Noname manuscript No.
(will be inserted by the editor)

Top-K Structural Diversity Search in Large Networks

Xin Huang · Hong Cheng · Rong-Hua Li · Lu Qin · Jeffrey Xu Yu

the date of receipt and acceptance should be inserted later

Abstract Social contagion depicts a process of information (e.g., fads, opinions, news) diffusion in the online social networks. A recent study reports that in a social contagion process the probability of contagion is tightly controlled by the number of connected components in an individual's neighborhood. Such a number is termed *structural diversity* of an individual and it is shown to be a key predictor in the social contagion process. Based on this, a fundamental issue in a social network is to find top-k users with the highest structural diversities. In this paper, we, for the first time, study the top-k structural diversity search problem in a large network. Specifically, we study two types of structural diversity measures, namely, component-based structural diversity measure and core-based structural diversity measure. For component-based structural diversity, we develop an effective upper bound of structural diversity for pruning the search space. The upper bound can be incrementally refined in the search process. Based on such upper bound, we propose an efficient framework for top-k structural diversity search. To further speed up the structural diversity evaluation in the search process, several carefully devised search strategies are proposed. We also design efficient techniques to handle frequent updates in dynamic networks and maintain the top-k results. We further show how the techniques proposed in component-based structural diversity measure can be extended to handle the core-based structural diversity measure. Extensive experimental studies are conducted in real-world large networks and synthetic graphs, and the results demonstrate the efficiency and effectiveness of the proposed methods.

Keywords Structural diversity · Disjoint-set forest · A* search · Dynamic graph

X. Huang · H. Cheng · J. X. Yu
Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong
New Territories, Hong Kong
E-mail: {xhuang, hcheng, yu}@se.cuhk.edu.hk

R. Li
Guangdong Province Key Laboratory of Popular High Performance Computers
Shenzhen University, China
E-mail: rhli@se.cuhk.edu.hk

L. Qin
Quantum Computation and Intelligent Systems
Department of Engineering and Information Technology
University of Technology, Sydney
E-mail: lu.qin@uts.edu.au

1 Introduction

Recently, online social networks such as Facebook, Twitter and LinkedIn have attracted growing attention in both industry and research communities. Online social networks are becoming more and more important media for users to communicate with each other and to spread information in the real world [17]. In an online social network, the phenomenon of information diffusion, such as diffusion of fads, political opinions, and the adoption of new techniques, has been termed *social contagion* [25], which is a similar process as epidemic diseases.

Traditionally, the models of social contagion are based on analogies with biological contagion, where the probability that a user is influenced by the contagion grows monotonically with the number of his or her friends who have been affected already [10,3,26]. However, such models have recently been challenged [22,25], as the social contagion process is typically more complex and the social decision can depend more subtly on the network structure. Ugander et al. [25] study two social contagion processes in Facebook: the process that a user joins Facebook in response to an invitation email from an existing Facebook user (recruitment), and the process that a user becomes an engaged user after joining (engagement). They find that the probability of contagion is tightly controlled by the number of connected components in a user's neighborhood, rather than by the number of friends in the neighborhood. A connected component represents a distinct social context of a user, and the multiplicity of social contexts is termed *structural diversity*. A user is much more likely to join Facebook if he or she has a larger structural diversity, i.e., a larger number of distinct social contexts. This finding reveals that the structural diversity of a user is an important factor in the social contagion process. As suggested in [25], the analysis of structural diversity in a social network can be beneficial to a wide range of application domains. For example, in a political campaign, to convince individuals to change their attitude, it is obviously more important that they receive messages from multiple directions than that they receive many endorsements [25]. In the promotion of health practices, we can find such top users with the highest structural diversity, and inject vaccine for them for reducing their influenced probability. In the marketing, to promote a new product, we can find such top customers as the first priority.

Among all of these applications, a fundamental problem is to find the individuals in a social network with high structural diversity [25]. Motivated by this, in this paper, we study a problem of finding top-k individuals with the highest structural diversity in a social network. Following the definition in [25], the structural diversity of a node u is the number of connected components in a subgraph induced by u's immediate neighbors. Take the network in Figure 1 (a) as an example. The structural diversity of vertex f is 2, as the induced subgraph by f's neighbors shown in Figure 1 (b) has two connected components. This structural diversity definition has been shown to be a good predictor for the recruitment study on Facebook in [25]. However, it may fall short in some other scenarios. For example, in the engagement study, the friendship neighborhoods on Facebook are significantly larger than the email contact neighborhoods from the recruitment study. In such a situation, a large number of one-node components, or "singletons", is not an accurate reflection of social context diversity.

To address this problem, [25] proposed two distinct parametric generalizations of the component count. First, it measures diversity simply by counting only components over a certain size t. This is called *component-based structural diversity*. Second, it measures diversity by the component count of the t-core of the neighborhood graph, where a t-core is the subgraph formed by repeatedly deleting all vertices of degree less than t. This measure is called *core-based structural diversity*. We have studied the problem of top-k component-based structural diversity search in our previous work [14]. To have a comprehensive investigation of the structural diversity search problem, we further extend our study by adopting the core-based structural diversity in this work. The core-based structural diversity measure has been proven effective through case studies in [25], as the t-core notion can exclude small and loose components more effectively.

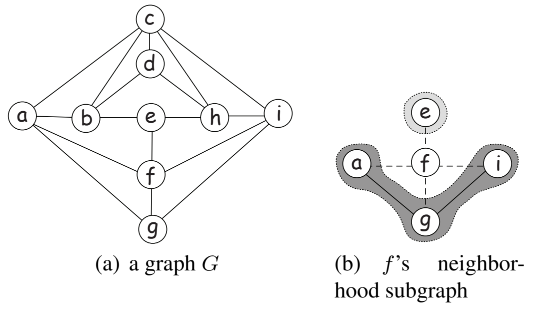


Fig. 1 An example of component-based structural diversity

To solve the top-k structural diversity search problem, a naive method is to compute the structural diversity for all the vertices and then return the top-k vertices. Clearly, such a naive method is too expensive. To efficiently find the top-k vertices, the idea of traditional top-k query processing techniques [15] can be used, which finds the top-k answers according to some search order, and prunes the search space based on some upper bound score. Following this framework, in our problem, we have to address two key issues: (1) how to develop an effective upper bound for the structural diversity of a vertex, and (2) how to devise a good search order in the computation.

In this paper, we propose several efficient and effective techniques to address these issues. For the component-based structural diversity measure, we find that for two vertices connected by an edge, some structural information of them can be shared. For example, in Figure 1 (b), vertex e forms a component of size 1 in f's neighborhood. From this fact, we can infer that vertex f also forms a component of size 1 in e's neighborhood. Based on this important observation, the structural diversity computation for different vertices can also be possibly shared. To achieve this, we design a Union-Find-Isolate data structure to keep track of the known structural information of a vertex so as to avoid the computation of structural diversity for every vertex. A novel upper bound of the structural diversity is developed for pruning unpromising vertices effectively. Interestingly, the upper bound can be incrementally refined in the search process to become increasingly tighter. Based on the upper bound and the Union-Find-Isolate data structure, we propose a novel Top-k-search framework for top-k structural diversity search.

Beyond this, we explore how to apply our Top-k-search framework to support the core-based structural diversity measure. We find that this definition brings new structural properties which are different from those of the component-based definition. Thus our proposed Union-Find-Isolate data structure and upper bound are not applicable to the core-based structural diversity measure. We study new properties of this measure and leverage it to design a new upper bound. We also propose an efficient algorithm for computing the core-based structural diversity score, and finally integrate these new techniques into our Top-k-search framework. This study demonstrates that our Top-k-search framework can be generalized to handle different instantiations of the structural diversity measure.

The main contributions of our study are summarized as follows.

– We study top-k structural diversity search for the first time by adopting two measures, i.e., the component-based and core-based structural diversity. Structural diversity has been proven to be a positive predictor in social contagion [25]. We develop a novel Top-k-search framework to efficiently identify the individuals that play a key role in social contagion.
– For the component-based structural diversity measure, we design a Union-Find-Isolate data structure to keep track of the known structural information during the computation, and an effective upper bound for pruning. We devise a useful search order to traverse the components in a vertex's neighborhood. According to this search order, we propose a novel A* search based algorithm to compute the structural diversity of a vertex.
– We also design efficient techniques to handle frequent updates in dynamic networks and maintain the top-k results. We use the Union-Find-Isolate structure and a spanning tree structure to efficiently handle edge insertions and deletions respectively.
– For the core-based structural diversity measure, a new upper bound and an efficient search algorithm are designed.
– We conduct extensive experimental studies on large real networks to show the efficiency of our proposed methods. We also conduct case studies on DBLP and a word association network, which show that structural diversity is useful for identifying ambiguous names in DBLP and finding words with diverse meanings in the word association network.

The rest of this paper is organized as follows. We formulate the top-k structural diversity search problem in Section 2, and then discuss and compare the component-based and core-based structural diversity measures in Section 3. For the component-based measure, we first present a simple degree-based algorithm in Section 4, and then design a novel and efficient Top-k-search framework in Section 5. We design two useful search strategies in Section 6 and discuss update in dynamic networks in Section 7. For the core-based measure, we design a new upper bound and an efficient search algorithm in Section 8. Extensive experimental results are reported in Section 9. We discuss related work in Section 10 and conclude this paper in Section 11.

2 Problem Definition

Consider an undirected and un-weighted graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. Denote by $N(v)$ the set of neighbors of a vertex v, i.e., $N(v) = \{u \in V : (v, u) \in E\}$, and by $d(v) = |N(v)|$ the degree of v. Let d_{max} be the maximum degree of the vertices in G. Given a subset of vertices $S \subseteq V$, the induced subgraph of G by S is defined as $G_S = (V_S, E_S)$, where $V_S = S$ and $E_S = \{(v, u) : v, u \in S, (v, u) \in E\}$. The neighborhood induced subgraph is defined as follows.

Definition 1 (Neighborhood Induced Subgraph) For a vertex $v \in V$, the neighborhood induced subgraph of v, denoted by $G_{N(v)}$, is a subgraph of G induced by the vertex set $N(v)$.

Consider the graph in Figure 1 (a). For vertex f, the set of neighbors is $N(f) = \{a, e, g, i\}$. The neighborhood induced subgraph of f is $G_{N(f)} = (\{a, e, g, i\}, \{(a, g), (g, i)\})$, as shown in Figure 1 (b). We define the structural diversity of a vertex as follows.

Definition 2 (Component-based Structural Diversity [25]) Given an integer t where $1 \leq t \leq n$, the structural diversity of a vertex $v \in V$, denoted by $score(v)$, is the number of connected components in $G_{N(v)}$ whose size measured by the number of vertices is larger than or equal to t. t is called the component size threshold.

$G_{N(f)}$ in Figure 1 (b) contains a size-1 connected component $\{e\}$ and a size-3 connected component $\{a, g, i\}$. If $t = 1$, then $score(f) = 2$. Alternatively, if $t = 2$, $score(f) = 1$ as there is only one component $\{a, g, i\}$ whose size is no less than 2.

Ugander et al. [25] gave another definition of structural diversity based on the core subgraph concept [6]. Their study showed that, the core subgraph based definition suffices to provide a positive predictor of future long-term engagement in a social network.

A t-core of a graph is the largest subgraph in which every vertex is connected to at least t vertices within the subgraph. Note that a t-core subgraph may be disconnected and have several components. For instance, consider a graph G in Figure 2 (a). The entire graph is a 2-core, and the subgraph inside the dashed circle is a 3-core. As another example, the neighborhood induced subgraph $G_{N(e)}$ in Figure 2 (b) is a 1-core containing 2 connected components $\{a, b, c, d\}$ and $\{f, g, h\}$. Based on the t-core subgraph, we define the structural diversity of a vertex as follows.

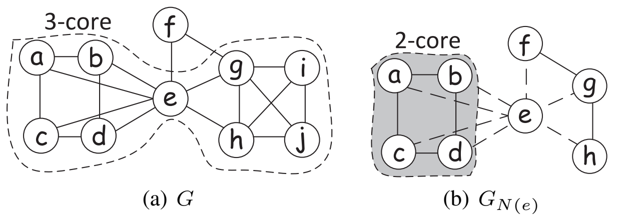


Fig. 2 An example of core-based structural diversity

Definition 3 (Core-based Structural Diversity [25]) Given an integer t where $1 \leq t \leq n$, the structural diversity of vertex $v \in V$, denoted by $score^*(v)$, is measured by the number of connected components in the t-core of $G_{N(v)}$. t is called the core value threshold.

Consider $G_{N(e)}$ in Figure 2 (b). If $t = 1$, there are two connected components $\{a, b, c, d\}$ and $\{f, g, h\}$ in the 1-core, so $score^*(e) = 2$. But if $t = 2$, there is only one component $\{a, b, c, d\}$ in the 2-core of $G_{N(e)}$, so $score^*(e) = 1$ in this case.

Based on the two different structural diversity definitions above, we can formulate our top-k component-based and core-based structural diversity search problems, which are respectively denoted as CC-TopK and Core-TopK.

Problem 1 (CC-TopK): Given a graph G and two integers k and t where $1 \leq k, t \leq n$, top-k structural diversity search is to find a set of k vertices in G with the highest structural diversity w.r.t. the component size threshold t.

Let us re-consider the example in Figure 1 for CC-TopK. Suppose that $k = 1$ and $t = 1$. Then, $\{e\}$ is the answer, as e is the vertex with the highest structural diversity ($score(e) = 3$).

Problem 2 (Core-TopK): Given a graph G and two integers k and t where $1 \leq k, t \leq n$, top-k structural diversity search is to find a set of k vertices in G with the highest core-based structural diversity w.r.t. the core value threshold t.

It is important to note that although we focus on the top-k structural diversity search, the proposed techniques can also be easily extended to process the iceberg query, which finds all vertices whose structural diversity is greater than or equal to a pre-specified threshold. Unless otherwise specified, in the rest of this paper, we assume that a graph is stored in the adjacency list representation. Each vertex is assigned a unique ID. In addition, for convenience, we assume that $m \in \Omega(n)$, which does not affect the complexity analysis of the proposed algorithms. Similar assumption has been made in [18].

3 Problem Comparison

In this section, we discuss and compare the problems of CC-TopK and Core-TopK in terms of measure definition, computational cost, and result quality.

Measure definition: For the core-based structural diversity, every component of a t-core subgraph has at least $t + 1$ vertices, i.e., it forms a size-$(t + 1)$ connected component. When $t = 0$, the core-based structural diversity score is equivalent to the component-based structural diversity, which is simply the component count of the original graph; when $t = 1$, the core-based structural diversity score is equivalent to the component-based structural diversity (with a component size of at least 2) in Definition 2; when $t > 1$, a component in a t-core subgraph is more cohesive than a size-$(t + 1)$ connected component in Definition 2, due to the t-core definition that every vertex is connected to at least t vertices in the t-core. Thus, all tree-like components will be discarded and the remaining components are counted for the core-based structural diversity score. On the other hand, the tree-shaped structure may exist and be counted for the component-based structural diversity for any t.

Computational cost: Compared with the component-based structural diversity, the core-based structural diversity additionally requires to compute the t-core and remove unqualified components. Thus, computing Core-TopK is more costly than CC-TopK.

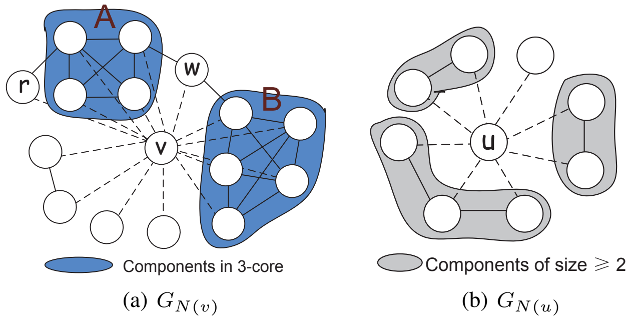


Fig. 3 Comparison of Core-TopK and CC-TopK

Result quality: Compared with the component-based structural diversity which only imposes a constraint of connectivity, the core-based structural diversity considers both the size and cohesiveness of each component. Thus the core-based definition can help identify densely connected and more meaningful and distinct social contexts among a user's friends. For example, Figure 3(a) shows the $G_{N(v)}$ of node v containing 15 nodes, 11 of which are connected in one component. If we apply the component-based structural diversity on $G_{N(v)}$ with $t = 3$, the component with the 11 nodes is counted. However, this large component is loosely connected through node w. But if we apply the core-based structural diversity on $G_{N(v)}$ with $t = 3$, two components A and B in the 3-core of $G_{N(u)}$ can be discovered as shown in the shadow regions of Figure 3(a). Each node in A and B has at least 3 neighbors in the corresponding component, which are densely connected. Obviously, in such a case, the core-based structural diversity can capture the two dense social contexts A and B more precisely than the component-based structural diversity.

On the other hand, the component-based structural diversity is more suitable for analyzing the social context diversity for nodes whose neighbors are not densely connected, since very few results can be discovered by the core-based structural diversity in this case. For example, Figure 3(b) shows the $G_{N(u)}$ of node u containing 8 nodes. If we apply the component-based structural diversity with $t = 2$, we can find three connected components of size no less than 2 in $G_{N(u)}$ as marked in the shadow regions. However, if we apply the core-based structural diversity with $t = 2$, no component can be found. Therefore, the component-based diversity is better than the core-based structural diversity in such a case.

In summary, CC-TopK is simpler. However, it does not consider the closeness of members in each component. Core-TopK has more constraints by considering both cohesiveness and size. However, it is more difficult to compute and may lose the information of vertices that do not participate in a cohesive subgraph. Therefore, both definitions have advantages and disadvantage, and they can be jointly used to discover more social context diversity information in a large network. More comparisons and meaningful results for both CC-TopK and Core-TopK using real-world networks can be found in the case studies in our experiments.

4 A Simple Degree-Based Approach for CC-TopK

In this section, we present a simple degree-based algorithm for top-k component-based structural diversity search. To compute the structural diversity $score(v)$ for a vertex v, we can perform a breadth-first search in $G_{N(v)}$ to find connected components and return the number of components whose sizes are no less than t. We call this procedure bfs-search, the pseudocode of which is omitted for brevity.

Next we introduce a useful lemma which leads to a pruning strategy in the degree-based algorithm.

Lemma 1 *For any vertex v in G, $score(v) \leq \lfloor \frac{d(v)}{t} \rfloor$ holds.*

Proof We prove this lemma by contradiction. Suppose to the contrary that $score(v) > \lfloor \frac{d(v)}{t} \rfloor$. By the definition of structural diversity, $G_{N(v)}$ has $\lfloor \frac{d(v)}{t} \rfloor + 1$ or more components whose size is greater than or equal to t. Then, the total number of vertices in these components is $\geq (\lfloor \frac{d(v)}{t} \rfloor + 1) \cdot t > \frac{d(v)}{t} \cdot t = d(v)$, which contradicts to the fact that the number of vertices in $G_{N(v)}$ is $d(v)$. Hence, the lemma is established.

We denote $\lfloor \frac{d(v)}{t} \rfloor$ by $\overline{bound}(v)$. Equipped with Lemma 1 and the bfs-search procedure, we present the degree-based approach in Algorithm 1, which computes the structural diversity of the vertices in descending order of their degree. After initialization (lines 1-2), Algorithm 1 sorts the vertices in descending order of their degree (line 3). Then it iteratively finds the unvisited vertex v^* with the maximum degree, and calculates $\overline{bound}(v^*)$ (lines 5-6). If the answer set $\mathcal{S}$ has k vertices and $\overline{bound}(v^*) \leq \min_{v\in\mathcal{S}} score(v)$, the algorithm terminates (lines 7-8). The rationale is as follows. By Lemma 1, we have $score(v^*) \leq \overline{bound}(v^*) \leq \min_{v\in\mathcal{S}} score(v)$. For any vertex $w \in V$ with a smaller degree, we have $score(w) \leq \overline{bound}(w) \leq \overline{bound}(v^*) \leq \min_{v\in\mathcal{S}} score(v)$. Therefore, we can safely prune the remaining vertices and terminate. On the other hand, if such conditions are not satisfied, then the algorithm computes $score(v^*)$ by invoking bfs-search, and checks whether v^* should be added into the answer set $\mathcal{S}$ (lines 10-13). Finally, the algorithm outputs $\mathcal{S}$.

Algorithm 1 degree (G, k, t)

Input: $G = (V, E)$, an integer k, the component size threshold t.
Output: Top-k search result $\mathcal{S}$.

```
1: $\mathcal{S} \leftarrow \emptyset$;
2: for $v \in V$ do $score(v) \leftarrow -1$;
3: sort all vertices in descending order of their degree;
4: while $\exists v \in V$ s.t. $score(v) = -1$
5:     $v^* \leftarrow \arg\max_{v\in V,\ score(v)=-1} d(v)$;
6:     $\overline{bound}(v^*) \leftarrow \lfloor \frac{d(v^*)}{t} \rfloor$;
7:     if $|\mathcal{S}| = k$ and $\overline{bound}(v^*) \leq \min_{v\in\mathcal{S}} score(v)$ then
8:         break;
9:     $score(v^*) \leftarrow$ bfs-search ($G$, $t$, $v^*$);
10:    if $|\mathcal{S}| < k$ then $\mathcal{S} \leftarrow \mathcal{S} \cup \{v^*\}$;
11:    else if $score(v^*) > \min_{v\in\mathcal{S}} score(v)$ then
12:        $u \leftarrow \arg\min_{v\in\mathcal{S}} score(v)$;
13:        $\mathcal{S} \leftarrow (\mathcal{S} - \{u\}) \cup \{v^*\}$;
14: return $\mathcal{S}$;
```

The following example illustrates the working of Algorithm 1.

Example 1 Consider the graph in Figure 1 (a). Suppose that $k = 1$ and $t = 1$. The top-k running process is illustrated in Figure 4. The sorted vertex list is $c, a, b, f, h, i, d, e, g$ in descending order of their degree. The algorithm computes the structural diversity of these vertices in turn, and terminates before computing $score(g)$. This is because we have $\min_{v\in\mathcal{S}}\ score(v) = score(e) = 3$ and $\overline{bound}(g) = 3 \leq \min_{v\in\mathcal{S}}\ score(v)$. Therefore, Algorithm 1 can save one structural diversity computation.

v	c	a	b	f	h	i	d	e	g
bound	5	4	4	4	4	4	3	3	3
score	1	2	2	2	2	2	1	3	-
S	{c}	{a}						{e}	

Fig. 4 Illustration of the degree algorithm

Theorem 1 *For $1 \leq k \leq n$ and $1 \leq t \leq n$, Algorithm 1 performs top-k structural diversity search in $O(\sum_{v\in V}(d(v))^2)$ time and $O(m)$ space.*

Proof The algorithm first sorts all vertices in $O(n)$ time using the bin-sort algorithm [9]. It has to calculate the structural diversity for every vertex to answer a top-k query in the worst case. Consider a vertex v. When the algorithm computes $score(u)$ for each neighbor $u \in N(v)$, it has to scan the adjacency list of v in $O(d(v))$ time. Since there are $|N(v)| = d(v)$ neighbors, the total cost for scanning v's adjacency list is $O((d(v))^2)$. Thus, it takes $O(\sum_{v\in V}(d(v))^2)$ time to calculate the structural diversities for all vertices. In addition, one can maintain the top-k results in $O(n)$ time and $O(n)$ space using a variant of bin-sort list. Thus the time complexity of Algorithm 1 is $O(\sum_{v\in V}\ (d(v))^2)$.

In terms of the space consumption, the graph storage takes $O(n + m)$ space, and $\mathcal{S}$ takes $O(n)$ space. Thus, the space complexity of Algorithm 1 is $O(n + m) \subseteq O(m)$.

Remark 1 The worst-case time complexity of Algorithm 1 is bounded by $O(\sum_{v\in V} d(v) \cdot d_{max}) = O(md_{max}) \subseteq O(mn)$.

5 A Novel Top-K Search Framework for CC-TopK

The degree algorithm is not very efficient for top-k search because the degree-based upper bound in Lemma 1 is loose. To improve the efficiency, the key issue is to develop a tighter upper bound. To this end, in this section, we propose a novel framework with a tighter pruning bound and a new algorithm called bound-search to compute the structural diversity score. Before introducing the framework, we present two structural properties in a graph, which are very useful for developing the new bound.

5.1 Two Structural Properties

Property 1 For any vertex $v \in V$, if a vertex $u \in N(v)$ and u forms a size-1 component in $G_{N(v)}$, then v also forms a size-1 component in $G_{N(u)}$.

Proof We prove it by contradiction. Suppose that in $G_{N(u)}$, v is connected with another vertex w in a component. Then we can infer that $w \in N(u)$ and $w \in N(v)$. As u and w are connected and both are in $N(v)$, u and w form a size-2 component in $G_{N(v)}$, which contradicts to the fact that u forms a size-1 component in $G_{N(v)}$. This completes the proof.

As an example, in Figure 1 (b), vertex e forms a size-1 component in $G_{N(f)}$. Symmetrically, vertex f also forms a size-1 component in $G_{N(e)}$.

Property 2 If three vertices u, v, w form a triangle in G, then we have the sets $\{u, v\}$, $\{v, w\}$, and $\{u, w\}$ belong to the same component in $G_{N(w)}$, $G_{N(u)}$, and $G_{N(v)}$ respectively.

Proof This property can be easily derived by definition, thus we omit the proof for brevity.

For instance, in Figure 1 (a), vertices a, f, g form a triangle in G. We can observe that $\{a, g\}$ belong to a connected component in $G_{N(f)}$ in Figure 1 (b). Similarly, $\{a, f\}$ ($\{f, g\}$) belong to a connected component in $G_{N(g)}$ ($G_{N(a)}$).

Remark 2 Property 2 is based on the structure of a triangle (a clique of 3 nodes). We can extend the property to k-cliques for any $k \geq 3$. The following property can be similarly obtained: "In a clique $C = \{v_1, ..., v_k\}$ of k ($k \geq 3$) nodes in G, for each node $v_i \in C$, all other nodes in $C \backslash \{v_i\}$ belong to the same component in $G_{N(v_i)}$". Based on a k-clique with $k > 3$, we can obtain more structural information than using a triangle. However, computing k-cliques is more costly than computing triangles. In [7], it mentions that the time complexity to list all k-cliques for a constant k is $O(k\rho^{k-2}m)$, where ρ is the arboricity of the graph G. This indicates that enumerating 4-cliques is more costly than enumerating 3-cliques (triangles), while our algorithm fast-bound-search algorithm which will be introduced in Section 6.1, can achieve $O(\rho m)$ time which is the same as enumerating all 3-cliques. In addition, for a fixed $k \geq 3$, there may be $\Theta(m^{\frac{k}{2}})$ k-cliques in graph, since the graph may contain a clique of $\sqrt{m}$ vertices (thus containing $\binom{\sqrt{m}}{k} \in \Theta(m^{\frac{k}{2}})$ k-cliques. $\Theta(m^{\frac{k}{2}})$ is low bound for time complexity of any k-cliques listing algorithms (the theorem for $k = 3$ is shown in [18] of this paper). Hence, even the low bound of time complexity for enumerating all 4-cliques as $\Theta(m^2)$ is higher than the time complexity of our algorithms shown in Remark 3 and 4. Therefore, the extra cost taken by computing k-cliques is much larger than the cost saving obtained by using the new property based on k-cliques. Based on the above discussion, in this paper, we only make use of triangles other than larger cliques in order to guarantee the efficiency of our algorithms.

Based on these two properties, we can save a lot of computational costs in computing the structural diversity scores.

For example, if we find that vertex u forms a size-1 component in $G_{N(v)}$, then we know that v also forms a size-1 component in $G_{N(u)}$ by Property 1. Thus, when we compute $score(u)$, we do not need to perform a breadth-first search from v, because we already know v forms a size-1 component in $G_{N(u)}$. If we can efficiently record such *structural information* of v's neighbors when we compute $score(v)$, we can save a lot of computational costs. More importantly, such structural information can help us to get a tighter upper bound of the structural diversity. In the following subsection, we shall introduce a modified disjoint-set forest data structure to maintain such structural information efficiently.

5.2 Disjoint-Set Forest Data Structure

Intuitively, for the vertices in the same component, we can simply regard them as elements in the same set, while for the vertices in different components, we can represent them as elements in different sets. Thus we modify the classical disjoint-set forest data structure and the Union-Find algorithm [9] to maintain the structural information for each vertex efficiently. The modified structure consists of four operations: Make-Forest, Find-Set, Union, and Isolate. Compared to the classical disjoint-set forest data structure, the new structure includes an additional operation Isolate which is used to record the structural information described in Property 1, i.e., a vertex forms a size-1 component. Thus the modified structure is called Union-Find-Isolate. Algorithm 2 describes the four operations.

Make-Forest: For each vertex $v \in V$, we create a disjoint-set forest structure, denoted as $g[v]$, for its neighbors $N(v)$ using the Make-Forest (v) procedure in Algorithm 2. Specifically, for each $u \in N(v)$, we build a single-node tree $T[u]$ with three fields: parent, rank and count. The parent is initialized to be u itself, the rank is set to 0 and the count is set to 1, as there is only one vertex u in the tree. In addition, we also create a virtual node $T[0]$ which is used to collect all size-1 components in $G_{N(v)}$. The parent of $T[0]$ is set to 0 and the count is set to 0 because there is no size-1 component identified yet. For convenience, we refer to the operation of creating a single-node tree (line 4) or a virtual node (line 5) as a Make-Set operation.

Find-Set: The Find-Set (x) procedure is to find the root of $T[x]$ using the *path compression* strategy. The *path compression* strategy is a way of flattening the structure of the tree $T[x]$ whenever Find-Set (x) is used on it. Specifically, the idea is that each node visited on the path to a root node may as well be attached directly to the root node, because they are all in the same set and share the same representative. As a result, the obtained tree is much flatter, which can speed up future operations not only on these elements but on those referencing them, directly or indirectly.

Union: The Union(x, y) procedure applies the *union by rank* strategy to union two trees $T[fx]$ and $T[fy]$ which x and y belong to respectively. The *union by rank* strategy is to always attach the smaller tree to the root of the larger tree. For example, fx and fy are the roots of these two trees $T[fx]$ and $T[fy]$. If fx and fy have unequal rank, the one with a higher rank is set to be the parent of the other with a lower rank. Otherwise, we arbitrarily choose one of them as the parent and increase its rank by 1. For both cases, we update the $count$ of the root of the new tree.

Isolate: Procedure Isolate(x) unions a size-1 tree $T[x]$ into the virtual tree $T[0]$. It sets $T[x].parent$ to 0, and increases $T[0].count$ by 1. Isolate(x) essentially labels x as a size-1 component if we find x is not connected with any other node in a neighborhood induced subgraph.

We can apply the disjoint-set forest structure to maintain the connected components in $G_{N(v)}$. For any vertex $v \in V$, we create a rooted tree for every neighbor $u \in N(v)$ initially. If we find that u and w are connected in $G_{N(v)}$, we process it by $g[v]$.Union(u, w). If we identify that u forms a size-1 component in $G_{N(v)}$, we process it by $g[v]$.Isolate(u). Take $G_{N(f)}$ in Figure 1 (b) as an example again. First, we create $g[f]$ by Make-Forest (f) as shown in Figure 5 (a). Since vertices a and g are connected, we invoke $g[f]$.Union(a, g) and the resulted structure is shown in Figure 5 (b). The combined tree is rooted by g and has 2 vertices. Vertex e forms a size-1 component, thus we invoke $g[f]$.Isolate(e) and the result is shown in Figure 5 (c).

	parent	rank	count
T[0] = {	0 ,	0 ,	0 }
T[a] = {	a ,	0 ,	1 }
T[e] = {	e ,	0 ,	1 }
T[g] = {	g ,	0 ,	1 }
T[i] = {	i ,	0 ,	1 }

(a) Make-Forest(f)

	parent	rank	count
T[0] = {	0 ,	0 ,	0 }
T[a] = {	***g*** **,**	**0 ,**	**1 }**
T[e] = {	e ,	0 ,	1 }
T[g] = {	**g ,**	***1*** **,**	***2*** **}**
T[i] = {	i ,	0 ,	1 }

(b) g[f].Union(a,g)

	parent	rank	count
T[0] = {	**0 ,**	**0 ,**	***1*** **}**
T[a] = {	g ,	0 ,	1 }
T[e] = {	***0*** **,**	**0 ,**	**1 }**
T[g] = {	g ,	1 ,	2 }
T[i] = {	i ,	0 ,	1 }

(c) g[f].Isolate(e)

Fig. 5 Disjoint-Set Forest Data Structure g[f]

The time complexity of the Union-Find-Isolate algorithm is analyzed in the following lemma.

Lemma 2 *A sequence of M* Make-Set, Union, Find-Set *and* Isolate *operations, N of which are* Make-Set *operations, can be performed on a disjoint-set forest with "union by rank" and "path compression" strategies in worst-case time $O(M\alpha(N))$. $\alpha(N)$ is the inverse Ackermann function, which is incredibly slowly growing and at most 4 in any conceivable application. Thus, the time complexity of the* Union-Find-Isolate *algorithm can be regarded as $O(M)$.*

Proof The proof is similar to that in [9], thus is omitted.

In the following, for simplicity, we treat $\alpha(N)$ as a constant in the complexity analysis.

Algorithm 2 Union-Find-Isolate

```
1: procedure Make-Forest ($v$)
2: $g[v] = \{T[u] : u \in N(v)\} \cup \{T[0]\}$;
3: for $u \in N(v)$ do
4:     $T[u].(parent, rank, count) \leftarrow (u, 0, 1)$;
5: $T[0].(parent, rank, count) \leftarrow (0, 0, 0)$;

6: procedure Find-Set ($x$)
7: if $x \neq T[x].parent$ then
8:     $T[x].parent \leftarrow$ Find-Set ($T[x].parent$);
9: return $T[x].parent$;

10: procedure Union ($x, y$)
11: $fx \leftarrow$Find-Set ($x$); $fy \leftarrow$Find-Set ($y$);
12: if $fx \neq fy$ then
13:     if $T[fx].rank > T[fy].rank$ then
14:         $T[fy].parent \leftarrow fx$;
15:         $T[fx].count \leftarrow T[fx].count + T[fy].count$;
16:     else
17:         $T[fx].parent \leftarrow fy$;
18:         $T[fy].count \leftarrow T[fx].count + T[fy].count$;
19:         if $T[fx].rank = T[fy].rank$ then
20:             $T[fy].rank \leftarrow T[fy].rank + 1$;

21: procedure Isolate ($x$)
22: $T[x].parent \leftarrow 0$;
23: $T[0].count \leftarrow T[0].count + 1$;
```

5.3 A Tighter Upper Bound

With the disjoint-set forest data structure $g[v]$, we can keep track of the structural information of the connected components in $G_{N(v)}$ and derive a tighter upper bound of $score(v)$ than the degree-based bound in Lemma 1. Before introducing the upper bound, we give a definition of the *identified size-1 set* as follows.

Definition 4 (Identified Size-1 Set) In the disjoint-set forest structure $g[v]$, if $u \in N(v)$ and $T[u].parent = 0$, we denote $\mathbb{S}_u = \{u\}$ as an *identified size-1 set*, and $|\mathbb{S}_u| = 1$. If $u \in N(v)$, $T[u].parent = u$, we denote $\mathbb{S}_u = \{w \in N(v) : \text{Find-Set}(w) = u\}$ as an *unidentified set*, and $|\mathbb{S}_u| = T[u].count$.

By Definition 4, we know that each identified size-1 set is resulted from an Isolate operation, and the total number of the identified size-1 sets is $T[0].count$. According to Property 1, all these sets do not union with other sets. On the other hand, unidentified sets may further union with other sets or become an identified size-1 set. Consider the example in Figure 5 (c). $\mathbb{S}_e = \{e\}$ is an *identified size-1 set* and $T[0].count = 1$. Both $\mathbb{S}_g = \{a, g\}$ and $\mathbb{S}_i = \{i\}$ are *unidentified sets*.

Let $\mathbb{S} = \{\mathbb{S}_u : u \in N(v), T[u].parent = u$ or $T[u].parent = 0\}$ denote all disjoint sets in $g[v]$, excluding the virtual set $T[0]$. After traversing all the vertices and edges in $G_{N(v)}$, $\mathbb{S}$ contains all actual sets corresponding to the connected components in $G_{N(v)}$, and we have $score(v) = |\{\mathbb{S}_u : \mathbb{S}_u \in \mathbb{S}, |\mathbb{S}_u| \geq t\}|$. However, before traversing the neighborhood induced subgraph $G_{N(v)}$, $\mathbb{S}$ may not contain all the actual sets corresponding to the connected components, but includes some intermediate results. Even with such intermediate results maintained in $\mathbb{S}$, we can still use them to derive an upper bound. Specifically, we have the following lemma.

Lemma 3 *Let $\mathbb{S} = \{\mathbb{S}_1, \ldots, \mathbb{S}_l\}$ be the disjoint sets of $g[v]$, a be the number of identified size-1 sets, b be the number of sets whose sizes are larger than or equal to t, and c be the total size of these b sets. Then, we have an upper bound of $score(v)$ as follows. If $t = 1$, $\overline{bound}(v) = b$; if $t > 1$, $\overline{bound}(v) = b + \lfloor \frac{d(v)-c-a}{t} \rfloor$.*

Proof First, it is important to note that the current disjoint sets in $\mathbb{S}$ are not final, if we have not traversed all vertices and edges of $G_{N(v)}$. That is, some of them may be further combined by the Union operation and the number of sets may be reduced.

Next, we consider the following two cases.

If $t = 1$, we have $\overline{bound}(v) = b$, as the current number of sets whose sizes are greater than or equal to 1 is b and this number can only be reduced with the Union operation.

If $t > 1$, the current number of sets whose sizes are greater than or equal to t is b and this number can only be reduced with the Union operation. In addition, besides a identified size-1 sets and c vertices from the above b sets, there are still $d(v) - c - a$ vertices which may form sets whose sizes are greater than or equal to t. The maximum number of such potential sets is $\lfloor \frac{d(v)-c-a}{t} \rfloor$. Thus we have $\overline{bound}(v) = b + \lfloor \frac{d(v)-c-a}{t} \rfloor$.

For any vertex $v \in V$, at the initialization stage, each neighbor vertex $u \in N(v)$ forms a size-1 component. Thus $\overline{bound}(v) = 0 + \lfloor \frac{d(v)-0-0}{t} \rfloor = \lfloor \frac{d(v)}{t} \rfloor$, the same as the bound in Lemma 1. As the disjoint sets are gradually combined, $\overline{bound}(v)$ is refined towards $score(v)$ and becomes tighter. For example, in Figure 5 (c), suppose $t = 2$, we obtain $\mathbb{S} = \{\mathbb{S}_e, \mathbb{S}_g, \mathbb{S}_i\}$ and the three parameters in Lemma 3 are $a = 1$, $b = 1$ and $c = 2$. It follows that $\overline{bound}(f) = 1 + \lfloor \frac{4-2-1}{2} \rfloor = 1$, which is equal to $score(f) = 1$. This bound based on the disjoint-set forest is obviously tighter than the degree-based bound $\lfloor \frac{4}{2} \rfloor = 2$ derived in Lemma 1.

5.4 Top-K Search Framework

Based on the disjoint-set forest data structure and the tighter upper bound, we propose an advanced search framework in Algorithm 3 for top-k structural diversity search.

Advanced top-k framework: For each vertex $v \in V$, the algorithm initializes the disjoint-set forest data structure $g[v]$ by invoking Make-Forest (line 4). It also pushes each vertex v with the initial bound $\lfloor \frac{d(v)}{t} \rfloor$ into $\mathcal{H}$ which is a variant of

Algorithm 3 Top-k-search

Input: $G = (V, E)$, an integer k, the component size threshold t, gradient ratio $\theta \geq 1$.
Output: Top-k search result $\mathcal{S}$.

```
1: $\mathcal{H} \leftarrow \emptyset$; $\mathcal{S} \leftarrow \emptyset$;
2: for $v \in V$ do
3:     $score(v) \leftarrow -1$;
4:     Make-Forest $(v)$;
5:     $\mathcal{H}.push((v, \lfloor \frac{d(v)}{t} \rfloor))$;
6: while $\mathcal{H} \neq \emptyset$
7:     $(v^*, topbound) \leftarrow \mathcal{H}.pop()$;
8:     compute $\overline{bound}(v^*)$ according to Lemma 3;
9:     if $\theta \cdot \overline{bound}(v^*) < topbound$ then
10:        if $|\mathcal{S}| < k$ or $\overline{bound}(v^*) > \min_{v \in \mathcal{S}} score(v)$ then
11:            $\mathcal{H}.push((v^*, \overline{bound}(v^*)))$;
12:        continue;
13:    if $|\mathcal{S}| = k$ and $topbound \leq \min_{v \in \mathcal{S}} score(v)$ then
14:        break;
15:    $score(v^*) \leftarrow$ bound-search $(G, t, v^*)$;
16:    if $|\mathcal{S}| < k$ then $\mathcal{S} \leftarrow \mathcal{S} \cup \{v^*\}$;
17:    else if $score(v^*) > \min_{v \in \mathcal{S}} score(v)$ then
18:        $u \leftarrow \arg\min_{v \in \mathcal{S}} score(v)$;
19:        $\mathcal{S} \leftarrow (\mathcal{S} - \{u\}) \cup \{v^*\}$;
20: return $\mathcal{S}$;
```

bin-sort list. Then the algorithm iteratively finds the top-k results (lines 6-19). It first pops the vertex with the largest upper bound value from $\mathcal{H}$. Such a vertex and its bound are denoted as v^* and $topbound$ respectively (line 7). The algorithm re-evaluates $\overline{bound}(v^*)$ from $g[v^*]$ based on Lemma 3, as the component information in $g[v^*]$ may have been updated. And then, it compares the refined bound $\overline{bound}(v^*)$ with the old bound $topbound$.

In order to avoid frequently calculating the upper bounds and updating $\mathcal{H}$, we introduce a new parameter $\theta \geq 1$, and compare $\theta \cdot \overline{bound}(v^*)$ with $topbound$.

If $\theta \cdot \overline{bound}(v^*) < topbound$, it suggests that $\overline{bound}(v^*)$ is substantially smaller than $topbound$. That is, the old bound $topbound$ is too loose. Under this condition, if $|\mathcal{S}| < k$ or $\overline{bound}(v^*) > \min_{v \in \mathcal{S}} score(v)$, the algorithm pushes v^* back to $\mathcal{H}$ with the refined bound $\overline{bound}(v^*)$ (lines 10-11). Otherwise, the algorithm can safely prune v^*. In both cases, the algorithm continues to pop the next vertex from $\mathcal{H}$ (lines 9-12).

If $\theta \cdot \overline{bound}(v^*) \geq topbound$, it means that $\overline{bound}(v^*)$ is not substantially smaller than $topbound$. In other words, the old bound is a relatively tight estimation. Then the algorithm moves to lines 13-14 to check the termination condition. If $|\mathcal{S}| = k$ and $topbound \leq \min_{v \in \mathcal{S}} score(v)$, the algorithm can safely prune all the remaining vertices in $\mathcal{H}$ and terminate, because the upper bound of those vertices is smaller than $topbound$.

If the early termination condition is not satisfied, the algorithm invokes the bound-search algorithm (line 15) to compute $score(v^*)$. bound-search is shown in Algorithm 4 and will be described later. After computing $score(v^*)$, the algorithm uses the same process to update the set $\mathcal{S}$ by v^* as the degree algorithm does (lines 16-19).

Bound-Search: Algorithm 4 shows the bound-search procedure to compute $score(v)$. Based on the disjoint-set forest $g[v]$, we know that any vertex $u \in N(v)$ with $T[u].parent = 0$ corresponds to an identified size-1 component resulted from an Isolate operation. So bound-search does not need to search them again. It only adds the vertices whose $parent \neq 0$ into an unvisited vertex hashtable R (lines 1-2). This is an improvement from bfs-search, as bound-search avoids scanning the identified size-1 components. For each vertex $u \in R$, the algorithm invokes the procedure bound-bfs (lines 5-18) to search u's neighborhood in a breadth-first search manner. For u's neighbor vertex w, if $w \in R$, i.e., $w \in N(v)$, the algorithm unions u and w into one set in $g[v]$. According to Property 2, we also union v and w into one set in $g[u]$, and union v and u into one set in $g[w]$ (lines 11-15). If u does not union with any other vertex, the algorithm invokes an Isolate operation on u to mark that u forms a size-1 component in $g[v]$ (lines 16-18). Symmetrically, by Property 1, the algorithm invokes an Isolate operation on v to mark that v forms a size-1 component in $g[u]$ too. After the BFS search, the algorithm can compute $score(v)$ using the procedure count-components (lines 19-25) to count the number of sets in $g[v]$ whose sizes are at least t. The following example illustrates how the Top-k-search framework (Algorithm 3) works.

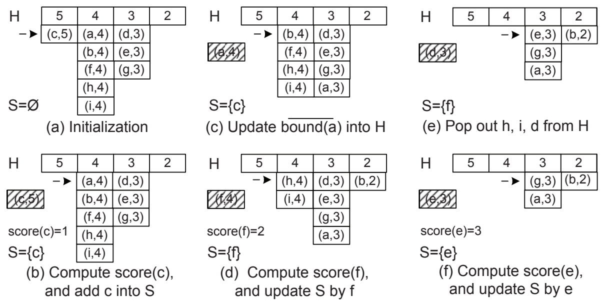

(a) Initialization
(c) Update $\overline{\text{bound}}$(a) into H
(e) Pop out h, i, d from H
(b) Compute score(c), and add c into S
(d) Compute score(f), and update S by f
(f) Compute score(e), and update S by e

Fig. 6 Illustration of Top-k-search with bound-search running on the graph in Figure 1 (a). $k = 1$, $t = 1$, and $\theta = 1$.

Example 2 Consider the graph shown in Figure 1 (a). Suppose that $t = 1$ and $k = 1$. We apply the Top-k-search algorithm with $\theta = 1$ and the running steps are depicted in Figure 6. For initialization, we push each vertex v with the upper bound $\lfloor \frac{d(v)}{t} \rfloor$ into $\mathcal{H}$, as shown in Figure 6 (a).

In the first iteration, as shown in Figure 6 (b), we pop vertex c from $\mathcal{H}$ with $topbound = 5$. We calculate $\overline{bound}(c) = 5$ according to Lemma 3. Then, we compute $score(c)$ by bound-search. In $G_{N(c)}$, there is a single path connecting all vertices a, b, d, h, i in $N(c)$, so $score(c) = 1$. When the

Algorithm 4 bound-search (G, t, v)

Input: $G = (V, E)$, the component size threshold t, vertex v.
Output: $score(v)$.

```
1: $R \leftarrow \emptyset$;
2: for $u \in N(v)$ and $T[u].parent \neq 0$ do $R \leftarrow R \cup \{u\}$;
3: for $u \in R$ do bound-bfs $(u)$;
4: return count-components $(g[v], t)$;
5: procedure bound-bfs $(u)$
6: $Q \leftarrow \emptyset$; $UnionFlag \leftarrow$ false;
7: $Q.EnQueue(u)$; $R \leftarrow R - \{u\}$;
8: while $Q \neq \emptyset$
9:     $u \leftarrow Q.DeQueue()$;
10:    for $w \in N(u)$ do
11:        if $w \in R$ then
12:            $Q.EnQueue(w)$; $R \leftarrow R - \{w\}$;
13:            $g[v]$.Union $(u, w)$; $UnionFlag \leftarrow$ true;
14:            if $score(u) = -1$ then $g[u]$.Union $(v, w)$;
15:            if $score(w) = -1$ then $g[w]$.Union $(v, u)$;
16: if $UnionFlag =$ false then
17:     $g[v]$.Isolate $(u)$;
18:     if $score(u) = -1$ then $g[u]$.Isolate $(v)$;
19: procedure count-components $(g[v], t)$
20: $score \leftarrow 0$;
21: for $u \in N(v)$ do
22:     if $T[u].parent = u$ and $T[u].count \geq t$ then
23:         $score \leftarrow score + 1$;
24: if $t = 1$ then $score \leftarrow score + T[0].count$;
25: return $score$;
```

algorithm traverses the edge (a, b), we perform two operations $g[a]$.Union (c, b) and $g[b]$.Union (c, a) in $g[a]$ and $g[b]$ respectively according to Property 2. Then we push vertex c into $\mathcal{S}$.

In the next iteration, as shown in Figure 6 (c), we pop vertex a from $\mathcal{H}$ which has $topbound = 4$. Then we update $\overline{bound}(a) = 3$ as we know that vertices b and c are in the same set in $g[a]$. Since $\theta \cdot \overline{bound}(a) < topbound$ and $\overline{bound}(a) > \min_{v \in \mathcal{S}} score(v)$, we push $(a, 3)$ into $\mathcal{H}$ again.

When the algorithm goes to process vertex f in Figure 6 (d), we have $\theta \cdot \overline{bound}(f) = topbound = 4$ and $topbound > \min_{v \in \mathcal{S}} score(v)$. And then we compute $score(f) = 2$ and replace vertex c in $\mathcal{S}$ with f.

After that, we pop vertices h, i, d from $\mathcal{H}$ in turn, as shown in Figure 6 (e). One can easily check that none of them satisfies the condition in line 10 of Algorithm 3. Thus, we do not push h, i, d back into $\mathcal{H}$ again.

Next we pop vertex e, compute $score(e) = 3$ and update $\mathcal{S}$ by e, as shown in Figure 6 (f). Since $topbound$ in $\mathcal{H}$ is no greater than $score(e) = 3$, we can safely terminate. In this process, we only invoke bound-search three times to calculate the structural diversity score, while the previous degree algorithm calculates the structural diversity score of eight vertices, which is clearly more expensive.

5.5 Complexity Analysis

Lemma 4 *The upper bound $\overline{bound}(v)$ defined in Lemma 3 for any vertex $v \in V$ can be updated in $O(1)$ time in Algorithm 3.*

Proof We need to maintain a, b and c in $g[v]$ to recompute $\overline{bound}(v)$. Obviously, $a = T[0].count$, and b, c can be easily maintained in the Union operation of $g[v]$ without increasing the time complexity. Thus $\overline{bound}(v)$ can be updated in $O(1)$ time.

Lemma 5 *The total time to compute $\overline{bound}$ for all vertices in Algorithm 3 is $O(\frac{m}{t})$.*

Proof According to Lemma 4, $\overline{bound}(v)$ for a vertex v can be computed in $O(1)$ time. The initial upper bound of v is $\lfloor \frac{d(v)}{t} \rfloor$, and $\overline{bound}(v)$ is updated in non-increasing order. In line 9 of Algorithm 3, we compare $\theta \cdot \overline{bound}(v)$ and $topbound$ to check whether v should be pushed into $\mathcal{H}$. Since $topbound \leq \lfloor \frac{d(v)}{t} \rfloor$, $\overline{bound}(v)$ can be updated for at most $\lfloor \frac{d(v)}{t} \rfloor$ times. Thus the total time cost is $O(\sum_{v \in V} \frac{d(v)}{t}) = O(\frac{m}{t})$.

Lemma 6 *In* Top-k-search, *$\mathcal{H}$ can be maintained in $O(\frac{m}{t} + n)$ time using $O(n)$ space.*

Proof $\mathcal{H}$ can be implemented by a variant of bin-sort list which supports a push operation in constant time and l pop operations in $O(l + n)$ time (illustrated in Figure 6). Each time, estimating the upper bound $\overline{bound}$ in line 8 causes at most one push operation (line 11) in $\mathcal{H}$. By Lemma 5, we know that for each vertex $v \in V$, there are at most $\lfloor \frac{d(v)}{t} \rfloor$ bound refinements. Thus, there are at most $\sum_{v \in V} \lfloor \frac{d(v)}{t} \rfloor$ bound refinements in total for all vertices. In addition, there are n initial push operations. Therefore, the algorithm uses $O(\sum_{v \in V} \frac{d(v)}{t} + n) = O(\frac{m}{t} + n)$ time for all the push operations. The number of pop operations is no larger than the number of push operations. Put it all together, the time complexity to maintain $\mathcal{H}$ is $O(\frac{m}{t} + n)$. The space complexity of $\mathcal{H}$ is $O(n)$.

Theorem 2 *Algorithm 3 takes $O\left(\sum_{v \in V} (d(v))^2\right)$ time and $O(m)$ space.*

Proof Since the time to access the adjacency lists in bound-search is $O(\sum_{v \in V} (d(v))^2)$, and all Union operations are in the loop of accessing adjacency lists (lines 13-15 of Algorithm 4), the number of Union operations is $O(\sum_{v \in V} (d(v))^2)$. The algorithm invokes n Make-Forest operations (line 4 of Algorithm 3), which includes $\sum_{v \in V} (d(v) + 1) = 2m + n$ Make-Set operations. Next, all Isolate operations are in the procedure bound-bfs (lines 17-18 of Algorithm 4). The number is no greater than $\sum_{v \in V} 2d(v) = 4m$. No Find-Set operation is directly invoked. Thus, Union-Find-Isolate includes $O(\sum_{v \in V} (d(v))^2)$ Make-Set, Union, Find-Set, Isolate

Algorithm 5 fast-bound-search (G, t, v)

Input: $G = (V, E)$, the component size threshold t, vertex v.
Output: $score(v)$.

```
1:  $R \leftarrow \emptyset$;
2:  **for** $u \in N(v)$ and $T[u].parent \neq 0$ **do** $R \leftarrow R \cup \{u\}$;
3:  **for** $u \in R$ **do** fast-bound-bfs $(u)$;
4:  **return** count-components $(g[v], t)$;
5:  **procedure** fast-bound-bfs $(u)$
6:  $Q \leftarrow \emptyset$; $UnionFlag \leftarrow$ false;
7:  $Q.EnQueue(u)$; $R \leftarrow R - \{u\}$;
8:  **while** $Q \neq \emptyset$
9:      $u \leftarrow Q.DeQueue()$;
10:     **if** $d(u) > d(v)$ **then** $MinAdjL \leftarrow N(v)$;
11:     **else** $MinAdjL \leftarrow N(u)$;
12:     **for** $w \in MinAdjL$ **do**
13:         **if** $(w, u) \in E$ and $w \in R$ **then**
14:             $Q.EnQueue(w)$; $R \leftarrow R - \{w\}$;
15:             $g[v]$.Union $(u, w)$; $UnionFlag \leftarrow$ true;
16:             **if** $score(u) = -1$ **then** $g[u]$.Union $(v, w)$;
17:             **if** $score(w) = -1$ **then** $g[w]$.Union $(v, u)$;
18: **if** $UnionFlag =$ false **then**
19:     $g[v]$.Isolate $(u)$;
20:     **if** $score(u) = -1$ **then** $g[u]$.Isolate $(v)$;
```

operations, $2m+n$ of which are Make-Set. By Lemma 2, the time complexity of Union-Find-Isolate is $O(\sum_{v\in V}(d(v))^2)$.

By Lemma 6, $\mathcal{H}$ takes $O(\frac{m}{t}+n)$ time. $\mathcal{S}$ maintains the top-k results using $O(n)$ time. By Lemma 5, updating the upper bounds for all vertices takes $O(\frac{m}{t})$ time. Therefore, the time complexity of Algorithm 3 is $O\ (\sum_{v\in V}\ (d(v))^2)$.

Next, we analyze the space complexity. For $v \in V$, $g[v]$ contains $d(v)+1$ initial disjoint singleton trees, in which each node takes constant space. Hence, the disjoint-set forest structure takes $O(m)$ space for all vertices. $\mathcal{S}$ and $\mathcal{H}$ both consume $O(n)$ space. In summary, the space complexity of Algorithm 3 is $O(m)$.

Hence, Theorem 2 is established.

6 Fast Computation of Component-based Structural Diversity Score

In this section, on top of the Top-k-search framework, we propose two methods for fast computing the structural diversity score for a vertex. The first method is fast-bound-search which improves bound-search and achieves a better time complexity using the same space. The second is an A* search method which uses a new search order and a new termination condition.

6.1 Fast Bound-Search

We present fast-bound-search in Algorithm 5, which is built on bound-search. The major difference is in procedure fast-bound-bfs for traversing a connected component. When accessing the adjacency list of vertex u having $d(u) > d(v)$, we will access the adjacency list of v instead (lines 10-13), i.e., we always select the vertex with a smaller degree to access. Checking whether $(w, u) \in E$ in line 13 can be done efficiently by keeping all edges in a hashtable. Moreover, R can also be implemented by a hashtable. Thus line 13 can be done in expected constant time by hashing.

To show the effectiveness of this improvement, we consider an example $G_{N(r)}$ in Figure 7. Suppose that r has two neighbors p and q with degree 1 and 100 respectively. To compute $score(r)$, bound-search needs to access the adjacency lists of p and q, and check $|N(p)|+|N(q)| = 101$ vertices. In contrast, fast-bound-search accesses $N(r)$ instead of $N(q)$ because $d(q) > d(r)$, thus the number of visited vertices is reduced to $|N(p)| + |N(r)| = 3$.

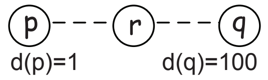


Fig. 7 $G_{N(r)}$ has two vertices p and q with degree 1 and 100

Complexity Analysis: Using fast-bound-search to compute structural diversity scores, we achieve a better time complexity of the Top-k-search framework shown in the following theorem.

Theorem 3 *The* Top-k-search *framework using* fast-bound-search *takes* $O(\sum_{(u,v)\in E}\min\{d(u), d(v)\})$ *time and* $O(m)$ *space.*

Proof For a vertex v, the time cost of accessing the adjacency lists is $\sum_{u\in N(v)}\min\{d(u), d(v)\}$ for computing its score. To compute scores for all vertices, accessing the adjacency lists consumes $O(\ \sum_{v\in V}\sum_{u\in N(v)}\ \min\{d(u), d(v)\}) = O(\ \sum_{(u,v)\in E}\min\{d(u), d(v)\})$.

Since the number of Union operations is bounded by the number of accessing adjacency lists, the number of Union operations is $O(\sum_{(u,v)\in E}\min\{d(u), d(v)\})$. Moreover, there are $2m+n$ Make-Set operations, $O(m)$ Isolate operations and no direct Find-Set operation invoked by the algorithm. By Lemma 2, Union-Find-Isolate takes $O(\ \sum_{(u,v)\in E}\min\{d(u), d(v)\})$ time in total. The other steps in the loop of accessing adjacency list take constant time. Therefore, it takes $O(\sum_{(u,v)\in E}\ \min\{d(u),\ d(v)\})$ time to calculate all vertices' structural diversity scores using the fast-bound-search algorithm.

By Lemma 5, the total time of estimating upper bound is $O(\frac{m}{t}) \subseteq O(m)$, and by Lemma 6, the total time to maintain $\mathcal{H}$ is $O(\frac{m}{t}+n) \subseteq O(m)$.

The remaining steps in Algorithm 3 using fast-bound-search totally cost $O(m)$ time.

Compared with bound-search, fast-bound-bfs needs extra $O(m)$ space for storing the edge hashtable. Thus, the space consumption is still $O(m)$.

Hence, Theorem 3 is established.

Remark 3 According to [7], we have

$$O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m),$$

where ρ is the arboricity of a graph G and $\rho \leq \min\{\lceil\sqrt{m}\,\rceil, d_{max}\}$ for any graph G. Thus the worst-case time complexity of the Top-k-search framework using fast-bound-search is bounded by

$$O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m) \subseteq O(m^{1.5}).$$

6.2 A*-Based Bound-Search

In this subsection we design a new search order and a new termination condition to compute the structural diversity score for a vertex.

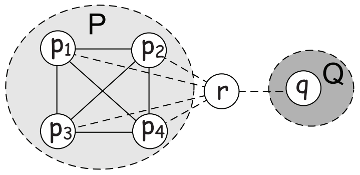


Fig. 8 $G_{N(r)}$ containing two components P and Q

Take Figure 8 as an example which shows the neighborhood induced subgraph of r. Suppose that before examining $G_{N(r)}$, the algorithm has computed the structural diversity scores for r's neighbors $p_1, \ldots, p_4$. Then, by Property 2, the vertices $p_1, \ldots, p_4$ are combined into one component P in $G_{N(r)}$. There is another component Q in $G_{N(r)}$ with only one vertex q. To compute $score(r)$, the algorithm needs to further check whether the components P and Q are connected or not. If the algorithm first checks vertex q in the component Q, then it will go through q's adjacency list $N(q)$ to verify whether q connects with any vertices in $p_1, \ldots, p_4$. If q is not connected with any one of them, we can conclude that Q forms a size-1 component and P forms a size-4 component in $G_{N(r)}$. Thus, the algorithm does not need to traverse the adjacency lists of $p_1, \ldots, p_4$, and it can terminate early. In contrast, if the algorithm first checks the component P, then it needs to go through the adjacency lists of vertices $p_1, \ldots, p_4$ to verify whether they connect with q or not. This is clearly more expensive than starting from the component Q. Motivated by this observation, we propose an A* search strategy to efficiently compute the structural diversity in the neighborhood induced subgraph of a vertex. Below, we first give the definition of *component cost* which is used as a cost function to determine the component visiting order in the A* search process.

Definition 5 (Component Cost) Given a component S in a neighborhood induced subgraph, the component cost of S is the sum of degree of the unvisited vertices in S, denoted as $cost(S) = \sum_{unvisited\ v\in S} d(v)$.

Suppose that in Figure 8 all vertices in $N(r)$ are unvisited. The component costs are $cost(P) = 16$ and $cost(Q) = 1$. The component cost measures the cost of accessing the adjacency lists of a component. If we check the low-cost components first and the high-cost components later, we can potentially save more computation. Thus in A* search, we always pick a component $T[x]$ in $G_{N(v)}$ which has the least cost to traverse.

To record the cost, we add the component cost as a field in the Union-Find-Isolate data structure. Specifically, for a vertex u, when we create a single-node component $T[u]$, we initialize $T[u].cost = d(u)$. When we union two components $T[u]$ and $T[v]$, we add up their costs, i.e., $T[u].cost + T[v].cost$.

The Algorithm: A*-bound-search uses the component cost for determining a fast search order to traverse the components in $G_{N(v)}$ until there is only one unvisited component left. In traversing a component, the algorithm accesses the adjacency lists of the unvisited vertices in increasing order of their degrees until the component is connected with other components or traversed.

Algorithm 6 shows A*-bound-search. For a vertex v, the algorithm uses a minimum heap $\mathcal{TC}$ to maintain all the unidentified components in $G_{N(v)}$ ordered by their component costs. For a component rooted by a vertex u, the algorithm makes use of a minimum heap $\mathcal{C}[u]$ to maintain all vertices in this component ordered by their degree. Initially, for each vertex u whose $parent$ is not 0, the algorithm pushes u with cost $d(u)$ into the minimum heap $\mathcal{C}[T[u].parent]$, and adds u into the hashtable R which stores all the unvisited vertices (line 5). Moreover, if u is the root of $T[u]$, the algorithm pushes the component of u and its component cost $T[u].cost$ into the heap $\mathcal{TC}$ (lines 6-7).

Let us consider an example. Figure 9 (a) shows the neighborhood induced subgraph $G_{N(r)}$, and Figure 9 (b) shows the degree and the parent in $T[.]$ for each vertex in $N(r)$. We know that p_1, p_2, p_3 are in a component rooted by p_1, and q_1, q_2 are in a component rooted by q_1, and s is in a component rooted by s itself. After initialization, the minimum heaps $\mathcal{TC}$, $\mathcal{C}[s]$, $\mathcal{C}[p_1]$, $\mathcal{C}[q_1]$ and the hashtable R are illustrated in Figure 9 (c).

The algorithm iteratively pops a component with the minimum cost from $\mathcal{TC}$, denoted as x with cost $tcost_x$ (line 9). If the component is rooted by vertex x and $tcost_x = T[x]$

Algorithm 6 A*-bound-search (G, t, v)

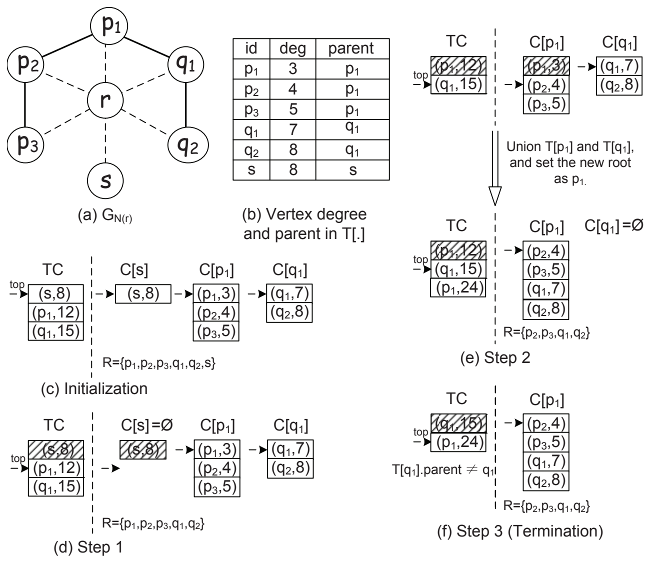
```
Input: $G = (V, E)$, the component size threshold $t$, vertex $v$.
Output: $score(v)$.

1:  $R \leftarrow \emptyset$; $\mathcal{TC} \leftarrow \emptyset$;
2:  for $u \in N(v)$ do $\mathcal{C}[u] \leftarrow \emptyset$;
3:  for $u \in N(v)$ do
4:      if Find-Set $(u) \neq 0$ then
5:          $\mathcal{C}[T[u].parent].push((u, d(u)))$; $R \leftarrow R \cup \{u\}$;
6:          if $T[u].parent = u$ then
7:              $\mathcal{TC}.push((u, T[u].cost))$;
8:  while $\mathcal{TC} \neq \emptyset$
9:      $(x, tcost_x) \leftarrow \mathcal{TC}.pop()$; $UnionFlag \leftarrow$ false;
10:     if $x \neq$Find-Set $(x)$ then continue;
11:     if $tcost_x \neq T[x].cost$ then
12:         $\mathcal{TC}.push((x, T[x].cost))$; continue;
13:     if $|R| = |C[x]|$ then goto Step 33;
14:     while $\mathcal{C}[x] \neq \emptyset$ and $UnionFlag =$ false
15:         $(u, cost_u) \leftarrow \mathcal{C}[x].pop()$; $R \leftarrow R - \{u\}$;
16:         $T[x].cost \leftarrow T[x].cost - cost_u$;
17:         w.l.o.g, we assume $d(u) < d(v)$;
18:         for $w \in N(u)$ do
19:             if $(w, v) \in E$ and $w \in R$ then
20:                 $fu \leftarrow$Find-Set $(u)$; $fw \leftarrow$Find-Set $(w)$;
21:                 if $fu \neq fw$ then
22:                     $\mathcal{Q} \leftarrow Heap\_Merge(\mathcal{C}[fu], \mathcal{C}[fw])$;
23:                     $g[v]$.Union $(u, w)$;
24:                     $\mathcal{C}$[Find-Set $(x)$] $\leftarrow \mathcal{Q}$; $UnionFlag \leftarrow$true;
25:             if $score(u) = -1$ then $g[u]$.Union $(v, w)$;
26:             if $score(w) = -1$ then $g[w]$.Union $(v, u)$;
27:     if $UnionFlag =$ true and Find-Set $(x) = x$ then
28:         $\mathcal{TC}.push((x, T[x].cost))$;
29:     if $UnionFlag =$ false and $\mathcal{C}[x] = \emptyset$ then
30:         if $T[x].count = 1$ then
31:             $g[v]$.Isolate $(x)$;
32:             if $score(x) = -1$ then $g[x]$.Isolate $(v)$;
33: return count-components $(g[v], t)$;
```

$.cost$, the algorithm will examine the vertices in the component of x. Otherwise, if the component is no longer rooted by x or $tcost_x \neq T[x].cost$, it means that the component of x has been combined with another component in a previous iteration. Then, the algorithm pops the next component from $\mathcal{TC}$. If $|R| = |\mathcal{C}(x)|$ holds, then all the unvisited nodes in R are from the same component rooted by x, and this component is the last to be traversed in $G_{N(v)}$ (line 13). By the early termination condition, the algorithm does not need to traverse this component and can directly go to count the number of components in $G_{N(v)}$ (line 33).

For a popped component rooted by x, the algorithm iteratively examines the vertices in the component in increasing order of their degree (lines 14-26). For such a vertex u, we will access its adjacency list $N(u)$ to find out those vertices denoted as w that are also in $N(v)$. Then we will union the components which contain u and w respectively into one. This process is very similar to the previous algorithms. So we omit the details for brevity.

Continuing with our example in Figure 9. After initialization, we pop the first component $(s, 8)$ from $\mathcal{TC}$, as shown



id	deg	parent
p1	3	p1
p2	4	p1
p3	5	p1
q1	7	q1
q2	8	q1
s	8	s

Fig. 9 A*-bound-search example for computing $score(r)$

in step 1 (Figure 9 (d)). Then, we examine vertex s in this component and find that it is not connected with other components in $G_{N(r)}$. Next, we move to step 2 (Figure 9 (e)) to pop the component $(p_1, 12)$. In this component, we first examine the adjacency list of p_1, i.e., $N(p_1)$. We find that p_1 is connected with q_1, so we union the components rooted by p_1 and q_1. Assume that the new component is rooted by p_1. Then we set $T[q_1].parent = p_1$ and merge $\mathcal{C}(q_1)$ into $\mathcal{C}(p_1)$. We push the new component $(p_1, 24)$ into $\mathcal{TC}$ again. In step 3 (Figure 9 (f)), we pop the component $(q_1, 15)$ and find that $T[q_1].parent \neq q_1$, as the component of q_1 has been combined with that of p_1 in step 2. In this step, there is only one component in $\mathcal{TC}$, which meets the early termination condition.

Complexity Analysis: In the component union process (line 22 of Algorithm 6), we need to merge two heaps $\mathcal{C}[fw]$ and $\mathcal{C}[fu]$ into one. We can implement $\mathcal{C}[.]$ by the mergeable heap such as leftist heap or binomial heap [9], which support the merge of two heaps in $O(\log n)$ time and a push/pop operation in $O(\log n)$ time for a heap with n elements.

Lemma 7 *In Algorithm 6, the operations for $\mathcal{TC}$ and all $\mathcal{C}[.]$ take $O(d(v) \log d(v))$ time and $O(d(v))$ space in total.*

Proof Since the number of components in $G_{N(v)}$ is no greater than $d(v)$, we perform at most $d(v) - 1$ Union operations before termination. Hence, there are at most $d(v) - 1$ new components to be pushed into $\mathcal{TC}$ (lines 12 and 28). In addition, for initialization $|\mathcal{TC}| \leq d(v)$ holds, which indicates that $|\mathcal{TC}| \leq 2d(v)$ always holds. As there are at most $2d(v)$ push and pop operations respectively, and each operation takes $O(\log d(v))$ time, overall $\mathcal{TC}$ takes $O(d(v) \log d(v))$ time using $O(d(v))$ space.

For initialization, all $\mathcal{C}[.]$ heaps take $d(v)$ push operations in total (line 5), and the time cost of each operation is $O(\log d(v))$ as the size of the largest heap is smaller than $d(v)$. Hence, the initialization time is $O(d(v) \log d(v))$. As analyzed above, there are at most $d(v)-1$ heap merging operations and each operation costs $O(\log d(v))$, the total time cost in line 22 is $O(d(v) \log d(v))$. Moreover, there are at most $d(v)$ pop operations in line 15, the time cost of which is $O(d(v) \log d(v))$. All $\mathcal{C}[.]$ heaps contain at most $d(v)$ vertices totally costing $O(d(v))$ space. As a result, all $\mathcal{C}[.]$ heaps take $O(d(v) \log d(v))$ time and $O(d(v))$ space.

Theorem 4 *The* Top-k-search *framework using* A*-bound-search *takes* $O(\sum_{(u,v)\in E}(\min\{d(u), d(v)\} + (\log d(u) + \log d(v))))$ *time and* $O(m)$ *space.*

Proof The proof is similar to the proof of Theorem 3. A difference is that we use the Find-Set operations in A*-bound-search. Since the Find-Set operations in lines 20 and 24 are in the loop of accessing adjacency list, the total number of such operations is $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ for the whole process. Consider the process of computing $score(v)$ for a vertex v, we perform $d(v)$ Find-Set operations in line 4. Since lines 10 and 27 are both in the outer while loop (line 8), and $\mathcal{TC}$ has at most $2d(v)$ pop operations according to Lemma 7, the algorithm performs at most $2d(v)$ Find-Set operations in lines 10 and 27 respectively. Hence, it performs

$$O(\sum_{(u,v)\in E} \min\{d(u), d(v)\} + \sum_{v\in V} 5d(v))$$
$$= O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$$

Find-Set operations. By Lemma 2, Union-Find-Isolate takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time in total.

Another difference is that we maintain two types of heaps $\mathcal{TC}$ and $\mathcal{C}[.]$. By Lemma 7, the total time of $\mathcal{TC}$ and $\mathcal{C}[.]$ are

$$O(\sum_{u\in V} d(u) \log d(u))$$
$$= O(\sum_{u\in V} \sum_{v\in N(u)} \log d(u))$$
$$= O(\frac{1}{2} \sum_{(u,v)\in E} (\log d(u) + \log d(v)))$$
$$= O(\sum_{(u,v)\in E} (\log d(u) + \log d(v))).$$

The additional space overhead is $O(m)$.

Hence, Theorem 4 is established.

Remark 4 The worst-case time complexity of Top-k-search framework using A*-bound-search is bounded by

$$O(\sum_{(u,v)\in E} (\min\{d(u), d(v)\} + (\log d(u) + \log d(v))))$$
$$\subseteq O((\rho + \log d_{max})m)$$
$$\subseteq O(m^{1.5}),$$

where ρ is the arboricity of the graph as mentioned in Remark 3.

6.3 Complexity Comparison

We compare the time complexity of algorithms degree and Top-k-search, to understand why the proposed Top-k-search framework is more efficient.

According to Theorem 1, degree takes $O(\sum_{v\in V}(d(v))^2)$ time, which can be equivalently rewritten as

$$O(\sum_{v\in V}(d(v))^2)$$
$$= O(\sum_{v\in V} \sum_{u\in N(v)} d(u))$$
$$= O(\sum_{(u,v)\in E} (d(u) + d(v)))$$
$$= O(\sum_{(u,v)\in E} (\max\{d(u), d(v)\} + \min\{d(u), d(v)\}))$$
$$= O(\sum_{(u,v)\in E} \max\{d(u), d(v)\})$$

For Top-k-search using fast-bound-search, according to Theorem 3, it takes $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ time, obviously better than $O(\sum_{(u,v)\in E} \max\{d(u), d(v)\})$, the time complexity of degree. The two algorithms have the same complexity **only if** all vertices in the graph have the same degree. In a power-law graph such as a social network, the degrees of vertices have a large variance, thus Top-k-search using fast-bound-search is much better than degree in such networks. For example, on a star graph with n nodes, Top-k-search using fast-bound-search takes $O(n)$ time while degree takes $O(n^2)$ time.

For Top-k-search using A*-bound-search, its time complexity, which is $O(\sum_{(u,v)\in E} (\min\{d(u), d(v)\} + (\log d(u) + \log d(v))))$, is also better than $O(\sum_{(u,v)\in E} \max\{d(u), d(v)\})$ of degree. This is because the first part $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\})$ is better for the same reason as stated above, and the second part $O(\sum_{(u,v)\in E} (\log d(u)+\log d(v)))$ is also better since $O(\log d(u) + \log d(v)) \leq O(\max\{d(u), d(v)\})$ holds.

7 Handling Update for CC-TopK in Dynamic Networks

Many real-world networks undergo frequent updates. When the network is updated, the top-k structural diversity results

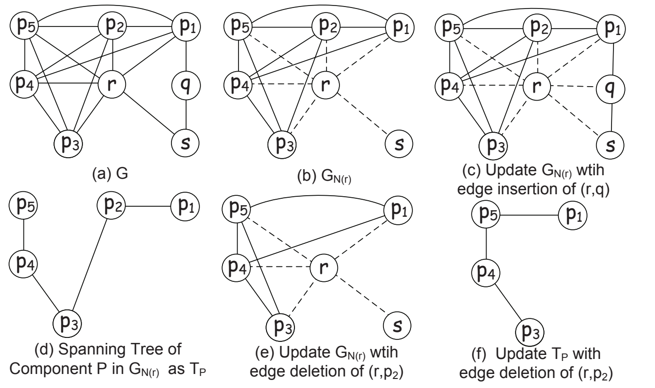

(a) G
(b) G_N(r)
(c) Update G_N(r) wtih edge insertion of (r,q)
(d) Spanning Tree of Component P in G_N(r) as T_P
(e) Update G_N(r) wtih edge deletion of (r,p2)
(f) Update T_P with edge deletion of (r,p2)
p1
p2
p3
p4
p5
r
q
s

Fig. 10 Illustration of updates in a dynamic graph

also need to be updated. The challenge, however, is that inserting or deleting a single edge (u, v) can trigger updates in a series of neighborhood induced subgraphs including $G_{N(u)}$, $G_{N(v)}$ and $G_{N(w)}$ where $w \in N(u) \cap N(v)$. This can be a costly operation because the corresponding structural diversity scores need to be recomputed, and the top-k results need to be updated too.

In the following, we will show that our Top-k-search framework can be easily extended to handle updates in dynamic graphs. We consider two types of updates: edge insertion and edge deletion. Vertex insertion/deletion can be regarded as a sequence of edge insertions/deletions preceded/ followed by the insertion/deletion of an isolated vertex, while it is trivial to handle the insertion/deletion of an isolated vertex.

7.1 Handling Edge Insertion

Consider the insertion of an edge (u, v). Let $L = N(u) \cap N(v)$ denote the set of common neighbors of u and v. The insertion of (u, v) causes the insertions of vertex v and a set of $|L|$ edges $\{(v, w)|w \in L\}$ into u's neighborhood induced subgraph $G_{N(u)}$. For each $w \in L$, we perform a Union operation $g[u]$.Union(v, w) to update the components and $score(u)$. For vertex v, $G_{N(v)}$ is updated in a similar way.

The insertion of (u, v) also affects $G_{N(w)}$ for each $w \in L$. We check the disjoint-set forest structure $g[w]$. If u, v belong to the same connected component before the edge insertion, then all components remain unchanged and so does $score(w)$. If u, v are in different components before the edge insertion, we merge the two components into one with a Union operation $g[w]$.Union(u, v) and update $score(w)$ accordingly.

Consider the graph G in Figure 10 (a) as an example. Suppose that $t = 2$ and the inserted edge is (r, q). $L = N(r) \cap N(q) = \{s, p_1\}$. Figure 10 (c) shows the updated $G_{N(r)}$ with the edge insertion. $G_{N(r)}$ has two new edges (p_1, q) and (s, q), but $score(r) = 1$ remains unchanged. For vertex $s \in L$, vertices r, q are now connected in the same component in $G_{N(s)}$ with the insertion of (r, q), so we update $score(s)$ from 0 to 1.

7.2 Handling Edge Deletion

Consider the deletion of an edge (u, v). To handle the edge deletion, we maintain a spanning tree for each connected component in the affected subgraphs $G_{N(u)}, G_{N(v)}$ and $G_{N(w)}$ where $w \in L$. For example, consider the component $P = \{p_1, \ldots, p_5\}$ of $G_{N(r)}$ in Figure 10 (b) and the corresponding spanning tree T_P in Figure 10 (d). The edges in the spanning tree are called *tree edges*, and other edges in the component are called *non-tree edges*, e.g., (p_1, p_2) is a tree edge and (p_1, p_5) is a non-tree edge.

For each $w \in L$, we consider updating $G_{N(w)}$ with the deletion of (u, v). We check whether (u, v) is a tree edge in the spanning tree of the component. If (u, v) is a non-tree edge, $score(w)$ remains unchanged because vertices u, v are still in the same component connected by the corresponding spanning tree. Continuing with the example above, the deletion of the non-tree edge (p_1, p_5) will not split the component P in $G_{N(r)}$, and p_1, p_5 are still in the same component. If (u, v) is a tree edge, then the deletion of (u, v) splits the spanning tree into two trees denoted as T_u and T_v. We will search for a replacement edge so as to reconnect T_u and T_v. If a replacement edge (u', v') exists, we insert (u', v') to connect T_u, T_v into a new spanning tree. Then the original component is still connected, and $score(w)$ remains unchanged. If the replacement edge does not exist, the deletion of (u, v) splits the original connected component into two components, and the corresponding spanning trees are T_u and T_v. So we update $score(w)$ accordingly. Maintaining the spanning tree can be implemented easily with the Union operation by keeping track of the bridge edge between two different components. In the example above, if a tree edge (p_1, p_2) is deleted, we can find a replacement edge (p_1, p_4) to reconnect the spanning tree in Figure 10 (d).

The deletion of (u, v) also affects $G_{N(u)}$ and $G_{N(v)}$. Consider u as an example. For all $w \in L$, we remove those non-tree edges (v, w) from $G_{N(u)}$, and remove those tree edges (v, w) from the spanning tree which is then split into multiple trees. Then we search for replacement tree edges to reconnect the spanning tree. Finally, we remove v from $G_{N(u)}$ and update $score(u)$. Figures 10 (e) and (f) show the updates of $G_{N(r)}$ and T_P with the deletion of (r, p_2).

The above techniques apply to updating both the actual score and the upper bound in our Top-k-search framework given edge insertions/deletions. In updating an upper bound $\overline{bound}(v)$ for vertex v, given an edge deletion as a tree edge, we only split the original spanning tree into two, but do not have to search for the replacement edge. This will only relax $\overline{bound}(v)$ without affecting the result correctness. This strategy can avoid the cost of finding the replacement edge and achieve higher efficiency.

Summary: Handling edge insertion is trivial using our disjoint-set forest structure, while handling deletion is more costly as it maintains the spanning tree. In the real-world networks, edge insertions are usually more frequent than deletions. Our update techniques do not increase the space complexity of Top-k-search.

8 Top-K Core-based Structural Diversity Search

Ugander et al. [25] gave another definition of structural diversity based on the core subgraph concept [6]. Their study showed that, the core-based structural diversity suffices to provide a positive predictor of future long-term engagement in a social network. In this section, we further study top-k structural diversity search using this definition.

Due to the different definition of structural diversity, the structural properties for the component-based measure (Definition 2) may not hold for the core-based measure. For example, in Figure 2 (a), vertices e, b and d form a triangle, and b, d belong to the same component in the 2-core of $G_{N(e)}$, as depicted in Figure 2 (b). But vertices e and d do not belong to the same component in the 2-core of $G_{N(b)}$, neither do vertices e and b belong to the same component in the 2-core of $G_{N(d)}$. This shows that Property 2 does not hold for the core-based definition. In addition, the Union-Find-Isolate data structure is designed for keeping the information of components that vertices belong to, however, we can not know whether two vertices belong to the same component in a t-core beforehand. Hence, our proposed Union-Find-Isolate structure and the upper bound which leverage such property can not be directly applied to the top-k core-based structural diversity search. In the following, we will derive a new upper bound and a new algorithm for computing the core-based structural diversity score.

8.1 Upper Bound of Core-based Structural Diversity

Similar to Lemma 1, a simple upper bound of the core-based structural diversity can be derived based on the vertex degree and the parameter t.

Lemma 8 *For any vertex $v \in V$, $score^*(v) \le \lfloor \frac{d(v)}{t+1} \rfloor$ holds.*

Proof First, the t-core subgraph of $G_{N(v)}$ has at most $d(v)$ vertices. Second, every vertex of t-core is connected with at least t other vertices, thus every component of the t-core has a size of at least $(t+1)$. Then the number of components in the t-core is no greater than $\lfloor \frac{d(v)}{t+1} \rfloor$, and $score^*(v) \le \lfloor \frac{d(v)}{t+1} \rfloor$ holds.

However, this upper bound can be too loose for efficient pruning. In the following, we introduce an important structural property, which is useful for designing a tighter upper bound. First we give a definition of *core value*.

Definition 6 (Core Value) The core value of a vertex $v \in V$ in a graph G, denoted by $\varphi(v)$, is the maximum integer t such that there exists a t-core subgraph of G that contains v.

Continue with the above example. For the graph in Figure 2 (a), vertex f has a core value of 2, and the other vertices have a core value of 3. Based on the core value, we can derive a new structural property as follows.

Property 3 If a vertex $v \in V$ has $\varphi(v) \le t$, then for each vertex $u \in N(v)$, v is not in the t-core of $G_{N(u)}$.

Proof We prove this property by contradiction. Suppose to the contrary that there exist two vertices v and u such that the t-core subgraph of $G_{N(u)}$, denoted as H, contains vertex v with $\varphi(v) \le t$. By the definition of t-core, each vertex in H has degree at least t. Since each vertex $w \in H$ is a neighbor of u, we can form a new graph H' by adding u and the edge (w, u) for every vertex $w \in H$ into H'. Then each vertex in H' has degree at least $t+1$, and H' is a seed of $(t+1)$-core of the graph G. Then, for vertex $v \in H'$, v has $\varphi(v) > t$ by the definition of core value, which is a contradiction.

A tighter upper bound is derived in Lemma 9 based on Property 3.

Lemma 9 *For any vertex $v \in V$, we have an upper bound of $score^*(v)$ as*

$\overline{bound}(v) = \lfloor \frac{d(v)-w(v)-q(v)}{t+1} \rfloor$,

where $w(v)$ and $q(v)$ are defined as follows:

$w(v) = |\{u : u \in N(v), \varphi(u) \le t\}|$,

$q(v) = |\{u : u \in N(v), \varphi(u) > t, |N(v) \cap N(u)| < t\}|$.

Proof First, according to Property 3, these $w(v)$ vertices with $\varphi(u) \le t$ is not in the t-core of $G_{N(v)}$.

Next, we prove that any vertex $u \in N(v)$ with $\varphi(u) > t$ and $|N(v) \cap N(u)| < t$ is not in the t-core of $G_{N(v)}$, because u has less than t neighbors in $G_{N(v)}$. Therefore, these $q(v)$ vertices are not in the t-core of $G_{N(v)}$.

Therefore, the maximum possible number of vertices in the t-core is $d(v) - w(v) - q(v)$. Similar to the proof of Lemma 8, we can derive $\overline{bound}(v) = \lfloor \frac{d(v)-w(v)-q(v)}{t+1} \rfloor$ as an upper bound of $score^*(v)$.

For example, in Figure 2 (b), let $t = 2$. We have $w(e) = 1$ because there is one vertex, f, with $\varphi(f) = 2 \le t$. We also have $q(e) = 1$ because there is one vertex, h, with $\varphi(h) = 3 > t$ and $|N(e) \cap N(h)| = 1 < t$. It follows that $\overline{bound}(e) = \lfloor \frac{7-1-1}{2+1} \rfloor = 1$, which is equal to $score^*(e) = 1$. This bound is tighter than the degree-based bound $\lfloor \frac{7}{2+1} \rfloor = 2$.

Algorithm 7 Core Decomposition

```
Input: $G = (V, E)$
Output: $\varphi(v)$ for each $v \in V$

1: Sort the vertices in $G$ in ascending order of their degree;
2: while ($G$ is not empty)
3:     let $d$ be the minimum vertex degree in $G$;
4:     while (there exists a vertex $v$ with degree of at most $d$)
5:         $\varphi(v) \leftarrow d$;
6:         remove $v$ and all edges incident to $v$ from $G$;
7:         reorder the remaining vertices in ascending order
               of their degree;
8: return $\varphi(v)$ for each $v \in V$;
```

8.2 Core-based Structural Diversity Computation

8.2.1 Top-K Search Framework

For top-k core-based structural diversity search, we adopt the Top-k-search framework in Section 5 with the new upper bound in Lemma 9. To assign an initial upper bound to every vertex, we apply core decomposition [4] on G to compute the *core value* of every vertex in G. For the self-completeness of this paper, the core decomposition algorithm [4] is outlined in Algorithm 7. The algorithm first sorts the vertices in G in ascending order of their degree. Then the algorithm iteratively removes from G a vertex v with the minimum degree, together with all the edges incident to it, and assigns d, the current minimum degree in G, as its core value $\varphi(v)$. Upon the removal of v, we also update the degree of the remaining vertices and reorder them according to their new degree. The algorithm terminates when all vertices are removed from G. In this way, we compute the core value of all vertices in G.

Next we initialize the upper bound of each vertex $v \in V$ as $\overline{bound}(v) = \lfloor\frac{d(v)-w(v)}{t+1}\rfloor = \lfloor\frac{|\{u:u\in N(v),\varphi(u)>t\}|}{t+1}\rfloor$ by Lemma 9, where we set the parameter $q(v) = 0$ at this stage. Note that $\overline{bound}(v)$ can be incrementally refined during the top-k search process, because the vertex $u \in N(v)$ with $\varphi(u) > t$ and $|N(v) \cap N(u)| < t$ can be identified when computing $score^*(u)$.

In summary, we make the following modifications in the Top-k-search framework (Algorithm 3) to adapt it to the core-based structural diversity search.

1. For initialization, we compute the core value $\varphi(v)$ for every $v \in V$ using Algorithm 7.
2. We initialize the upper bound in line 5 using $\overline{bound}(v) = \lfloor\frac{d(v)-w(v)}{t+1}\rfloor$.
3. We compute $\overline{bound}(v^*)$ according to Lemma 9 in line 8.
4. In line 15, a new A*-core-search algorithm is invoked, which will be introduced in details in Section 8.2.2.

8.2.2 A-based Search Algorithm*

Next we propose an efficient method for computing the core-based structural diversity score for a vertex. This new method, called A*-core-search, is shown in Algorithm 8. It has two main steps: (1) for a vertex v, it first applies core decomposition on $G_{N(v)}$ and computes the t-core of $G_{N(v)}$; and (2) it then invokes A*-bound-search (Algorithm 6) to efficiently compute the number of components in the t-core and returns $score^*(v)$.

In the first step for computing the t-core, a graph H is initialized to be $G_{N(v)}$. Since any vertex $u \in N(v)$ with $\varphi(u) \leq t$ is not in the t-core of $G_{N(v)}$ by Lemma 3, u is removed from the graph H, together with all the incident edges (lines 1-3). Then, for each vertex u in H, we compute the degree $d(u)$. If $d(u) < t$, the algorithm inserts u into the queue Q, which keeps the candidate vertices for removal from H, and updates $\overline{bound}(u)$ according to Lemma 9 (lines 4-13). Next, it performs core decomposition by iteratively removing vertex u in Q from H, and inserting new vertices with degree less than t into Q. This process terminates when Q is empty (lines 14-21). The remaining graph H is the t-core of $G_{N(v)}$.

In the second step, the algorithm applies the A* strategies in Algorithm 6 for computing the number of connected components in the t-core of H as $score^*(v)$, and returns it. Specifically, we modify Algorithm 6 by removing lines 25-26 (designed according to Property 2) and lines 29-32 (designed according to Property 1), since neither property is applicable for core-based structural diversity search. The other parts of Algorithm 6 remain unchanged.

8.2.3 Complexity Analysis

We analyze the time and space complexity of the core-based Top-k-search framework as follows.

Theorem 5 *The core-based* Top-k-search *framework using* A*-core-search *takes* $O(\sum_{(u,v)\in E} (\min\{d(u), d(v)\}+ (\log d(u) + \log d(v))))$ *time and* $O(m)$ *space.*

Proof First, in the initialization step, the algorithm applies core decomposition (Algorithm 7) on the whole graph G, which can be done in $O(m)$ time [4]. The total time cost of estimating the upper bound for all vertices is $O(m)$.

Second, for a vertex v, we consider the time cost of computing $score^*(v)$. Compared with A*-bound-search, it takes extra $O(\sum_{u\in N(v)} \min\{d(u), d(v)\})$ time for computing the degree of vertices in $N(v)$. Moreover, performing core decomposition on $G_{N(v)}$(lines 14-21) also takes extra $O(\sum_{u\in N(v)} \min\{ d(u), d(v)\})$ time. Finally, it invokes Algorithm 6 for computing the number of components in the t-core of $G_{N(v)}$. By Theorem 4, the top-k search process takes

Algorithm 8 A*-core-search (G, t, v)

Input: $G = (V, E)$, the core value threshold t, vertex v.
Output: $score^*(v)$.

```
1: $Q \leftarrow \emptyset$; $H \leftarrow G_{N(v)}$;
2: for $u \in N(v)$ do
3:     if $\varphi(u) \leq t$ then $H.delete(u)$;
4: for $u \in V_H$ do
5:     $d(u) \leftarrow 0$;
6:     if $d(u) > d(v)$ then $MinAdjL \leftarrow N(v)$;
7:     else $MinAdjL \leftarrow N(u)$;
8:     for $w \in MinAdjL$ do
9:         if $(w, u) \in E$ and $w \in V_H$ then
10:            $d(u) \leftarrow d(u) + 1$;
11:    if $d(u) < t$ then
12:        $Q.push(u)$;
13:        Update $\overline{bound}(u)$ by Lemma 9;
14: while $Q \neq \emptyset$
15:    $u \leftarrow Q.DeQueue()$; $H.delete(u)$;
16:    if $d(u) > d(v)$ then $MinAdjL \leftarrow N(v)$;
17:    else $MinAdjL \leftarrow N(u)$;
18:    for $w \in MinAdjL$ do
19:        if $(w, u) \in E$ and $d(w) \geq t$ then
20:            $d(w) \leftarrow d(w) - 1$;
21:            if $d(w) < t$ then $Q.push(w)$;
22: Invoke Algorithm 6 to compute $score^*(v)$ from $H$;
23: return $score^*(v)$;
```

$O(\sum_{(u,v) \in E} (\min\{d(u), d(v)\} + (\log d(u) + \log d(v))))$ time.

Compared with A*-bound-search, A*-core-search needs extra $O(n)$ space for storing the core value for all vertices. Thus, the space complexity is still $O(m)$.

Hence, Theorem 5 is established.

8.3 Network Structural Properties Analysis

In this subsection, we discuss the impact of real world network properties on the performance of the proposed algorithms for CC-TopK and Core-TopK. We consider three well-known structural properties, namely, *power-law degree distribution*, *average path length*, and *clustering coefficient*.

Power-law Degree Distribution: Social networks usually follow a power-law degree distribution, that is, a small percentage of nodes are high-degree ones and the vast majority of nodes are low-degree ones. According to our upper bound definition which depends on the node degree, those high-degree nodes will be examined first in the top-k search framework. As the degree difference between the high-degree and low-degree nodes can be quite large, we can easily find that many low-degree nodes are impossible to be the top-k answers simply based on their upper bound estimate; thus they can be pruned at an early stage. Therefore, our algorithms are expected to work more efficiently on graphs with *power-law degree distribution* than those with *uniform degree distribution*.

Average Path Length: Social networks usually have a small average path length, that is, the small-world effect. For a node in such a network, the neighbors of the node are more likely to be closely connected, leading to few large components. As a result, the node tends to have a low structural diversity score. In contrast, for a node in a network with a large average path length, the neighbors of the node are loosely connected or even scattered far apart, leading to many components. Thus the node tends to have a high structural diversity score. In the latter case, the upper bound estimate of the nodes tends to be tighter and thus provides more effective pruning for top-k search.

Clustering Coefficient: Social networks often have a high clustering coefficient, in which nodes tend to cluster together with high density of ties. Obviously, in a graph with a larger clustering coefficient, the probability of a triplet to form a triangle is higher. As a result, for the component-based structural diversity, Property 2 is easier to be satisfied and thus will be more frequently used for upper-bound refinement to reduce the overall computational cost. The clustering coefficient has no obvious impact on the algorithms for the core-based structural diversity since it does not make use of triangles for upper-bound refinement.

9 Experiments

We conduct extensive performance study to evaluate the algorithms proposed in this paper. All algorithms are implemented in C++ and all the experiments are conducted on the Linux operating system with 2.67GHz six-core CPU and 50GB main memory.

Evaluation metrics: We use the running time and the number of vertices whose structural diversity scores are computed in the search process as two metrics. The latter evaluates the number of vertices that are pruned by the algorithm.

Datasets: We use 13 real-world networks covering social, communication, collaboration, location-based networks and webgraphs. The network statistics are shown in Table 1. Except for Epinions, Digg and KDDTrack1[1] which are from their respective websites, the other 10 networks are downloaded from the Stanford Network Analysis Project(`snap.stanford.edu`). We treat all the networks as undirected.

9.1 Connected Component-based Structural Diversity

Comparison methods: To the best of our knowledge, we are the first to study top-k structural diversity search. In the literature, no algorithms have been proposed to address this problem yet. Thus, we compare our algorithms with the degree-based approach (Algorithm 1) which serves as a

[1] `https://www.kddcup2012.org`

Table 1 Network statistics (K $= 10^3$ and M $= 10^6$)

Name	$\|V_G\|$	$\|E_G\|$	d_{max}	Description
WikiVote	5**K**	104**K**	1065	Social networks
Epinions	76**K**	509**K**	3044	
Slashdot	82**K**	948**K**	2552	
Gowalla	196**K**	1.9**M**	14730	
Digg	771**K**	7.3**M**	17643	
KDDTrack1	1.9**M**	100.2**M**	456907	
EmailEnron	37**K**	368**K**	1383	Communication networks
EmailEuAll	265**K**	420**K**	7636	
WikiTalk	2.4**M**	5.0**M**	100029	
HepPh	12**K**	237**K**	491	Collaboration networks
AstroPh	19**K**	396**K**	504	
NotreDame	326**K**	1.5**M**	10721	Web graph
Flickr	80**K**	11.8**M**	5706	Flickr

baseline. Compared with the baseline method, our proposed algorithms produce exactly the same true top-k results. We evaluate four algorithms.

– Deg: The degree-based approach in Algorithm 1.
– Bou: Top-k-search equipped with bound-search (Algorithm 4) and $\theta = 1$.
– FB: Top-k-search equipped with fast-bound-search (Algorithm 5) and $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$.
– A*-B: Top-k-search equipped with A*-bound-search (Algorithm 6) and $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$.

In our experiments, we find that $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$ which is close to 1 always yields a good performance in the Top-k-search framework. For FB and A*-B, their performances are not very sensitive to the value of θ as long as $\theta \in (1.001, 1.05)$ on all datasets.

9.1.1 Efficiency Comparison

In this experiment, we compare the efficiency of different methods over all networks. We set $k = 100$ and $t = 2$. Similar results can be observed for other k and t values. Table 2 reports the results. We can see that A*-B is the most efficient, followed by FB, Bou, and Deg. Notice that the performance of A*-B, FB, and Bou which adopt the Top-k-search framework is substantially better than that of the degree-based algorithm Deg. The speedup ratio between Deg and A*-B defined as $R_s = t_{\mathsf{Deg}}/t_{\mathsf{A^*-B}}$ is between 2.1 and 69.1 (column 6 in Table 2). The result conforms with the complexity analysis in Section 6. In addition, we define the pruning ratio between Deg and A*-B as $R_p = S_{\mathsf{Deg}}/S_{\mathsf{A^*-B}}$, where S_{Deg} and $S_{\mathsf{A^*-B}}$ denote the number of vertices whose structural diversity scores are computed by the respective methods. The pruning ratio is between 2.1 and 11.1 over all networks (column 11 in Table 2). This result suggests that the upper bound derived in Lemma 3 is indeed tighter than the degree-based upper bound in Lemma 1.

When we compare Bou and Deg, the reduction of running time and search space by Bou demonstrates the effectiveness of the tighter upper bound in Lemma 3 and the Union-Find-Isolate data structure. When we compare Bou and FB, the reduction of running time by FB shows the effectiveness of the fast-bound-search method. Finally we observe that A*-B is more efficient than FB, which proves the effectiveness of the A* search order.

9.1.2 Performance Evaluation by Varying k

In this experiment, we evaluate the performance of all the methods by varying the parameter k. We set $t = 2$ and focus on six networks *Digg, WikiTalk, AstroPh, Gowalla, NotreDame* and *Flickr*. Similar results can be observed for other t values and on other networks. Figures 11 (a)-(f) depict the running time of different algorithms. Again, we can see that A*-B is the most efficient and Deg is the least efficient in most networks. The running time of A*-B is very stable as k increases.

Figures 12 (a)-(f) show the number of vertices whose structural diversity scores are computed by different methods on the six networks. A*-B is the clear winner by pruning the largest number of vertices, and Deg performs worst. In addition, we find that FB and Bou achieve very similar performance in terms of the number of vertices that are pruned. This is because $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$ in FB is very close to 1 (as listed in the last column of Table 2), and θ in Bou is set to 1 in our experiment. Thus, the pruning condition in FB and Bou is very similar. But on the other hand, FB runs much faster than Bou as shown in Figure 11, which conforms with the time complexity analysis in Theorems 2 and 3.

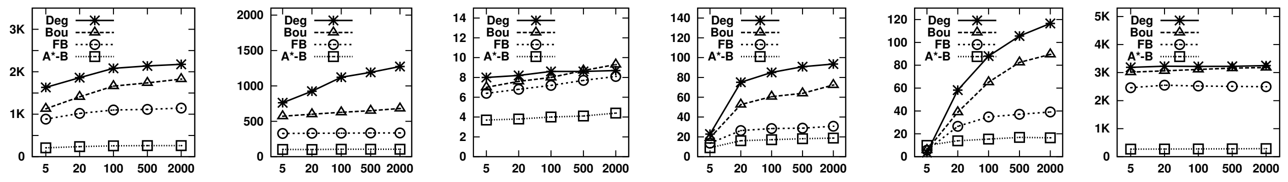
9.1.3 Performance Evaluation by Varying t

We evaluate the performance of all methods by varying the parameter t. In this experiment, we set $k = 100$ and similar results can be observed for other k values. Figures 13 (a)-(f) show the running time of different algorithms. Once again, A*-B is the most efficient algorithm, and Deg is the least efficient one. We also observe that in many cases, the running time of all methods increases with increasing t at first, but it may drop slightly when t further increases. A possible reason is that when t is large, the number of the *qualified components* (i.e., the components whose sizes are no less than t) reduces. Thus, by the estimated upper bound, the search space can be quickly pruned.

Figures 14 (a)-(f) show the number of vertices whose structural diversity scores are computed in different networks by varying t. We observe that A*-B prunes the most number of vertices, and Deg prunes the least number of vertices.

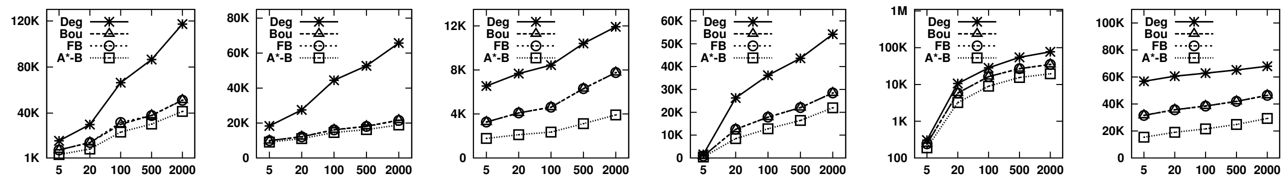
Table 2 Comparison of running time (wall-clock time in seconds) and search space (the number of vertices whose structural diversity score are computed in search process) of different algorithms. Here $k = 100$ and $t = 2$.

Network	Running Time					Number of Computed Vertices					$\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$
	Deg	Bou	FB	A*-B	R_s	Deg	Bou	FB	A*-B	R_p	
WikiVote	9.3	8.7	6.6	**3.1**	3.0	3362	2110	2111	**1612**	2.1	1.027
Epinions	37.6	35.9	24.9	**10.4**	3.6	11546	6349	6314	**4875**	2.4	1.017
Slashdot	31.4	26.9	19.7	**11.5**	2.7	12278	6459	6459	**5968**	2.1	1.015
Gowalla	83.8	60.3	28.3	**17.3**	4.9	36192	17883	17883	**12777**	2.8	1.012
Digg	2090.6	1670.1	1075.9	**253.0**	8.3	66403	30221	31866	**23465**	2.8	1.005
KDDTrack1	155087.0	7661.3	4370.0	**2244.1**	69.1	59163	7689	7668	**5333**	11.1	1.002
EmailEnron	10.6	10.1	6.9	**3.6**	3.0	6365	3031	3032	**1545**	4.1	1.023
EmailEuAll	12.5	11.1	7.9	**5.9**	2.1	4426	2045	2045	**1774**	2.5	1.020
WikiTalk	1153.7	642.1	331.0	**102.1**	11.3	44476	16156	16064	**14592**	3.0	1.007
HepPh	14.4	13.9	12.5	**2.3**	6.3	3988	2480	2480	**1394**	2.9	1.026
AstroPh	9.1	8.2	7.2	**3.9**	2.4	8439	4613	4613	**2352**	3.6	1.021
NotreDame	86.6	66.9	34.9	**16.0**	5.4	28347	16421	16417	**8976**	3.2	1.012
Flickr	3254.6	3136.9	2451.6	**270.1**	12.0	62814	38475	38460	**21544**	2.9	1.004



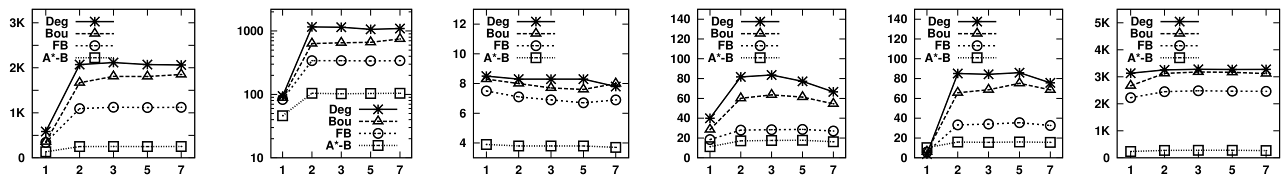
(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 11 Running time (in second) of different algorithms versus parameter k



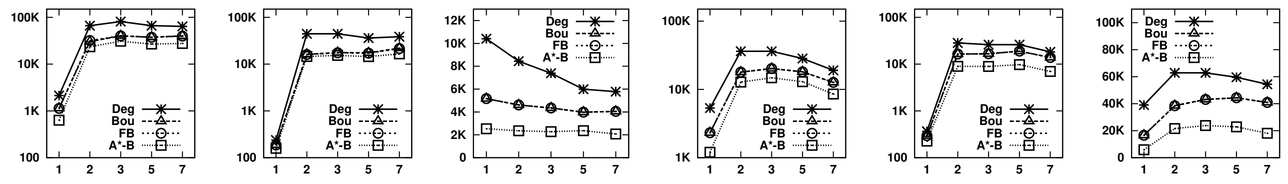
(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 12 Number of vertices whose structural diversity scores are computed versus parameter k



(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 13 Running time (in second) of different algorithms versus parameter t



(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 14 Number of vertices whose structural diversity scores are computed versus parameter t

9.1.4 Handling Update in Dynamic Networks

In this experiment, we evaluate the time for incrementally maintaining the top-k results when the input network is updated. For each network, we randomly insert/delete 1K edges, and update the top-k results after every edge insertion/deletion. The average update time per edge insertion/deletion is reported in Table 3. In addition, we report the batch update time for the 1K edge insertions/deletions. We repeat this experiment for 50 times and report the average performance. For comparison, we also report the time for computing the top-k results from scratch when the network is updated with an edge insertion/deletion.

The result in Table 3 shows that handling edge insertions is highly efficient. The update time per edge insertion is 0.01 or 0.02 millisecond on most networks, and the batch update time for 1K edge insertions is within 10 milliseconds on most networks. Handling edge deletions is more costly, because an edge deletion may trigger to check whether the two endpoints of the deleted edge are still in the same component or not in a number of neighborhood induced subgraphs. The update time per edge deletion is within 1 millisecond on most networks, and the batch update time for 1K edge deletions is less than 1 second on most networks. Finally we can see the incremental update (per edge as well as batch update of 1K edges) is several orders of magnitude faster than recomputing the top-k results from scratch.

9.2 Core-based Structural Diversity

Comparison methods: In this experiment, we focus on evaluating the top-k core-based structural diversity search, and compare two proposed algorithms.

– Core-Deg: a simple degree method with the degree-based upper bound in Lemma 8. Specifically, we adapt Algorithm 1 with the following changes: (1) in line 6, the upper bound is changed using the new upper bound in Lemma 8; and (2) in line 9, we compute the t-core of $G_{N(v^*)}$ and count the number of connected components in the t-core as $score^*(v^*)$.
– Core-A*: the core-based Top-k-search framework introduced in Section 8.2.1 equipped with A*-core-search (Algorithm 8) and $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$.

Compared with the baseline Core-Deg, Core-A* produces exactly the same true top-k results. For all datasets, we also find that $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$ which is close to 1 always yields a good performance in the core-based Top-k-search framework. For Core-A*, its performances is not very sensitive to the value of θ as long as $\theta \in (1.001, 1.05)$.

9.2.1 Performance Evaluation by Varying k

In this experiment, we evaluate the performance of Core-Deg and Core-A* by varying the parameter k. We set the parameter $t = 2$ by default and test on six networks *Digg, WikiTalk, AstroPh, Gowalla, NotreDame* and *Flickr*. Similar results can be observed for other t values and on other networks. Figures 15 (a)-(f) show the running time of different algorithms on the six networks. We can see that Core-A* is the clear winner by running much faster than Core-Deg on all networks, which demonstrates the advantage of the core-based Top-k-search framework and the A*-core-search method.

Figures 16 (a)-(f) show the number of vertices whose structural diversity scores are computed by different methods on the six networks. Core-A* prunes many more vertices than Core-Deg. This result shows that the upper bound derived in Lemma 9 is indeed much tighter than the degree-based upper bound in Lemma 8.

9.2.2 Performance Evaluation by Varying t

In this experiment, we evaluate the performance of all methods by varying the core number t on the six networks above. In this experiment, we set $k = 100$ and similar results can be observed for other k values. Figures 17 (a)-(f) show the running time of different algorithms. The running time of Core-A* is 2-4 times shorter than that of Core-Deg on all networks. In addition, the performance of Core-A* remains very stable as t increases.

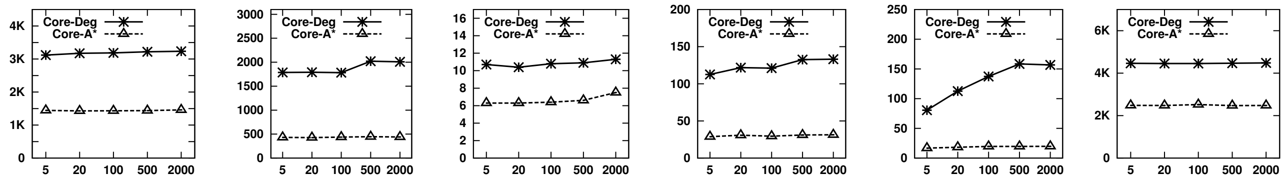
Figures 18 (a)-(f) show the number of vertices whose structural diversity scores are computed in different networks by varying t. Again Core-A* prunes many more vertices than Core-Deg.

9.3 Comparison Between A*-B and Core-A*

In this experiment, we evaluate the efficiency and memory usage of two different methods as component-based A*-B and core-based Core-A* with the same parameter setting $k = 100$ and $t = 2$ on the six networks *Digg, WikiTalk, AstroPh, Gowalla, NotreDame* and *Flickr*. Figure 19 (a) shows the running time of two algorithms on the six networks. A*-B is more efficient than Core-A* on all networks. It is because Core-A* additionally requires to compute the t-core and discard unqualified components. Figure 19 (b) shows the memory usage of two algorithms on the six networks. A*-B consumes more memory space than Core-A* on all networks, due to the additional Union-Find-Isolate data structure by A*-B. On the largest dataset KDD_Track1, the memory consumption by our method A*-B is 15.3 GB.

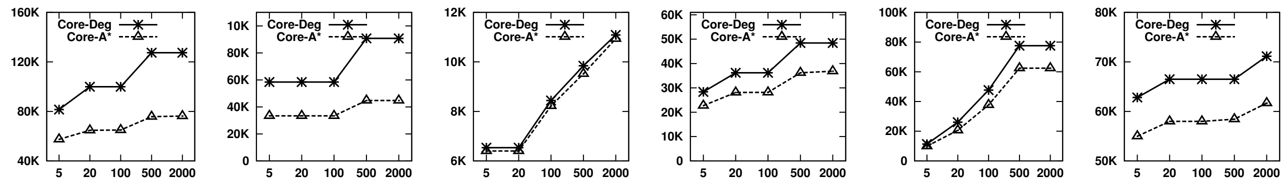
Table 3 Update Time (wall-clock time in milliseconds). Here $k = 100$ and $t = 2$.

Network	Insertion Per Edge	Insertion 1K Edges	Deletion Per Edge	Deletion 1K Edges	Computing from scratch
WikiVote	0.02	11.5	0.77	576	3100
Epinions	0.01	9.2	0.49	347	10400
Slashdot	0.01	7.3	0.35	317	11500
Gowalla	0.01	7.3	1.51	1179	17300
Digg	0.01	7.2	1.47	1404	253000
KDDTrack1	0.05	44.8	800	660139	2244100
EmailEnron	0.01	6.9	0.59	440	3600
EmailEuAll	0.01	5.2	0.16	162	5900
WikiTalk	0.01	6.6	1.52	1513	102100
HepPh	0.02	8.2	0.45	292	2300
AstroPh	0.02	10.7	0.38	326	3900
NotreDame	0.01	6.2	0.85	696	16000
Flickr	0.08	61.5	7.81	4943	270100



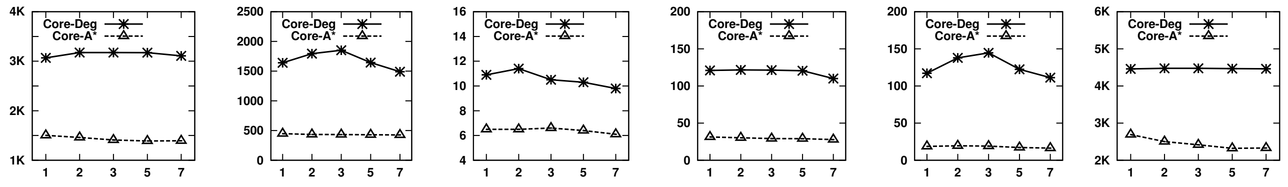
(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 15 Running time (in second) of different algorithms versus parameter k



(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 16 Number of vertices whose structural diversity scores are computed versus parameter k



(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

Fig. 17 Running time (in second) of different algorithms versus parameter t

9.4 Performance Evaluation on Synthetic Datasets

In this experiment, we generate a series of synthetic graph datasets, and evaluate the performance of our best methods A*-B and Core-A* by varying three different structural properties, namely, *power-law degree distribution*, *average path length*, and *clustering coefficient*. We set the same default parameter $k = 100$ and $t = 2$ for A*-B and Core-A* method on all the following synthetic graphs.

9.4.1 Performance Evaluation by varying Power-law Degree Distribution

The first tested synthetic networks are *power-law graphs*, in which the degrees of nodes follow a power-law distribution.

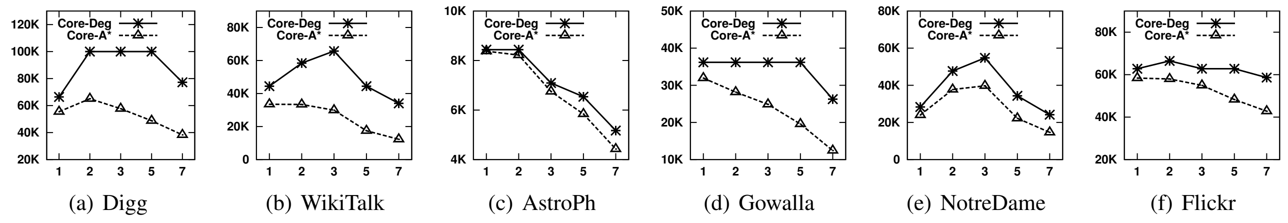


Fig. 18 Number of vertices whose structural diversity scores are computed versus parameter t

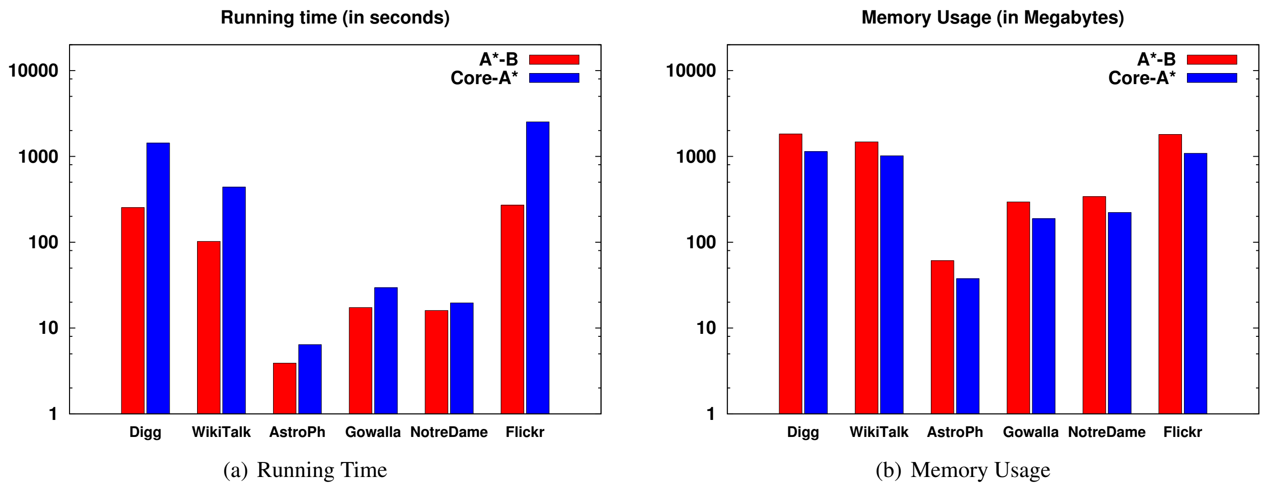


Fig. 19 Comparison of running time (wall-clock time in seconds) and main memory usage (in Megabytes) of two algorithms A*-B and Core-A*. Here $k = 100$ and $t = 2$ for both methods.

That is, the fraction $P(c)$ of nodes in the network having c neighbors goes for large values of c as $P(c) \sim c^{-\gamma}$, where γ is a parameter of power-law whose value is typically in the range $2 < \gamma < 3$. The smaller the γ is, the smaller the fraction of high-degree nodes is in all nodes. We generate a series of graphs using a preference attachment method in [11]. We set the node size $|V| = 100,000$ and edge size $|E| = 1,000,000$, and vary the different parameter γ from 2.2 to 2.8.

Figure 20 (a) shows the running time of two algorithms on power-law graphs by varying γ. As we can see that both our methods take a slight less time with the increase of parameter γ. Therefore, the performance of our methods can be regarded as stable on the graphs with different degree distributions. Figure 20 (b) shows the number of vertices whose structural diversity scores are computed by the two methods on different power-law graphs. With the increasing γ, both methods reduce the number of vertices whose structural diversity scores are computed. Moreover, A*-B is more efficient than Core-A* in terms of both running time and the pruning efficiency in all test cases.

9.4.2 Performance Evaluation by varying Average Path Length

In this experiment, we test a series of graphs with different *average path length*. The average path length of a graph can be defined as the average length of the shortest distance between all possible pairs of nodes in the graph. The average path length is a measure of the efficiency of information or mass transport on a network. We generate 5 random graphs with $100,000$ nodes and $500,000$ edges with different parameters of average path length from 2 to 5.5.

Figure 21 (a) and Figure 21 (b) show the running time and search space of two methods A*-B and Core-A* respectively. With the increasing of average path length in the graph, the running time and search space for both A*-B and Core-A* decrease slightly, which is consistent with the analysis for average path length in Section 8.3. Thus, in term of running time evaluation, the efficiency performance of both algorithms can be regarded as stable in Figure 21 (a) when varying the *average path length*.

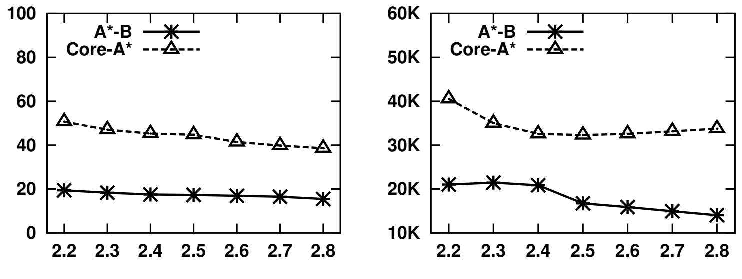


Fig. 20 Comparison of running time (wall-clock time in seconds) and search space (number of vertices whose structural diversity scores are computed) of two algorithms A*-B and Core-A* versus parameter γ *on power-law graphs*.

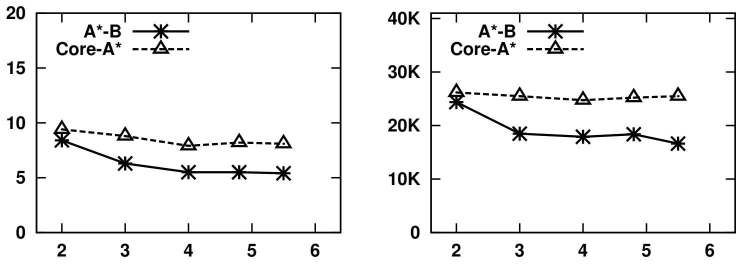


Fig. 21 Comparison of running time (wall-clock time in seconds) and search space (number of vertices whose structural diversity scores are computed) of two algorithms A*-B and Core-A* versus the parameter *average path length*.

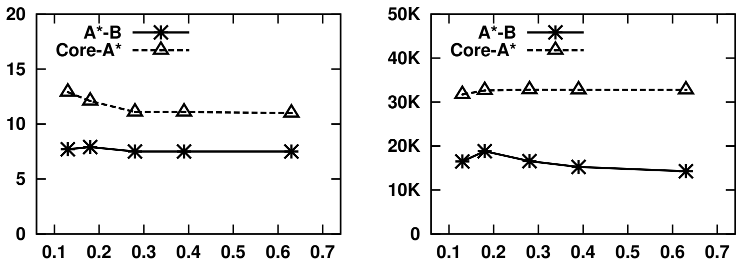


Fig. 22 Comparison of running time (wall-clock time in seconds) and search space (number of vertices whose structural diversity scores are computed) of two algorithms A*-B and Core-A* versus the parameter *clustering coefficient*.

9.4.3 Performance Evaluation by varying Clustering Coefficient

In this experiment, we test the performance of our methods on the graphs with different global clustering coefficient. The global clustering coefficient of a graph is the number of closed triplets over the total number of triplets, which can be computed as $\frac{2\cdot\sum_{(v,u)\in E}|N(v)\cap N(u)|}{\sum_{v\in V}d(v)*(d(v)-1)}$. The clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. We generate 5 random graphs with $100,000$ nodes and $500,000$ edges with different clustering coefficients from 0.13 to 0.63.

Figure 22 (a) and Figure 22 (b) show the running time and search space of two methods A*-B and Core-A* respectively. When the clustering coefficient increases, the processing time for A*-B decreases. Both two methods have a stable performance of running time and search space on all test cases by varying the clustering coefficient. Such results are consistent with the analysis for clustering coefficient in Section 8.3.

Summary: In above experiments, we can see that for graphs with the same number of nodes and edges, both A*-B and Core-A* have relatively stable performance when varying any of the three structural properties, namely, *power-law degree distribution*, *average path length*, and *clustering coef-*

ficient. The results are consist with the complexity analysis shown in Theorem 4 and Theorem 5 for the two algorithms respectively, as well as the property analysis shown in Section 8.3.

9.5 Case Study

9.5.1 Identifying Ambiguous Names

Name ambiguity has long been viewed as a challenging problem in social network analysis. For example, when we search a person named "Wei Wang" from the DBLP website, there are at least 52 distinct persons with the same name. Our top-k structural diversity search method provides a novel approach for identifying ambiguous names in a social network, which is the first and important step for name disambiguation. In this case study, we build a collaboration network from the DBLP dataset[2]. A vertex represents an author name and an edge is added between two authors if they have co-authored 3 times or more. The network contains 234,879 vertices and 541,814 edges.

We first apply the component-based structural diversity measure on the DBLP network for finding top-5 authors with the highest number of connected components of size over 2. The result is shown in Table 4. As we can see, these five names are indeed popular but ambiguous ones which can correspond to different distinct persons in the real world. For comparison, we select five famous authors who have a large degree, i.e., a large number of collaborators, and report their structural diversity score. Despite the large degree, their structural diversity score is far smaller than that of the ambiguous names. Intuitively, we can imagine that an ambiguous name corresponds to different distinct persons, each of who has his/her own research communities. This leads to a large number of non-overlapping research communities associated with an ambiguous name. In addition, we also observe that many of such communities are tiny and loosely connected. This is because the component-based structural diversity does not enforce a cohesive structure in the components.

We also apply the core-based structural diversity measure on the DBLP network for finding top-5 authors using 2-core subgraphs, and report the result in Table 5. By definition, the 2-core based measure discards all tree-shaped components, and counts the remaining cohesive components into the score. As a result, the obtained scores in Table 5 are smaller than the component-based scores in Table 4. Moreover, we observe that the core-based top-5 ambiguous names are different from the component-based top-5 names in Table 4. This suggests that these two diversity measures can complement each other.

[2] http://dblp.uni-trier.de/xml/

This case study shows that our top-k structural diversity search provides an effective mechanism for finding ambiguous names in a social network.

9.5.2 Words with Diverse Meanings

In this case study, we apply the two structural diversity measures on a word association network[3]. The expected result is to find words with the most diverse meanings, and to analyze and understand the different meanings of these words in different contexts. In this network, a vertex represents a word, and an edge between two words indicates that they are meaningfully related or strongly associated. The network contains 7,207 vertices and 31,784 edges.

We first query top-1 vertex in the word association network by counting the number of connected components of size over 2. The result is depicted in Figure 23. The word "black" has the highest structural diversity score as 9, indicating 9 distinct connected components in the neighborhood induced subgraph of "black", and each distinct component represents a certain meaning of "black". The largest connected component (depicted in red) contains 22 words, and most of those words can be roughly summarized by three words as "color", "race" and "dark". For example, "black" is a "color", and "black" is related to other colors such as "white", "red", "blue", "yellow", etc. For the other 8 connected components, each contains 2-3 words, and represents a distinct context of words associated with "black", such as, {"penguin", "tuxedo" }, {"panther", "cat" }, {"death", "widow", "funeral" }, and so on.

Next, we query top-2 vertices in the word association network using 2-core subgraphs. Two words "word" and "Christmas" have the highest two structural diversity scores of 4 and 3, which are shown in Figure 24. As we can see, each vertex in the 2-core component in Figure 24 has at least two neighbor words. Specifically, the word "word" in Figure 24 (a) has 4 distinct contexts of associated words with different meanings. For example, {"swear", "oath", "promise" } represent the synonym of "words" as "promise", and {"verb", "noun", "pronoun" } are different types of "word". The word "Christmas" has three distinct contexts of associated words, as shown in Figure 24 (b), {"reindeer", "sleigh", "Santa" } describe the "Santa", {"present", "gift", "package" } represent the "Christmas gifts" and {"tree", "ornament", "decoration" } are related to the "Christmas tree".

When we compare Figure 19 and Figure 20, using the component-based structural diversity, we can find words with many different meanings. However, the terms in a certain context may be loosely related to each other (e.g., "hair" and "blue" in the same connected component in Figure 19). On the other hand, using the core-based structural diversity,

[3] http://www.netcom-analyzer.org/datasets/166

Table 4 Ambiguous names (Top-5 structural diversity result based on size-2 connected component) and selected famous authors in DBLP network. Ambiguous names obviously have much higher structural diversity scores than famous authors.

Ambiguous Name			Famous Authors		
Name	Degree	Score	Name	Degree	Score
Yang Liu	126	33	Christos Faloutsos	97	10
Xin Li	150	31	Philip S. Yu	140	5
Yan Zhang	157	29	Jiawei Han	132	4
Wei Wang	117	29	H. V. Jagadish	62	4
Wei Liu	151	28	Gerhard Weikum	103	2

Table 5 Ambiguous names (Top-5 structural diversity result based on 2-core) and selected famous authors in DBLP network. Ambiguous names obviously have much higher structural diversity scores than famous authors.

Ambiguous Name			Famous Authors		
Name	Degree	Score	Name	Degree	Score
Yang Yang	107	15	Christos Faloutsos	97	6
Yu Zhang	105	15	Philip S. Yu	140	3
Ming Li	149	15	H. V. Jagadish	62	3
Peng Wang	80	14	Jiawei Han	132	2
Xin Li	150	13	Gerhard Weikum	103	2

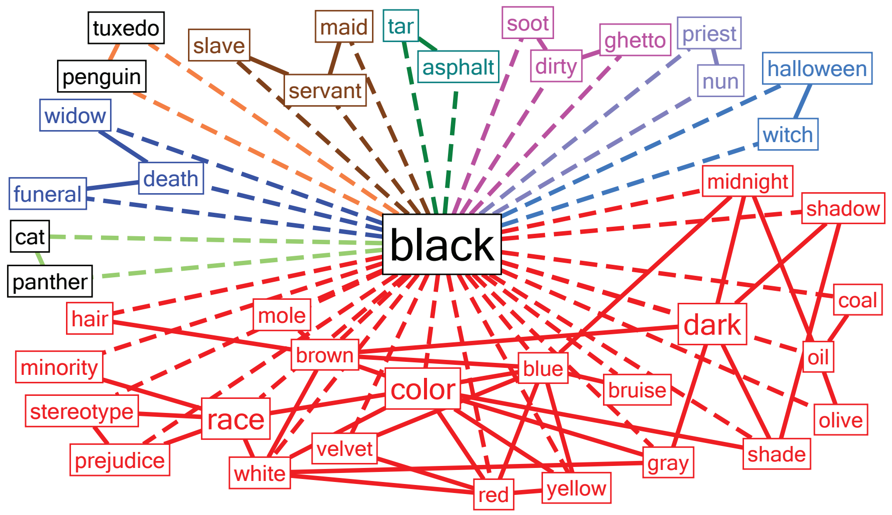


Fig. 23 Top-1 structural diversity based on size-2 connected component in word association network. Here "black" has the highest structural diversity score as 9.

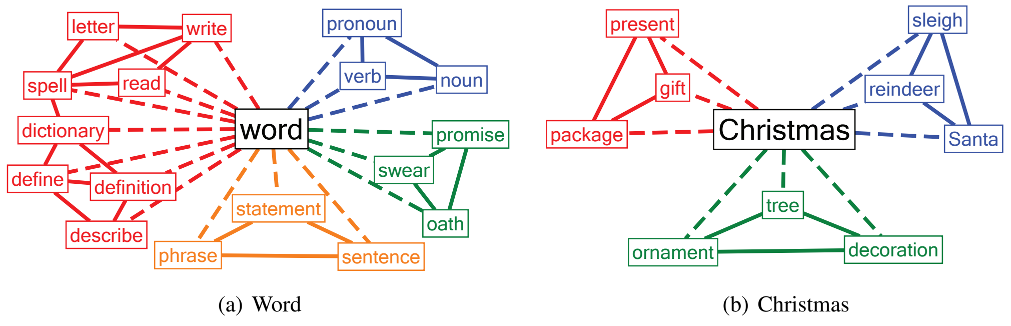


Fig. 24 Top-2 structural diversity based on 2-core in word association network. Here "word" and "Christmas" respectively has the top-2 highest structural diversity score as 4 and 3.

the terms in each component are highly related to each other. But the core-based measure may discard some loosely connected components, thus fail to extract the corresponding meanings of a word.

This case study shows that our top-k structural diversity search can be applied for finding different meanings of a word, which is a fundamental problem in natural language understanding.

10 Related Work

To the best of our knowledge, top-k structural diversity search has not been studied before. In the following, we briefly review the existing work that are related to ours.

First, our work is closely related to the work on top-k query processing. The goal of top-k query processing is to find k objects with the highest rank based on some predefined ranking function. A commonly used framework for this problem is to examine the candidates in a heuristic order and prune the search space using an upper bound. After the seminal work by Fagin et al. [12,13], a large number of studies on top-k query processing have been done for different application scenarios, such as processing distributed preference queries [5], keyword queries [20], set similarity join queries [27]. Recently, many studies take the diversity into consideration in top-k query processing, in order to return diversified ranking results [28,21,1,19,2,29]. A comprehensive survey of top-k query processing can be found in [15].

Second, our proposed techniques are related to the algorithms for the triangle listing problem, which is to find all triangles in a graph. Itai and Rodeh in [16] first proposed an $O(m^{1.5})$ algorithm for the triangle listing problem. In [18], Latapy proved that the time complexity $O(m^{1.5})$ is optimal. Subsequently, Schank and Wagner [24,23] proposed a simpler and particularly fast solution with the optimal complexity based on the vertex ordering and efficient lookup of the adjacency lists for neighborhood testing. Recently, Chu and Cheng [8] proposed an I/O-efficient algorithm for triangle listing in a massive graph, which cannot fit into the main memory. In this paper, we study the top-k structural diversity search problem. The complexity of our algorithm is proved to be $O(m^{1.5})$.

11 Conclusions

In this paper, we study the top-k structural diversity search problem motivated by a number of network analysis applications. We develop a novel Top-k-search framework to tackle this issue. Specifically, we design a Union-Find-Isolate data structure to keep track of the known structural information of each vertex, and an effective upper bound for pruning. For further speeding up the structural diversity evaluation, several well-designed search strategies are proposed. Our Top-k-search framework can be generalized to handle both the connected component based measure and the core base measure for structural diversity search. We evaluate the proposed algorithms on real-world large networks and synthetic graphs, and the results demonstrate the effectiveness and efficiency of the proposed algorithms.

ACKNOWLEDGMENTS

This work is supported by the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411211, 411310, 418512, and the Chinese University of Hong Kong Direct Grant No. 4055015.

References

1. R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong. Diversifying search results. In *WSDM*, pages 5–14, 2009.
2. A. Angel and N. Koudas. Efficient diversity-aware search. In *SIGMOD*, pages 781–792, 2011.
3. L. Backstrom, D. P. Huttenlocher, J. M. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, pages 44–54, 2006.
4. V. Batagelj and M. Zaversnik. An o (m) algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.
5. K. Chang and S. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.
6. J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.
7. N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.
8. S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *KDD*, pages 672–680, 2011.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.
10. P. S. Dodds and D. J. Watts. Universal behavior in a generalized model of contagion. *Physical Review Letters*, 92:218701, 2004.
11. S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of growing networks with preferential linking. *Physical Review Letters*, 85(21):4633, 2000.
12. R. Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.
13. R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.
14. X. Huang, H. Cheng, R.-H. Li, L. Qin, and J. X. Yu. Top-k structural diversity search in large networks. *PVLDB*, 6(13):1618–1629, 2013.
15. I. Ilyas, G. Beskales, and M. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.
16. A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.
17. H. Kwak, C. Lee, H. Park, and S. B. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.
18. M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.
19. R.-H. Li and J. X. Yu. Scalable diversified ranking on large graphs. In *ICDM*, pages 1152–1157, 2011.

20. Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, 2007.
21. L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *PVLDB*, 5(11):1124–1135, 2012.
22. D. M. Romero, B. Meeder, and J. M. Kleinberg. Differences in the mechanics of information diffusion across topics: idioms, political hashtags, and complex contagion on twitter. In *WWW*, pages 695–704, 2011.
23. T. Schank. Algorithmic aspects of triangle-based network analysis. *Ph.D. Dissertation, University Karlsruhe*, 2007.
24. T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, pages 606–609, 2005.
25. J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences*, 109(16):5962–5966, 2012.
26. D. J. Watts and P. S. Dodds. Influentials, networks, and public opinion formation. *J. Consum Res*, 34:441–458, 2007.
27. C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.
28. Y. Zhang, J. Callan, and T. Minka. Novelty and redundancy detection in adaptive filtering. In *SIGIR*, pages 81–88, 2002.
29. X. Zhu, J. Guo, X. Cheng, P. Du, and H. Shen. A unified framework for recommending diverse and relevant queries. In *WWW*, pages 37–46, 2011.