BOTBIN: Accelerated Indexing for Structural Graph Clustering on Dynamic Graphs

Fangyuan Zhang, Qintian Guo, Junhao Gan, and Sibo Wang

Abstract—Graph clustering is a fundamental data mining task that clusters vertices into different groups. The structural graph clustering algorithm ($SCAN$) is a widely used graph clustering algorithm that derives not only clustering results, but also special roles of vertices like hubs and outliers. In this paper, we consider structural graph clustering on dynamic graphs under Jaccard similarity. The state-of-the-art index-based solution focuses on static graphs and incurs prohibitive update costs to maintain indices. Lately, an efficient approximate dynamic structural graph clustering algorithm DynStrClu under Jaccard similarity is proposed. However, their solution needs to fix input parameters while parameter settings of SCAN usually need to be fine-tuned to achieve good clustering results. Motivated by these limitations, we present a study on devising effective index structures for SCAN algorithm on dynamic graphs. Similar to the state-of-the-art dynamic scheme, our main idea to reduce the time complexity is still by bringing approximation to clustering results. However, our solution does not need to fix the input parameters. To achieve this, our solution includes two key components. The first is to maintain a bottom-k sketch for each vertex so that the similarities of affected vertices can be easily updated. The second key is a bucketing strategy that allows us to update clustering results and roles of vertices efficiently. Our theoretical analysis shows that our proposed algorithm achieves $O(\log\log\frac{M+m}{p_f}\cdot\log\frac{M+m}{p_f})$ expected update cost and guarantees to return approximate clustering results with probability $1-p_f$ after up to M updates. Extensive experiments show that our solution is up to two orders of magnitude faster than the state-of-the-art index-based solution while still achieving high-quality clustering results.

Index Terms—Graph algorithms, graphs and networks.

1 Introduction

Graph is a fundamental type of data structure to represent objects and their relationships. Much real-world data can be represented as graphs such as online social networks, the Web network, protein interaction networks, and road networks. Graph clustering is one of the most important tasks on graph data analysis, which aims to divide the vertices into different groups where vertices within a group are densely connected while vertices in different groups are sparsely connected. Graph clustering finds many real-world applications, such as finding functional modules in metabolic networks [1], identifying communities on social networks [2], detecting research groups across collaborating networks [3], to name a few.

Structural graph clustering [4] is one of the well-known approaches to graph clustering and Xu et al. [4] present the first algorithm SCAN to solve this problem. The main idea of SCAN is that if two vertices are similar enough in terms of their neighborhood, then the two vertices tend to belong to the same cluster. Unlike other graph clustering algorithms that only find the clustering results, SCAN further finds the special roles of different vertices. For instance, it identifies hub vertices that connect different clusters and outliers that lack strong ties to any cluster. Given an undirected graph $G=\langle V,E\rangle$ with n vertices and m edges and a set similarity measure σ, e.g., the Jaccard similarity, the SCAN algorithm takes two input parameters: a similarity threshold ϵ and an integer μ. If v is a neighbor of vertex u (i.e., there exists an edge between u and v) and they have a similarity score $\sigma(u,v)$ no smaller than ϵ, then v is called an *ϵ-neighbor* of vertex u. If the number of ϵ-neighbors of vertex u is no smaller than μ, u is called a *core vertex*. Then, considering only the edges among core vertices with similarity above the threshold ϵ, we can construct different connected components, resulting in different clusters. For a vertex v that is not a core vertex but an ϵ-neighbor of core vertex(s), it is assigned to the cluster(s) of its ϵ-neighbor(s) that is/are core vertex(s). *Hub vertices* and *outliers* are further defined for unclustered vertices, depending on whether the vertex connects to multiple clusters. Different roles enrich the structural information and help us understand networks.

It is shown that SCAN can help find meaningful clustering results in biological data [5], [6], [7], [8], web data [9], [10], [11], [12], [13], [14], and social networks [15], [16]. It can also be used for community detection [16], image segmentation [17], and fraud detection on blockchain data [18]. Due to its important applications, it attracts a plethora of research studies, e.g., [19], [20], [21], on improving the efficiency of the SCAN algorithm. As the SCAN algorithm needs to calculate the similarity among all neighbor vertex pairs, some existing research focuses on reducing the costs of similarity computations via pruning unnecessary pairs [19], [21], [22] or exploiting parallelism [22], [23], [24], [25], [26], [27], [28]. In the meantime, it is also pinpointed in existing research studies [29], [30], [31] that input parameters ϵ and μ need to be fine-tuned so as to achieve good clustering results. Thus,

- *Qintian Guo is the corresponding author.*
- *Fangyuan Zhang and Sibo Wang are with The Chinese University of Hong Kong, Hong Kong SAR. Emails: {fzhang, swang}@se.cuhk.edu.hk.*
- *Qintian Guo is with The Hong Kong University of Science and Technology, Hong Kong SAR. Email: qtguo@ust.hk.*
- *Junhao Gan is with The University of Melbourne, Australia. Email: junhao.gan@unimelb.edu.au.*

Manuscript received XXXX XX, XXXX; revised XXXX XX, XXXX.

another direction is to devise effective indices, e.g., GS-Index [20], [29], so that clustering results and different roles of vertices can be more efficiently derived for different choices of ϵ and μ. Nevertheless, all these research works mainly consider static graphs while real-life graphs, like web graphs and social graphs, are dynamically changing. It is possible to compute the exact GS-Index [29] from scratch with multi-core parallelization [20]. However, the $O(m^{1.5})$ construction cost is still prohibitive on huge graphs, resulting in long waiting time for clustering analysis.

Most recently, Ruan et al. [32] present a dynamic scheme *DynStrClu* for SCAN with the popular Jaccard similarity as the underlying set similarity measure. The expected update cost of DynStrClu is $O\left(\log^2 n + \log n \cdot \log \frac{M+m}{p_f}\right)$, which guarantees that the returned clustering result satisfies the approximation definition with probability $1 - p_f$ after up to M updates. However, the main issue of DynStrClu is that it needs to fix the two input parameters and only maintains the clustering result for such a fixed set of input parameters. It is difficult for DynStrClu to help us find good clustering results when inputting different parameters ϵ and μ. If we simply apply DynStrClu with various settings of ϵ and μ, the space consumption will immediately explode as there are too many such combinations.

Motivated by limitations of existing solutions, we present an effective indexing scheme *BOTBIN (Bottom-k and bucket indexing)* for the SCAN algorithm on dynamic graphs under Jaccard similarity that are insensitive to the input parameters. Following the previous work [32], we consider the case when the updates are edge insertions/deletions as vertex insertion/deletions can be mapped to a series of edge insertions/deletions. Following [32], we consider unweighted graphs and focus on the Jaccard similarity, which is a widely adopted similarity for unweighted sets. As shown in [29], deriving an exact dynamic scheme to SCAN incurs $O(\min\{d_{max}^2, m\} \cdot \log n)$ update cost. To reduce the time complexity, we turn to approximate solutions, which relax the definition of ϵ-neighbors, and show that we can update the index with an expected cost of $O(\log n \cdot \log \frac{M+m}{p_f})$ for each edge update. The returned clustering result satisfies the approximation definition with probability $1-p_f$ after up to M edge updates. Note that this result is non-trivial as our BOTBIN reduces a term of $\log^2 n$ in update cost compared to the state-of-the-art dynamic scheme DynStrClu, while at the same time it supports different choices of μ and ϵ.

To achieve above goals, our solution includes two key components. The first is a similarity index that can support efficient maintenance of similarity scores on dynamic graphs. Our similarity index includes a bottom-k sketch for each vertex so that we can derive the estimated similarities with approximation guarantees between neighboring vertices by inspecting the sketches of two vertices. We note that using bottom-k sketch to estimate Jaccard similarity is not a new idea. Our key contribution here is to dynamically maintain the bottom-k sketch after graph updates with bounded cost. The key observation is that the larger the number of neighborhood a vertex u has, the smaller the chance is that an edge update to (u, v) will affect the bottom-k sketch of vertex u. To the best of our knowledge, we are the first to make full use of this observation and combine it with the bottom-k sketch to reduce the time complexity of dynamic graph clustering algorithms. It may further inspire other research works to improve the time complexity of graph clustering or dynamic graph algorithms with similar issues. We show that the update cost can be bounded with $O(k^2)$ where $k = O(\log \frac{M+m}{p_f})$ is the size of the bottom-k sketch of each vertex and is independent of the degree of vertex u. Further observing that the update of an edge (u, v) affects at most two entries of the bottom-k sketch of vertex u. Subsequently, we present optimization techniques to reduce the update cost to $O(k \log k)$.

The second key component of our BOTBIN is an effective clustering index that helps return the clustering result given the input parameters ϵ and μ. Notice that even if we can efficiently maintain and update the similarity scores with a cost independent of the degree, it may still incur high update cost if we want to report the clustering results with a cost proportional to the size of the cluster subgraph (Ref. to Definition 6). For example, with the state-of-the-art GS-Index, even if we only change the similarity score of one vertex pair, it still incurs up to $O(d_{max} \cdot \log n)$ update cost (reasons explained in Section 2.2), where d_{max} is the maximum degree of the graph. To tackle this issue, the second key of BOTBIN is a bucketing strategy that maintains different buckets where each bucket is associated with a range and maintains a list of edges whose incident vertex pairs have a similarity score falling inside the associated range. When similarity scores are updated, it only needs to update edges to the correct bucket, which significantly reduces the maintaining overhead. We further present a detailed theoretical analysis and show that our index scheme can achieve an update cost of $O(\log n \cdot \log \frac{M+m}{p_f})$, which is independent of d_{max}.

This manuscript is an extended version of our previous conference paper [33]. The main differences between this manuscript and the conference version are outlined as follows: First, in addition to Jaccard similarity, we extend our similarity estimation scheme to support Dice similarity (see Section 3.3), and provide experimental evaluations on real datasets to validate its effectiveness. Second, we propose an optimized solution for updating the bucket index. Our solution reduces the update time for each bucket to $O(1)$, improving upon the $O(\log n)$ time complexity in the conference version (see Section 4.3). This further helped us to arrive at a solution with a better theoretical complexity for edge insertion/deletion. Note that in our conference version, the expected running cost of BOTBIN is $O(\log n \cdot \log \frac{M+m}{p_f})$ for each edge update. With the O(1) cost for bucket index update, we reach the final (expected) time complexity of $O(\log \log \frac{M+m}{p_f} \cdot \log \frac{M+m}{p_f})$. Experimental results confirm with this theoretical improvement.

To summarize, our main contributions in this paper are listed as follows.

- We propose an index-based scheme, BOTBIN, for structural graph clustering, which consists of a similarity index and a clustering index (see Section 3). It is designed to efficiently return clustering results given different input parameters μ and ϵ.
- For dynamic graphs, we present update algorithms for the BOTBIN scheme to handle cases of edge insertion or

deletion (see Section 4.1).
- To further improve the efficiency of the BOTBIN solution on dynamic graphs, we propose two optimization techniques (see Sections 4.2–4.3). Theoretical analysis shows that the optimized BOTBIN achieves an expected update cost of $O(\log\log\frac{M+m}{p_f}\cdot\log\frac{M+m}{p_f})$ and guarantees to return approximate clustering results with probability $1-p_f$ after up to M updates.
- Extensive experiments on real-world graphs with up to 3.9 billion edges show that our proposed scheme BOTBIN achieves up to 3 orders of magnitude improvement over the existing state-of-the-art solution without compromising the clustering quality.

2 PRELIMINARIES

2.1 Problem Definition

Notations in graphs. We consider an unweighted and undirected graph $G=(V,E)$, where V is the set of vertices and E is the set of edges in the graph. We use n to indicate the number of vertices and m to indicate the number of edges. An edge e connecting vertex u and v is denoted as an unordered pair of vertices, (u,v) or (v,u), and u and v are called a neighbor of each other or simply neighboring vertices. For a vertex $u\in V$, the *neighborhood* $N[u]$ consists of *(i)* the set of nodes who are neighbors of node u and *(ii)* u itself. The *degree* d_u of vertex u is defined as the number of neighbors of vertex u.

Terminologies in SCAN. Given two neighboring vertices u and v, the similarity $\sigma(u,v)$ between u and v is defined as the set similarity between $N[u]$ and $N[v]$. We focus on the case when the elements in the set are equally weighted. In such a scenario, Jaccard similarity is adopted as the set similarity measure in existing studies, e.g., [19], [20], [32], [34]. The definition of Jaccard similarity is as follows.

Definition 1 (Jaccard Similarity)**.** *Given two sets X and Y, the Jaccard similarity between these two sets is defined as $\frac{|X\cap Y|}{|X\cup Y|}$.*

In other words, given two neighboring vertices u and v, their similarity $\sigma(u,v)=|N[u]\cap N[v]|/|N[u]\cup N[v]|$. If u and v are not neighbors, their similarity is 0. In SCAN, the first input parameter is a threshold ϵ, which identifies the ϵ-neighbors, defined as follows.

Definition 2 (ϵ-neighbor)**.** *Given a threshold $\epsilon\in(0,1]$ of the similarity, v is called an ϵ-neighbor of u if $v\in N[u]$ and the similarity $\sigma(u,v)$ between u and v is no smaller than ϵ. The ϵ-neighbor set $N_\epsilon[u]$ is defined as the set of ϵ-neighbors of node u, i.e., $N_\epsilon[u]=\{v\in N[u]|\sigma(u,v)\geq\epsilon\}$.*

Given the definition of ϵ-neighbors, the second input parameter is an integer threshold $\mu\geq 2$, which is used to define *core vertices*.

Definition 3 (core vertex)**.** *A vertex $u\in V$ is a core vertex if u has at least a number μ of ϵ-neighbors, i.e., $|N_\epsilon[u]|\geq\mu$.*

Definition 4 (Core graph)**.** *Given the set V_{core} of core vertices, the core graph G_{core} consists of V_{core} and E_{core}, where E_{core} includes an edge (u,v) if and only if* (i) $(u,v)\in E$, (ii) $u,v\in V_{core}$, (iii) *u and v are ϵ-neighbors.*

Definition 5 (cluster)**.** *Each connected component of G_{core} corresponds to a cluster. For a vertex v that is not a core vertex, if it is an ϵ-neighbor of a core vertex u, then it belongs to the cluster containing u. Notice that the non-core vertex v might be the ϵ-neighbor of multiple core vertices, then the vertex v will be assigned to multiple clusters.*

Definition 6 (Cluster Subgraph)**.** *Given a clustering result $\mathcal{C}$, the cluster subgraph is a subgraph $G_\mathcal{C}=(V_\mathcal{C},E_\mathcal{C})$ of the input graph G where $v\in V_\mathcal{C}$ iff* (i) *v is a core vertex or ϵ-neighbor of a core vertex;* (ii) *an edge $(u,v)\in E_\mathcal{C}$ iff $u,v\in V_\mathcal{C}$ and at least one is a core vertex.*

Definition 7 (Hubs and outliers)**.** *For a vertex v that is not in any clusters, if it has neighbors belonging to multiple clusters, then it is a* hub vertex*. Otherwise, it is an* outlier*.*

Example 1. Figure 1 shows the clustering result of the input graph G_1 when $\epsilon=0.5$ and $\mu=5$. The similarity for every pair of adjacent vertices is shown close to each edge. Vertices v_3 and v_8 are core vertices, which are filled with black colors. The core-graph then consists of only two isolated vertices v_3 and v_8. The clustering result $\mathcal{C}=\{C_1,C_2\}$ consists of two clusters where $C_1=\{v_1,v_2,v_3,v_4,v_5\}$ and $C_2=\{v_6,v_7,v_8,v_9,v_{10}\}$, as v_1,v_2,v_4,v_5 are the ϵ-neighbors of v_1 and v_6,v_7,v_8,v_9 are the ϵ-neighbors of v_8. The cluster subgraph $G_\mathcal{C}=(V_\mathcal{C},E_\mathcal{C})$ then includes v_1 to v_{10}, i.e., $V_\mathcal{C}=\{v_1,v_2,\cdots,v_{10}\}$ as they are either core vertices or the ϵ-neighbors of core vertices. The edge set $E_\mathcal{C}=\{(v_1,v_3),(v_2,v_3),(v_4,v_3),(v_5,v_3),(v_6,v_8),(v_7,v_8),(v_9,v_8),(v_{10},v_8)\}$ as at least one of the vertices in an edge must include a core vertex and both vertices of an edge must belong to $V_\mathcal{C}$. There are four un-clustered vertices where v_{11} is a hub vertex as it has neighbors in different clusters, while vertices v_{12}, v_{13}, and v_{14} are outliers according to the definition. □

As mentioned in Section 1, it is important to tune the two input parameters to find good clustering results. Therefore, some existing studies, e.g. [29], present effective index structures which output the clustering results with a cost proportional to the size of result subgraphs. However, such solutions cannot handle index updates efficiently on dynamic graphs. To tackle this issue, we will bring ρ-approximation into the clustering results where ρ is used to balance the trade-off between the update efficiency and clustering quality.

ρ-approximate SCAN. Following previous work [20], we consider ρ-approximate SCAN where we relax the definition of ϵ-neighbor. Then, we define the (ϵ,ρ)-neighbor as follows.

Definition 8 ((ϵ,ρ)-neighbor)**.** *Given an error parameter ρ and two vertices u and v,*

- *if $\sigma(u,v)>\epsilon+\rho$, u must be an (ϵ,ρ)-neighbor of node v;*
- *if $\sigma(u,v)<\epsilon-\rho$, u must not be an (ϵ,ρ)-neighbor of node v;*
- *otherwise, it does not matter. In other words, u is allowed to be either an (ϵ,ρ)-neighbor of v or not.*

Notice that, the definition here uses the ρ-absolute error. However, it is easy to convert it to ρ'-relative error by setting $\rho=\epsilon\cdot\rho'$. In ρ-approximate SCAN, the core vertices are defined according to the number of (ϵ,ρ)-neighbors. It is worth noting that the clustering results in ρ-approximate

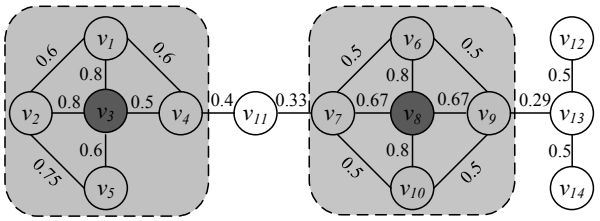


Fig. 1: Clustering result with $\epsilon = 0.5, \mu = 5$ for graph G_1.

SCAN are not unique due to the uncertainty of the "does not matter" cases in the definition of (ϵ, ρ)-neighbors. However, when ρ is small enough, the ϵ-neighbors and (ϵ, ρ)-neighbors of the same node usually do not differ much. Moreover, assume that we have the clustering result $\mathcal{C}_{\epsilon+\rho,\mu}$ of SCAN with similarity threshold to be $\epsilon + \rho$ and the clustering result $\mathcal{C}_{\epsilon-\rho,\mu}$ with similarity threshold to be $\epsilon - \rho$. Then the ρ-approximate SCAN clustering results, dubbed as $\mathcal{C}^{\rho}_{\epsilon,\mu}$, have the following guarantee.

Theorem 1 (Guarantee of ρ-approximate SCAN)**.** *For any clustering result $\mathcal{C}^{\rho}_{\epsilon,\mu}$ outputted by ρ-approximate SCAN, we have:* (i) *For every cluster $C_{\epsilon+\rho} \in \mathcal{C}_{\epsilon+\rho,\mu}$, there exists a cluster $C \in \mathcal{C}^{\rho}_{\epsilon,\mu}$ such that $C_{\epsilon+\rho} \subseteq C$;* (ii) *For every cluster $C \in \mathcal{C}^{\rho}_{\epsilon,\mu}$, there exists a cluster $C_{\epsilon-\rho} \in \mathcal{C}_{\epsilon-\rho,\mu}$ such that $C \subseteq C_{\epsilon-\rho}$.*

The proof of the above theorem is omitted as it directly follows from the definitions of $\mathcal{C}_{\epsilon+\rho,\mu}$, $\mathcal{C}_{\epsilon-\rho,\mu}$, and $\mathcal{C}^{\rho}_{\epsilon,\mu}$.

SCAN on dynamic graphs. We consider dynamic graphs where the input graph is updated with edge insertions/deletions. Note that node insertions/deletions can be converted to a series of edge insertions/deletions. In this paper, we assume that the sequence of edge updates is uniformly random [35] for ease of analysis.

In dynamic graph setting, we aim to maintain an indexing scheme so that we can return the clustering result with a cost proportional to the size of the cluster subgraph. However, even with the state-of-the-art indexing scheme for the exact solution, it still takes $O(\min\{d^2_{max}, m\} \cdot \log n)$ update cost for each edge insertion/deletion. Thus, we focus on maintaining effective index structures for ρ-approximate SCAN, defined as follows.

Problem definition. Given a number $M = O(m)$ of edge updates, the dynamic indexing scheme aims to maintain an effective index with $O(polylog(n))$ cost for each edge update and can output the clustering result (with a cost bounded by the size of the cluster subgraph) satisfying the definition of ρ-approximate SCAN with probability $1 - p_f$, where p_f is used to bound the failure probability.

2.2 Existing Solutions Revisited

SCAN and pSCAN. The problem of structural graph clustering is first presented in [4], where the authors propose the SCAN algorithm to solve this problem. The main idea of SCAN is to calculate the similarity $\sigma(u, v)$ of each edge(u, v) and then scan all vertices in G to obtain the correct clustering results for the given input parameters ϵ and μ. The time complexity of the SCAN algorithm is $O(\alpha m)$, where α is the arboricity of graph G, which is also equal to the minimum number of forests that needs to cover all edges of graph G. The worst case running time is $O(m^{1.5})$.

The major cost of SCAN is the computation of set similarities between neighbors. Chang et al. [19] propose *pSCAN* to reduce the cost of calculating the set similarity between neighbors with pruning techniques on identifying ϵ-neighbors and core vertices. With these pruning strategies, pSCAN returns exact results much faster than SCAN, but still has $O(m^{1.5})$ worst-case running time.

GS-Index. The above solutions focus on online calculations for all clusters given the two input parameters ϵ and μ. However, as we mentioned, the two input parameter settings of SCAN usually need to be fine-tuned to achieve good clustering results. Thus, we may need to process SCAN multiple times with different input parameters. It is observed that when we fix one parameter and decrease the other parameter, the new clustering result must be included in the previous clustering result. Based on this observation, Wen et al. [29] propose GS-Index, an index-based method that quickly returns query results by maintaining a *neighbor-order* and a *core-order* index. In the neighbor-order index, each node v maintains a list $L[v]$ of neighbors sorted in decreasing order of the similarity score. Clearly, the neighbor-order index takes $O(m + n)$ space cost. By maintaining the neighbor-order index, given an arbitrary similarity threshold ϵ, each node can quickly identify ϵ-neighbors by scanning $L[v]$ and stop as soon as it reaches a neighboring node with similarity score smaller than ϵ. For the core-order index, it maintains d_{max} ordered set $C[2], \cdots, C[d_{max} + 1]$ where d_{max} is the maximum degree in the input graph. For each vertex v, it appears in $C[2], C[3], \cdots, C[d_v + 1]$. More specifically, a pair (v, ϵ_i) is in ordered set $C[i]$ where $2 \leq i \leq d_v + 1$ and ϵ_i is the largest threshold such that v has i ϵ-neighbors. This can be derived by computing the similarity scores between v and vertices in its neighborhood $N[v]$ and then taking the i-th largest similarity score. The pairs in the ordered set $C[i]$ are then sorted in descending order of their ϵ_i thresholds. It is easy to verify that the core-order index still takes $O(m + n)$ space. Given the GS-Index and input parameters ϵ and μ, it outputs clusters as follows. Firstly, it goes through the ordered set $C[\mu]$ in the core-order index and returns the set of vertices whose ϵ_μ score is no smaller than ϵ. These vertices are the core vertices. Then, it scans the neighbor-order index of each core vertex, and includes all neighboring vertices with a similarity score no smaller than ϵ. This can be bounded with a running time proportional to the size of the cluster subgraph.

GS-Index also supports dynamic updates. Given an edge update to (u, v), it incurs an update cost of $O\left(|E_{2hop}(u)| + |E_{2hop}(v)|\right)$, where $E_{2hop}(x)$ is the set of edges that both ends of vertices in the edge belong to the vertex set $V_{2hop}(x)$, consisting of vertices with distance to x no larger than 2. In the worst case, this incurs $O(\min\{d^2_{max}, m\} \cdot \log n)$ running cost for each update. On one hand, it needs to handle the updates of the similarity scores due to the insertion/deletion of an edge (u, v), which affects the similarity score of up to $d_u + d_v$ neighboring vertex pairs. This incurs $O(d_{max})$ update cost if it takes $O(1)$ cost to check if u (resp. v) exists in the neighbor of vertex w, where w is a neighbor of v (resp. u). On the other hand, it needs to update the neighbor-order and core-order to support efficient clustering for arbitrary input parame-

ters. When the similarity of u and one of its neighbors is changed due to the insertion of an edge, it may change up to $O(d_u)$ ordered sets in the core-order index. There are d_{max} updates to each ordered set in the core-index in the worst case. To support efficient update to the ordered sets of the core-index, a binary search tree is maintained [29]. This incurs $O(d_{max} \cdot \log n)$ running cost to update the core-index for each affected similarity score. Thus, GS-Index takes $O(\min\{d_{max}^2, m\} \cdot \log n)$ running cost to update the index in the worst case.

DynStrClu. The dynamic property of real-world graph data brings new challenges to structural graph clustering. To reduce the update cost, Ruan et al. [32] propose *DynStrClu*, an approximate solution that relax the definition of ϵ-neighbors to (ϵ, ρ)-neighbors (Ref. to Definition 8) and reduce the amortized update cost to $O\left(\log^2 n + \log n \cdot \log \frac{M+m}{p_f}\right)$ for each update. It guarantees that clustering results fulfill the approximation definition with a probability of $1 - p_f$ after up to M updates. DynStrClu considers the case that two input parameters ϵ and μ are fixed. When an edge (u, v) is updated, we only need to check if *(i)* u and v are (ϵ, ρ)-neighbors, *(ii)* if the (ϵ, ρ)-neighbor relationships between the neighbors of u (resp. v) and u (resp. v) have changed. The impact of an insertion/deletion of one edge on the similarity $\sigma(u, v)$ of other edges can be limited to $1/d_{max}(u, v)$, where $d_{max}(u, v) = max\{d_u, d_v\}$. Combining with the definition of (ϵ, ρ)-neighbors, DynStrClu includes a lazy update mechanism that can afford multiple updates for each edge as long as their (ϵ, ρ)-neighbor relationship does not change. When the (ϵ, ρ)-neighbor relationship of an edge (u, v) is unclear given the bounds, DynStrClu recalculates the similarity $\sigma(u, v)$ with a sampling-based method. Combining both techniques, it achieves the improved time complexity. However, DynStrClu can only maintain clustering results under fixed parameters of ϵ and μ, but cannot return clustering results under new input parameters.

3 Our Solution

Next, we present our proposed index scheme *BOTBIN* in details. There are two main challenges in maintaining an effective index structure to support dynamic (approximate) structural graph clustering. The first main challenge is the insertion/deletion of an edge (u, v) will result in the change of similarity scores of up to $d_u + d_v$ neighbor pairs. To tackle this issue, we maintain a bottom-k sketch for each vertex. The rough idea is that we generate a random number for each node, and then for each vertex u, we maintain the k min values of the neighbors of vertex u. We then use these bottom-k sketches of two vertices u and v to calculate the approximate similarity score. The advantage of our design is that, an edge insertion/deletion will affect the bottom-k sketch with bounded probability. We show that our solution can update the affected similarity scores with $O(k \cdot \log k)$ expected running time, where $k = O(\frac{1}{\rho^2} \cdot \log \frac{m+M}{p_f})$, independent of the degree of u and v.

Another challenge is to maintain an effective index structure on dynamic graphs so that it can return the clustering results with a cost proportional to the size of the cluster subgraph. Recap that given an updated similarity score, GS-Index will result in the change of all the ordered sets in the core-index in the worst case. To tackle this issue, we present the bucket index, which is built with a bucketing approach which maintains δ evenly split buckets. The discussion of δ will be elaborated more in Section 3.1. In bucket i $(1 \leq i \leq \delta)$, it maintains an ordered set $B[i]$ which consists of a set of pairs $(u, |N_{1-i/\delta,\rho}[u]|)$, where $N_{1-i/\delta,\rho}[u]$ is the set of (ϵ, ρ)-neighbors of u when $\epsilon = 1 - i/\delta$. In each ordered set $B[i]$, it is sorted based on the number of (ϵ, ρ)-neighbors of each node when $\epsilon = 1 - i/\delta$. Then, given an updated similarity score, it will affect at most $O(\delta)$ ordered list, which can be regarded as a constant that is independent of the graph parameter. Moreover, given an edge update (u, v), we can further bound the probability that the similarity score between u and a neighbor of u gets affected. This together reduces the expected update cost of our bucketing index to $O(\log n \cdot \log \frac{m+M}{p_f})$.

Next, we first present our BOTBIN index scheme in Section 3.1. Then, we show how to build the index and handle query processing with BOTBIN in Section 3.2. We will present how to update the index when the graph dynamically changes in Section 4.

3.1 Index Scheme

Our BOTBIN includes two indices: the *similarity index* and the *clustering index*. The similarity index is mainly used to derive and maintain the Jaccard similarity of adjacent vertices efficiently by exploiting the bottom-k sketch. The clustering index is mainly used to maintain the core vertices and ϵ-neighbors of core vertices efficiently with the bucketing idea for an arbitrary ϵ and μ.

Similarity index. Deriving the approximate similarity score for adjacent vertex pairs are well studied in the literature. For example, Tseng et al. [20] propose to explore k-min hash to estimate the Jaccard similarity score between adjacent vertex pairs. In particular, for every vertex $v \in G$, it first generate a random number $r_v \in (0, 1)$ as the hash value of v. Then, for vertex v, it maintains the min hash value $r_{min}(v)$ among $N[v]$, the neighborhood of vertex v. Then, the Jaccard similarity $\sigma(u, v)$ between adjacent vertices u and v is:

$$\sigma(u, v) = \Pr[r_{min}(u) = r_{min}(v)].$$

Let X be an indicator random variable where it takes 1 if $r_{min}(u) = r_{min}(v)$ and 0 otherwise. Then, $E[X] = \sigma(u, v)$. With this unbiased estimator, we may derive multiple instances of X by sampling k different hash values for each vertex and generate a variable X_i according to the i-th hash value of each vertex. It is shown in [20] that by setting $k = O(\log n/\rho^2)$, it can provide ρ-absolute error to the similarity score. However, a main deficiency of the k-min hash-based solution is that it takes $O(n \cdot k)$ space while k is usually much larger than the average degree of the input graph. For example, consider a social network Orkut with 3 million nodes and 0.2 billion edges. To provide an absolute error of 0.05 to the estimation of the similarity score, it needs to set $k \approx 6000$ and every vertex needs to maintain k min-hash values. This incurs an additional 90 times space (of the input graph), which is too expensive for large graphs.

To tackle this issue, we adopt the bottom-k sketch [36], rather than the popular k-min hash-based solution, to estimate the Jaccard similarity scores. The advantage of the

bottom-k sketch is that, instead of maintaining k values for each vertex v, it only needs to maintain $\min\{k, d_v + 1\}$ values. This bounds the space consumption of the bottom-k sketch to be the same as that of the input graph, saving far more space than the k-min hash-based solution. Next, we elaborate more details on our bottom-k sketch-based similarity index. We first define the bottom-k sketch as:

Definition 9 (Bottom-k sketch)**.** *Let U be the universe of the elements that may appear in a set S. Let $h : U \to [1 \cdots |U|]$ be a hash function that is a random permutation of U. The bottom-k sketch of S is the set of k smallest hash values among elements in S. If $k > |S|$, it is the set of hash values of all elements in S.*

In our problem, given a vertex v, the set $S = N[v]$ is the neighborhood of node v and the universe U is equal to V, the entire vertex set in the input graph G. The similarity index of a node v thus consists of the bottom-k sketch of its neighborhood $N[v]$. As shown in the following lemma, the bottom-k sketch can be further used as an unbiased estimator of the Jaccard similarity. Notice that here we assume that U is sufficiently large and thus we generate enough random numbers at the beginning for vertices.

Lemma 1 (Bottom-k Estimator of Jaccard similarity)**.** *Given two sets $A \subseteq U$ and $B \subseteq U$, let $S_k(A)$ and $S_k(B)$ denote their bottom-k sketches, respectively. If $|A| \geq k$ or $|B| \geq k$, then,*

$$\hat{J}(A, B) = \frac{S_k(A) \cap S_k(B) \cap S_k(A \cup B)}{k} \tag{1}$$

serves as an unbiased estimator of the Jaccard similarity $J(A, B)$ of set A and set B.

Here we restrict that either A or B has a size no smaller than k to use Equation 1 to estimate the Jaccard similarity. Otherwise, we can directly compute the Jaccard similarity between A and B with $O(k)$ cost. Although there has been a lot of research work on bottom-k sketch, as far as we know [36], [37], there is no explicit bound for the bottom-k sketch. Dahlgaard et al. [37] focus on the case when the set has no more than k elements and show how to add more entries until there are k elements in the sketch to provide approximation guarantee. In such a scenario, they will face a similar issue as that in k-min hash. For self-completeness, we present the approximation bound with bottom-k sketch and give a formal proof. In k-min hash sketch, it can be derived by sampling with replacement and thus different hash values are independent of each other. Therefore, the estimation can directly apply the Hoeffding inequality to derive a bounded error. However, with the bottom-k sketch, it is derived via sampling without replacement and thus existing concentration bound cannot be directly applied to derive a bounded error. Our proof of Theorem 2 makes a careful connection of the bottom-k sketch problem to the martingale process.

Theorem 2. *Given two sets A and B from a universe U, let $S_k(A)$ and $S_k(B)$ denote their bottom-k sketches, respectively. Given a failure probability p and an error parameter ρ, by setting $k = \frac{1}{2\rho^2} \ln \frac{2}{p}$, we have that $\Pr[|\hat{J}(A, B) - J(A, B)| > \rho] \leq p$.*

Omitted proofs can be found in our technical report [38]. Given Theorem 2, our similarity index is constructed as follows. Firstly, we generate a random permutation h of V. Then, according to h, the similarity index $S_k(v)$ of a vertex v consists of the bottom-k sketch of set $N[v]$. The choice of k is determinate according to Theorem 2 and more discussion will be given. The time complexity to build the similarity index can be bounded with $O(m + n)$. The space cost can be further bounded by $O(\min\{k \cdot n, n + m\})$ as each vertex maintains $\min\{k, 1 + d_v\}$ hash values.

Example 2. Still consider the graph G_1 in Figure 1. Assume that the random permutation h is $\{9, 14, 8, 2, 4, 1, 7, 5, 3, 6, 12, 13, 11, 10\}$. In other words, $h(v_1) = 9, h(v_2) = 14, h(v_3) = 8$, etc. Assume that $k = 3$. Then, the similarity index $S_k(v_3)$ of v_3 is $\{2, 4, 8\}$, since $h(v_1), h(v_2), h(v_3), h(v_4), h(v_5)$ are 9, 14, 8, 2, 4, respectively, and 2, 4, and 8 are the 3 smallest hash values. Similarly, the similarity index $S_k(v_2)$ of v_2 is $\{4, 8, 9\}$. Then the estimated similarity $\hat{\sigma}(v_2, v_3)$ between vertex v_2 and vertex v_3 is $2/3$ according to Equation 1.

Clustering index. Next, we elaborate on the details of the clustering index of the proposed BOTBIN. The clustering index is mainly used to quickly return the ρ-approximate SCAN (Section 2.1) given the input ϵ and μ. Note that our clustering index is inspired by the GS-Index. However, we tackle the challenging issue with dynamic update on GS-Index as pinpointed at the beginning of Section 3.

The key observation is that given an arbitrary input ϵ, if a vertex u is the (ϵ, ρ)-neighbor (Ref. to Definition 8) of vertex v, given a smaller input $\epsilon' < \epsilon$, then vertex u is still the (ϵ, ρ)-neighbor of vertex v. Thus, to find the (ϵ, ρ)-neighbors of each vertex v efficiently, for each vertex v, BOTBIN maintains a sorted set $NO[v]$, dubbed as the *neighboring index*, of its neighborhood $N[v]$ in the form of pairs $\langle v, \hat{\sigma}(u, v)\rangle$, and the pairs are sorted in descending order of $\hat{\sigma}(u, v)$, where $u \in N[v]$. Note that, the estimated similarity is derived via Equation 1. This obviously takes a space proportional to the graph size $O(n + m)$. Next, we elaborate on the *bucket index* which is used to find the core vertices efficiently.

The bucket index takes δ as the number of the buckets and then divide the similarity score into δ evenly partition ranges: $[1 - 1/\delta, 1], [1 - 2/\delta, 1 - 1/\delta), \cdots [1 - i/\delta, 1 - (i - 1)/\delta) \cdots, [0, 1/\delta)$. For the i-th bucket $[1 - i/\delta, 1 - (i - 1)/\delta)$, it maintains a sorted set $B[i]$ of pairs $\langle u, |N_{1-i/\delta,\rho}[u]|\rangle$, where $N_{1-i/\delta,\rho}[u]$ is the set of (ϵ, ρ)-neighbors of u when $\epsilon = 1 - i/\delta$. For the pairs stored in bucket i, they are sorted in descending order of the number of (ϵ, ρ)-neighbors of each vertex with $\epsilon = 1 - i/\delta$. For the sorted set $B[i]$, we use a binary search tree to maintain the sorted pairs to support efficient updates and searches. Since each vertex appears at most δ times in the bucket index, the space consumption of the bucket index is bounded by $O(\delta \cdot n)$.

Given the bucket index, it can efficiently identify the set of core vertices. In particular, given input parameters ϵ and μ, we first find the bucket i^* whose range covers ϵ, i.e., $\epsilon \in [1 - i^*/\delta, 1 - (i^* - 1)/\delta)$. Then, with bucket $B[i^*]$, it traverses the entries in decreasing order of the $(1 - i^*/\delta, \rho)$ -neighbors and identifies the set of nodes with the size of $(1 - i^*/\delta, \rho)$-neighbors no smaller than μ as core vertices. This takes a cost proportional to the number of core vertices.

Example 3. Still consider the graph G_1 in Figure 1. Assume

Algorithm 1: BOTBIN-Index-Construction

Input: Input graph $G = (V, E)$, a random permutation h
Output: BOTBIN-Index of G

1 **for** *vertex* $v \in V$ *in increasing order of their* $h(v)$ *values* **do**
2 update $S_k(u)$ with $u \in N[v]$;
3 **for** *vertex* $v \in V$ **do**
4 **for** *vertex* $u \in N[v]$ *such that* $h(u) > h(v)$ **do**
5 Derive the estimated similarity $\hat{\sigma}(u, v)$ of u and v with $S_k(u), S_k(v)$ via Equation 1;
6 Add $\langle u, \hat{\sigma}(u, v)\rangle$ into sorted set $NO[v]$;
7 **for** *i from* δ *to 1* **do**
8 cnt $\leftarrow |\{w | w \in N[v], \hat{\sigma}(v, w) \geq 1 - \frac{i}{\delta}\}|$;
9 **if** $cnt == 1$ **then** break;
10 Add $\langle v, cnt\rangle$ into bucket $B[i]$;
11 **return** S_k, B, NO;

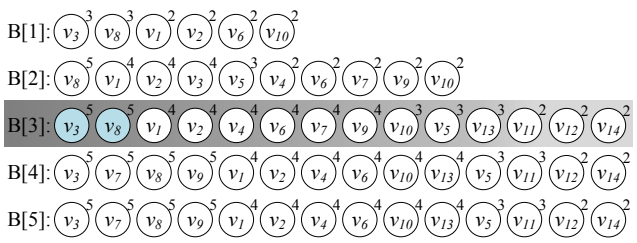


Fig. 2: The bucket index with $\delta = 5$, where the number attached to the top-right corner of each vertex represents the count of $(1 - i/\delta, \rho)$-neighbors of that vertex.

that the estimated similarity scores are the same as the exact similarity scores. Given $\delta = 5$, the corresponding bucket index is shown in Figure 2. Then, given $\mu = 5$ and $\epsilon = 0.5$, we use the bucket index to first find the core vertices. In particular, $0.5 \in [0.4, 0.6)$ and thus we use $B[3]$ to find the core vertices. Then, in $B[3]$, the set of vertices that has the $(1 - i^*/\delta, \rho)$-neighbor no smaller than $\mu = 5$ consists of $\{v_3, v_8\}$. Then, v_3 and v_8 are returned as the core vertices.

Next, we provide the following theorem for the approximation guarantee with our BOTBIN index scheme.

Theorem 3 (Approximation guarantee with bucket index)**.** *Given the bucket size δ and the ρ-absolute approximation guarantee of the estimated similarity score provided by the similarity index, the returned clustering result provides a* $(\rho + 1/\delta)$*-approximate SCAN.*

3.2 Index Construction and Query Processing

Index construction. Algorithm 1 shows the pseudo-code of the BOTBIN index construction algorithm. Given the hash function h generated via a random permutation, BOTBIN builds the similarity index from the vertex with the smallest hash value. Then, the similarity index can be constructed easily with $O(n+m)$ cost (Lines 1-2). After building the similarity index of each vertex, we can calculate approximate similarity of any neighboring node pairs with a running cost of $O(\frac{1}{\rho^2} \log \frac{1}{p})$ (Line 5). In total, it takes $O(\frac{m}{\rho^2} \log \frac{1}{p})$ cost to compute the similarities. Then it costs $O(\log d_u)$ time to add $\langle v, \hat{\sigma}(u, v)\rangle$ into sorted set $NO[u]$ as shown in Line 6. In total, to build the neighboring index, it takes $O(\sum_{v \in V} d_v \log d_v)$ cost, which can be further bounded by $O(m \log d_{max})$.

After constructing the neighboring index for vertex v, BOTBIN inserts entries about v in the bucket index (Lines 7-9). This can be finished via a linear scan of the neighboring index of v, which takes $O(d_v)$ cost. For the bucket index, notice that each vertex v maintains $O(\delta \cdot \sigma_{max}(v))$ entries among δ buckets, where $\sigma_{max}(v)$ is the maximum similarity score between v and its neighbors. To explain, there is no vertex with similarity larger than $\sigma_{max}(v)$ (except vertex v itself). Thus, for a bucket whose corresponding range is a subset of $(\sigma_{max}(v), 1]$, it will not contain vertex v. Let $\bar{\sigma}_{max}$ denote the average of this maximum similarity score, i.e., $\bar{\sigma}_{max} = \frac{1}{n} \sum_{v \in V} \sigma_{max}(v)$. Then, the bucket index takes $O(\delta \cdot n \cdot \bar{\sigma}_{max})$ space in total. For each entry in the bucket index, it takes $O(\log n)$ cost to do the insertion. In total, the running cost to build the bucket index can be bounded by $O(m + \delta \cdot n \cdot \bar{\sigma}_{max} \log n)$. According to the above discussions, we have the following theorem:

Theorem 4. *Given an error parameter ρ, a failure probability p, BOTBIN takes* $O\left(m\left(\frac{1}{\rho^2} \ln \frac{1}{p} + \log d_{max}\right) + \delta \cdot n \cdot \bar{\sigma}_{max} \log n\right)$ *running cost to build the index and has a space cost of* $O(m + \delta \cdot n \cdot \bar{\sigma}_{max})$.

For GS-Index, it takes a worst case of $O(m^{1.5})$ time to build the index and its space cost is $O(m + n)$. Clearly, our proposed BOTBIN is more scalable in terms of index construction time complexity. In terms of space cost, both methods has a space cost linear to the input graph. As we will show in our experiment, our BOTBIN may have slightly smaller memory consumption compared to GS-Index (Figure 8) with our default choice of δ. To explain, it is observed that on almost all datasets, $\delta \cdot \bar{\sigma}_{max}$ is actually smaller than the average degree of all the tested graphs on our default choice of δ.

Query processing. Algorithm 2 shows the pseudo-code of the query algorithm. Given the input parameter ϵ and μ, the BOTBIN index, it firstly finds the bucket i^* that covers ϵ (Line 1). Next, it initializes the clustering result $\mathcal{C}$ and maintains a hash table H of the core vertices. As described in Section 3.1, it visits bucket $B[i^*]$ in descending order of the size of their $(1 - i^*/\delta, \rho)$-neighbors and stops as soon as the number is smaller than μ. The core vertices are added to the hash table H (Lines 3-5).

Next, it initializes a queue Q and a size n array $visit$ for the cluster generation. The $visit$ array is used to identify the core vertices that have been traversed and added to previous clusters (Line 6). Then, for each core vertex v, it first checks if v has been traversed and added to previous clusters. If the answer is yes, it then continues to the next core vertex. Otherwise, it traverses the graph to generate the clusters corresponding to core vertex v. In particular, it adds v to the cluster C, adds v to Q, and sets $visit[v]$ to be 1. Then, it proceeds a pruned BFS where only neighbors who are core vertices are added to the queue (Lines 11-17). Firstly, it finds the front element u of queue Q. Then, it visits all its neighbors in descending order of the estimated similarity

Algorithm 2: BOTBIN-Query

Input: Parameters ϵ and μ, BOTBIN index
Output: the set $\mathcal{C}$ of clustering result

1 $i^* \leftarrow \arg_i\{\epsilon \in [1-i/\delta, 1-(i-1)/\delta)\}$;
2 $\mathcal{C} \leftarrow \emptyset, H \leftarrow$ an empty hash table;
3 **for** $\langle v, cnt\rangle \in B[i^*]$ *in descending order of the* cnt *values* **do**
4 **if** $cnt < \mu$ **then** break ;
5 Add v to H;
6 Init Q as an empty queue, $visit$ a size n array with all zeros;
7 **for** $v \in H$ **do**
8 **if** $visited[v]$ **then** continue;
9 $C \leftarrow \{v\}$, add v into Q, $visit[v] \leftarrow 1$;
10 **while** Q *is not empty* **do**
11 $u \leftarrow Q.dequeue()$;
12 **for** $\langle w, \hat{\sigma}(w,u)\rangle$ *in sorted set* $NO[u]$ **do**
13 **if** $\hat{\sigma}(w,u) < \epsilon$ **then** break;
14 $C \leftarrow C \cup \{w\}$;
15 **if** $visit[w]$ **then** continue;
16 $visit[w] \leftarrow 1$;
17 **if** $w \in H$ **then** $Q.enqueue(w)$;
18 $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$;
19 return $\mathcal{C}$;

scores via the neighboring index $NO[u]$. If the neighbor w of vertex u has a similarity no smaller than ϵ, w is an (ϵ, ρ)-neighbor of u. Thus, w is added into cluster C (Line 14). If w has been visited before, the pruned BFS continues (Line 15). Otherwise, it sets $visit[w]$ to be 1 (Line 16). Next, w is added to the queue if w is not visited in previous iterations (Line 15) and is a core vertex (Line 17). The iteration stops when the queue is empty. Cluster C is then added to the clustering result $\mathcal{C}$. When all core vertices finish pruned BFSs, it returns $\mathcal{C}$ as the final result.

Clearly, the running cost is proportional to the number of vertices and edges visited during the pruned BFS traversals. It is easy to verify that this cost can be easily bounded by the size of the cluster subgraph (Definition 6) as we will visit each core vertex at most and the (ϵ, ρ)-neighbor of each core vertex at most once. Theorem 5 summarizes the running cost for query processing.

Theorem 5. *The time complexity of Alg. 2 is bounded by* $O(|V_{\mathcal{C}}| + |E_{\mathcal{C}}|)$ *where* $V_{\mathcal{C}}$ *and* $E_{\mathcal{C}}$ *are the vertex set and edge set of the cluster subgraph of the final clustering result* $\mathcal{C}$.

Compared to GS-Index, our BOTBIN achieves the same query time complexity that depends on the size of cluster subgraph, i.e., $O(|V_{\mathcal{C}}|+|E_{\mathcal{C}}|)$. As we will show in the experiment, our BOTBIN achieves an identical query performance as that of GS-Index.

3.3 Extension to Dice similarity

In addition to the Jaccard similarity, our estimation scheme that utilizes the bottom-k sketch can be adapted to estimate the Dice similarity between two sets.

Definition 10. *Given two sets* X *and* Y*, the Dice similarity between set* X *and set* Y *is defined as* $\frac{2|X\cap Y|}{|X|+|Y|}$.

Let $\sigma_D(X,Y)$ and $\sigma(X,Y)$ denote the Dice similarity and Jaccard similarity of sets X and Y, respectively. Their intrinsic connection is derived from the definitions:

$$\sigma_D(X,Y) = \frac{2|X\cap Y|}{|X|+|Y|} = \frac{2\sigma(X,Y)}{1+\sigma(X,Y)} = 2 - \frac{2}{1+\sigma(X,Y)}. \tag{2}$$

Thus, if an ρ-absolute error guarantee is provided for the Jaccard similarity using our proposed method, then based on Equation 2, a 2ρ-absolute error guarantee for the Dice similarity can also be derived as explained below.

Let $\hat{\sigma}_D(X,Y) = 2 - \frac{2}{1+\hat{\sigma}(X,Y)}$, where $\hat{\sigma}(X,Y)$ is the estimator for the Jaccard similarity defined by Equation 1. According to Theorem 2, the estimator $\hat{\sigma}(X,Y)$ satisfies $|\sigma(X,Y) - \hat{\sigma}(X,Y)| \le \rho$ with high probability p, when setting $k = \frac{1}{2\rho^2}\ln\frac{2}{p}$. Consequently, we derive that $|\sigma_D(X,Y) - \hat{\sigma}_D(X,Y)| < 2\rho$ as follows:

$$\begin{aligned}\hat{\sigma}_D(X,Y) = 2 - \frac{2}{1+\hat{\sigma}(X,Y)} &\le 2 - \frac{2}{1+\sigma(X,Y)+\rho} \\ &\le 2 - \frac{2}{1+\sigma(X,Y)} + 2\rho = \sigma_D(X,Y) + 2\rho.\end{aligned}$$

The other side of the inequality can be derived in a similar manner. Thus, our BOTBIN (i.e., Algorithms 1–2) can be seamlessly extended to address the structural graph clustering problem based on the Dice similarity. The time complexity of building the index for BOTBIN for the Dice similarity remains the same as for the Jaccard similarity, as the variation lies only in a constant term.

In our experimental section (i.e., Section 6.2), we will further examine our effectiveness to Dice similarity in terms of the quality of the clustering results. From the experimental results, we will observe that on Dice similarity, our solution still achieves highly accurate clustering results.

4 Dynamic Index Maintenance

Next, we present a basic solution to handle dynamic updates in Section 4.1. We assume that the edge updates arrive in a random order. Then, to further improve the performance and reduce the time complexity, we present two optimization techniques in Section 4.2 and 4.3, respectively.

4.1 Basic Methods

Edge insertion. We first discuss the effects after adding an new edge (u,v). Firstly, v (resp. u) is added to the set $N[u]$ (resp $N[v]$) of the neighborhood of vertex u (resp. v). Therefore, the similarity index of vertex u (resp. v) might be changed. Thus, the first step is to update the similarity index of u and v if necessary. Next, if the similarity index of vertex u (resp. v) is not changed, then the similarity between u (resp. v) and its neighborhood $N[u]$ (resp. $N[v]$) before the v (resp. u) is added is also unchanged. No further update is required. On the other hand, if the similarity index gets updated, then further updates are required. We focus on the case when the similarity index $S_k(u)$ of u gets changed as the case for v can be discussed in the same way. When $S_k(u)$ gets changed, for a vertex w in $N[u]$, their similarity score

Algorithm 3: BOTBIN-Update-Insertion

Input: Inserted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Update $S_k(u), S_k(v)$;
2 **if** $S_k(u)$*is modified* **then**
3 **for** *vertex* $w \in N[u]$ **do**
4 $\hat{\sigma}_{old} \leftarrow$ estimated similarity $\hat{\sigma}(u, w)$ before update;
5 Derive $\hat{\sigma}(u, w)$ of u and w with $S_k(u), S_k(w)$;
6 Insert/update $\langle w, \hat{\sigma}(u, w)\rangle$ in $NO[u]$;
7 Insert/update $\langle u, \hat{\sigma}(w, u)\rangle$ in $NO[w]$;
8 UpdateBucket(w, $\hat{\sigma}_{old}$, $\hat{\sigma}(u, w)$, $w = v$);
9 Update all records of B associated with u;
10 **else**
11 Derive $\hat{\sigma}(u, v)$ of u and v with $S_k(u), S_k(v)$;
12 Insert $\langle v, \hat{\sigma}(u, v)\rangle$ to $NO[u]$;
13 UpdateBucket(u, 0, $\hat{\sigma}(u, v)$, 1);
14 Repeat Lines 2-13 for vertex v;
15 **return** S_k, B, NO;
16 **procedure** *UpdateBucket*(w, σ_{old}, σ_{new}, $is\_updated\_edge$)**:**
17 $i^* \leftarrow \arg_i\{\sigma_{old} \in [1 - i/\delta, 1 - (i-1)/\delta)\}$;
18 $j^* \leftarrow \arg_i\{\sigma_{new} \in [1 - i/\delta, 1 - (i-1)/\delta)\}$;
19 **if** $i^* > j^*$ **then** swap(i^*, j^*);
20 **if** !$is\_updated\_edge$ **then** $j^* \leftarrow j^* - 1$;
21 **for** $i \in [i^*, j^*]$ **do**
22 The record of w in $B[i]$ increases by 1 if $\sigma_{old} < \sigma_{new}$ otherwise decreases by 1;
23 Insert/update the entry of w in $B[i]$;

Algorithm 4: BOTBIN-Update-Deletion

Input: Deleted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Update $S_k(u), S_k(v)$;
2 **if** $S_k(u)$*is modified* **then**
3 Do the same operations of Lines 3-9 in Algorithm 3;
4 $\hat{\sigma}_{old} \leftarrow$ estimated similarity $\hat{\sigma}(u, w)$ before update;
5 Delete the entry of v from $NO[u]$;
6 UpdateBucket(u, $\hat{\sigma}_{old}$, 0, 1);
7 Repeat Lines 2-6 for vertex v;
8 **return** S_k, B, NO;

may also get changed. Then, the neighboring index $NO[w]$ of vertex w and the entries related to w in the bucket index might also need to be updated.

Algorithm 3 shows the pseudo-code of BOTBIN index update for insertion. Firstly, it updates the similarity index of u and v respectively if it is needed (Line 1). If the similarity index of vertex u is changed, it examines all vertices in the neighborhood of u. Let w be a vertex in $N[u]$, it obtains the similarity score derived before the update (Line 4). Next, given the updated similarity index, it computes the updated similarity score between u and w with $S_k(u)$ and $S_k(w)$ via Equation 1. Then it updates (resp. inserts) $\langle w, \hat{\sigma}(u, w)\rangle$ in $NO[u]$ if w is not v (resp. w is v) (Line 6). Similarly, $NO[w]$ needs to be updated as $\hat{\sigma}(w, u)$ is also changed with vertex u as the key in $NO[w]$ (Line 7). After that, it updates the bucket index for vertex w by invoking the UpdateBucket procedure (Lines 16-23). Given the old similarity score σ_{old} and the updated similarity score σ_{new}, it first identifies the bucket i^* of σ_{old} and bucket j^* of σ_{new}. If i^* is equal to j^*, the bucket is unchanged. Notice that if the updated bucket is related to $\hat{\sigma}(u, v)$, i.e., the edge to be updated (insertion/deletion), the information in bucket i^* also needs to be updated. That is why it includes an $is\_updated\_edge$ flag. If it is not related to the updated edge, then there is no need to update the bucket information. Otherwise, if the bucket changes from i^* to j^* (assuming $i^* < j^*$ here), entries $\langle w, cnt\rangle$ in buckets $B[i^*]$ to $B[j^*]$ all need to be updated from i^* to j^* (resp $j^* - 1$) if the similarity is (resp. is not) for the updated edge (Lines 16-23). The case when $i^* > j^*$ can be handled in the same way. To facilitate the update, for each vertex w, BOTBIN maintains the count of approximate neighbors in each bucket. Thus, we can easily use the maintained count to find the location of each entry $\langle w, cnt\rangle$ in bucket $B[i]$ via a binary search. After updating all neighborhoods of u, it further updates the bucket information for vertex u via the maintained count. If $S_k(u)$ is not changed, then the similarity between u and $w \in N[u] \setminus \{v\}$ is not changed. Thus, only the similarity between u and v must be computed (Line 11). Then, an entry $\langle v, \hat{\sigma}(u, v)\rangle$ is added to $NO[u]$. Lastly, it invokes the UpdateBucket algorithm to update the bucket index related to u with the maintained count for each bucket. This process is repeated for v, whose steps are identical to that for u. This finishes the update to BOTBIN index.

Edge deletion. Next, we elaborate on how to update the index when there is an edge deletion. With the deletion of an edge (u, v), the first step is still to update the similarity index of vertices u and v. Then, if the similarity index of u is updated, it might affect the similarity between u and all nodes in its neighborhood $N[u]$. Thus, similar to Algorithm 3, it updates the neighboring index of w where $w \in N[u]$, and further updates the bucket index related to w. The steps are the same as that in the insertion by repeating Lines 3-9 in Algorithm 3 (Algorithm 4 Line 3). Next, it handles the deleted edge (u, v) (Lines 4-6). In particular, it deletes the entry of v from the neighboring index $NO[u]$ of vertex u. Then, it further updates entries related to u in the bucket index. Finally, it repeats this process for vertex v (Line 6). After that, it finishes the index update and returns the updated index (Line 8).

Notice that similar to edge insertion, the major cost comes from the recalculation of the similarity scores and the updates to the bucket index. Thus, the time complexity of the BOTBIN index update given an edge deletion is the same as that of edge insertion. We have the following theorem to bound the index update cost.

Theorem 6. *Given a failure probability p, an error parameter ρ, for an edge insertion/deletion, the expected cost to update the BOTBIN index by Algo. 3 /Algo. 4 is*

Algorithm 5: BOTBIN-Update-Insertion-Opt

Input: Inserted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Let pop_u be the k-th smallest element in original $S_k(u)$;
2 Update $S_k(u), S_k(v)$;
3 **if** $S_k(u)$*is modified* **then**
4 **for** *vertex* $w \in N[u]$ **do**
5 $k_{old} \leftarrow kth(u, w)$;
6 **if** $h(v) \leq k_{old}$ **then**
7 **if** $h(v) \in S_k(w)$ **then**
8 $cnt(u, w) \leftarrow cnt(u, w) + 1$
9 **else**
10 $kth(u, w) \leftarrow find\_kth(S_k(u), S_k(w), k_{old})$;
11 **if** (k_{old} *in* $S_k(u)$ *or* $pop_u = k_{old}$) *and* k_{old} *in* $S_k(w)$ **then**
12 $cnt(u, w) \leftarrow cnt(u, w) - 1$;
13 $\hat{\sigma}_{old} \leftarrow$ estimated similarity $\hat{\sigma}(u, w)$ before update;
14 Derive $\hat{\sigma}(u, w)$ of u and w as $cnt(u, w)/k$;
15 Do the same operations of Lines 6-8 in Algorithm 3;
16 Update all records of B associated with u;
17 **else**
18 Do the same operations of Lines 11-13 in Algorithm 3;
19 Repeat Lines 2-18 for vertex v;
20 **return** S_k, B, NO;

Algorithm 6: BOTBIN-Update-Deletion-Opt

Input: Inserted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Update $S_k(u), S_k(v)$;
2 **if** $S_k(u)$*is modified* **then**
3 **for** *vertex* $w \in N[u]$ **do**
4 $k_{old} \leftarrow kth(u, w)$;
5 **if** $h(v) \leq k_{old}$ **then**
6 **if** $h(v)$ *in* $S_k(w)$ **then**
7 $cnt(u, w) \leftarrow cnt(u, w) - 1$
8 **else**
9 $kth(u, w) \leftarrow find\_kth(S_k(u), S_k(w), k_{old})$;
10 **if** $kth(u, w) \in S_k(u)$ *and* $kth(u, w) \in S_k(w)$ **then**
11 $cnt(u, w) \leftarrow cnt(u, w) + 1$;
12 Do the same operations of Lines 13-15 of Algorithm 5
13 Update all records of B associated with u;
14 Do the same operations of Lines 4-6 of Algorithm 4
15 Repeat Lines 2-14 for vertex v;
16 **return** S_k, B, NO;

$$O\left(\frac{1}{\rho^2}\log\frac{1}{p}\cdot\left(\frac{1}{\rho^2}\log\frac{1}{p}+\delta\log n\right)\right).$$

4.2 Optimized Solution for Similarity Recalculation

One key bottleneck of solutions in Section 4.1 is that it takes $O(\frac{1}{\rho^2}\log\frac{1}{p})$ time to recalculate the similarity between u (resp. v) and its neighborhood $N[u]$ (resp. $N[v]$) each time after an edge insertion/deletion to (u, v). This cost is the main update cost as we will show in the experiment. A natural question is: can we reduce the cost for the recalculation of the similarity between u (resp. v) and its neighborhood $N[u]$ (resp. $N[v]$)? Intuitively, the need to recalculate the similarity between u and $w \in N[u]$ after updating the sketch of u seems to be redundant as we only update at most one entry in the sketch of u, making the majority of the sketch untouched. This motivates us to devise a more efficient algorithm to re-compute $\hat{\sigma}(u, w)$, reducing the cost to re-compute the similarity score from $O(\frac{1}{\rho^2}\ln\frac{1}{p})$ to $O(\log(\frac{1}{\rho^2}\log\frac{1}{p}))$. Next, we elaborate on the improved solutions to re-compute the similarity scores.

Faster update for insertion. Consider the impact of adding an edge (u, v) to the sketch of vertex u. If $h(v)$ is larger than the k-th smallest hash value in similarity index $S'_k(u)$ before the update, then the sketch $S_k(u)$ after the insertion remains the same. There is no need to re-compute the similarity between u and w where $w \in N[u] \setminus \{v\}$. If $h(v)$ is among the k smallest hash values in $N[u]$, $h(v)$ will be added into the updated similarity index $S_k(u)$ and the k-th smallest hash value is removed from original similarity index $S'_u(k)$. It is easy to verify that $S'_k(u)$ and $S_k(u)$ differ by at most two elements. The first element that may be different is the k-th smallest element of original similarity index $S'_k(u)$. Let pop_u denote this element and it will be eliminated from $S_k(u)$. The second different element is $h(v)$ because it has just been added to $S_k(u)$.

Given this information, how can we quickly compute the approximate similarity of $S_k(u)$ and $S_k(w)$? To speed up the similarity re-computation, we maintain two values attached to each edge (u, w). The first value is $cnt(u, w)$, which stores the number of elements in the intersection of $S_k(u)$, $S_k(w)$, and $S_k(N[u] \cup N[w])$. Notice that, given this $cnt(u, w)$, we can then directly derive the estimated similarity $\hat{\sigma}(u, w)$ between u and w as $cnt(u, w)/k$ according to Equation 1. The second element that we maintain for edge (u, w) is $kth(u, w)$, which stores the k-th element of $S_k(N[u] \cup N[w])$. All these values can be initialized during the index construction without affecting the time complexity of the index construction algorithm. Since this incurs $O(m)$ cost, it also does not affect the space complexity of the BOTBIN index. Algorithm 5 shows the pseudo-code of the optimized insertion algorithm.

Recap that pop_u is the k-th element in $S'_k(u)$ before the update (Line 2). If $S_k(u)$ is modified, then it affects the similarity between u and $w \in N[u]$ and re-computation is required (Line 3). For each vertex w in the neighborhood of u, denote k_{old} as the k-th element of $S_k(N[u] \cup N[w])$ before the insertion, i.e., being equal to $kth(u, w)$ before the edge

insertion (Line 5). We focus on the case when $h(v) \leq k_{old}$ (Line 6) as otherwise, it will not affect $cnt(u,w)$ or $kth(u,w)$ and thus the estimated similarity between u and w is unchanged. In this case, i.e., $h(v) \leq k_{old}$, if $h(v)$ is also in $S_k(w)$, then we can know that $cnt(u,w)$ will be incremented by one (Lines 7-8). Notice that for the removed pop_u, it will be larger than $h(v)$, and further larger than k_{old}. Hence, it is not in the intersection of $S_k[u] \cap S_k[w] \cap S_k[S_k[u] \cup S_k[w]]$ and will not affect $cnt(u,w)$. If $h(v)$ is not in $S_k(w)$, then we first update $kth(u,w)$ according to $S_k(u)$, $S_k(w)$, and k_{old}. In particular, as we only modify the k-th smallest by one position, we can first find the largest element in $S_k(u)$ that is smaller than k_{old} and the largest element in $S_k(w)$ that is smaller than k_{old} then take the larger one of these two. This will be the k-th smallest element in $S_k(N[u] \cup N[w])$ in the new similarity index (Line 10). After that, we check if k_{old} that is kicked out from $S_k(N[u] \cup N[w])$ is in both $S_k(u)$ and $S_k(w)$. If this is the case, we decrease $cnt(u,w)$ by 1 (Lines 11-12). Notice that we might not be able to find k_{old} in $S_k(u)$ as it has been removed from $S_k(u)$, i.e., pop_u. Thus, if $k_{old} = pop_u$, it also indicates that k_{old} was in $S_k(u)$ (Line 11). After updating $cnt(u,w)$ and $kth(u,w)$, we are able to derive the updated similarity between u and w via $cnt(u,w)/k$. For the remaining parts to update th neighboring index and bucket index, it is the same as that in Algorithm 3 (Lines 15-18). Finally, it repeats this process for v and returns the updated index.

Faster update for deletion. The improved update algorithm for edge deletion shares the similar idea as that for insertion, i.e., maintaining $cnt(u,w)$ and $kth(u,w)$ for each pair of edges and dynamically tracking the changes of these two. Algorithm 6 shows the pseudo-code of the index update algorithm for edge deletion. Firstly, the similarity between u and nodes in the neighborhood of u are not changed if $S_k(u)$ is not changed. When $S_k(u)$ is updated, then $kth(u,w)$ is the k-th smallest hash value for set $N[u] \cup N[w]$ before the graph update. Denote this as k_{old} (Line 4). Next, we only need to handle the case when $h(v) \leq k_{old}$, otherwise the deletion of $h(v)$ will not affect $cnt(u,w)$ and $kth(u,w)$. In such a case, i.e., $h(v) \leq k_{old}$, if $h(v)$ is in $S_k(w)$, then $cnt(u,w)$ decreases by 1. However, $kth(u,w)$ remains unchanged as $h(v)$ remains in $S_k(N[u] \cup N[v])$ and thus $S_k(N[u] \cup N[v])$ is not changed. If $h(v)$ is not in $S_k(w)$, we first derive the updated $kth(u,w)$ by computing the smallest element in $S_k(u)$ that is larger than k_{old} and the smallest element in $S_k(w)$ that is larger than k_{old} and taking the smaller one. Then, if $kth(u,w)$ appears in both $S_k(u)$ and $S_k(w)$, then $cnt(u,w)$ increases by 1. This finishes the update to $kth(u,w)$ and $cnt(u,w)$. Next, the remaining steps to update the bucket index and neighboring index are similar to that of Algorithm 5 (Lines 12-13). Next, it removes v from the neighboring index of u, updates the bucket index of u based on the estimated similarity score $\hat{\sigma}(u,w)$ before the update (Line 14), which is the same as Algorithm 4 Lines 4-6. Finally, it repeats this process for vertex v (Line 15) and returns the updated index. We have the following theorem to bound the index update cost for the improved algorithms.

Theorem 7. *Given a failure probability p, an error parameter ρ, for an edge insertion/deletion, the expected cost to update the BOTBIN index by Algo. 5/Algo. 6 is bounded by* $O\left(\frac{\delta}{\rho^2} \cdot \log \frac{1}{p} \cdot \log n\right)$.

4.3 Optimized Bucket Index

In Section 4.2, we have developed an efficient solution to accelerate similarity recomputation with improved time complexity. When seeking to further improve the update efficiency, we identify that the theoretical bottleneck lies at the overhead associated with updating the bucket index. To explain, for bucket $B[i]$, we use a Binary Search Tree to maintain a set of pairs $\langle vid, cnt \rangle$, sorted by their cnt values, where cnt denotes the number of $(1 - i/\delta, \rho)$-neighbors of vertex vid. Therefore, updating the cnt value of an element in $B[i]$ requires $O(\log n)$ time, since the bucket size is $O(n)$. In contrast, updating the similarity index portion takes only $O(\log k)$ time per vertex. According to Theorem 9, which will be presented at the end of this section, we can conclude that setting $k = O(\log n)$ is sufficient to achieve clustering results with approximation guarantees. Consequently, the cost of updating the bucket index becomes a major bottleneck for our BOTBIN. Therefore, to further improve update efficiency, we focus on the maintenance of the bucket index.

We observe that, when updating a bucket $B[i]$, the cnt value of the specified vertex changes by exactly 1, either increasing or decreasing. Therefore, this allows us to leverage this characteristic to design a more efficient bucket index. For simplicity, we refer to a pair $\langle vid, cnt \rangle$ as an element in the bucket. We propose the SegOrder (Segmented Ordered) structure to handle updates. The main idea of the proposed structure is to treat elements with the same cnt value as a contiguous segment and manage updates by maintaining the left and right endpoints of this segment. When an entry is updated, assuming this involves an increment operation, we swap this element with the left endpoint of the segment it belongs to. Then, we only need to update the left and right endpoints of the segment and its adjacent segments. This approach still incurs linear space overhead, while the update cost for increment or decrement of a specific element can be reduced from logarithmic time cost to $O(1)$ time cost complexity. Next, we will provide a detailed explanation of how the new bucket index handles edge updates.

SegOrder Structure. For each bucket, we utilize a SegOrder structure that comprises the following members: $elements$, pos, $leftP$, and $rightP$. The member $elements$ is an array containing pairs $\langle vid, cnt \rangle$, sorted in descending order of cnt values. The member pos is a hash table designed to enable querying the position of a given vertex ID within the $elements$ array. Additionally, the members $leftP$ and $rightP$ are arrays that store the positional range within the $elements$ array corresponding to a specific cnt value. For instance, for a given value c, the range from $leftP[c]$ to $rightP[c]$ includes all elements where cnt equals c. Utilizing the SegOrder structure, we can efficiently scan the $elements$ array to identify core vertices in the bucket. Next, we will describe how to increment or decrement the cnt value of a specific vertex with this new SegOrder structure.

SegOrder increment. Algorithm 7 shows the pseudo-code for how SegOrder handles the increment operation for a given vertex v. Given the current bucket denoted as $B[cur]$, we update the index by modifying the auxiliary structures $B[cur].pos$, $B[cur].elements$, $B[cur].leftP$ and

Algorithm 7: SegOrder-Increment

Input: cur-th bucket index $B[cur]$ needs to be updated, vertex v needs to be updated
Output: Updated bucket index $B[cur]$

1 $p_v \leftarrow pos.find(v)$;
2 $cnt_v \leftarrow elements[p_v].cnt$;
3 $p_{new} \leftarrow leftP[cnt_v]$;
4 $elements[p_v].cnt \leftarrow elements[p_v].cnt + 1$
5 **if** $p_{new} \neq p_v$ **then**
6 $\quad pos[elements[p_{new}].vid] = p_v$;
7 $\quad pos[elements[p_v].vid] = p_{new}$;
8 $\quad swap(elements[p_v], elements[p_{new}])$;
9 $leftP[cnt_v] \leftarrow leftP[cnt_v] + 1$;
10 **if** $leftP[cnt_v] > rightP[cnt_v]$ **then**
11 $\quad leftP[cnt_v] \leftarrow rightP[cnt_v] \leftarrow -1$;
12 **if** $leftP[cnt_v + 1] = -1$ **then**
13 $\quad leftP[cnt_v + 1] \leftarrow rightP[cnt_v + 1] \leftarrow p_{new}$;
14 **else**
15 $\quad rightP[cnt_v + 1] \leftarrow rightP[cnt_v + 1] + 1$;
16 **return** $B[cur]$;

Algorithm 8: SegOrder-Decrement

Input: cur-th bucket index $B[cur]$ needs to be updated, vertex v needs to be updated
Output: Updated bucket index $B[cur]$

1 $p_v \leftarrow pos.find(v)$;
2 $cnt_v \leftarrow elements[p_v].cnt$;
3 $p_{new} \leftarrow rightP[cnt_v]$;
4 $elements[p_v].cnt \leftarrow elements[p_v].cnt - 1$;
5 Do the same operations of Lines 5-8 of Algorithm 7;
6 $rightP[cnt_v] \leftarrow rightP[cnt_v] - 1$;
7 **if** $leftP[cnt_v] > rightP[cnt_v]$ **then**
8 $\quad leftP[cnt_v] \leftarrow rightP[cnt_v] \leftarrow -1$;
9 **if** $leftP[cnt_v - 1] = -1$ **then**
10 $\quad leftP[cnt_v - 1] \leftarrow rightP[cnt_v - 1] \leftarrow p_{new}$;
11 **else**
12 $\quad leftP[cnt_v - 1] \leftarrow leftP[cnt_v - 1] - 1$;
13 **return** $B[cur]$;

$B[cur].rightP$. For brevity, we omit the prefix $B[cur]$ in the pseudocode. First, we locate the position of vertex v in the array $elements$, recording it as p_v (Line 1). We use cnt_v to denote the cnt value of vertex v (Line 2). Let p_{new} denote the leftmost index of the segment having the value cnt_v. The p_{new} indicates the new position to which vertex v will be swapped (Line 3). We increment the cnt value of vertex v, i.e., $elements[p_v].cnt$ (Line 4). Since the cnt value of vertex v has been modified, we have to re-locate this vertex in the array $elements$. Therefore, if $p_{new} \neq p_v$, we swap the elements at the positions p_v and p_{new} in the list $elements$, and also update the position table based on the information from p_v and p_{new} (Lines 5–8). Next, we update the left endpoint of the segment where the vertices with cnt_v value resides (Line 9). If the updated left endpoint of the segment for value cnt_v exceeds its right endpoint, we conclude that there exists no elements with the cnt equal to value cnt_v among $elements$. Consequently, we set both endpoints to -1 (Lines 10-11). On the other hand, if the segment for value $cnt_v + 1$ is empty, we initialize its endpoints based on p_{new} (Lines 12-13). Otherwise, we update the right endpoint of the segment for value $cnt_v + 1$ (Lines 14-15). Finally, we return the updated bucket $B[cur]$.

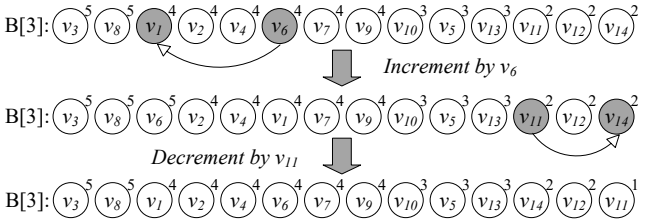


Fig. 3: Update example for the proposed SegOrder structure.

SegOrder decrement. Algorithm 8 shows the pseudo-code for how SegOrder handles the decrement operation for a specified vertex v. Similar to Algorithm 7, we first locate the position of vertex v in $elements$ (Line 1), and denote cnt_v as the cnt value of vertex v (Line 2). Let p_{new} be the *leftmost* index of the segment of elements having the value cnt_v (Line 3), indicating the new position to which vertex v will be swapped. Note that this p_{new} differs from the one in Line 3 of Algorithm 7. Then, we decrement the cnt value of vertex v, i.e., $elements[p_v].cnt$. Lines 5–8 follow the same steps as in Algorithm 7, where the elements at positions p_v and p_{new} are swapped. This indicates that, due to the modified cnt value, the segment containing vertex v has changed. Next, we update the left endpoint of the segment for value cnt_v (Line 9). If the updated left endpoint exceeds its right endpoint, we know that in the array $elements$, there are no elements with the cnt equal to cnt_v, thus setting both endpoints to -1 (Lines 10–11). If the segment for $cnt_v - 1$ is empty, its two endpoints is initialized as p_{new} (Lines 12–13). Otherwise, we update the left endpoint of the segment for value $cnt_v - 1$ accordingly (Lines 14-15). Finally, we return the updated bucket $B[cur]$.

Theorem 8. *Given a failure probability p, an error parameter ρ, for an edge insertion/deletion, the expected cost to update the BOTBIN index with the above bucket scheme is bounded by* $O\left(\frac{\delta}{\rho^2} \cdot \log \frac{1}{p} \cdot \log k\right)$.

Example 4. Figure 3 shows an example of updating a bucketing with the SegOrder structure. Suppose we are now updating the bucket $B[3]$, and we need to perform an increment operation on v_6. First, we retrieve the cnt value of v_6, which is 4. Then, we can see that the left endpoint of the segment, in which the elements with the cnt equal to 4 reside, is 2. It is the vertex v_1. We then swap the positions of v_1 and v_6, and increment the cnt value of v_6 by 1. Next, we update the right endpoint of the segment of elements with $cnt = 5$, increasing from 1 to 2. Similarly, we update the left endpoint of the segment with $cnt = 4$, increasing from 2 to 3. This completes the increment operation.

Subsequently, we perform a decrement operation on v_{11}. We take the cnt value of vertex v_{11}, which is 2. The right

endpoint of the segment of elements with $cnt = 2$ is 13, which stores vertex v_{14}. We then swap the positions of v_{11} and v_{14}, and update the value of v_{11}. Subsequently, we update the right endpoint of the segment for $cnt = 2$ from 13 to 12, and both the left and right endpoints of the segment for $cnt = 1$ are initialized to 13.

Choice of p. Finally, we discuss how to set p so that we can provide an approximation guarantee with $1-p_f$ probability after a sequence of updates. Suppose that there are m edges in the initial graph. Regardless of whether they modify the similarity index of u, v or not, each update of edge (u, v) can be seen as a re-sampling of the neighbors of u and v. After adding an edge, this is equivalent to $d_u + d_v + 1$ times of re-sampling. For deleting an edge, it affects the similarity score of $d_u + d_v - 2$ edges. Assuming that there are M random updates, on average, it affects $M \cdot d_E$ edges, where d_E represents the mean of the degrees of the two endpoints of each edge in the edge set E. So if we want to handle M updates, we should set $p = \frac{p_f}{M \cdot d_E}$. A total of m edges need to be initialized during the construction of the BOTBIN, and this part also needs to be considered in the error probability. By setting $p = p_f/(M \cdot d_E + m)$, we can guarantee that we have $1 - p_f$ success probability to produce an $(\epsilon, \rho + 1/\delta)$-approximate SCAN. We have the following theorem on the choice of k according to the above discussion.

Theorem 9. *Given an approximate parameter ρ and a probability parameter p_f, let $k = \frac{1}{\rho^2} \log \frac{2(M \cdot d_E + m)}{p_f}$, then BOTBIN can support M random updates and return $(\epsilon, \rho + 1/\delta)$-approximate SCAN with probability $1 - p_f$ for input graph G.*

Notice that $\log \frac{2(M \cdot d_E + m)}{p_f} = O(\log \frac{M+m}{p_f})$. Since δ and ρ can be treated as small constants, the final time complexity for each edge update becomes $O(\log \log \frac{M+m}{p_f} \cdot \log \frac{M+m}{p_f})$. With edge insertions/deletions, the number of affected edges may exceed $M \cdot d_E$. We set a reconstruction threshold $\zeta = M \cdot d_E$ with $M = m$. When the total number of affected edges reaches ζ, we rebuild our index and then double ζ. By setting $p = \frac{p_f}{M \cdot d_E}$ to Theorem 4, the index construction time is then bounded by $O\left(m\left(\frac{1}{\rho^2} \ln \frac{M+m}{p_f} + \log d_{max}\right) + \delta n \log n\right)$.

5 THEORETICAL ANALYSIS

Proof of Lemma 1. Given two sets A and B, assume that there are a elements in their intersection set and b elements in their union set. The bottom-k sketch of set A or B can be regarded as the sampling without replacement according to the permutation of h (assume that $k \leq b$). It is easy to verify that $S_k(A) \cap S_k(B) \cap S_k(A \cup B)$ indicates a subset of the bottom-k sketch of $A \cup B$ that also appears in the bottom-k sketch of set $A \cap B$. Let X_i be the indicator random variable such that the i-th smallest hash value in $S_k(A \cup B)$ also appears in the bottom-k sketch of $A \cap B$. Then, the probability of $X_i = 1$ is $\frac{a(b-1)!}{(b)!}$. Let $X = \sum_{i=1}^{k} X_i$. Then, we know that $X = S_k(A) \cap S_k(B) \cap S_k(A \cup B)$. By the linearity of expectation, we have $E[X] = k \cdot a/b$. So $S_k(A) \cap S_k(B) \cap S_k(A \cup B)/k$ is an unbiased estimator of the Jaccard similarity of A and B. □

Proof of Theorem 2. We use the following theorem for martingales to prove the guarantees in Theorem 2.

Theorem 10 (Azuma-Hoeffding Inequality [39])**.** *Let $M_0, M_1, \cdots, M_t$ be a martingale such that:*

$$B_k \leq M_k - M_{k-1} \leq B_k + d_k$$

for some constants d_k and for some random variables B_k that may be functions of $M_0, M_1, \ldots, M_{k-1}$. Then, for all $t \geq 0$ and any $\lambda > 0$,

$$Pr[|M_t - M_0| \geq \lambda] \leq 2exp(-2\lambda^2/(\sum_{k=1}^{t} d_k^2))$$

Given above theorem, we will define the martingale step by step. Define X as the same in Lemma 1. We consider the concentration of X. Define $f(X_1, \ldots, X_k) = \sum_{i=1}^{k} X_i$. Let $M_0 = \mathbb{E}[f(X_1, \ldots, X_k)]$, $M_i = \mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_i]$ and $M_k = \mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_k]$. Then, $M_0, M_1, \ldots, M_k$ forms a martingale, i.e., $\mathbb{E}[M_{i+1}|X_1, \ldots, X_i] = M_i$. To prove that $\mathbb{E}[M_{i+1}|X_1, \ldots, X_i] = M_i$, we have the following:

$\mathbb{E}[M_{i+1}|X_1, \ldots, X_i] =$
$\mathbb{E}[\mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_{i+1}]|X_1, \ldots, X_i]$ (Definition of M_{i+1})
$= \mathbb{E}[\mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_i]$ (The law of total expectation)
$= M_i$ (Definition of M_{i+1})).

In the second equation, we use the law of total expectation, i.e., $\mathbb{E}[X|Y] = \mathbb{E}[\mathbb{E}[X|U, Y]|Y]$. Let $X = f(X_1, \ldots, X_k), U = X_{i+1}$ and $Y = X_1, \ldots, X_i$. Therefore, $\{M_i\}_{i=0}^{k}$ is a martingale. Then we aim to get the gap of $M_i - M_{i-1}$ so as to apply Theorem 10. Let R_i be the number of elements in the intersection of A and B among the first i elements, i.e., $R_i = \sum_{j=1}^{i} X_j$. Clearly,

$$M_i = R_i + (k-i)\frac{a - R_i}{b - i}, M_{i-1} = R_{i-1} + (k-i+1)\frac{a - R_{i-1}}{b - i + 1}.$$

Then, using the fact that $R_i - R_{i-1} = X_i$, we can derive that:

$$M_i - M_{i-1} = \frac{b-k}{b-i}(X_i + \frac{R_{i-1} - a}{b - (i-1)}).$$

Fix $\frac{b-k}{b-i} \cdot \frac{R_{i-1}-a}{b-(i-1)}$ as B_i. Since X_i is 0 or 1. We have that:

$$B_i \leq M_i - M_{i-1} \leq B_i + \frac{b-k}{b-i}$$

Thus, we can set $b_i = 1$ as $\frac{b-k}{b-1} < 1$. Then we divide by k for every $M_i, 0 \leq i \leq k$. Clearly, $\frac{M_k}{k} = \hat{J}(A, B)$, and $\frac{M_0}{k} = \frac{a}{b} = J(A, B)$. Then, $\frac{d_i}{k}$ will be also less than $\frac{1}{k}$. Then, $\sum_{i=0}^{k} (\frac{d_i}{k})^2 < \frac{1}{k}$. Applying Theorem 10, we have:

$$\Pr[|\hat{J}(A, B) - J(A, B)| > \rho] \leq 2 \exp(-2k\rho^2)$$

When $k = \frac{1}{2\rho^2} \ln \frac{2}{p_f}$, then we have:

$$\Pr[|\hat{J}(A, B) - J(A, B)| > \rho] \leq p_f$$

This finishes the proof of Theorem 2. □

Proof of Theorem 3. Consider the cluster $C_{\epsilon+\rho+1/\delta} \in \mathcal{C}_{\epsilon+\rho+1/\delta,\mu}$. Because the similarity score σ of the edge in $C_{\epsilon+\rho+1/\delta}$ is no smaller than $\epsilon + \rho + 1/\delta$. With the ρ-absolute approximation guarantee of the estimated similarity score,

we know $\hat{\sigma} > \sigma - \rho$ and $\hat{\sigma} > \epsilon + \frac{1}{\delta}$. So this edge must be included in the bucket where ϵ is located. Since all edges in cluster $C_{\epsilon+\rho+1/\delta}$ will be handled in this bucket. There exists a cluster $C \in \mathcal{C}_{\epsilon,\mu}^{\rho+1/\delta}$, such that $C_{\epsilon+\rho+1/\delta} \subseteq C$. So by definition, we can know the first condition is satisfied. For every cluster $C \in \mathcal{C}_{\epsilon,\mu}^{\rho+1/\delta}$, there exists a cluster $C_{\epsilon-\rho-1/\delta} \in \mathcal{C}_{\epsilon-\rho-1/\delta,\mu}$, such that $C \subseteq C_{\epsilon-\rho-1/\delta}$. Consider an edge whose similarity $\hat{\sigma}$ is in this bucket. Since $\epsilon - \hat{\sigma} < \frac{1}{\delta}$ and $\sigma > \hat{\sigma} - \rho$, we get $\sigma > \epsilon - \rho - \frac{1}{\delta}$. Then, $C_{\epsilon-\rho-1/\delta}$ would contain this approximate cluster. □

Proof of Theorem 6. We analyze the expected update cost of updates step by step. We focus on insertion as the running cost of deletion can be similarly analyzed. First, consider the probability of adding $h(v)$ to the similarity index of u, which should be k/d_u. After adding $h(v)$ to the similarity index of u, we need to scan all the neighborhood of u and recompute the approximate similarity $\hat{\sigma}(u, w)$ between u and w. This takes a cost of $O(d_u \cdot k)$ cost in total. When the similarity index of u is not changed, then there is no need to re-calculate the similarity between u and the neighborhood of u.

After updating the similarity index and the approximate similarity scores, we further update the neighboring index of w for $w \in N[u]$ if $S_k(u)$ is changed. This can be finished with $O(\log d_w)$ time. To update the bucket index, for each w in $N[u]$, we update the bucket index for each w at most δ times and each time it takes at most $O(\log n)$ cost as the size of bucket-index is $O(n)$ and we use a binary search tree to maintain each bucket index. So the total time to update the bucket index is bounded by $O(d_u \cdot \delta \cdot \log n)$. To summarize, with k/d_u probability, it incurs $d_u \cdot (k+\delta \cdot \log n)$ running cost. When $S_k(u)$ is not modified, we only need to estimate the similarity between u and v, insert $\langle v, \hat{\sigma}(u, v)\rangle$ into $NO[u]$, and modify the bucket index related to u. The time complexity of this part can be bounded by $O(k+\delta \log n)$. Combining these together, we have the expected update cost to be:

$$\begin{aligned}&\frac{k}{d_u} \cdot d_u(k + \delta \cdot \log n) + (1 - \frac{k}{d_u})(k + \delta \cdot \log n)\\ =&O\left(k \cdot (k + \delta \cdot \log n)\right).\end{aligned}$$

The running cost to update for vertex v will be the same, and thus will not change the time complexity. Taking $k = \frac{1}{2\rho^2} \ln \frac{2}{p_f}$, we derive the final result in Theorem 6. □

Proof of Theorem 7. As analyzed in Section 4.2, we reduce the cost to re-compute the similarity from $O(k)$ to $O(\log k)$. We assume that $k \ll n$ and the expected time complexity is:

$$\frac{k}{d_u} \cdot d_u(\log k + \delta \cdot \log n) + (1 - \frac{k}{d_u})(\log k + \delta \cdot \log n) = O(k \cdot \delta \cdot \log n).$$

Taking $k = \frac{1}{2\rho^2} \ln \frac{2}{p}$, we derive the result in Theorem 7. □

Proof of Theorem 8. According to the disscusions in Section 4.3, we can see that the cost to update a bucket is $O(1)$. Therefore, from the proof of Theorems 6–7, the expected time complexity of updating an edge become:

$$\frac{k}{d_u} \cdot d_u(\log k + \delta) + (1 - \frac{k}{d_u})(\log k + \delta) = O(k \cdot \log k).$$

where δ is a constant number. Taking $k = \frac{1}{2\rho^2} \ln \frac{2}{p}$, we derive the result in Theorem 8. □

TABLE 1: Summary of datasets ($K = 10^3, M = 10^6, B = 10^9$).

Dataset	n	m	$\bar{d}$	$\bar{c}$	$\bar{\sigma}_{max}$
Skitter	1.7M	22.2M	13.1	0.258	0.245
Pokec	1.6M	44.6M	27.3	0.109	0.171
Topcats	1.8M	50.9M	28.4	0.274	0.202
Livejournal (LJ)	4.8M	85.7M	17.7	0.117	0.297
Orkut	3.1M	234.4M	76.3	0.166	0.202
Brain	784.3K	535.7M	683.1	0.486	0.724
PP-Miner (PP)	8.3M	1.8B	223.8	0.341	0.455
Twitter	41.7M	2.4B	57.7	0.073	0.091
Friendster (FS)	65.6M	3.6B	55.1	0.134	0.130
Web	90.3M	3.9B	42.9	0.194	0.178

6 EXPERIMENT

We experimentally evaluate our BOTBIN against the state-of-the-art index-based method GS-Index in terms of efficiency and effectiveness. All experiments are conducted on a Linux machine with an Intel Xeon(R) CPU clocked at 2.30GHz with 768GB memory.

6.1 Experimental Settings

Datasets. We use 10 large real datasets in the experiments. All these datasets are publicly available at SNAP [40], Konect [41] and Network Repository [42]. Among them, Pokec, Orkut, Twitter, Livejournal, and Friendster are social networks; Skitter, Topcats, and Web are web networks; Brain and PP are biological networks where Brain is a gene-gene interaction network and PP is a protein-protein association network. Following previous works [29], [32], directed graphs, like Twitter, are converted to an undirected graph by treating directed edges as undirected ones. The statistics of datasets are shown in Table 1, where $\bar{d}$ is the average degree, $\bar{c}$ is the average clustering coefficient, and $\bar{\sigma}_{max}$ is the average of the maximum similarity between each vertex to its neighbors (Ref. to Section 3.2).

Main Competitor. We mainly compare our BOTBIN against the state-of-the-art index-based method GS-Index [29], as discussed in Section 2.2. The implementation of GS-Index is provided by their inventors. We do not compare against DynStrClu [32] as it cannot support different input parameters of μ and ϵ as we discussed in Section 2.2. To examine the update performance, we further include the solution presented in Section 4.1 as a baseline, dubbed as BOTBIN-Basic. The solution that incorporates both optimization techniques described in Sections 4.2 and 4.3 is denoted as BOTBIN. For the ablation study, we also include a variant of BOTBIN-Basic with only the optimization technique from Section 4.2, referred to as BOTBIN-Lite.

Parameter settings. Recall that all our algorithms include a failure probability parameter p_f. We set the failure probability as $\frac{1}{1000}$. We set $M = m$ and d_E based on each dataset (Ref. to Theorem 9 for their definition), and then we can calculate k for the bottom-k sketch. In addition, BOTBIN includes an error parameter ρ to control the trade-off between the update efficiency and clustering quality. We tune the impact of ρ in Exp 5. The experimental results suggest that when $\rho = 0.1$, it achieves the best trade-off between the update efficiency and clustering quality. For the parameter of bucket size δ (Refer to Section 3 for its definition), we tune δ and examine its impact in Exp 5. We

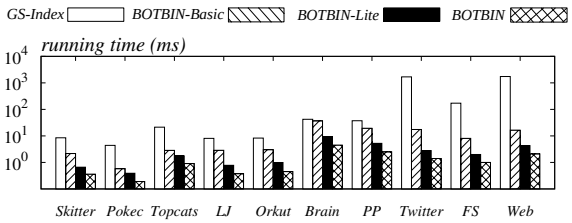


Fig. 4: Index update time with edge insertions.

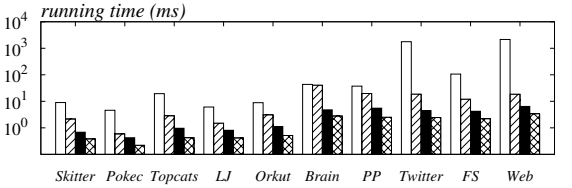


Fig. 5: Index update time with edge deletions.

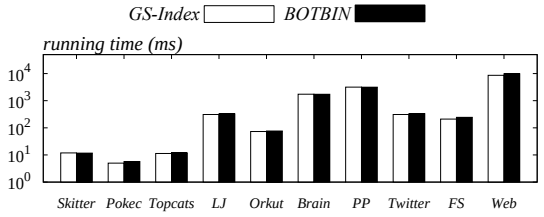


Fig. 6: Query time performance.

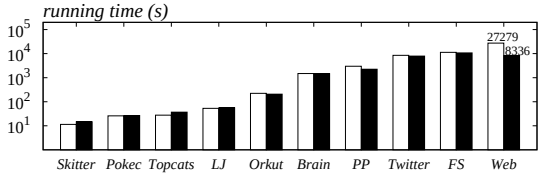


Fig. 7: Index construction time.

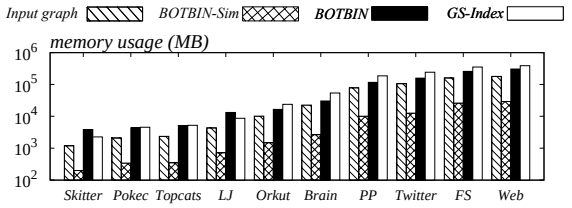


Fig. 8: Memory usage.

observe that when $\delta = 100$, it achieves the best trade-off between the efficiency and accuracy and thus we choose $\delta = 100$ as the default setting in the remaining experiments.

6.2 Experimental Comparisons

Exp 1: Update efficiency. In the first set of experiments, we examine the update performance of our BOTBIN scheme. Firstly, we examine the performance of our index update algorithms for edge insertions. Figure 4 shows the results of the average index update cost for 10,000 random edge insertions. Notice that the y-axis is log-scale. As we can observe, our BOTBIN achieves superb performance and can update the index with milliseconds even on billion edge graphs like PP, Twitter, FS, and Web. Compared to GS-Index, our BOTBIN achieves up to 3 orders of magnitude speed-up on the Twitter dataset and an order of magnitude speedup on most datasets. Next, we compare the performance of BOTBIN-Basic, BOTBIN-Lite, and the fully optimized BOTBIN. Our BOTBIN-Lite achieves a significant speed-up over BOTBIN-Basic as it reduces the cost of re-computing similarity scores from $O(k)$ to $O(\log k)$. However, it is outperformed by BOTBIN, which updates the bucket index more efficiently as described in Section 4.3. In particular, BOTBIN can run $2 \sim 3$x faster than BOTBIN-Lite on most of datasets. Finally, even our non-optimized version, BOTBIN-Basic, outperforms GS-Index on all datasets, and achieves up to an order of magnitude speed-up over the baseline GS-Index on the Twitter dataset.

Next, we examine the performance of the index update algorithms for edge deletions. Figure 5 shows results of average index update costs for 10,000 random edge deletions. The performance gain shows a similar trend as that on edge insertions. We further examine the performance of our index scheme under a mixing workload of insertions and deletions. We find that all algorithms show a stable performance under mixing workloads and is insensitive to the change of workloads. For the interest of space, we delay the results to our technical report [38].

Exp 2: Query efficiency. Finally, we randomly choose 100 queries with different ϵ and μ parameters by varying ϵ in $\{0.2, 0.21, \ldots, 0.80\}$ and μ in $\{2, 3, 4, \ldots, 15\}$, which follows the previous work [29]. Figure 6 shows the average query time. We can see that GS-Index and BOTBIN show identical query performance as both GS-Index and BOTBIN has a query cost linearly depending on the size of the cluster subgraph. The results show that BOTBIN gains a good trade-off among index update cost, query cost, and query accuracy, and is the preferred choice on dynamic graphs.

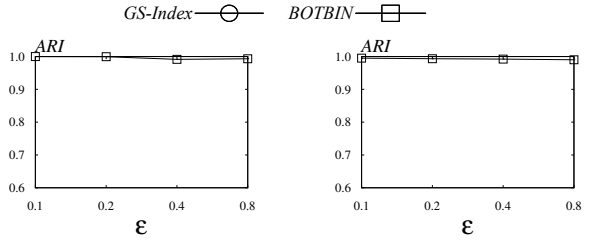
Exp 3: Clustering quality. We evaluate the clustering quality of our approximate algorithm and the baseline algorithm on all datasets. In the first set of experiment, the queries are generated following the settings in Exp 2. Following

TABLE 2: Clustering Quality: Jaccard Similarity.

Dataset	ARI (%)	Precision (%)	Recall (%)	Num of cores
Skitter	99.98	99.94	99.47	11.9K
Pokec	99.43	99.96	99.78	4.0K
Topcats	99.90	100.00	99.73	11.8K
LJ	99.97	100.00	99.90	212.5K
Orkut	99.46	100.00	99.92	64.5K
Brain	99.00	99.60	99.61	524.8K
PP	98.88	99.91	99.92	1.4M
Twitter	98.54	99.89	99.75	86.0K
FS	99.41	99.37	99.43	104.2K
Web	99.16	99.46	99.95	2.6M

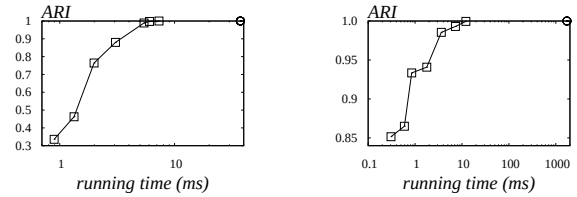
TABLE 3: Clustering Quality: Dice Similarity

Dataset	ARI (%)	Precision (%)	Recall (%)	Num of cores
Skitter	99.87	99.97	99.95	43.1K
Pokec	99.99	99.99	99.99	35.9K
Topcats	99.98	99.99	99.99	44.3K
LJ	99.92	100.00	99.99	458.9K
Orkut	99.97	99.99	99.99	305.1K
Brain	99.32	99.94	99.94	652.0K
PP	99.41	99.95	99.80	2.6M
Twitter	98.94	99.88	99.92	306.7K
FS	99.96	99.98	99.99	633.8K
Web	99.24	99.98	99.97	3.9M



(a) PP (b) Twitter

Fig. 9: Impact of ϵ: ARI of clustering results by BOTBIN



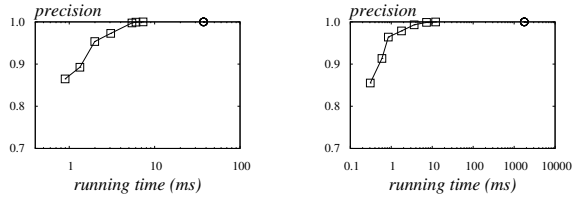
(a) PP (b) Twitter

Fig. 10: Impact of ρ: ARI and update time by BOTBIN.

previous work [32], our evaluation uses three metrics: *precision*, *recall*, and *adjusted rand index* (ARI). The precision and recall here are defined based on core-vertex. We use the baseline algorithm to get the ground truth of the core vertex, and then we test the recall and precision of the core vertices found by the approximate algorithm. For ARI, it is a popular metric to evaluate the clustering quality of outputted clustering results under the ground-truth of known clustering results. The higher (up to 1) the ARI score is, the better the clustering results are. Table 2 reports the precision, recall, and ARI of the clustering result returned by BOTBIN, when the set similarity is measured by Jaccard Similarity. We further report the average number of core vertices per query as a reference. As we can observe, BOTBIN achieves close to 100% accuracy, recall, and ARI on all datasets. Recap that in Section 3.3, we extend our BOTBIN to the case that the similarity of two sets is measured by Dice Similarity. We further conduct experiments to evaluate this extension. As reported in Table 3, we can see that our BOTBIN can also yield a high-quality clustering output in the case of Dice Similarity, compared to the ground truth clustering obtained by the base algorithm.

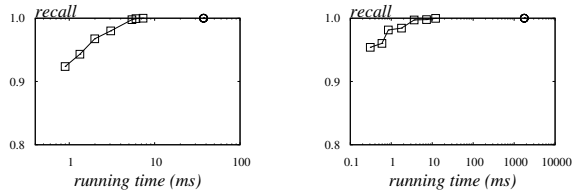
In the second set of experiment, we examine the impact of ϵ on the quality of the clustering results returned by BOTBIN by varying ϵ in $\{0.1, 0.2, 0.4, 0.8\}$. To examine clustering results, we report the average ARI score with different choice of μ from 2 to 15. Figure 9 reports ARI scores when we vary ϵ on two large datasets PP and Twitter. As we can observe, our BOTBIN still provides almost identical clustering result as that of the exact SCAN algorithm. This shows the effectiveness of our approximate solution.

Exp 4: Indexing Costs. Next, we compare the indexing cost of BOTBIN against GS-Index. Figure 7 reports the index construction time of both methods. It can be observed that our BOTBIN can be up to 3.3x faster than GS-Index on index construction. To explain, our BOTBIN has an improved time complexity over GS-Index on index construction. Thus, on huge graphs, BOTBIN tends to be more scalable than GS-Index. In addition, the index construction cost is almost 10^6 times as that of the index update. This further demonstrates



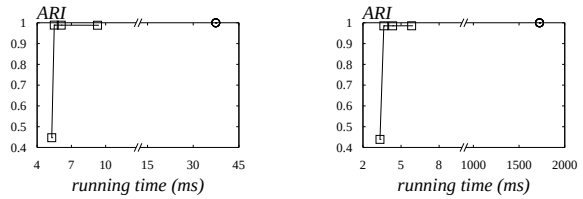
(a) PP (b) Twitter

Fig. 11: Impact of ρ: precision and update time by BOTBIN



(a) PP (b) Twitter

Fig. 12: Impact of ρ: recall and update time by BOTBIN



(a) PP (b) Twitter

Fig. 13: Impact of δ: ARI and update time by BOTBIN.

the effectiveness of our proposed index update scheme. Figure 8 reports the memory usage of the built indices. We have reported the size of the similarity index, dubbed as BOTBIN-Sim, and the size with both similarity index and clustering index, dubbed as BOTBIN. We further include the memory cost of the input graph as a reference. Note that to maintain the input graph, we need to use dynamic data structures such as BST or hash table (we used hash table) to store the neighboring vertices of each node. Thus, the memory consumption is generally higher than that of the compressed sparse array design on static graphs. The results show that the major space cost of BOTBIN comes from the clustering index. The results also show that BOTBIN takes less space cost compared to GS-Index on large graphs since $\delta \cdot \bar{\sigma}_{max}$ is generally smaller than the average degree of the input graph when $\delta = 100$.

Exp 5: Tuning parameters. Finally, we examine the impact of ρ (error parameter) and δ (bucket number). Due to space limit, we only show results on two representative datasets: PP and Twitter. To examine the impact of ρ, we vary ρ in $\{0.05, 0.075, 0.1, 0.2, 0.25, 0.4, 0.5\}$ and see the trade-off between index update time and the query accuracy in terms of ARI as shown in Figure 10. Notice that the smaller ρ it is, the higher the update cost it is while the more accurate the clustering results are. We observe that when $\rho = 0.1$, it achieves a good trade-off between the update cost and clustering quality.We further examine the impact of ρ to precision (resp. recall) as shown in Figure 11 (resp. Figure 12). It shows a similar trend and when $\rho = 0.1$, it strikes a good balance between update time and clustering quality. Thus, we set $\rho = 0.1$ as the default settings. Next, we examine the impact of δ. Figure 13 shows the results when

we vary δ with $\{50, 100, 200, 400\}$. Notice that the larger δ is, the more accurate the clustering results are while the higher update cost it incurs. The results show that when we set $\delta = 100$, it achieves almost 100% ARI score. Thus, we set $\delta = 100$ as the default setting.

7 CONCLUSION

In this paper, we present an efficient and effective index scheme for dynamic structural graph clustering, verified with extensive experiments. For further work, we plan to investigate how to devise efficient estimation schemes for weighted similarity measures like Cosine and weighted Jaccard similarities on weighted graphs so as to support efficient index updates.

REFERENCES

[1] R. Guimera and L. A. N. Amaral, "Functional cartography of complex metabolic networks," *Nature*, vol. 433, no. 7028, pp. 895–900, 2005.

[2] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3-5, pp. 75–174, 2010.

[3] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities," *PVLDB*, vol. 2, no. 1, pp. 718–729, 2009.

[4] X. Xu, N. Yuruk, Z. Feng, and T. A. Schweiger, "Scan: a structural clustering algorithm for networks," in *SIGKDD*, 2007, pp. 824–833.

[5] Y. Ding, M. Chen, Z. Liu, Y. Ye, M. Zhang, R. Kelly, L. Guo, Z. Su, S. Harris, F. Qian, W. Ge, H. Fang, X. Xu, and W. Tong, "atbionet–an integrated network analysis tool for genomics and biomarker discovery," *BMC genomics*, vol. 13, p. 325, 2012.

[6] Z. Liu, Q. Shi, R. Kelly, H. Fang, and W. Tong, "Translating Clinical Findings into Knowledge in Drug Safety Evaluation - Drug Induced Liver Injury Prediction System (DILIps)," *PLOS computational biology*, vol. 7, 12, 2011.

[7] V.-S. Martha, Z. Liu, L. Guo, Z. Su, Y. Ye, H. Fang, D. Ding, W. Tong, and X. Xu, "Constructing a robust protein-protein interaction network by integrating multiple public databases," in *BMC bioinformatics*, vol. 12, no. 10, 2011, pp. 1–10.

[8] M. Mete, F. Tang, X. Xu, and N. Yuruk, "A structural approach for finding functional modules from large biological networks," in *BMC Bioinformatics*, vol. 9, no. 9, 2008, pp. 1–14.

[9] C. X. Lin, Y. Yu, J. Han, and B. Liu, "Hierarchical web-page clustering via in-page and cross-page link structures," in *PAKDD*, 2010, pp. 222–229.

[10] S. Papadopoulos, Y. Kompatsiaris, and A. Vakali, "Leveraging collective intelligence through community detection in tag networks," in *CKCaR*, 2009.

[11] ——, "A graph-based clustering scheme for identifying related tags in folksonomies," in *DaWaK*, 2010, pp. 65–76.

[12] S. Papadopoulos, C. Zigkolis, G. Tolias, Y. Kalantidis, P. Mylonas, Y. Kompatsiaris, and A. Vakali, "Image clustering through community detection on hybrid image similarity graphs," in *ICIP*, 2010, pp. 2353–2356.

[13] M. Schinas, S. Papadopoulos, Y. Kompatsiaris, and P. A. Mitkas, "Visual event summarization on social media using topic modelling and graph-based ranking algorithms," in *ICMR*, 2015, pp. 203–210.

[14] M. Schinas, S. Papadopoulos, G. Petkos, Y. Kompatsiaris, and P. A. Mitkas, "Multimodal graph-based event detection and summarization in social media streams," in *ACM Multimedia*, 2015, pp. 189–192.

[15] S. Lim, S. Ryu, S. Kwon, K. Jung, and J. Lee, "LinkSCAN*: Overlapping community detection using the link-space transformation," in *ICDE*, 2014, pp. 292–303.

[16] S. Papadopoulos, Y. Kompatsiaris, A. Vakali, and P. Spyridonos, "Community detection in social media - performance and application considerations," *Data Min. Knowl. Discov.*, vol. 24, no. 3, pp. 515–554, 2012.

[17] S. Papadopoulos, C. Zigkolis, Y. Kompatsiaris, and A. Vakali, "Cluster-based landmark and event detection for tagged photo collections," *IEEE MultiMedia*, vol. 18, no. 1, pp. 52–63, 2011.

[18] S. S. Chawathe, "Clustering blockchain data," in *Clustering Methods for Big Data Analytics*, 2019, pp. 43–72.

[19] L. Chang, W. Li, L. Qin, W. Zhang, and S. Yang, "pscan: Fast and exact structural graph clustering," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 2, pp. 387–401, 2017.

[20] T. Tseng, L. Dhulipala, and J. Shun, "Parallel index-based structural graph clustering and its approximation," in *SIGMOD*, 2021, pp. 1851–1864.

[21] H. Shiokawa, Y. Fujiwara, and M. Onizuka, "SCAN++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs," *PVLDB*, vol. 8, no. 11, pp. 1178–1189, 2015.

[22] Y. Che, S. Sun, and Q. Luo, "Parallelizing pruning-based graph structural clustering," in *ICPP*, 2018, pp. 77:1–77:10.

[23] J. Chen, J. Chen, J. Liu, and V. Huang, "PSCAN: A parallel structural clustering algorithm for networks," in *ICMLC*, 2013, pp. 839–844.

[24] S. T. Mai, S. Amer-Yahia, I. Assent, M. S. Birk, M. S. Dieu, J. Jacobsen, and J. Kristensen, "Scalable interactive dynamic graph clustering on multicore cpus," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 7, pp. 1239–1252, 2019.

[25] T. R. Stovall, S. Kockara, and R. Avci, "GPUSCAN: gpu-based parallel structural clustering algorithm for networks," *IEEE Trans. Parallel Distributed Syst.*, vol. 26, no. 12, pp. 3381–3393, 2015.

[26] T. Takahashi, H. Shiokawa, and H. Kitagawa, "SCAN-XP: parallel structural graph clustering algorithm on intel xeon phi coprocessors," in *NDA@SIGMOD*, 2017, pp. 6:1–6:7.

[27] W. Zhao, V. S. Martha, and X. Xu, "PSCAN: A parallel structural clustering algorithm for big networks in mapreduce," in *AINA*, 2013, pp. 862–869.

[28] Q. Zhou and J. Wang, "Sparkscan: A structure similarity clustering algorithm on spark," in *BDTA*, 2016, pp. 163–177.

[29] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Efficient structural graph clustering: an index-based approach," *VLDB J.*, vol. 28, no. 3, pp. 377–399, 2019.

[30] J. Huang, H. Sun, Q. Song, H. Deng, and J. Han, "Revealing density-based clustering structure from the core-connected tree of a network," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 8, pp. 1876–1889, 2013.

[31] D. Bortner and J. Han, "Progressive clustering of networks using structure-connected order of traversal," in *ICDE*, 2010, pp. 653–656.

[32] B. Ruan, J. Gan, H. Wu, and A. Wirth, "Dynamic structural clustering on graphs," in *SIGMOD*, 2021, pp. 1491–1503.

[33] F. Zhang and S. Wang, "Effective indexing for dynamic structural graph clustering," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2908–2920, 2022.

[34] N. Yuruk, X. Xu, and T. A. J. Schweiger, "On structural analysis of large networks," in *HICSS*, 2008, p. 143.

[35] G. Hou, Q. Guo, F. Zhang, S. Wang, and Z. Wei, "Personalized pagerank on evolving graphs with an incremental index-update scheme," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 25:1–25:26, 2023.

[36] E. Cohen and H. Kaplan, "Summarizing data using bottom-k sketches," in *PODC*, 2007, pp. 225–234.

[37] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup, "Fast similarity sketching," in *FOCS*, 2017, pp. 663–671.

[38] "Technical report," https://github.com/zzzzzfy/BOTBIN, 2024.

[39] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, 2005.

[40] "SNAP Datasets," http://snap.stanford.edu/data, 2014.

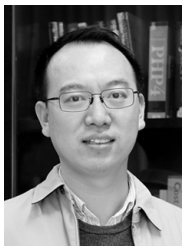
[41] "KONECT," http://konect.cc/networks/, 2013.

[42] "Network Repository," https://networkrepository.com, 2015.

Fangyuan Zhang is currently a Ph.D. candidate at Database Group of the Chinese University of Hong Kong advised by Professor Sibo WANG. Previously, he obtained his bachelor degree from ACM Class at Shanghai Jiao Tong University in 2021. His research interests include algorithms for big data analytics and processing.

Qintian Guo is currently a post-doctoral fellow in the Department of Computer Science and Engineering at The Hong Kong University of Science and Technology. Before that, he received his Ph.D. degree from the Chinese University of Hong Kong, and bachelor degree from the University of Science and Technology of China. His research interests include graph data management and graph mining.

Junhao Gan is a senior lecturer in School of Computing and Information Systems (CIS) at The University of Melbourne (UoM). Before joining the UoM, he was a post-doctoral research fellow in School of Information Technology and Electrical Engineering (ITEE) at The University of Queensland (UQ) from April 2017 to July 2018. He received his PhD degree in the same school at UQ in 2017. His research interests include practical algorithms with theoretical guarantees for solving problems on massive data.



Sibo Wang is an Assistant Professor in the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong. He received his B.E. in Software Engineering in 2011 from Fudan University and his Ph.D. in Computer Science in 2016 from Nanyang Technological University. He is currently interested in graph data management, big data analysis, especially social network analysis, and efficient algorithms with indexing and approximation.