University Library



A gateway to Melbourne's research publications

Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
Zhao, Z;Gan, J;Ruan, B;Bao, Z;Qi, J;Wang, S

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

Date:
2025

Citation:
Zhao, Z., Gan, J., Ruan, B., Bao, Z., Qi, J. & Wang, S. (2025). Dynamic Structural Clustering Unleashed: Flexible Similarities, Versatile Updates and for All Parameters. KDD '25: Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2, 2, pp.3980-3991. Association for Computing Machinery. https://doi.org/10.1145/3711896.3736918.

Persistent Link:
https://hdl.handle.net/11343/361934

License:
CC-BY





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

Zhuowei Zhao
The University of Melbourne
Parkville, VIC, Australia
zhuoweiz1@student.unimelb.edu.au

Junhao Gan
The University of Melbourne
Parkville, VIC, Australia
junhao.gan@unimelb.edu.au

Boyu Ruan
The Hong Kong University of Science and Technology
Hong Kong, China
boyuruan@ust.hk

Zhifeng Bao
RMIT University
Melbourne, VIC, Australia
zhifeng.bao@rmit.edu.au

Jianzhong Qi
The University of Melbourne
Parkville, VIC, Australia
jianzhong.qi@unimelb.edu.au

Sibo Wang
The Chinese University of Hong Kong
Hong Kong, China
swang@se.cuhk.edu.hk

Abstract

We study structural clustering on graphs in *dynamic* scenarios, where graphs can be updated by *arbitrary* insertions or deletions of edges/vertices. Our goal is to efficiently compute structural clustering results under three conditions: 1) for any clustering parameters ε and μ provided *on the fly*, 2) for arbitrary graph update patterns, and 3) for all typical similarity measurements. To achieve this, we propose an algorithm named *VD-STAR* that is much simpler yet more efficient than state of the art. With a theoretical guarantee on clustering result's quality, *VD-STAR* can produce clustering results with up to 99.9% accuracy. Moreover, *VD-STAR* is easy to implement as it just needs to maintain sorted linked lists and hash tables, making it highly deployable in practice. Most importantly, *VD-STAR* improves the expected per-update time bound from state-of-the-art $O(\log^2 n)$, which *relies on* specific assumption on update pattern, to $O(\log n)$ *amortized in expectation without* any assumption on update pattern. We further design two variants of *VD-STAR* to enhance its empirical performance. Experimental results show that our algorithms consistently outperform state-of-the-art competitors by up to 9,315 times in update time across nine real datasets, while maintaining similar update time and memory usage.

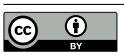
CCS Concepts

• **Theory of computation → Data structures and algorithms for data management**; **Dynamic graph algorithms**.

Keywords

Structural Clustering, Graphs, Dynamic Scenarios

ACM Reference Format:
Zhuowei Zhao, Junhao Gan, Boyu Ruan, Zhifeng Bao, Jianzhong Qi, and Sibo Wang. 2025. Dynamic Structural Clustering Unleashed: Flexible Similarities, Versatile Updates and for All Parameters. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3711896.3736918



This work is licensed under a Creative Commons Attribution 4.0 International License.
KDD '25, Toronto, ON, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1454-2/2025/08
https://doi.org/10.1145/3711896.3736918

KDD Availability Link:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.15486055.

1 Introduction

Graph clustering is a fundamental problem that aims to group similar vertices of a graph into clusters. In this paper, we focus on a popular scheme named *Structural Clustering* [21] 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 *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, and 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*.

Applications. Structural clustering has a wide range of applications. In genomics and biomarker discovery, proteins are modeled as vertices while edges represent physical or functional associations; each structural cluster in a protein-protein interaction network is considered as a functional module [11, 12]. In social networks, it is used for community detection [5, 10, 16], where each vertex represents a user, and an edge signifies an interactive relationship between users, such as following or friendship. Users with similar interactive relationships can be identified and grouped into the same cluster, forming a community. Similarly, structural clustering also aids in web data analysis [14, 15] and fraud detection in transaction networks [2, 13]. Among these applications, dynamic scenarios are particularly significant, since graphs rapidly evolve

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 [18]	GS*-Index [20]
Update running time	**Arbitrary update**	$O(\log n)$ amortized expected	$O(d_{\max} \log n)$	$O(\log^2 n)$ amortized	$O(d_{\max}^2 \log n)$
	Uniformly at random	$O(\log n)$ amortized expected	$O(\log^2 n)$ expected	$O(\log^2 n)$ amortized	$O(d_{\max}^2 \log n)$
Similarity measurement	**Jaccard similarity**	✓	✓	✓	✓
	Cosine similarity	✓	X	✓	✓
	Dice similarity	✓	X	✓	✓
Query running time for ε and μ given on the fly		$O(m_{cr}+1)$	$O(m_{cr}+1)$	$O(m \log^2 n)$ running from scratch	$O(m_{cr}+1)$
Approximation Guarantee		ρ-absolute approximation	ρ-absolute approximation	ρ-absolute approximation	Exact

over time, as seen in clinical data, social networks, web-page hyperlinks, and transaction networks. In other applications, such as protein-protein interactions, an efficient dynamic algorithm can also benefit research, as certain protein structures can transform into others, which can be modeled as graph updates in dynamic scenarios. Recently, the AI community has turned its attention to structural clustering to improve model training [3, 22], and there are growing interests [4, 7, 19, 23] in graph neural network regarding dynamic graphs. Enhancing the efficiency of dynamic structural clustering can significantly speed up model training.

Related Work and Limitations. Structural clustering was first proposed by Xu *et al.* [21] and has since opened a line of studies. Among these, pSCAN [1] is the state-of-the-art (SOTA) *exact* algorithm for *static* graphs, where no updates to the input graph G are allowed. The running time complexity of pSCAN is bounded by $O(m^{1.5})$, where m is the number of edges. This $O(m^{1.5})$ bound has been shown to be worst-case optimal [1].

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 [20] is the SOTA for *Dynamic Structural Clustering*, which can return the exact clustering result 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 [18] and BOTBIN [24] are two SOTA *approximate* algorithms, yet BOTBIN 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 provided 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 celebrities on X. If this assumption does not hold, the per-update complexity of BOTBIN degenerates to $O(d_{\max} \cdot \log n)$. The merits and limitations of these SOTA algorithms are summarized in Table 1.

Our Methods. Given the importance of structure clustering, we propose an ultimate algorithm, *Versatile Dynamic Structural Clustering* (*VD-STAR*), which unifies the state of the art and meanwhile addresses all their limitations. Specifically, we adopt the notion of ρ-absolute approximation for solving the problem. With this approximation formulation, we propose a novel sampling method to estimate similarity, which we show works on all three similarity measurements (Jaccard, Cosine and Dice) suggested by Xu *et al.* [21] (Section 4.2). We further adopt the notion of update affordability (Section 4.1) to capture the moment when an edge can possibly affect the clustering results, and hence, the similarity of its endpoints is re-computed. To efficiently track the update affordability for each edge, we propose a fast yet easy-to-implement algorithm (Section 4.2). Ulitmately, we propose a unified framework (Section 3) for GS*-Index, BOTBIN, and our *VD-STAR*, with which users just need to implement the specified interfaces to access the algorithms. This framework offers flexibility for customization, allowing users to swap between different algorithm implementations to make their own "new" solutions. As we will see in Section 6, this is precisely how we design the two variants of *VD-STAR*.

Our Contributions. We make the following contributions:

- We propose a unified algorithm framework for solving the problem of Dynamic Structural Clustering for All Parameters, with which one can easily design their own algorithm by implementing each component.
- 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* advances the SOTA in the following aspects:
  - It improves the per-update time complexity of the SOTA algorithms, BOTBIN and DynELM, from $O(\log^2 n)$ expected to $O(\log n)$ amortized in expectation.
  - It supports arbitrary update patterns lifting BOTBIN's assumption that the updates have to be uniformly at random.
  - It supports all three similarity measurements, whereas the SOTA algorithm, BOTBIN, supports Jaccard similarity only.
- While the theoretical analysis is technical, our *VD-STAR* can be easily implemented in practice, as it just needs to maintain and scan a number of sorted lists and hash tables.
- 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 $9,315\times$ regarding update efficiency.

Remark. Throughout this paper, some of the missing proofs for the observations, lemmas, and theorems marked with [*] are in the appendix, while all proofs can be found in our technical report [26].

2 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, denoted by $\sigma(u, v)$, is defined as follows: $\sigma(u, v) = 0$ if there is *no* edge between u and v, otherwise $\sigma(u, v)$ 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*: $\sigma(u, v) = \frac{I(u,v)}{(n_u+n_v)/2}$.

Similar Neighbors, Edge Labels and Core Vertices. Given a *similarity threshold* $0 < \varepsilon < 1$, vertices u and v are *ε-similar neighbors* if $\sigma(u, v) \geq \varepsilon$, and (u, v) is labelled as an *ε-similar edge*; otherwise, it is labeled as an *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.

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*. 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 $O(m_{cr} + 1)$ time.*

Definition 2.1 (Problem Definition). Consider a given similarity measurement $\sigma(\cdot, \cdot)$ (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 problem of ***Dynamic Structural Clustering for All Parameters*** (**DynStrClu-AllPara**) asks to:

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

3 A Unified Algorithm Framework

In this section, we propose a *unified algorithm framework* for solving *DynStrClu-AllPara* with pseudo code shown in Algorithm 1. The SOTA exact algorithm, GS*-Index, and the SOTA approximate algorithm, BOTBIN, as well as our VD-START algorithm (proposed in Section 4), all follow this framework. Besides, users can also design their own algorithms with this framework by customizing the operation interface implementations.

There are three main data structures in the framework:

- ***Sorted Neighbor Lists***: for each vertex $u \in V$, a *non-increasing sorted* list of u's neighbors by their similarities to u; we simply use $N(u)$ to refer to this sorted list, and each neighbor of u in this list is stored along with its similarity to u. Therefore, given a similarity threshold ε, by scanning the sorted neighbor list $N(u)$, all the similar neighbors of u can be retrieved efficiently.
- ***EdgeSimStr***: This data structure is to maintain the (approximate) similarities for all the edges under updates. Specially, for an update of edge (u, v), both the degrees $d(u)$ and $d(v)$ alter and hence, the similarity value of each edge incident on u or v gets changed such that its similarity currently recorded in *EdgeSimStr* may differ too much from the true value. An important goal of *EdgeSimStr* is to efficiently identify all those "affected" edges whose similarities are considered "invalid" and need to be re-computed, ideally without touching every neighbor of u and v, so as to get rid of the expensive $O(d_{\max})$ cost for each update (u, v). There are five operation interfaces:
  - 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 for u and v in *EdgeSimStr* accordingly;
  - insert((x, y)): insert an edge (x, y) to *EdgeSimStr*;
  - delete((x, y)): delete an edge (x, y) from *EdgeSimStr*;
  - find((u, v), op): return a set F of all the affected edges whose similarities are considered "invalid" and need to be re-computed;
  - cal-sim((x, y)): given an edge (x, y), return $\sigma(x, y)$;
- ***CoreFindStr***: This data structure is to support efficient retrieval of all core vertices for given parameters ε and μ. In fact, without *CoreFindStr*, one can also retrieve all core vertices in $O(n)$ time, by scanning the sorted neighbor list $N(u)$ for each vertex $u \in V$. The goal of *CoreFindStr* is to achieve this more efficiently, e.g., ideally in $O(1 + |V_{\text{core}}|)$ time. It has two operation interfaces:
  - update(u): given a vertex $u \in V$, update the information maintained for u in *CoreFindStr*;
  - find-core(ε, μ): return the set V_{core} of all the core vertices with respect to the given parameters ε and μ;

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

Per-Update Running Time. In the Update Procedure in Algorithm 1, Line 2 takes $O(cost_{\text{EU}})$ time and Lines 3-10 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 11 is bounded $O(cost_{\text{EF}})$ while that of Lines 12-16 is bounded by $O(|F| \cdot (cost_{\text{EC}} + cost_{\text{EI}} + cost_{\text{ED}} + \log n))$. Finally, Lines 17 - 19 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

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

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

Given an implementation of Algorithm 1, substituting the corresponding costs to the above analysis, the query and per-update running time bounds follow. Detailed implementations and analysis of the SOTA methods are in our technical report [26].

4 Our Versatile DynStrClu Algorithm

Next, we introduce our solution, called *Versatile Dynamic Structural Clustering* (*VD-STAR*), and its detailed implementation under the unified framework (Algorithm 1).

4.1 Approximation and Update Affordability

Our *VD-STAR* adopts the following approximation formulation:

ρ-Absolute-Approximation. Given a *constant* parameter $0 < \rho < 1$ and a 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., $\varepsilon - \rho \le \sigma(u, v) \le \varepsilon + \rho$, (u, v) can be considered as either similar or dissimilar.

Given all the edge labels determined under the ρ-absolute approximation, all the notions introduced in Section 2 are well defined.

A crucial feature of the ρ-absolute-approximation formulation is that Case (3) creates a leeway for us to label the edges. Specifically, the label of an edge (u, v) determined by some approximate similarity $\tilde{\sigma}(u, v)$ is considered correct as long as $\tilde{\sigma}(u, v)$ is within an ρ-absolute error from the true similarity $\sigma(u, v)$, namely, $\tilde{\sigma}(u, v) \in [\sigma(u, v) - \rho, \sigma(u, v) + \rho]$. As a result, there are two immediate benefits. First, the similarity can now be estimated approximately within certain error. Second, if an edge (u, v) falls in the "must-be-similar" case (Case (1)) at the current moment, then $\sigma(u, v)$ has to be decreased by at least 2ρ before moving into the "must-be-dissimilar" case (Case (2)). As we will show in Lemma 4.1, such a decrement would allow (u, v) to afford a certain number, denoted by $\tau(u, v)$, of *affecting* updates (incident on either u or v) before its approximate similarity value needs to be *re-computed*.

Lemma 4.1. [*] *For any edge (u, v), let $\sigma(u, v)$ be the true similarity over time, i.e., $\sigma(u, v)$ changes under updates. Consider the moment when a $\frac{1}{2}\rho$-absolute approximate similarity $\tilde{\sigma}(u, v)$, which satisfies $|\tilde{\sigma}(u, v) - \sigma(u, v)| \le \frac{1}{2}\rho$, is just computed. Since then, (u, v) can afford at least $\tau(u, v) \ge \frac{1}{4}\rho^2 d_{\max}(u, v) \in \Omega(d_{\max}(u, v))$ affecting updates (which are incident on either u or v) such that $\tilde{\sigma}(u, v)$ is still "valid" in the sense that $|\tilde{\sigma}(u, v) - \sigma(u, v)| \le \rho$. Here, $d_{\max}(u, v) = \max\{n_u, n_v\}$.*

The value of $\tau(u, v)$ in Lemma 4.1 is called the *update affordability* of edge (u, v). By definition, one may need to *re-compute* $\tilde{\sigma}(u, v)$ *no later* than the arrival of the $\tau(u, v) + 1^{\text{st}}$ affecting update for each $(u, v) \in E$. However, simply tracking the number of affecting updates since $\tilde{\sigma}(u, v)$ was last computed for each edge (u, v) can be expensive. To see this, for an update of edge (u, w), it needs to "touch" each affected edge (incident on either u or w) to maintain its counter, which incurs a cost of $d_u + d_w \in O(d_{\max}) = O(n)$. Therefore, to bypass such an $O(d_{\max})$ cost, the key is to identify the set F of all invalid edges when an update arrives (Line 11 in Algorithm 1), without touching each of the affected edges.

4.2 Our Implementation of *EdgeSimStr*

Rationale of Our Algorithm. The basic idea of our approach to identify invalid edges is as follows. For each edge $(u, v) \in E$, when $\tilde{\sigma}(u, v)$ is just computed, we compute its update affordability $\tau(u, v) = \frac{1}{4}\rho^2 \max\{n_u, n_v\}$. Instead of tracking the exact arrival moment of the $\tau(u, v)^{\text{th}}$ affecting update, we aim to just identify an *arbitrary* moment when there have been at least $\frac{1}{4}\lfloor\tau(u, v)\rfloor_2$ affecting updates occurred. Here, $\lfloor\tau(u, v)\rfloor_2$ is the *largest power-of-two* integer that is no more than $\tau(u, v)$, namely, $\lfloor\tau(u, v)\rfloor_2 = 2^{\lfloor\log_2 \tau(u,v)\rfloor}$. Such a moment is called a *checkpoint moment* of edge (u, v). Clearly, $\lfloor\tau(u, v)\rfloor_2 \ge \frac{1}{2}\tau(u, v)$. When a checkpoint moment of (u, v) is identified, there must have been at least $\frac{1}{4}\lfloor\tau(u, v)\rfloor_2 \ge \frac{1}{8}\tau(u, v) \in \Omega(\tau(u, v)) = \Omega(d_{\max}(u, v))$ affecting updates.

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 vertices u and v each. 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 in F (Line 11 in Algorithm 1), for which the approximate similarity $\tilde{\sigma}(u, v)$ needs to be re-computed.

The Data Structure for *EdgeSimStr*. To achieve this, 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 $0 \le i \le \lceil \log_2 n \rceil$);
  - 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 *ascending 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 when B_i was 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 another affecting update has occurred on them.

Implementation of *EdgeSimStr*.insert. To insert an edge (u, v) into *EdgeSimStr*, our algorithm (Algorithm 4) first computes the update affordability $\tau(u, v) = \frac{1}{4}\rho^2 \max\{n_u, n_v\}$ and the affordability quota $q(u, v) = \frac{1}{4}\lfloor \tau(u, v) \rfloor_2$, and then inserts v (resp., u) into the corresponding bucket in $\mathcal{B}(u)$ (resp., $\mathcal{B}(v)$). If the bucket does not exist, 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, it is removed from the bucket list. We defer the pseudo code to Appendix D.

Implementation of *EdgeSimStr*.find. Algorithm 2 gives the 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 counter c_u is increased by one, it suffices to scan the sorted bucket list to check all the non-empty buckets B_i with $\lfloor \frac{c_u}{2^i} \rfloor > \lfloor \frac{\bar{c}_u(B_i)}{2^i} \rfloor$, because they were last visited when $c_u = \bar{c}_u(B_i)$ (see Line 4 in Algorithm 2). For each of such buckets B_i, our algorithm reports and adds the edge (u, w) to F for each $w \in B_i$ if 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 5.1.

Implementation of *EdgeSimStr*.cal-sim. We propose a sampling-based algorithm (Algorithm 3) to compute a $\frac{1}{2}\rho$-absolute approximate similarity $\tilde{\sigma}(u, v)$ for a given edge (u, v).

A Running Example. Figure 1 shows a running example of how our *EdgeSimStr* implementation works under the unified framework (Algorithm 1). At the current state, 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 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 one to $c_u = 16$. Then, *EdgeSimStr*.find (Algorithm 2) is invoked. 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 vertices in B_2 need to be 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, $\bar{c}_u(B_2) \leftarrow 16$ records 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 scan at B_5 because $\lfloor \frac{16}{32} \rfloor = \lfloor \frac{2}{32} \rfloor$. Next, *EdgeSimStr*.delete 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. As B_3 becomes empty, it is then removed from $\mathcal{B}(u)$. After the re-calculation of the similarities for (u, w_2) and (u, w_3) with *EdgeSimStr*.cal-sim (Algorithm 3), *EdgeSimStr*.insert is invoked to insert w_2 and w_3 to buckets B_4 and B_2 into $\mathcal{B}(u)$, respectively.

Algorithm 2: 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**
   `// c_u has passed a multiple of 2^i since B_i was last visited; check B_i again`
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 3: Our Implementation of *EdgeSimStr*.cal-sim

Input: an edge (x, y)
Output: a $\frac{1}{2}\rho$-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 return** $\tilde{\sigma}(x, y) = 0$;
2 $L \leftarrow$ the number of samples as required by Lemma 5.1; $X \leftarrow 0$;
3 **for** $i = 1, 2, \cdots, L$ **do**
4 flip a coin z such that $\Pr[z = 1] = \frac{n_x}{n_x + n_y}$ and $\Pr[z = 0] = \frac{n_x}{n_x + n_y}$;
5 **if** $z = 1$ **then**
6 uniformly at random pick a vertex $w \in N[x]$;
7 **else**
8 uniformly at random pick a vertex $w \in N[y]$;
9 **if** $w \in N[x] \cap N[y]$ **then**
10 $X \leftarrow X + 1$;
11 $\bar{X} \leftarrow X/L$;
12 **return**

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

Algorithm 4: Our Implementation of *EdgeSimStr*.insert

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

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

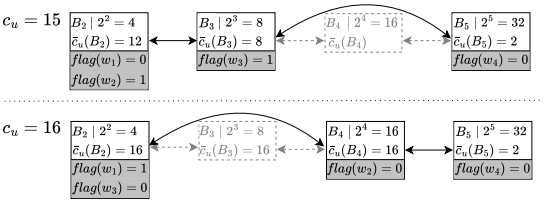


Figure 1: A Running Example of Our *EdgeSimStr*

4.3 Our Implementation of *CoreFindStr*

We adopt the same implementation of BOTBIN [24] for *CoreFindStr*. Specifically, we maintain a so-called Δ-*Table*, denoted by T_Δ, which is an array of $\lceil \frac{1}{\Delta} \rceil$ sorted list of vertices, where $0 < \Delta < 1$ is a *constant* parameter. The i^{th} (for $i = 0, 1, \cdots, \lceil \frac{1}{\Delta} \rceil - 1$) sorted list, denoted by $T_\Delta[i]$, stores all vertices $u \in V$ in *non-ascending* order by $\mu_{u,i}$, where $\mu_{u,i}$ is the number of neighbors $w \in N(u)$ for which the approximate similarity $\tilde{\sigma}(u,w) \geq i\Delta$.

Implementation of *CoreFindStr*.find. Given a similarity threshold ε and parameter μ, it first identifies $i^* = \lfloor \varepsilon/\Delta \rfloor$. Then it retrieves all the core vertices by scanning and reporting the vertices in the sorted list $T_\Delta[i^*]$ until the first vertex x that satisfies $\mu_{x,i^*} < \mu$ or the entire list has been retrieved. In this way, all the core vertices with respect to similarity threshold $\lfloor \varepsilon/\Delta \rfloor \cdot \Delta$ and μ can be found in $O(1 + |V_{\text{core}}|)$ time. However, in return, it introduces an additive absolute error Δ to the approximation factor, resulting in $(\rho + \Delta)$-absolute approximation.

Implementation of *CoreFindStr*.update. Given a vertex x whose sorted neighbor list $N(x)$ is altered, *CoreFindStr*.update(x) maintains all $\lceil \frac{1}{\Delta} \rceil \in O(1)$ sorted lists in T_Δ for x with respect to the possibly updated $\mu_{x,i}$.

5 Theoretical Analysis

In this section, we prove the correctness (Theorem 5.3) of *VD-STAR*, analyze the amortized per-update running time (Lemma 5.5) and the space consumption (Lemma 5.6). Putting these results together constitute the following theorem:

Theorem 5.1. *Our* VD-STAR *algorithm supports all three similarity measurements (Jaccard, Cosine and Dice). It can return a ρ-absolute-approximate clustering result with a high probability, at least $1 - \frac{1}{n}$, for each query, and it 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.*

5.1 Correctness

First, we show that our algorithm maintains a correct ρ-absolute-approximate similarity for each edge at all times.

Theorem 5.2. *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 5.2, it suffices to show these two lemmas :

Lemma 5.1. [*] *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 3 satisfies $|\tilde{\sigma}(u,v) - \sigma(u,v)| \leq \frac{1}{2}\rho$ with probability at least $1 - \frac{1}{2n^4}$.*

Lemma 5.2. [*] *For any $(u,v) \in E$, its approximate similarity $\tilde{\sigma}(u,v)$ must be re-computed by Algorithm 3 before its update affordability $\tau(u,v)$ is fully consumed, i.e., before the arrival of its $\tau(u,v)^{th}$ affecting update, since $\tilde{\sigma}(u,v)$ was last computed.*

Proof of Theorem 5.2. The proofs of Lemmas 5.1 and 5.2 can be found in our technical report [26]. Suppose these two lemmas 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 3. Without loss of generality, we assume that there are at most $M \leq n^2$ updates, because otherwise, we can re-compute the approximate similarity for every edge in the current graph and hence, reset the number of updates M to 0. Therefore, Algorithm 3 is invoked for at most $2n^3$ times. According to Lemma 5.1, each invocation fails with probability at most $\frac{1}{2n^4}$. By Union Bound, the whole process succeeds with probability at least $1 - \frac{1}{n}$. □

By Theorem 5.2 and the fact that *VD-STAR* adopts the Δ-Table for *CoreFindStr*, Theorem 5.3 immediately follows, which completes the correctness proof for *VD-STAR*.

Theorem 5.3. 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.

5.2 Running Time and Space Cost Analysis

Query Running Time. According to the implementation of our *CoreFindStr*, given parameters ε and μ, all core vertices can be retrieved in $O(1 + |V_{\text{core}}|)$ time. All similar edges incident on a core vertices can be obtained by scanning the sorted neighbor list of each core vertex in $O(m_{cr} + 1)$ time. By Observation 1, the overall running time of each query is bounded by $O(m_{cr} + 1)$, because $m_{cr} \geq |V_{\text{core}}|$. Thus, the following lemma holds:

Lemma 5.3. VD-STAR *can answer each query in $O(m_{cr} + 1)$ 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* approximate similarity calculations for (u,v). Consider

the moment when the similarity of an edge (u, v) needs to be recomputed; according to the unified framework (Algorithm 1), the maintenance for (u, v) involves the following operations:

- a similarity calculation which takes $cost_{\text{EC}}$;
- an invocation of deletion 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 *EdgeSimStr.insert* 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; 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 to 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:

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

Next, we analyze $cost_{\text{EC}}$, $cost_{\text{ED}}$ and $cost_{\text{EI}}$, respectively. For the cost of similarity calculation, $cost_{\text{EC}}$, by Lemma 5.1, we know that $L \in O(\log n)$ samples suffice. Each sample checks 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 $cost_{\text{ED}}$, 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 costs 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. By the following Fact 1, each of these operations can be performed in $O(1)$ expected time. Thus, $cost_{\text{EI}} + cost_{\text{ED}} \in O(1)$ in expectation.

Fact 1 ([6, 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 the cost bounds above into Expression (1), we have:

Lemma 5.4. *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. The amortized per-update cost follows from this lemma whose proof can be found in our technical report [26]:

Lemma 5.5. [*] *The amortized cost of each update is bounded by $O(\log n)$ in expectation.*

Space Consumption. For each $u \in V$, the space consumption of (i) the *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 $O(n_u)$.

Lemma 5.6. *The overall space consumption of* VD-STAR *is bounded by $O(n + m)$ at all times.*

6 Optimizations

We introduce two optimizations, namely *VD-STAR-NoT* and *VD-STAR-μT* 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 $O(m_{cr} + 1)$. If we relax this query bound, 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.

6.1 *VD-STAR-NoT*

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 $\geq \varepsilon$ 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 $O(m_{cr} + 1)$ 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.

6.2 *VD-STAR-μT*

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 $O(m_{cr} + 1)$ as desired. Otherwise, we just run the above version of *VD-STAR* with no *CoreFindStr* to answer the query.

7 Experiments

7.1 Experimental Settings

Datasets. We evaluate our algorithms on nine real-world datasets from the Stanford Network Analysis Project [9] and Network Repository [17] which are also used in the baseline papers [18, 20, 24]. We present the results on the six larger datasets and defer the rest three to Appendix C as the results resemble. Following previous works [18, 24], we treat all graphs as undirected and remove all the self-loops. Table 2 summarizes the datasets.

Competitors. We study the performance of our three algorithms: *VD-STAR*, *VD-STAR-NoT* and *VD-STAR-μT*, which are respectively denoted by ***Ours***, ***Ours-NoT*** and ***Ours-μT*** for short. The source code and technical report can be found in [26]. We compare these algorithms with the SOTA exact and approximate algorithms **GS*-Index** [20] and **BOTBIN** [24]. In *Ours-μT*, only a μ-Table with $\mu_{\max} = 15$ is constructed. Since *DynELM* does not support Structural Clustering query when the parameters are given on the fly (which is our problem setting), and hence it is omitted in our experiments.

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, 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-existing 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.

Due to space limit, we put the experimental results under Cosine and Dice similarities to Appendix C.3, which show similar results to that under Jaccard.

7.2 Study on Update Efficiency

Average Update Time with Default Parameters. We first study the average running time of processing updates. As shown in Figure 2a, 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 7.3. It outperforms SOTA methods by up to 193 times on `soc-Friendster` vs. GS*-Index.

Table 2: Dataset Summary

Datasets	n($\times 10^6$)	m($\times 10^6$)	$\bar{d}$	Domain
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
soc-Friendster	65.61	1,806.07	55.13	Social network
web-2012	90.32	1,940.85	42.91	Website hyperlink

Memory Consumption. As shown in Figure 2b, 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.

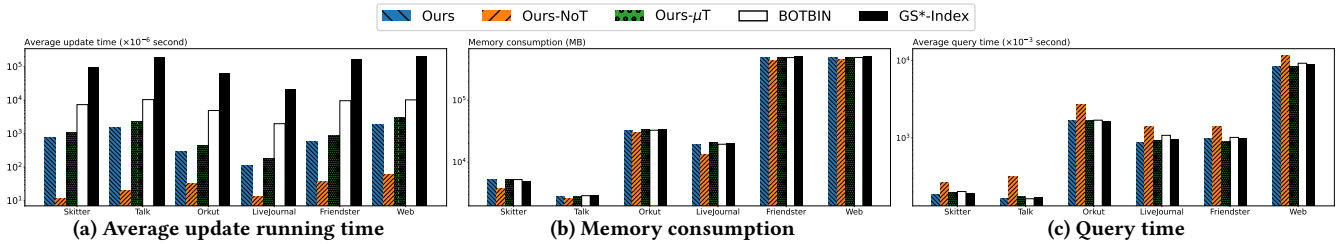
Impact of Target Overall Approximation ρ^*. To test how ρ^* affects the update time, we vary ρ^* from 0.001 to 0.1. GS*-Index is excluded from this experiment as it is an exact algorithm. As shown in Figure 3, the update time decreases when ρ^* grows, as expected. A larger ρ^* value leads to smaller 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 69×10^{-6} second on a graph with 1.9 billion 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.

Impact of Update Distributions. As shown in Figure 3, the average update times of all algorithms increase as updates follow more skewed distributions (from RR to DR 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 6,098 times on **RR**, 9,315 times on **DR**, and 4,857 times on **DD**. We defer the discussion of the impact of the deletion-to-insertion ratio (η) to Appendix C.2 due to page limit. When the insertion rate increases, the performance changes are analogous to those observed between smaller and larger graphs.

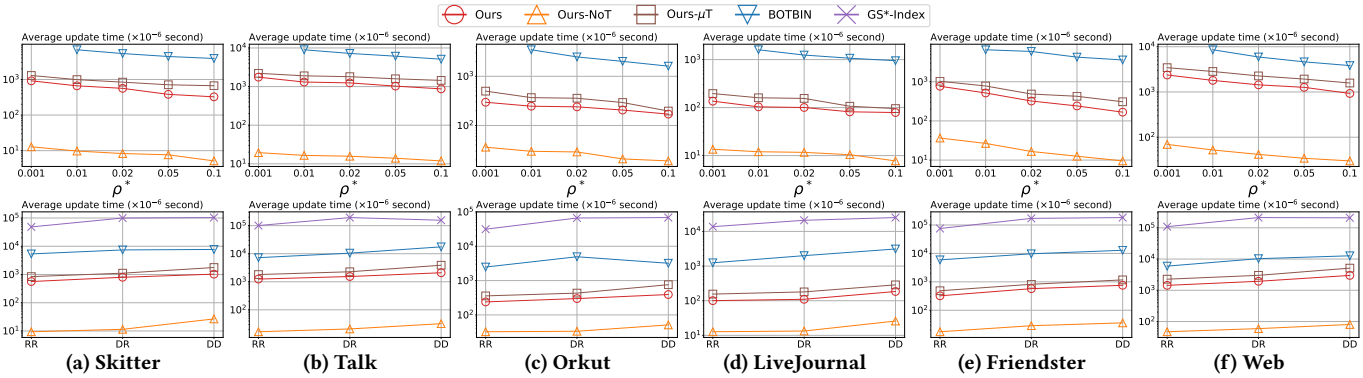
7.3 Study on Query Efficiency

The query efficiency results are shown in Figure 2c. Three observations are made: (1) GS*-Index, BOTBIN, and *Ours* exhibit similar query times as they are all bounded by $O(m_{cr} + 1)$, linear to the size of the clustering result graph. *Ours-μT*, utilizing a fixed-size μ-table,



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

Figure 2: Update processing and query processing performance results



(a) Skitter (b) Talk (c) Orkut (d) LiveJournal (e) Friendster (f) Web

Figure 3: Average update running time vs. ρ^* (upper) and update distribution (lower)

Table 3: Clustering quality results

	$\rho = 0.02$				$\rho = 0.1$			
	BOTBIN		***Ours***		**BOTBIN**		***Ours***	
	ARI ↑	**MLR ↓**	**ARI ↑**	**MLR ↓**	**ARI ↑**	**MLR ↓**	**ARI ↑**	**MLR ↓**
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%
soc-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%**
soc-Friendster	0.9944	**0.64%**	**0.9947**	0.69%	0.9636	6.73%	**0.9673**	**6.40%**
web-2012	**0.9928**	0.60%	0.9926	**0.56%**	**0.9609**	**9.79%**	0.9607	9.91%

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 the $O(m_{cr}+1)$-time methods, as evidenced by the results showing very similar query times. (2) Across all datasets, *Ours-NoT*'s query times are within the same magnitude as the other algorithms. (3) In structural clustering problems, the number of clustering result vertices (n_{cr}) has substantial 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.

7.4 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**) [8]. 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 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 0.7% across all datasets, with ARI values ranging from 0.9926 to 0.9996. Even with $\rho^* = 0.1$, *Ours* maintains an average ARI of at least 0.9548 (on `soc-Orkut`) and MLR of at most 9.91% (on `web-2012`). *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 on real datasets.

8 Conclusion

We proposed an algorithm called *VD-STAR* for the problem of *Dynamic Structural Clustering for All Parameters*. *VD-STAR* can return a ρ-absolute-approximate clustering result with a high probability for every query in $O(m_{cr}+1)$ time, and it can process each update in $O(\log n)$ amortized expected time, while its space consumption is bounded by $O(n+m)$ at all times. The algorithm works well with Jaccard, Cosine and Dice similarity measurements and supports arbitrary updates. It significantly improves the state-of-the-art approximate algorithm BOTBIN which has an $O(\log^2 n)$ expected time per-update for *random* updates under Jaccard similarity only. We evaluate our algorithm on real datasets, which shows strong empirical results in update and query efficiency, clustering result quality, and robustness in handling various update distributions.

Acknowledgments

In this work, Junhao Gan is in part supported by ARC DP230102908, Jianzhong Qi in part supported by ARC FT240100170, DP240101006, and DP230101534, Zhifeng Bao supported by ARC FT240100832 and DP240101211 in part. Sibo Wang is supported by the RGC GRF grant (No. 14217322) and the Tencent WeChat Rhino-Bird Focused Research Program. We also would like to thank the anonymous reviewers for their valuable feedback and suggestions.

References

[1] 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.

[2] Sudarshan S Chawathe. 2019. Clustering blockchain data. *Clustering Methods for Big Data Analytics: Techniques, Toolboxes and Applications* (2019), 43–72.

[3] Ricky Maulana Fajri, Yulong Pei, Lu Yin, and Mykola Pechenizkiy. 2024. A Structural-Clustering Based Active Learning for Graph Neural Networks. In *International Symposium on Intelligent Data Analysis*. 28–40.

[4] ZhengZhao Feng, Rui Wang, TianXing Wang, Mingli Song, Sai Wu, and Shuibing He. 2024. A comprehensive survey of dynamic graph neural networks: Models, frameworks, benchmarks, experiments and challenges. *arXiv preprint arXiv:2405.00476* (2024).

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

[6] Junhao Gan, Seeun William Umboh, Hanzhi Wang, Anthony Wirth, and Zhuo Zhang. 2024. Optimal dynamic parameterized subset sampling. *Proceedings of the ACM on Management of Data* 2, 5 (2024), 209:1–209:26.

[7] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–10.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] Amrutanshu Panigrahi, Ajit Kumar Nayak, Rourab Paul, Bibhuprasad Sahu, and Shashi Kant. 2022. CTB-PKI: Clustering and trust enabled blockchain based PKI system for efficient communication in P2P network. *IEEE Access* 10 (2022), 124277–124290.

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

[15] 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.

[16] 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.

[17] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *AAAI*.

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

[19] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2021. Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey. *IEEE Access* 9 (2021), 79143–79168.

[20] 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.

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

[22] Yaming Yang, Ziyu Guan, Zhe Wang, Wei Zhao, Cai Xu, Weigang Lu, and Jianbin Huang. 2022. Self-supervised heterogeneous graph pre-training based on structural clustering. In *NeurIPS*. 16962–16974.

[23] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: graph learning framework for dynamic graphs. In *KDD*. 2358–2366.

[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.

[26] Zhuowei Zhao. 2025. Source code and technical report. https://github.com/alvinzhaowei/DynStrClu

A Proofs for Section 2 (Problem Formulation)

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 $O(m_{cr}+1)$ 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 $O(m_{cr}+1)$. □

B Proofs for Section 4 (Our Versatile DynStrClu Algorithm)

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

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

Lemma 4.1. *For any edge (u,v), let $\sigma(u,v)$ be the true similarity over time, i.e., $\sigma(u,v)$ changes under updates. Consider the moment when a $\frac{1}{2}\rho$-absolute approximate similarity $\tilde{\sigma}(u,v)$, which satisfies $|\tilde{\sigma}(u,v) - \sigma(u,v)| \leq \frac{1}{2}\rho$, is just computed. Since then, (u,v) can afford at least $\tau(u,v) \geq \frac{1}{4}\rho^2 d_{\max}(u,v) \in \Omega(d_{\max}(u,v))$ affecting updates (which are incident on either u or v) such that $\tilde{\sigma}(u,v)$ is still "valid" in the sense that $|\tilde{\sigma}(u,v) - \sigma(u,v)| \leq \rho$. Here, $d_{\max}(u,v) = \max\{n_u, n_v\}$.*

Proof. 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.

We prove that this lemma holds for the following two cases separately.

Case 1: $n_u \leq \frac{1}{4}\rho^2 n_v$**.** According to Algorithm 3, we set $\tilde{\sigma}(u,v) = 0$ for all three similarity measurements in this case. By Observation 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 Lemma 4.1 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 show the increment case only and the decrement case is analogous and thus, omitted here.

It is easy to verify that, 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 $\sigma(u,v)$. Each update of this kind increases $I(u,v)$ by one and either n_u or n_v by one. Without loss of generality, suppose that n_u and n_v are increased by $t-b$ and b, respectively, after t affecting updates.

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

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

Therefore, for any of these similarity measurements, the update affordability $\tau(u,v) \geq t = \frac{1}{4}\rho^2 n_v \in \Omega(d_{\max}(u,v))$ holds for Case 2. This completes the whole proof for Lemma 4.1. □

C Additional Experimental Results

C.1 Experimental Results on More Datasets

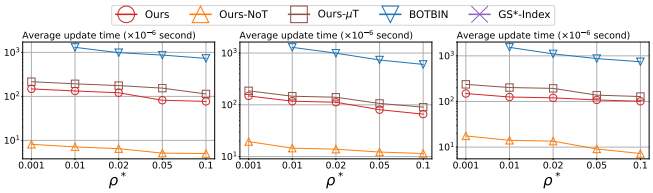
Table 4 summarizes the additional 3 datasets tested. The results are presented in Figure 4, Figure 5, Figure 6, and Table 5 regarding the impact on ρ^*, the impact on update distribution, update and query performance, and the clustering quality.

Table 4: Dataset Summary

Datasets	n($\times 10^6$)	m($\times 10^6$)	$\bar{d}$	Domain
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

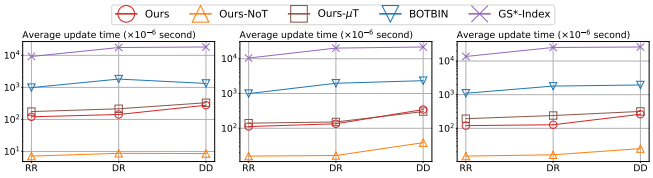
Table 5: Clustering quality results

	$\rho = 0.02$				$\rho = 0.1$			
	BOTBIN		*Ours*		BOTBIN		*Ours*	
	ARI ↑	MLR ↓	ARI ↑	MLR ↓	ARI ↑	MLR ↓	ARI ↑	MLR ↓
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%



(a) Google (b) Topcats (c) Pokec

Figure 4: Average update running time vs. ρ^*

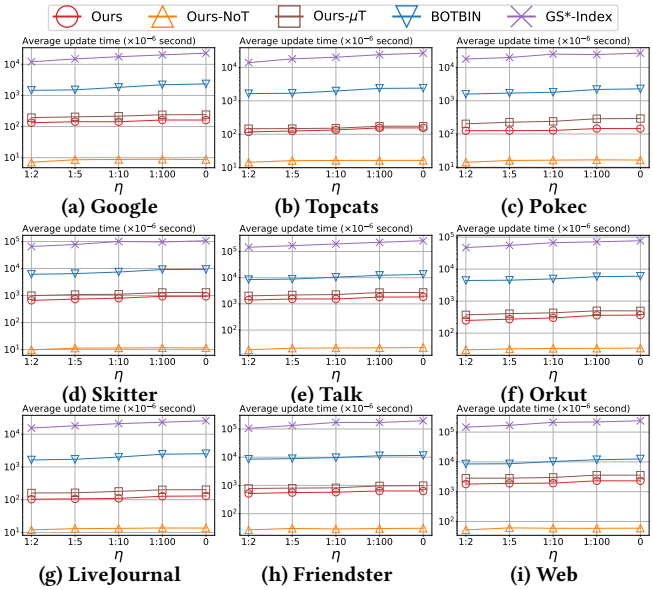


(a) Google (b) Topcats (c) Pokec

Figure 5: Average update time vs. update distribution

C.2 Impact of Deletion-to-Insertion Ratio

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.

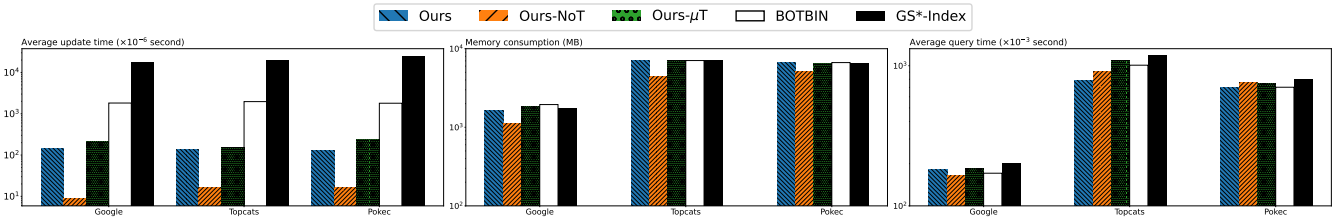


(a) Google (b) Topcats (c) Pokec

(d) Skitter (e) Talk (f) Orkut

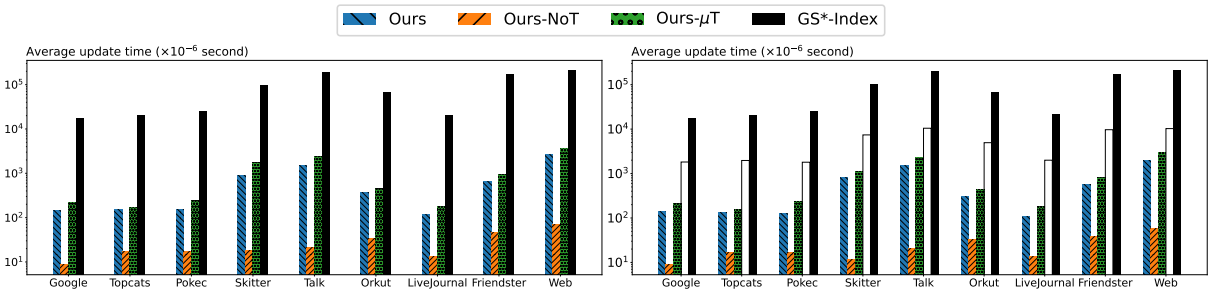
(g) LiveJournal (h) Friendster (i) Web

Figure 7: Average update time vs. η



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

Figure 6: Update processing and query processing performance results



(a) Cosine (b) Dice

Figure 8: Average update running time on Cosine and Dice

C.3 Experiments on DICE and Cosine

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

Cosine. As Figure 8a shows, the comparative pattern in update running time is similar to that observed under the Jaccard similarity setting (Figure 2a above). However, the update time of *Ours-NoT* shows a slight increase compared with the Jaccard similarity setting (Figure 2a 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 5.1 and Section 4.1.

The clustering quality results are shown in Table 6. 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 8b (update running time) and Table 6 (clustering quality). As expected, our algorithms show similar performance as that on Jaccard.

Table 6: 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 ↓**
`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%
`soc-Friendster`	0.9947	0.69%	0.9653	0.94%	0.9937	0.62%
`web-2012`	0.9926	0.56%	0.9551	1.00%	0.9958	0.57%

D Missing Pseudo Code

In this section, we include the missing pseudo code for our implementation of *EdgeSimStr*.

Algorithm 5: Our Implementation of *EdgeSimStr*.delete

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

remove v from its corresponding bucket B_i;

if B_i *becomes empty* **then**

  remove B_i from $\mathcal{B}(u)$;

perform the above steps for v symmetrically;