# Top-K Structural Diversity Search in Large Networks

Xin Huang[†], Hong Cheng[†], Rong-Hua Li[‡], Lu Qin[†], Jeffrey Xu Yu[†]

[†]*The Chinese University of Hong Kong*

[‡]*Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University*

{xhuang,hcheng,rhli,lqin,yu}@se.cuhk.edu.hk

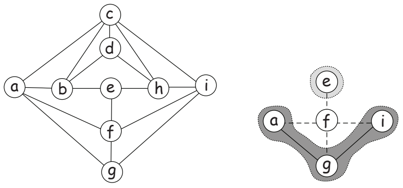
## ABSTRACT

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 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 heuristic search strategies are proposed. Extensive experimental studies are conducted in 13 real-world large networks, and the results demonstrate the efficiency and effectiveness of the proposed methods.

## 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 medias for users to communicate with each other and to spread information in the real world [14]. 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* [22], 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 [9, 3, 24]. However, such models have recently been challenged [19, 22], as the social contagion process is typically more complex and the social decision can depend more subtly on the network structure. Ugander et al. [22] 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, and the process that a user becomes an engaged user after joining. 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 [22], the analysis of structural diversity in a social network can be beneficial to a wide range of application domains, for example, political campaign, the promotion of health practices, marketing, and so on.



(a) $G$ (b) $G_{N(f)}$

**Figure 1: A running example**

Among all of these applications, a fundamental problem is to find the individuals in a social network with high structural diversity. Given a social network, we study a problem of finding top-$k$ individuals with the highest structural diversity in this paper. Following the definition in [22], 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.

To find the top-$k$ vertices with the highest structural diversity, 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 [12] can be used, which finds the top-$k$ answers according to some heuristic 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 heuristic search order in the computation.

In this paper, we propose several efficient and effective techniques to address these issues. 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 compo-

*Proceedings of the VLDB Endowment, Vol. 6, No. 13*

nent 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. The main contributions are summarized as follows.

- We develop a novel Top-k-search framework to tackle the top-$k$ structural diversity search problem. 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 heuristic 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.
- We conduct extensive experimental studies on large real networks to show the efficiency of our proposed methods.

The rest of this paper is organized as follows. We formulate the top-$k$ structural diversity search problem in Section 2. We present a simple degree-based algorithm in Section 3. A novel Top-k-search framework is proposed in Section 4. We design two heuristic search strategies in Section 5 and discuss update in dynamic networks in Section 6. Extensive experimental results are reported in Section 7. We discuss related work in Section 8 and conclude this paper in Section 9.

## 2. PROBLEM STATEMENT

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 a 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 (STRUCTURAL DIVERSITY [22]). *Given an integer $t$ where $1 \leq t \leq n$, the structural diversity of 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. We define the top-$k$ structural diversity search problem as follows.

**Problem definition**: Given a graph $G$ and two integers $k$ and $t$ where $1 \leq k, t \leq n$, the goal of 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. Suppose that $k = 1$ and $t = 1$. Then, $\{e\}$ is the answer, as $e$ is the vertex with the highest structural diversity ($score(e) = 3$). 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 [15].

## 3. A SIMPLE DEGREE-BASED APPROACH

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

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 $\mathcal{S}$. The following example illustrates how Algorithm 1 works.

EXAMPLE 1. *Consider the graph in Figure 1 (a). Suppose that $k = 1$ and $t = 1$. The running process on this graph is illustrated in Figure 2. 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 $\min_{v \in \mathcal{S}} score(v) = score(e) = 3$ and $\overline{bound}(g) = 3 \leq \min_{v \in \mathcal{S}} score(v)$. Therefore, Algorithm 1 can save one structural diversity computation.*

**Algorithm 1** degree $(G, k, t)$

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

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

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

**Figure 2: 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 [8]. 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. Put it all together, the time complexity of Algorithm 1 is $O(\sum_{v \in V}(d(v))^2)$. For the space complexity, 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 $O(\sum_{v \in V} d(v) \cdot d_{max}) = O(md_{max}) \subseteq O(mn)$.*

## 4. A NOVEL TOP-K SEARCH FRAMEWORK

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.

### 4.1 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 $G_{N(g)}$ ($G_{N(a)}$).

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.

### 4.2 Disjoint-Set Forest Data Structure

We modify the classical disjoint-set forest data structure and the Union-Find algorithm [8] 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 : Following [8], the Find-Set $(x)$ procedure is to find the root of $T[x]$ using the *path compression* heuristic.

Union : Following [8], the Union$(x, y)$ procedure applies the *union by rank* heuristic to union two trees $T[fx]$ and $T[fy]$ which $x$ and $y$ belong to respectively. $fx$ and $fy$ are the roots of these two trees. 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 3 (a). Since vertices $a$ and $g$ are connected, we invoke $g[f]$.Union($a, g$) and the resulted structure is shown in Figure 3 (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 3 (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)

**Figure 3: 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" heuristics 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 [8], thus is omitted. □

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

## 4.3 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 3. *In the disjoint-set forest structure $g[v]$, if $u \in N(v)$ and $T[u].parent = 0$, we denote $S_u = \{u\}$ as an* identified size-1 set, *and $|S_u| = 1$. If $u \in N(v)$, $T[u].parent = u$, we denote $S_u = \{w \in N(v) :$ Find-Set$(w) = u\}$ as an* unidentified set, *and $|S_u| = T[u].count$.*

By Definition 3, 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 3 (c). $S_e = \{e\}$ is an *identified size-1 set* and $T[0].count = 1$. Both $S_g = \{a, g\}$ and $S_i = \{i\}$ are *unidentified sets*.

**Algorithm 2** Union-Find-Isolate

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

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

procedure Union ($x, y$)
$fx \leftarrow$Find-Set ($x$); $fy \leftarrow$Find-Set ($y$);
if $fx \neq fy$ then
    if $T[fx].rank > T[fy].rank$ then
        $T[fy].parent \leftarrow fx$;
        $T[fx].count \leftarrow T[fx].count + T[fy].count$;
    else
        $T[fx].parent \leftarrow fy$;
        $T[fy].count \leftarrow T[fx].count + T[fy].count$;
        if $T[fx].rank = T[fy].rank$ then
            $T[fy].rank \leftarrow T[fy].rank + 1$;

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

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

LEMMA 3. *Let $S = \{S_1, \ldots, S_l\}$ 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 $S$ 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. Second, 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 3 (c), suppose $t = 2$, we obtain $S = \{S_e, S_g, S_i\}$ 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.

**Algorithm 3** Top-k-search

**Input:** $G = (V, E)$, the top-$k$ value $k$, 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 $\mathcal{S}$;
```

## 4.4 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 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 bound-search (Algorithm 4) to compute $score(v^*)$. 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).

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

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

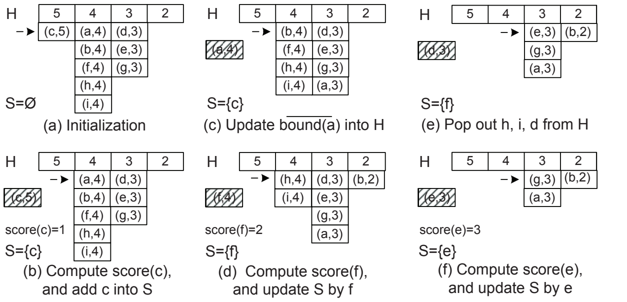
```
 1: $R \leftarrow \emptyset$;
 2: for $u \in N(v)$ and $T[u].parent \neq 0$ do $R \leftarrow R \cup \{u\}$;
 3: for $u \in R$ do bound-bfs $(u)$;
 4: return count-components $(g[v], t)$;

 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 $score$;
```

**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.

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 4. First, we push each vertex v with the upper bound $\lfloor \frac{d(v)}{t} \rfloor$ into $\mathcal{H}$, as shown in Figure 4 (a). Second, 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 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. Third, we push vertex c into $\mathcal{S}$, as shown in Figure 4 (b). In the next iteration, we pop vertex a from $\mathcal{H}$ with 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*



(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

**Figure 4: Illustration of** Top-k-search **with** bound-search **running on the graph in Figure 1 (a).** $k = 1$**,** $t = 1$**, and** $\theta = 1$**.**

*$\theta \cdot \overline{bound}(a) < topbound$ and $\overline{bound}(a) > \min_{v \in \mathcal{S}} score(v)$, we push $(a, 3)$ into $\mathcal{H}$ again, as shown in Figure 4 (c). When the algorithm goes to process vertex $f$, 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 $f$, as shown in Figure 4 (d). After that, we pop vertices $h, i, d$ from $\mathcal{H}$ in turn. 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, as shown in Figure 4 (e). Next we pop vertex e, compute $score(e) = 3$ and update $\mathcal{S}$ by e, as shown in Figure 4 (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 scores, while the previous* degree *algorithm performs eight computations of structural diversity score which is clearly more expensive.*

## 4.5 Complexity Analysis

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

PROOF. We need to maintain $a$, $b$ and $c$ in $g[v]$ to compute $\overline{bound}(v)$. 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 computed 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 4). 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 $O(n)$. □

**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 $(v)$;
```

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 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, computing the upper bounds for all vertices takes $O(\frac{m}{t})$ time. Therefore, the time complexity of Algorithm 3 is $O\left(\sum_{v \in V} (d(v))^2\right)$.

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. □

# 5. FAST COMPUTATION OF 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$^*$-based search method which uses a new search order and a new termination condition.

## 5.1 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.

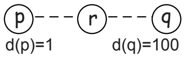


**Figure 5: $G_{N(r)}$ has two vertices $p$ and $q$ with degree 1 and 100**

To show the effectiveness of this improvement, we consider an example $G_{N(r)}$ in Figure 5. 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 $|N(p)| + |N(r)| = 3$.

**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 $score(v)$. 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 $O(\frac{m}{t} + n) \subseteq O(m)$.

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 2. *According to [6], $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m)$ where $\rho$ 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 $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) \subseteq O(\rho m) \subseteq O(m^{1.5})$.*

## 5.2 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. Take Figure 6 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$^*$-based heuristic search method 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 heuristic function to determine the component visiting order in the A$^*$ search process.

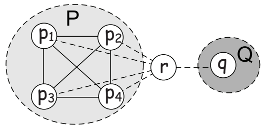


**Figure 6: $G_{N(r)}$ containing two components $P$ and $Q$**

DEFINITION 4. *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 6 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 heuristic 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 7 (a) shows the neighborhood induced subgraph $G_{N(r)}$, and Figure 7 (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 7 (c).

**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 $(g[v], t)$;
```

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 $tcost_x = T[x].cost$, 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 the 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 7. After initialization, we pop the first component $(s, 8)$ from $\mathcal{TC}$, as shown in step 1 (Figure 7 (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 7 (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 7 (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.

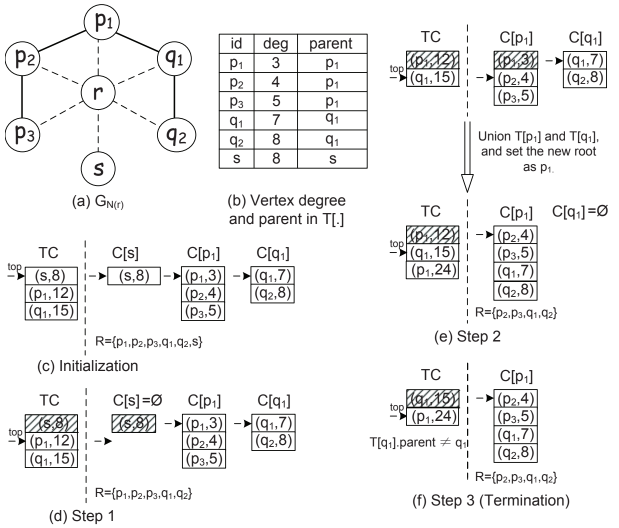


**Figure 7: A$^*$-bound-search example for computing $score(r)$**

**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 [8], which can 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 $O(\sum_{(u,v) \in E}(\min\{d(u), d(v)\} + \log d(u)))$ 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 take $d(v)$

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 takes at most $2d(v)$ Find-Set operations in lines 10 and 27 respectively. Hence, it takes $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\}+\sum_{v\in V}5d(v))=O(\sum_{(u,v)\in E}\min\{d(u),d(v)\})$ 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 $O(\sum_{u\in V} d(u)\log d(u))=O(\sum_{u\in V}\sum_{v\in N(u)}\log d(u))=O(\sum_{(u,v)\in E}\log d(u))$. The additional space overhead is $O(m)$.

Hence, Theorem 4 is established. □

REMARK 3. *The worst-case time complexity of the* Top-k-search *framework using* A*-bound-search *is bounded by* $O(\sum_{(u,v)\in E}(\min\{d(u),d(v)\}+\log d(u)))\subseteq O((\rho+\log d_{max})m)\subseteq O(m^{1.5})$, *where* $\rho$ *is the arboricity of the graph as mentioned in Remark 2.*

**Complexity Comparison:** We compare the time complexity of algorithms degree and Top-k-search.

According to Theorem 1, degree takes $O(\sum_{v\in V}(d(v))^2)$ time, which can be equivalently rewritten as $O(\sum_{v\in V}\sum_{u\in N(v)}d(u))$ $=O(\sum_{(u,v)\in E}(d(u)+d(v)))=O(\sum_{(u,v)\in E}(\max\{d(u),d(v)\}$ $+\min\{d(u),d(v)\}))=O(\sum_{(u,v)\in E}\max\{d(u),d(v)\})$.

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, which is obviously better than $O(\sum_{(u,v)\in E}\max\{d(u),d(v)\})$, the time complexity of degree, and $\sum_{(u,v)\in E}\min\{d(u),d(v)\}=\sum_{(u,v)\in E}\max\{d(u),d(v)\}$ 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 a social network. 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, according to Theorem 4, its time $O(\sum_{(u,v)\in E}(\min\{d(u),d(v)\}+\log d(u)))$ is also better than $O(\sum_{(u,v)\in E}\max\{d(u),d(v)\})$ of degree. The first part $O(\sum_{(u,v)\in E}\min\{d(u),d(v)\})$ is the same as Theorem 3, and the second part $O(\sum_{(u,v)\in E}\log d(u))$ is obviously better than $O(\sum_{(u,v)\in E}\max\{d(u),d(v)\})$ as $\log d(u)\leq\max\{d(u),d(v)\}$.

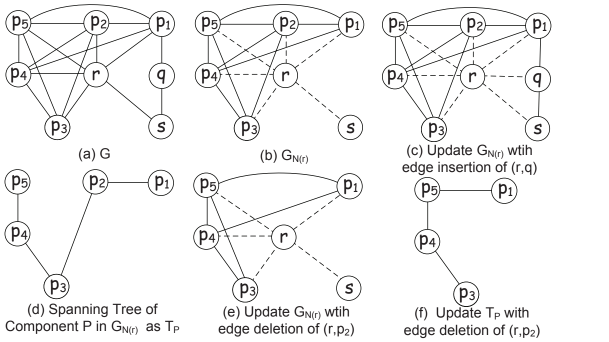
# 6. UPDATE IN DYNAMIC NETWORKS

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

## 6.1 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.



(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,p$_2$)

(f) Update $T_P$ with edge deletion of (r,p$_2$)

**Figure 8: Illustration of updates in a dynamic graph**

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 8 (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 8 (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.

## 6.2 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 8 (b) and the corresponding spanning tree $T_P$ in Figure 8 (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 8 (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 8 (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.

# 7. EXPERIMENTS

We conduct extensive performance studies 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.

**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 baseline. 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 $\theta$ as long as $\theta \in (1.001, 1.05)$ on all datasets. Due to the lack of space, we do not show the curves by varying $\theta$, and simply set $\theta = (\frac{n}{t})^{\frac{1}{\sqrt{m}}}$ for both FB and A*-B.

**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 publicly available real-world networks covering social, communication, collaboration 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.

## 7.1 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.

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

| Name | $\lvert V_G \rvert$ | $\lvert E_G \rvert$ | $d_{max}$ | 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 |

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 5. In addition, we define the pruning ratio between Deg and A*-B as $R_p = S_{\mathsf{Deg}}/S_{\mathsf{A*-B}}$, where $S_{\mathsf{Deg}}$ 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.

## 7.2 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 9 (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 10 (a)-(f) show the number of vertices whose structural diversity scores are computed by different methods in 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 $\theta$ 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 9, which conforms with the time complexity analysis in Theorems 2 and 3.

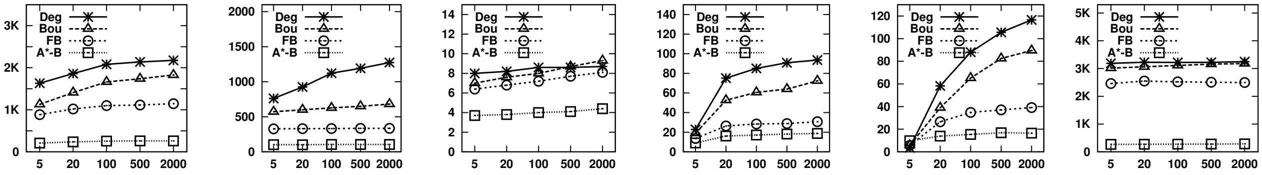
## 7.3 Performance Evaluation by Varying t

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 11 (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 in-

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

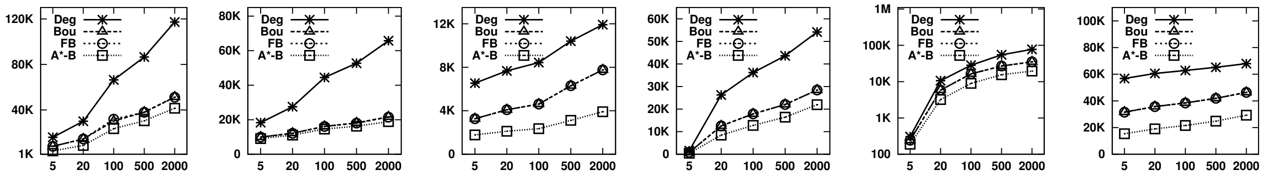
**Table 2: Comparison of running time (wall-clock time in seconds) and search space (the number of vertices whose structural diversity score are computed in search process) of different algorithms. Here $k = 100$ and $t = 2$.**

| 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 |



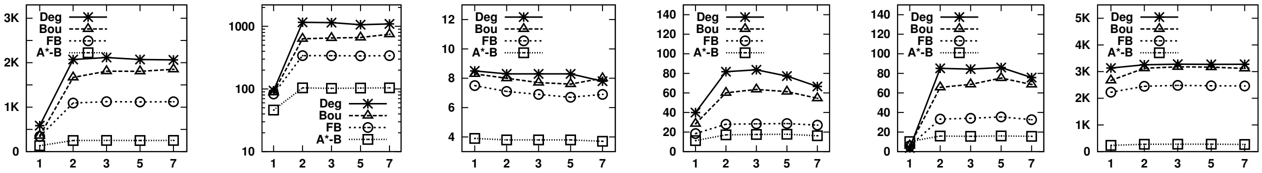
(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

**Figure 9: Running time (in second) of different algorithms versus parameter $k$**



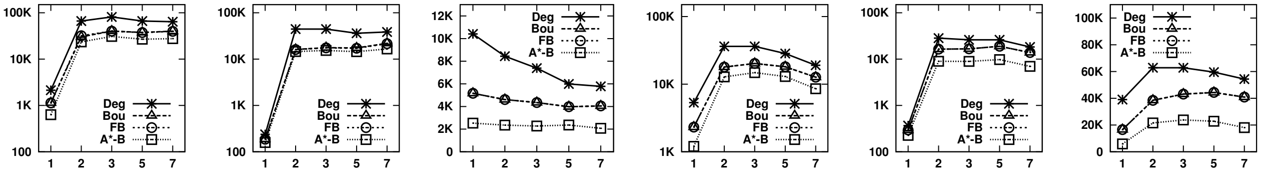
(a) Digg (b) WikiTalk (c) AstroPh (d) Gowalla (e) NotreDame (f) Flickr

**Figure 10: Number of vertices whose structural diversity scores are computed versus parameter $k$**



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

**Figure 11: Running time (in second) of different algorithms versus parameter $t$**



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

**Figure 12: Number of vertices whose structural diversity scores are computed versus parameter $t$**

creases. 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 12 (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.

## 7.4 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 in-

**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 |

duced 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.

# 8. 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 pre-defined 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. [10, 11], a large number of studies on top-$k$ query processing have been done for different application scenarios, such as processing distributed preference queries [4], keyword queries [17], set similarity join queries [25]. Recently, many studies take the diversity into consideration in top-$k$ query processing, in order to return diversified ranking results [26, 18, 1, 16, 2, 27]. A comprehensive survey of top-$k$ query processing can be found in [12].

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 [13] first proposed an $O(m^{1.5})$ algorithm for the triangle listing problem. In [15], Latapy proved that the time complexity $O(m^{1.5})$ is optimal. Subsequently, Schank and Wagner [21, 20] 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 [7] proposed an I/O-efficient algorithm for triangle listing in a massive graph, which cannot fit into the main memory.

# 9. 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. We evaluate the proposed algorithms on 13 large networks, and the results demonstrate the effectiveness and efficiency of the proposed algorithms.

Our study in this paper serves as the first step to the exciting topic of top-$k$ structural diversity search. [22] gives two more definitions of structural diversity based on k-core [5] and k-truss [23]. It would be interesting to extend the proposed techniques to these two definitions as a future work.

# 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.

# 10. 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] K. Chang and S. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.

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

[6] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[7] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *KDD*, pages 672–680, 2011.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[9] P. S. Dodds and D. J. Watts. Universal behavior in a generalized model of contagion. *Physical Review Letters*, 92:218701, 2004.

[10] R. Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[12] 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.

[13] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

[14] 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.

[15] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[16] R.-H. Li and J. X. Yu. Scalable diversified ranking on large graphs. In *ICDM*, pages 1152–1157, 2011.

[17] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, 2007.

[18] L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *PVLDB*, 5(11):1124–1135, 2012.

[19] 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.

[20] T. Schank. Algorithmic aspects of triangle-based network analysis. *Ph.D. Dissertation, University Karlsruhe*, 2007.

[21] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, pages 606–609, 2005.

[22] 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.

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

[24] D. J. Watts and P. S. Dodds. Influentials, networks, and public opinion formation. *J. Consum Res*, 34:441–458, 2007.

[25] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[26] Y. Zhang, J. Callan, and T. Minka. Novelty and redundancy detection in adaptive filtering. In *SIGIR*, pages 81–88, 2002.

[27] 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.