarities are still no more than ρ for all the three similarity measurements, and therefore, $\tilde{\sigma}(u, v) = 0$ is still a valid ρ-absolute approximation.

It suffices to consider those affecting updates that increase the value of β only. Since $n_u \leq n_v$, without loss of generality, we assume that there are $0 \leq b \leq t$ decrements on n_v while $t - b$ increments on n_u. Let $n_u' = n_u + (t - b)$ and $n_v' = n_v - b$. After such t updates, we have: $\rho^2 n_v' = \frac{2}{4}\rho^2 n_v + \frac{2}{4}\rho^2 n_v - \rho^2 b \geq 2n_u + 2t - 2b = 2n_u'$. Therefore, after these $t = \frac{1}{4}\rho^2 n_v$ updates, the value of $\beta = \frac{n_u'}{n_v'} \leq \frac{1}{2}\rho^2$ holds. This completes the proof of Claim 1 for Case 1.

Case 2: $n_v \geq n_u > \frac{1}{4}\rho^2 n_v$**.** Again consider the moment when a $\frac{1}{2}\rho$-absolute-approximate $\tilde{\sigma}(u, v)$ is computed and the exact similarity at this moment denoted by $\sigma^*(u, v)$. Next, we examine after $t = \frac{1}{4}\rho^2 n_v$ affecting updates, the value of $\sigma(u, v)$ cannot be increased nor decreased by more than $\frac{1}{2}\rho$. And therefore, $\tilde{\sigma}(u, v)$ remains a valid ρ-absolute approximation to the exact similarity at the current moment.

We first show the increment case. As verified above, affecting updates of edges that increase the intersection size $I(u, v)$ of $N[u]$ and $N[v]$ is the most effective way to increase the exact similarity $\sigma(u, v)$. Without loss of generality, suppose that n_u and n_v are increased by $t - b$ and b, respectively, after t affecting updates.

Unpublished working draft. Not for distribution.

For Jaccard similarity, with $t = \frac{1}{4}\rho^2 n_v \le \frac{1}{2}\rho n_v$, the increased exact similarity becomes $\sigma(u,v) = \frac{I(u,v)+t}{(n_u+t-b)+(n_v+b)-(I(u,v)+t)} \le \sigma^*(u,v) + \frac{t}{n_u+n_v-I(u,v)} \le \sigma^*(u,v) + \frac{\frac{1}{2}\rho n_v}{n_u+n_v-I(u,v)} \le \sigma^*(x,y) + \frac{1}{2}\rho$.

For Cosine similarity, with $t = \frac{1}{4}\rho^2 n_v$, the increased exact similarity becomes $\sigma(u,v) = \frac{I(u,v)+t}{\sqrt{(n_u+t-b)\cdot(n_v+b)}} < \sigma^*(u,v) + \frac{t}{\sqrt{n_u \cdot n_v}} < \sigma^*(u,v) + \frac{1/4\cdot\rho^2 n_v}{\sqrt{1/4\cdot\rho^2 n_v \cdot n_v}} = \sigma^*(u,v) + \frac{1}{2}\rho$.

For Dice similarity, with $t = \frac{1}{4}\rho^2 n_v \le \frac{1}{4}\rho n_v$, the increased exact similarity becomes $\sigma(u,v) = \frac{I(u,v)+t}{(n_u+n_v+t)/2} \le \sigma^*(u,v) + \frac{t}{(n_u+n_v)/2} \le \sigma^*(u,v) + \frac{1/4\rho n_v}{n_v/2} = \sigma^*(u,v) + \frac{1}{2}\rho$.

For the decrement case, affecting updates of edges that decrease the intersection size $I(u,v)$ of $N[u]$ and $N[v]$ is the most effective way to decrease the exact similarity $\sigma(u,v)$. Without loss of generality, suppose that n_u and n_v are decreased by $t-b$ and b, respectively, after t affecting updates.

For Jaccard similarity, with $t = \frac{1}{4}\rho^2 n_v \le \frac{1}{2}\rho n_v$, the decreased exact similarity becomes $\sigma(u,v) = \frac{I(u,v)-t}{(n_u-t+b)+(n_v-b)-(I(u,v)-t)} \ge \sigma^*(u,v) - \frac{t}{n_u+n_v-I(u,v)} \ge \sigma^*(u,v) - \frac{\frac{1}{2}\rho n_v}{n_u+n_v-I(u,v)} \ge \sigma^*(u,v) - \frac{1}{2}\rho$.

For Cosine similarity, with $t = \frac{1}{4}\rho^2 n_v$, the decreased exact similarity becomes $\sigma(u,v) = \frac{I(u,v)-t}{\sqrt{(n_u-t+b)\cdot(n_v-b)}} > \sigma^*(u,v) - \frac{t}{\sqrt{n_u \cdot n_v}} > \sigma^*(u,v) - \frac{1/4\cdot\rho^2 n_v}{\sqrt{1/4\cdot\rho^2 n_v \cdot n_v}} = \sigma^*(u,v) - \frac{1}{2}\rho$.

For Dice similarity, with $t = \frac{1}{4}\rho^2 n_v \le \frac{1}{4}\rho n_v$, the decreased exact similarity becomes $\sigma(u,v) = \frac{I(u,v)-t}{(n_u+n_v-t)/2} \ge \sigma^*(u,v) - \frac{t}{(n_u+n_v)/2} \ge \sigma^*(u,v) - \frac{1/4\rho n_v}{n_v/2} = \sigma^*(u,v) - \frac{1}{2}\rho$.

Therefore, for any of these similarity measurements, the update affordability $\tau(u,v) \ge t = \frac{1}{4}\rho^2 n_v \in \Omega(d_{\max}(u,v))$ holds for Case 2. This completes the whole proof for Claim 1.

5 Optimizations

We introduce two optimizations to enhance the practical performance of our *VD-STAR*. The idea stems from a crucial observation – the *CoreFindStr* is designed for finding core vertices efficiently in $O(|V_{\text{core}}|+1)$ time to achieve the target query time complexity $O(m_{cr})$. If we relax this query bound, then it is not necessary to implement the *CoreFindStr*. In this way, we can considerably improve the update efficiency by not only shaving the maintenance cost for *CoreFindStr* but also, importantly, releasing the "approximation budget": recall that *VD-STAR* uses a Δ-Table which introduces a Δ-absolute error in the approximation. Hence, we can set a larger ρ for *EdgeSimStr* that achieves the same approximation guarantee.

5.1 *VD-STAR* with No *CoreFindStr*

Consider an implementation of our *VD-STAR* without *CoreFindStr*. When a query with parameters ε and μ arrives, to identify all the core vertices, it suffices to check for each vertex $u \in V$ whether u is a core vertex with u's sorted neighbor linked list $N(u)$. This can be achieved by scanning $N(u)$ from the beginning and checking whether the similarity between u and the μ^{th} (largest) neighbor is $\ge \varepsilon$ or not. The time complexity is clearly bounded by $O(\mu)$ for each vertex u, and hence, the overall running time of identifying all the vertices is bounded by $O(\mu \cdot n)$. If the sorted neighbor list $N(u)$ is maintained with a binary search tree, finding the μ^{th} largest similarity can be achieved in $O(\log d_{\max})$ time. In this case, the core vertex identification cost is bounded by $O(n \cdot \log n)$. Therefore, without the *CoreFindStr*, our *VD-STAR* can answer each query in $O(\min\{\mu, \log n\} \cdot n + m_{cr})$ time.

Note that, in practice, this query time complexity is acceptable because: (i) the parameter μ in practice is often a small constant for which $\mu \cdot n \in O(n)$ often holds, and (ii) for reasonable clustering parameters, m_{cr} often dominates the term $O(\min\{\mu, \log n\} \cdot n)$. If either of these cases happens, the query time complexity is still bounded by $O(m_{cr})$ the same as before with *CoreFindStr*. As we will see in experiments, *VD-STAR* with no *CoreFindStr*, which is named *Ours-NoT*, significantly improves the update efficiency with just a negligible sacrifice in the query efficiency.

5.2 *VD-STAR* with a Small μ-Table

Recall that *CoreFindStr* can be implemented with a μ-Table which is used in GS*-Index and does not "consume" any approximation budget. Inspired by this, our other version of *VD-STAR* is to implement *CoreFindStr* with a *small* μ-Table. In the sense that, we do not implement the μ-Table *in full* to capture all possible values of the given parameter μ. Instead, we just implement it *partially* for the μ values up to a small constant, say 15. As a result, the maintenance of the small μ-Table would not affect the update time complexity of *VD-STAR*. In addition, it releases the approximation budget consumed by the Δ-Table implementation, and hence, we can increase the value of ρ for the *EdgeSimStr* accordingly.

To answer a query with parameters ε and μ, if μ is captured by the small μ-Table, then we use the μ-Table to retrieve all the core vertices in $O(|V_{\text{core}}|+1)$ and hence, the query time complexity is bounded by $O(m_{cr})$ as desired. Otherwise, we just run the above version of *VD-STAR* with no *CoreFindStr* to answer the query.

6 Experiments

6.1 Experimental Settings

Datasets. We evaluate our algorithms on nine real-world datasets from the Stanford Network Analysis Project [11] which are also used in the baseline papers [19, 21, 24]. Following previous works [19, 24], we treat all graphs as undirected and remove all self-loops. Table 2 summarizes the dataset statistics.

Competitors. We study the performance of our three algorithms: *VD-STAR*, *VD-STAR-NoT* (Section 5.1) and *VD-STAR-μT* (Section 5.2), which are respectively denoted by ***Ours***, ***Ours-NoT*** and ***Ours-μT*** for short. We compare these algorithms with the SOTA exact and approximate algorithms **GS*-Index** [21] and **BOTBIN** [24]. In *Ours-μT*, only a μ-Table with $\mu_{\max} = 15$ is constructed.

Experiment Environment. All experiments are conducted on a Ubuntu virtual server with a 2 GHz CPU and 64 GB memory. All source codes are in C++ and compiled with -O3 turned on. The source code of our implementations can be found in [1].

Default Parameter Settings. By default, the *target overall approximation budget*, denoted by ρ^*, is set as $\rho^* = 0.02$. For BOTBIN, we set $\Delta = 0.01$ by default as suggested in its paper. Since the use of Δ-Table would introduce a Δ error, and hence, to meet the overall

Unpublished working draft. Not for distribution.

Table 2: Dataset Summary

Datasets	n($\times 10^6$)	m($\times 10^6$)	$\bar{d}$	Domain
soc-Slashdot0811	0.08	0.47	12.13	Social network
web-NotreDame	0.33	1.09	6.69	Website hyperlink
web-Google	0.88	4.32	9.86	Website hyperlink
wiki-topcats	1.79	25.44	28.38	Website hyperlink
soc-Pokec	1.63	22.30	27.36	Social network
as-skitter	1.70	11.10	13.06	Traceroute graph
wiki-Talk	2.39	4.66	3.90	Interaction graph
soc-Orkut	3.07	117.19	76.22	Social network
soc-LiveJournal1	4.85	42.85	17.69	Social Network

approximation budget ρ^*, we set $\rho = \rho^* - \Delta = 0.01$ for the *EdgeSimStr* in BOTBIN to achieve an overall ρ^*-approximation. We set the same Δ and ρ for *Ours*. As both *Ours-NoT* and *Ours-μT* do not adopt the Δ-Table, we set $\rho = \rho^*$ for fair comparison.

Update Generation. To simulate graph updates in real-world applications, we randomly generate a sequence of edge insertions and deletions for each dataset. To testify different scenarios, we vary the ratio η of #deletion to #insertion by setting the probabilities of an insertion and a deletion to $\frac{1}{1+\eta}$ and $\frac{\eta}{1+\eta}$, respectively. To perform an edge deletion, we uniformly at random choose an existing edge and delete it. To perform an edge insertion, we employ three strategies:

- random-random (**RR**): a non-existent edge is randomly added.
- degree-random (**DR**): Each vertex u has a probability of $\frac{d_u}{2m}$ to be chosen, where d_u is the degree of u and m is the number of edges in the current graph. Once u is chosen, the second vertex, v, is randomly chosen from those vertices not yet linked to u.
- degree-degree (**DD**): Vertex u is chosen as in DR; vertex v is chosen from the vertices not yet linked to u with $\frac{d_v}{2m}$ probability.

By default, we set $\eta = \frac{1}{10}$. For each dataset and a configuration of η and update generation strategy, we generate $M = 2m^*$ updates, where m^* is the number of edges in the initial graph.

Query Simulation. To simulate the query process in real-world applications, we randomly generate a query after every 20 updates, with $\varepsilon \in [0.1, 0.5]$ and $\mu \in [2, 2\bar{d}]$ of each graph ($\bar{d}$: the average degree). With a total of $M = 2m^*$ updates, $0.1m^*$ queries are tested.

6.2 Study on Update Efficiency

6.2.1 **Average Update Time with Default Parameters.** We first study the average running time of processing updates. As shown in Figure 4a, we have the following observations: (1) *Ours-NoT* consistently achieves the best update efficiency. Particularly, it accelerates update processing by as much as 9,315 times (on wiki-Talk) compared with GS*-Index and 647 times (on as-skitter) compared with BOTBIN. GS*-Index uses exact similarity calculation such that it has the highest update time. (2) *Ours* is up to 18 times faster in update processing (on soc-LiveJournal1) compared to BOTBIN which also uses a Δ-table. (3) *Ours-μT* achieves a significant improvement in updating speed, while maintaining competitive query time, as elaborated later in Section 6.3. Specifically, it outperforms SOTA methods by up to 208 times on soc-Slashdot vs. GS*-Index.

6.2.2 **Memory Consumption.** As shown in Figure 4b, all methods exhibit minor differences in memory consumption owing to that their space consumption are all linear to the graph size. *Ours-NoT* has the smallest memory consumption because it has no implementation for *CoreFindStr*. *Ours* consumes slightly less memory compared with BOTBIN possibly because maintaining $\mathcal{B}(u)$ might be more space-efficient than maintaining the bottom-k signatures.

6.2.3 **Impact of Target Overall Approximation ρ^*.** To test how ρ^* affects the update time, we vary ρ^* from 0.001 to 0.1. GS*-Index is excluded from this experiment as it is an exact algorithm. As shown in Figure 5, the update time decreases when ρ^* grows, as expected. A larger ρ^* value leads to smaller sample sizes for similarity estimation and larger update affordability (τ) for our algorithms. As expected, *Ours-NoT* has the lowest update time under all ρ^* values tested, with speedups reaching up to 700 times over BOTBIN (on as-skitter when $\rho^* = 0.01$).

Compared with BOTBIN, *Ours-μT* and *Ours* are also about an order of magnitude faster. Remarkably, even with $\rho = 0.001$, our *Ours-NoT* achieves an average update time of 36×10^{-6} second on a graph with 100 million edges. For BOTBIN and *Ours*, Δ is set to 0.01 except for $\rho^* = 0.01$ and $\rho^* = 0.001$ where Δ is set as half of ρ^* to satisfy the ρ^*-absolute-approximation for these two algorithms.

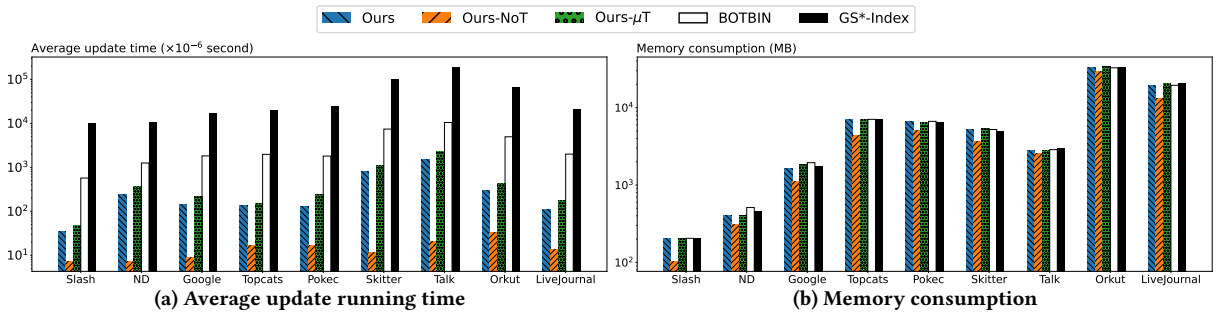
6.2.4 **Impact of Update Distributions.** As shown in Figure 6, the average update times of all algorithms increase as updates follow more skewed distributions (from RR to RD and to DD). This trend primarily arises because inserting or deleting a neighbor for a vertex of a larger degree takes a longer time (i.e., $O(\log n_u)$). Additionally, vertices of larger degrees appear more frequently in the μ-Table. Despite these challenges, our algorithms consistently outperform all competitors, with speedups of up to 5,656 times on **RR**, 9,315 times on **DR**, and 4,857 times on **DD**.

6.2.5 **Impact of Deletion-to-Insertion Ratio.** Next, we vary deletion-to-insertion ratios, and Figure 7 reports the results. We observe the following: (1) The average update times increase with insertions occurring more frequently across all algorithms and datasets. This is expected since the number of edges grows with an increase in insertions, and the growth accelerates when the deletion-to-insertion ratio η decreases. (2) Across all settings tested, all our algorithms outperform the SOTA algorithms, with *Ours-NoT* having the lowest update times. (3) When $\eta = 0$, all updates are insertions. In this case, *Ours-NoT* amplifies the speedup from up to 9,315 times to 11,959 times (vs. GS*-Index), and from up to 647 times to 805 times (vs. BOTBIN), compared with $\eta = \frac{1}{10}$. This reaffirms the robustness of our algorithms in terms of scalability.

6.3 Study on Query Efficiency

The query efficiency results are shown in Figure 8. Several observations are made: (1) GS*-Index, BOTBIN, and *Ours* exhibit similar query times as their query time are all bounded by $O(m_{cr})$, linear to the size of the clustering result graph. *Ours-μT*, utilizing a fixed-size μ-table (in Section 5.2), only incurs a cost bounded by $O(\min\{\mu, \log n\} \cdot n + m_{cr})$ when the input μ exceeds a certain threshold $\mu_{\max}$ which is set as 15 in this experiment. In practice, it can still achieve performance similar to $O(m_{cr})$ methods, as evidenced by the results showing very similar query times. (2) Across all datasets, our *Ours-NoT* algorithm's query times are within the same magnitude and are at most 0.7×10^{-3} second slower compared with the other algorithms. Notably, on datasets like web-Google and web-topcats, *Ours-NoT* has a lower query time. This is because the size of the clustering result graph is large in these datasets, and

Unpublished working draft. Not for distribution.



(a) Average update running time (b) Memory consumption

Figure 4: Update processing performance results

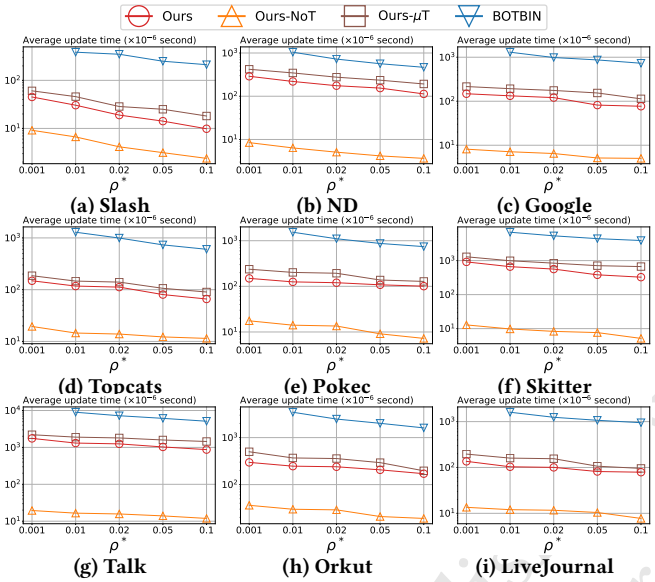


Figure 5: Average update running time vs. ρ^*

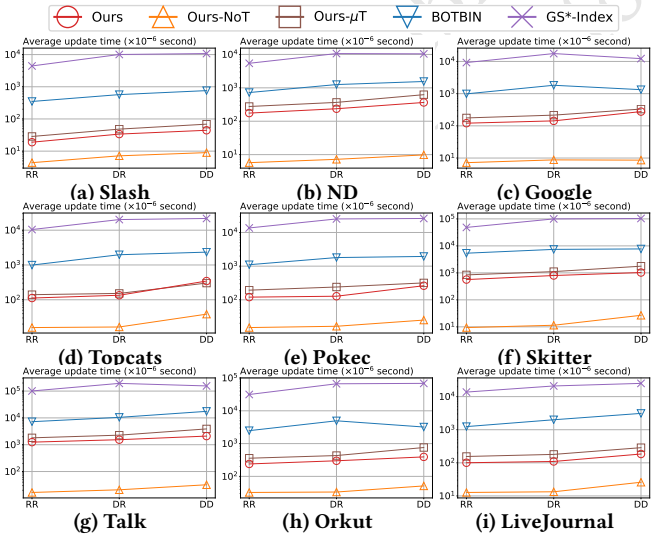


Figure 6: Average update time vs. update distribution

hence, $O(m_{cr})$ dominates the query time, making the query overhead of *Ours-NoT* negligible. (4) In structural clustering problems, the number of the clustering result vertices (n_{cr}) has substantial

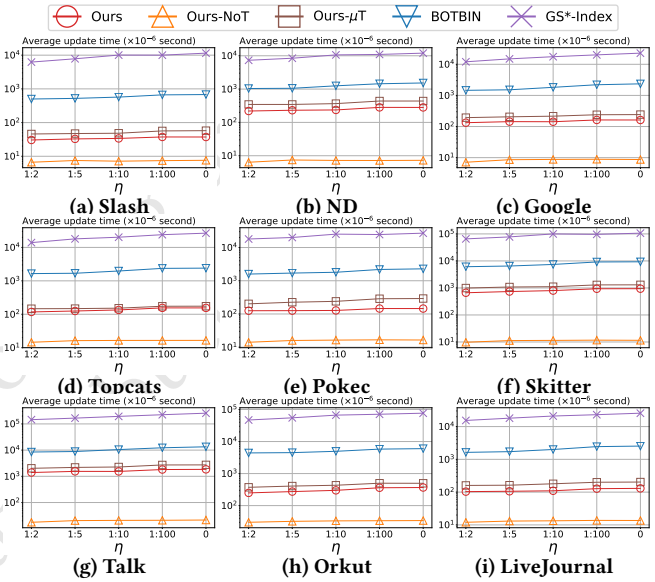


Figure 7: Average update time vs. η

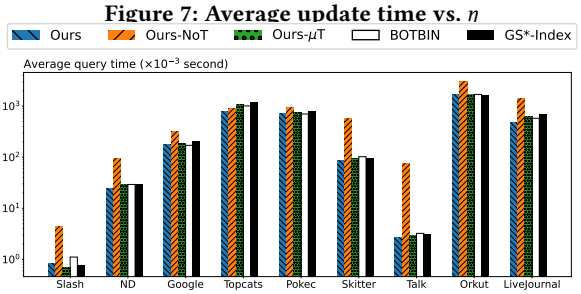
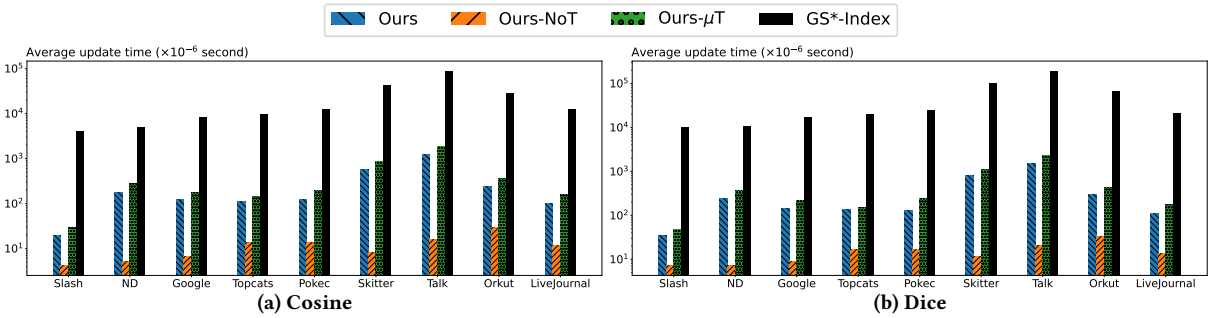


Figure 8: Query processing performance results

practical implications. If n_{cr} is small, the structural clustering results may lose significance because most vertices are excluded. In cases where n_{cr} approaches n, *Ours-NoT* introduces a negligible overhead in queries while accelerating updates by over 100 times.

6.4 Study on Clustering Quality

We look into the clustering quality of both our algorithms and BOTBIN, in terms of the *mislabeled rate* (**MLR**) and *adjusted rand index* (**ARI**) [9]. MLR is calculated as dividing the number of incorrectly labeled edges by the number of edges, m, of the current graph. ARI is widely used to evaluate the clustering quality which outputs a



(a) Cosine (b) Dice

Figure 9: Average update running time on Cosine and Dice

Table 3: Clustering quality results

	$\rho = 0.02$				$\rho = 0.1$			
	BOTBIN		*Ours*		BOTBIN		*Ours*	
	ARI ↑	MLR ↓	ARI ↑	MLR ↓	ARI ↑	MLR ↓	ARI ↑	MLR ↓
soc-Slashdot0811	**0.9967**	**0.02%**	0.9965	**0.02%**	0.9709	1.60%	**0.9715**	**1.57%**
web-NotreDame	0.9994	**0.13%**	**0.9995**	**0.13%**	**0.9632**	4.51%	0.9626	**4.46%**
web-Google	**0.9991**	**0.14%**	0.9990	**0.14%**	0.9695	5.52%	**0.9701**	**5.50%**
wiki-topcats	0.9990	**0.06%**	**0.9996**	**0.06%**	**0.9896**	**0.86%**	0.9887	0.92%
soc-Pokec	**0.9957**	0.18%	**0.9955**	**0.17%**	**0.9676**	**5.32%**	0.9653	5.33%
as-skitter	0.9995	**0.19%**	**0.9996**	**0.19%**	0.9826	6.38%	**0.9842**	**6.34%**
wiki-Talk	0.9987	0.45%	**0.9989**	**0.44%**	0.9716	**7.29%**	**0.9722**	7.30%
Orkut	0.9946	**0.12%**	**0.9954**	**0.12%**	**0.9548**	**4.38%**	**0.9548**	4.40%
soc-LiveJournal1	**0.9998**	**0.12%**	0.9995	**0.12%**	0.9975	4.56%	**0.9982**	**4.56%**

value from 0 to 1, where 1 means that the clusters are exactly the same as the ground truth. We evaluate the result quality using the default $\rho^* = 0.02$ and a larger $\rho^* = 0.1$. Note that here ρ^* represents an absolute error, and 0.1 is already a relatively large error value. Both MLR and ARI are measured for each query as described above and their average values are reported in Table 3.

Both BOTBIN and *Ours* have high-quality results, leveraging error bounds to their advantage. Notably, *Ours* outperforms BOTBIN on more datasets. When $\rho^* = 0.02$, *Ours* achieves MLR of less than 0.2% across all datasets except for `wiki-Talk`, with ARI values ranging from 0.9954 to 0.9996. Even with $\rho^* = 0.1$, *Ours* maintains an average ARI of at least 0.9626 (on `web-NortreDame`) and MLR of at most 7.30% (on `wiki-Talk`). *Ours-μT* and *Ours-NoT* have similar results to *Ours*. For brevity, they are not detailed here. These results again underscore the practical significance of theoretical error bounds with a high success rate on real datasets.

Table 4: Clustering quality results on all three measurements (The result of Jaccard Similarity with $\rho = 0.02$ is copied and pasted from Table 3 to here for easy comparison.)

	Jaccard		Cosine		Dice	
Datasets	ARI ↑	MLR ↓	ARI ↑	MLR ↓	ARI ↑	MLR ↓
`soc-Slashdot0811`	0.9967	0.02%	0.9900	0.10%	0.9965	0.01%
`web-NotreDame`	0.9994	0.13%	0.9863	0.15%	0.9990	0.12%
`web-Google`	0.9991	0.14%	0.9599	0.29%	0.9990	0.14%
`wiki-topcats`	0.9990	0.06%	0.9700	0.07%	0.9999	0.06%
`soc-Pokec`	0.9957	0.18%	0.9609	0.23%	0.9958	0.15%
`as-skitter`	0.9995	0.19%	0.9806	0.28%	0.9996	0.13%
`wiki-Talk`	0.9987	0.45%	0.9672	0.53%	0.9989	0.42%
`Orkut`	0.9946	0.12%	0.9673	0.14%	0.9958	0.13%
soc-LiveJournal1	0.9998	0.12%	0.9843	0.18%	0.9994	0.12%

6.5 Experiments on Cosine and Dice Similarities

Since BOTBIN does not work for Cosine or Dice similarities, the experiments are conducted with our algorithms and GS*-Index only.

Cosine. As Figure 9a shows, the comparative pattern in update running time is similar to that observed under the Jaccard similarity setting (Figure 4a above). However, the update time of *Ours-NoT* shows a slight increase compared with the Jaccard similarity setting (Figure 4a above) due to larger constant factors in the sample size for similarity estimation and smaller constant factors in τ for update affordability to ensure the complexity bounds, as described in Section 4.1 and Section 4.4.

The clustering quality results are shown in Table 4. Our algorithms also show solid results for Cosine similarity-based structural clustering. Compared with exact algorithms like GS*-Index (whose ARI is 1 and MLR is 0 and are omitted from the table), our algorithm can achieve up to 0.9901 average ARI (on `soc-Slashdot0811`) and as low as 0.06% MLR (on `wiki-topcats`).

Dice. The results of Dice similarity are shown in Figure 9b (update running time) and Table 4 (clustering quality). As expected, our algorithms show similar performance as that on Jaccard.

7 Conclusion

In this paper, we propose an algorithm called *VD-STAR* for the problem of *Dynamic Structural Clustering for All Parameters*. Our *VD-STAR* can return an ρ-absolute-approximate clustering result with high probability for every query in $O(m_{cr})$ time, and can process each update in $O(\log n)$ amortized expected time, while its space consumption is bounded by $O(n + m)$ at all times. Our algorithm works well with Jaccard, Cosine and Dice similarity measurements and supports arbitrary updates. *VD-STAR* significantly improves the state-of-the-art approximate algorithm BOTBIN which achieves $O(\log^2 n)$ expected time per-update for random updates under Jaccard similarity only. We evaluate our algorithm on nine real datasets, which shows strong empirical results in terms of update and query efficiency, clustering result quality, and robustness in handling various update distributions.

Unpublished working draft. Not for distribution.

References

[1] Anonymous. 2024. Source code and technical report. https://anonymous.4open.science/r/ver-DynStrClu-08DF/

[2] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang, and Shiyu Yang. 2017. pSCAN: Fast and exact structural graph clustering. *IEEE Transactions on Knowledge and Data Engineering* 29, 2 (2017), 387–401.

[3] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical review E* 70, 6 (2004), 066111.

[4] Graham Cormode, Shanmugavelayutham Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms* 7, 2 (2011), 1–20.

[5] Yijun Ding, Minjun Chen, Zhichao Liu, Don Ding, Yanbin Ye, Min Zhang, Reagan Kelly, Li Guo, Zhenqiang Su, Stephen C. Harris, Feng Qian, Weigong Ge, Hong Fang, Xiaowei Xu, and Weida Tong. 2012. atBioNet–an integrated network analysis tool for genomics and biomarker discovery. *BMC Genomics* 13 (2012), 1–12.

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

[7] Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30.

[8] Zengfeng Huang, Ke Yi, and Qin Zhang. 2012. Randomized algorithms for tracking distributed count, frequencies, and ranks. In *PODS*. 295–306.

[9] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of Classification* 2 (1985), 193–218.

[10] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. 2006. Communication-efficient distributed monitoring of thresholded counts. In *SIGMOD*. 289–300.

[11] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data.

[12] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN: Overlapping community detection using the link-space transformation. In *ICDE*. 292–303.

[13] Zhichao Liu, Qiang Shi, Don Ding, Reagan Kelly, Hong Fang, and Weida Tong. 2011. Translating clinical findings into knowledge in drug safety evaluation-drug induced liver injury prediction system (DILIps). *PLoS Computational Biology* 7, 12 (2011), e1002310.

[14] Venkata-Swamy Martha, Zhichao Liu, Li Guo, Zhenqiang Su, Yanbin Ye, Hong Fang, Don Ding, Weida Tong, and Xiaowei Xu. 2011. Constructing a robust protein-protein interaction network by integrating multiple public databases. *BMC Bioinformatics* 12, Suppl 10 (2011), S7.

[15] Mark E. J. Newman. 2004. Finding and evaluating community structure in networks. *Physical Review E* 69, 26113 (2004), 1–16.

[16] Symeon Papadopoulos, Yiannis Kompatsiaris, and Athena Vakali. 2009. Leveraging collective intelligence through community detection in tag networks. *Proceedings of CKCaR* 9 (2009), 1–9.

[17] Symeon Papadopoulos, Yiannis Kompatsiaris, and Athena Vakali. 2010. A graph-based clustering scheme for identifying related tags in folksonomies. In *International Conference on Data Warehousing and Knowledge Discovery*. 65–76.

[18] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in social media: Performance and application considerations. *Data Mining and Knowledge Discovery* 24 (2012), 515–554.

[19] Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2021. Dynamic structural clustering on graphs. In *SIGMOD*. 1491–1503.

[20] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. Scan++ efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1178–1189.

[21] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient structural graph clustering: an index-based approach. *Proceedings of the VLDB Endowment* 11, 3 (2017), 243–255.

[22] Scott White and Padhraic Smyth. 2005. A spectral clustering approach to finding communities in graphs. In *SDM*. 274–285.

[23] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *KDD*. 824–833.

[24] Fangyuan Zhang and Sibo Wang. 2022. Effective indexing for dynamic structural graph clustering. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2908–2920.

[25] Zhuo Zhang, Junhao Gan, Zhifeng Bao, Seyed Mohammad Hussein Kazemi, Guangyong Chen, and Fengyuan Zhu. 2022. Approximate range thresholding. In *SIGMOD*. 1108–1121.

Unpublished working draft. Not for distribution.