Efficient Structural Graph Clustering: An Index-Based Approach

Dong Wen[♮], Lu Qin[♮], Ying Zhang[♮], Lijun Chang[§], and Xuemin Lin[‡],
[♮]Centre for Artificial Intelligence, University of Technology Sydney, Australia
[§]The University of Sydney, Australia
[‡]The University of New South Wales, Australia
[♮]dong.wen@student.uts.edu.au; {lu.qin, ying.zhang}@uts.edu.au;
[§]lijun.chang@sydney.edu.au; [‡]lxue@cse.unsw.edu.au;

ABSTRACT

Graph clustering is a fundamental problem widely experienced across many industries. The structural graph clustering (SCAN) method obtains not only clusters but also hubs and outliers. However, the clustering results closely depend on two sensitive parameters, ϵ and μ, while the optimal parameter setting depends on different graph properties and various user requirements. Moreover, all existing SCAN solutions need to scan at least the whole graph, even if only a small number of vertices belong to clusters. In this paper we propose an index-based method for SCAN. Based on our index, we cluster the graph for any ϵ and μ in $O(\sum_{C \in \mathbb{C}} |E_C|)$ time, where $\mathbb{C}$ is the result set of all clusters and $|E_C|$ is the number of edges in a specific cluster C. In other words, the time expended to compute structural clustering depends only on the result size, not on the size of the original graph. Our index's space complexity is bounded by $O(m)$, where m is the number of edges in the graph. To handle dynamic graph updates, we propose algorithms and several optimization techniques for maintaining our index. We conduct extensive experiments to practically evaluate the performance of all our proposed algorithms on 10 real-world networks, one of which contains more than 1 billion edges. The experimental results demonstrate that our approaches significantly outperform existing solutions.

PVLDB Reference Format:
Dong Wen, Lu Qin, Ying Zhang, Lijun Chang and Xuemin Lin. Efficient Structural Graph Clustering: An Index-Based Approach. *PVLDB*, 11(3): 243 - 255, 2017.
DOI: 10.14778/3157794.3157795

1. INTRODUCTION

Graphs are widely used for representing the relationships across countless interests. A proliferation in graph-building applications has steered research efforts toward challenges in managing and analyzing graph data; those efforts identify graph clustering as a fundamental problem. This has led to extensive studying of graph clustering [25, 13, 19, 2, 8, 17, 16].

A graph cluster is a group of vertices that are densely connected within a group and sparsely connected to vertices outside that group.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.

Proceedings of the VLDB Endowment, Vol. 11, No. 3

Copyright 2017 VLDB Endowment 2150-8097/17/11... $ 10.00.

DOI: 10.14778/3157794.3157795

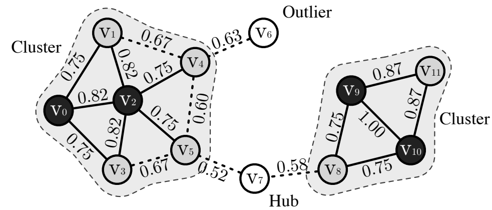


Figure 1: Clusters, hubs and outliers under $\epsilon = 0.7, \mu = 4$

Graph clustering mainly aims to detect all clusters in a given graph. Such analysis is required for metabolic networks [8], identifying communities that are populating social networks [7], creating relevant search results for web crawlers, and identifying research groups across collaborating networks [27], among others.

While detecting all clusters is important, also worthwhile is identifying the specific role—either hub or outlier—of each vertex that is not a member of any cluster. Hubs are vertices that bridge different clusters, and outliers are vertices that do not. Distinguishing between hubs and outliers is important for mining various complex networks [25, 11]. Normally, hubs are regarded as influential vertices, and outliers are treated as noise.

Structural Graph Clustering. Many different graph clustering methods are proposed in the literature. They include the modularity-based methods [15, 18, 15], graph partitioning [17, 6, 23] and the density-based methods [10]. However, most of these clustering methods only focus on computing clusters, ignoring identification of hubs and outliers. To handle this issue, a *structural graph clustering* (SCAN) method is proposed in [25]. Its basic premise is that two vertices belong to the same cluster if they are similar enough.

SCAN defines the *structural similarity* between adjacent vertices, with two vertices being considered similar if their structural similarity is not less than a given parameter ϵ. To construct clusters, SCAN detects a special kind of vertex, named *core*. A core is a vertex that is neighbored closely by many similar vertices and is regarded as the seed of a cluster. The number of similar neighbors for a core is evaluated by a given parameter μ. A cluster is constructed by a core expanding to all vertices that are structurally similar to that core. SCAN successfully finds all these clusters, and also the hubs and outliers. If a vertex does not belong to any cluster, it is a hub if its neighbors belong to more than one cluster, and an outlier otherwise. Fig. 1 gives an example of graph clustering. In it, two clusters are colored gray, and cores are colored black. Hubs and outliers are also labeled.

Existing Solutions. The effectiveness of structural graph clustering across many applications means that plenty of related re-

searches has been proposed. The original algorithm for SCAN is proposed in [25]; for the ease of presentation, we also use the generic term SCAN to represent this algorithm. It iteratively processes each vertex u that has not been assigned to any cluster. If u is a core, it creates a cluster containing u and recursively adds the similar vertices for cores into the cluster. However, the algorithm needs to compute the structural similarity for every pair of adjacent vertices; this requires high computational cost and does not scalable to large graphs. Several methods are proposed for overcoming this drawback. In one method, SCAN++ [19] defines the vertex set containing only vertices that are two hops away from a given vertex. This helps SCAN++ compute fewer structural similarities than SCAN. LinkSCAN* [13] improves the efficiency of SCAN via sampling edges and obtains an approximate result of SCAN.

The state-of-the-art solution for improving SCAN's algorithm efficiency is pSCAN [2]. Under this method, identification of all cores is the key to structural graph clustering. To reduce the number of similarity computations, pSCAN maintains an upper bound (*effective-degree*) and a lower bound (*similar-degree*) for the number of similar neighbors of each vertex. pSCAN always processes the vertex that has the largest effective-degree, which means that vertex has a high probability of being a core. pSCAN avoids a large number of similarity computations and is significantly faster than previous methods.

Several other methods address the problem of structural graph clustering across different computational environments. A parallel version of SCAN is studied in [26]; the algorithm for SCAN on multicore CPUs is studied in [14]. More details about related work are summarized in Section 7.

Motivation. Even though these various SCAN methods can successfully compute clusters, hubs, and outliers, several challenges remain:

- *Parameters Tuning.* The results heavily depend on two sensitive input parameters, μ and ϵ, and the optimal parameter setting is dependent on different graph properties and user requirements. To obtain reasonable clusters, users may need to run an algorithm several times to tune the parameters. Therefore, efficiently computing the clustering given to each parameter setting is a critical factor.
- *Query Efficiency.* pSCAN proposes several pruning rules to improve efficiency. However, it still needs to compute the structural similarity for every pair of adjacent vertices in the worst-case scenario. At a minimum, it needs to process all of a graph's vertices and their adjacent edges to obtain the clusters, even if there exist only a small number of vertices belonging to clusters.
- *Network Update.* Many real-world networks are frequently updated. The clustering results may change when an edge is inserted or removed. Solutions for structural graph clustering should consider handling dynamic graphs.

Our Solution. Motivated by the above challenges, in this paper, we propose a novel index structure, named GS*-Index, for structural graph clustering. GS*-Index has two main parts: *core-orders* and *neighbor-orders*. Given parameters ϵ and μ, we can easily obtain all cores with the help of core-orders, while neighbor-orders help us group all cores and add non-core vertices into each cluster. The space complexity of GS*-Index is bounded by $O(m)$ and the time complexity to construct GS*-Index is bounded by $O((m + \alpha) \cdot \log n)$, where n is the number of vertices, m is the number of edges and α is graph arboricity [5].

Based on GS*-Index, we propose an efficient algorithm to answer the query for any possible ϵ and μ. We compute all clusters in $O(\sum_{C\in\mathbb{C}} |E_C|)$ time complexity, where $\mathbb{C}$ is the result set of all clusters and $|E_C|$ is the number of edges in a specific cluster C in $\mathbb{C}$. In other words, the running time of our algorithm is only dependent on the result size, and not on the size of the original graph.

Most real-world networks are frequently updated. Thus we provide algorithms for updating GS*-Index when an edge inserts or is removed. We further propose techniques for improving the efficiency of updating both neighbor-orders and core-orders.

Contributions. This paper's main contributions in response to the SCAN challenges are as follows:

- *The first index-based algorithm for structural graph clustering.* We propose an effective and flexible index structure, named GS*-Index, for structural graph clustering. To the best of our knowledge, this is the first index-based solution for the structural graph clustering problem. The size of GS*-Index can be well bounded by $O(m)$.
- *Efficient query processing.* Based on our GS*-Index, we propose an efficient algorithm for answering the query for any possible ϵ and μ. The time complexity is linear to the number of edges in the resulting clusters.
- *Optimized algorithms for index maintenance.* We propose algorithms for maintaining our proposed index when graphs update. Several optimizations are proposed for achieving significant algorithmic speedup.
- *Extensive performance studies in real-world networks.* We do extensive experiments for all our proposed algorithms in 10 real-world datasets, one of which contains more than 1 billion edges. The results demonstrate that our algorithm can achieve several orders of magnitude speedup in query processing compared to the state-of-the-art algorithm.

Outline. The rest of this paper is organized as follows. Section 2 gives preliminary definitions and formally defines the problem. Section 3 reviews existing solutions. Section 4 introduces our index structure and proposes the query algorithm. Section 5 describes the algorithms for maintaining index when graph updates occur. Section 6 is a practical evaluation of our proposed algorithms and reports the experimental results. Section 7 summarizes related works, and Section 8 concludes the paper.

2. PRELIMINARY

In this paper, we consider an undirected and unweighted graph $G(V, E)$. We denote the number of vertices $|V|$ and the number of edges $|E|$ by n and m respectively. For each vertex u, the open neighborhood of u, denoted by $N(u)$, is the set of neighbors of u, i.e., $N(u) = \{v \in V | (u, v) \in E\}$. Before stating the problem, we introduce basic definitions for the graph clustering method SCAN.

DEFINITION 1. (STRUCTURAL NEIGHBORHOOD) *The structural neighborhood of a vertex u, denoted by $N[u]$, is defined as $N[u] = \{v \in V | (u, v) \in E\} \cup \{u\}$.*

Given a vertex u, we define the degree of u as the cardinality of $N[u]$, i.e., $deg[u] = |N[u]|$. Based on the concept of *structural neighborhood*, the *structural similarity*[25] is defined as follows.

DEFINITION 2. (STRUCTURAL SIMILARITY) *The structural similarity between two vertices u and v, denoted by $\sigma(u, v)$, is defined as the number of common structural neighbors between u and v, normalized by the geometric mean of their cardinalities of the structural neighborhood. That is,*

$$\sigma(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{|N[u]||N[v]|}} \quad (1)$$

From the definition, we can see that the structural similarity between two vertices becomes large when they share many common

structural neighbors. Intuitively, it is highly possible that two vertices belong to the same cluster if their structural similarity is large. In SCAN, a parameter ϵ is used as a threshold for the similarity value. Given a vertex u and a parameter ϵ ($0 < \epsilon \leq 1$), the *ϵ-neighborhood*[25] for u is defined as follows.

DEFINITION 3. (*ϵ-neighborhood*) *The ϵ-neighborhood for a vertex u, denoted by $N_\epsilon[u]$, is defined as the subset of $N[u]$, in which every vertex v satisfies $\sigma(u, v) \geq \epsilon$. That is, $N_\epsilon[u] = \{v \in N[u] | \sigma(u, v) \geq \epsilon\}$.*

Note that the ϵ-neighborhood for a given vertex u includes u itself, since $\sigma(u, v) = 1$; when the number of ϵ-neighbors is large enough, we call u a *core*. The formal definition of a *core* is given below. A parameter μ is used as the threshold for the cardinality of ϵ-neighborhood.

DEFINITION 4. (CORE) *Given a similarity threshold ϵ ($0 < \epsilon \leq 1$) and an integer μ ($\mu \geq 2$), a vertex u is a core if $|N_\epsilon[u]| \geq \mu$.*

A vertex is called a *non-core vertex* if it is not a core. Identifying cores is a crucial task in SCAN, given clusters can be obtained by expanding the cores. Specifically, each core u is considered as a seed vertex, and all of its ϵ-neighbors v belong to the same cluster of u, since u and v are similar enough. Once v is also a core, the seed scope will be expanded and all ϵ-neighbors of v will be added into the cluster as well. To describe such transitive relation, *structural reachability*[25] is defined as follows.

DEFINITION 5. (STRUCTURAL REACHABILITY) *Given two vertices u and v, v is structurally reachable from u if there is a sequence of vertices $v_1, v_2, ...v_l \in V (l \geq 2)$ such that: (i) $v_1 = u, v_l = v$; (ii) for all $1 \leq i \leq l-1$, v_i is core, and $v_{i+1} \in N_\epsilon[v_i]$.*

A cluster is obtained when all structurally reachable vertices from any core vertex are identified. Below, we formally summarize the definition of a *cluster*.

DEFINITION 6. (CLUSTER) *A cluster $C \in V$ is a non-empty subset of V such that:*

- (CONNECTIVITY) *For any two vertices $v_1, v_2 \in C$, there exists a vertex $u \in C$ such that both of v_1 and v_2 are structurally reachable from u.*
- (MAXIMALITY) *For a core $u \in C$, all vertices that are structurally reachable from u are also belong to C.*

EXAMPLE 1. *We give an example of clustering result for the graph G in Fig. 1, where $\epsilon = 0.7, \mu = 4$. The structural similarity for every pair of adjacent vertices is given. For each pair of adjacent vertices, we represent the edge by a solid line if the structural similarity between them is not less than 0.7. Otherwise, we use a dashed line. We have four cores, namely v_0, v_2, v_9, and v_{10}. They are marked with the color black. All vertices that are structurally reachable from cores are marked in gray. We can see that there are two clusters obtained in the graph. In the cluster $\{v_0, v_1, v_2, v_3, v_4, v_5\}$, all inside vertices are structurally reachable from v_2 (connectivity). The vertices that are structurally reachable from v_0 and v_2 are all included in the cluster (maximality).*

Problem Statement. Given a graph $G(V, E)$ and two parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$, in this paper we aim to efficiently compute the set $\mathbb{C}$ of all clusters in G.

Hub and Outlier. SCAN effectively identifies not only clusters but also *hubs* and *outliers*. The definition of *hub* and *outlier* in SCAN is given as follows.

DEFINITION 7. (HUB AND OUTLIER) *Given a vertex u that does not belong to any cluster, u is a hub if it has neighbors belonging to two or more different clusters. Otherwise, u is an outlier.*

Algorithm 1 SCAN [25]

Input: a graph $G(V, E)$ and parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$
Output: the set $\mathbb{C}$ of clusters in G

```
1: for each edge (u, v) ∈ E do compute σ(u, v);
2: ℂ ← ∅;
3: for each unexplored vertices u ∈ V do
4:     C ← {u};
5:     for each unexplored vertices v ∈ C do
6:         mark v as explored;
7:         if |N_ε[v]| ≥ μ then C ← C ∩ N_ε[v];
8:     if |C| > 1 then ℂ ← ℂ ∩ {C};
9: return ℂ;
```

EXAMPLE 2. *In the case of Fig. 1, vertex v_7 is a hub, given it has two neighbors, namely v_5 and v_8, belonging to different clusters. Vertex v_6 is an outlier.*

In this paper, we mainly focus on computing all clusters. Given the set of clusters in G, all hubs and outliers can be linearly obtained in $O(m + n)$ time, according to the definition.

3. EXISTING SOLUTIONS

In this section, we briefly review existing solutions. First, we introduce the original algorithm SCAN [25]. Then we present the state-of-the-art algorithm pSCAN [2].

3.1 SCAN

The original algorithm SCAN is proposed in [25]. For clearness of presentation, we give the pseudocode of SCAN in Algorithm 1, which is a version rearranged by [2] and is equivalent to the original one [25]. The pseudocode is self-explanatory.

The total time complexity of SCAN is $O(\alpha \cdot m)$. Here α is the arboricity of G, which is the minimum number of spanning forests needed to cover all the edges of the graph G and $\alpha \leq \sqrt{m}$ [5]. In line 1, it costs $O(\alpha \cdot m)$ time to compute the structural similarity for each pair of adjacent vertices, which dominates the total time complexity in the algorithm. Given all structural similarities, computing all clusters only needs $O(m)$ time.

3.2 pSCAN

Even though the algorithm SCAN is worst-case optimal [2], it requires a high computational cost to compute structural similarities. To handle this issue, [2] proposes a new algorithm called pSCAN, which is the state-of-the-art solution for this problem. The main idea of pSCAN is based on three observations: 1) The clusters may overlap; 2) The clusters of cores are disjointed; and 3) The clusters of non-core vertices are uniquely determined by cores.

pSCAN first clusters all cores, as every resulting cluster is uniquely identified by the cores inside it. Then, it assigns non-core vertices to corresponding clusters. The pseudocode is given in Algorithm 2.

To reduce unnecessary similarity computations, pSCAN maintains a lower bound $sd(u)$ and an upper bound $ed(u)$ of the number of ϵ-neighbors for each vertex u. Specifically, let $N'[u]$ be the set of neighbors of u such that the structural similarity between u and every $v \in N'[v]$ has been computed. $N'[u]$ is dynamically updated in the algorithm. Accordingly, $sd(u)$ and $ed(u)$ are assigned by $|\{v \in N'[u] | \sigma(u, v) \geq \epsilon\}|$ and $deg[u] - |\{v \in N'[u] | \sigma(u, v) < \epsilon\}|$ respectively. A vertex is a core if $sd(u) \geq \mu$ and is a non-core vertex if $ed(u) < \mu$. $sd(u)$ and $ed(u)$ are initialized by 0 and $deg[u]$ respectively (line 3 and line 4).

In line 6 of Algorithm 2, CheckCore(u) is invoked for confirming whether the given vertex u is a core, and for updating $sd(v)$ and $ed(v)$ for all unexplored neighbors $v \in N[u]$. In line 7, ClusterCore(u) assigns u and each $v \in N[u]$ to the same cluster based on the disjoint-set data structure if v is also a core and

Algorithm 2 pSCAN [2]

Input: a graph $G(V, E)$ and parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$
Output: the set $\mathbb{C}$ of clusters in G

1: initialize a disjoint-set data structure with all u in V;
2: **for each** $u \in V$ **do**
3: $\quad sd(u) \leftarrow 0$;
4: $\quad ed(u) \leftarrow deg[u]$;
5: **for each** $u \in V$ in non-increasing order w.r.t. $ed(u)$ **do**
6: $\quad$ CheckCore(u);
7: $\quad$ **if** $sd(u) \geq \mu$ **then** ClusterCore(u);
8: $\mathbb{C}_c \leftarrow$ the set of subsets of cores in the disjoint-set data structure;
9: ClusterNonCore();
10: **return** $\mathbb{C}$;

$\sigma(u, v) \geq \epsilon$. All clusters with only cores are obtained after line 7. Finally, in line 9, non-core vertices are assigned to corresponding clusters, where necessary.

Several optimization techniques for similarity-checking exist in pSCAN. Given two vertices, u and v, these techniques propose necessary conditions for both $\sigma(u, v) < \epsilon$ and $\sigma(u, v) \geq \epsilon$. Specifically, if $deg[u] < \epsilon^2 \cdot deg[v]$ *or* $deg[v] < \epsilon^2 \cdot deg[u]$, *then* $\sigma(u, v) < \epsilon$; and if $|N[u] \cap N[v]| \geq \lceil \epsilon \cdot \sqrt{deg[u] \cdot deg[v]} \rceil$, then $\sigma(u, v) \geq \epsilon$. This helps the algorithm achieve increased speed when structural similarity checking. In worst-case scenarios, the time complexity of pSCAN is still bounded by $O(\alpha \cdot m)$.

Drawbacks of Existing Solutions. All existing solutions focus on the online-computing of all clusters via two exact given parameters. However, the change of input parameter value may heavily influence on the clustering result, especially in large graphs. We consider an example in Fig. 2. We use a dashed line to circle the clusters obtained by given ϵ and μ. The results may change even though we only slightly adjust ϵ or μ. Additionally, all existing methods always need to scan an entire graph to obtain its result clusters; for big graphs, this may consume a huge chunk of time.

Motivated by this, we propose an index-based method. With the index, we can answer the query for any given $0 < \epsilon \leq 1$ and $\mu \geq 2$ in a time complexity that is proportional to only the size of result subgraphs. To make our solution scalable to big graphs, the index size should be well bounded, and the time cost to index construction should be acceptable. Additionally, to handle the frequent updates in many real-world graphs, we also propose maintenance algorithms for our index structure. We discuss the details of index implementation in the following section.

4. INDEX-BASED ALGORITHMS

A Basic Index Structure. A straightforward idea for our index structure is the maintenance of structural similarity for each pair of adjacent vertices. We name such index by GS-Index. The construction for GS-Index is the same as in line 1 of Algorithm 1. Specifically, to calculate all structural similarities, we need to compute the number of common neighbors for each pair of adjacent vertices. This is equivalent to enumerating all triangles in the graph [5]. The space complexity of GS-Index, and time complexity of GS-Index construction, are summarized in the following lemmas.

LEMMA 1. *The space complexity of* GS-Index *is* $O(m)$ *and the time complexity for constructing* GS-Index *is* $O(\alpha \cdot m)$.

LEMMA 2. *The time complexity of the query algorithm based on* GS-Index *is* $O(m)$.

Given all similarities, the result clusters are easily obtained by scanning the graph following the same procedures as detailed in lines 2–8 in Algorithm 1. We can see that even though the space usage of GS-Index can be well bounded, it still needs to traverse the entire graph to obtain the result clusters in query processing,

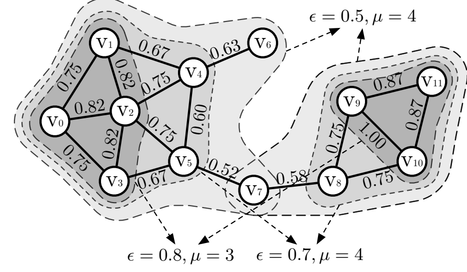


Figure 2: Clusters under different μ and ϵ

and this may be hard to tolerate in big graphs. To address this issue, we propose a novel index structure, namely GS*-Index. With GS*-Index, we can query all result clusters in $O(m_c)$ time, where m_c is the number of edges in the induced subgraphs of the result clusters. We use $O(m)$ space and $O((\alpha + \log n) \cdot m)$ time to save and construct the index, respectively.

We provide an overview of our index structure in Subsection 4.1. In Subsection 4.2, we provide the implementation details for index construction. We propose the query algorithm in Subsection 4.3.

4.1 Index Overview

In this section, we introduce a novel index structure, namely GS*-Index. GS*-Index contains *core-orders* and *neighbor-orders* in addition to the structural similarity for each pair of adjacent vertices. Recall that different clusters may overlap while each core only belongs to a unique cluster. In this section, we first discuss the index for clustering cores. We will also show that our index can be naturally used to cluster together non-core vertices.

4.1.1 Core-Orders: Efficient Core Detection

The general idea underpinning our index is the maintenance of cores for every given ϵ and μ. In other words, with such index, we can efficiently obtain all cores by given any ϵ and μ. We address this problem by first computing all cores of any given similarity threshold ϵ under a specific μ. This is because that the parameter μ cannot be larger than the maximum degree, and there exists only a limited number of possible μ. Next, in the rest of this section, we discuss the candidate vertices nominated as cores within a given μ, and then we consider the precise obtainment of cores within a given ϵ in the candidate set.

The following observation presents the maximum and minimum possible value of input parameter μ.

OBSERVATION 1. *Given a graph G, we have* $2 \leq \mu \leq d_{max}$, *where* $d_{max} = \max(deg[u] | u \in V)$.

If $\mu = 1$, each vertex u would be an isolated result cluster, which is meaningless; there would be no result if $\mu = 0$ or if $\mu > d_{max}$. Additionally, in each given μ, not every vertex has the probability of being a core in a result cluster.

OBSERVATION 2. *Given a graph G and a parameters* $2 \leq \mu \leq d_{max}$, *vertex u cannot be a core in any cluster if* $deg[u] < \mu$.

According to Observation 1 and Observation 2, the set of all candidate vertices that will be cores is $\{u \in V | deg[u] \geq \mu\}$.

Now, given the candidate set for each specific μ, we consider detecting cores by similarity threshold ϵ. Recall that a vertex u is a core if there exist not less than μ ϵ-neighbors. We have the following lemma.

LEMMA 3. *Given a graph G, a parameter* $\mu \geq 2$, *and two similarity thresholds* $0 < \epsilon \leq \epsilon' \leq 1$, *a vertex u is a core in a cluster obtained by* μ *and* ϵ *if it is a core in a cluster obtained by* μ *and* ϵ'.

According to the above lemma, we only need to save the maximum similarity value ϵ for each vertex u that will be a core under each specific μ. We call such a value the *core-threshold* and it is formally defined as follows.

DEFINITION 8. (CORE-THRESHOLD) *Given a parameter μ, the core-threshold for a vertex u, denoted by $\mathcal{CT}_\mu[u]$, is the maximum value of ϵ such that u is a core in clusters obtained by $[\mu, \epsilon]$.*

Given parameters ϵ and μ, we know a vertex u is a core if $\mathcal{CT}_\mu[u] \geq \epsilon$ and u is not a core otherwise. Assume that we have $\mathcal{CT}_\mu[u]$ for all vertices u under every $2 \leq \mu \leq d_{max}$. For any given μ' and ϵ', to obtain all cores, a straightforward method is the checking of $\mathcal{CT}_{\mu'}[u]$ for all vertices u such that $deg[u] \geq \mu'$. However, this costs n times checking, even though there exists only a small number of cores. To reduce unnecessary checking, we give the following lemma.

LEMMA 4. *Given a graph G and two parameters ϵ and μ, a vertex u is a core in result clusters if (i) v is a core; and $(ii)\mathcal{CT}_\mu[v] \leq \mathcal{CT}_\mu[u]$.*

According to Lemma 4, we know that if a given vertex u is a core, all vertices v with $\mathcal{CT}_\mu[v] \geq \mathcal{CT}_\mu[u]$ must be cores. To efficient obtain all cores, we sort the vertices in non-increasing order of their core-thresholds for each μ. We define such order as follows.

DEFINITION 9. (CORE-ORDER) *The core-order for a given parameter μ, denoted by $\mathcal{CO}_\mu$, is a vertex order such that: (i) for all $v \in \mathcal{CO}_\mu, deg[u] \geq \mu$; and (ii) for any two vertices u and v, u appears before v if the core-threshold of u is not smaller than that of v, i.e., $\mathcal{CT}_\mu[u] \geq \mathcal{CT}_\mu[v]$.*

Given parameters μ and ϵ, all cores can be successfully obtained using the core-order. The identification starts from the first vertex in the core-order $\mathcal{CO}_\mu$ and terminates once there is a vertex whose core-threshold is less than the given similarity threshold ϵ.

We compute core-orders for all μ as a part of our index. The space complexity of all core-orders is given as follows:

LEMMA 5. *The space cost of core-orders for all possible μ is bounded by $O(m)$.*

4.1.2 Neighbor-Orders:Efficient Cluster Construction

Core-Threshold Computation. A crucial task in core identification is computing the core-thresholds for each vertex under every possible μ. Based on the definition of cores, we propose following lemma.

LEMMA 6. *Given a parameter μ, the core-threshold of a vertex u ($deg[u] \geq \mu$) is the μ-th largest value in structural similarities between u and its structural neighbors.*

According to Lemma 6, for each vertex u, we compute the structural similarities between u and all neighbors $v \in N[u]$, and sort the neighbors of u in a non-increasing order of their structural similarities. Consequently, the core-thresholds for all possible μ of u are obtained. We define such order as follows.

DEFINITION 10. (NEIGHBOR-ORDER) *The neighbor-order for a given vertex u, denoted by $\mathcal{NO}_u$, is a vertex order such that: (i) for all $v \in \mathcal{NO}_u, v \in N[u]$; and (ii) for any two vertices v_1 and v_2, v_1 appears before v_2 if the structural similarity between u and v_1 is not smaller than that between u and v_2, i.e., $\sigma(u, v_1) \geq \sigma(u, v_2)$.*

Based on Lemma 6 and Definiton 10, given a parameter μ, the core-threshold for a vertex u can be easily obtained by using the neighbor-order $\mathcal{NO}_u$. In addition to obtaining all core-thresholds for each vertex, we can also use neighbor-orders to construct clusters based on obtained cores.

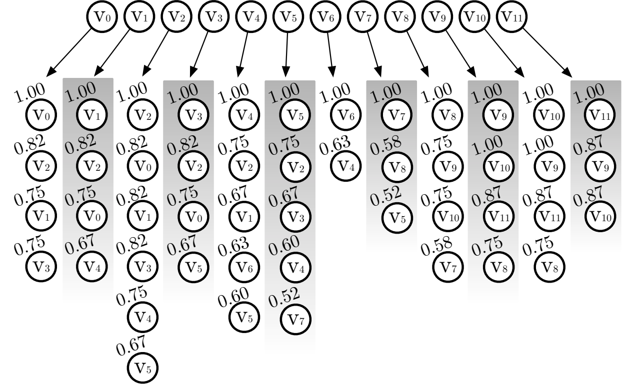


Figure 3: Neighbor-order for each vertex in graph G

Clusters Construction. Thanks to core-orders, we can efficiently obtain all cores. Then, to construct clusters from known cores u, we need to obtain all vertices v that are structurally reachable from u. Recall that the structural reachability transmits between cores only if they are ϵ-neighbor of each other. Thus, the construction of clusters can be solved by computing the ϵ-neighbors of each core.

All ϵ-neighbors of a given vertex u can be efficiently identified via the neighbor-order of u. Identification originates in the neighbor-order's first vertex, and terminates upon identification of a vertex whose structural similarity to u is smaller than the given similarity threshold ϵ. GS*-Index also computes neighbor-orders for all vertex u as a part of GS*-Index. There are $deg[u]$ items in the neighbor-order of each u, and the size of all neighbor-orders can be well bounded, as follows:

LEMMA 7. *The space cost of neighbor-orders for all vertices in the graph is bounded by $O(m)$.*

Based on Lemma 5 and Lemma 7, we have:

THEOREM 1. *The space cost of* GS*-Index *is bounded by $O(m)$.*

4.1.3 An Example of GS-Index*

We continue to use the graph G in Fig. 1 to give the example of our proposed GS*-Index.

EXAMPLE 3. *The neighbor-order for each vertex in G is presented in Fig. 3. For each vertex u, the order is shown vertically and the structural similarity $\sigma(u, v)$ is presented over each neighbor v. Consider vertices v_3. There are four structural neighbors, namely v_3, v_0, v_2 and v_5. We sort the neighbors by their structural similarity to v_3, and obtain the neighbor-order of v_3, which is $[(1.00, v_3),(0.82, v_2), (0.75, v_0),(0.67, v_5)]$. Given $\epsilon = 0.7$, to obtain ϵ-neighbors of vertex v_3, we iteratively check items in the neighbor-order of v_3. Its ϵ-neighbors are v_3, v_2 and v_0. The iteration terminates when checking neighbor v_5, since the structural similarity between v_5 and v_3 is less than 0.7.*

Given the neighbor-order for each vertex u, we obtain the core-threshold of u under any given μ. Assume $\mu = 3$. The core-threshold of v_3 is the structural similarity between v_3 and the third vertex in the neighbor-order of v_3, which is 0.75. That means v_3 is a core if the other given parameter ϵ is not larger than 0.75, i.e., $\epsilon \leq 0.75$. Otherwise, v_3 is not a core.

Based on the neighbor-orders, we compute the core-orders for all possible μ, and they are presented in Fig. 4. For each μ, the order is shown horizontally and the core-threshold $\mathcal{CT}_u$ is presented

Algorithm 3 $\mathsf{GS}^*\text{-Construct}(G)$

Input: a graph $G(V, E)$
Output: GS^*-Index of G

1: **for each** edge $(u, v) \in E$ **do** compute $\sigma(u, v)$;
2: $\mathcal{NO} \leftarrow \emptyset$;
3: **for each** vertex $u \in V$ **do**
4: $\quad \mathcal{NO}_u \leftarrow N[u]$;
5: $\quad$ sort vertices in $\mathcal{NO}_u$ according to Definiton 10;
6: $\quad \mathcal{NO} \leftarrow \mathcal{NO} \cup \{\mathcal{NO}_u\}$;
7: $\mathcal{CO} \leftarrow \emptyset$;
8: **for each** $\mu \in [2, d_{max}]$ **do**
9: $\quad \mathcal{CO}_\mu \leftarrow \{u \in V | deg[u] \geq \mu\}$;
10: $\quad$ sort vertices in $\mathcal{CO}_\mu$ according to Definiton 9;
11: $\quad \mathcal{CO} \leftarrow \mathcal{CO} \cup \{\mathcal{CO}_\mu\}$;
12: **return** $\mathcal{NO}$ and $\mathcal{CO}$;

over u. Assume $\epsilon = 0.7$ and $\mu = 4$. To obtain all cores, we focus on the core-order for $\mu = 4$ and check vertices from left to right iteratively. v_2, v_0, v_9 and v_{10} are cores because their core-thresholds are all not less than 0.7. The iteration terminates when checking vertex v_1, as its core-threshold is less than 0.7.

4.2 Index Construction

We present the index construction algorithm, which is named GS^*-Construct, in this section. The pseudocode of GS^*-Construct is given in Algorithm 3.

Algorithm 3 computes the similarities for every pair of adjacent vertices in line 1, which is equivalent to counting all triangles in the graph. We adopt the same method in [2].

Neighbor-Order Computation. Algorithm 3 computes neighbor-orders for all vertices in lines 2-6. $\mathcal{NO}$ can be implemented as a two-dimensional array that saves the neighbor-order for each vertex. In each neighbor-order $\mathcal{NO}_u$, we sort the structural $v \in N[u]$ in a non-increasing order of $\sigma(u, v)$ (line 5).

Core-Order Computation. Algorithm 3 computes core-orders for all possible μ in lines 7-11. All vertices that have probability of being cores in the current μ are collected in line 9. Each item in $\mathcal{CO}_\mu$ is a vertex id. They are sorted by their core-thresholds under μ. Based on Lemma 6 and Definiton 10, the core-threshold $\mathcal{CT}_\mu[u]$ of a given vertex u is the structural similarity between u and μ-th items in $\mathcal{NO}_u$.

THEOREM 2. *The time complexity of Algorithm 3 is $O((\alpha + \log n) \cdot m)$.*

PROOF. The time complexity for computing all structural similarities (line 1) is $O(\alpha \cdot m)$, and has been discussed previously.

For each vertex u in lines 4–6, the time cost for sorting all neighbors is bounded by $O(deg[u] \cdot \log deg[u])$. Thus the time cost for all vertices is bounded by $O(\sum_{u \in V} deg[u] \cdot \log deg[u])$. Given $\sum_{u \in V} deg[u] = 2m$, we have $\sum_{u \in V} deg[u] \cdot \log deg[u] \leq 2m \cdot \log n$. The time complexity of line 2–6 is bounded by $O(m \cdot \log n)$.

For each μ in lines 9–11, it costs $O(|\mathcal{CO}_\mu| \log |\mathcal{CO}_\mu|)$ time to sort vertices in $\mathcal{CO}_\mu$. The time complexity for all μ in lines 7–11 is $O(\sum_{2 \leq \mu \leq d_{max}} |\mathcal{CO}_\mu| \log |\mathcal{CO}_\mu|)$. Each vertex u totally appears in $deg[u]$ arrays in $\mathcal{CO}$, resulting in $\sum_{2 \leq \mu \leq d_{max}} |\mathcal{CO}_\mu| = 2m$. $|\mathcal{CO}_\mu|$ gradually decreases when increasing μ and $\mathcal{CO}_2 = n$. We have $|\mathcal{CO}_\mu| \leq n$ for any μ. Thus the time complexity of lines 7–11 can be bounded by $O(m \cdot \log n)$.

Summing the time complexity of all three parts above, total time complexity of Algorithm 3 is $O((\alpha + \log n) \cdot m)$. □

Normally, the arboricity α is much greater than $\log n$ especially for big graphs. Thus, we can bound the total time complexity of Algorithm 3 by $O(\alpha \cdot m)$, which is same as that for pSCAN in the worst case.

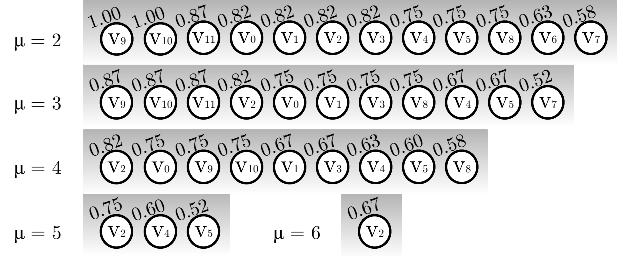


Figure 4: Core-Order for each μ in Graph G

4.3 Query Processing

In this section, we discuss the query processing procedure, named GS^*-Query, which is based on our proposed index GS^*-Index. The general idea of GS^*-Query is iteratively finding an unexplored core u; then computing all vertices v that are structurally reachable from u, and grouping them into a result cluster. Before introducing the details of GS^*-Query, we give the following observation based on $Definiton$ 6.

OBSERVATION 3. *Given a cluster C and any core $u \in C$, all vertices $v \in C$ are structural reachable from u.*

From Observation 3, we know that the cluster obtained from any inside core is complete. The detailed pseudocode is given in Algorithm 4. Note that GS^*-Query no longer needs the original graph G as an input parameter, given the neighborhood information can be obtained via the neighbor-order for each vertex.

To obtain all clusters, Algorithm 4 iteratively processes vertices according the order in $\mathcal{CO}_\mu$. It performs until we find a vertex u whose core-threshold is less than given ϵ (line 4).

Algorithm 4 computes the cluster C containing a core u from line 5–20. This is done by using a queue. For each vertex v in the queue, we add all ϵ-neighbors of v into the cluster and add new discovered cores into the queue (line 11–20). Specifically, we iteratively process each neighbor w of v according to their position in $\mathcal{NO}_v$ (line 10). The iteration terminates once we find a vertex w that is not ϵ-neighbor of v (line 12). The explored neighbors are skipped, as they have been added into the cluster previously (line 13). We add the unexplored neighbors into the cluster in line 15. In line 17, we identify whether w is a core by checking the core-threshold of w, which is the μ-th structural similarity in $\mathcal{NO}_w$. We add w to the queue to detect more structurally reachable vertices if w is a core in line 18. The cluster containing u is added to the result set in line 21. In line 22 we mark all non-core vertices back to unexplored before computing new clusters, given the non-core vertices may overlap between different clusters. A running example of Algorithm 4 is given below.

EXAMPLE 4. *We give an example of Algorithm 4 for $\epsilon = 0.7$, $\mu = 4$ in Fig. 5. All neighbor-orders and core-orders for G in Fig. 1 can be found in Fig. 3 and Fig. 4, respectively. Given $\mu = 4$, Algorithm 4 first focuses on the core-order $\mathcal{CO}_4$, which is shown in the top of Fig. 5. Then, vertices in $\mathcal{CO}_4$ are processed iteratively. We mark in the color black the vertices that are inserted into the queue (line 7 and line 18 of Algorithm 4).*

Vertex v_2 is the first vertex inserted into the queue, given its core-threshold is larger than 0.7. Algorithm 4 then assesses neighbors of v_2 following the neighbor-order $\mathcal{NO}_{v_2}$. The neighbor-order for each vertex in queue is shown around clockwise below the core-order in Fig. 5. v_2 has been explored and thus is skipped. Then, v_0 is added to the cluster and inserted into the queue, as its core-threshold is 0.75. After that, vertices $v_1, v_3 and v_4$ are added to the cluster. Given none of them is core, they are not inserted into

Algorithm 4 GS*-Query

Input: GS*-Index and parameters $0 < \epsilon \leq 1, \mu \geq 2$;
Output: the set $\mathbb{C}$ of clusters in G;

1: $\mathbb{C} \leftarrow \emptyset$;
2: **for each** $u \in \mathcal{CO}_\mu$ **do**
3: $\quad v \leftarrow \mu$-th vertex in $\mathcal{NO}_u$;
4: $\quad$**if** $\sigma(u, v) < \epsilon$ **then break**;
5: $\quad C \leftarrow \{u\}$;
6: $\quad \mathcal{Q} \leftarrow$ initialize an empty queue;
7: $\quad \mathcal{Q}.insert(u)$;
8: $\quad$mark u as explored;
9: $\quad$**while** $\mathcal{Q} \neq \emptyset$ **do**
10: $\quad\quad v \leftarrow \mathcal{Q}.pop()$;
11: $\quad\quad$**for each** $w \in \mathcal{NO}_v$ **do**
12: $\quad\quad\quad$**if** $\sigma(v, w) < \epsilon$ **then break**;
13: $\quad\quad\quad$**if** w is explored **then continue**;
14: $\quad\quad\quad$mark w as explored;
15: $\quad\quad\quad C \leftarrow C \cup \{w\}$;
16: $\quad\quad\quad t \leftarrow \mu$-th vertex in $\mathcal{NO}_w$;
17: $\quad\quad\quad$**if** $\sigma(w, t) \geq \epsilon$ **then**
18: $\quad\quad\quad\quad \mathcal{Q}.insert(w)$;
19: $\quad\quad\quad$**else**
20: $\quad\quad\quad\quad$mark w as *non-core vertex*;
21: $\quad \mathbb{C} \leftarrow \mathbb{C} \cup \{C\}$;
22: $\quad$**for each** non-core vertex v **do** mark v as unexplored;
23: **return** $\mathbb{C}$;

the queue. The iteration terminates after checking v_5, given the structural-similarity between v_5 and v_2 is less than 0.7. Algorithm 4 continues to process v_0 in the queue, and adds all its ϵ-neighbors to the cluster. A cluster is successfully obtained, since no vertex exists in the queue any more.

In the core-order, vertex v_0 is explored, and thus skipped. Algorithm 4 inserts v_9 into the queue and obtains the second cluster. Algorithm 4 terminates when checking v_1 in the core-order, since its core-threshold is less than 0.7. That means v_1 and all following vertices can not be cores.

THEOREM 3. *Given parameters $\mu \geq 2$ and $0 < \epsilon \leq 1$, Algorithm 4 correctly computes all clusters of the graph G.*

PROOF. *(Correctness)* We first prove the correctness for each obtained cluster. Based on Lemma 6 and Definiton 8, Algorithm 4 successfully identifies cores by comparing given ϵ with the μ-th largest similarity value between a given vertex and its structural neighbors (line 4 and line 17). For each core v, the neighbors w are sorted in the non-increasing order of their structural similarities to v in $\mathcal{NO}_v$. Following this order, Algorithm 4 successfully obtains all ϵ-neighbors of v and inserts their cores into the queue. Then it traverses the unexplored ϵ-neighbors of the core in the queue until the queue is empty. That means all vertices that are structurally reachable from the original u in line 3 are obtained. According to Observation 3, we obtained the correct and complete cluster containing u. Note that all non-core vertices are marked back to unexplored. This guarantees we do not lose any non-core vertices in the following cluster computation.

(Completeness) Next, we prove the completeness of the resulting cluster set. Recall that given a parameter μ, vertices are sorted in the non-increasing order of their core threshold under μ in $\mathcal{CO}_\mu$. Based on Definiton 8, Algorithm 4 correctly finds all cores by following this order, and terminates once a non-core vertex is found (line 4). Since a cluster can be correctly obtained by an inside core, all clusters are successfully obtained in Algorithm 4.

(Redundancy Free) Since each core only belongs to a unique cluster, Algorithm 4 only loads ϵ-neighbors for each core, and marks all explored cores. This guarantees no repeated cluster exists in the result set. $\square$

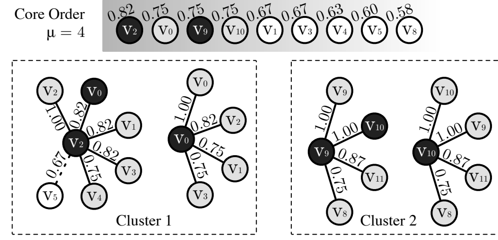


Figure 5: A running example for $\epsilon = 0.7, \mu = 4$

Let E_C be the set of edges in the induced subgraph of cluster C. The time complexity of Algorithm 4 is given as follows.

THEOREM 4. *The time complexity of Algorithm 4 is bounded by $O(\sum_{C \in \mathbb{C}} |E_C|)$.*

PROOF. Let C_{core} be the set of cores in a cluster C. The total number of visited vertices in line 2 is $\sum_{C \in \mathbb{C}} |C_{core}| + 1$. Since all cores that belong to the same cluster are processed in lines 5–20, the total number of explored vertices found in line 2 is $|\mathbb{C}|$.

To obtain each cluster C, only cores are inserted into the queue. Thus, the number of iterations in the while loop (line 9) is $|C_{core}|$. For each core v in line 10, all of its ϵ-neighbors are visited and the number of iterations in the for loop (line 11) is $deg_C[v] + 1$, where $deg_C[v]$ is the degree of v in the induced subgraph of cluster C containing v. Thus the time complexity for computing a cluster (lines 5–20) is $O(\sum_{u \in C_{core}} deg_C[u])$. This can be also represented by $O(|E_C|)$. The total complexity of Algorithm 4 is $O(\sum_{C \in \mathbb{C}} |E_C|)$. $\square$

Local Cluster Query. Our query processing algorithm can also be naturally extended to compute result clusters containing a query vertex u. Given a query vertex u and two parameters $\mu \geq 2$ and $0 < \epsilon \leq 1$, if u is a core, the algorithm is the same as in lines 5–20 of Algorithm 4.

Otherwise, u may be contained in several clusters. In this case, we process ϵ-neighbors v of u by following the order in $\mathcal{NO}_u$. If v is a core, we obtained the cluster containing u and v by similarly invoking lines 5–20 of Algorithm 4. We skip v if it is not a core.

Let $\mathbb{C}$ be the result set of clusters containing query vertex u. The time complexity for each of the two cases detailed above is also bounded by $O(\sum_{C \in \mathbb{C}} |E_C|)$. Note that for the case in which the query vertex is a core, only one cluster C exists in the result set $\mathbb{C}$.

5. INDEX MAINTENANCE

In the previous section we discussed our index-based solution for the graph clustering, namely GS*-Query. Compared to existing online-computing solutions, GS*-Query can correctly and efficiently compute the clusters by given any given parameters μ and ϵ. However, most of real-word graphs are frequently updated. A fixed and outdated index structure can hardly meet the expectations of efficiently clustering large, dynamic graphs. Motivated by this issue, in this section we discuss the algorithms for maintaining our proposed index structure when graphs update. Note that in this paper, we mainly focus on the edge insertion and deletion, as the vertex updates can be handled by performing several edge updates.

5.1 Basic Solutions

In this section we give a basic solution for index maintenance. For the ease of presentation, we mainly discuss edge insertion. The ideas can be easily extended to handle edge removal.

Edge Insertion. Recall that our index structure GS*-Index contains two parts: neighbor-orders and core-orders. We give the following observation to reveal the relationship between them.

Algorithm 5 GS*-Insert

Input: an edge (u, v);
Output: updated index structure;

1: insert the edge(u, v) into graph G;
2: $deg[u] \leftarrow deg[u] + 1, deg[v] \leftarrow deg[v] + 1$;
3: UpdateVertex(u);
4: UpdateVertex(v);

5: **Procedure** UpdateVertex(u) :
6: $\mathcal{NO}_u \leftarrow \emptyset$;
7: **for each** vertex $w \in N(u)$ **do**
8: recompute $\sigma(u, w)$;
9: **if** $u \in \mathcal{NO}_w$ **then**
10: update u in $\mathcal{NO}_w$ according to $\sigma(u, w)$;
11: **else**
12: insert u into $\mathcal{NO}_w$ according to $\sigma(u, w)$;
13: UpdateCores(w);
14: insert w into $\mathcal{NO}_u$;
15: UpdateCores(u);

16: **Procedure** UpdateCores(u) :
17: $i \leftarrow 0$;
18: **for each** $v \in \mathcal{NO}_u$ **do**
19: $i \leftarrow i + 1$
20: **if** $v = u$ **then continue**;
21: **if** $u \in \mathcal{CO}_i$ **then**
22: update u in $\mathcal{CO}_i$ according to $\sigma(u, v)$;
23: **else**
24: insert u into $\mathcal{CO}_i$ according to $\sigma(u, v)$;

OBSERVATION 4. *The core-order $\mathcal{CO}_\mu$ for parameter μ does not change if the neighbor-order $\mathcal{NO}_u$ for each vertex $u \in \mathcal{CO}_\mu$ does not change.*

According to Observation 4, when inserting an edge, we first update the neighbor-orders of the corresponding vertices, and then update the core-orders, if necessary. In the neighbor-order for vertex u, the structural similarity between u and each neighbor $v \in N[u]$ is computed in accordance with their common neighbors and degrees. We have the following observation.

OBSERVATION 5. *The neighbor-order of vertex w changes if $\exists u \in N[w]$, and an edge (u, v) inserts into or is removed from u.*

Let (u, v) be an edge inserting into graph G. Following the above observation, we only need to update the neighbor-orders for the following vertices: $u, v, w \in N[u]$, and $w' \in N[v]$. Then we update the position of these vertices in the core-order for each parameter μ, based on their new core-thresholds.

The algorithm for edge insertion, namely GS*-Insert, is given in Algorithm 5. In Algorithm 5, we first physically insert u and v into the neighbor list of each other in line 1, and update their degree in line 2. For the ease of presentation, we call u and v root vertices.

We invoke procedure UpdateVertex for root vertices. To maintain neighbor-orders, we first compute the new structural similarity between u and each $v \in N[u]$ (line 8). When the new structural similarity $\sigma(u, w)$ is found, we correspondingly update the neighbor-order for v correspondingly (line 10). Note that we check the existence for v because when inserting a new edge (u, v), vertex v does not exist in the neighbor-order of u. Since the neighbor-order for w changes, we update the corresponding core order by invoking UpdateCores(w). Given all items in the neighbor-order of root vertex u change, we update the core-orders influenced by u (line 15) after constructing an entire neighbor-order for u.

The procedural details for UpdateCores are also given in Algorithm 5. The first vertex is skipped, since it is the vertex itself (line 20). We update the position of vertex u in each core-order $\mathcal{CO}_i$ by the i-th structural similarity in $\mathcal{NO}_u$. In cases where u is a root vertex, u does not exist in the core-order $\mathcal{CO}_{deg[u]}$. Thus, we

Algorithm 6 GS*-Remove

Input: an edge (u, v);
Output: updated index structure;

1: remove the edge(u, v) from graph G;
2: $deg[u] \leftarrow deg[u] - 1, deg[v] \leftarrow deg[v] - 1$;
3: remove u from order $\mathcal{CO}_{deg[u]}$;
4: UpdateVertex(u);
5: remove v from order $\mathcal{CO}_{deg[v]}$;
6: UpdateVertex(v);

check the existence for vertex u in line 21. Vertex u is inserted into $\mathcal{CO}_{i+1}$ (line 24) if it does not exist.

Given a vertex u, let $\mathbb{N}[u]$ be the set of vertices whose distances to v are not larger than 2, and $E_{\mathbb{N}[u]}$ be the set of edges in the induced subgraph of $\mathbb{N}[u]$.

THEOREM 5. *Given an inserted edge (u, v), the time complexity of Algorithm 5 is bounded by $O((|E_{\mathbb{N}[u]}| + |E_{\mathbb{N}[v]}|) \cdot \log n)$.*

PROOF. In Algorithm 5, line 1 and line 2 cost constant time. The dominating cost is the invocation of UpdateVertex. For each neighbor w in line 7 of Algorithm 5, computing new structural similarity $\sigma(u, w)$ needs $O(deg[w])$ of time. This can be done by using a bitmap to mark all neighbors of u in advance. The algorithm updates the neighbor-order of w in lines 9–12. The update (line 10) or insertion (line 12) can be finished in $O(\log deg[w])$ time in the worst-case scenarios via a self-balancing binary search tree that maintains the order. Similarly, we use such data structures to implement the core-orders. UpdateCores is invoked for a vertex w in line 13. We know that the size of any core-order is not larger than n. Thus the time complexity of line 8 is bounded by $O(deg[w] \cdot \log n)$. In line 14, it costs $O(\log deg[u])$ of time to insert a new vertex w into the neighbor-order of u. Summarizing the above time cost: the time complexity of UpdateVertex is $O(\sum_{w \in N[u]} deg[w] \cdot \log n)$, which can be also represented as $O(|E_{\mathbb{N}[u]}| \cdot \log n)$. The total time complexity of GS*-Insert can be easily bounded by $O((|E_{\mathbb{N}[u]}| + |E_{\mathbb{N}[v]}|) \cdot \log n)$. □

Edge Removal. Following the similar idea for handling edge insertion, the procedure for edge removal is given in Algorithm 6. We invoke the same subroutine UpdateVertex for each vertex. A difference here is that we need to remove the root-vertex from the highest core-order in advance (line 3 and line 5).

THEOREM 6. *Given a removed edge (u, v), the time complexity of Algorithm 6 is bounded by $O((|E_{\mathbb{N}[u]}| + |E_{\mathbb{N}[v]}|) \cdot \log n)$.*

PROOF. The proof is similarity to that for Theorem 5, and thus is purposefully omitted here. □

5.2 Improved Algorithms

In the previous section, we gave a basic solution for maintaining our proposed index. In the procedure UpdateVertex of Algorithm 5, the dominating cost is the recomputing of the structural similarity for each neighbor w in line 8, and updating $deg[w]$ core-orders in line 13. In this section, we discuss several optimizations for reducing the time cost of these two tasks and propose optimized algorithms for the index maintenance.

Avoiding Structural Similarity Recomputation. The key to computing the structural similarity is calculating the number of common neighbors. Our general idea of avoiding recomputing structural similarity is to efficiently maintain the number of common neighbors for each adjacent pair of vertices. Note that storing the number of common neighbors for each pair of adjacent vertices needs $O(m)$ space, and thus the total space complexity can be still bounded by $O(m)$.

Algorithm 7 GS*-Insert*

Input: an edge (u, v);
Output: updated index structure;

1: insert the edge(u, v) into graph G;
2: $deg[u] \leftarrow deg[u] + 1, deg[v] \leftarrow deg[v] + 1$;
3: UpdateAdd(u);
4: UpdateAdd(v);

5: **Procedure** UpdateAdd(u, v) :
6: $\mathcal{NO}_u \leftarrow \emptyset$;
7: $cn \leftarrow 2$;
8: **for each** vertex $w \in N(u)$ **do**
9:    **if** $w = v$ **then continue**;
10:    $\sigma_{old}(u, w) \leftarrow$ existing structural similarity between u and w;
11:    **if** $w \in N(v)$ **then**
12:       $cn \leftarrow cn + 1$;
13:       $\mathcal{CN}(u, w) \leftarrow \mathcal{CN}(u, w) + 1$;
14:    $\sigma(u, w) \leftarrow \mathcal{CN}(u, w)/\sqrt{deg[u] \cdot deg[w]}$;
15:    update u in $\mathcal{NO}_w$ according to $\sigma(u, w)$;
16:    UpdateCores*$(w, \sigma_{old}(u, w), \sigma(u, w))$;
17:    insert w into $\mathcal{NO}_u$ according to $\sigma(u, w)$;
18: $\mathcal{CN}(u, v) \leftarrow cn$;
19: $\sigma(u, v) \leftarrow \mathcal{CN}(u, v)/\sqrt{deg[u] \cdot deg[v]}$;
20: insert v into $\mathcal{NO}_u$ according to $\sigma(u, v)$;
21: UpdateCores*$(u, 1, 0)$;

22: **Procedure** UpdateCores*$(u, \epsilon_l, \epsilon_r)$:
23: **if** $\epsilon_r > \epsilon_l$ **then** $swap(\epsilon_r, \epsilon_l)$;
24: $i \leftarrow 0$;
25: **for each** $v \in \mathcal{NO}_u$ **do**
26:    $i \leftarrow i + 1$;
27:    **if** $\sigma(u, v) > \epsilon_l$ **then continue**;
28:    **if** $\sigma(u, v) < \epsilon_r$ **then break**;
29:    **if** $u \in \mathcal{CO}_i$ **then**
30:       update u in $\mathcal{CO}_i$ according to $\sigma(u, v)$;
31:    **else**
32:       insert u into $\mathcal{CO}_i$ according to $\sigma(u, v)$;

Recall that when inserting or removing an edge (u, v), only the neighbor-orders of structural neighbors of vertex u and v are influenced. We categorize vertices into the following three types:

- (TYPE I) the neighbors of either u or v, i.e., $N(u) \cup N(v) - N(u) \cap N(v)$;
- (TYPE II) the common neighbors of u and v, i.e., $N(u) \cap N(v)$;
- (TYPE III) the edge's terminals, i.e., u and v.

Considering an inserting or removing edge (u, v), u is an type III vertex. For each neighbor w of u, let $\mathcal{CN}_{old}(w, u)$ be the original number of common neighbors between w and u, and $\mathcal{CN}_{new}(w, u)$ be the updated value. We have the following observations.

OBSERVATION 6. *Given an inserted edge (u, v), for an open neighbor $w \in N(u)$, $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u)$ if w is a type I vertex, and $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u) + 1$ if w is a type II vertex.*

OBSERVATION 7. *Given an removed edge (u, v), for an open neighbor $w \in N(u)$, $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u)$ if w is a type I vertex, and $\mathcal{CN}_{new}(w, u) = \mathcal{CN}_{old}(w, u) - 1$ if w is a type II vertex.*

Observation 6 and Observation 7 shows that we can efficiently obtain the new number of common neighbors for type I and type II vertices, and consequently obtain the new structural similarities.

In our basic solution (Algorithm 5), we compute the number of common neighbors between two type III vertices. Note that the type II vertices essentially are their open common neighbors; thus we can count the number of type II neighbors and avoid recomputing the number of common neighbors between two type III vertices. Details are shown in the pseudocodes.

Algorithm 8 GS*-Remove*

Input: an edge (u, v);
Output: updated index structure;

1: remove the edge(u, v) from graph G;
2: $deg[u] \leftarrow deg[u] - 1, deg[v] \leftarrow deg[v] - 1$;
3: UpdateDel(u);
4: UpdateDel(v);

5: **Procedure** UpdateDel(u) :
6: $\mathcal{NO}_u \leftarrow \emptyset$;
7: remove u from order $\mathcal{CO}_{deg[u]}$;
8: **for each** $vertex w \in N(u)$ **do**
9:    $\sigma_{old}(u, w) \leftarrow$ existing similarity between u and w;
10:    **if** $w \in N(v)$ **then**
11:       $\mathcal{CN}(u, w) \leftarrow \mathcal{CN}(u, w) - 1$;
12:    $\sigma(u, w) \leftarrow \mathcal{CN}(u, w)/\sqrt{deg[u] \cdot deg[w]}$;
13:    update u in $\mathcal{NO}_w$ according to $\sigma(u, w)$;
14:    UpdateCores*$(w, \sigma_{old}(u, w), \sigma(u, w))$;
15:    insert w into $\mathcal{NO}_u$ according to $\sigma(u, w)$;
16: UpdateCores*$(u, 1, 0)$;

Accurate Core-Order Updates. Besides avoiding naively recomputing structural similarities, we can also achieve increased speed in core-orders maintenance. Recall that the position of vertex u in the core-order $\mathcal{CO}_\mu$ may change only if its core threshold for μ (the μ-th structural similarity in its neighbor-order $\mathcal{NO}_u$) changes. Given a vertex u and a vertex $v \in N[u]$, assume that structural similarity $\sigma(u, v)$ between u and v changes from sim_{old} to sim_{new}. Let sim_l and sim_r be the smaller value and the larger value respectively in sim_{old} and sim_{new}. We have the following observation.

OBSERVATION 8. *Given a parameter μ, the core-threshold for vertex u changes if the μ-th structural similarity value in $\mathcal{NO}_u$ falls in range $[sim_l, sim_r]$.*

Based on Observation 8, we avoid the operation for updating the core-order if the corresponding core-threshold does not fall into the specified range.

We name the optimized insertion algorithm by GS*-Insert*. The pseudocode for GS*-Insert* is given in Algorithm 7. GS*-Insert* invokes UpdateAdd for each vertex. In the procedure UpdateAdd, cn is used to count the common neighbors between two type III vertices u and v. We check whether w is a type II vertex in line 11. If so, we increase cn and update the number of common neighbors between w and u according to Observation 6. We invoke an optimized procedure UpdateCores* to update core-orders. In the procedure UpdateCores*, we only update core-orders if the structural similarity falls into the range $[\epsilon_r, \epsilon_l]$ (line 27–28).

The pseudocode of our optimized edge removal algorithm, namely GS*-Remove*, is given in Algorithm 8, and it invokes UpdateDel for each vertex, which is similar to UpdateAdd. Based on Observation 7, we decrease the common neighbor between u and w in line 11 when w is a type II vertex. The detailed explanation for UpdateDel is purposefully omitted here.

6. EXPERIMENTS

We conduct extensive experiments to evaluate the performance of our proposed solutions. Algorithms that appeared in the experiments are summarized as follows:

- pSCAN: The state-of-the-art algorithm for structural clustering in [2].
- GS-Construct: The algorithm that constructs the basic GS-Index, which directly computes structural similarity between every pair of adjacent vertices.
- GS*-Construct: The algorithm that constructs our proposed index GS*-Index in Section 4.

Table 1: Network Statistics

Datasets	$|V|$	$|E|$	$\overline{d}$	c
DBLP	317,080	10,498,66	6.62	0.6324
Amazon	403,394	3,387,388	16.79	0.4177
WIKI	2,394,385	5,021,410	4.19	0.0526
Google	875,713	5,105,039	11.66	0.5143
Cit	3,774,768	16,518,948	8.75	0.0757
Pokec	1,632,803	30,622,564	37.51	0.1094
LiveJournal	3,997,962	34,681,189	17.35	0.2843
Orkut	3,072,441	117,185,083	76.28	0.1666
UK-2002	18,520,486	298,113,762	32.19	0.6891
Webbase	118,142,155	1,019,903,190	17.27	0.5533

- GS-Query: The query algorithm based on GS-Index.
- GS*-Query: The query algorithm based on GS*-Index.
- GS*-Insert: The naive algorithm for edge insertion.
- GS*-Insert*: The optimized algorithm for edge insertion.
- GS*-Remove: The naive algorithm for edge removal.
- GS*-Remove*: The optimized algorithm for edge removal.

All algorithms are implemented in C++ and compiled using g++ complier at -O3 optimization level. All the experiments are conducted on a Linux operating system running on a machine with an Intel Xeon 2.8GHz CPU, 256GB 1866MHz DDR3-RAM. The time cost for algorithms is measured as the amount of wall-clock time elapsed during the program execution.

Datasets. We use 10 publicly available real-world networks to evaluate the algorithms. The detailed statistics are shown in Table 1, where the average degree ($\overline{d}$) and average clustering coefficient (c) are shown in the last two columns. The networks are displayed in non-decreasing order regarding the number of edges. All networks and corresponding detailed description can be found in SNAP[1] and Webgraph[2].

6.1 Performance of Query Processing

We compare our best query algorithm GS*-Query with GS-Query and the state-of-the-art algorithm pSCAN for structural clustering in this section.

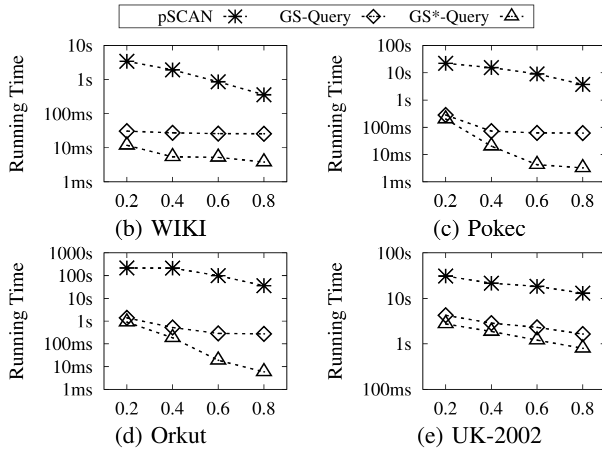


Figure 6: Query time for different ϵ ($\mu = 5$)

We first report the performance of the algorithms by varying μ and ϵ. For parameter settings, we adopt a similar settings that are used in [2] and [19]. For $0 < \epsilon \leq 1$, we choose $0.2, 0.4, 0.6$, and 0.8, with 0.6 as default and for $\mu \geq 2$, we choose $2, 5, 10$, and 15, with 5 as default. Because of space limitations, we only show the charts for WIKI, Pokec, Orkut and UK-2002, and note that the results of the other datasets present similar trends. The results for all datasets under default parameters settings are reported later.

Eval-I: Varying ϵ. The time cost of GS*-Query, GS-Query, and pSCAN by varying ϵ is given in Fig. 6. We can see that GS*-Query is faster than GS-Query under every ϵ on all datasets, and the gap between them gradually increases when ϵ grows. For example, in Orkut, GS*-Query spends about 0.9 seconds while GS-Query spends about 1.4 seconds when ϵ is 0.2. When ϵ reaches 0.8, it costs only 5 milliseconds to perform GS*-Query, while GS-Query still costs almost 0.3 seconds. Additionally, when ϵ grows, lines for GS*-Query present a significant downward trend on all datasets. This is because the time cost of GS*-Query is strictly dependent on the result size. The result size becomes smaller when ϵ increases. The lines for GS-Query perform relatively steadier compared with GS*-Query. Lines for pSCAN on some datasets also present slightly downward trends, reflecting the pruning techniques used. However, GS*-Query is significantly faster than pSCAN especially when ϵ is large.

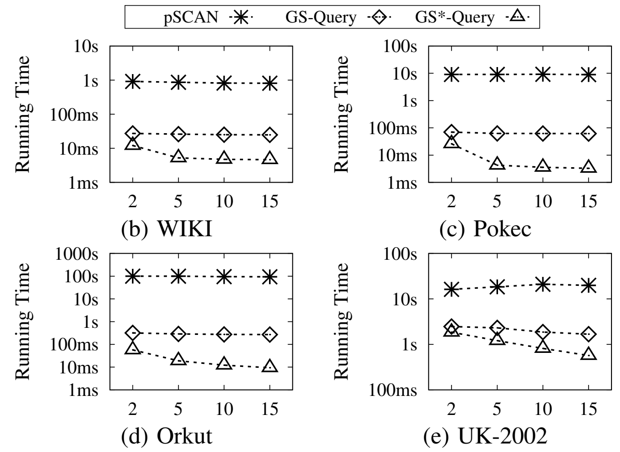


Figure 7: Query time for different μ ($\epsilon = 0.6$)

Eval-II: Varying μ. The time cost for GS*-Query, GS-Query and pSCAN by varying μ is given in Fig. 7. Similar to Fig. 6, the lines for GS*-Query present downward trends when μ increases from 2 to 15 on all datasets. By comparison, the changes of time cost for GS-Query and pSCAN are not obvious when increasing μ. Overall, GS*-Query significantly outperforms both GS-Query and pSCAN. Moreover, it becomes still faster when the result size decreases.

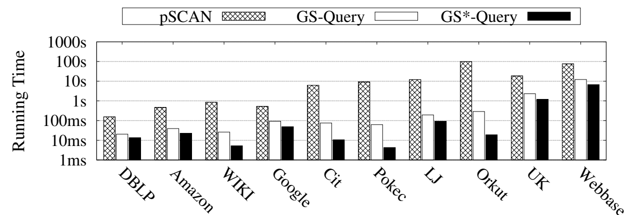


Figure 8: Query Time on Different Datasets

Eval-III: Query Performance on Different Datasets. The time cost for GS*-Query, GS-Query and pSCAN under the default parameters $\mu = 5, \epsilon = 0.6$ on all datasets is reported in Fig. 8. We can see that GS*-Query is far more efficient than GS-Query and

[1] http://snap.stanford.edu/index.html

[2] http://webgraph.di.unimi.it/

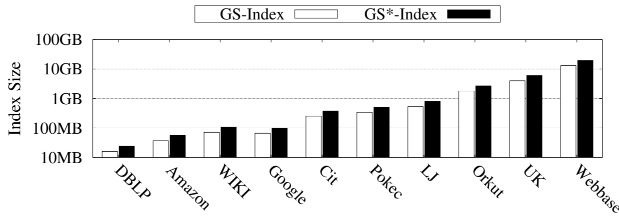


Figure 9: Index size for different datasets

pSCAN on all datasets. The time cost to perform GS*-Query on $Pokec$ is about only 4 milliseconds, which is the smallest value in all results. Meanwhile, GS-Query and pSCAN cost 63 milliseconds and 9 seconds respectively on that dataset. In the dataset $Webbase$ with over 1 billion edges, GS*-Query spends only 6 seconds, while GS-Query and pSCAN spend 12 seconds and 76 seconds respectively.

6.2 Performance of Index Construction

Eval-IV: Index Size on Different Datasets. The size of GS*-Index for different datasets is reported in Fig. 9. We add GS-Index as a comparison, which only saves the structural similarity for every pair of adjacent vertices. The size of GS*-Index gradually grows when the edge number increases, and the total size of GS*-Index can be always bounded by 1.5x the size used for GS-Index. For example, in Fig. 9, it costs 16MB to save the GS-Index while the GS*-Index costs 24MB in DBLP. In $Webbase$, we use about 12.7GB and 19.1GB to save the GS-Index and GS*-Index respectively.

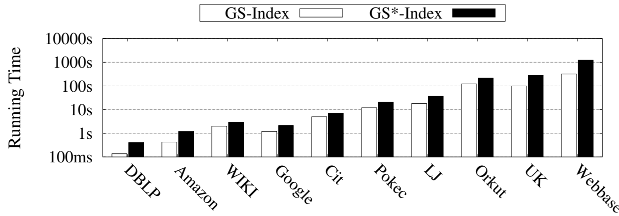


Figure 10: Time Cost for Index Construction

Eval-V: Construction Time on Different Datasets. The time cost for constructing GS-Construct and GS*-Construct are both reported in Fig. 10. Compared with GS-Construct, we use additional time in GS*-Construct for sorting. On the largest dataset Webbase, GS*-Construct and GS-Construct cost about 20 minutes and 7 minutes respectivelly. Note that the time cost for GS*-Construct decreases from 3 seconds on WIKI to 2 seconds on Google, even though Google has more edges. This is because the number of vertices in WIKI is much larger than it is in Google.

Eval-VI: Scalability Testing. We test the scalability of our proposed algorithms. We choose two real graph datasets Orkut and UK-2002 as representatives. For each dataset, we vary the graph size and graph density by randomly sampling nodes and edges respectively from 20% to 100%. When sampling nodes, we obtain the induced subgraph of the sampled nodes, and when sampling edges, we get the incident nodes of the edges as the vertex set. We report the time cost of GS*-Construct under different percentages, with GS-Construct as a comparison. The experimental results are shown in Fig. 11 and Fig. 12.

As we can see from Fig. 11, GS*-Construct expends more time to sort the core-orders and neighbor-orders when the sampling ratio increases. The gap between GS*-Construct and GS-Construct remains stable. Comparing lines in Fig. 11, the lines of GS*-Construct in Fig. 12 are gentler and have near linearly upward trends on both

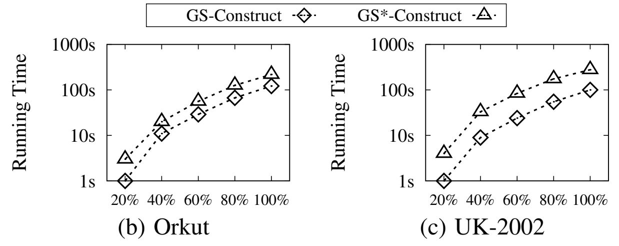


Figure 11: Index Construction (Vary $|V|$)

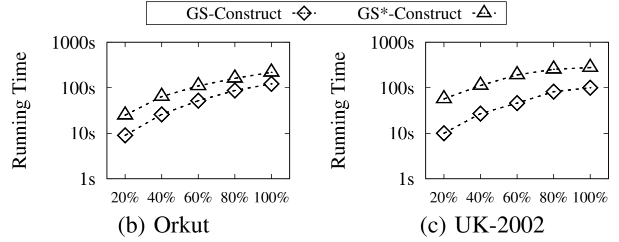


Figure 12: Index Construction (Vary $|E|$)

datasets. This demonstrates that the time cost of GS*-Construct is mainly dominated by the number of edges.

6.3 Performance of Index Maintenance

We test the performance of our maintenance algorithms. Since there is no previous work on this problem, we report on the algorithms GS*-Insert* and GS*-Remove*, with our basic algorithms GS*-Insert and GS*-Remove as comparisons. We randomly select 1000 distinct existing edges in the graph for each test. To test the performance of edge deletion, we remove the 1000 edges from the graphs one-by-one and record the average processing time. To test the performance of edge insertion, after the 1000 edges are removed, we insert them into the graph one by one and record the average processing time.

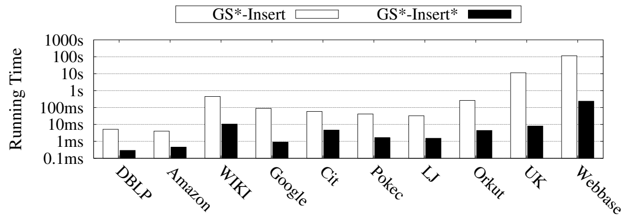


Figure 13: Time Cost for Edge Insertion

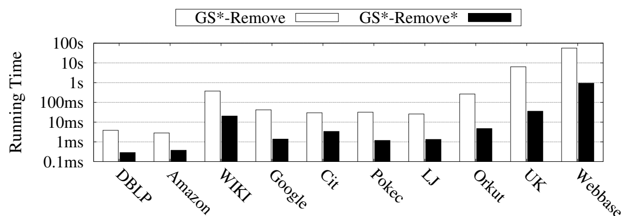
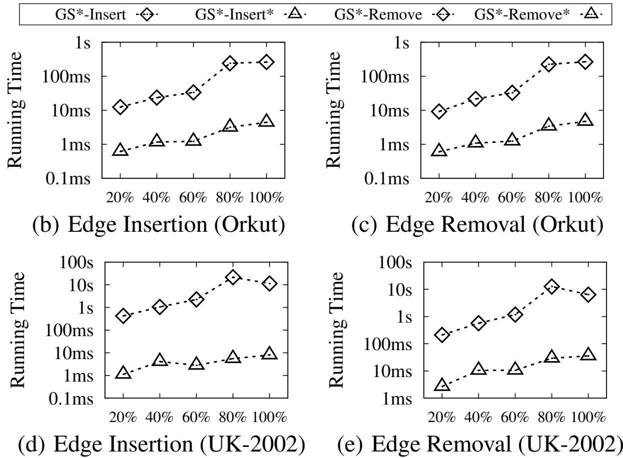


Figure 14: Time Cost for Edge Removal

Eval-VII: Index Maintenance on Different Datasets. The time cost of GS*-Insert* and GS*-Insert is reported in Fig. 13. We can find that GS*-Insert* is significantly faster than GS*-Insert. Of particular note, the gap increases as the graph size increases. For example, in $Webbase$, GS*-Insert spends about 114 seconds in

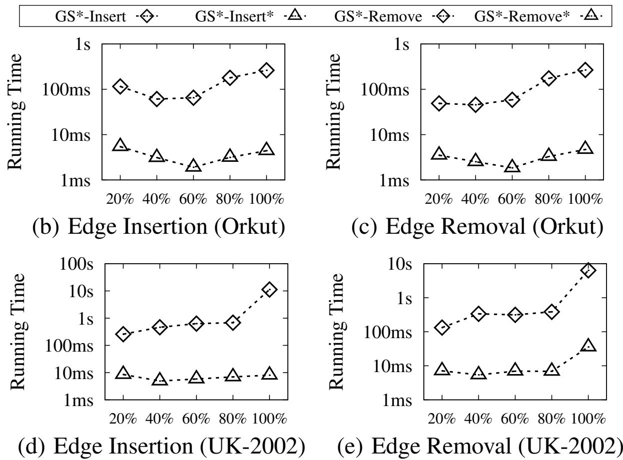
handling the edge insertion, while GS*-Insert* only spends about 0.3 seconds. Similar results appear in Fig. 14 for edge removal. GS*-Remove* is at least about 10x faster than GS*-Remove on all datasets and in some specific graphs, such as UK-2002, GS*-Remove spends more than 6 seconds compared to GS*-Remove*'s 35 milliseconds. The result shows our proposed algorithms GS*-Insert* and GS*-Remove* are significantly more efficient than GS*-Insert and GS*-Remove.



(b) Edge Insertion (Orkut) (c) Edge Removal (Orkut)

(d) Edge Insertion (UK-2002) (e) Edge Removal (UK-2002)

Figure 15: Index Maintenance (Vary $|V|$)



(b) Edge Insertion (Orkut) (c) Edge Removal (Orkut)

(d) Edge Insertion (UK-2002) (e) Edge Removal (UK-2002)

Figure 16: Index Maintenance (Vary $|E|$)

Eval-VIII: Scalability Testing. We evaluate the index-maintenance algorithm's performance by sampling $|V|$ and sampling $|E|$ respectively. The sampling method is the same as that in Eval-VI of Subsection 6.2. The statistics for varying $|V|$ and varying $|E|$ are reported in Fig. 15 and Fig. 16 respectively. All figures show that the time cost of GS*-Insert* (resp. GS*-Remove*) is significantly less than that of GS*-Insert (resp. GS*-Remove) and the gap between them grows when the sampling ratio increases. The increase of GS*-Insert* and GS*-Remove* is not obvious when graph size increases especially in Fig. 16. This suggests that the time cost for maintenance algorithms does not closely rely on the graph size. Overall, the results imply that our optimization techniques are effective and our final algorithms are efficient.

7. RELATED WORK

Structural Graph Clustering. The original algorithm for SCAN is proposed in [25]. It successfully identifies not only clusters but also hubs and outliers. However, SCAN computes the structural similarities for all pairs of adjacent vertices, which requires high computational cost and is not scalable to large graphs. To alleviate this problem, [19] proposes an algorithm named SCAN++. It is based on an intuitive idea that it is highly probable that many common neighbors exist between a vertex and its two-hop-away vertices. [2] proposes the algorithm pSCAN, which prior processes vertex with large probabilities as cores. An approximate solution, LinkSCAN* is proposed in [13]. It improves algorithm efficiency by simpling edges.

Other researchers are working to parallelize SCAN. [26] provides a parallel structural graph clustering algorithm based on MapReduce. [14] proposes an algorithm, named anySCAN, which progressively produces an approximate result for clustering on multi-core CPUs.

There also exist some parameter-free methods for SCAN, mainly focusing on computing a suitable ϵ during the algorithm. [1] proposes the algorithms SCOT and HintClus. SCOT is used to compute a sequence containing information for all ϵ-clusters, while HintClus computes hierarchical clusters. gSkeletonClu [20] uses a weighted version of structural similarity and computes clusters based on a tree-decomposition-based method. SHRINK [9] computes clusters by combining structural similarity and modularity-based methods.

Other Graph Clustering Methods. Other graph clustering methods have also been studied in the literature; they include graph partitioning [17, 6, 23], the modularity-based method [15, 18, 15], and the density-based method [10]. Efficient cohesive subgraphs detection has also been studied recently. Enumeration of all *cliques* and *quasi-cliques* are studied in [4] and [21] respectively. To relax the strict definition of clique, clique-relaxed models have also been studied, such as k-core [3], k-truss [22] and DN-subgraph [24], etc. More details can be found in a survey [12].

8. CONCLUSION

In this paper, we propose an index-based method for structural graph clustering. Our proposed index, named GS*-Index, contains two parts, which are core-orders and neighbor orders. We use core-orders to efficiently detect all cores. We use neighbor-orders to group all cores and construct the clusters. Based on GS*-Index, we efficiently answer the query for any given ϵ and μ, and the time cost for the query algorithm is only proportional to the size of clusters. To handle graph updates, we propose index maintenance algorithms. The experimental results demonstrate that our solution significantly outperforms the state-of-the-art algorithm.

9. REFERENCES

[1] D. Bortner and J. Han. Progressive clustering of networks using structure-connected order of traversal. In *Proc. of ICDE'10*, pages 653–656, 2010.

[2] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang. pscan: Fast and exact structural graph clustering. In *ICDE*, pages 253–264, 2016.

[3] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[4] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458, 2010.

[5] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SICOMP*, 14(1):210–223, 1985.

[6] C. H. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *ICDM*, pages 107–114, 2001.

[7] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[8] R. Guimera and L. A. N. Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028):895, 2005.

[9] J. Huang, H. Sun, J. Han, H. Deng, Y. Sun, and Y. Liu. Shrink: a structural clustering algorithm for detecting hierarchical communities in networks. In *CIKM*, pages 219–228, 2010.

[10] P. Jiang and M. Singh. Spici: a fast clustering algorithm for large biological networks. *Bioinformatics*, 26(8):1105–1111, 2010.

[11] U. Kang and C. Faloutsos. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, pages 300–309, 2011.

[12] V. E. Lee, N. Ruan, R. Jin, and C. Aggarwal. A survey of algorithms for dense subgraph discovery. In *Managing and Mining Graph Data*, pages 303–336. 2010.

[13] S. Lim, S. Ryu, S. Kwon, K. Jung, and J.-G. Lee. Linkscan*: Overlapping community detection using the link-space transformation. In *ICDE*, pages 292–303, 2014.

[14] S. T. Mai, M. S. Dieu, I. Assent, J. Jacobsen, J. Kristensen, and M. Birk. Scalable and interactive graph clustering algorithm on multicore cpus. In *ICDE*, pages 349–360, 2017.

[15] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[16] S. E. Schaeffer. Graph clustering. *Computer science review*, 1(1):27–64, 2007.

[17] J. Shi and J. Malik. Normalized cuts and image segmentation. *TPAMI*, 22(8):888–905, 2000.

[18] H. Shiokawa, Y. Fujiwara, and M. Onizuka. Fast algorithm for modularity-based graph clustering. In *AAAI*, pages 1170–1176, 2013.

[19] H. Shiokawa, Y. Fujiwara, and M. Onizuka. Scan++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. *PVLDB*, 8(11):1178–1189, 2015.

[20] H. Sun, J. Huang, J. Han, H. Deng, P. Zhao, and B. Feng. gskeletonclu: Density-based network clustering via structure-connected tree division or agglomeration. In *ICDM*, pages 481–490. IEEE, 2010.

[21] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In *KDD*, pages 104–112. ACM, 2013.

[22] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[23] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE*, 2014.

[24] N. Wang, J. Zhang, K.-L. Tan, and A. K. Tung. On triangulation-based dense neighborhood graph discovery. *PVLDB*, 4(2):58–68, 2010.

[25] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. Scan: A structural clustering algorithm for networks. In *KDD*, pages 824–833, 2007.

[26] W. Zhao, V. Martha, and X. Xu. Pscan: a parallel structural clustering algorithm for big networks in mapreduce. In *AINA*, pages 862–869, 2013.

[27] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1):718–729, 2009.