

Dynamic Structural Clustering on Graphs

BOYU RUAN, The University of Queensland, Australia

JUNHAO GAN, The University of Melbourne, Australia

HAO WU, The University of Melbourne, Australia

ANTHONY WIRTH, The University of Melbourne, Australia

Structural Clustering (StrClu) is one of the most popular graph clustering paradigms. In this paper, we consider *StrClu* under two commonly adapted similarities, namely Jaccard similarity and cosine similarity on a *dynamic* graph, $G = \langle V, E \rangle$, subject to edge insertions and deletions (updates). The goal is to maintain certain information under updates, so that the *StrClu* clustering result on G can be retrieved in $O(|V| + |E|)$ time, upon request. The state-of-the-art worst-case cost is $O(|V|)$ per update; we improve this update-time bound *significantly* with the ρ -approximate notion. Specifically, for a specified failure probability, δ^* , and *every* sequence of M updates (no need to know M 's value in advance), our algorithm, *DynELM*, achieves $O(\log^2 |V| + \log |V| \cdot \log \frac{M}{\delta^*})$ amortized cost for each update, *at all times* in linear space. Moreover, *DynELM* provides a provable “sandwich” guarantee on the clustering quality at all times after *each update* with probability at least $1 - \delta^*$. We further develop *DynELM* into our ultimate algorithm, *DynStrClu*, which also supports *cluster-group-by* queries. Given $Q \subseteq V$, this puts the non-empty intersection of Q and each *StrClu* cluster into a distinct group. *DynStrClu* not only achieves *all* the guarantees of *DynELM*, but also runs *cluster-group-by* queries in $O(|Q| \cdot \log |V|)$ time. We demonstrate the performance of our algorithms via extensive experiments, on 15 real datasets. Experimental results confirm that our algorithms are up to *three orders of magnitude* more efficient than state-of-the-art competitors, and still provide quality structural clustering results. Furthermore, we study the difference between the two similarities w.r.t. the quality of approximate clustering results.

Additional Key Words and Phrases: Structural Clustering; Dynamic Graphs; Algorithms

ACM Reference Format:

Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2018. Dynamic Structural Clustering on Graphs. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 37 pages. <https://doi.org/10.1145/1122445.1122456>

1 INTRODUCTION

Clustering on graphs is a fundamental and highly applicable data-mining task [33]. The graph vertices are assigned to clusters so that similar vertices are put into the same cluster, while dissimilar vertices are separated from each other, in different clusters. While most clustering approaches [1, 9, 11, 15, 16, 28, 29, 32, 37] aim to partition the vertices into *disjoint* sets, *Structural Clustering* (aka *StrClu*, the focus of this paper) [6, 34, 39, 40, 42] allows clusters to overlap with each other, so that a vertex might be assigned to multiple clusters. In particular, in *StrClu*, there are different roles each vertex might play in the clustering. Some vertices are *core* members of the clusters, while others may be *noise* (i.e., belonging to no cluster), and still others might be *hubs* (i.e., being members of, and hence bridging, multiple clusters).

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 ACM 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.

© 2018 Association for Computing Machinery.

Manuscript submitted to ACM

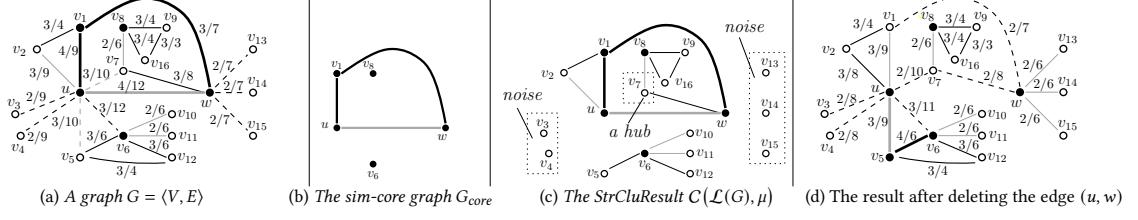


Fig. 1. The *StrCluResult* with $\epsilon = 1/3$ and $\mu = 3$. In (a) and (d), the Jaccard Similarities between the endpoints are shown beside the edges. Similar edges are shown as solid lines and dissimilar edges appear as dashed lines. The core and non-core vertices are in black and white, respectively. All the *sim-core* edges are further highlighted in bold. Moreover, the edges whose labels are *does not matter* under the ρ -approximate notion with $\rho = 1/6$ are shown in gray.

Structural Clustering. The success of structural clustering arises directly from a family of careful definitions, governing the roles of the edges and vertices. Given an undirected graph $G = (V, E)$, the neighbourhood of a vertex u is defined as the set of vertices adjacent to u in G , plus u itself. In *StrClu*, each edge in E has a *label*: either *similar* or *dissimilar*. An edge (u, v) is labelled as similar if and only if the *structural similarity* (e.g., the *Jaccard similarity* or cosine similarity) between the *neighbourhoods* of u and v is at least a certain specified *similarity threshold*, ϵ . If at least a specified integer μ of similar edges are incident on vertex u , then u is a *core* vertex: consequently, all vertices that share a similar edge with u are put in the same cluster as u . Of course, another core vertex v might be “similar-adjacent” to u , and thus in its cluster: repeating this principle, all vertices similar-adjacent to v are in the same cluster as v and hence as u . This chain effect continues until no more vertices are added to u ’s cluster, resulting in a *StrClu* cluster. We repeat the above process until all the core vertices are in some cluster and return the *StrClu* result. As some non-core vertex w may be “similar-adjacent” to multiple core vertices that are in different clusters, w is thus included in multiple clusters, as a *hub* bridging these clusters. On the other hand, a non-core vertex also possibly belongs to no cluster, and thus becomes *noise* (i.e., an outlier). These different roles information of the vertices and edges have greatly enriched the structural information of a graph. The power of *StrClu* arises from these specific vertex roles.

An Example. Figure 1(a) shows an example of Structural Clustering under Jaccard similarity, where all the similar (respectively., dissimilar) edges appear as solid (respectively., dashed) edges; and all the core (respectively., non-core) vertices are coloured black (respectively., white). Since u is a core, each vertex in u ’s neighbourhood sharing a similar edge with u , i.e., $\{v_1, v_2, w\}$, is added to the same cluster as u . Due to the core vertex w , the non-core vertex v_7 is further added to the cluster. Thus, $C_1 = \{u, w, v_1, v_2, v_7\}$ is a *StrClu* cluster. Likewise, $C_2 = \{v_8, v_9, v_{16}, v_7\}$ and $C_3 = \{v_6, v_5, v_{10}, v_{11}, v_{12}\}$ are the other two *StrClu* clusters, whereas $v_3, v_4, v_{13}, v_{14}, v_{15}$ are *noise*. Finally, observe that v_7 is a *hub* vertex, belonging to both clusters C_1 and C_2 .

Applications. Before continuing with our technical description of the problems in this paper, we first consider the significance of *StrClu*. Since its introduction [40] in 2007, *StrClu* has not only attracted significant follow-up work¹, but has also served a wide range of real-world applications. For example, *StrClu* is an essential component of the *atBioNet*, developed by the U.S. Food and Drug Administration’s (FDA) National Center for Toxicological Research (NCTR) [10, 25]. *atBioNet* is a web-based tool for genomic and proteomic data, that can perform network analysis follower by biological interpretation for a list of seed proteins or genes (i.e., proteins or genes provided by user). Here, *StrClu* supports identifying functional modules in protein-protein interaction networks and enrichment analysis. Another example is in community detection [30], where the users are modelled as vertices and the following relationships are modelled as

¹Over 850 citations in Google Scholar, as at March 2021.

edges. Each cluster in the clustering results of *StrClu* can be regarded as a community in the social network. Given a collection of tagged photos, *StrClu* can be utilized to identify landmarks and events [31] by applying *StrClu* on a *hybrid similarity image graph*, constructed by taking both visual similarity and tag similarity into consideration. One interesting application of *StrClu* is detecting frauds on blockchain data [7]. The noise information of *StrClu* result is deployed on a graph constructed using the features extracted from blockchain data. The outliers in the *StrClu* result are regarded as frauds that need to be paid attention to.

The Dynamic Scenario. While the importance and usefulness of *StrClu* is witnessed from its various applications, new challenges arise from the *dynamic* nature (subject to updates) of contemporary graph data. The significance and application of *StrClu* is only increased by considering dynamic graphs, where edges might be added or deleted. The consequent imperative research is to design highly efficient algorithms that can handle updates to the graph G so that queries on the *StrClu* result are answered efficiently, i.e., without re-computing from scratch. By maintaining the edge labels under updates, the *StrClu* result can be obtained in $O(n + m)$ time, where n and m are the *current* numbers of vertices and edges in G . We refer to this linear-time problem as *dynamic StrClu*.

The Challenges. For dynamic *StrClu*, there are two state-of-the-art algorithms, *pSCAN* [6] and *hSCAN* [39], which can process each update within $O(n)$ and $O(n \log n)$ time, respectively. While these bounds are unfortunately too high for an update (on large graphs with millions of vertices), this $O(n)$ bound seems unlikely to be improved. To see this, when an edge (u, w) is inserted, labelling (u, w) may require computing the structural similarity between the neighbourhoods of u and w , and hence, may require $\Omega(n)$ time in the worst case. Worse still, as shown in Figure 1(a), after removing the edge (u, w) , the label of every edge incident on w flips (as shown in Figure 1(d)). When this edge is re-inserted in Figure 1(d), those edge labels revert. Consequently, the maintenance on these labels already takes up to $O(n)$ time when the degree of w is large.

Our Solutions. We can, however, reduce the running time of updates from the bound of $O(n)$ to roughly $O(\log^2 n)$ under both Jaccard similarity and cosine similarity. There are two (reasonable) trade-offs: (i) an approximate notion of the edge labels, and (ii) amortized rather than worst-case analysis of the update time. Specifically, we adapt the ρ -approximate notion to edge labels in *StrClu*: this notion circumvents computational hardness in other key clustering problems [12–14]. Experiments confirm that our approach loses only a little in clustering quality but provides a dramatic improvement (i.e., 1000-fold) in update efficiency.

Cluster-group-by Queries. In addition to simply returning entire clustering results, we further enhance our solution to support the so-called *cluster-group-by* queries [12], which are examples of traditional group-by queries in database systems. Given a subset $Q \subseteq V$, the cluster-group-by query of Q asks to group the vertices in Q by the *identifiers* of the clusters (if any) containing them. Consider Figure 1(a), for $Q = \{u, v_7, v_9, v_{14}\}$, the cluster-group-by query returns two groups: $\{u, v_7\}$ and $\{v_7, v_9\}$. This is because both u and v_7 belong to C_1 , meanwhile, both v_7 and v_9 belong to C_2 , and v_{14} is noise. As a special case, when $Q = V$, the cluster-group-by query is equivalent to retrieving the whole *StrClu* result, where each group is exactly a cluster. Thus, the cluster-group-by query is a more general form of clustering query. Furthermore, since $|Q|$ is typically much smaller than $|V|$, an efficient algorithm should be able to answer the query much faster, with an output cost depending on $|Q|$ rather than $|V|$. We note that cluster-group-by queries are not only applicable in *all* general clustering applications, but also more favourable in scenarios where users are focused on the part of clustering results *corresponding to a certain set of vertices*.

Our Contributions. This article delivers the following contributions:

- We first adapt Jaccard similarity as our definition of structural similarity. Under the ρ -approximate notion, we propose the *dynamic edge labelling maintenance (DynELM)* algorithm for maintaining edge labels under updates. Specifically, for a specified failure probability, δ^* , for every sequence of M updates, where M 's value need not be known in advance, *DynELM* guarantees:
 - the amortized update cost is $O(\log^2 n + \log n \cdot \log \frac{M}{\delta^*})$, significantly better than the state-of-the-art $O(n)$ bound;
 - with probability at least $1 - \delta^*$, the edge labels are correct (under the ρ -approximate notion) *all the time*, i.e., the clustering result is correct after *each* update;
 - the space consumption is $O(n + m)$, linear in the graph size;
 - on request, the *StrClu* result can be retrieved in $O(n + m)$ time.
- To support fast cluster-group-by queries, we introduce our ultimate algorithm: the *Dynamic StrClu* algorithm (*DynStrClu*). Although *DynStrClu* maintains some extra data structures, it not only achieves all the guarantees provided by *DynELM*, but also answers every cluster-group-query Q in $O(|Q| \cdot \log n)$ time, substantially less than $O(n + m)$ when $|Q|$ is far smaller than n .
- We conduct extensive experiments on 15 real datasets, where the largest dataset, *Twitter*, contains up to 1.2 billion edges. The experimental results confirm that our *DynELM* and *DynStrClu* algorithms are up to *three-orders-of-magnitude* (i.e., 1000 \times) faster on updates than the state-of-the-art algorithms, while still returning a high-quality *StrCluResult* under Jaccard similarity.
- We extend our *DynELM* and *DynStrClu* algorithms to support cosine similarity. With non-trivial analysis, we prove that the amortized time cost and space consumption remains the same as these algorithms under Jaccard similarity.
- We visualise the clustering results under both Jaccard similarity and cosine similarity using *Gephi* [2], and confirm that the clustering results of Structural Clustering are meaningful and human understandable.
- Finally, we conduct extra experiments on 5 representative datasets. The experimental results confirm that our *DynELM* and *DynStrClu* algorithms still have outstanding performance and return a high-quality *StrCluResult* under cosine similarity.

Paper Organization. Section 2 introduces the preliminaries. Section 3 defines the problems. In Section 4, we design a similarity estimator. In Sections 5 and 6, we propose the *DynELM* algorithm and prove its theoretical guarantees. We discuss *DynStrClu* in Section 7. Section 8 presents the extension of our *DynELM* and *DynStrClu* algorithms under cosine similarity and theoretical analysis. Section 9 shows experimental results under both Jaccard similarity and cosine similarity. Finally, Section 10 concludes the paper.

2 PRELIMINARIES

2.1 Structural Clustering Setup

Basic Definitions. Consider an undirected graph $G = \langle V, E \rangle$ with $n = |V|$ vertices and $m = |E|$ edges. Two vertices $u, v \in V$ are *neighbors* of each other if they share an edge, $(u, v) \in E$. For a vertex $v \in V$, the *neighborhood* of v , denoted by $N[v]$, is defined as the set of all neighbors of v , as well as v itself, i.e., $N[v] = \{u \in V \mid (u, v) \in E\} \cup \{v\}$. The *degree* of v , denoted by $d[v]$, is the number of neighbors of v and hence, $d[v] = |N[v]| - 1$. Each edge $(u, v) \in E$ can be labelled as either *similar* or *dissimilar*, and (u, v) is called a *similar* or *dissimilar* edge, respectively. We define an *edge labelling* of G , denoted by $\mathcal{L}(G)$, as a function: $V \times V \rightarrow \{\text{similar}, \text{dissimilar}\}$, specifying a label for each edge.

Given a *constant* integer *core threshold*, $\mu \geq 1$, and an edge labelling, $\mathcal{L}(G)$, we introduce the following definitions.

Similar Neighbors. If an edge $(u, v) \in E$ is similar, then each of u and v is a *similar neighbor* to the other.

Core Vertex. A vertex $u \in V$ is a *core vertex* if u has at least μ similar neighbors. Otherwise, u is a *non-core vertex*.

Sim-Core Edge. An edge $(u, v) \in E$ is called a *sim-core edge* if (u, v) is similar and both u and v are core vertices, e.g., (u, w) is a sim-core edge, while (u, v_2) and (u, v_6) are not because v_2 is a non-core and (u, v_6) is dissimilar.

Sim-Core Graph. The *sim-core graph* G_{core} consists of all the core vertices and all the sim-core edges, e.g., Figure 1(b).

StrClu Cluster. There is a one-to-one mapping between the connected components (CCs) of G_{core} and the *StrClu* clusters. For each CC of G_{core} , its corresponding *StrClu* cluster is the set of all the vertices in G that are similar to some (core) vertex in this CC. The G_{core} in Figure 1(b) has three connected components: $\{u, w, v_1, v_2, v_7\}$, $\{v_8\}$ and $\{v_6\}$. The three corresponding *StrClu* clusters are: $C_1 = \{u, w, v_1, v_2, v_7\}$, $C_2 = \{v_8, v_9, v_{16}, v_7\}$ and $C_3 = \{v_6, v_5, v_{10}, v_{11}, v_{12}\}$, shown in Figure 1(c). In particular, v_7 is a *hub* belonging to both the clusters C_1 and C_2 , while vertices v_3, v_4, v_{13}, v_{14} and v_{15} are *noise*, belonging to no cluster.

StrCluResult. The *StrCluResult* on G , for the parameter μ , and the labelling $\mathcal{L}(G)$, is the set of all *StrClu* clusters, denoted by $C(\mathcal{L}(G), \mu)$.

FACT 1. Given μ and $\mathcal{L}(G)$, the *StrCluResult* $C(\mathcal{L}(G), \mu)$ is unique and can be computed in $O(n + m)$ time.

The StrClu Problem. Denote the similarity between a pair of vertices $u, v \in V$ by $\sigma(u, v)$. If u and v are not adjacent, then $\sigma(u, v) = 0$. Otherwise, $\sigma(u, v)$ is measured by

$$\sigma(u, v) = \frac{|N[v] \cap N[u]|}{|N[v] \cup N[u]|},$$

under *Jaccard similarity* between their neighbourhoods; or

$$\sigma(u, v) = \frac{|N[v] \cap N[u]|}{\sqrt{d[u] \cdot d[v]}},$$

under *cosine similarity* between their neighbourhoods. In the following contents, we first focus on Jaccard similarity and use Jaccard similarity as our definition of structural similarity due to its simpler form. We defer the discussion on cosine similarity to Section 8.

Given a *constant similarity threshold* $\epsilon \in (0, 1]$, $\mu \geq 1$ and G , the *StrClu* problem computes the *StrCluResult*, $C(\mathcal{L}_\epsilon(G), \mu)$, with respect to a *valid* edge labelling $\mathcal{L}_\epsilon(G)$, viz.

Definition 2.1 (Valid Edge Labelling). Edge labelling $\mathcal{L}_\epsilon(G)$ is *valid* if and only if for every edge (u, v) , $\mathcal{L}_\epsilon(G)(u, v) = \text{similar} \iff \sigma(u, v) \geq \epsilon$.

2.2 Related Work

The *StrClu* problem was first proposed in [40] and the *SCAN* algorithm was proposed for solving the problem. *SCAN* computes the similarities between the endpoints of each edge in G , and thus, its worst-case running time is bounded by $O(m^{1.5})$. While there are a considerable number of follow-up works on *StrClu*, such as *SCAN++* [34], *pSCAN* [6] and *hSCAN* [39], they are all heuristic and none of them is able to break the $O(m^{1.5})$ -time barrier.

Lim et al. proposed a related algorithm, called *linkSCAN*, and its approximate version *linkSCAN** [24]. The (structural) clustering results computed by these algorithms are on a transformation of the original graph (i.e., not on the original graph). Thus, there is no guaranteed connection to their clustering results and those of *SCAN*. Moreover, the running time of these two algorithms is $O(n \cdot d^2)$, where d is the average degree of all the vertices, and hence, in the worst case, the bound still degenerates to $O(n^3)$.

The problem becomes more challenging when the graph is subject to edge insertions and deletions. *pSCAN* and *hSCAN* are two state-of-the-art algorithms that can support updates and can return the *StrCluResult* in $O(n + m)$ time upon request, where m is the current number of edges in the graph. For an update (either insertion or deletion) with an edge (u, w) , both *pSCAN* and *hSCAN* need to retrieve $N[u]$ and $N[w]$ in the worst case. The update cost of *pSCAN* is bounded by $O(n)$, while *hSCAN* requires $O(n \log n)$ time, as it aims for a more general purpose, of supporting *StrCluResult* reporting in $O(n + m)$ time with ϵ and μ given on the fly.

2.3 The ρ -Approximate Notion

The notion of ρ -approximation was initially proposed to circumvent the computational hardness in some clustering problems [12–14]. We adopt this notion to relax slightly the validity requirement (Definition 2.1) for edge labellings. We introduce an additional *constant* parameter, $\rho \in [0, \min\{1, \frac{1}{\epsilon} - 1\}]$, where the value range of ρ is intentionally set to ensure: (i) $(1 - \rho)\epsilon > 0$ and (ii) $(1 + \rho)\epsilon \leq 1$.

Definition 2.2 (Valid ρ -Apprxomiate Edge Labelling). Given $\rho \in [0, \min\{1, \frac{1}{\epsilon} - 1\}]$, an edge labelling $\mathcal{L}_{\epsilon, \rho}(G)$ is a valid ρ -approximate edge labelling, if for every edge $(u, v) \in E$,

- if $\sigma(u, v) \geq (1 + \rho)\epsilon$, (u, v) *must be* labelled as similar;
- if $\sigma(u, v) < (1 - \rho)\epsilon$, (u, v) *must be* labelled as dissimilar;
- for every other $\sigma(u, v)$ value, between $(1 - \rho)\epsilon$ and $(1 + \rho)\epsilon$, the label of (u, v) *does not matter*, namely, it is allowed to be either similar or dissimilar. As shown in Figure 1, the edges in color gray fall in this does-not-matter case.

Two observations are worth noticing here. First, when $\rho > 0$, due to the “does-not-matter” case, there may exist multiple valid ρ -approximate edge labellings, i.e., multiple valid $\mathcal{L}_{\epsilon, \rho}(G)$ ’s. Nonetheless, given $\mathcal{L}_{\epsilon, \rho}(G)$, by Fact 1, the *StrCluResult* $C(\mathcal{L}_{\epsilon, \rho}(G), \mu)$ is still uniquely defined. Second, only the edges with $\sigma(u, v) \in [(1 - \rho)\epsilon, (1 + \rho)\epsilon]$ fall into the does-not-matter case. When ρ is small (e.g., $\rho = 0.01$), $\mathcal{L}_{\epsilon, \rho}(G)$ and $\mathcal{L}_{\epsilon}(G)$ are usually close, and their *StrCluResults* would not differ much. This intuition is confirmed on 15 real datasets in our experiments (see Section 9) and formalized by the theorem below.

THEOREM 2.3 (SANDWICH GUARANTEE). Given ϵ, μ and ρ , let $\mathcal{L}_{(1+\rho)\epsilon}(G)$ and $\mathcal{L}_{(1-\rho)\epsilon}(G)$ be the valid edge labellings with respect to similarity thresholds $(1 + \rho)\epsilon$ and $(1 - \rho)\epsilon$, respectively. For an arbitrary valid ρ -approximate edge labelling, $\mathcal{L}_{\epsilon, \rho}(G)$, we have:

- for every cluster $C_+ \in C(\mathcal{L}_{(1+\rho)\epsilon}(G), \mu)$, there exists a cluster $\tilde{C} \in C(\mathcal{L}_{\epsilon, \rho}(G), \mu)$ such that $C_+ \subseteq \tilde{C}$;
- for every cluster $\tilde{C} \in C(\mathcal{L}_{\epsilon, \rho}(G), \mu)$, there exists a cluster $C_- \in C(\mathcal{L}_{(1-\rho)\epsilon}(G), \mu)$ such that $\tilde{C} \subseteq C_-$.

PROOF. Let E_- , E_+ and E_ρ be the sets of *similar* edges labelled in $\mathcal{L}_{(1-\rho)\epsilon}(G)$, $\mathcal{L}_{(1+\rho)\epsilon}(G)$ and $\mathcal{L}_{\epsilon, \rho}(G)$, respectively. Consider an edge $(u, v) \in E_+$, it must satisfy $\sigma(u, v) \geq (1 + \rho)\epsilon$. By the definition of ρ -approximate edge labelling, (u, v) must be labelled as similar under $\mathcal{L}_{\epsilon, \rho}(G)$, i.e., $(u, v) \in E_\rho$. Thus it can be verified that $E_+ \subseteq E_\rho$. On the other hand, for a edge $(u, v) \in E_\rho$, by the definition, it must satisfy $\sigma(u, v) \geq (1 - \rho)\epsilon$. Therefore, it must be labelled as similar under $\mathcal{L}_{(1-\rho)\epsilon}(G)$, i.e., $(u, v) \in E_-$. In summarisation, E_- , E_+ and E_ρ satisfy the relationship $E_+ \subseteq E_\rho \subseteq E_-$.

Consider a cluster $C_+ \in C(\mathcal{L}_{(1+\rho)\epsilon}(G), \mu)$, the similar edges inside C_+ all belong to E_+ and thus all belong to E_ρ . Therefore, for any vertex $v \in C_+$, either core or non-core vertex, will belong to the same cluster $\tilde{C} \in C(\mathcal{L}_{\epsilon, \rho}(G), \mu)$. As a result, $C_+ \subseteq \tilde{C}$. The first bullet is proven. The second bullet can be proven symmetrically. \square

2.4 Distributed Tracking

Our final preliminary is the *Distributed Tracking* (DT) problem [8, 19, 21] and its solutions. The DT problem is defined in a *distributed* environment: there are h participants, s_1, s_2, \dots, s_h , and a *coordinator*, q . Each participant s_i has a two-way communication channel with the coordinator q , while direct communications between participants are prohibited. Furthermore, each s_i has an integer *counter* c_i , which is initially 0. At each time stamp, *at most one* (possibly *none*) of these h counters is incremented, by 1. Given an integer threshold $\tau > 0$, the job of the coordinator q is to report (immediately) the “*maturity*” of the condition $\sum_{i=1}^h c_i = \tau$. The goal is to minimize the *communication cost*, measured by the total number of *messages* (each of $O(1)$ words) sent and received by q .

A straightforward solution is for each participant to inform q whenever its counter is incremented. The total communication cost of this approach is clearly τ messages, which can be expensive if τ is large. The DT problem actually admits an algorithm [19] with $O(h \log(\tau/h))$ messages. The algorithm performs in *rounds* and in each round, it works as follows:

- If $\tau \leq 4h$, use the straightforward algorithm with $O(h)$ messages.
- If $\tau > 4h$, q sends to each s_i a *slack* $\lambda = \lfloor \tau/(2h) \rfloor$.
 - Define \hat{c}_i as the *checkpoint* value, indicating when s_i next needs to check in with q . Initially, $\hat{c}_i = \lambda$.
 - As soon as $c_i = \hat{c}_i$, s_i sends a signal to q , and then \hat{c}_i is increased by λ , indicating the next check-in time of s_i .
 - When q receives the h^{th} signal in this round, q obtains the precise value of c_i from each s_i and computes $\tau' = \tau - \sum_{i=1}^h c_i$. If $\tau' = 0$, q reports maturity. Otherwise, q starts a new round with $\tau \leftarrow \tau'$, from scratch with the new threshold τ .

Analysis. In each round, q sends h slacks, receives h signals and collects h counters from the participants. The communication cost in each round is bounded by $O(h)$ messages. Furthermore, it can be verified that at the end of each round, $\tau' \leq 3/4 \cdot \tau$. Referring to the original τ , there are at most $O(\log(\tau/h))$ rounds. The overall communication cost is bounded by $O(h \log(\tau/h))$ messages.

3 PROBLEM FORMULATION & RATIONALE

In this paper, we consider the *StrClu* problem in a dynamic scenario, where the graph $G = \langle V, E \rangle$ is subject to updates. Each update is either an insertion of a *new* edge or a deletion of an *existing* edge.

Definition 3.1 (Basic Dynamic StrClu Problem). For a specified valid edge labelling definition (e.g., Definitions 2.1 and 2.2), given $\epsilon \in (0, 1]$ and $\mu \geq 1$, the goal of the *Basic Dynamic StrClu Problem* is to maintain a valid edge labelling $\mathcal{L}(G)$.

By Fact 1, with a valid edge labelling $\mathcal{L}(G)$ being maintained at hand, the *StrCluResult* $C(\mathcal{L}(G), \mu)$ is uniquely defined and can be returned in $O(n + m)$ time, where n and m are the *current* numbers of vertices and edges in G , respectively.

As mentioned earlier, we significantly reduce the state-of-the-art $O(n)$ update cost to $O(\log^2 n + \log n \cdot \log \frac{M}{\delta})$ amortized for every sequence of M updates, and with probability at least $1 - \delta^*$, the clustering result is correct at all times under the ρ -approximate notion.

The Rationale in Our Solution. Observe that the does-not-matter case in the ρ -approximate notion essentially provides a *leeway* for maintaining edge labels. Each edge can: (i) be labelled with approximate similarity, and (ii) “afford” a certain number of updates without needing to flip the label. Our solution is designed based on these two crucial properties. First,

we propose a sampling-based method (in Section 4) to estimate the Jaccard similarity, by which the cost of labelling an edge is reduced to poly-logarithmic. Second, we show that each edge can afford k *affected* updates (formally defined in Section 5) without needing to check its label. As such, we deploy a DT instance to track the moment of the $(k+1)^{\text{th}}$ affected update for each edge, at which moment, the edge needs to be relabelled. However, these two ideas alone are not sufficient to beat the $O(n)$ update bound. To complete the design of our solution, we further need to organize the DT instances carefully with heaps. Finally, by performing a non-trivial amortized analysis (in Section 6), our solution to the basic problem is thus obtained.

Embarking from this solution, we further study a more challenging problem to support *cluster-group-by queries*:

Definition 3.2 (Cluster-Group-By Query). Consider an edge labelling $\mathcal{L}(G)$; for an arbitrary subset $Q \subseteq V$, on a *cluster-group-by* query of Q on G , we return $C_i \cap Q$ as a distinct group (with a unique identifier), for every cluster $C_i \in C(\mathcal{L}(G), \mu)$ satisfying $C_i \cap Q \neq \emptyset$.

Finally, the ultimate problem is defined as follows:

Definition 3.3 (Ultimate Dynamic StrClu Problem). In addition to maintaining a valid edge labelling $\mathcal{L}(G)$, the goal of the Ultimate Dynamic StrClu Problem is to further maintain certain data structures, by which every cluster-group-by query $Q \subseteq V$, with respect to $\mathcal{L}(G)$, can be answered in $O(|Q| \cdot \log n)$ time.

4 ESTIMATING JACCARD SIMILARITY

In this section, we propose a *sampling-based* method for estimating the similarity coefficient.

The Sampling-Estimator. Consider an arbitrary edge $(u, v) \in E$; let $a = |N[u] \cap N[v]|$ and $b = |N[u] \cup N[v]|$. Our sampling technique relies on a *biased* estimator. First, we define a random variable X , generated by the following steps.

- Flip a coin z , where $z = 1$ with probability $\frac{|N[u]|}{a+b}$ and $z = 0$ with probability $\frac{|N[v]|}{a+b}$.
- If $z = 1$, then uniformly at-random pick a vertex from $N[u]$; otherwise, uniformly at-random pick a vertex from $N[v]$. Denote the vertex picked by w .
- If $w \in N[u] \cap N[v]$, then $X = 1$; otherwise, $X = 0$.

According to the above generation procedure,

$$\begin{aligned} \Pr[X = 1] &= \Pr[X = 1 \wedge z = 1] + \Pr[X = 1 \wedge z = 0] \\ &= \frac{|N[u]|}{a+b} \cdot \frac{a}{|N[u]|} + \frac{|N[v]|}{a+b} \cdot \frac{a}{|N[v]|} = \frac{2a}{a+b}. \end{aligned}$$

Let X_1, X_2, \dots, X_L be L independent instances of X , and define $\bar{X} = \frac{1}{L} \sum_{i=1}^L X_i$. We have

$$E[\bar{X}] = \frac{2a}{a+b} = \frac{2a/b}{1+a/b} = \frac{2\sigma(u, v)}{1+\sigma(u, v)} \Leftrightarrow \sigma(u, v) = \frac{E[\bar{X}]}{2 - E[\bar{X}]} \quad (1)$$

THEOREM 4.1. Define $\tilde{\sigma}(u, v) = \frac{\bar{X}}{2 - \bar{X}}$. By setting $L = \frac{2}{\Delta^2} \ln \frac{2}{\delta}$, we have $\Pr[|\tilde{\sigma}(u, v) - \sigma(u, v)| > \Delta] \leq \delta$.

PROOF. Observe that

$$\begin{aligned} \Pr[|\tilde{\sigma}(u, v) - \sigma(u, v)| > \Delta] &= \Pr\left[\left| \frac{\bar{X}}{2 - \bar{X}} - \frac{E[\bar{X}]}{2 - E[\bar{X}]} \right| > \Delta \right] \\ &= \Pr\left[\frac{2 \cdot |\bar{X} - E[\bar{X}]|}{(2 - \bar{X})(2 - E[\bar{X}])} > \Delta \right] \leq \Pr[|\bar{X} - E[\bar{X}]| > \frac{\Delta}{2}], \end{aligned}$$

where the inequality is from \bar{X} and $E[\bar{X}]$ being in $[0, 1]$, and hence, $(2 - \bar{X})(2 - E[\bar{X}]) \geq 1$. By the Hoeffding Bound [17], the probability such that $|\bar{X} - E[\bar{X}]| \geq t$ is bounded by $2e^{-2Lt^2}$. Thus, when $L = \frac{1}{2 \cdot (\Delta/2)^2} \ln \frac{2}{\delta}$, we have $\Pr[|\bar{X} - E[\bar{X}]| > \frac{\Delta}{2}] \leq \delta$. \square

We call the estimator, $\tilde{\sigma}(u, v) = \frac{\bar{X}}{2 - \bar{X}}$, a (Δ, δ) -similarity-estimator, with which we label edges by the following strategy.

Definition 4.2 (The (Δ, δ) -Strategy). Every edge $(u, v) \in E$, is labelled as similar if and only if $\tilde{\sigma}(u, v) \geq \epsilon$.

LEMMA 4.3. *With $\Delta \leq \rho\epsilon$, with probability at least $1 - \delta$, the (Δ, δ) -strategy labelling is ρ -approximate valid.*

PROOF. The correctness of the labels in $\mathcal{L}_{\epsilon, \rho}(G)$ follows from the fact that for any edge $(u, v) \in E$, with probability at least $1 - \delta$, $|\sigma(u, v) - \tilde{\sigma}(u, v)| \leq \Delta \leq \rho\epsilon$. Thus, with the same probability,

- if $\sigma(u, v) \geq (1 + \rho)\epsilon$, we have $\tilde{\sigma}(u, v) \geq \epsilon$;
- if $\sigma(u, v) < (1 - \rho)\epsilon$, we have $\tilde{\sigma}(u, v) < \epsilon$.

In either of these cases, (u, v) must be labelled correctly under the ρ -approximate notion. For all other cases, the label of (u, v) does not matter. \square

Remark. An important superiority of our sampling-estimator over Min-Hash [4] is that it allows us to compute $\tilde{\sigma}(u, v)$ in $O(L)$ time in an *ad hoc manner*. That is, need not maintain any data structures (e.g., min-hash signatures), and thus, a $O(n + m)$ overall space consumption suffices. In the dynamic scenario, this feature saves substantially on maintenance costs.

5 MAINTAINING THE EDGE LABELLING

Next, we reveal the details of the main tools behind the *DynELM* algorithm. This algorithm maintains a valid ρ -approximate edge labelling, $\mathcal{L}_{\epsilon, \rho}(G)$, and is detailed in Section 6. In particular, we adopt the (Δ, δ) -strategy, with $\Delta = \frac{1}{2}\rho\epsilon$ and with δ to be set later, aka, the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, to determine edge labels.

5.1 Update Affordability

Observe that when an update (u, w) occurs, the *affected* similarity values are those between u and its neighbors, and those between w and its neighbors. These edges are called the *affected edges* of (u, w) , while (u, w) is an *affecting update* for each of these affected edges.

OBSERVATION 1. Consider an update (either an insertion or a deletion) of (u, w) ; if (u, v) is an arbitrary affected edge incident on u , with $a = |N[u] \cap N[v]|$ and $b = |N[u] \cup N[v]|$ immediately before the update, the effects of the update are:

- Case 1: an insertion of (u, w) ,
 - if $w \in N[v]$, $\sigma(u, v)$ increases to $(a + 1)/b$;
 - if $w \notin N[v]$, $\sigma(u, v)$ decreases to $a/(b + 1)$.
- Case 2: a deletion of (u, w) ,
 - if $w \in N[v]$, $\sigma(u, v)$ decreases to $(a - 1)/b$;
 - if $w \notin N[v]$, $\sigma(u, v)$ increases to $a/(b - 1)$.

Symmetric changes occur with each edge (v, w) incident on w .

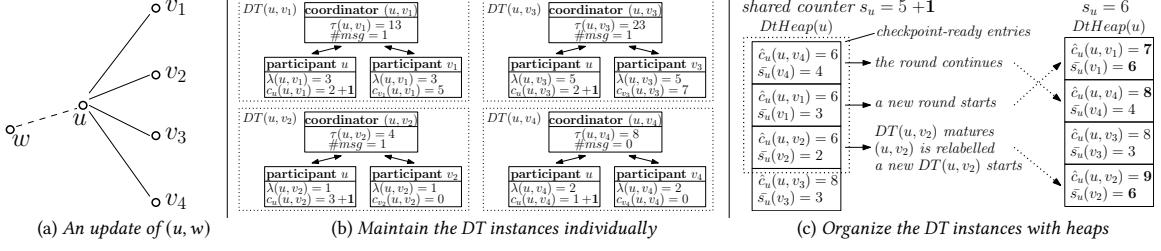


Fig. 2. An example of handling an update of (u, w) with DT instances on the u side (the update process on the w side is symmetric and thus omitted), where the un-affected edges incident on v_i (for $i = 1, \dots, 4$) are omitted.

Our crucial observation is that the *does-not-matter case* in the ρ -approximate notion, affords each edge a certain number of updates that do not instigate a label change.

LEMMA 5.1. *If an edge (u, v) is labelled as dissimilar by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, then, with probability $\geq 1 - \delta$, (u, v) can afford at least $k = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\max}(u, v) \rfloor$ affecting updates before its label flips (from dissimilar to similar), where $d_{\max}(u, v) = \max\{d[u], d[v]\}$.*

PROOF. Let $a = |N[u] \cap N[v]|$ and $b = |N[u] \cup N[v]|$, initially. Edge (u, v) being labelled dissimilar by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy implies $\tilde{\sigma}(u, v) < \epsilon$, and thus, with probability $\geq 1 - \delta$, $\sigma(u, v) = a/b \leq (1 + \frac{1}{2}\rho)\epsilon < (1 + \rho)\epsilon \leq 1 \Rightarrow a < b$. Since both a and b are integers, we have $a + 1 \leq b$, and hence $(a + 1)/b \geq a/(b - 1)$. Therefore, in considering the minimum number of affecting updates to cause (u, v) 's label flip (from dissimilar to similar), we focus on the edge insertions that increase $\sigma(u, v)$. After $k = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\max}(u, v) \rfloor \leq \frac{1}{2}\rho\epsilon b$ such updates, by the first bullet of Case 1 in Observation 1,

$$\sigma(u, v) = \frac{a}{b} + \frac{k}{b} \leq (1 + \frac{1}{2}\rho)\epsilon + \frac{\frac{1}{2}\rho\epsilon b}{b} = (1 + \rho)\epsilon.$$

Therefore, after k arbitrary affecting updates, the dissimilar label of (u, v) remains valid with probability $\geq 1 - \delta$. \square

LEMMA 5.2. *If edge (u, v) is labelled as similar by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, then, with probability $\geq 1 - \delta$, (u, v) can afford at least $k = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\max}(u, v) \rfloor$ affecting updates before its label flips.*

PROOF. The proof is analogous to that of Lemma 5.1. \square

5.2 Distributed Tracking on Updates

Lemmas 5.1 and 5.2 together show that an edge (u, v) labelled by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy can afford at least $k = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\max}(u, v) \rfloor$ affecting updates without its label flipping. It thus suffices to check its label upon the $(k + 1)^{\text{th}}$ affecting update since it was last labelled.

Creating DT Instances for Edges. To achieve this purpose, we adopt distributed tracking (DT) to *track* the number of affecting updates for each edge. Specifically, for each edge (u, v) , we *simulate* a DT instance, denoted by $DT(u, v)$, in a single thread in main memory. The edge (u, v) itself is the coordinator, with its endpoints u and v the participants; and the tracking threshold is set to

$$\tau(u, v) = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\max}(u, v) \rfloor + 1. \quad (2)$$

The counter $c_u(u, v)$ (resp., $c_v(u, v)$) is the current number of affecting updates of (u, v) incident on u (resp., v). As soon as $c_u(u, v) + c_v(u, v) = \tau(u, v)$, the coordinator at (u, v) reports *maturity*. Since the label (u, v) could be invalid, we

relabel (u, v) with the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy. After then, a new $DT(u, v)$ instance, with a new threshold $\tau(u, v)$ (based on the new $d_{\max}(u, v)$) is instantiated.

However, as we are simulating DT in main memory and counting running time only, in addition to the simulated $O(\log \tau(u, v))$ communication cost, each counter increment costs $O(1)$ time. Incrementing $c_u(u, v)$ for all neighbors $v \in N[u]$ still leads to a $\Omega(d[u]) = \Omega(n)$ cost for an update. Figures 2(a) and 2(b) show an example; for an update of (u, w) , one needs to *individually* increase the counter of the participant u in each of the DT instances.

Organizing DT Instances by Heaps. The key to address this issue is to maintain a *shared* common counter, instead of increasing $c_u(u, v)$ for each $v \in N[u]$ individually. Let every vertex u have, instead of $c_u(u, v)$, a single counter s_u (shared among all $DT(u, v)$), initially set to 0, recording the number of affecting updates of edges incident on u . The crucial observation here is that the checkpoint value $\hat{c}_u(u, v)$ is only updated when there would have been $\lambda(u, v)$ (the slack value in $DT(u, v)$) affecting updates incident on u . Thus, the number of increments (i.e., $\lambda(u, v)$) is important, rather than the value of $\hat{c}_u(u, v)$. Therefore, we can *shift* the checkpoint value $\hat{c}_u(u, v)$ by the value s_u at the time the checkpoint is set. After shifting, for each vertex u , we set up a *min-heap*, denoted by $DtHeap(u)$, with the shifted checkpoints, $\hat{c}_u(u, v)$, as keys. For each (u, v) , we maintain an entry in $DtHeap(u)$ associated with:

- $\bar{s}_u(v)$: the value of s_u when the current round in $DT(u, v)$ starts. With $\bar{s}_u(v)$, the *unshifted* counter value in this participant can be computed, by $s_u - \bar{s}_u(v)$, when the coordinator needs it;
- $\hat{c}_u(u, v)$: the key of the entry, initialized as $\hat{c}_u(u, v) = \bar{s}_u(v) + \lambda(u, v)$, where $\lambda(u, v)$ is the slack value in the current round of $DT(u, v)$.

When an affecting update arrives, u only needs to inform its coordinators if there is some entry with a key equal to s_u . Each of these entries is called a *checkpoint-ready* entry. Therefore, in this way, we no longer have to scan the whole neighbourhood of a vertex.

A Running Example. Figure 2(c) shows an example, where the shared counter $s_u = 5$ before the update, and the entries corresponding to the DT instances of (u, v_4) , (u, v_1) and (u, v_2) are at the *top* of $DtHeap(u)$ as they have the same smallest key values, i.e., $\hat{c}_u = 6$. When the update of (u, w) is performed, s_u is increased to $s_u = 6$, and hence, the three entries at the heap top become checkpoint-ready. For the entry of $DT(u, v_4)$, the participant u only needs to notify the coordinator (u, v_4) and the current round continues (as shown in Figure 2(b), this is the first notification in the round): $\bar{s}_u(v_4)$ does not change and $\hat{c}_u(u, v_4)$ is increased by $\lambda(u, v_4) = 2$ indicating that when s_u reaches 8, this entry will become checkpoint-ready again. For the entry of $DT(u, v_1)$, after the notification sent for this entry, the current round ends: $\tau(u, v_1) \leftarrow (13 - 3) - 5 = 5$ and $\lambda(u, v_1) \leftarrow \lfloor \frac{5}{2 \times 2} \rfloor = 1$. Thus, $\bar{s}_u(v_1) \leftarrow s_u = 6$ and $\hat{c}_u(u, v_1) = 6 + 1 = 7$.

Finally, for the entry of $DT(u, v_2)$, the DT instance is matured. Hence, the edge (u, v_2) is relabelled by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy and a new DT instance with respect to the new $\tau(u, v_2)$ is started.

6 THE DynELM ALGORITHM

An outline of the *DynELM* algorithm for handling an update, of edge (u, w) , is as follows:

- **Step 1.** Initialize the set of label-flipping edges $\mathcal{F} \leftarrow \emptyset$; and increment s_u and s_w (by 1), respectively.
- **Step 2.** There are two cases:
 - Case 1: this update is an insertion. Insert (u, w) into G and label it by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy. If (u, w) is labelled as similar, add (u, w) to \mathcal{F} . Moreover, create $DT(u, w)$ with $\tau(u, w)$.
 - Case 2: this update is a deletion. If (u, w) is labelled as similar, add (u, w) to \mathcal{F} . Delete (u, w) from G ; and delete $DT(u, w)$.

- **Step 3.** While there is a checkpoint-ready entry in $DtHeap(u)$, pop the entry (from the top). Let $DT(u, v)$ be the DT instance corresponding to this entry. Instruct u to inform the coordinator (u, v) . When $DT(u, v)$ is mature, relabel (u, v) by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy. If its label flipped, add (u, v) to \mathcal{F} . Remove its entry from $DtHeap(v)$, and restart the DT with a new $\tau(u, v)$. Repeat until there is no checkpoint-ready entry in $DtHeap(u)$.
- **Step 4.** Perform a symmetric process of Step 3 for w .
- **Step 5.** Return \mathcal{F} , the set of edges whose labels flipped.

6.1 Theoretical Analysis

In this subsection, we prove the following theorem:

THEOREM 6.1. *Given a specified failure probability δ^* , for every sequence of M updates (the value of M value need not be known in advance), there exists an implementation of the DynELM algorithm that achieves the following guarantees:*

- *the amortized cost of each update is $O(\log^2 n + \log n \cdot \log \frac{M}{\delta^*})$;*
- *the space consumption is always linear in the size of G , i.e., $O(n + m)$;*
- *with probability at least $1 - \delta^*$, the ρ -approximate edge labelling $\mathcal{L}_{\epsilon, \rho}(G)$ maintained is always valid. Hence, with the same probability, the clustering result, $C(\mathcal{L}_{\epsilon, \rho}(G), \mu)$, is always correct (under the ρ -approximate notion).*

COROLLARY 6.2. *If the number of vertices, n , is fixed over the whole update sequence and the total number of updates M is bounded by $O(n^c)$ for some constant c , e.g., $M = n^{99}$, by setting $\delta^* = 1/n$, the amortized update bound can be simplified to $O(\log^2 n)$.*

We consider the following implementation of *DynELM*.

- The neighborhood $N[u]$ of each vertex u is maintained by a binary search tree: each neighbor insertion, deletion and search can be performed in $O(\log d[u]) = O(\log n)$ time.
- The $DtHeap(u)$ of each vertex u is implemented with a binary heap: each heap operation takes $O(\log d[u]) = O(\log n)$ time. Moreover, according to the DT algorithm, there are at most $O(\log \tau(u, v)) = O(\log n)$ rounds (before its maturity) for each $DT(u, v)$, and each round takes $O(h) = O(1)$ (because only $h = 2$ participants in the instance) operations in the DT heaps of u and v . The overall cost of each $DT(u, v)$ is bounded by $O(\log^2 n)$.
- For the i^{th} invocation of the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, the parameter δ is set to

$$\delta_i = \frac{\delta^*}{i \cdot (i+1)}, \text{ where } i = 1, 2, \dots \quad (3)$$

Let κ be the total number of invocations of the strategy. According to Theorem 4.1, the required sample size for the i^{th} invocation is

$$L_i = \frac{2}{(\frac{1}{2}\rho\epsilon)^2} \ln \frac{2}{\delta_i} = O(\log \frac{\kappa}{\delta^*}). \quad (4)$$

Therefore, the cost of each invocation of $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy is bounded by $O(L_i \cdot \log n) = O(\log n \cdot \log \frac{\kappa}{\delta^*})$.

Next, we prove the three bullets in Theorem 6.1 one by one.

Bullet 1 in Theorem 6.1: Amortized Update Cost. We analyse the amortized update cost of *DynELM* step by step. Clearly, the running time of Step 1 in the *DynELM* algorithm is $O(1)$, and Step 2 can be performed in $O(\log n \cdot \log \frac{\kappa}{\delta^*})$ time in the worst case. It remains to bound the amortized cost of Step 3 and Step 4.

Consider an update of (u, w) . A crucial observation is that the update of (u, w) can *only* contribute (via a counter increment) to the maturity of the DT instances of its affected edges, which exist at the *current moment*. Let $DT(u, v^*)$ be

the instance with the *smallest* threshold value $\tau(u, v^*)$ among all the affected DT instances at the current moment, and $d'[u]$ the degree of u when $DT(u, v^*)$ was created. We claim that the degree of u , $d[u]$, at the current moment is at most $d'[u] + \tau(u, v^*)$. This is because since the creation of $DT(u, v^*)$, there can be at most $\tau(u, v^*)$ insertions adjacent on u ; otherwise, $DT(u, v^*)$ must have matured and hence, would not exist at the current moment. Therefore,

$$d[u] \leq d'[u] + \tau(u, v^*) \leq \frac{\tau(u, v^*)}{\rho\epsilon/2} + \tau(u, v^*) = O(1) \cdot \tau(u, v^*).$$

Furthermore, as each of the affected $DT(u, v)$ requires at least $\tau(u, v)$ affecting updates to mature, the current update of (u, w) is actually accounted for only $\frac{1}{\tau(u, v)}$ of the cost of the DT maturity as well as the following edge re-labelling cost, i.e., $\frac{1}{\tau(u, v)} \cdot O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})$. Summing up over all the neighbors of u , the amortized cost instigated by an update of (u, w) in Step 3 is:

$$\begin{aligned} & \sum_{v \in N[u]} \frac{O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})}{\tau(u, v)} \leq \sum_{v \in N[u]} \frac{O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})}{\tau(u, v^*)} \\ &= \frac{d[u]}{\tau(u, v^*)} \cdot O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*}) = O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*}). \end{aligned}$$

By symmetry, the update of (u, w) is also charged a $O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})$ cost from the affected DT instances of the vertex w . Therefore, combining the costs of all the four steps, the amortized cost of each update is bounded by $O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})$.

To complete our analysis, we claim that each update of (u, w) can instigate at most $O(1)$ *amortized invocations* of the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, where one is for labelling (u, w) when the update is an insertion, the others are for the $O(1)$ DT maturity charged to the current update. Thus, the total number of the invocations of the strategy, κ , is at most $O(1)$ times of the number of updates, i.e., $\kappa = O(M)$, and the amortized update cost follows.

LEMMA 6.3. *For any sequence of M updates, the amortized update cost of DynELM is bounded by $O(\log^2 n + \log n \cdot \log \frac{M}{\delta^*})$.*

Bullet 2 in Theorem 6.1: Overall Space Consumption. Based on the aforementioned implementation of *DynELM*, for each vertex u , a binary search tree and a DT heap on the neighborhood $N[u]$ are maintained, the space consumption of each vertex is bounded by $O(1 + d[u])$. Summing up over all the vertices, the overall space consumption of the *DynELM* algorithm is bounded by $O(n + m)$.

LEMMA 6.4. *At all times, DynELM consumes $O(n + m)$ space.*

Bullet 3 in Theorem 6.1: Correctness and Failure Probability. The correctness of the approximate edge labelling $\mathcal{L}_{\epsilon, \rho}(G)$ maintained by *DynELM* follows immediately from the correctness of the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy (Lemma 4.3) and the update affordability (Lemmas 5.1 and 5.2). It remains to bound the failure probability. According to our implementation and by Union Bound, the failure probability is bounded by:

$$\sum_{i=1}^{\kappa} \delta_i = \sum_{i=1}^{\kappa} \frac{\delta^*}{i \cdot (i+1)} = \delta^* \cdot \sum_{i=1}^{\kappa} \left(\frac{1}{i} - \frac{1}{i+1} \right) = \delta^* \cdot \left(1 - \frac{1}{\kappa+1} \right) \leq \delta^*.$$

LEMMA 6.5. *With probability at least $1 - \delta^*$, the ρ -approximate edge labelling maintained by DynELM is always valid.*

7 THE ULTIMATE ALGORITHM

We round out our algorithm development by designing the *DynStrClu* algorithm for solving the *ultimate* dynamic *StrClu* problem. Specifically, we prove the following theorem.

THEOREM 7.1. *The DynStrClu algorithm both*

- *admits all the same guarantees as DynELM in Theorem 6.1; and*
- *answers every cluster-group-by query in time linear-polylog in the query size, i.e., for $Q \subseteq V$, in $O(|Q| \cdot \log n)$ time.*

The Algorithm Framework. The *DynStrClu* algorithm mainly consists of the following three modules:

- Edge Label Manager (ELM): This module invokes the *DynELM* algorithm as a black box, to maintain $\mathcal{L}_{\epsilon, \rho}(G)$.
- Vertex Auxiliary Information (vAuxInfo): For each vertex $u \in V$, we maintain auxiliary information:
 - a counter, $SimCnt(u)$, for recording the current number of similar neighbors of u ;
 - a partition of u 's neighbors, which partitions u 's neighbors into three self-explanatory categories: (i) sim-core neighbors, (ii) sim-non-core neighbors, and (iii) dissimilar neighbors.
- CC Structure of G_{core} : In this module, we maintain the connected components in G_{core} . In particular, we maintain a data structure, denoted by $CC-Str(G_{core})$, to support the following operations:
 - *Insert* a sim-core edge (u, v) into G_{core} .
 - *Remove* from G_{core} an edge (u, v) .
 - *FindCclID*(u): Return the identifier of the connected component in G_{core} that contains the (core) vertex u .

FACT 2 ([18, 35]). *There exists a $O(n + m)$ -space data structure that implements $CC-Str(G_{core})$ and can support: (i) each edge insertion or deletion in $O(\log^2 n)$ amortized time, and (ii) each *FindCclID* operation in $O(\log n)$ worst-case time.*

The *DynStrClu* Algorithm Steps. To process an update, the *DynStrClu* algorithm maintains $\mathcal{L}_{\epsilon, \rho}(G)$, with the *DynELM* algorithm; this returns the set \mathcal{F} of edges whose labels have flipped due to the update. Given this flipped set \mathcal{F} , *DynStrClu* maintains the two other modules as follows.

Maintaining vAuxInfo. For each edge $(u, v) \in \mathcal{F}$,

- update $SimCnt(u)$ and $SimCnt(v)$ in constant time: if the label of (u, v) is flipped to similar, both $SimCnt(u)$ and $SimCnt(v)$ are increased by 1; otherwise, they are decreased by 1, respectively.
- if necessary, flip u 's (resp., v 's) *core status*, and hence change the neighbor category of u for its *similar* neighbors.

Let V' comprise every vertex whose core status has flipped, while \mathcal{F}' is the set of all the edges whose sim-core status have flipped between *sim-core* and *non-sim-core*.

Maintaining G_{core} . For each edge $(u, v) \in \mathcal{F}'$, if the status of (u, v) flipped from *non-sim-core* to *sim-core*, insert (u, v) into G_{core} . Otherwise, remove (u, v) from G_{core} . Furthermore, for each vertex $u \in V'$, if u flipped from *non-core* to *core*, insert u to G_{core} by *conceptually* inserting to G_{core} a self-loop edge (u, u) , which does not necessarily physically exist. Otherwise, remove u from G_{core} by *conceptually* removing the self-loop edge (u, u) , in which case, u must be a singleton vertex in G_{core} with no incident edge other than the self-loop edge. This is because, all its incident edges in G_{core} have been removed when processing the edges in \mathcal{F}' . All these operations on G_{core} can be performed via $CC-Str(G_{core})$.

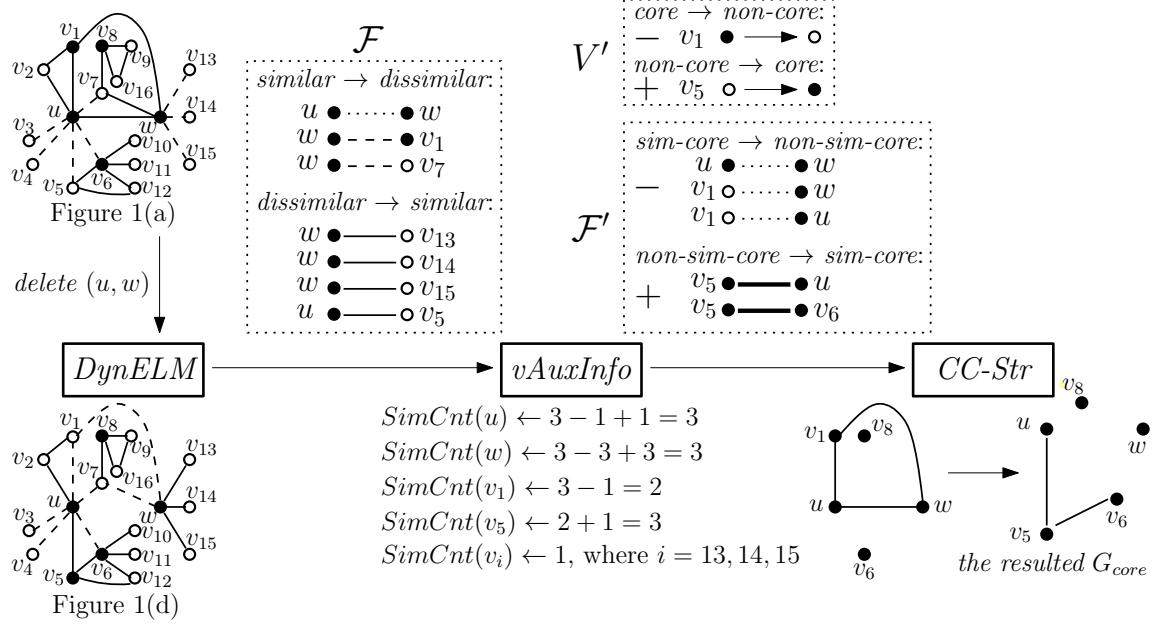


Fig. 3. The process of *DynStrClu* handling a deletion of (u, w) from Figure 1(a), with $\epsilon = 1/3$, $\mu = 3$ and $\rho = 0.01$. The resulted state of after this update is shown in Figure 1(d).

A Running Example. Figure 3 shows the maintenance process of *DynStrClu* for deleting the edge (u, w) from Figure 1(a), where the resulted state is as shown in Figure 1(d). To process the deletion of (u, w) , *DynStrClu* invokes *DynELM* to maintain the edge labelling $\mathcal{L}_{\epsilon, \rho}(G)$. The returned set \mathcal{F} of edges with labels flipped is shown in the figure. In particular, since (u, w) is a similar edge getting deleted, its label is treated as flipping from similar to dissimilar. Next, the *vAuxInfo* module updates the *SimCnt* information for the endpoints of the edges in \mathcal{F} ; details are shown in the figure. Since $SimCnt(v_1)$ is decreased from 3 to 2 and $SimCnt(v_5)$ is increased from 2 to 3, the core status of v_1 is flipped from core to non-core, while v_5 's is from non-core to core. Thus, $V' = \{v_1, v_5\}$, the set of all the vertices with core status flipped. Furthermore, with \mathcal{F} and V' , the set \mathcal{F}' of the edges whose sim-core status are flipped can be easily obtained. For example, the sim-core status of edges (u, w) and (v_1, w) are flipped from sim-core to non-sim-core for different reasons. While the flip of (u, w) is because of its label flipping to dissimilar, the flip of (v_1, w) is caused by v_1 becoming non-core. Likewise, since v_5 becomes a core vertex, the similar edges (v_5, u) and (v_5, v_6) become sim-core. Finally, the CC-Str maintains G_{core} with V' and \mathcal{F}' by: (i) removing v_1 and adding v_5 ; (ii) removing all the edges in \mathcal{F}' turning into non-sim-core, while adding those edges becoming sim-core. The resulting G_{core} is as shown in Figure 3.

Theoretical Analysis. We analyse the overall maintenance cost on the above two modules.

LEMMA 7.2. $|V'| = O(|\mathcal{F}|)$ and $|\mathcal{F}'| = O(|\mathcal{F}|)$.

PROOF. Observe that the core status of vertex u is flipped only if $SimCnt(u)$ changes: at least one edge incident on u has its label flipped. As such edge must be in \mathcal{F} , $|V'| \leq 2 \cdot |\mathcal{F}| = O(|\mathcal{F}|)$ holds.

Next, we bound $|\mathcal{F}'|$. We define *persistently similar edges* as those edges that remain similar after the update. There are only two possibilities for edge (u, v) to be in \mathcal{F}' : (i) the label of (u, v) is flipped, or (ii) (u, v) is persistently similar

and has at least one endpoint with core status flipped. Clearly, there are at most $|\mathcal{F} \cap \mathcal{F}'| \leq |\mathcal{F}|$ edges added to \mathcal{F}' due to the first case. For the edges in $\mathcal{F}' \setminus \mathcal{F}$, they must belong to the second case. Thus, this is at most the number of persistently similar edges incident on some vertex in V' . For each $u \in V'$, there can be at most $\mu - 1 = O(1)$ persistently similar edges incident on u . Because otherwise, there is a contradiction with the fact that u 's core status has been flipped. For example, in Figure 3, $v_1 \in V'$ has one persistent similar edge (v_1, v_2) and $v_5 \in V'$ has two: (v_5, v_6) and (v_5, v_{12}) . Both of these numbers are at most 2. Otherwise, v_1 would not become a non-core and v_5 would have been a core before the update. Thus, there can be at most $O(|V'|) = O(|\mathcal{F}|)$ persistently similar edges incident on the vertices in V' . Therefore, $|\mathcal{F}' \setminus \mathcal{F}| = O(|\mathcal{F}|)$, and hence $|\mathcal{F}'| = O(|\mathcal{F}|)$. \square

LEMMA 7.3. *The cost of maintaining $vAuxInfo$ and G_{core} is bounded by $O(|\mathcal{F}| \cdot \log^2 n)$.*

PROOF. We bound the maintenance cost of $vAuxInfo$ first. Since only the endpoints of the edges in \mathcal{F} can have $SimCnt$ changed, the cost of maintaining $SimCnt$ is clearly bounded by $O(|\mathcal{F}|)$. As for the neighbor category, there are only two possible types of changes: (i) between similar and dissimilar neighbors, or (ii) between sim-core and sim-non-core neighbors. While the former is caused by edge-label flips, the latter is due to sim-core status change. The total number of neighbor category alternations caused by this update is thus at most $|\mathcal{F}| + |\mathcal{F}'|$. As each such alternation takes $O(1)$ time, the maintenance cost is $O(|\mathcal{F}| + |\mathcal{F}'|)$. Therefore, by Lemma 7.2, the per-update maintenance cost for $vAuxInfo$ is $O(|\mathcal{F}|)$.

As for the maintenance of G_{core} , it is clear that there are $|V'| + |\mathcal{F}'| = O(|\mathcal{F}|)$ operations with $CC-Str(G_{core})$. By Fact 2, each such operation incurs a $O(\log^2 n)$ amortized cost. The $O(|\mathcal{F}| \cdot \log^2 n)$ per-update maintenance cost of G_{core} thus follows. \square

THEOREM 7.4. *The DynStrClu Algorithm admits all the same guarantees as DynELM in Theorem 6.1*

PROOF. First, to analyse the amortized update cost, observe that an edge can be added to \mathcal{F} only when it is relabelled. Thus, we can amortize the $O(|\mathcal{F}| \cdot \log^2 n)$ cost over all these edges in \mathcal{F} . Hence, each of such edges is charged for an extra $O(\log^2 n)$ cost when it is relabelled; this charging increases the relabelling cost bound to $O(\log^2 n + \log n \cdot \log \frac{M}{\delta^*})$. Therefore, the amortized update cost bound remains the same as that of *DynELM*. Second, by Fact 2, the space consumption of *DynStrClu* is also bounded by $O(n + m)$. Finally, as the maintenance for $vAuxInfo$ and $CC-Str(G_{core})$ is deterministic, the correctness probability remains the same as *DynELM*. \square

The Cluster-Group-By Query Algorithm. Given $Q \subseteq V$, the cluster-group-by query algorithm is as follows:

- Initialize an empty (vertex, ccid)-pair set: $P \leftarrow \emptyset$.
- For each $u \in Q$:
 - If u is core, obtain the ID of the CC containing u in G_{core} , denoted by $ccid(u)$. Add the pair $(u, ccid(u))$ to P .
 - If u is non-core, for each sim-core neighbor v of u (possibly none exists), add a pair $(u, ccid(v))$ to P .
- Sort the pairs in P by the $ccid(\cdot)$ keys. Put all vertices with the same $ccid(\cdot)$ into the same group and output the resulted groups.

LEMMA 7.5. *The running time complexity of this Cluster-Group-By Query Algorithm is $O(|Q| \cdot \log n)$.*

PROOF. While each core vertex in Q produces exactly one pair, each non-core vertex in Q can produce at most $\mu - 1$ pairs. The size of P is thus $O(|Q|)$. Furthermore, by Fact 2, the $ccid(\cdot)$ of each pair can be obtained with $CC-Str(G_{core})$ in $O(\log n)$ time. Combining the sorting cost, the total running time is $O(|Q| \cdot \log n)$. \square

Proof of Theorem 7.1. The theorem follows immediately from Theorem 7.4 and Lemma 7.5.

Remark. The amortized update cost bound in Theorem 7.1 (and hence, in Theorem 6.1) is general enough for “hot-start” cases, where a graph G with m_0 edges is given at the beginning. To handle this case, one can first insert each of these m_0 edges one by one, with a total cost $\tilde{O}(m_0)$, and then charge this $\tilde{O}(m_0)$ cost to the next $\Omega(m_0)$ updates. There is just a constant factor blow-up in the amortized update cost.

8 EXTENSION TO COSINE SIMILARITY

In this section, we introduce our extension work that adopts cosine similarity as the definition of structural similarity.

The cosine similarity $\sigma_c(u, v)$ between two vertices u and v is defined as in [40]:

- if $(u, v) \in E$, then

$$\sigma_c(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}},$$

- if $(u, v) \notin E$, then

$$\sigma_c(u, v) = 0.$$

To see how it follows the definition of cosine similarity, we can construct a n -dimension vector \vec{p} and let $\vec{p}[i] = 1$ if the i -th vertex is in $N[u]$ and $\vec{p}[i] = 0$ otherwise. Another n -dimension vector \vec{q} can also be constructed in the same way based on $N[v]$. Then $|N[u] \cap N[v]| = \vec{p} \cdot \vec{q}$, $d[u] = \|\vec{p}\|$, and $d[v] = \|\vec{q}\|$. Therefore, $\sigma_c(u, v)$ equals the cosine similarity between these two vectors \vec{p} and \vec{q} .

For cosine similarity, we have obtained similar results as Jaccard similarity. Specifically, we extend *DynELM* and *DynStrClu* algorithms *under cosine similarity* and their performances are guaranteed in the following theorem.

THEOREM 8.1. *The DynELM algorithm admits all the same guarantees in Theorem 6.1 under cosine similarity, and the DynStrClu algorithm admits all the same guarantees in Theorem 7.1 under cosine similarity.*

We also adopt ρ -approximate notion for approximate edge labeling. To complete the proof of Theorem 8.1, we need to extend three main components in our algorithm, namely: *similarity estimator*, *update affordability*, and *distributed tracking on updates*. We proof similar results on these three components in the following subsections.

8.1 Estimating Cosine Similarity

In this subsection, we proof that the *sampling-based* method also works for estimating cosine similarity between two vertices. Let X_1, X_2, \dots, X_L be L independent instances of the random variable X defined in Section 4, and define $\bar{X} = \sum_{i=1}^L X_i$. Recall in Equation 1, we have:

$$\sigma(u, v) = \frac{|N[u] \cap N[v]|}{|N[u] \cup N[v]|} = \frac{E[\bar{X}]}{2 - E[\bar{X}]}.$$

Combining the fact that:

$$|N[u] \cap N[v]| = d[u] + d[v] - |N[u] \cup N[v]|.$$

We can compute that:

$$\begin{aligned} |N[u] \cap N[v]| &= d[u] + d[v] - |N[u] \cup N[v]| \\ &= d[u] + d[v] - \frac{|N[u] \cap N[v]|}{\sigma(u, v)} \\ &= d[u] + d[v] - \frac{(2 - E[\bar{X}])|N[u] \cap N[v]|}{E[\bar{X}]} \end{aligned}$$

As a result, we have:

$$|N[u] \cap N[v]| = \frac{1}{2}(d[u] + d[v])E[\bar{X}]. \quad (5)$$

Thus, $\frac{1}{2}(d[u] + d[v])\bar{X}$ can serve as an estimator for the size of intersection between two neighbourhoods. To compute the cosine similarity $\sigma_c(u, v)$, we have:

$$\sigma_c(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}} = \frac{(d[u] + d[v])E[\bar{X}]}{2\sqrt{d[u] \cdot d[v]}}. \quad (6)$$

Let $d_{\min}(u, v) = \min\{d[u], d[v]\}$ and $d_{\max}(u, v) = \max\{d[u], d[v]\}$. Then, we estimate the cosine similarity in the following manners:

- if $d_{\min}(u, v) < \epsilon^2 d_{\max}(u, v)$, edge (u, v) can directly be labelled as dissimilar;
- otherwise, define $\tilde{\sigma}_c(u, v) = \frac{(d[u] + d[v])\bar{X}}{2\sqrt{d[u] \cdot d[v]}}$ and use it as an estimator for $\sigma_c(u, v)$.

For the first case, we prove its correctness with the following lemma:

LEMMA 8.2. *For an edge (u, v) , if $d_{\min}(u, v) < \epsilon^2 d_{\max}(u, v)$, then $\sigma_c(u, v) < \epsilon$.*

PROOF. The cosine similarity between u and v can be upper bounded by:

$$\begin{aligned} \sigma_c(u, v) &= \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}} \\ &\leq \frac{d_{\min}(u, v)}{\sqrt{d_{\min}(u, v) \cdot d_{\max}(u, v)}} \\ &= \sqrt{\frac{d_{\min}(u, v)}{d_{\max}(u, v)}} < \epsilon. \end{aligned}$$

□

For the second case, we can guarantee the quality of our estimator by the theorem below.

THEOREM 8.3. *Suppose $\epsilon^2 d_{\max}(u, v) \leq d_{\min}(u, v) \leq d_{\max}(u, v)$, by setting $L = \frac{(\epsilon^2 + 1)^2}{8\epsilon^2 \Delta^2} \ln(\frac{2}{\delta})$, we have $\Pr[|\tilde{\sigma}_c(u, v) - \sigma_c(u, v)| > \Delta] < \delta$.*

PROOF. In this case, we can result in the following bound:

$$\begin{aligned} \frac{d[u] + d[v]}{\sqrt{d[u] \cdot d[v]}} &= \frac{d_{\min}(u, v) + d_{\max}(u, v)}{\sqrt{d_{\min}(u, v) \cdot d_{\max}(u, v)}} \\ &= \sqrt{\frac{d_{\min}(u, v)}{d_{\max}(u, v)}} + \sqrt{\frac{d_{\max}(u, v)}{d_{\min}(u, v)}} \\ &\leq \epsilon + \frac{1}{\epsilon}. \end{aligned}$$

The final inequality results from the fact that $1 \leq \sqrt{\frac{d_{\max}(u,v)}{d_{\min}(u,v)}} \leq \frac{1}{\epsilon}$.

Observe that

$$\begin{aligned} \Pr[|\tilde{\sigma}_c(u,v) - \sigma_c(u,v)| > \Delta] &= \Pr\left[\frac{d[u] + d[v]}{2\sqrt{d[u] \cdot d[v]}} |\bar{X} - E[\bar{X}]| > \Delta\right] \\ &\leq \Pr[|\bar{X} - E[\bar{X}]| > \frac{2\Delta}{\epsilon + \frac{1}{\epsilon}}]. \end{aligned}$$

By the Hoeffding Bound [17], when $L = \frac{(\epsilon + \frac{1}{\epsilon})^2}{8\Delta^2} \ln(\frac{2}{\delta})$, we have $\Pr[|\bar{X} - E[\bar{X}]| > \frac{2\Delta}{\epsilon + \frac{1}{\epsilon}}] \leq \delta$. \square

Based on the above estimation for cosine similarity, we can utilize the same (Δ, δ) -strategy to label edges. Specifically, in the extension, we adopt $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy to determine edge labels as in the original work.

8.2 Update Affordability

When an update (u, w) occurs, the cosine similarity of the affected edges will be affected in a similar way to Jaccard similarity as in Section 5.1.

OBSERVATION 2. Consider an update of edge (u, w) , and an arbitrary affected edge (u, v) , let $\sigma_c(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{d[u]d[v]}} = \frac{a}{\sqrt{d[u]d[v]}}$ be the structural similarity between u and v before the update of (u, w) , where $a = |N[u] \cap N[v]|$. The effect of such update (u, w) can be:

- Case 1: an insertion of (u, w) ,
 - if $w \in N[v]$, $\sigma_c(u, v)$ is increased to $(a+1)/\sqrt{(d[u]+1)d[v]}$;
 - if $w \notin N[v]$, $\sigma_c(u, v)$ is decreased to $a/\sqrt{(d[u]+1)d[v]}$.
- Case 2: an deletion of (u, w) ,
 - if $w \in N[v]$, $\sigma_c(u, v)$ is decreased to $(a-1)/\sqrt{(d[u]-1)d[v]}$;
 - if $w \notin N[v]$, $\sigma_c(u, v)$ is increased to $a/\sqrt{(d[u]-1)d[v]}$.

Symmetric changes occur with each edge (v, w) incident to w .

Like Jaccard similarity, we can also calculate update affordability for cosine similarity. However, here for each edge $(u, v) \in E$ we divide the calculation of update affordability into two cases;

- $d_{\min}(u, v) \geq 0.81\epsilon^2 \cdot d_{\max}(u, v)$, and
- $d_{\min}(u, v) < 0.81\epsilon^2 \cdot d_{\max}(u, v)$

The update affordability for each edge (u, v) is computed by the following lemmas.

LEMMA 8.4. If an edge (u, v) is labelled as dissimilar by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, then with probability at least $1 - \delta$, (u, v) can afford at least $k = \lfloor 0.45\rho\epsilon^2 \cdot d_{\max}(u, v) \rfloor$ affecting updates if $d_{\min}(u, v) \geq 0.81\epsilon^2 \cdot d_{\max}(u, v)$ or $k = \lfloor 0.19\epsilon^2 \cdot d_{\max}(u, v) \rfloor$ affecting updates if $d_{\min}(u, v) < 0.81\epsilon^2 \cdot d_{\max}(u, v)$ before its label flips from dissimilar to similar.

PROOF. First consider the case where $d_{\min}(u, v) \geq 0.81 \cdot d_{\max}(u, v)$. Initially, let $a = |N[u] \cap N[v]| \geq 2$. Since (u, v) is labelled as dissimilar by $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, we have $\tilde{\sigma}_c(u, v) < \epsilon$. Then with probability at least $1 - \delta$, $\sigma_c(u, v) < (1 + \frac{1}{2}\rho)\epsilon$. Without loss of generality, we first consider an update incident to u . If $d[u] = a$, then $\forall w \in N[u], w \in N[v]$. Thus deleting edge (u, w) will only decrease $\sigma_c(u, v)$. That is to say, in order to increase $\sigma_c(u, v)$, we need to consider insertions

of edge (u, w) . Otherwise if $d[u] > a$, since $d[u]$ and a are both integers, we must have $d[u] \geq a + 1$. Therefore,

$$d[u] \geq a + 1 > \frac{2a^2 + 2a + 1}{2a + 1}.$$

It can be easily verified that

$$\frac{a+1}{\sqrt{(d[u]+1)d[v]}} > \frac{a}{\sqrt{(d[u]-1)d[v]}}.$$

As a result, in considering the minimum number of affecting updates to cause (u, v) 's label flips from dissimilar to similar, we only need to focus on the edge insertions that increase $\sigma_c(u, v)$. Let $k = \lfloor 0.45\rho\epsilon^2 \cdot d_{\max}(u, v) \rfloor$ be the total number of such updates, and t be the number of such affecting updates that involve u , by the first bullet of Case 1 in Observation 2:

$$\begin{aligned} \sigma_c(u, v) &= \frac{a+k}{\sqrt{(d[u]+t)(d[v]+k-t)}} < \frac{a+k}{\sqrt{d[u]d[v]}} \\ &= \frac{a}{\sqrt{d[u]d[v]}} + \frac{k}{\sqrt{d[u]d[v]}} \\ &\leq (1 + \frac{1}{2}\rho)\epsilon + \frac{\lfloor 0.45\rho\epsilon^2 \cdot d_{\max}(u, v) \rfloor}{\sqrt{d[u]d[v]}} \\ &\leq (1 + \frac{1}{2}\rho)\epsilon + \frac{\frac{1}{2} \cdot 0.9\rho\epsilon^2 d_{\max}(u, v)}{\sqrt{d[u]d[v]}} \\ &\leq (1 + \frac{1}{2}\rho)\epsilon + \frac{\frac{1}{2}\rho\epsilon\sqrt{d[u]d[v]}}{\sqrt{d[u]d[v]}} \\ &= (1 + \rho)\epsilon \end{aligned}$$

The last inequality is because

$$\sqrt{d[u]d[v]} = \sqrt{d_{\min}(u, v) \cdot d_{\max}(u, v)} \geq \sqrt{0.81\epsilon^2 \cdot d_{\max}^2(u, v)} = 0.9\epsilon \cdot d_{\max}(u, v).$$

Therefore, after k arbitrary affecting updates, the dissimilar label of (u, v) remains valid with probability at least $1 - \delta$.

Then consider the case where $d_{\min}(u, v) < 0.81\epsilon^2 \cdot d_{\max}(u, v)$. Note that by Lemma 8.2, since $d_{\min}(u, v) < \epsilon^2 d_{\max}(u, v)$, (u, v) will only be labelled as dissimilar. To get the minimum number of affecting updates (u, v) can afford before its label flips, we only need to consider the updates that narrow the gap between $d_{\min}(u, v)$ and $d_{\max}(u, v)$. Suppose there are t edge deletions incident to the vertex with the maximum degree and there are $k - t$ edge insertion incident to the vertex with the minimum degree, where $k = \lfloor 0.19\epsilon^2 \cdot d_{\max}(u, v) \rfloor$. Then the degree after k updates in total becomes:

$$\begin{aligned} d'_{\max}(u, v) &= d_{\max}(u, v) - t \\ d'_{\min}(u, v) &= d_{\min}(u, v) + k - t. \end{aligned}$$

Current degrees satisfy the following inequality:

$$\begin{aligned} \epsilon^2 \cdot d'_{\max}(u, v) &= \epsilon^2 d_{\max}(u, v) - \epsilon^2 t \\ &> d_{\min}(u, v) + 0.19\epsilon^2 \cdot d_{\max}(u, v) - t \\ &\geq d_{\min}(u, v) + k - t = d'_{\min}(u, v) \end{aligned}$$

The first inequality is because $0.81\epsilon^2 \cdot d_{\max}(u, v) = (\epsilon^2 - 0.19\epsilon^2) \cdot d_{\max}(u, v) > d_{\min}(u, v)$. Therefore, after k arbitrary affecting updates the dissimilar label of (u, v) remains valid. \square

LEMMA 8.5. *If an edge (u, v) is labelled as similar by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, then with probability at least $1 - \delta$, (u, v) can afford at least $k = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\min}(u, v) \rfloor$ affecting updates before its label flips from similar to dissimilar.*

PROOF. Initially, let $a = |N[u] \cap N[v]| \geq 2$. Since (u, v) is labelled as similar by $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy, we have $\tilde{\sigma}_c(u, v) \geq \epsilon$. Then with probability at least $1 - \delta$, $\sigma_c(u, v) \geq (1 - \frac{1}{2}\rho)\epsilon$. Without loss of generality, we consider an update incident to u . Since we have $d[u] \geq a$ and $a \geq 2$, we have,

$$d[u] \geq a > \frac{2a^2 - 2a + 1}{2a - 1}.$$

Thus it can be easily verified that

$$\frac{a-1}{\sqrt{(d[u]-1)d[v]}} < \frac{a}{\sqrt{(d[u]+1)d[v]}}.$$

As a result in considering the minimum number of affecting updates to cause (u, v) 's label flips from similar to dissimilar, we only need to focus on the edge deletions that decrease $\sigma_c(u, v)$. Let $k = \lfloor \frac{1}{2}\rho\epsilon \cdot d_{\min}(u, v) \rfloor$ be the total number of such updates, and t be the number of such affecting updates that involve u , by the first bullet of Case 2 in Observation 2:

$$\begin{aligned} \sigma_c(u, v) &= \frac{a-k}{\sqrt{(d[u]-t)(d[v]-k+t)}} > \frac{a-k}{\sqrt{d[u]d[v]}} \\ &= \frac{a}{\sqrt{d[u]d[v]}} - \frac{k}{\sqrt{d[u]d[v]}} \\ &\geq (1 - \frac{1}{2}\rho)\epsilon - \frac{\lfloor \frac{1}{2}\rho\epsilon \cdot d_{\min}(u, v) \rfloor}{\sqrt{d[u]d[v]}} \\ &\geq (1 - \frac{1}{2}\rho)\epsilon - \frac{\frac{1}{2}\rho\epsilon\sqrt{d[u]d[v]}}{\sqrt{d[u]d[v]}} \\ &= (1 - \rho)\epsilon \end{aligned}$$

Therefore, after k arbitrary affecting updates, the dissimilar label of (u, v) remains valid with probability at least $1 - \delta$. \square

8.3 Distributed Tracking on Updates

For an edge (u, v) incident to u , we first put it into one of these two categories:

- if $d_{\min}(u, v) \geq 0.81\epsilon^2 \cdot d_{\max}(u, v)$, we track it by a DT instance $DT(u, v)$ with tracking threshold set to

$$\tau(u, v) = \lfloor 0.45\rho\epsilon^2 \cdot d_{\max}(u, v) \rfloor + 1; \quad (7)$$

- if $d_{\min}(u, v) < 0.81\epsilon^2 \cdot d_{\max}(u, v)$, we track it by the another DT instance $DT^*(u, v)$ with tracking threshold set to

$$\tau^*(u, v) = \lfloor 0.19\epsilon^2 \cdot d_{\max}(u, v) \rfloor + 1; \quad (8)$$

The correctness is proven by Lemma 8.4 and 8.5. Note that for the second case, instead of using the update affordability as tracking threshold directly, we set the gap between $\epsilon^2 d_{\max}(u, v)$ and $d_{\min}(u, v)$ as the tracking threshold for $DT^*(u, v)$ and track the number of affecting updates with $DT^*(u, v)$.

For $DT(u, v)$ and $DT^*(u, v)$, we organize them together with $DtHeap(u)$ on u as in Section 2.4.

8.4 Algorithm Procedure

In this subsection, we first consider the *DynELM under cosine similarity* algorithm.

The running process of *DynELM* under cosine similarity is very like the one in Section 6 and it is outlined as follows:

- **Step 1.** Initialize the set of label-flipping edges $\mathcal{F} \leftarrow \emptyset$; and increment s_u and s_w (by 1), respectively.
- **Step 2.** There are two cases:
 - Case 1: this update is an insertion. Insert (u, w) into G and label it by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy. If (u, w) is labelled as similar, add (u, w) to \mathcal{F} . Moreover, if (u, w) satisfies the first bullet in Section 8.3, create $DT(u, w)$ with $\tau(u, w)$; otherwise create $DT^*(u, w)$ with $\tau^*(u, w)$.
 - Case 2: this update is a deletion. If (u, w) is labelled as similar, add (u, w) to \mathcal{F} . Delete (u, w) from G ; and delete $DT(u, w)$ or $DT^*(u, w)$.
- **Step 3.** While there is a checkpoint-ready entry in $DtHeap(u)$, pop the entry (from the top). Let $DT(u, v)$ be the DT instance corresponding to this entry. Instruct u to inform the coordinator (u, v) . When $DT(u, v)$ is mature, relabel (u, v) by the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy. If its label flipped, add (u, v) to \mathcal{F} . Remove its entry from $DtHeap(v)$, and if it satisfies the first bullet in Section 8.3, create $DT(u, v)$ with $\tau(u, v)$; otherwise, create $DT^*(u, v)$ with $\tau^*(u, v)$. Repeat until there is no checkpoint-ready entry in $DtHeap(u)$.
- **Step 4.** Perform a symmetric process of Step 3 and 4 for w .
- **Step 5.** Return \mathcal{F} , the set of edges whose labels flipped.

8.4.1 Theoretical Analysis.

In this subsection, we analyze the details of the *DynELM* algorithm under cosine similarity.

The implementation is the same as in Section 6.1, and thus given a failure probability δ^* and the total number of invocations of the $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy κ , the cost of each invocation of $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy is bounded by $O(\log n \cdot \log \frac{\kappa}{\delta^*})$. Since for an vertex u , its neighbors are maintained in $DtHeap(u)$, the space consumption of each vertex is still bounded by $O(1 + d[u])$. Thus the overall space consumption directly follows Lemma 6.4. And the correctness and failure probability directly follows Lemma 6.5.

Now it suffices to bound the amortized update cost. For each DT instance, the maturity cost maintains the same as in the original work, which is bounded $O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})$, where κ is number of invocations of $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy. Consider an update (u, w) , the same crucial observation is that the update of (u, w) can *only* contribute (via a counter increment) to the maturity of the DT instances of its affected edges, which exist at the *current moment*. Let $DT(u, v_1^*)$ and $DT^*(u, v_2^*)$ be the instance with the *smallest* threshold value $\tau(u, v_1^*)$ and $\tau^*(u, v_2^*)$ among all the affected DT instances at the current moment, respectively.

For $DT(u, v_1^*)$, we have $d_{\min}(u, v_1^*) \geq 0.81\epsilon^2 \cdot d_{\max}(u, v_1^*)$ at the moment when $DT(u, v_1^*)$ is created. Let $d'[u]$ be the degree of u when $DT(u, v_1^*)$ was created. We claim that the degree of u , $d[u]$, at the current moment is at most $d'[u] + \tau(u, v_1^*)$ for the same reason as in Section 6.1. Since $d_{\min}(u, v_1^*) \geq 0.81\epsilon^2 \cdot d_{\max}(u, v_1^*)$, we have:

$$\begin{aligned} \tau(u, v_1^*) &= \lfloor 0.45\rho\epsilon^2 d_{\max}(u, v_1^*) \rfloor + 1 \geq 0.45\rho\epsilon^2 d_{\max}(u, v_1^*) \\ &\geq 0.45\rho\epsilon^2 d'[u]. \end{aligned}$$

Thus,

$$d[u] \leq d'[u] + \tau(u, v_1^*) \leq \frac{\tau(u, v_1^*)}{0.45\rho\epsilon^2} + \tau(u, v_1^*) = O(1) \cdot \tau(u, v_1^*). \quad (9)$$

For $DT^*(u, v_2^*)$, we have $d_{\min}(u, v_2^*) < 0.81\epsilon^2 \cdot d_{\max}(u, v_2^*)$ at the moment when $DT^*(u, v_2^*)$ is created. Let $d''[u]$ be the degree of u when $DT^*(u, v_2^*)$ was created. We claim that the degree of u , $d[u]$, at the current moment is at most $d''[u] + \tau(u, v_2^*)$ for the same reason. Therefore

$$\tau^*(u, v_2^*) = \lfloor 0.19\epsilon^2 \cdot d_{\max}(u, v) \rfloor + 1 \geq 0.19\epsilon^2 \cdot d_{\max}(u, v) \geq 0.19\epsilon^2 \cdot d''[u].$$

Then we have

$$d[u] \leq d''[u] + \tau^*(u, v_2^*) \leq \frac{\tau^*(u, v_2^*)}{0.19\epsilon^2} + \tau^*(u, v_2^*) = O(1) \cdot \tau^*(u, v_2^*). \quad (10)$$

Furthermore, as each of the affected $DT(u, v)$ and $DT^*(u, v)$ requires at least $\tau(u, v)$ and $\tau^*(u, v)$ affecting updates to mature, respectively, the current update of (u, w) is actually accounted for only $\frac{1}{\tau(u, v)}$ of the cost of the $DT(u, v)$ maturity as well as the following edge re-labeling cost and $\frac{1}{\tau^*(u, v)}$ of the cost corresponding to $DT^*(u, v)$. Summing over all neighbors of u , the amortized cost of an update (u, w) is bounded by:

$$\begin{aligned} & \sum_{v \in N[u]: v \text{ belongs to } DT(u, v)} \frac{O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})}{\tau(u, v)} + \\ & \sum_{v \in N[u]: v \text{ belongs to } DT^*(u, v)} \frac{O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})}{\tau^*(u, v)} \\ & \leq \left(\frac{d[u]}{\tau(u, v_1^*)} + \frac{d[u]}{\tau^*(u, v_2^*)} \right) \cdot O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*}) \\ & = O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*}) \end{aligned} \quad (11)$$

By symmetry, all affected DT instances corresponding to w also charge $O(\log^2 n + \log n \cdot \log \frac{\kappa}{\delta^*})$ cost to the update of edge (u, w) . Combining with the fact that the number of invocations of $(\frac{1}{2}\rho\epsilon, \delta)$ -strategy is bounded by the number of updates, the amortized cost of each update remains $O(\log^2 n + \log n \cdot \log \frac{M}{\delta^*})$, where M is the number of updates in a sequence.

For the *DynStrClu under cosine similarity* algorithm, it follows the same procedures as *DynStrClu* algorithm except for the maintenance of edge labelling. From the above analysis, it is easy to be seen that all guarantees in the *DynStrClu* algorithm are admitted. Thus, Theorem 8.1 is proven.

9 EXPERIMENTS

Datasets. We deploy 15 real datasets in the experiments. Detailed descriptions of all these 15 datasets can be found at the Stanford Network Analysis Project (SNAP)². We pre-process each of these datasets in the following way: (i) treat the graph as undirected; (ii) remove all self-loops and duplicate edges; and (iii) relabel vertex identifiers to be in the set $\{1, \dots, n\}$. The meta information of all processed datasets are listed in Table 1. The first five datasets highlighted in bold are chosen as representatives: their vertex counts increase roughly geometrically (factor two), and each has a reasonable average degree; and they are used to explore both clustering effectiveness and efficiency of the algorithms, with varying parameter settings. For easy reference, we rename the five representatives as *Slashdot*, *Notre*, *Google*, *Wiki* and *LiveJ*, respectively. In addition, the last dataset in bold, renamed as *Twitter*, with 1.2 billion edges, is further used to study the scalability of our proposed algorithms. The remaining nine datasets are then listed, in ascending order of n .

9.1 Clustering Visualisations

We start with the visualisation results of our 5 representative datasets with μ set to 5 under both similarities. Since the sizes of these graphs are large, we only show the clustering results of the top-20 clusters w.r.t. cluster size (i.e., the number of vertices contained in a cluster). Different clusters are shown in different colours. For a hub u that belongs to

²<http://snap.stanford.edu/data/index.html>

Table 1. Dataset Meta Information and Memory Footage over the Whole Update Sequence ($K = 10^3$, $M = 10^6$ and $B = 10^9$)

Datasets	#Vertices	#Edges	#Updates	Memory Footage (GigaBytes)			
				DynELM	DynStrClu	pSCAN	hSCAN
soc-Slashdot0811	77.3K	469K	4.69M	0.50	0.58	0.47	0.82
web-NotreDame	326K	1.09M	10.9M	1.17	1.87	1.10	1.93
web-Google	876K	4.32M	43.2M	4.51	6.23	3.62	7.54
wiki-topcats	1.79M	25.4M	254M	25.79	29.39	(26.82)	(51.66)
soc-LiveJournal1	4.85M	42.9M	429M	43.51	58.85	(44.13)	(87.70)
<i>email-Eu-core</i>	0.99K	16.1K	161K	0.02	0.02	0.02	0.03
<i>ca-GrQc</i>	5.24K	14.5K	145K	0.02	0.03	0.02	0.03
<i>ca-CondMat</i>	23.1K	93.4K	934K	0.10	0.16	0.09	0.17
<i>soc-Epinions1</i>	75.8K	406K	4.06M	0.43	0.54	0.41	0.71
<i>dblp</i>	317K	1.05M	10.5M	1.11	1.97	1.05	1.86
<i>amazon0601</i>	403K	2.44M	24.4M	2.54	4.08	2.43	4.27
<i>soc-Pokec</i>	1.63M	22.3M	223M	22.44	24.90	(18.26)	(30.85)
<i>as-skitter</i>	1.70M	11.1M	111M	11.71	14.27	(8.47)	(42.32)
<i>wiki-Talk</i>	2.39M	4.66M	46.6M	5.48	7.29	(4.71)	(24.06)
twitter-2010	41.65M	1.20B	1.32B	204.09	257.46	(135.11)	(273.32)

Table 2. Approximate Clustering Quality under Jaccard Similarity: Mis-Labelled Rate, Overall Clustering Quality (ARI) and Individual Cluster Quality

	<i>Slashdot</i> ($\epsilon = 0.15$)		<i>Notre</i> ($\epsilon = 0.19$)		<i>Google</i> ($\epsilon = 0.15$)		<i>Wiki</i> ($\epsilon = 0.19$)		<i>LiveJ</i> ($\epsilon = 0.6$)		<i>Twitter</i> ($\epsilon = 0.2$)	
	$\rho = 0.01$	$\rho = 0.5$	$\rho = 0.01$	$\rho = 0.5$	$\rho = 0.01$	$\rho = 0.5$	$\rho = 0.01$	$\rho = 0.5$	$\rho = 0.01$	$\rho = 0.5$	$\rho = 0.01$	$\rho = 0.5$
%mis-labelled	0.02%	2.37%	0.10%	5.86%	0.16%	8.74%	0.04%	1.82%	0.14%	6.33%	0.01%	0.07%
ARI	.996386	.971871	.999748	.962753	.998872	.970845	.999933	.989068	.999767	.999470	.994647	.976576
Top-k Clusters	Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality	
	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg
1	.987	.987	.961	.961	1.00	1.00	1.00	1.00	.999	.999	1.00	1.00
5	.987	.996	.961	.988	1.00	1.00	1.00	1.00	.998	.999	.953	.990
10	.987	.998	.961	.994	1.00	1.00	1.00	1.00	.987	.998	.992	.983
20	.987	.999	.961	.997	1.00	1.00	.997	.999	.980	.980	.650	.965
50	.987	.999	.961	.999	0.977	.999	.761	.989	.987	.998	.998	.929
100	.987	.999	.961	.999	0.977	.999	.345	.967	.952	.997	.990	.853

Table 3. Approximate Clustering Quality under Cosine Similarity: Mis-Labelled Rate, Overall Clustering Quality (ARI) and Individual Cluster Quality

	<i>Slashdot</i> ($\epsilon = 0.3$)				<i>Notre</i> ($\epsilon = 0.36$)				<i>Google</i> ($\epsilon = 0.3$)				<i>Wiki</i> ($\epsilon = 0.34$)				<i>LiveJ</i> ($\epsilon = 0.67$)			
	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$	$\rho = 0.01$	$\rho = 0.1$				
%mis-labelled	0.11%	0.83%	0.19%	2.91%	0.33%	2.61%	0.08%	0.61%	0.19%	0.14%	0.97321	.973321	.924053							
ARI	.989941	.978194	.984992	.943016	.969585	.709121	.975939	.794518												
Top-K Clusters	Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality		Indv. Cluster Quality					
	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg	min	avg				
1	.981	.981	.949	.949	.971	.971	.867	.867	.992	.992	.716	.716	.995	.995	.833	.833				
5	.958	.979	.853	.916	.890	.966	.710	.855	.959	.983	.716	.754	.900	.952	.550	.742				
10	.958	.989	.435	.846	.890	.980	.710	.902	.958	.980	.423	.696	.900	.969	.550	.773				
20	.857	.985	.000	.787	.890	.988	.710	.929	.808	.961	.423	.681	.557	.949	.288	.709				
50	.545	.980	.000	.790	.782	.989	.380	.891	.767	.957	.177	.656	.557	.946	.276	.704				
100	.545	.986	.000	.817	.782	.992	.197	.879	.509	.956	.026	.607	.439	.954	.057	.678				

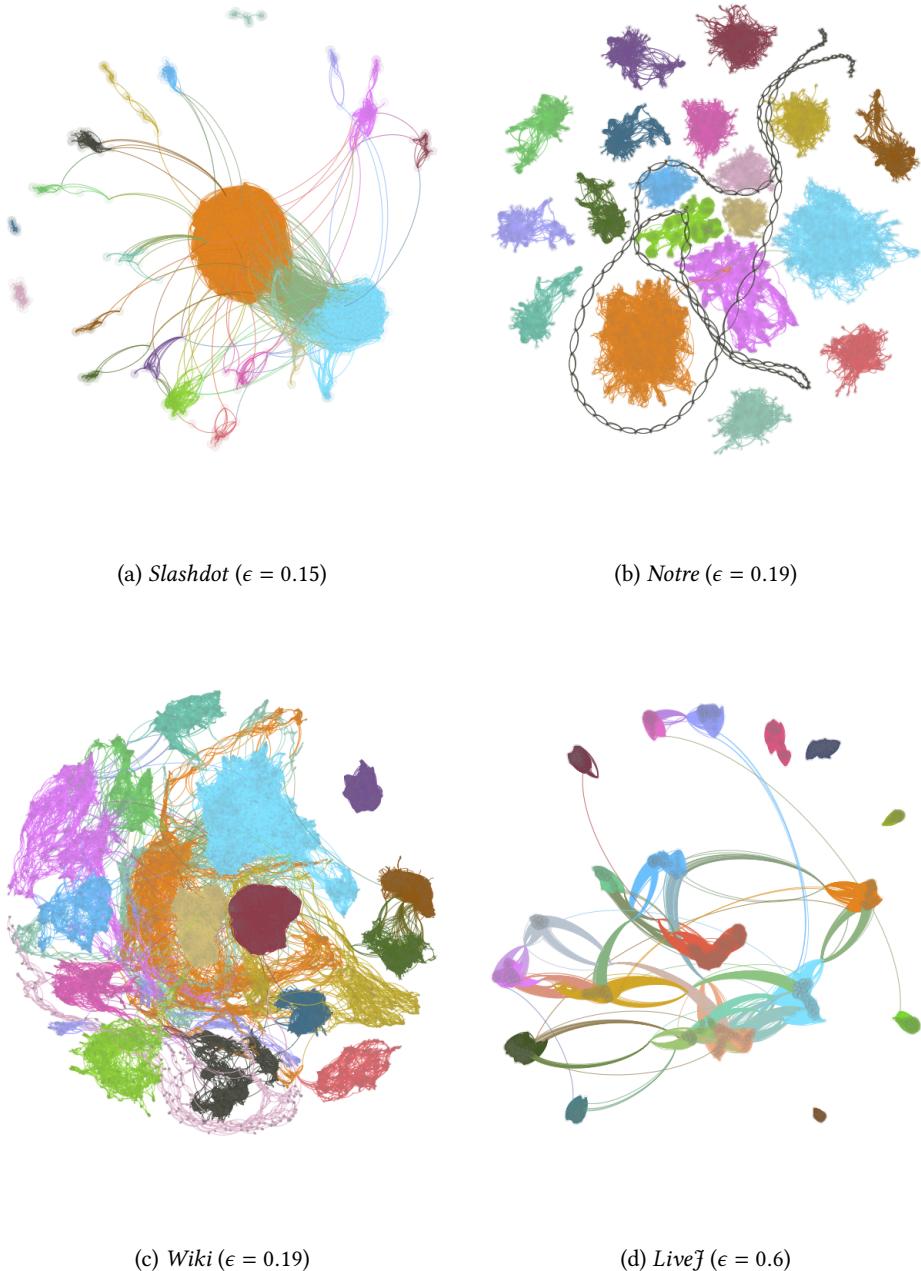


Fig. 4. Clustering Visualisations under Jaccard Similarity: Top-20 Clusters with $\mu = 5$

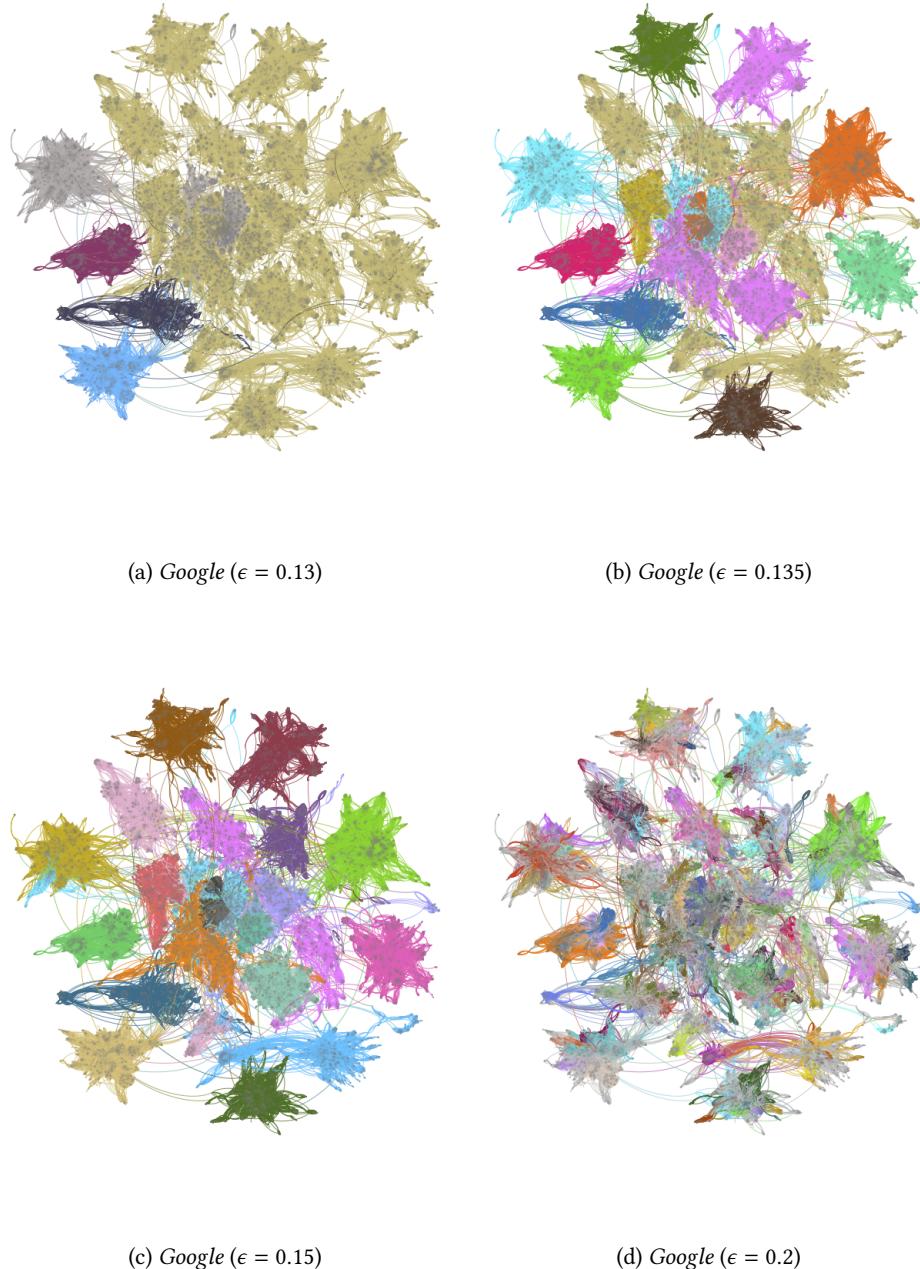


Fig. 5. Evolutions of the Top-20 Clusters under Jaccard Similarity on *Google* with varying ϵ and $\mu = 5$

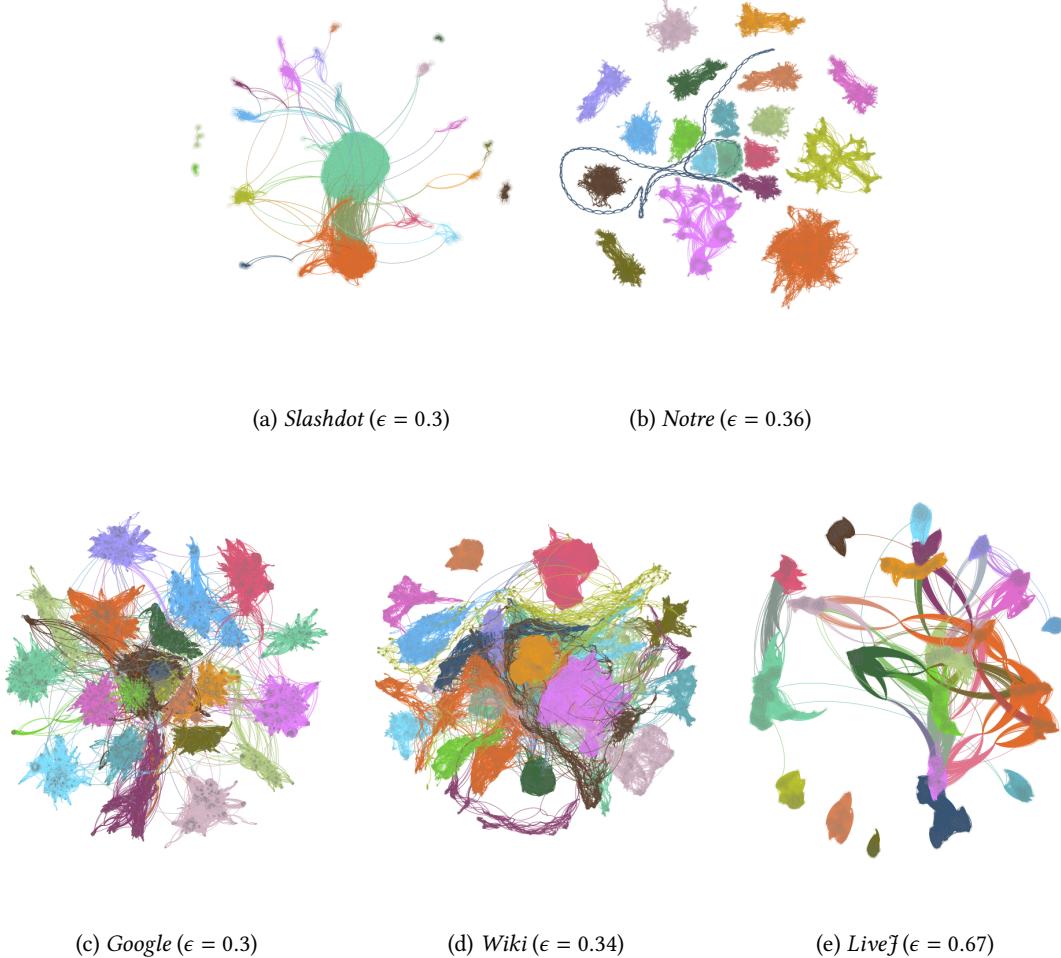


Fig. 6. Clustering Visualisations under Cosine Similarity: Top-20 Clusters with $\mu = 5$

multiple clusters, we only assign it to the cluster containing u 's "smallest" similar core neighbour (i.e., the similar core neighbour vertex with the smallest id). Furthermore, we omit the noises in the visualisations.

In choosing proper ϵ for each dataset, our target is that in the clustering results the sizes of each cluster do not vary too much. We take *Google* as an example to show the effect of ϵ on *StrCluResult* in Figure 5. The vertices shown are those in the top-20 clusters when $\epsilon = 0.15$ as shown in Figure 5(c). $\epsilon = 0.15$ is also the proper ϵ we choose for *Google*. If we increase ϵ to 0.2 in Figure 5(d), the clusters are separated into more clusters whose sizes are smaller comparing to $\epsilon = 0.15$. The reason is that when ϵ is increased, some edges which are originally labelled as similar under $\epsilon = 0.15$ will become dissimilar. Thus some core vertices will become non-core vertices and some similar edges linking two core vertices will be “broken”. As a result, more clusters with smaller sizes are produced. On the contrary, if we decrease ϵ

to 0.135 and 0.13 in Figure 5(b) and Figure 5(a), the clusters begin to merge with each other to form larger clusters. It is because under smaller ϵ , some edges originally labelled as dissimilar under $\epsilon = 0.15$ become similar. Thus some non-core vertices become core vertices and more dissimilar edges become similar core edges linking core vertices that originally belong to different clusters. The evaluation of *StrCluResult* on *Google* with varying ϵ under Jaccard similarity is shown in Figure 5. For other datasets, the chosen ϵ value and the visualisation results of the top-20 clusters are shown in Figure 4.

For cosine similarity, the visualisation results are shown in Figure 6 with different colours representing different clusters. The value of ϵ is picked such that the visualisation results are similar to the results shown in Figure 4 and Figure 5(c). From these visualisation results, it can be confirmed that the intra-cluster edges are much denser than the inter-cluster edges, which indicates the quality of structural clustering results is good and the results are meaningful for human to understand.

On the other hand, comparing Figure 4, Figure 5(c) and Figure 6, an observation is that while the clustering results is similar, the values of ϵ under cosine similarity are generally larger than the values of ϵ under Jaccard similarity. The reason is that by the definition of these two similarities, the cosine similarity between two vertices is always no smaller than the Jaccard similarity between them. To see this, consider an edge $(u, v) \in E$ and without loss of generality, suppose $d[u] \geq d[v]$:

$$|N[u] \cup N[v]| \geq d[u] \geq \sqrt{d[u] \cdot d[v]}.$$

Therefore,

$$\frac{|N[u] \cap N[v]|}{|N[u] \cup N[v]|} \leq \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}}.$$

9.2 Approximate Clustering Quality

Next, we evaluate the quality of the ρ -approximate *StrCluResult* computed by *DynELM* (equivalently, by *DynStrClu*) on the five representative datasets plus *Twitter* under Jaccard similarity and on the five representative datasets under cosine similarity. Our evaluation adopts three measurements: (i) *mis-labelled rate*, (ii) *overall clustering quality*, and (iii) *individual cluster quality*; the details of these measurements will be introduced shortly. Furthermore, the approximate clustering results we considered are obtained with $\rho = 0.01$ and $\rho = 0.5$, respectively, under $\mu = 5$ and customized ϵ 's (shown beside the dataset names in Table 2). These customized ϵ values are chosen based on the visualisations of the clustering results shown in Figure 4, Figure 5(c) and Figure 6 Under these ϵ values, the obtained clustering results are natural to human sensibility: the intra-cluster edges are much denser than the inter-cluster edges.

Mis-Labelled Rate. Recall that the *StrCluResult* is uniquely determined by the edge labelling. If the ρ -approximate edge labelling, $\mathcal{L}_{\epsilon,\rho}(G)$, is highly similar to its exact counterpart, $\mathcal{L}_\epsilon(G)$, their corresponding clustering results should be highly similar. To measure the similarity between $\mathcal{L}_\epsilon(G)$ and $\mathcal{L}_{\epsilon,\rho}(G)$, we consider the *mis-labelled rate*, which is defined as the *percentage* of the edges that have different labels in these two labelling's. As shown in Table 2, when $\rho = 0.5$, the mis-labelled rates across the six datasets are 2.37%, 5.86%, 8.74%, 1.82%, 6.33% and 0.07% under Jaccard similarity, respectively. While these mis-labelled rates are small already, these numbers can be even significantly smaller when $\rho = 0.01$. They are just: 0.02%, 0.1%, 0.16%, 0.04%, 0.14% and 0.01%, respectively: none of them is more than 16/10000. In other words, when $\rho = 0.01$, all the approximate edge labellings are almost identical to the exact labellings on these datasets. Therefore, one can expect that by setting $\rho = 0.01$, the approximate results computed by *DynELM* are

“merely identical” to the exact ones under Jaccard similarity. To quantify this intuition, we study the *overall clustering quality* and the *individual cluster quality*.

Overall Clustering Quality. We quantify the overall clustering quality with the *Adjusted Rand Index* (ARI) [20], which is adopted by some of the authors of SCAN in their later work [41] to measure the quality of structural clustering results. In fact, the ARI is a widely adopted [3, 20, 22, 23, 26, 27, 36] similarity measurement between two *partitions* of a same set. The ARI values are between 0 and 1, where *the closer the ARI value is to 1, the more similar the two partitions are*. However, in general, ARI is not applicable to *StrCluResult*'s, because the *StrCluResult* is not necessarily a partition of the vertices (some non-core vertices may belong to none or multiple clusters). To address this subtlety, we assign each non-core vertex u *only* to the cluster which contains u 's “smallest” similar core neighbour (in terms of the identifier value), and ignore all the noise vertices. In Table 2, when $\rho = 0.5$, the ARI scores (between the approximate and the exact clusterings) across all the six datasets are at least 0.96. Even better, when $\rho = 0.01$, these scores are at least 0.994 (this worst value occurs on *Twitter*), where, impressively, the score is up to 0.9999 on *Wiki*. These high ARI scores indicate that the overall ρ -approximate clusterings are of very high quality under Jaccard similarity.

Individual Cluster Quality. Since the high overall clustering quality may not necessarily reflect the high quality of each individual cluster, we thus look into the individual cluster quality of the approximate results. Specifically, consider a cluster C in an ρ -approximate clustering result; let $S \subseteq C$ be the set of all the vertices in C that are *core* under the *exact* edge labelling. Moreover, let C^* be the set of *exact clusters* that contain at least one core vertices in S . The *individual cluster quality* of the approximate cluster C is defined as the *largest* Jaccard similarity between C and each cluster $C' \in C^*$, i.e., $\max_{C' \in C^*} \frac{|C \cap C'|}{|C \cup C'|}$. The closer this value is to 1, the more similar of the approximate cluster C is to an exact cluster C' , and hence, the higher quality of C . Table 2 shows the *minimum* and *average* individual cluster qualities among the top- k largest (in terms of size) ρ -approximate clusters on the six datasets, where $k \in \{1, 5, 10, 20, 50, 100\}$. The average value is the average quality of the top- k clusters; and the minimum value shows how bad the cluster with the *least* quality is. Furthermore, the “worst” minimum and average values across all cases are underlined in the table.

For $\rho = 0.5$, the average individual qualities are at least 0.952 across all cases. However, as highlighted in bold in Table 2, we do see two big drops in the minimum quality among the top-100 and top-50 clusters respectively on *Google* and *Wiki*. We looked into these two cases and eventually found out the cause behind these drops. When $\rho = 0.5$, some similar edges have been mis-labelled as dissimilar due to the does-not-matter case. As a result, the corresponding exact clusters happened to split into two smaller clusters in the approximate clusterings, resulting in the low individual cluster qualities. In contrast, for $\rho = 0.01$, these cases did not happen and both the average and minimum individual cluster qualities are consistently good across all cases; they are 0.987 and 0.853, respectively, where the latter indicates that even for the “worst” cluster, it still has 0.853 Jaccard similarity with its corresponding exact cluster. Let alone that in most of other cases, the minimum individual quality is at least 0.95.

In summary, the quality of the approximate results obtained with $\rho = 0.01$ are consistently high in terms of all the three measurements under Jaccard similarity. We thus set $\rho = 0.01$ as our default value in the subsequent experiments. As we will see shortly, with such a tiny sacrifice in the clustering quality, our algorithms can gain up to *three-orders-of-magnitude* improvements in efficiency.

9.3 Comparison between Jaccard Similarity and Cosine Similarity in Approximate Quality

For cosine similarity, the results of the above three measures are shown in Table 3.

In Table 2, if we list the five representative datasets in ascending orders with respect to the mis-labelled rate, the order will be: *Slashdot*, *Wiki*, *Notre*, *LiveJ* and *Google* with $\rho = 0.01$; and *Wiki*, *Slashdot*, *Notre*, *LiveJ* and *Google* with $\rho = 0.1$. Comparing this order to the mis-labelled rate presented in Table 3, these two orders above match the orders obtained from Table 3 under $\rho = 0.01$ and $\rho = 0.5$, respectively. Moreover, when ρ is both set to 0.01, the mis-labelled rates for all 5 datasets under cosine similarity are all higher than those under Jaccard similarity. As discussed above, when producing similar clustering results, the value of ϵ under cosine similarity is in general larger than under Jaccard similarity. Therefore, the value of $\rho\epsilon$ is larger under cosine similarity so the range of “don’t care case” is larger for cosine similarity. As a result, an edge where the similarity between its endpoints is close to ϵ is more likely to be mis-labelled. Thus the mis-labelled rate is higher under cosine similarity. The value of ARI and the result of individual cluster quality also confirm that the approximation quality for cosine similarity is worse than that for Jaccard similarity when ρ is both set to 0.01.

When ρ is set to 0.1, however, the approximation quality for cosine similarity drops significantly, which is shown by both ARI and individual cluster quality. Although for most datasets, the worst average individual cluster quality is larger than 0.6, the worst minimum individual cluster similarities all go smaller than 0.2. One case worth to mention is *Slashdot*. The 23-rd largest cluster in the approximation clustering is consist of some vertices that are not core vertices in the exact clustering result. Therefore, the Jaccard similarity between this cluster and any other clusters in the exact clustering result is all 0, making the minimum individual cluster similarity to be 0. Note that from the visualisation of *Slashdot*, it can be seen that there are only two big clusters in this graph, and the sizes of remaining clusters are far smaller than the big ones. That’s why the ARI of *Slashdot* under $\rho = 0.1$ is still very large.

From the above analysis, we can confirm that when the target is to produce similar clustering results with ρ -approximate notion, Jaccard similarity would be more favourable since its *StrCluResult* better reflects the exact clustering result.

9.4 Efficiency Experiment Setup

Update Simulations. In addition to the original edges, we generate a sequence of edge insertions and deletions to simulate the *update process* for each dataset. Specifically, for a fixed value of η , we generate an update independently with probability $\frac{1}{1+\eta}$ to insert a *new* edge, and probability $\frac{\eta}{1+\eta}$ to delete an *existing* edge. In this way, on average, the frequency of deletions is roughly η proportion of the frequency of the insertions. For a deletion, the edge to be removed is picked *independently and uniformly at random* from the current edges. For an insertion, we generate all of them *consistently*, with one of the following three strategies:

- *Random-Random (RR)*: Uniformly-at-random pick an edge that is *not* in the current graph.
- *Degree-Random (DR)*: First, choose a vertex u with probability $\frac{d[u]}{2m}$, where m is the current number of edges in the graph. If the degree of u is $n - 1$, repeat this step. Second, uniformly-at-random pick a vertex v from those vertices not currently adjacent to u .
- *Degree-Degree (DD)*: Similar to *DR*, but the second vertex v is also chosen (independently) with probability $\frac{d[v]}{2m}$. If (u, v) is already in the graph or a self loop, repeat.

Denote the number of *original* edges in the graph (after our pre-processing) by m_0 , shown in the meta information. Except *Twitter*, for each of the 14 datasets, we generate $9 \cdot m_0$ updates, including both deletions (if $\eta > 0$) and insertions. The *update process* is simulated as follows. Starting from an empty graph, insert each of the m_0 original edges one by one, and then perform each of the $9 \cdot m_0$ generated updates. Therefore, in the update process, in total $10 \cdot m_0$ updates are

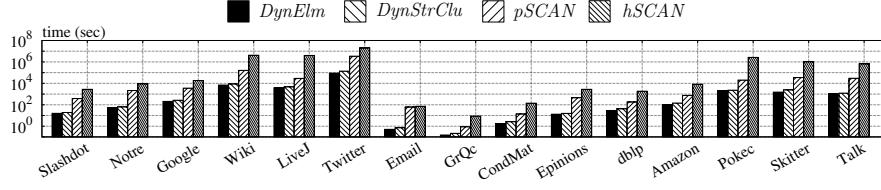
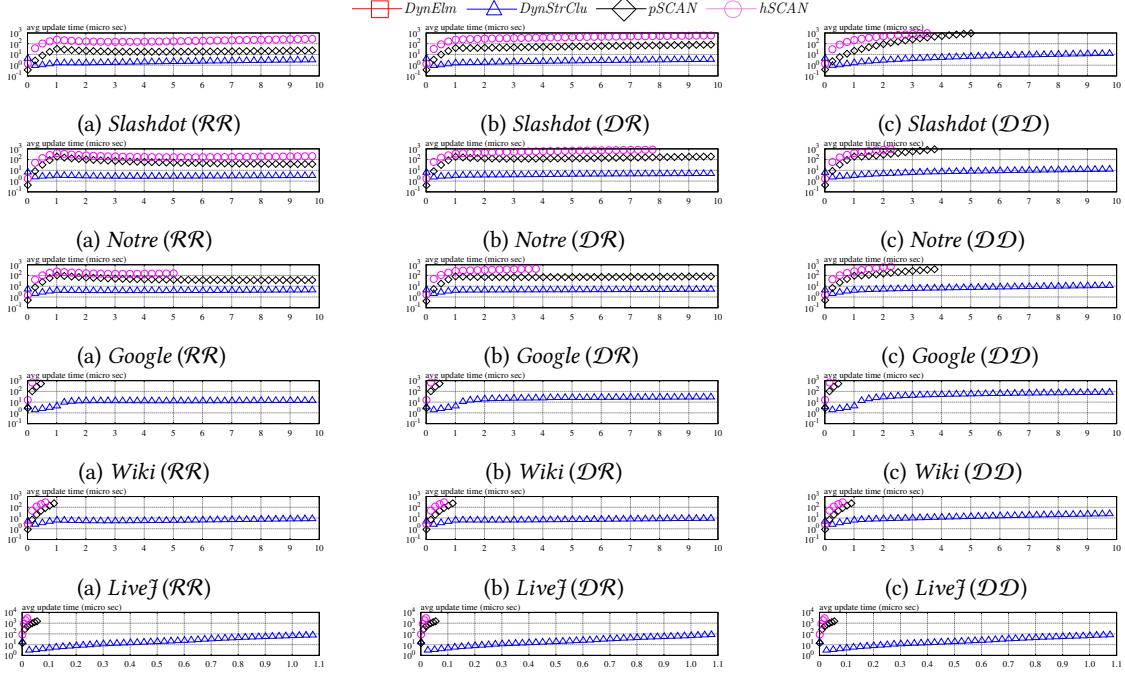
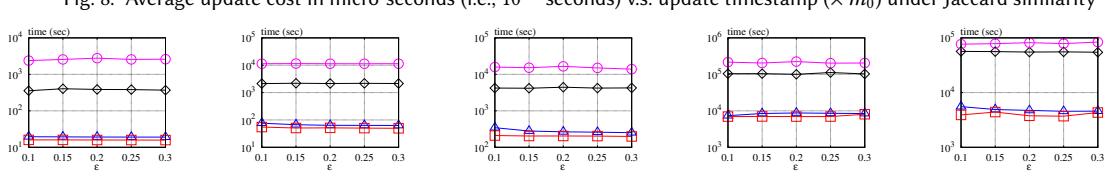


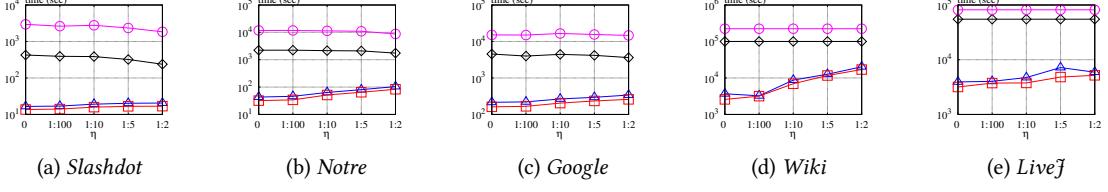
Fig. 7. Overall running time (logarithmic scale) under default settings for all four algorithms on all 15 datasets.

Fig. 8. Average update cost in micro-seconds (i.e., 10^{-6} seconds) v.s. update timestamp ($\times m_0$) under Jaccard similarityFig. 9. Overall running time v.s. ϵ on default settings ($\mu = 5$ and $\rho = 0.01$) under Jaccard similarity

performed on each graph. As for *Twitter*, we only further generate $0.1 \cdot m_0$ updates, because $m_0 = 1.2 \times 10^9$ is already large enough. The total numbers of updates can be found in Table 1.

Parameters. The experiments are conducted with the following parameter settings under Jaccard similarity, where the default value of each parameter is highlighted in bold and underlined.

- $\mu = \underline{5}$ and $\delta^* = \underline{1/n}$;

Fig. 10. Overall running time v.s. η on default settings ($\epsilon = 0.2$, $\mu = 5$ and $\rho = 0.01$) under Jaccard similarity

	DynElm		pSCAN		hSCAN
--	--------	--	-------	--	-------

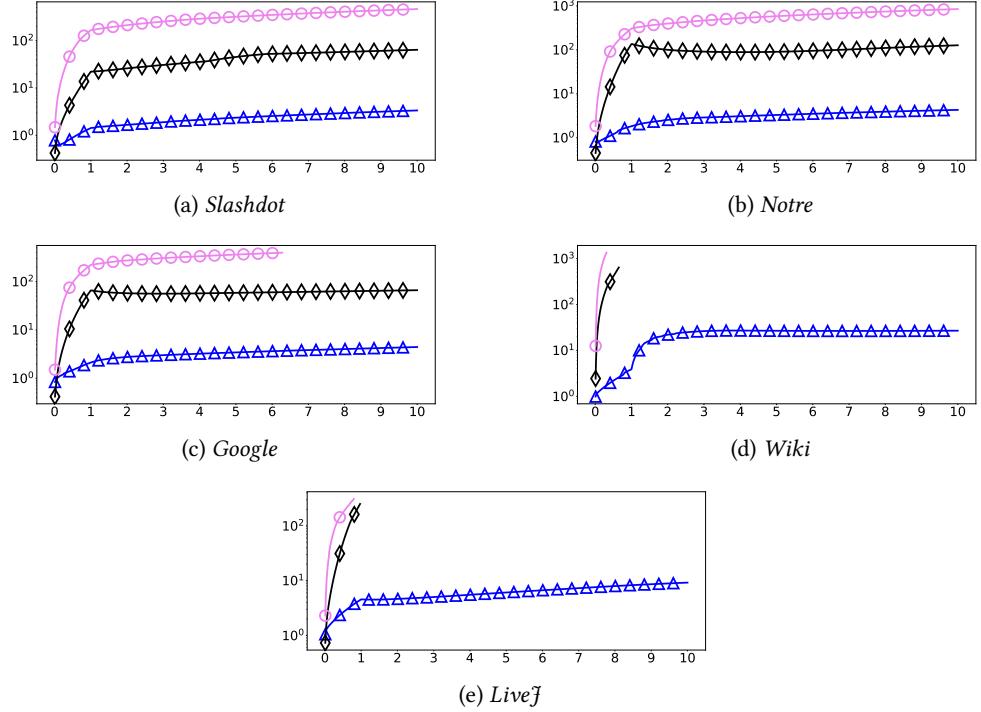


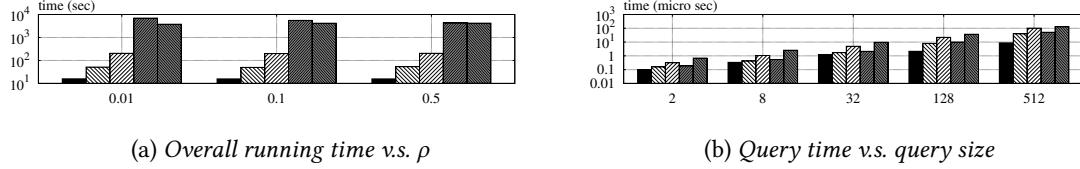
Fig. 11. Average update cost in micro-seconds v.s. update timestamp under cosine similarity

- $\epsilon \in \{0.1, 0.15, \underline{0.2}, 0.25, 0.3\}$;
- $\rho \in \{\underline{0.01}, 0.05, 0.1, 0.2, 0.5\}$;
- $\eta \in \{0, 1/100, \underline{1/10}, 1/5, 1/2\}$;
- insertion strategies: $\{\mathcal{RR}, \underline{\mathcal{DR}}, \mathcal{DD}\}$.

Unless stated otherwise, when a particular parameter is varied, all the other parameters are set to their default values.

For cosine similarity, we set all parameters to the default setting above except for $\epsilon = 0.6$, which is also the default value of ϵ in state-of-the-art works [5, 6, 38].

Competitors. As the superiority of *pSCAN* and *hSCAN* over other existing methods has been shown in their seminal papers [6, 39], in our experiments, we focus on the comparisons between *pSCAN*, *hSCAN*, and our methods: *DynELM*

Fig. 12. The bars in each group respectively represent in order: *Slashdot*, *Notre*, *Google*, *Wiki*, and *LiveJ*.

and *DynStrClu*. All these four methods are implemented in C++ (compiled by gcc 9.2.0 with -O3). The implementations of *pSCAN* and *hSCAN* are provided by their respective authors. Following the instructions in their seminal papers, we adapted them to work with Jaccard similarity. We use their source code directly for the experiments under cosine similarity.

Machine and OS. All the experiments are run on a machine equipped with an Intel(R) Xeon(R) CPU (E7-4830 v2 @ 2.20GHz) and 1TB memory running on Linux (CentOS 7.2).

9.5 Overall Performance on All Datasets

Overall Efficiency on All Datasets. We next compare the time efficiency under default settings under Jaccard similarity. The results are shown in Figure 7. *DynELM* is always the most efficient, followed by the slightly slower *DynStrClu*. The reason for this is that although *DynStrClu* needs to maintain some extra data structures, the major computational cost still lies in maintaining edge labels. Except *Twitter*, for each dataset, both of our methods can process the whole update sequence within three hours (and within 30 hours on *Twitter*). However, *pSCAN* and *hSCAN* cannot make that bound. Worse still, on *Twitter*, for 30 hours, both *pSCAN* and *hSCAN* can only complete $0.057 \cdot m_0$ and $0.023 \cdot m_0$ updates, respectively; on the other five large datasets: *Wiki*, *LiveJ*, *Pokec*, *Skitter*, and *Talk*, they also failed to finish the first m_0 insertions of the corresponding original edges within three hours. We therefore *underestimate* their running times. For *Twitter*, we simply scale the 30 hours according to their progresses to the end of the whole update process, i.e., $1.1 \cdot m_0$, as the update cost increases with the growth of the graph. For the other five large datasets, the estimated time is computed as follows: (i) we run the two algorithms on the graph with all original edges, as if in the static case; (ii) based on the state after first step, let the two algorithms process the $9 \cdot m_0$ generated updates for one hour; (iii) according to the percentage of updates processed within the hour, we estimate the total update time for the $9 \cdot m_0$ updates by scaling; and (iv) add the three hours for the first m_0 insertions to this estimated time.

From Figure 7, except for *GrQc*, *CondMat* and *dblp*, both of our methods are 10 \times faster than *pSCAN* on every dataset. In particular, on the aforementioned large datasets, our methods outperform *pSCAN* by up to two orders of magnitude. This is because on these datasets, the edge in an update is more likely to incident on a large-degree vertex u , for which *pSCAN* needs $O(d[u])$ time, while our methods guarantee $O(\log^2 n)$ amortized cost (according to our current parameter setting). Finally, *hSCAN* is slower than *pSCAN* on every dataset, as its update cost complexity is $O(n \log n)$, having a factor of $\log n$ over the update cost of *pSCAN*. Particularly on *Wiki*, *LiveJ* and *Pokec*, *DynELM* is almost 1000 \times faster than *hSCAN*.

Memory Consumption. The last four columns in Table 1 show the peak memory usage of the four competing methods over the whole update process under Jaccard similarity, where the numbers in parentheses are estimates, for the corresponding methods fail to complete the update process within a reasonable time. From the table, we can see that the space consumptions of *DynELM* and *pSCAN* are similar, and they are the most space efficient. *DynStrClu*

consumes about 10% to 20% more space than *DynELM*, because it needs to maintain the CC structures. Nonetheless, the space consumptions of all are linear in the graph size, consistent with the theoretical guarantees.

9.6 Efficiency on the Representative Datasets

Average Cost v.s. Updates. Next, we study the average update cost over the simulated update process, with varying insertion generation strategies under Jaccard similarity. That is for each of \mathcal{RR} , \mathcal{DR} and \mathcal{DD} , we assess the running time on the five representative datasets plus *Twitter* with all other parameters set to default values. During the update process, the average update cost at timestamp t is calculated as the average running time over all the first t updates. The results are shown in Figure 8. As explained earlier, *DynELM* is slightly faster than *DynStrClu*, leading to the curves of their average update costs over updates are very close to each other in a log-scale y-axis. For clarity, we omit the curve of *DynELM* from Figure 8.

For all the strategies on all the datasets, as expected, *DynStrClu* is the most efficient, followed by *pSCAN*, and then *hSCAN*. For a fixed dataset, the average update costs of all the three algorithms increase with the strategies in the order of \mathcal{RR} , \mathcal{DR} and \mathcal{DD} . This is because, the chance of a generated update incident on a large-degree vertex increases with the strategy changing from \mathcal{RR} , to \mathcal{DR} and then to \mathcal{DD} . Since both *pSCAN* and *hSCAN* need to scan the neighbourhoods of the vertices of the edge in an update, they have to pay more costs on those updates generated with \mathcal{DD} . This trend can be seen from Figure 8, and it also explains why *pSCAN* fails to complete the update sequence within three hours on all the five datasets under \mathcal{DD} . In contrast, as our methods guarantee a $O(\log^2 n)$ amortized update cost; and are somewhat immune to the insertion strategy change. Moreover, *pSCAN* fails to complete the first m_0 insertions within three hours on *Wiki* and *LiveJ*, and within 30 hours on *Twitter*. However, according to the first few data points plotted in the figures, *DynStrClu* already outperforms *pSCAN* by up to two-to-three orders of magnitude on *LiveJ*, *Wiki* and *Twitter*, respectively, with all three strategies. Not to mention *hSCAN*, which is even slower than *pSCAN*. Moreover, it is worth mentioning that even on the dataset with 1.2 billion edges, *Twitter*, our algorithm can still perform an update, in average, less than 100 micro-seconds. This clearly shows the scalability of our methods.

For cosine similarity, since *DynELM* and *DynStrClu* performs similarly, we choose *DynELM*, *pSCAN* and *hSCAN* as three competing methods. The results are shown in Figure 11. The curves are very like the curves shown in Figure 8. Among all three algorithms, *DynELM* is always the most efficient one on every representative datasets, which matches our expectation. Under cosine similarity, both *pSCAN* and *hSCAN* still need to scan the neighbourhoods of the effected vertices to process one single update. On *Wiki* and *LiveJ*, our algorithm has already outperformed *pSCAN* by two-to-three orders of magnitude after 3 hours. Since *hSCAN* takes more time in processing one update than *pSCAN*, the gap between it and our algorithm will be even huger.

Although from the theoretical analysis, the amortized cost under cosine similarity suffer from a roughly $O(\frac{1}{\epsilon})$ factor loss, the real performances of our algorithm under these two similarities are nearly identical. The reason is that, as mentioned above, the value of ϵ under cosine similarity is generally larger. Therefore, the theoretical performance of our algorithms under these two similarities won't vary much. This is an evidence that our algorithms are highly capable under different forms of structural similarity.

Varying Parameters under Default Setting. Figure 9 and Figure 10 show the overall running time on the five representatives, varying ϵ and η , respectively. Again, in both of these experiments, our methods are up to 1000 \times faster than the two competitors. With ϵ increasing, the overall running times of our algorithms drop slightly, while with η increasing, i.e., more deletions in the update process, both *DynELM* and *DynStrClu* have the running time increased.

However, in this case, the running times of *pSCAN* and *hSCAN* both decrease slightly: with more deletions, the degrees of the vertices tend to decrease. Finally, the last two bar charts show the experimental results on overall running time of *DynELM* v.s. ρ and cluster-group-by query time of *DynStrClu* v.s. query size, both on the five representative datasets. As for the former, *DynELM* is not as sensitive to ρ , as the theoretical complexity suggested, while for the latter, the query time of *DynStrClu* increases roughly linearly with the query size, consistent with the query time complexity.

10 CONCLUSION

In this paper, we study structural clustering (*StrClu*) graphs subject to edge insertions and deletions with the Jaccard similarity and cosine similarity. Given a failure probability δ^* , for every sequence of M updates, our algorithm *DynELM* can process each update in $O(\log^2 n + \log n \cdot \log \frac{M}{\delta^*})$ amortized time – a significant improvement on the state-of-the-art $O(n)$ bound under both similarities. Furthermore, its space consumption is linear in the current size of the graph, i.e., $O(n + m)$, at all times. Meanwhile, it guarantees that the *StrCluResult* can be retrieved in $O(n + m)$ time upon request and the clustering result is correct under the ρ -approximate notion with probability at least $1 - \delta^*$. Based on *DynELM*, our ultimate algorithm *DynStrClu* not only achieves all the above guarantees of *DynELM*, but also answers any cluster-group-by query of $Q \subseteq V$ in $O(|Q| \cdot \log n)$ time. We conducted extensive experiments on 15 real datasets including a billion-edge dataset *Twitter*. The experimental results confirm that both of our methods are up to 1000× faster than the state-of-the-art competitors in handling updates, while still produce quality clustering. We also study the difference between these two similarities in terms of approximation quality. We find that the quality of produced approximate clustering results are better under Jaccard similarity while the performances are similar. This may serve as a guidance when choosing proper similarity definition for real applications.

11 ACKNOWLEDGEMENT

In this work, Junhao Gan is supported by Australian Research Council (ARC) Discovery Early Career Researcher Award (DECRA) DE190101118, and Anthony Wirth is supported in part by ARC Discovery Project (DP) DP190102078.

REFERENCES

- [1] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. 2004. Correlation Clustering. *Machine Learning* 56, 1-3 (2004), 89–113.
- [2] Mathieu Bastian, Sébastien Heymann, and Mathieu Jacomy. 2009. Gephi: An Open Source Software for Exploring and Manipulating Networks. <http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154>
- [3] James C Bezdek and Nikhil R Pal. 1998. Some new indexes of cluster validity. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 28, 3 (1998), 301–315.
- [4] Andrei Z. Broder. 1997. On the resemblance and containment of documents. In *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings*. 21–29.
- [5] Lijun Chang, Wei Li, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. pSCAN: Fast and exact structural graph clustering. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 253–264.
- [6] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang, and Shiyu Yang. 2017. pSCAN: Fast and Exact Structural Graph Clustering. *IEEE Trans. Knowl. Data Eng.* 29, 2 (2017), 387–401.
- [7] Sudarshan S Chawathe. 2019. Clustering blockchain data. In *Clustering Methods for Big Data Analytics*. Springer, 43–72.
- [8] Graham Cormode, S. Muthukrishnan, and Ke Yi. 2011. Algorithms for Distributed Functional Monitoring. *ACM Trans. Algorithms* 7, 2, Article 21 (March 2011), 21:1–21:20 pages.
- [9] Chris H. Q. Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and Horst D. Simon. 2001. A Min-max Cut Algorithm for Graph Partitioning and Data Clustering. In *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA*. 107–114.
- [10] Yijun Ding, Minjun Chen, Zhichao Liu, Don Ding, Yanbin Ye, Min Zhang, Reagan Kelly, Li Guo, Zhenqiang Su, Stephen C Harris, et al. 2012. atBioNet—an integrated network analysis tool for genomics and biomarker discovery. *BMC genomics* 13, 1 (2012), 325.

- [11] Junhao Gan, David F. Gleich, Nate Veldt, Anthony Wirth, and Xin Zhang. 2019. Graph Clustering in All Parameter Regimes. *CoRR* abs/1910.06435 (2019). <http://arxiv.org/abs/1910.06435>
- [12] Junhao Gan and Yufei Tao. 2017. Dynamic Density Based Clustering. In *Proceedings of ACM Management of Data (SIGMOD)*. 1493–1507.
- [13] Junhao Gan and Yufei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 14:1–14:45.
- [14] Junhao Gan and Yufei Tao. 2018. Fast Euclidean OPTICS with Bounded Precision in Low Dimensional Space. In *Proceedings of ACM Management of Data (SIGMOD)*. 1067–1082.
- [15] David F. Gleich, Nate Veldt, and Anthony Wirth. 2018. Correlation Clustering Generalized. In *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16–19, 2018, Jiaoxi, Yilan, Taiwan*. 44:1–44:13.
- [16] Stephan Günnemann, Ines Färber, Brigitte Boden, and Thomas Seidl. 2014. GAMer: a synthesis of subspace clustering and dense subgraph mining. *Knowl. Inf. Syst.* 40, 2 (2014), 243–278.
- [17] Wassily Hoeffding. 1994. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*. Springer, 409–426.
- [18] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* 48, 4 (2001), 723–760.
- [19] Zengfeng Huang, Ke Yi, and Qin Zhang. 2019. Randomized Algorithms for Tracking Distributed Count, Frequencies, and Ranks. *Algorithmica* 81, 6 (2019), 2222–2243.
- [20] Lawrence Hubert and Phipps Arabie. 1985. Comparing partitions. *Journal of classification* 2, 1 (1985), 193–218.
- [21] Ram Keralapura, Graham Cormode, and Jeyashankher Ramamirtham. 2006. Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27–29, 2006*. 289–300.
- [22] Ludmila I Kuncheva, Stefan Todorov Hadjitodorov, and Ludmila P Todorova. 2006. Experimental comparison of cluster ensemble methods. In *2006 9th International Conference on Information Fusion*. IEEE, 1–7.
- [23] Ludmila I Kuncheva and Dmitry P Petrov. 2006. Evaluation of stability of k-means cluster ensembles with respect to random initialization. *IEEE transactions on pattern analysis and machine intelligence* 28, 11 (2006), 1798–1808.
- [24] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN*: Overlapping community detection using the link-space transformation. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 292–303.
- [25] Mutlu Mete, Fusheng Tang, Xiaowei Xu, and Nurcan Yuruk. 2008. A structural approach for finding functional modules from large biological networks. In *Bmc Bioinformatics*, Vol. 9. Springer, S19.
- [26] Glenn W Milligan and Martha C Cooper. 1986. A study of the comparability of external criteria for hierarchical cluster analysis. *Multivariate behavioral research* 21, 4 (1986), 441–458.
- [27] Stefano Monti, Pablo Tamayo, Jill Mesirov, and Todd Golub. 2003. Consensus clustering: a resampling-based method for class discovery and visualization of gene expression microarray data. *Machine learning* 52, 1-2 (2003), 91–118.
- [28] Flavia Moser, Recep Colak, Arash Rafiey, and Martin Ester. 2009. Mining Cohesive Patterns from Graphs with Feature Vectors. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA*. 593–604.
- [29] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the overlapping community structure of complex networks in nature and society. *nature* 435, 7043 (2005), 814.
- [30] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in social media. *Data Mining and Knowledge Discovery* 24, 3 (2012), 515–554.
- [31] S Papadopoulos, C Zikolos, Y Kompatsiaris, and A Vakali. 2010. Cluster-based landmark and event detection on tagged photo collections. *IEEE 99* (2010), 1–1.
- [32] Vishnu Sankar, Balaraman Ravindran, and S. Shivashankar. 2015. CEIL: A Scalable, Resolution Limit Free Approach for Detecting Communities in Large Networks. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015*. 2097–2103.
- [33] Satu Elisa Schaeffer. 2007. Graph clustering. *Computer science review* 1, 1 (2007), 27–64.
- [34] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-scale Graphs. *PVLDB* 8, 11 (2015), 1178–1189.
- [35] Mikkel Thorup. 2000. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21–23, 2000, Portland, OR, USA*. 343–350.
- [36] Nguyen Xuan Vinh, Julian Epps, and James Bailey. 2010. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research* 11 (2010), 2837–2854.
- [37] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. 2014. How to partition a billion-node graph. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 568–579.
- [38] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient Structural Graph Clustering: An Index-Based Approach. *PVLDB* 11, 3 (2017), 243–255.
- [39] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Efficient structural graph clustering: an index-based approach. *VLDB J.* 28, 3 (2019), 377–399.

- [40] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. SCAN: a structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, California, USA, August 12-15, 2007*. 824–833.
- [41] Nurcan Yuruk, Xiaowei Xu, and Thomas AJ Schweiger. 2008. On Structural Analysis of Large Networks. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. IEEE, 143–143.
- [42] Nurcan Yuruk, Xiaowei Xu, and Thomas A. J. Schweiger. 2008. On Structural Analysis of Large Networks. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*. 143.