pSCAN: Fast and Exact Structural Graph Clustering

Lijun Chang[†], Wei Li[†], Xuemin Lin[†], Lu Qin[‡], Wenjie Zhang[†]
[†]University of New South Wales, Australia
[‡]University of Technology Sydney, Australia
{ljchang,weili,lxue,zhangw}@cse.unsw.edu.au, lu.qin@uts.edu.au

***Abstract*—In this paper, we study the problem of structural graph clustering, a fundamental problem in managing and analyzing graph data. Given a large graph $G = (V, E)$, structural graph clustering is to assign vertices in V to clusters and to identify the sets of hub vertices and outlier vertices as well, such that vertices in the same cluster are densely connected to each other while vertices in different clusters are loosely connected to each other. Firstly, we prove that the existing SCAN approach is worst-case optimal. Nevertheless, it is still not scalable to large graphs due to exhaustively computing structural similarity for every pair of adjacent vertices. Secondly, we make three observations about structural graph clustering, which present opportunities for further optimization. Based on these observations, in this paper we develop a new two-step paradigm for scalable structural graph clustering. Thirdly, following this paradigm, we present a new approach aiming to reduce the number of structural similarity computations. Moreover, we propose optimization techniques to speed up checking whether two vertices are structure-similar to each other. Finally, we conduct extensive performance studies on large real and synthetic graphs, which demonstrate that our new approach outperforms the state-of-the-art approaches by over one order of magnitude. Noticeably, for the twitter graph with 1 billion edges, our approach takes 25 minutes while the state-of-the-art approach cannot finish even after 24 hours.**

I. Introduction

Due to the strong expressive power of the graph model, many real-world applications model data and relationships among the data as a graph $G = (V, E)$, where vertices in V represent entities of interest and edges in E represent relationships between entities. With the proliferation of graph applications, such as social networks, information networks, web search, collaboration networks, E-commerce networks, communication networks, and biology, significant research efforts have been devoted towards efficiently and effectively managing and analyzing graph data. Among them, graph clustering is a fundamental problem and has been extensively studied (*e.g.*, in [7], [11], [14], [20], [21], [22], [27]).

Given a large graph G, graph clustering (or graph partitioning) is to cluster vertices in G such that there is a dense set of edges among vertices in the same cluster and there are few edges among vertices belonging to different clusters. Graph clustering has many applications. For example, graph clustering can be used for detecting hidden structures in a graph [27]. In social networks (*e.g.*, Facebook), clusters in a graph can be regarded as communities in the graph [12]. In a collaboration network (*e.g.*, DBLP), a cluster may be a group of researchers with similar research interests. In computational biology, computing functional clusters of genes can help biologists to conduct the study of gene microarrays.

In the literature, many different clustering definitions have been proposed, such as modularity-based method ([7], [14], [20]), graph partitioning ([11], [21], [24]), and density-based method ([16]). However, all these definitions/methods do not distinguish the different roles of the vertices in a graph; that is, some vertices are members of clusters while other vertices are not, among which some vertices are hubs that bridge many clusters and other vertices are just outliers [17], [27]. To distinguish the different roles of the vertices, *structural graph clustering* was proposed in [27]. It uses the neighborhoods of the vertices as clustering criteria, and defines the *structural similarity* $\delta(u, v)$ between vertices u and v as the number of their common neighbors normalized by the geometric mean of their degrees. Given parameters ϵ and μ, vertices u and v are *structure-similar* to each other if $\delta(u, v) \geq \epsilon$, and u is a *core vertex* if it has at least μ neighbors that are structure-similar to it. Then, clusters are growing from core vertices by including into each cluster all vertices that are structure-similar to core vertices in the cluster. Finally, for each vertex that is not a member of clusters, it is a *hub vertex* if its neighbors belong to two or more clusters, and it is an *outlier vertex* otherwise. For example, Figure 1 shows two clusters, respectively in Data Mining research area and Information Retrieval research area, extracted from the coauthor network (http://arnetminer.org/) by structural graph clustering; the hub vertices and outlier vertices are shown as well.

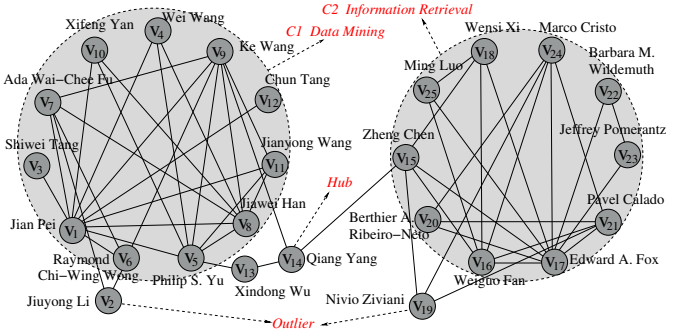


Fig. 1: Example structural graph clustering

Note that, after the clusters in a graph are computed, it is fairly easy to obtain the hub vertices and outlier vertices by following the definition. Thus, in this paper we mainly focus on efficiently computing clusters for structural graph clustering.

Existing Approaches and Challenges. There exist two approaches for exact structural graph clustering: SCAN and SCAN++. The SCAN approach [27] iterates through all vertices that have not yet been assigned to clusters. For each such vertex u, if it is a core vertex, then the SCAN approach creates a new cluster C initially containing only u and iteratively adds into C all such vertices that are structure-similar to a core vertex in C. Thus, the SCAN approach essentially computes the structural similarity for every pair of adjacent vertices in

G (*i.e.*, for every $(u, v) \in E$); this requires high computational cost and makes it not scalable to large graphs.

SCAN++ [22] was recently proposed to overcome the drawback of SCAN, and is the state-of-the-art approach. It is designed based on the property that a vertex and its two-hop-away vertices are expected to share large parts of their neighborhoods due to the large values of clustering coefficients in real-world graphs [26]. Thus, SCAN++ may avoid computing structural similarity between vertices that are shared between the neighbors of a vertex and its two-hop-away vertices. Compared with SCAN, SCAN++ computes less structural similarities. However, the number of structural similarity computations in SCAN++ is still large.

In summary, there are two main challenges for structural graph clustering.

- How to reduce the number of structural similarity computations?
- How to efficiently check whether two vertices are structure-similar to each other?

Note that, for each pair of non-pruned adjacent vertices, all existing approaches compute the exact value of their structural similarity to determine whether the two vertices are structure-similar to each other; this is time-consuming.

Our Approaches and Contributions. In this paper, we prove that the existing SCAN approach is worst-case optimal; nevertheless, it is still not scalable to large graphs due to exhaustively computing structural similarity for every pair of adjacent vertices. Thus, in this paper, we propose a new efficient approach aiming at reducing the number of structural similarity computations as well as optimizing the checking of structure-similar between two vertices. Our approach is also worst-case optimal.

Firstly, we make three observations about structural graph clustering: 1) the clusters in structural graph clustering may overlap; 2) the clusters of core vertices are disjoint; and 3) the clusters of non-core vertices are uniquely determined by core vertices. Base on these observations, we develop a two-step paradigm for structural graph clustering. In the paradigm, we first cluster all core vertices by partitioning them into clusters, and then cluster non-core vertices by assigning each non-core vertex v to the same clusters as its neighbors that are core vertices and are structure-similar to v.

Following the paradigm, we propose a *pruned* SCAN (denoted pSCAN) approach for structural graph clustering. We incrementally maintain an effective-degree $ed(v)$ and a similar-degree $sd(v)$ for every vertex $v \in V$, with $ed(v) \geq sd(v)$. A vertex v is determined to be a core vertex once $sd(v) \geq \mu$, and it is determined to be a non-core vertex once $ed(v) < \mu$; thus, the set of core vertices can be efficiently identified. Then, the clusters of core vertices are computed based on the transitive property that, two core vertices, u and v, are in the same cluster if $(u, v) \in E$ and they are structure-similar. Moreover, we avoid computing the structural similarity between core vertices u and v if they have already been assigned to the same cluster. Consequently, pSCAN saves a lot of structural similarity computations.

To improve the efficiency of pSCAN, we further propose three optimization techniques, cross link, pruning rules, and adaptive structure-similar checking, for speeding up the checking of structure-similar between two vertices without computing the exact value of their structural similarity.

Our primary contributions are summarized as follows.

- *A Proof of Worst-Case Optimality.* We prove that the existing SCAN approach is worst-case optimal. Nevertheless, it is still not scalable to large graphs.
- *A New Two-Step Paradigm.* We develop a new two-step paradigm for structural graph clustering based on our three observations.
- *An Optimization Approach.* We propose an optimization approach by reducing the number of structural similarity computations as well as optimizing the checking of structure-similar between two vertices.

We conduct extensive empirical studies on large real and synthetic graphs. The empirical studies confirm that our new approach significantly outperforms the state-of-the-art approaches by over one order of magnitude. Noticeably, for the twitter graph with 1 billion edges, our approach computes the structural graph clustering in 25 minutes while the state-of-the-art approach cannot finish even after 24 hours.

Organization. The rest of the paper is organized as follows. A brief overview of related work is given below. Section II gives the definition of the studied problem. In Section III, we analyze the existing SCAN algorithm and prove that it is worst-case optimal. Based on our three observations, we propose a new paradigm for structural graph clustering in Section IV, while our new approach aiming to reduce the number of structural similarity computations is presented in Section V. To speed up checking whether two vertices are structure-similar to each other, we further propose three optimization techniques in Section VI. Section VII presents our experimental results, and Section VIII finally concludes the paper.

Related Work. We categorize the related works as follows.

Structural Graph Clustering. Exact structural graph clustering was studied in [22], [27] as discussed above. Approximation techniques were proposed in [19] to improve the efficiency of SCAN by sampling edges to reduce the number of structural similarity computations; however, this produces approximate results. In this paper, we propose a new approach for exact structural graph clustering, which improves the state-of-the-art approach by over one order of magnitude.

Other Graph Clustering Models. Other graph clustering models have also been studied; for example, modularity-based method ([7], [14], [20]), graph partitioning ([11], [21], [24]), and density-based method ([16]). However, due to inherently different problem definitions, these methods cannot be applied to structural graph clustering.

Dense Subgraph Extraction. Efficient techniques for computing all maximal *cliques* and *quasi-cliques* of a graph were presented in [1], [4], and [28], respectively. Problems of efficiently computing other dense subgraphs, including k-core [3], DN-subgraph [25], triangle k-core motifs [29], k-edge connected components [2], etc., have also been recently investigated. Nevertheless, due to inherently different problem definitions, these techniques are inapplicable to structural graph clustering.

Online Community Search. Given a set q of query vertices and a graph G, the problem of online community search that computes the communities in G containing q has also been studied recently. Different semantics for community search have been studied; for example, local modularity based community search [6], k-core based community search [23], [10], k-truss based community search [15], and α-adjacency γ-quasi-k-clique based community search [9]. Nevertheless, due to inherently different problem natures, none of these techniques can be used for structural graph clustering.

II. Preliminary

In this paper, we focus on *an unweighted undirected graph* $G = (V, E)$ [13], where V is the set of vertices and E is the set of edges. We denote the number of vertices, $|V|$, and the number of edges, $|E|$, in G by n and m, respectively. Let $(u, v) \in E$ denote an edge between u and v; u (resp. v) is said to be a neighbor of v (resp. u). In the following, for ease of presentation we simply refer an unweighted undirected graph as a graph.

Definition 2.1: The **structural neighborhood** of a vertex u, denoted by $N[u]$, is defined as the *closed neighborhood* [13] of u; that is $N[u] = \{v \in V \mid (u, v) \in E\} \cup \{u\}$.

In this paper, we focus on the structural neighborhood (*i.e.*, closed neighborhood) of a vertex, and the *degree of* u, denoted by $d[u]$, is the cardinality of $N[u]$ (*i.e.*, $d[u] = |N[u]|$). Note that, the *open neighborhood* [13] of u, denoted by $N(u)$, is the set of neighbors of u ($N(u) = \{v \in V \mid (u, v) \in E\}$). For example, for vertex v_5 in Figure 2, its structural neighborhood is $N[v_5] = \{v_4, v_5, v_6\}$, its degree is $d[v_5] = |N[v_5]| = 3$, and its open neighborhood is $N(v_5) = \{v_4, v_6\}$.

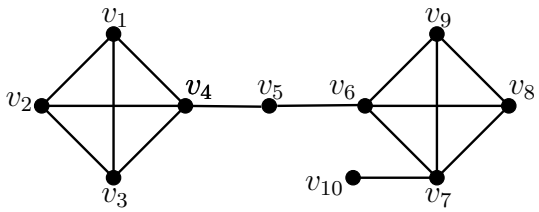


Fig. 2: An example graph

Definition 2.2: The **structural similarity** between vertices u and v, denoted by $\sigma(u, v)$, is defined as the number of common vertices in $N[u]$ and $N[v]$ normalized by the geometric mean of their cardinalities; that is,

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

Intuitively, for two vertices, the more common vertices in their structural neighborhoods, the larger the structural similarity value. Note that, the structural similarity is between 0 and 1; that is, $0 \leq \sigma(u, v) \leq 1, \forall u, v \in V$. For example, in Figure 2, $N[v_5] = \{v_4, v_5, v_6\}$ and $N[v_4] = \{v_1, v_2, v_3, v_4, v_5\}$; thus $\sigma(v_4, v_5) = \frac{|\{v_4, v_5\}|}{\sqrt{3 \cdot 5}} = \frac{2}{\sqrt{15}}$. Similarly, $N[v_1] = \{v_1, v_2, v_3, v_4\}$ and $\sigma(v_1, v_4) = \frac{|\{v_1, v_2, v_3, v_4\}|}{\sqrt{4 \cdot 5}} = \frac{2}{\sqrt{5}}$.

Given a similarity threshold $0 < \epsilon \leq 1$, the *ϵ-neighborhood* of u, denoted by $N_\epsilon[u]$, is defined as the subset of vertices in $N[u]$ whose structural similarities with u are at least ϵ; that is, $N_\epsilon[u] = \{v \in N[u] \mid \sigma(u, v) \geq \epsilon\}$. Note that, $N_\epsilon[u]$ also includes u for each $u \in V$, since $\sigma(u, u) = 1$. For example, in Figure 2, $N_{0.8}[v_4] = \{v_1, v_2, v_3, v_4\}$, and $N_{0.8}[v_7] = \{v_6, v_7, v_8, v_9\}$.

Definition 2.3: Given a similarity threshold $0 < \epsilon \leq 1$ and an integer $\mu \geq 2$, a vertex u is a **core vertex** if $N_\epsilon[u] \geq \mu$.

A vertex is a *non-core vertex* if it is not a core vertex. For example, given $\epsilon = 0.8$ and $\mu = 4$ in Figure 2, v_4 and v_7 are core vertices (since $N_{0.8}[v_4] = N_{0.8}[v_7] = 4$) while v_5 and v_{10} are non-core vertices.

Given parameters $0 < \epsilon \leq 1$ and $\mu \geq 2$, vertex v is *structure-reachable* from vertex u if there is a sequence of vertices $v_1, v_2, \ldots, v_l \in V$ (for some integer $l \geq 2$) such that: (i) $v_1 = u$ and $v_l = v$; (ii) $v_1, v_2, \ldots, v_{l-1}$ are core vertices; and (iii) $v_{i+1} \in N_\epsilon[v_i]$ for each $1 \leq i \leq l - 1$.

Definition 2.4: A **cluster** C is a subset of V with at least two vertices (*i.e.*, $|C| \geq 2$) such that:

- (Maximality) If a core vertex $u \in C$, then all vertices that are structure-reachable from u also belong to C.
- (Connectivity) For any two vertices $v_1, v_2 \in C$, there is a vertex $u \in C$ such that both v_1 and v_2 are structure-reachable from u.

For example, given $\epsilon = 0.8$ and $\mu = 4$, there are two clusters in Figure 2: $C_1 = \{v_1, v_2, v_3, v_4\}$ and $C_2 = \{v_6, v_7, v_8, v_9\}$.

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

In the following, we assume that there always exist two parameters, $0 < \epsilon \leq 1$ and $\mu \geq 2$, without explicitly mentioning them, and we say that vertices u and v are *structure-similar* to each other if $\sigma(u, v) \geq \epsilon$.

A. Hubs and Outliers

After computing the set of clusters in a graph G, similar to [22], [27] we can also obtain the set of hubs and outliers in G, which are defined as follows.

Definition 2.5: Given the set $\mathbb{C}$ of clusters in a graph G, a vertex u that is not in any cluster in $\mathbb{C}$ is a **hub** vertex if its neighbors belong to two or more clusters, and it is an **outlier** vertex otherwise.

For example, given $\epsilon = 0.8$ and $\mu = 4$, in Figure 2 $\mathbb{C} = \{\{v_1, v_2, v_3, v_4\}, \{v_6, v_7, v_8, v_9\}\}$, v_5 is a hub vertex since its neighbors v_4 and v_6 belong to different clusters, and v_{10} is an outlier vertex since it has only one neighbor.

Since, given the set of clusters in G, it is straightforward to obtain the set of hubs and the set of outliers in a graph G in $O(n + m)$ time by following the definition, we only focus on computing the set of clusters in G in the following. Nevertheless, our techniques can be easily extended to obtain hubs and outliers in G as well.

III. Analysis of The SCAN Algorithm

In this section, we briefly review the existing SCAN algorithm proposed in [27] which serves as one of the baseline algorithms in our experiments. We prove that SCAN is *worst-case optimal*; nevertheless, it is not scalable to large graphs, due to exhaustively computing structural similarities for all pairs of adjacent vertices.

Algorithm 1: SCAN [27]

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 vertex u ∈ V do
4     C ← {u};
5     for each unexplored vertex v in C do
6         if |N_ε[v]| ≥ μ then  C ← C ∪ N_ε[v];
7         Mark v as explored;
8     if |C| > 1 then  ℂ ← ℂ ∪ {C};
9 return ℂ;
```

The pseudocode of SCAN is shown in Algorithm 1. It first computes structural similarities for all pairs of adjacent vertices, and then obtains the clusters by traversing the graph once. Note that, for ease of analysis, Algorithm 1 separates the computation of structural similarities (*e.g.*, see Line 1) from the other parts; nevertheless, it is equivalent to the original algorithm in [27]. The correctness of SCAN directly follows from the lemma below, which was proved in [27].

Lemma 3.1: *[27] For any cluster $C \in \mathbb{C}$ in G, it equals the set of vertices that are structure-reachable from u, where u is an arbitrary core vertex in C,*

In Algorithm 1, Line 1 essentially enumerates all triangles in G. The rationality is that, when computing $\sigma(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}}$, each vertex $w \in (N[u] \cap N[v]) \backslash \{u, v\}$ forms a triangle with u and v; on the other hand, each triangle (u, v, w) in G contributes one vertex to each of $N[u] \cap N[v]$, $N[v] \cap N[w]$, and $N[u] \cap N[w]$. Thus, Line 1 has the same time complexity as triangle enumeration, which is $O(\alpha(G) \cdot m)$ with $\alpha(G) \leq \sqrt{m}$ [5]; here, $\alpha(G)$ is the arboricity of G, which equals the minimum number of edge-disjoint forests needed to cover all edges of G. On the other hand, it is easy to see that the time complexity of Lines 2–8 is $O(m)$. Consequently, computing structural similarities is the dominating cost in structural graph clustering. The time complexity of Algorithm 1 is $O(\alpha(G) \cdot m)$, and it is $O(m^{1.5})$ in the worst case.

Theorem 3.1: *Algorithm* SCAN *is worst-case optimal for structural graph clustering.*

Proof Sketch: Let's consider a graph G with $2n$ vertices, $\{v_1, v_2, \ldots, v_{2n}\}$, where vertices $\{v_1, v_2, \ldots, v_n\}$ form a clique and each vertex v_{i+n} for $1 \leq i \leq n$ is only connected to v_i. Given $\mu = n + 1$ and $\epsilon > 0$ being the largest value such that the entire graph G is a single cluster. Then, any algorithm for structural graph clustering needs to compute structural similarities for all pairs of adjacent vertices in G to confirm that every vertex in the clique is a core vertex; this is equivalent to enumerating all triangles in G, thus has a time complexity $O(\alpha(G) \cdot m) = O(m^{1.5})$. Note that, there are $\theta(n^3) = \theta(m^{1.5})$ triangles in the graph, and each triangle needs to be enumerated to get the correct clusters. Thus, the theorem holds. □

Since SCAN is worst-case optimal, it is unlikely that there is an algorithm for structural graph clustering with better worst-case time complexity than SCAN.[1] Thus, in this paper we develop practical techniques for scalable structural graph clustering. In particular, since computing structural similarities for all pairs of adjacent vertices in G is the dominating cost, we mainly focus on optimizing the structural similarity computations in the following.

IV. A New Paradigm

In this section, we develop a new paradigm for structural graph clustering aiming to reduce the number of structural similarity computations. In the following, we first present our three observations about structural graph clustering in Section IV-A, based on which we then develop our new paradigm in Section IV-B.

A. Observations

We have three important observations about structural graph clustering that has not been stated and utilized in the existing works.

Observatioin-I: The Clusters May Overlap. The first observation is that the clusters computed by structural graph clustering may overlap. For example, consider the graph in Figure 3 with $\epsilon = \frac{2}{\sqrt{15}}$ and $\mu = 4$, where structural similarities of pairs of adjacent vertices are shown beside the edges. One can verify that, $\sigma(u, v) \geq \epsilon$ for all edges (u, v) in the graph, and all vertices except v_5 are core vertices. Consequently, there are two clusters in the graph in Figure 3, $C_1 = \{v_1, \ldots, v_5\}$ and $C_2 = \{v_5, \ldots, v_9\}$, where v_5 belongs to both clusters. Therefore, partitioning a graph into disjoint subgraphs will not generate the correct clusters in the graph.

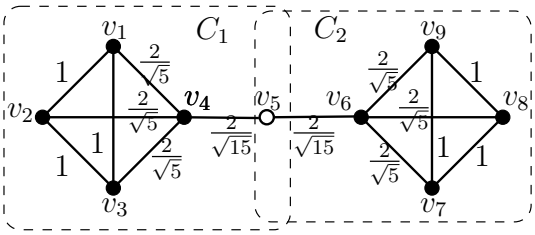


Fig. 3: Overlapping clusters

Observation-II: The Clusters of Core Vertices Are Disjoint. Although clusters may overlap, we prove that each core vertex belongs to a unique cluster in the following lemma.

Lemma 4.1: *Each core vertex belongs to a unique cluster in structural graph clustering.*

Proof Sketch: We prove the lemma by contradiction. Assume there is a core vertex u that belongs to two clusters, C_1 and C_2.

[1] Note that, the time complexity in [22] is based on average-case analysis.

Then, following Lemma 3.1, every vertex in C_1 is structure-reachable from u and every vertex in C_2 is also structure-reachable from u. Thus, all vertices in $C_1 \cup C_2$ are structure-reachable from u; this contradicts that C_1 (respectively, C_2) is a cluster with $u \in C_1$ (respectively, $u \in C_2$). □

From Lemma 4.1, we can see that the clusters of core vertices form a partition of the entire set of core vertices. For example, the two clusters of core vertices in Figure 3 are $\{v_1, \ldots, v_4\}$ and $\{v_6, \ldots, v_9\}$, which form a partition of the entire set of core vertices. Thus, it is feasible to compute clusters of core vertices by partitioning the entire set of core vertices.

Observation-III: The Clusters of Non-core Vertices Are Uniquely Determined By Core Vertices. Our third observation is that, given the clusters $\mathbb{C}_c$ of core vertices in G, non-core vertices in G can be clustered by assigning each non-core vertex v to every cluster $C_c \in \mathbb{C}_c$ such that there is a core vertex $u \in C_c$ and $v \in N_\epsilon[u]$. For example, after clustering the core vertices in Figure 3, the non-core vertex v_5 is assigned to both clusters since $v_5 \in N_\epsilon[v_4]$ and $v_5 \in N_\epsilon[v_6]$. We prove the correctness of this assignment in the following lemma.

Lemma 4.2: *Let $\mathbb{C}_c$ be the set of clusters of core vertices in G. Then, the set of clusters of all vertices in G is $\mathbb{C} = \{C_c \bigcup_{u \in C_c} N_\epsilon[u] \mid C_c \in \mathbb{C}_c\}$.*

Proof Sketch: Lemma 3.1 says that, each cluster C in G equals the set of vertices that are structure-reachable from v where v is an arbitrary core vertex in C. From Lemma 4.1, we also know that v belongs to a unique cluster in $\mathbb{C}_c$, let this cluster be C_c. Now, we only need to prove that C and $C_c \bigcup_{u \in C_c} N_\epsilon[u]$ are the same.

($\Rightarrow$) First, we prove that every vertex in C is also in $C_c \bigcup_{u \in C_c} N_\epsilon[u]$. As C is the set of vertices that are structure-reachable from v, then any vertex $w \in C$ is structure-reachable from v. If w is a core vertex, then $w \in C_c$ since $\mathbb{C}_c$ is the set of clusters of core vertices. Otherwise, there is a sequence of vertices $v_1 = v, v_2, \ldots, v_l = w$ such that v_i is a core vertex for each $1 \leq i < l$ and $w \in N_\epsilon[v_{l-1}]$; thus, $v_{l-1} \in C_c$ and $w \in \bigcup_{u \in C_c} N_\epsilon[u]$.

($\Leftarrow$) It is easy to see that $C_c \subseteq C$. Moreover, for an arbitrary vertex $w \in \bigcup_{u \in C_c} N_\epsilon[u]$, it is easy to verify that w is structure-reachable from v; thus $w \in C$.

Thus, the lemma holds. □

Therefore, after clustering core vertices in a graph G, following Lemma 4.2 we can obtain the set of clusters in G by assigning neighbors of core vertices to proper clusters while ignoring other vertices and their connections in G.

B. The Paradigm

Based on the three observations in Section IV-A, in this paper we develop a new paradigm for structural graph clustering consisting of two steps: step-1) clustering core vertices; and step-2) clustering non-core vertices.

Clustering Core Vertices. For efficiently clustering the set of core vertices, we propose the concept of connectivity graph.

Definition 4.1: Given a graph $G = (V, E)$, we define a **connectivity graph** $G_c = (V_c, E_c)$, where V_c is the set of core vertices in G and there is an edge $(u, v) \in E_c$ if and only if $u, v \in V_c$, $(u, v) \in E$, and $\sigma(u, v) \geq \epsilon$ in G.

We prove in the following lemma that, there is a one-to-one correspondence between clusters of core vertices in G and connected components in G_c, where a connected component of a graph is a maximal subgraph in which any two vertices are connected to each other by a path.

Lemma 4.3: *For any two core vertices u and v in G, they are in the same cluster in G if and only if they are in the same connected component in G_c.*

Proof Sketch: Firstly, we prove that if u and v are in the same connected component in G_c then they are in the same cluster in G. Since u and v are in the same connected component in G_c, then there is a path of core vertices in G_c, $(v_1 = u, \ldots, v_l = v)$, such that for each $1 \leq i < l$, $(v_i, v_{i+1}) \in E_c$ (*i.e.*, $(v_i, v_{i+1}) \in E$ and $\sigma(v_i, v_{i+1}) \geq \epsilon$ in G). Therefore, v is structure-reachable from u, and thus u and v are in the same cluster in G.

Secondly, we prove that if u and v are in the same cluster in G then they are in the same connected component in G_c. Since u and v are in the same cluster in G, then there is a core vertex $w \in V_c$ such that both u and v are structure-reachable from w. That is, there is a sequence of vertices $v_1 = w, v_2, \ldots, v_l = u$ in G such that $v_1, \ldots, v_{l-1}$ are core vertices and for each $1 \leq i < l$ $\sigma(v_i, v_{i+1}) \geq \epsilon$. Since u is also a core vertex, $(v_1, \ldots, v_l)$ is a path in G_c, thus w and u are in the same connected component in G_c. Similarly, we can prove that w and v are in the same connected component in G_c. Thus, u and v are in the same connected component in G_c. □

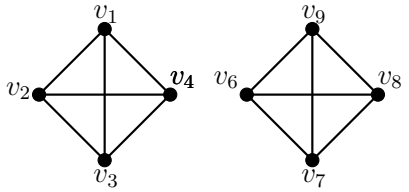


Fig. 4: Connectivity graph

For example, the connectivity graph of the graph in Figure 3 with $\epsilon = \frac{2}{\sqrt{15}}$ and $\mu = 4$ is shown in Figure 4. The two connected components of the connectivity graph are $\{v_1, \ldots, v_4\}$ and $\{v_6, \ldots, v_9\}$, which respectively correspond to the two clusters of core vertices in Figure 3.

Clustering Non-core Vertices. The clusters of non-core vertices can be computed by directly following Observation-III and Lemma 4.2.

The Paradigm. Following Lemma 4.2 and Lemma 4.3, the pseudocode of our two-step paradigm for structural graph clustering is shown in Algorithm 2, which first computes the clusters of core vertices and then assigns non-core vertices to clusters. Note that, for presentation simplicity, in Algorithm 2 we assume that the connectivity graph G_c contains all vertices in G; nevertheless, non-core vertices can be pruned from G_c as a postprocessing.

In Algorithm 2, Lines 1–8 cluster core vertices while Line 9 clusters non-core vertices. To cluster core vertices, following Lemma 4.3 we construct the connectivity graph G_c. For each

Algorithm 2: Paradigm

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

```
   /* Step-1: cluster core vertices                          */
1  Initialize the connectivity graph to be G_c = (V, ∅);
2  for each vertex u ∈ V do core(u) ← false;
3  for each vertex u ∈ V do
4      if u is a core vertex then core(u) ← true;
       /* Add edges into G_c                                 */
5      if core(u) then
6          for each vertex v ∈ N_ε[u] do
7              if core(v) then Insert edge (u, v) into G_c;
8  Let ℂ_c be the set of connected components in G_c containing
   core vertices;
   /* Step-2: cluster non-core vertices                      */
9  ℂ ← {C_c ⋃_{u∈C_c} N_ε[u] | C_c ∈ ℂ_c};
10 return ℂ;
```

vertex $u \in V$, we first determine whether u is a core vertex (Line 4). If u is a core vertex (Line 5), then for each core vertex $v \in N_\epsilon[u]$ (Line 6), we add an edge (u, v) into G_c (Line 7). Then, each set of core vertices in a connected component in G_c is a cluster of core vertices in G (Line 8). Finally, the non-core vertices are added to clusters according to Lemma 4.2 (Line 9).

V. Our pSCAN Approach

Based on the observations and the new paradigm in Section IV, in this section we present our new approach to structural graph clustering. Our aim is to reduce the number of structural similarity computations. Thus, we partition the set of all pairs of adjacent vertices (*i.e.*, E) into three subsets, $E_{n,n}$, $E_{c,c}$, and $E_{c,n}$, and discuss how to reduce the number of computations for each subset, respectively; here, $E_{n,n}$ is the set of pairs of adjacent non-core vertices, $E_{c,c}$ is the set of pairs of adjacent core vertices, and $E_{c,n}$ is the set of adjacent vertex pairs between core vertices and non-core vertices.

The pSCAN Approach. Following the general paradigm in Algorithm 2, we propose a *pruned* SCAN approach, denoted pSCAN, to structural graph clustering. The pseudocode of pSCAN is shown in Algorithm 3. For every vertex $u \in V$, we initially compute two values (Lines 2–4), whose definitions shall be discussed shortly. Then, we iteratively check every vertex $u \in V$ whether it is a core vertex or not (Line 6); if u is a core vertex, then we cluster its neighbors that have been determined to be core vertices (Line 7). After checking all vertices (*i.e.*, the set of core vertices have been obtained), the set $\mathbb{C}_c$ of clusters of core vertices is generated from the disjoint-set data structure (Line 8). Finally, we cluster the non-core vertices (Line 9).

In Algorithm 3, instead of explicitly constructing the connectivity graph G_c, we use a disjoint-set data structure [8] to incrementally maintain the connected components in G_c. The data structure maintains a collection $\mathbb{S} = \{S_1, S_2, \ldots\}$ of disjoint dynamic subsets, and has two fundamental operations: find-subset and union; find-subset determines which subset a particular element is in and union joins two subsets into a single subset. Initially, each vertex in G forms a singleton

Algorithm 3: pSCAN

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

```
   /* Step-1: cluster core vertices                          */
1  Initialize a disjoint-set data structure with all vertices in V;
2  for each vertex u ∈ V do
3      sd(u) ← 0;    /* Initialize similar-degree */;
4      ed(u) ← d[u]; /* Initialize effective-degree */;
5  for each vertex u ∈ V in non-increasing order w.r.t. ed(u) do
6      CheckCore(u); /* Check if u is core vertex */;
7      if sd(u) ≥ μ then ClusterCore(u);
8  ℂ_c ← the set of subsets of core vertices in the disjoint-set data
   structure;
   /* Step-2: cluster non-core vertices                      */
9  ClusterNoncore();
10 return ℂ;
```

subset (Line 1); adding an edge (u, v) into G_c is achieved by union(u, v). Thus, *two vertices u and v are in the same connected component if and only if they are in the same subset in the data structure (i.e.,* find-subset(u) = find-subset(v)*).*

Vertex Exploration Order. With the aim of reducing the number of structural similarity computations, we explore vertices in G in non-increasing order of their effective-degrees (Line 5), which is defined as follows.

Definition 5.1: The **effective-degree** of a vertex $u \in V$, denoted ed(u), is the degree of u minus the number of u's neighbors that have been determined to be not structure-similar to u.

Intuitively, the effective-degree of a vertex u is an upper bound of the size of its ϵ-neighborhood (*i.e.*, $|N_\epsilon[u]| \leq$ ed(u)); thus, the larger the effective-degree of a vertex, the more likely it is a core vertex. For example, in Figure 5, assume $\sigma(v_4, v_6) < \epsilon$ and $\sigma(v_1, v_6) < \epsilon$ have been computed, then ed$(v_6) = d[v_6] - 2 = 2$.

The rationality that we first explore vertices that are more likely to be core vertices (*i.e.*, with higher effective-degrees) is as follows. From Section IV, we know that after computing the clusters $\mathbb{C}_c$ of core vertices in G, the ϵ-neighborhoods of all core vertices are sufficient for generating the clusters in G. That is, after obtaining the set V_c of core vertices in G and computing structural similarities between core vertices and their neighbors, we can correctly generate the clusters in G without computing structural similarity for other pairs of vertices. In this way, *we reduce the number of structural similarity computations for vertex-pairs in* $E_{n,n}$. For example, in Figure 5, assume $\{v_1, \ldots, v_5\}$ is determined to be the set of core vertices, then the structural similarity between v_6 and v_7 does not need to be computed in order to compute the clusters in the graph.

Note that, the effective-degree of a vertex u will be decreased when more of its neighbors have been explored and determined to be not structure-similar to u. Efficient techniques for iteratively retrieving the vertex with the maximum effective-degree will be discussed in Section V-A.

Checking Core Vertices. For efficiently checking whether a vertex is a core vertex, we compute a similar-degree for each vertex $u \in V$, which is defined as follows.

Algorithm 4: CheckCore(u)

```
1  if ed(u) ≥ μ and sd(u) < μ then
2      ed(u) ← d[u]; sd(u) ← 0;
3      for each vertex v ∈ N[u] do
4          Compute σ(u, v);
5          if σ(u, v) ≥ ε then sd(u) ← sd(u) + 1;
6          else ed(u) ← ed(u) − 1;
7          if vertex v has not been explored then
8              if σ(u, v) ≥ ε then sd(v) ← sd(v) + 1;
9              else ed(v) ← ed(v) − 1;
10         if ed(u) < μ or sd(u) ≥ μ then break;
11 Mark v as explored;
```

Definition 5.2: The **similar-degree** of a vertex $u \in V$, denoted $\mathsf{sd}(u)$, is the number of u's neighbors that have been determined to be structure-similar to u.

Intuitively, the similar-degree of a vertex u is a lower bound of the size of its ϵ-neighborhood (*i.e.*, $|N_\epsilon[u]| \geq \mathsf{sd}(u)$); thus, we can determine u to be a core vertex without exploring u if $\mathsf{sd}(u) \geq \mu$. For example, in Figure 5, assume $v_1, \ldots, v_5$ are structure-similar to each other and v_1, v_3, v_4 have been explored, then $\mathsf{sd}(v_2) = \mathsf{sd}(v_5) = 4$; if $\mu \leq 4$ then we can determine v_2 and v_5 to be core vertices without exploring them.

Based on the similar-degrees of vertices, the pseudocode for checking whether a vertex u is a core vertex or not is shown in Algorithm 4. If $\mathsf{ed}(u) < \mu$ then u is not a core vertex, or if $\mathsf{sd}(u) \geq \mu$ then u is a core vertex; note that $\mathsf{ed}(u) \geq \mathsf{sd}(u)$. Otherwise, $\mathsf{ed}(u) \geq \mu > \mathsf{sd}(u)$ (Line 1), and we need to compute structural similarities between u and its neighbors to determine whether u is a core vertex. We reinitialize $\mathsf{ed}(u)$ and $\mathsf{sd}(u)$ (Line 2). For each vertex $v \in N[u]$, we first compute the structural similarity between u and v (Line 4), and update $\mathsf{sd}(u)$ or $\mathsf{ed}(u)$ accordingly (Lines 5–6); that is, if $\sigma(u, v) \geq \epsilon$, then we increase $\mathsf{sd}(u)$ by one, otherwise, we decrease $\mathsf{ed}(u)$ by one. After computing $\sigma(u, v)$, we also update $\mathsf{sd}(v)$ or $\mathsf{ed}(v)$ if v has not been explored (Lines 7–9). Moreover, the algorithm can terminate early once $\mathsf{ed}(u) < \mu$ (*i.e.*, u is a non-core vertex) or $\mathsf{sd}(u) \geq \mu$ (*i.e.*, u is a core vertex) (Line 10).

Note that, at Lines 8–9, we also update the effective-degree and similar-degree of an unexplored vertex. Thus, in this way, a vertex v may be determined to be a core-vertex (*i.e.*, $sd(v) \geq \mu$) or a non-core vertex (*i.e.*, $ed(v) < \mu$) without exploring it.

Clustering Core Vertices. The pseudocode for clustering a core vertex's neighbors that are also core vertices is given in Algorithm 5. We first assign the neighbors of u that are core vertices and structure-similar to u to be in the same cluster as u (Line 1–3). Then, for each of u's neighbor v whose structural similarity to u has not been computed, if u and v have not been assigned to the same cluster yet (Line 5), then we compute $\sigma(u, v)$ (Line 6), and update $\mathsf{sd}(v)$ or $\mathsf{ed}(v)$ if v has not been explored (Lines 7–9); furthermore, we assign u and v to be in the same cluster if v has also been determined to be a core vertex and is structure-similar to u (Line 10). Through Line 5, *we reduce the number of structural similarity computations for vertex-pairs in* $E_{c,c}$.

Clustering Non-core Vertices. Following Lemma 4.2, given the set $\mathbb{C}_c$ of clusters of core vertices in G, non-core vertices in G are assigned to clusters by $\mathbb{C} = \{C_c \bigcup_{u \in C_c} N_\epsilon[u] \mid C_c \in \mathbb{C}_c\}$.

Algorithm 5: ClusterCore(u)

```
1  N′[u] ← {v ∈ N[u] | σ(u, v) has been computed};
2  for each vertex v ∈ N′[u] do
3      if sd(v) ≥ μ and σ(u, v) ≥ ε then union(u, v);
4  for each vertex v ∈ N[u]\N′[u] do
5      if find-subset(u) ≠ find-subset(v) and ed(v) ≥ μ then
6          Compute σ(u, v);
7          if vertex v has not been explored then
8              if σ(u, v) ≥ ε then sd(v) ← sd(v) + 1;
9              else ed(v) ← ed(v) − 1;
10         if sd(v) ≥ μ and σ(u, v) ≥ ε then union(u, v);
```

Algorithm 6: ClusterNoncore

Input: The set $\mathbb{C}_c$ of clusters of core vertices

```
1  ℂ ← ∅;
2  for each C_c ∈ ℂ_c do
3      C ← C_c;
4      for each u ∈ C_c do
5          for each v ∈ N[u] do
6              if sd(v) < μ and v ∉ C then
7                  Compute σ(u, v) if it has not been computed;
8                  if σ(u, v) ≥ ε then C ← C ∪ {v};
9      ℂ ← ℂ ∪ {C};
```

The pseudocode is shown in Algorithm 6; that is, for each $u \in C_c$, we include all vertices in $N_\epsilon[u]$ into the same cluster as C_c. To do so, we iterate through every neighbor v of u (*i.e.*, $v \in N[u]$), and check whether u and v are structure-similar. Note that, here we compute $\sigma(u, v)$ only if v is a non-core vertex and v has not been assigned to the same cluster as C_c. The reason is that, if v is also a core vertex, then whether u and v belong to the same cluster has already been checked in computing $\mathbb{C}_c$ since u is a core vertex. Thus, in this way, *we reduce the number of structural similarity computations for vertex-pairs in* $E_{c,n}$.

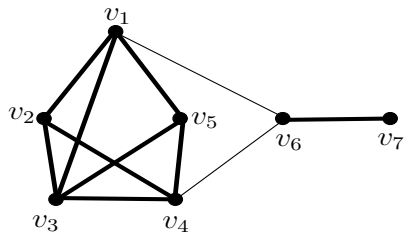


Fig. 5: A graph

Running Example. Consider the graph in Figure 5 with $\epsilon = 0.6$ and $\mu = 4$. The edges (u, v) with $\sigma(u, v) \geq \epsilon$ are shown in thick while other edges are thin edges. Firstly, v_1 is explored and determined to be a core vertex, and effective-degrees and similar-degrees are updated as $\mathsf{ed}(v_6) = 3$, $\mathsf{sd}(v_2) = \mathsf{sd}(v_3) = \mathsf{sd}(v_5) = 1$. Next, v_3 is explored and determined to be a core vertex, $\mathsf{sd}(v_2) = \mathsf{sd}(v_5) = 2$, $\mathsf{sd}(v_4) = 1$; v_1 and v_3 are assigned to the same cluster. Now, v_4 is being explored. As v_2 is structure-similar to v_4, the similar-degree of v_2 is updated to $\mathsf{sd}(v_2) = 3$, and v_2 is a core vertex because v_2 is also structure-similar to itself; thus v_2 and v_4 are assigned to the same cluster. Similarly, v_5 and v_4 are also assigned to the same cluster. Thus, we obtain the cluster $\{v_1, \ldots, v_5\}$ without exploring v_6 and v_7.

A. Complexity Analysis and Implementation Details

Let $E_s \subseteq E$ be the set of adjacent vertex-pairs whose structural similarities have been computed. The time complexity of our pSCAN approach is $O(m \cdot a(n) + \sum_{(u,v)\in E_s} \min\{d[u], d[v]\})$, where $a(n)$ is the extremely slowly growing inverse of the single-valued Ackermann function and is less than 5 for practical values of n [8]; here, the first part of the time complexity is related to disjoint-set data structure operations and the second part is related to structural similarity computations. As proved in [5], $\sum_{(u,v)\in E} \min\{d[u], d[v]\} \leq 2\alpha(G) \cdot m$ where $\alpha(G) \leq \sqrt{m}$ is the arboricity of a graph. Thus, *the worst-case time complexity of our approach is $O(\alpha(G) \cdot m)$, which is the same as the* SCAN *algorithm and is worst-case optimal as discussed in Section III*. Note that, the space complexity of pSCAN is also the same as SCAN (*i.e.*, $O(m + n)$). In the following, we discuss implementation details in order to achieve the above complexities.

Accessing Vertices in Non-increasing Effective-degree Order. At Line 5 of Algorithm 3, we access vertices in non-increasing effective-degree order, where the effective-degree of a vertex is dynamically updated (*e.g.*, in Algorithms 4 and 5). Thus, we need a data structure for iteratively retrieving elements with highest key values where key values of elements are dynamically updated. Fibonacci heap can support these operations; however, the time complexity is still too high and moreover the data structure is too complicated to be practical.

We propose a bin-sort [8] like data structure for this purpose based on the fact that the effective-degrees are integers in the range of 1 to n. In the data structure, we have n bins, one corresponding to each distinct effective-degree value; bin i consists of a doubly-linked list linking together the set of vertices with effective-degree i. Thus, updating the effective-degree of a vertex can be accomplished in $O(1)$ time by removing it from one doubly-linked list and adding it to another one. The total n operations of retrieving the vertex with the maximum effective-degree can be achieved in $O(n)$ time by maintaining the maximum effective-degree of the remaining vertices, which is non-increasing. Thus, the total time complexity related to this data structure is $O(m)$. Moreover, we update the effective-degree of a vertex in the data structure lazily (*i.e.*, only when it is chosen as the next vertex with maximum effective-degree); thus, singly-linked lists are sufficient.

Computing Structural Similarity. To compute the structural similarity between vertices u and v, firstly we need to compute $N[u] \cap N[v]$, then $\sigma(u, v)$ can be computed by Equation (1) in $O(|N[u] \cap N[v]|)$ time. $N[u] \cap N[v]$ can be computed in $O(\min\{d[u], d[v]\})$ time if we create a hash structure to store all edges in E; for example, assume $d[u] \leq d[v]$, then for each $w \in N[u]$, w is in $N[v]$ if and only if (v, w) is in the hash structure which can be tested in constant time. Thus, the total time complexity of computing structural similarities for all pairs of adjacent vertices in E_s is $O(\sum_{(u,v)\in E_s} \min\{d[u], d[v]\})$.

However, building a hash structure incurs both extra time and space which may be a significant overhead. Thus, we consider an alternative approach. We assume that the input graph is stored in the form of adjacency lists (each adjacency list is represented by an array), and vertices in each adjacency list are in increasing order according to their ids; that is, $N[v]$ for each vertex $v \in V$ is a sorted list.[2] Then, $N[u] \cap N[v]$ can be computed by intersecting $N[u]$ and $N[v]$ which can be accomplished in a sort-merge join style in $O(d[u] + d[v])$ time [8]. Note that, although this is theoretically inferior to the above hash structure implementation, our experiments demonstrate that this approach performs better in practice.

VI. Optimization Techniques

In this section, to speed up the checking of structure-similar between two vertices (*i.e.*, whether $\sigma(u, v) \geq \epsilon$), we propose three optimization techniques, cross link, pruning rules, and adaptive structure-similar checking, which will be demonstrated in the following three subsections, respectively.

A. Cross Link

In Algorithm 3, the structural similarity between vertices u and v is computed twice: one in the direction $\sigma(u, v)$ when exploring u and another in the direction $\sigma(v, u)$ when exploring v. Thus, if we link edge (u, v) with edge (v, u), then we only need to compute the structural similarity between u and v once which assigns both $\sigma(u, v)$ and $\sigma(v, u)$.

We propose to build a cross link for each edge which links the two directions of the same edge, based on which the overall number of structural similarity computations is expected to be reduced by half. Since each adjacency list $N[u]$ is an ordered list by vertex ids, for an edge (u, v), its reverse edge (v, u) can be found in $N[v]$ by conducting a binary search on $N[v]$; the time complexity is $\log d[v]$. Thus, the total time complexity of cross link is $O(\sum_{(u,v)\in E} \min\{\log d[u], \log d[v]\})$.

B. Pruning Rule

In Algorithm 3, instead of computing the exact structural similarity between vertices u and v, determining only whether u and v are structure-similar is sufficient. Thus, we propose a pruning rule in the following lemma, for efficiently determining that u and v are not structure-similar (*i.e.*, $\sigma(u, v) < \epsilon$).

Lemma 6.1: (Pruning Rule) *For vertices u and v, if $d[u] < \epsilon^2 \cdot d[v]$ or $d[v] < \epsilon^2 \cdot d[u]$, then $\sigma(u, v) < \epsilon$.*

Proof Sketch: We first prove that if $d[u] < \epsilon^2 \cdot d[v]$ then $\sigma(u, v) < \epsilon$. Given that $d[u] < \epsilon^2 \cdot d[v]$, the structural similarity between u and v is

$$\sigma(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}} \leq \frac{\min\{d[u], d[v]\}}{\sqrt{d[u] \cdot d[v]}} \leq \frac{d[u]}{\sqrt{d[u] \cdot d[v]}} < \frac{d[u]}{\sqrt{d[u] \cdot d[u]/\epsilon^2}} = \epsilon$$

The claim that if $d[v] < \epsilon^2 \cdot d[u]$ then $\sigma(u, v) < \epsilon$ can be proved similarly. □

Note that, this pruning rule is tight. That is, there exist graphs and vertices u and v, such that $d[v] = \epsilon^2 \cdot d[u]$ and $\sigma(u, v) \geq \epsilon$; for example, assume $d[v] = 2$, $d[u] = 8$, $(u, v) \in E$, and $\epsilon = 0.5$, it is easy to verify that $\sigma(u, v) = 0.5 = \epsilon$.

[2] Note that, all the n adjacency lists can be sorted altogether in $O(m)$ time by using radix sort combined with counting sort [8].

Algorithm 7: PruneAndCrossLink

```
1 for each vertex u ∈ V do
2     for each vertex v ∈ N[u] do
3         if d[u] < ε² · d[v] or d[v] < ε² · d[u] then
4             σ(u, v) ← 0;
5             ed(u) ← ed(u) − 1;
6         else Find edge (v, u) in N[v] using binary search,
              and build cross link between (u, v) and (v, u);
```

The pseudocode for pruning and building cross link is shown in Algorithm 7. For each edge $(u, v) \in E$ (Lines 1–2), we first apply the pruning rule in Lemma 6.1. If (u, v) is pruned (Line 3), we assign an arbitrary small value less than ϵ (*e.g.*, 0) to $\sigma(u, v)$ (Line 4), and decrease the effective-degree of u (Line 5). Otherwise, we build the cross link between (u, v) and (v, u). Since the time complexity of applying the pruning rule in Lemma 6.1 for each edge is $O(1)$. The total time complexity of Algorithm 7 is the same as building cross link for all edges, which is $O(\sum_{(u,v)\in E} \min\{\log d[u], \log d[v]\})$.

C. Adaptive Structure-Similar Checking

In this subsection, we propose an adaptive approach to terminating early without computing the exact structural similarity when checking structure-similar between two vertices. First, we compute the minimum number of common neighbors between u and v, denoted $\mathsf{cn}(u, v)$, required for u and v to be structure-similar, in the following lemma.

Lemma 6.2: *Let* $\mathsf{cn}(u, v)$ *be the smallest integer no less than* $\epsilon \cdot \sqrt{d[u] \cdot d[v]}$*; that is,* $\mathsf{cn}(u, v) = \left\lceil \epsilon \cdot \sqrt{d[u] \cdot d[v]} \right\rceil$*. Then,* $\sigma(u, v) \geq \epsilon$ *if and only if* $|N[u] \cap N[v]| \geq \mathsf{cn}(u, v)$*.*

Proof Sketch: Recall that, $\sigma(u, v) = \frac{|N[u] \cap N[v]|}{\sqrt{d[u] \cdot d[v]}}$. Thus $\sigma(u, v) \geq \epsilon$ if and only if $|N[u] \cap N[v]| \geq \epsilon \cdot \sqrt{d[u] \cdot d[v]}$. Moreover, $|N[u] \cap N[v]|$ is an integer, thus $|N[u] \cap N[v]| \geq \left\lceil \epsilon \cdot \sqrt{d[u] \cdot d[v]} \right\rceil = \mathsf{cn}(u, v)$. □

Algorithm 8: CheckStructureSimilar

Input: A graph $G = (V, E)$, vertices u and v, and a parameter $0 < \epsilon \leq 1$; assume $N[w]$ is sorted for each $w \in V$, and let $N[w]_i$ denote the i-th vertex in $N[w]$

Output: Return true if $\sigma(u, v) \geq \epsilon$, and return false otherwise

```
1  cn(u, v) ← ⌈ε · √(d[u] · d[v])⌉;   /* Compute the minimum
   required common neighbors */;
2  cn ← 0; i ← 1; j ← 1; d_u = d[u]; d_v = d[v];
3  while cn < cn(u, v) ≤ min{d_u, d_v} and i ≤ d[u] and j ≤ d[v] do
4      if N[u]_i < N[v]_j then
5          d_u ← d_u − 1; i ← i + 1;
6      else if N[u]_i > N[v]_j then
7          d_v ← d_v − 1; j ← j + 1;
8      else
9          cn ← cn + 1; i ← i + 1; j ← j + 1;
10 if cn < cn(u, v) then return false;
11 else return true;
```

Following Lemma 6.2, we only need to check whether $|N[u] \cap N[v]| \geq \mathsf{cn}(u, v)$ where $\mathsf{cn}(u, v)$ can be computed in $O(1)$ time. The pseudocode of checking structure-similar is shown in Algorithm 8; here, we assume $N[u] \cap N[v]$ is computed in a sort-merge join style. We terminate the checking early once $cn \geq \mathsf{cn}(u, v)$ (*i.e.*, we have found at least $\mathsf{cn}(u, v)$ common neighbors of u and v, thus $\sigma(u, v) \geq \epsilon$)), or $d_u < \mathsf{cn}(u, v)$ or $d_v < \mathsf{cn}(u, v)$ (*i.e.*, there is not enough vertex remaining for u and v to have at least $\mathsf{cn}(u, v)$ common neighbors, thus $\sigma(u, v) < \epsilon$). Algorithm 8 has the same time complexity as computing the structural similarity between u and v (*i.e.*, $O(d[u] + d[v])$).

VII. Experiments

We conduct extensive empirical studies to evaluate the efficiency of our proposed approaches for structural graph clustering. In specific, we evaluate the following algorithms:

- SCAN-HS: the algorithm in [27]; here, we use the hash structure (HS) based method for structural similarity computation as discussed in Section V-A.
- SCAN++: the state-of-the-art algorithm in [22].
- pSCAN-HS: the approach discussed in Section V with the hash structure (HS) based method for structural similarity computation.
- pSCAN: the approach discussed in Section V with the sort-merge join method for structural similarity computation.
- pSCAN-C: the pSCAN approach with the cross link optimization in Section VI.
- pSCAN-CR: the pSCAN approach with both cross link and pruning rule optimizations in Section VI.
- pSCAN*: the pSCAN approach with all optimizations in Section VI.

All algorithms are implemented in C++ and compiled with GNU GCC with the -O2 optimization; the program of SCAN++ is obtained from the authors in [22] and all other algorithms are implemented by us. All experiments are conducted on a machine with an Intel(R) Xeon(R) 2.9GHz CPU and 32GB memory running Linux. All programs are run as single-threaded programs with the entire graph held in the main memory. We evaluate the performance of all algorithms on both real and synthetic graphs as follows.

TABLE I: Statistics of real graphs ($\bar{d}$: average degree, c: average clustering coefficient)

Graph	#Edges	#Vertices	$\bar{d}$	c
ca-GrQc	13,422	4,158	6.46	0.557
ca-CondMat	91,286	21,363	8.55	0.642
email-EuAll	339,925	224,832	3.02	0.079
soc-Epinions	405,739	75,877	10.69	0.138
slashdot	468,554	77,350	11.12	0.055
dblp	1,049,866	317,080	6.62	0.632
amazon0601	2,443,311	403,364	12.11	0.418
web-Google	3,074,322	665,957	9.23	0.459
wiki-Talk	4,656,682	2,388,953	3.90	0.053
as-Skitter	11,094,209	1,694,616	13.09	0.258
LiveJournal	42,845,684	4,843,953	17.69	0.274
uk-2002	261,556,721	18,459,128	28.34	0.603
twitter-2010	1,202,513,344	41,652,230	57.7	0.073

Real Graphs. We evaluate the algorithms on thirteen real graphs, which are downloaded from the Stanford Network

Analysis Platform [3] and the Laboratory of Web Algorithmics [4]; descriptions of these graphs can also be found there. Sizes of these graphs are shown in Table I, where the last two columns show the average degree and the average clustering coefficient, respectively. The graphs in Table I are sorted in increasing order regarding the number of edges.

TABLE II: Statistics of LFR graphs

Graph	#Edges	#Vertices	$\overline{d}$	c
LFR1	979,573	100,000	19.59	0.113
LFR2	974,076	100,000	19.48	0.213
LFR3	977,008	100,000	19.54	0.345
LFR4	977,008	100,000	19.54	0.459
LFR5	975,986	100,000	19.50	0.534
LFR6	978,117	100,000	19.56	0.601
LFR7	980,295	100,000	19.60	0.700
LFR8	9,755	1,000	19.50	0.400
LFR9	97,396	10,000	19.46	0.461
LFR10	9,784,242	1,000,000	19.56	0.457
LFR11	97,836,053	10,000,000	19.56	0.457

Synthetic Graphs. We also evalute the algorithms on LFR benchmark graphs [18] by varying the number of vertices and the average clustering coefficients of the graphs generated. The number of vertices varies from 10^3 to 10^7, and the average clustering coefficient varies from 0.1 to 0.7 according to that of the real graphs in Table I. Similar to [22], we fix the average degree and the maximum degree at 20 and 50, respectively. We generate eleven LFR graphs, LFR1, ..., LFR11, whose sizes are shown in Table II.

Parameters. For each tested graph, we vary two parameters, $0 < \epsilon \leq 1$ and $\mu \geq 2$ in a similar way to [22], for testing. For ϵ, we choose 0.2, 0.4, 0.6, and 0.8, with $\epsilon = 0.6$ as default. For μ, we choose 2, 5, 10, and 15, with $\mu = 5$ as default.

Metrics. For each testing, we run the algorithm three times and report the average CPU time.

A. Performance Studies on Real-world Graphs

Due to space limits, we only show the results on four default graphs, ca-CondMat, soc-Epinions, amazon0601, and LiveJournal; the results on other graphs have similar trends.

Eval-I: Comparing pSCAN* with SCAN-HS and SCAN++. We compare our best algorithm, pSCAN*, with the existing algorithms, SCAN-HS and SCAN++, by varying ϵ and μ.

Varying ϵ. The running time of SCAN-HS, SCAN++, pSCAN* by varying ϵ is illustrated in Figure 6. The running time of SCAN-HS is steady for different ϵ values due to exhaustively computing all structural similarities and thus irrelevant to ϵ. SCAN++ takes slightly more time for larger ϵ. The reason is that for larger ϵ, it is less likely for a vertex to be a core vertex and thus less likely to prune the structural similarity computation between vertices (say, u and v) that are common neighbors of a vertex and its two-hop-away vertices. Note that, the computation of structural similarity between vertices in the common neighbors can be avoided only if there is a core vertex in the common neighbors [22]. pSCAN* runs faster for larger ϵ due to our three optimization techniques in Section VI. When ϵ increases, two adjacent vertices are more likely to be not structure-similar to each other; this can be pruned by our pruning rule without computing the actual structural similarity.

[3] http://snap.stanford.edu/

[4] http://law.di.unimi.it/datasets.php

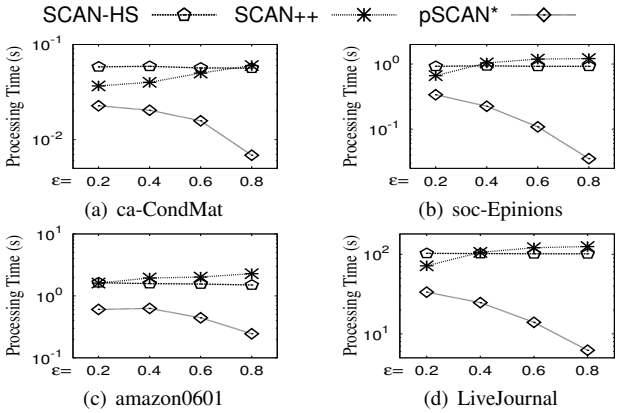


Fig. 6: (Eval-I) Against Existing Algorithms (vary ϵ)

Overall, pSCAN* is significantly faster than SCAN-HS and SCAN++ ($\geq 10\times$ faster for $\epsilon = 0.8$). SCAN-HS has similar performance as SCAN++, and sometimes even runs faster than SCAN++, contrary to what is concluded in [22]. The reason is that our implementation of SCAN-HS has a time complexity of $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$, while a naive implementation of structural similarity computation in SCAN will result in a time complexity of at least $O(\sum_{(u,v)\in E} \max\{d(u), d(v)\}) = O(\sum_{v\in V}(d(v))^2)$.

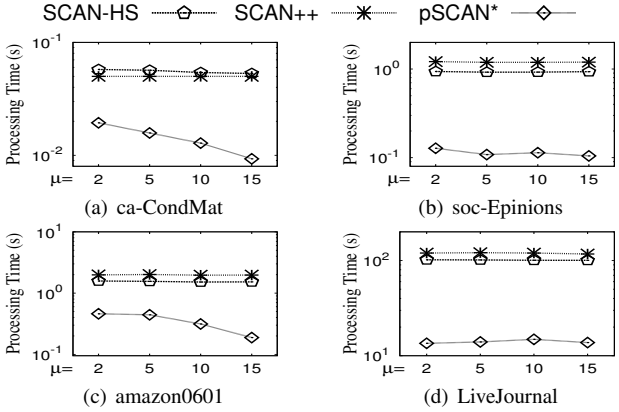


Fig. 7: (Eval-I) Against Existing Algorithms (vary μ)

Varying μ. The results of the three algorithms by varying μ are shown in Figure 7. Both SCAN-HS and SCAN++ perform quite steady regarding the different values of μ. The running time of pSCAN* decreases when μ increases. This is because, when μ increases, more vertices can be pruned to be core vertices by the effective-degree (see Section V) in our approach; thus less structural similarity computations between core vertices and faster running time. Hence, pSCAN* significantly outperforms both SCAN-HS and SCAN++.

Eval-II: Evaluating Our New Paradigm. In this testing, we evaluate the efficiency of our new paradigm. In particular, we compare pSCAN and pSCAN-HS with SCAN-HS.

Varying ϵ. Figure 8 presents the performances of pSCAN, pSCAN-HS, and SCAN-HS by varying ϵ. Generally, the running time of pSCAN and pSCAN-HS increases when ϵ becomes larger. This is because it is more costly to check whether a vertex is a core vertex; that is, more structural similarities involving the vertex need to be computed. From Figure 8, we can see that both pSCAN and pSCAN-HS outperforms the baseline

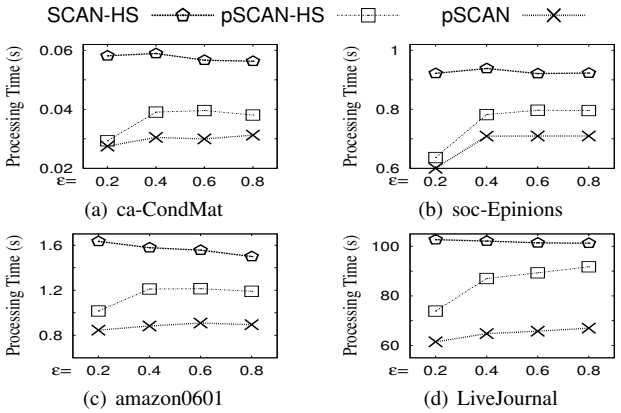


Fig. 8: (Eval-II) Evaluate Our New Paradigm (vary ϵ)

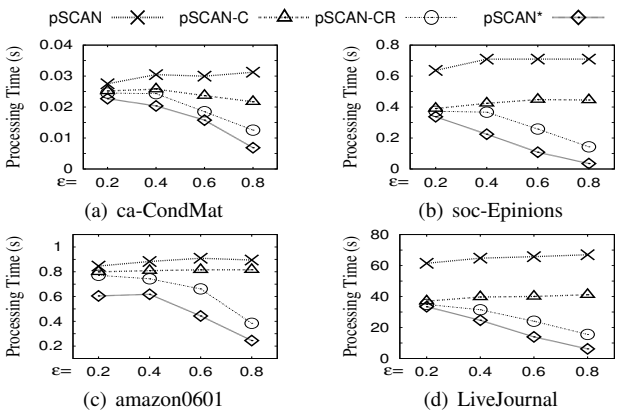


Fig. 10: (Eval-III) Evaluate Optimization Techniques (vary ϵ)

algorithm, SCAN-HS. Note that, pSCAN-HS and SCAN-HS use the same approach for structural similarity computation. Thus, this demonstrates that our new paradigm can reduce the number of structural similarity computations. Moreover, pSCAN outperforms pSCAN-HS; the only difference of these two algorithms is that pSCAN uses sort-merge join to compute structural similarities while pSCAN-HS uses a hash structure which incurs a non-neglectable constant overhead.

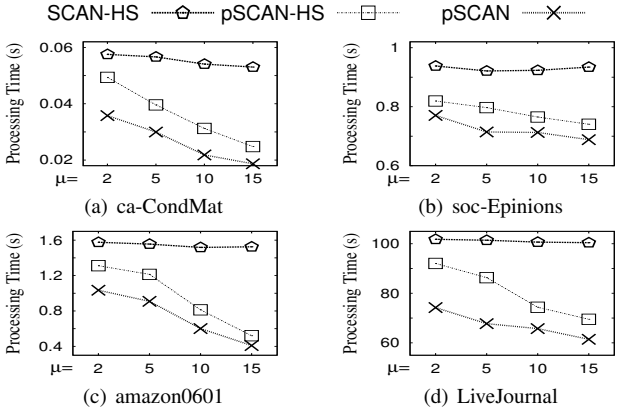


Fig. 9: (Eval-II) Evaluate Our New Paradigm (vary μ)

Varying μ. The results of SCAN-HS, pSCAN-HS, and pSCAN by varying μ are illustrated in Figure 9. The trends of pSCAN and pSCAN-HS are similar to that of pSCAN* in Figure 7. Overall, pSCAN outperforms pSCAN-HS which in turn outperforms SCAN-HS. *Thus, in the following, we only consider the sort-merge join approach for structural similarity computation, and we also ignore* SCAN-HS *from our testings.*

Eval-III: Evaluating Our Optimization Techniques. We evaluate the effect of our three optimization techniques, proposed in Section VI, on the performance of our pSCAN algorithm. Specifically, we compare pSCAN with pSCAN-C, pSCAN-CR, and pSCAN*. pSCAN-C is obtained from pSCAN by adding the cross link optimization, and pSCAN-CR has an additional pruning rule optimization, while pSCAN* integrates all three optimizations.

Varying ϵ. The experimental results of evaluating our optimization techniques by varying ϵ are shown in Figure 10. pSCAN-C has similar trends to pSCAN, since pSCAN-C only improves upon pSCAN by the cross link optimization which may potentially reduce the number of structural similarity computations by half. Both pSCAN-CR and pSCAN* runs faster for larger ϵ. This is because more structural similarity computations can be pruned by the pruning rule for larger ϵ. Overall, pSCAN-C outperforms pSCAN due to cross link optimization, pSCAN-CR outperforms pSCAN-C due to the pruning rule optimization, and pSCAN* outperforms pSCAN-CR due to the adaptive structure-similar checking optimization. In other words, each of the three optimization techniques in Section VI can improve the performance of the pSCAN algorithm.

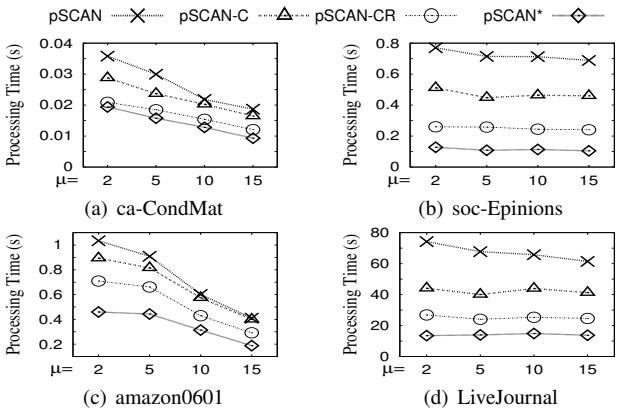


Fig. 11: (Eval-III) Evaluate Optimization Techniques (vary μ)

Varying μ. Figure 11 illustrates the performance of our algorithms with respect to different μ values. In general, all the four algorithms run faster for larger μ. This is because more vertices are non-core vertices and can be pruned based on our effective-degree maintenance in Section V. Still, pSCAN* outperforms pSCAN-CR which in turn outperforms pSCAN-C, with pSCAN performing the worst.

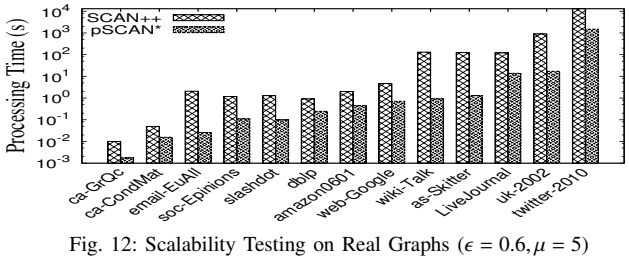


Fig. 12: Scalability Testing on Real Graphs ($\epsilon = 0.6, \mu = 5$)

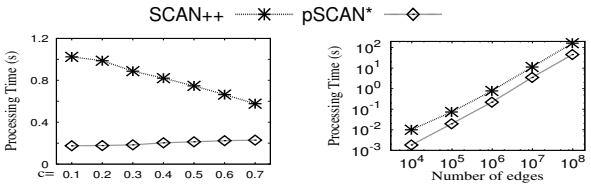
Eval-IV: Scalability Testing. Here, we test the scalability of our best algorithm, pSCAN*, against the state-of-the-art algorithm, SCAN++, on real graphs. The results are shown in Figure 12. The thirteen graphs are organized in increasing order regarding the number of edges. Generally, the running

time of both SCAN++ and pSCAN* increases along with the increasing of the number of edges in a graph. Nevertheless, pSCAN* consistently outperforms SCAN++. Noticeably, on three of the tested graphs, email-EuAll, wiki-Talk, and as-Skitter, our pSCAN* algorithm outperforms the state-of-the-art algorithm, SCAN++, by two orders of magnitude. Moreover, our pSCAN* algorithm scales well to large graphs; for example, for the twitter graph with 1 billion edges with $\epsilon = 0.6$ and $\mu = 5$, pSCAN* computes the structural graph clustering in 25 minutes while SCAN++ cannot finish even after 24 hours.

B. Performance Studies on Synthetic Graphs

In this subsection, we evaluate the performance of our pSCAN* algorithm against the state-of-the-art algorithm, SCAN++, on synthetic graphs.

Eval-V: Comparing pSCAN* with SCAN++ on LFR Graphs. The results of running these two algorithm on LFR graphs by varying the average clustering coefficient is shown in Figure 13(a), where x-axis shows the average clustering coefficient (c) corresponding to graphs LFR1, ..., LFR7 in Table II. We can see that, the running time of SCAN++ drops when c becomes larger; this conforms with that in [22]. Our pSCAN* algorithm takes relatively longer time for graphs with larger clustering coefficients; this is because more structural similarity computations are needed. Nevertheless, our pSCAN* still runs much faster than SCAN++ even for very large average clustering coefficient (*e.g.*, $c = 0.7$).



(a) Vary clustering coefficient (b) Vary #Edges ($c = 0.4$)

Fig. 13: (Eval-V) Evaluate on LFR Graphs ($\epsilon = 0.6$, $\mu = 5$)

The experimental results by varying the numbers of edges (thus also vertices) of LFR graphs are shown in Figure 13(b). Both algorithms scale well with the number of vertices, and our pSCAN* algorithm consistently outperforms SCAN++.

VIII. Conclusion

In this paper, we developed a new paradigm for structural graph clustering based on our three observations. Based on the paradigm, we proposed a new approach aiming to reduce the number of structural similarity computations. To improve the performance of our approach, we further proposed three optimization techniques to speed up the checking of structure-similar between two vertices. We proved that the existing SCAN approach as well as our new approach are worst-case optimal. Experiments show that our approach outperforms the existing approaches by over one order of magnitude.

Acknowledgment

Lijun Chang is supported by ARC DE150100563 and ARC DP160101513. Xuemin Lin is supported by NSFC61232006, ARC DP140103578 and ARC DP150102728. Lu Qin is supported by ARC DE140100999 and ARC DP160101513. Wenjie Zhang is supported by ARC DP150103071 and ARC DP150102728.

References

[1] L. Chang, J. X. Yu, and L. Qin. Fast maximal cliques enumeration in sparse graphs. *Algorithmica*, 66(1), 2013.

[2] L. Chang, J. X. Yu, L. Qin, X. Lin, C. Liu, and W. Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proc. of SIGMOD'13*, 2013.

[3] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *Proc. of ICDE'11*, 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 *Proc. of SIGMOD'10*, 2010.

[5] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1), 1985.

[6] A. Clauset. Finding local community structure in networks. *Phys. Rev. E*, 72, 2005.

[7] A. Clauset, M. E. J. Newman, , and C. Moore. Finding community structure in very large networks. *Physical Review E*, 2004.

[8] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[9] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *Proc. of SIGMOD'13*, 2013.

[10] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *Proc. of SIGMOD'14*, 2014.

[11] C. H. Q. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *Proc. of ICDM'01*, 2001.

[12] S. Fortunato. Community detection in graphs. *CoRR*, abs/0906.0612, 2009.

[13] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press.

[14] R. Guimerà and L. A. Nunes Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028), February 2005.

[15] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *Proc. of SIGMOD'14*, 2014.

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

[17] U. Kang and C. Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *Proc. of ICDM'11*, 2011.

[18] A. Lancichinetti and S. Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys. Rev. E*, 80, Jul 2009.

[19] S. Lim, S. Ryu, S. Kwon, K. Jung, and J. Lee. Linkscan*: Overlapping community detection using the link-space transformation. In *Proc. of ICDE'14*, 2014.

[20] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review*, E 69(026113), 2004.

[21] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8), 2000.

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

[23] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *Proc. of KDD'10*, 2010.

[24] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *Proc. of ICDE'14*, 2014.

[25] N. Wang, J. Zhang, K.-L. Tan, and A. K. H. Tung. On triangulation-based dense neighborhood graph discovery. *Proc. VLDB Endow.*, 4(2), Nov. 2010.

[26] D. Watts and S. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393), 1998.

[27] X. Xu, N. Yuruk, Z. Feng, and T. A. J. Schweiger. SCAN: a structural clustering algorithm for networks. In *Proc. of KDD'07*, 2007.

[28] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Out-of-core coherent closed quasi-clique mining from large dense graph databases. *ACM Trans. Database Syst.*, 32(2), June 2007.

[29] Y. Zhang and S. Parthasarathy. Extracting analyzing and visualizing triangle k-core motifs within networks. In *Proc. of ICDE'12*, 2012.