Efficient Top-k Edge Structural Diversity Search

Qi Zhang†, Rong-Hua Li†, Qixuan Yang†, Guoren Wang†, Lu Qin‡

†*Beijing Institute of Technology, Beijing, China;* ‡*University of Technology, Sydney, Australia;*
qizhangcs@bit.edu.cn; rhli@bit.edu.cn; qixuanyang@outlook.com; wanggrbit@126.com; Lu.Qin@uts.edu.au

***Abstract*—The structural diversity of an edge, which is measured by the number of connected components of the edge's ego-network, has recently been recognized as a key metric for analyzing social influence and information diffusion in social networks. Given this, an important problem in social network analysis is to identify top-k edges that have the highest structural diversities. In this work, we for the first time perform a systematical study for the top-k edge structural diversity search problem on large graphs. Specifically, we first develop a new online search framework with two basic upper-bounding rules to efficiently solve this problem. Then, we propose a new index structure using near-linear space to process the top-k edge structural diversity search in near-optimal time. To create such an index structure, we devise an efficient algorithm based on an interesting connection between our problem and the 4-clique enumeration problem. In addition, we also propose efficient index maintenance techniques to handle dynamic graphs. The results of extensive experiments on five large real-life datasets demonstrate the efficiency, scalability, and effectiveness of our algorithms.**

I. Introduction

Online social networks such as *Facebook*, *Twitter*, and *WeChat* have attracted much attention in recent years, and they are becoming an important tool for human beings to interact with each other and spread information in the real life. A central question in online social network analysis is the study of the spread of information on social networks. Such an information diffusion procedure on social networks is often termed as *social contagion*, which is similar as an epidemic spreading process.

A recent study [1] reveals that the probability of social contagion relies mainly on the number of connected components in a user's neighborhood, rather than depends on the number of friends in the neighborhood. A connected component in a user's neighborhood represents a social context of the user, and the number of social contexts is termed as structural diversity [1]. As indicated in [1], if a user has a higher structural diversity, he/she is more likely to participate in the social contagion procedure. The analysis of structural diversities for the users in a social network can be beneficial to a variety of applications such as viral marketing, political campaign, and promotion of health practices [1], [2].

Given the importance of structural diversity in network analysis, Dong et al. [3] study the structural diversity for a pair of vertices which is measured by the number of connected components in their common neighborhood. Dong et al. show that the structural diversity of a pair of vertices (u, v) has important practical applications for random graph design and friend suggestion in online social networks [3]. Inspired by this work, we study the problem of finding the top-k edges in a graph that have the highest structural diversities. The study of the top-k edges with highest structural diversities can be useful for many applications. For example, in social networks, the edges with highest structural diversities play crucial roles in promoting the information diffusion in the network, as those edges contain diverse social contexts and thus can promote the information spread over different social circles. In scientific collaboration networks, the edges with highest structural diversities may play important roles to improve collaborations over different communities, because those high-structural-diversity edges often connect multiple research communities in the network. In Natural Language Understanding (NLU), the edges with highest structural diversities indicate that the pairs of words have diverse meanings. Identifying these edges is very useful for analyzing and understanding the meanings of the pairs of words in different semantic contexts, which is a fundamental issue in NLU.

To solve the top-k edge structural diversity search problem, a straightforward algorithm is to calculate the structural diversities for all edges and then selects the top-k results. Such a straightforward algorithm, however, is very costly for large graphs, because the total cost for calculating the common neighborhood for each edge is very expensive in large graphs. To efficiently compute the top-k edges, the general idea of top-k structural diversity search algorithms [2], [4] can be used which explores the vertices based on a predefined ordering, and then applies some upper-bounding rules to prune the unpromising vertices. Motivated by this, we develop a new *dequeue-twice* online search framework using two basic upper bounds to prune the search space. To further improve the efficiency, we propose a novel index-based solution using near-linear space to process the top-k edge structural diversity search in near-optimal time. We present an efficient index construction algorithm by establishing an interesting connection between our problem and the 4-clique enumeration problem. In addition, we also devise efficient index maintenance techniques to handle dynamic graphs. To the best of our knowledge, this is the first work that studies indexing technique to solve the top-k structural diversity search problem. In summary, we make the following contributions.

An online search algorithm. We develop a new *dequeue-twice* online search framework with two basic upper bounds to find the top-k edges with highest structural diversities. We show that our new online search framework can significantly prune the edges that are definitely not contained in the top-k results.

An index-based algorithm. We develop a novel index structure, called ESDIndex, to efficiently support the top-k edge structural diversity search. We show that the ESDIndex consumes $O(\alpha m)$ space and can be used to process the top-k edge structural diversity query in $O(k \log m + \log n)$ time, where α is the arboricity of the graph [5] which is typically very small in real-life graphs [6], [7]. We present a new algorithm with time complexity $O((\alpha\gamma(n) + \log m)\alpha m)$ to create the ESDIndex based on an interesting connection between our problem and the 4-clique enumeration problem. In addition, we also propose a parallel index construction algorithm which

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00025

can achieve a high degree of parallelism. To handle dynamic graphs, we develop new index maintenance techniques which can efficiently update the ESDIndex when inserting/deleting an edge in the graph.

Extensive experiments. We conduct comprehensive experimental studies to evaluate the proposed algorithms using five large real-world datasets. The results show that our index-based algorithm processes the top-k edge structural diversity search in less than 1 millisecond on a large graph with more than 1 million vertices and 22 million edges, and it is at least four orders of magnitude faster than the online search algorithm. On the same large graph, the results also show that the ESDIndex can be constructed in around 266 seconds and the size of ESDIndex is around 5 times larger than the graph size. We also examine two case studies on DBLP and a word association network to evaluate the effectiveness of our algorithms. The results indicate that the top-k edges with the highest structural diversities are useful to find the important edges that connect different communities in DBLP and identify pairs of words with diverse meanings in the word association network.

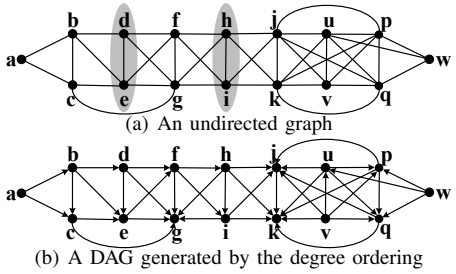
Organization. We introduce some important notations and formulate our problem in Section II. Section III presents the online search algorithm. The index-based solution is proposed in Section IV. We develop the index maintenance techniques in Section V. Section VI reports the experimental results. We survey the related work in Section VII and conclude this work in Section VIII.

II. Preliminaries

Let $G = (V, E)$ be an undirected and unweighted graph with $n = |V|$ vertices and $m = |E|$ edges. We denote the set of neighbors of a vertex u by $N(u)$, i.e., $N(u) = \{v \in V | (u, v) \in E\}$, and the degree of u by $d(u) = |N(u)|$. Similarly, the neighbors of an edge (u, v), denoted by $N(uv)$, is the set of vertices that are simultaneously adjacent to both u and v, i.e., $N(uv) = \{w \in V | (u, w) \in E, (v, w) \in E\}$. For a subset $S \subseteq V$, the subgraph of G induced by S is defined as $G_S = (V_S, E_S)$ where $V_S = S$ and $E_S = \{(u, v) | u, v \in S, (u, v) \in E\}$.

Given a graph $G = (V, E)$, we are able to obtain a directed graph $\vec{G} = (V, \vec{E})$ by assigning a direction for each edge in G. For each vertex u in $\vec{G}$, let $N^+(u)$ be the set of outgoing neighbors of u in $\vec{G}$, and $d^+(u) = |N^+(u)|$ be the out-degree of u. Denote by $\vec{G}_S$ the subgraph of $\vec{G}$ induced by S. A Directed Acyclic Graph (DAG) is a special directed graph in which the directed edges do not form cycles. Clearly, any undirected graph $G = (V, E)$ can be converted to a DAG $\vec{G}$ based on a total ordering on V [8]. Below, we introduce a total ordering on V based on the degree of vertices.

Degree ordering. We define a total ordering $\prec$ on V by an increasing ordering of the vertices by degrees (break ties by vertex ID). Specifically, for any two vertices u and v in V, we say that $u \prec v$ if and only if (1) $d(u) < d(v)$ or (2) $d(u) = d(v)$ and u has a smaller ID than v. Based on such a degree ordering $\prec$, we can construct a directed graph $\vec{G}$ by orientating each edge $e \in E$ from the low-rank vertex to the high-rank vertex, i.e., for each $(u, v) \in E$, we obtain a directed edge $(u, v) \in \vec{E}$ with $u \prec v$. It is easy to show that such a directed graph $\vec{G}$ is a DAG. Consider a graph G shown in Fig. 1(a). We can see that $e \prec f$, as $d(e) = d(f)$ and e has a smaller ID than f. The DAG generated by the degree ordering $\prec$ is shown in Fig. 1(b).



(a) An undirected graph

(b) A DAG generated by the degree ordering

Fig. 1. Running example

Below, we introduce a concept called edge ego-network, which is important to define the edge structural diversity.

Definition 1: (Edge ego-network) For an edge (u, v) in $G = (V, E)$, the ego-network of (u, v), denoted by $G_{N(uv)}$, is a subgraph of G induced by the vertex set $N(uv)$.

Example 1: Consider a graph G in Fig. 1(a). Take the edge (f, g) as an example, the set of its neighbors $N(fg)$ is $\{d, e, h, i\}$. The edge ego-network of (f, g) is $G_{N(fg)} = (\{d, e, h, i\}, \{(d, e), (h, i)\})$ which is illustrated in the shaded area of the graph in Fig. 1(a). □

Definition 2: (Edge structural diversity) Given a graph G and an integer $\tau \geq 1$, the edge structural diversity of (u, v) in G, denoted by $\mathsf{score}(u, v)$, is the number of connected components in $G_{N(uv)}$ with size no less than τ, where $G_{N(uv)}$ denotes the edge ego-network of (u, v).

Example 2: Reconsider the graph G in Fig. 1(a). The edge ego-network $G_{N(fg)}$ of the edge (f, g) (see the shaded area in Fig. 1(a)) contains 2 size-two connected components $\{d, e\}$ and $\{h, i\}$. If $\tau = 1$ or $\tau = 2$, we have $\mathsf{score}(f, g) = 2$, because the size of the two connected components is no less than τ. Similarly, for $\tau = 3$, $\mathsf{score}(f, g) = 0$ as there is no connected component with size greater than 3. □

Clearly, for each edge (u, v), we can perform a Breadth-First Search (BFS) on $G_{N(uv)}$ to compute $\mathsf{score}(u, v)$. Based on Definition 2, we formulate the top-k edge structural diversity search problem as follows.

Problem formulation. Given a graph G and two integers k and τ, the top-k edge structural diversity search problem is to identify the k edges in G with the highest structural diversities according to the component size threshold τ.

The following example illustrates the definition of our problem.

Example 3: Reconsider the graph G in Fig. 1(a). Suppose that $k = 3$ and $\tau = 2$. Then, we can easily derive that the three edges $\{(f, g), (h, i), (j, k)\}$ are the answers, because they have the highest structural diversities among all edges ($\mathsf{score}(f, g) = \mathsf{score}(h, i) = \mathsf{score}(j, k) = 2$). When $k = 3$ and $\tau = 5$, the answers are $\{(u, p), (u, q), (p, q)\}$. This is because there is only one connected component with size greater than 5 in $G_{N(up)}$, $G_{N(uq)}$ and $G_{N(pq)}$, and the structural diversities of the other edges in G are 0. □

Challenges. To solve the top-k edge structural diversity search problem, a straightforward algorithm is to compute the structural diversity for each edge, and then picks the top-k edges as the answers. Such an approach, however, is very costly for large graphs. This is because the algorithm needs to explore the edge ego-network $G_{N(uv)}$ to compute the structural diversity for each edge (u, v). The total size of all edge ego-networks

can be very large, thus the straightforward algorithm is very expensive for large graphs. Since we are only interested in the top-k results, we do not need to compute all edges' structural diversities. The challenges of the problem are (1) how to efficiently prune the edges that are definitely not contained in the top-k results, and (2) how to efficiently compute the structural diversities for all edges. To tackle these challenges, we will propose a new online search algorithm which can efficiently prune the unpromising edges and an index-based algorithm which is able to answer the top-k results in near-optimal time using near-linear space.

III. An online top-k search framework

In this section, we propose an online top-k search framework, called OnlineBFS, to solve our problem. The OnlineBFS algorithm is based on a novel *dequeue-twice* search framework with two different upper-bounding rules to prune the search space. Below, we first introduce two upper bounds of $\mathsf{score}(u,v)$ for each $(u,v) \in G$.

Min-degree upper bound. For each edge $(u,v) \in G$, we can easily derive that $\overline{\mathsf{ub}}(u,v) \triangleq \left\lfloor \frac{\min\{d(u),d(v)\}}{\tau} \right\rfloor$ is an upper bound of $\mathsf{score}(u,v)$. This is because the number of vertices of the edge ego-network of (u,v) is bounded by $\min\{d(u),d(v)\}$, thus the number of connected components with size no less than τ must be no larger than $\overline{\mathsf{ub}}(u,v)$.

Common-neighbor upper bound. For each $(u,v) \in G$, the common-neighbor upper bound is defined as $\left\lfloor \frac{|N(u) \cap N(v)|}{\tau} \right\rfloor$. Also, we can easily show that $\left\lfloor \frac{|N(u) \cap N(v)|}{\tau} \right\rfloor$ is a valid upper bound of $\mathsf{score}(u,v)$ by Definition 2, because the size of $G_{N(uv)}$ is bounded by $|N(u) \cap N(v)|$.

It should be noted that the common-neighbor upper bound is tighter than the min-degree upper bound (since $|N(u) \cap N(v)| \leq \min\{d(u),d(v)\}$), thus it can be more effective to prune the search space. However, such pruning benefit comes at computational costs, because the common-neighbor upper bounds are typically more expensive to calculate than the min-degree upper bounds.

A. The dequeue-twice search framework

Armed with the above two simple upper-bounding rules, we develop a *dequeue-twice* search framework to find the top-k edges with the highest structural diversities. The general idea is that we first explore the edges in G with large upper bounds, because such edges may have a high chance being the top-k answers. We make use of a priority queue $\mathcal{Q}$ to maintain all edges in G, and initialize the priority for each edge by its upper bound. When we pop an edge (u,v) from $\mathcal{Q}$ for the first time, we compute its exact structural diversity score $\mathsf{score}(u,v)$, and then push it again into $\mathcal{Q}$ using $\mathsf{score}(u,v)$ as its priority. When an edge (u,v) is popped from $\mathcal{Q}$ for the second time, then such an edge is an answer if the number of current results is smaller than k. Since each top-k edge will be dequeued twice, we refer to it as a *dequeue-twice* search framework. Note that the *dequeue-twice* search framework can avoid computing the exact structural diversity scores for the edges that have small upper bounds, thus it can significantly improve the efficiency of the algorithm.

The pseudo code of this framework is shown in Algorithm 1. As shown in Algorithm 1, the algorithm first pushes all edges into a priority queue $\mathcal{Q}$ using their upper bounds (min-degree or common-neighbor upper bounds) as priorities (lines 1-6). It also uses a *flag* variable for each edge (u,v) to indicate the number of times that (u,v) has been dequeued (line 4). The algorithm iteratively processes the edges based on their priorities until the top-k edges are found (lines 7-14). Note that in line 13, the algorithm performs a BFS on $G_{N(u^*v^*)}$ to compute the exact structural diversity $\mathsf{score}(u^*,v^*)$ for edge (u^*,v^*).

Algorithm 1: OnlineBFS

```
   Input: G = (V, E), two integers k and τ
   Output: The top-k edge set S
1  Let Q be a priority queue;
2  Q ← ∅; S ← ∅;
3  for (u, v) ∈ E do
4      flag(u, v) ← −1;
5      Compute an upper bound ub(u, v) for score(u, v);
6      Q.push((u, v), ub(u, v));
7  while |S| < k do
8      ((u*, v*), ub(u*, v*)) ← Q.pop();
9      flag(u*, v*) ← flag(u*, v*) + 1;
10     if flag(u*, v*) = 1 then
11         S ← S ∪ {(u*, v*)};
12         continue;
13     score(u*, v*) ← BFS(G_{N(u*v*)}, τ);
14     Q.push((u*, v*), score(u*, v*));
15 return S;
16 Procedure BFS(G_{N(uv)}, τ)
17 R ← the set of connected components in G_{N(uv)} by BFS;
18 cnt ← 0;
19 for each R ∈ R do
20     if |R| ≥ τ then cnt ← cnt + 1;
21 return cnt;
```

B. Analysis of Algorithm 1

Below, we analyze the correctness of Algorithm 1.

Theorem 1: Given a graph $G = (V,E)$ and two integers k and τ, Algorithm 1 correctly computes the top-k edges with the highest structural diversities.

Proof: Recall that Algorithm 1 iteratively processes the edges based on their priorities. When an edge (u,v) is dequeued from the priority queue $\mathcal{Q}$ for the second time, its priority must be equal to $\mathsf{score}(u,v)$. Moreover, at this point, $\mathsf{score}(u,v)$ is no less than the priorities of the other edges, thus it must also be no smaller than the structural diversities of the other edges. As a consequence, such a dequeued edge (u,v) must be a top-k answer if the answer size $|S|$ is smaller than k. Hence, the set S exactly contains the top-k answers when the algorithm terminates. □

Below, we analyze the time and space complexity of Algorithm 1. Let $d_{\max}$ be the maximum degree of the vertices in G, and α be the arboricity of G [5], [9]. The arboricity α of a graph G is defined as follows.

Definition 3: (Arboricity) Given a graph $G = (V,E)$ with $n \geq 2$, the arboricity α of G is defined as:

$$\alpha \triangleq \max_{\forall G_S=(V_S,E_S)\subseteq G} \left\lceil \frac{|E_S|}{|V_S|-1} \right\rceil. \quad (1)$$

The arboricity is an important metric to measure the sparsity of a graph, which is often very small for most real-world

graphs [6], [8]. Recently, it is widely used to bound the time complexity of many graph analysis algorithms [8], [10]–[15]. Below, we show that the time complexity of Algorithm 1 is also closely related to the arboricity of G.

Theorem 2: The worst-case time and space complexity of Algorithm 1 is $O((\alpha d_{\max} + \log m)m)$ and $O(m + n)$ respectively.

Proof: In the worst case, Algorithm 1 may compute the structural diversities for all edges. Note that for each edge (u, v), the algorithm needs to traverse the edge ego-network $G_{N(uv)}$ to compute $\mathsf{score}(u, v)$. Since the number of vertices in $G_{N(uv)}$ is bounded by $\min\{d(u), d(v)\}$, the number of edges in $G_{N(uv)}$ is bounded by $(\min\{d(u), d(v)\})^2/2$. Therefore, the total time cost to calculate the structural diversities for all edges can be bounded by $\sum_{(u,v)\in E}(\min\{d(u), d(v)\})^2/2$. Since $\min\{d(u), d(v)\} \leq d_{\max}$ and $O(\sum_{(u,v)\in E}\min\{d(u), d(v)\}) = O(\alpha m)$ [5], $O(\sum_{(u,v)\in E}(\min\{d(u), d(v)\})^2/2) \leq O(\alpha d_{\max} m)$. It is easy to verify that the algorithm takes $O(m)$ and $O(\alpha m)$ time to compute the min-degree upper bounds and the common-neighbor upper bounds, respectively. Thus, the time cost to compute the upper bounds is dominated by $O(\alpha d_{\max} m)$. In addition, the algorithm needs to maintain a priority queue with size $O(m)$, and the total maintenance cost is bounded by $O(m \log m)$. As a result, the worst-case time complexity of Algorithm 1 is $O((\alpha d_{\max} + \log m)m)$. We can easily derive that the space complexity of Algorithm 1 is $O(m + n)$, as the algorithm only needs to maintain several linear-size data structures (e.g., $\mathcal{Q}$ and $flag$). □

Note that in real-world graphs, the runtime of Algorithm 1 can be much lower than the worst-case time complexity shown in Theorem 2, because our algorithm can significantly prune the edges that have small upper bounds. In our experiments, we will show that Algorithm 1 is efficient in practice.

IV. An index-based solution

In this section, we develop a novel index structure, called ESDIndex, to efficiently support the top-k edge structural diversity search. Based on ESDIndex, we are able to process the top-k structural diversity search in near-optimal time $O((\alpha\gamma(n)+\log m)\alpha m)$, where $\gamma(n)$ is the inverse Ackermann function and it is smaller than 5 in practice [16]. Below, we first introduce our index structure, followed by the proposed index construction algorithms.

A. The ESDIndex structure

Given a graph $G = (V, E)$ and an edge $(u, v) \in E$, we let c be the size of a connected component in $G_{N(uv)}$. Let $C_{uv} \triangleq \{c | \textit{a connected component in } G_{N(uv)} \textit{ has size } c\}$ be the set of all component sizes c's in the ego-network $G_{N(uv)}$. Further, we define $C \triangleq \bigcup_{(u,v)\in E} C_{uv}$.

The basic idea of the ESDIndex structure is to maintain a sorted list of the edges for each size $c \in C$. Specifically, the ESDIndex structure, denoted by H, contains $|C|$ sorted lists. Let E_c be the set of all edges whose edge ego-networks have at least one connected component with size no less than c. For each $c \in C$, we compute the structural diversity for every edge $(u, v) \in E_c$ on the basis of the threshold c. Then, for each $c \in C$, we can obtain a sorted list $H(c)$ by sorting the edges in E_c in an non-increasing order based on their structural diversities. Note that for an edge (u, v), if the maximum size of the connected components in $G_{N(uv)}$ is smaller than c, then (u, v) cannot be included in $H(c)$. We make use of a self-balance binary search tree structure [16] to maintain $H(c)$ for each $c \in C$. The following example illustrates the ESDIndex structure.

c1 = 1	
edge	score
(b,c)	2
(b,e)	2
(c,e)	2
…	…
(q,w)	1

(a) $H(1)$

c2 = 2	
edge	score
(f,g)	2
(h,i)	2
(j,k)	2
…	…
(q,w)	1

(b) $H(2)$

c3 = 4	
edge	score
(j,k)	1
(j,u)	1
(j,v)	1
…	…
(k,q)	1

(c) $H(4)$

c4 = 5	
edge	score
(u,q)	1
(v,p)	1
(p,q)	1

(d) $H(5)$

Fig. 2. The ESDIndex structure of G in Fig. 1(a)

Example 4: Consider the graph G in Fig. 1(a). It is easy to derive that $C = \{1, 2, 4, 5\}$. Therefore, we have four sorted lists $H(1)$, $H(2)$, $H(4)$, and $H(5)$ as shown in Fig. 2. Clearly, $H(1)$ contains all edges, because all edges' ego-networks have at least one connected component with size no less than 1. The set of edges $\{(a, b), (a, c), (b, c), (b, d), (b, e), (c, e), (c, g)\}$ are not contained in $H(2)$, since the size of the maximum connected component in these edges' ego-networks is smaller than 2. Similarly, we can see that $H(4)$ contains 15 edges which are $\{(j, k), (j, u), (j, v), (k, u), (k, v), (u, v), (u, p), (u, q), (v, p), (v, q), (p, q), (j, p), (j, q), (k, p), (k, q)\}$ and $H(5)$ contains three edges, namely, $(u, p), (u, q), (p, q)$. Note that the edges in $H(c)$ for $c \in C$ are sorted in a non-increasing order based on their structural diversities. For example, when $c = 5$, we compute the structural diversities for all edges in G based on the threshold $c = 5$. We can easily check that under this case, the structural diversities of all edges in $\{(u, p), (u, q), (p, q)\}$ are equal to 1 as illustrated in Fig. 2(d).

The space overhead of ESDIndex. We analyze the space usage of the ESDIndex in the following theorem.

Theorem 3: Given a graph G, the worst-case space complexity of the ESDIndex is $O(\alpha m)$ where α is the arboricity of G.

Proof: For each edge (u, v), the size of the maximum connected component in $G_{N(uv)}$ is no larger than $\delta_{uv} = |N(u) \cap N(v)|$. Thus, an edge (u, v) is included in at most δ_{uv} sorted lists ($H(c)$ for $1 \leq c \leq \delta_{uv}$). As a result, the total size of the ESDIndex is bounded by $O(\sum_{(u,v)\in E} \delta_{uv}) \leq O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) = O(\alpha m)$. □

Note that the arboricity of a real-world graph is often very small [6], [7], thus the ESDIndex is space-efficient in practice, which is confirmed in our experiments.

B. Query processing algorithm

Equipped with the ESDIndex, we can easily process the top-k structural diversity query. In particular, for a given top-k structural diversity query (k, τ), we first find the sorted list $H(c^*)$ ($c^* \in C$) where c^* is the smallest integer in C such that $c^* \geq \tau$, and then report the top-k edges in $H(c^*)$. Below, we analyze the correctness of this query processing algorithm.

Theorem 4: Given two integers k and τ, the query processing algorithm correctly outputs the top-k edges with the highest structural diversities.

Proof: We consider two cases: (1) $\tau = c^*$ and (2) $\tau < c^*$. For the first case, the top-k results in $H(c^*)$ are clearly the

answers. For the second case, we let c' be the largest integer in C such that $c' < \tau$. Then, in this case, we have $c' < \tau < c^*$. Recall that by our definition, there is no connected component in the edge ego-networks that has a size τ. Since c^* is the smallest integer in C with $c^* > \tau$, the structural diversity of each edge in G computed by using a threshold τ equals the structural diversity calculated by using c^*. As a consequence, the top-k results in $H(c^*)$ are correct. □

Example 5: Reconsider the graph in Fig. 1(a). Suppose that $k = 3$ and $\tau = 2$. The query processing algorithm first finds the index structure $H(2)$ because $2 \in C$ is the smallest integer that is no less than τ. As shown in Fig. 2(b), (f, g), (h, i) and (j, k) have the highest structural diversities ($\mathsf{score}(f, g) = \mathsf{score}(h, i) = \mathsf{score}(j, k) = 2$) in $H(2)$, thus $\{(f, g), (h, i), (j, k)\}$ are the answers. □

The time complexity of the query processing algorithm is analyzed as follows.

Theorem 5: Given two integers k and τ, the query processing algorithm can answer the top-k structural diversity query in $O(k \log m + \log n)$ time.

Proof: First, we can make use of a binary search procedure to find the c^* in $O(\log n)$ time, because $|C|$ is obviously bounded by n. Second, since $H(c)$ is a self-balance binary search tree, finding the top-k results in $H(c)$ can be done in $O(k \log m)$. Therefore, the total time cost of the algorithm is $O(k \log m + \log n)$. □

Note that any top-k algorithm must consume $O(k)$ time to output the results, thus our query processing algorithm is near optimal (only with a $\log m$ factor).

C. Index construction: A basic algorithm

To construct the ESDIndex, a basic solution is to apply a BFS algorithm to compute the connected components in $G_{N(uv)}$ for each edge $(u, v) \in E$. The pseudo code of this basic solution is shown in Algorithm 2. First, for each edge (u, v), the algorithm computes the set of connected components in $G_{N(uv)}$ by using BFS (lines 1-2). Let C_{uv} be the set of all connected component sizes in $G_{N(uv)}$, and $C = \bigcup_{(u,v)\in E} C_{uv}$ (lines 3-4). Then, the algorithm initializes a self-balance binary search tree $H(c)$ for each $c \in C$ (line 5). After that, the algorithm computes the structural diversity for each edge (u, v), and inserts the edge (u, v) into $H(c)$ for $c \leq c_{\max}$, where $c_{\max}$ denotes the size of the maximum connected component in $G_{N(uv)}$ (lines 6-15). The correctness of Algorithm 2 can be guaranteed by the definition of the ESDIndex. Below, we analyze the time complexity of Algorithm 2.

Algorithm 2: ESDIndex

```
   Input: G = (V, E)
   Output: The ESDIndex H
1  for each (u, v) ∈ E do
2      Compute the set of connected components in G_{N(uv)} by
         BFS;
3      C_uv ← the set of all component sizes in G_{N(uv)};
4  C ← ⋃_{(u,v)∈E} C_uv;
5  H(c) ← ∅ for each c ∈ C;
6  for each (u, v) ∈ E do
7      Let c_i ∈ C_uv with c_1 ≤ c_2 ≤ ··· ≤ c_max;
8      x_i ← the number of components in G_{N(uv)} with size c_i;
9      s ← 0;
10     for i = max to 1 do
11         s ← s + x_i; s_i ← s;
12     for c = 1 to c_max do
13         if c ∈ C then
14             Find the smallest i such that c ≤ c_i;
15             H(c).insert((u, v), s_i);
16 return H;
```

Theorem 6: The worst-case time complexity of Algorithm 2 is $O((d_{\max} + \log m)\alpha m)$, where $d_{\max}$ and α denote the maximum degree and the arboricity of G respectively.

Proof: First, for each edge (u, v), the size of $G_{N(uv)}$ is bounded by $O(\delta_{uv}^2)$, where $\delta_{uv} = |N(u) \cap N(v)|$. Thus, lines 1-3 of Algorithm 2 take at most $O(\sum_{(u,v)\in E} \delta_{uv}^2)$. Since $\delta_{uv} \leq \min\{d(u), d(v)\}$ and $O(\sum_{(u,v)\in E} \min\{d(u), d(v)\}) = O(\alpha m)$, $O(\sum_{(u,v)\in E} \delta_{uv}^2)$ can be bounded by $O(\alpha d_{\max} m)$. Second, for each edge (u, v), both max and $c_{\max}$ in line 7 can be bounded by δ_{uv}. Note that inserting an edge (u, v) into a self-balance binary search tree $H(c)$ takes at most $O(\log m)$ time [16] (line 15). Thus, the total time cost taken in lines 6-15 is $O(\sum_{(u,v)\in E} \delta_{uv} \log m)$ which is bounded by $O(\alpha m \log m)$. As a result, the time complexity of Algorithm 2 is $O((d_{\max} + \log m)\alpha m)$. □

D. Index construction: An improved algorithm

As discussed above, the basic index construction algorithm may be very costly for large graphs, because the maximum degrees of many real-life large graphs are typically very large. Here we develop an improved algorithm which reduces the time complexity over the basic algorithm from $O((d_{\max} + \log m)\alpha m)$ to $O((\alpha\gamma(n) + \log m)\alpha m)$, where the arboricity α is typically much smaller than the maximum degree $d_{\max}$ and $\gamma(n) < 5$. The improved algorithm is based on the following crucial observation.

Observation 1: Given an edge (u, v) and its ego-network $G_{N(uv)}$, the four vertices $\{u, v, w_1, w_2\}$ form a 4-clique in G if and only if $(w_1, w_2) \in G_{N(uv)}$.

Proof: First, since $(w_1, w_2) \in G_{N(uv)}$, both w_1 and w_2 are common neighbors of u and v. Note that there is an edge connecting w_1 and w_2 in G, thus the four vertices $\{u, v, w_1, w_2\}$ form a 4-clique in G. Second, if the vertices $\{u, v, w_1, w_2\}$ form a 4-clique, both w_1 and w_2 must be contained in $G_{N(uv)}$. Since w_1 and w_2 are connected, we have $(w_1, w_2) \in G_{N(uv)}$. □

According to the Observation 1, we can make use of a 4-clique enumeration algorithm to construct the connected components for all edge ego-networks. Specifically, we first orientate the graph G in terms of the degree ordering. Denote by $\vec{G}$ the directed graph generated by the degree ordering. Then, we enumerate the 4-cliques on $\vec{G}$. When a 4-clique $\{u, v, w_1, w_2\}$ is found, we maintain the connecting relationships in the edge ego-networks of six edges in this 4-clique based on the Observation 1. To make this procedure efficient, we use a disjoint-set data structure (a Union-Find structure) to maintain each connected component in $G_{N(uv)}$. After we obtain the components for each edge ego-network, we can apply a similar method used in Algorithm 2 to construct the ESDIndex. The pseudo code of the improved index construction algorithm is shown in Algorithm 3.

Algorithm 3: ESDIndex+

```
Input: G = (V, E)
Output: The ESDIndex H
1  for (u, v) in G do
2      N(uv) ← N(u) ∩ N(v); M_uv ← ∅;
3      for each w ∈ N(uv) do
4          M_uv[w].root ← w; M_uv[w].count ← 1;
5  G⃗ = (V, E⃗) ← a directed graph generated by degree ordering;
6  for u ∈ V do
7      for v ∈ N+(u) do
8          G⃗_N(uv) ← a subgraph induced by N+(u) ∩ N+(v);
9          for each edge (w1, w2) ∈ G⃗_N(uv) do
10             M_uv.Union(u, v, w1, w2);
11             M_uw1.Union(u, w1, v, w2);
12             M_uw2.Union(u, w2, v, w1);
13             M_vw1.Union(v, w1, u, w2);
14             M_vw2.Union(v, w2, u, w1);
15             M_w1w2.Union(w1, w2, u, v);
16 C ← ∅;
17 for each (u, v) ∈ E do
18     C_uv ← ∅;
19     for each w ∈ N(uv) do
20         if M_uv[w].root = w then
21             C_uv ← C_uv ∪ {M_uv[w].count};
22 C ← ⋃_{(u,v)∈E} C_uv;
23 Using lines 5-15 of Algorithm 2 to build the ESDIndex H;
24 return H;
25 Procedure Find(M_uv, w)
26 while M_uv[w].root ≠ w do
27     w' ← M_uv[w].root; M_uv[w].root ← M_uv[w'].root;
       w ← w';
28 return w;
29 Procedure Union(u, v, w1, w2)
30 r1 ← Find(M_uv, w1); r2 ← Find(M_uv, w2);
31 if r1 ≠ r2 then
32     Merge the two sets with roots r1 and r2;
33     Let r be the root of the merged set;
34     M_uv[w1].root = r; M_uv[w2].root = r;
35     M_uv[r].count = M_uv[r1].count + M_uv[r2].count;
```

Algorithm 3 works as follows. First, it creates a disjoint-set structure M_{uv} for each edge (u, v). In M_{uv}, each vertex $w \in N(uv)$ ($N(uv) = N(u) \cap N(v)$) is initialized as a set, and a variable '*count*' is used to record the size of each set (lines 1-4). Then, in lines 5-15, the algorithm enumerates all 4-cliques in the directed graph $\vec{G}$. Upon finding a 4-clique $\{u, v, w_1, w_2\}$, for each edge (x, y) in the 4-clique, the algorithm merges the sets of the other two vertices in the disjoint-set structure M_{xy} using a standard Union operator (lines 10-15) because the other two vertices should be contained in the same connected component. After searching all 4-cliques in G, for each edge (u, v), all the connected components in $G_{N(uv)}$ are maintained in M_{uv}. Note that by Algorithm 3, each 4-clique is only enumerated once, thus it is more efficient than Algorithm 2, which needs to explore a 4-clique six times. Subsequently, the algorithm records the sizes of the connected components in all edges' ego-networks based on the variable '*count*' of all sets in disjoint-set structures (lines 16-22). Finally, the algorithm applies a similar procedure to construct the ESDIndex as used in Algorithm 2 (line 23). Below, we analyze the time complexity of Algorithm 3.

Theorem 7: The worst-case time complexity of Algorithm 3 is $O((\alpha\gamma(n) + \log m)\alpha m)$, where α is the arboricity of the graph and $\gamma(n)$ is the inverse Ackermann function ($\gamma(n) < 5$).

Proof: First, in lines 1-4, Algorithm 3 takes $O(\sum_{(u,v)\in E} \min\{d_u, d_v\}) = O(\alpha m)$ time to initialize the disjoint-set structures for all edges. Second, in lines 6-15, the algorithm needs to enumerate each 4-clique once which takes $O(\alpha^2 m)$ time [5]. Note that when enumerating a 4-clique, the algorithm requires to perform six Union operators. The time overhead of the Union operator can be bounded by $\gamma(n)$ which is the inverse Ackermann function and it is smaller than 5 in practice [16]. Hence, the total cost taken in lines 6-15 can be bounded by $O(\alpha^2\gamma(n)m)$. Third, in lines 16-23, the total time cost is bounded by $O(\alpha m \log m)$ as analyzed in Theorem 6. Putting it all together, the time complexity of Algorithm 3 is $O((\alpha\gamma(n) + \log m)\alpha m)$. □

E. Parallel implementation

Here we introduce a parallel implementation of the improved index construction algorithm. Specifically, in line 1 of Algorithm 3, we can process the edges in parallel, because each M_{uv} can be initialized independently in (u, v)'s ego-network. To enumerate 4-cliques (lines 6-15 of Algorithm 3), a simple parallel solution is to process each vertex $u \in V$ in parallel in line 6 of Algorithm 3. However, such a simple solution may be inefficient, because the out-degrees of the vertices typically exhibit a skew distribution, resulting in the workloads of different threads are unbalanced. A better solution is to enumerate 4-cliques for each directed edge in parallel. This is because the distribution of the number of common outgoing neighbors of the directed edges is typically not very skew, thus improving the degree of parallelism of the algorithm. In the experiments, we adopt such an edge-parallel approach in our parallel implementation. Additionally, in line 17 and line 23 of Algorithm 3, we are also able to process the edges in parallel. In the experiments, we will show that our parallel implementation can achieve a very good speedup ratio on real-life graphs.

V. Index Maintenance on Dynamic Graphs

In this section, we develop efficient solutions to maintain the ESDIndex when the graph is updated. We focus mainly on the edge insertion and deletion, as vertex insertion and deletion can be treated as a series of edge insertions and deletions.

A. Handling edge insertion

To maintain the ESDIndex after inserting an edge, the key is to maintain the structural diversities of the edges in G. Let $\hat{G}_{N(uv)}$ be a subgraph induced by the vertices $N(uv) \cup \{u, v\}$. The following key observation shows that we only need to update the structural diversities of the edges in the small subgraph $\hat{G}_{N(uv)}$ after inserting an edge (u, v).

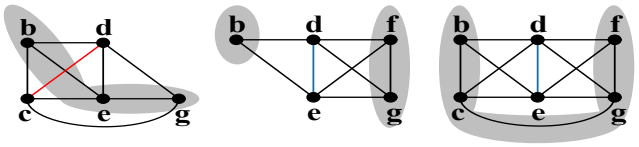
Observation 2: After inserting an edge (u, v) into G, the structural diversities of the edges in $\hat{G}_{N(uv)}$ need to be updated, and the structural diversities of the edges that are not in $\hat{G}_{N(uv)}$ remain unchanged.

Proof: Clearly, for any edge $(w_1, w_2) \in \hat{G}_{N(uv)}$, we need to update the ego-network of (w_1, w_2), thus its structural

Algorithm 4: Insertion

Input: $G = (V, E)$, H, M, $C_{w_1w_2}$ for each edge (w_1, w_2), and an inserted edge (u, v)
Output: the updated H, M, and $C_{w_1w_2}$ for each edge (w_1, w_2)

1 Insert (u, v) in G;
2 $N(uv) \leftarrow N(u) \cap N(v)$; $\hat{G}_{N(uv)} \leftarrow (\{u, v\}, \{(u, v)\})$;
3 $flag(w) \leftarrow 0$ for each $w \in V$; $M_{uv} \leftarrow \emptyset$;
4 **for** *each* $w \in N(uv)$ **do**
5 Add vertex w into $\hat{G}_{N(uv)}$; $flag(w) \leftarrow 1$;
6 Add edges (u, w) and (v, w) into $\hat{G}_{N(uv)}$;
7 $M_{uv}[w].root \leftarrow w$; $M_{uv}[w].count \leftarrow 1$;
8 $M_{uw}[v].root \leftarrow v$; $M_{uw}[v].count \leftarrow 1$;
9 $M_{vw}[u].root \leftarrow u$; $M_{vw}[u].count \leftarrow 1$;
10 **for** *each* $w_1 \in N(uv)$ **do**
11 **for** *each* $w_2 \in N^+(w_1)$ **do**
12 **if** $flag(w_2) = 1$ **then**
13 Add edge (w_1, w_2) into $\hat{G}_{N(uv)}$;
14 M_{uv}.Union(u, v, w_1, w_2);
15 $M_{w_1w_2}$.Union(w_1, w_2, u, v);
16 M_{uw_1}.Union(u, w_1, v, w_2);
17 M_{vw_1}.Union(v, w_1, u, w_2);
18 M_{uw_2}.Union(u, w_2, v, w_1);
19 M_{vw_2}.Union(v, w_2, u, w_1);
20 **for** *each edge* $(w_1, w_2) \in \hat{G}_{N(uv)}$ **do**
21 Using lines 18-21 of Algorithm 3 to update $C_{w_1w_2}$;
22 Using lines 7-15 of Algorithm 2 to update H;



(a) $\hat{G}_{N(cd)}$ (b) $G_{N(de)}$, before insert (c) $G_{N(de)}$, after insert

Fig. 3. Illustration of the edge insertion algorithm

diversity also requires to be updated. For any edge $(w_1, w_2) \notin \hat{G}_{N(uv)}$, w_1 and w_2 cannot form a triangle or a 4-clique with u and v, thus its ego-network keeps unchanged. □

Based on the Observation 2, we propose a *local-update* algorithm to handle the edge insertion in Algorithm 4. Note that in Algorithm 4, to efficiently maintain the ESDIndex H, we also need to maintain the disjoint-set structure $M_{w_1w_2}$ and the component size set $C_{w_1w_2}$ for each edge $(w_1, w_2) \in G$. First, the algorithm creates a new disjoint-set structure M_{uv} for the inserted edge (u, v) (line 3) and constructs the subgraph $\hat{G}_{N(uv)}$ online (lines 2-6 and line 13). For each vertex $w \in N(uv) = N(u) \cap N(v)$, the algorithm initializes a set for every edge in the triangle $\{u, v, w\}$ (lines 7-9), denoting a connected component with an isolated vertex. After that, the algorithm iteratively processes each edge (w_1, w_2) in (u, v)'s ego-network $G_{N(uv)}$ (lines 10-19). Note that in each iteration, the vertices w_1, w_2, u, and v form a 4-clique. For each edge (x, y) in the 4-clique, the algorithm merges the sets of the other two vertices in M_{xy} using a standard Union operator (lines 14-19), because the other two vertices are contained in the same connected component after inserting (u, v). Finally, equipped with the updated M_{uv}'s, the algorithm iteratively updates both $C_{w_1w_2}$ and H for each edge $(w_1, w_2) \in \hat{G}_{N(uv)}$ using similar approaches as used in Algorithm 3 and Algorithm 2. The following example illustrates the insertion algorithm.

Example 6: Reconsider the graph G shown in Fig. 1(a). Suppose that we insert an edge (c, d) into G. The subgraph $\hat{G}_{N(cd)}$ is shown in Fig. 3(a). By Observation 2, the ego-networks of all edges in $\hat{G}_{N(cd)}$ need to be updated. Take an edge $(d, e) \in \hat{G}_{N(cd)}$ as an example. The ego-network of (d, e) before inserting (c, d) is depicted in Fig. 3(b) (shaded area), where the vertices f and g form a connected component, and the isolated vertex b forms another component. After inserting (c, d), we can see that the vertices $\{b, c, d, e\}$ in G is a 4-clique, thus both b and c must be merged into the same component of (d, e)'s ego-network. Similarly, c and g are also in the same component. Therefore, the ego-network of (d, e) contains only one connected component after inserting (c, d) as shown in Fig. 3(c) (shaded area). □

Below, we analyze the time complexity of Algorithm 4. Let G_{uv} be a subgraph induced by the set of vertices $\bigcup_{w \in N(uv)} N(w) \cup \{u, v\}$ and m_{uv} be the number of edges in G_{uv}. Further, we let m' be the number of edges in $\hat{G}_{N(uv)}$.

Theorem 8: The worst-case time complexity of Algorithm 4 to process an insertion of (u, v) is $O(\alpha^2\gamma(n)m' + (\alpha + \log m)m_{uv})$ time.

Proof: First, it is easy to see that lines 4-9 of Algorithm 4 take $O(\min\{d_u, d_v\})$ time. Second, in lines 10-19, the algorithm needs to enumerate the 4-cliques in $\hat{G}_{N(uv)}$. For each edge (w_1, w_2) in those 4-cliques, the algorithm needs to update $M_{w_1w_2}$ which takes $O(\gamma(n))$ time. Since the number of 4-cliques is bounded by $O(\alpha^2 m')$, the total time cost taken in lines 10-19 is $O(\alpha^2\gamma(n)m')$. Finally, in lines 20-22, the algorithm needs to traverse the ego-network for each edge $(x, y) \in \hat{G}_{N(uv)}$ to maintain C_{xy} and the index H, which consumes at most $O(\alpha + \log m)m_{uv})$ time. □

Note that the time complexity of Insertion relies mainly on the size of a small subgraph G_{uv} (i.e., m_{uv}), rather than the size of the original graph. Since m_{uv} is typically not very large, our algorithm is efficient in practice, as confirmed in our experiments.

B. Handling edge deletion

Here we consider the case of deleting an edge (u, v) from G. Let $\tilde{G}_{N(uv)} = \hat{G}_{N(uv)} \setminus \{(u, v)\}$. Similar to the insertion case, we only need to update the structural diversities of the edges in $\tilde{G}_{N(uv)}$ when deleting (u, v). Specifically, we have the following observation.

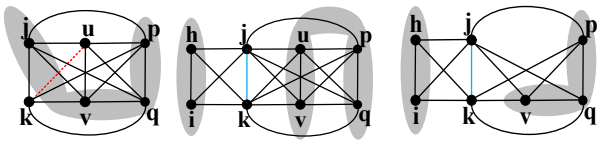
Observation 3: After deleting an edge (u, v) from G, the structural diversities of the edges in $\tilde{G}_{N(uv)}$ need to be updated, while for any edge $(x, y) \notin \tilde{G}_{N(uv)}$, its structural diversity keeps unchanged.

Based on the Observation 3, we devise a *local-update* algorithm to handle the edge deletion in Algorithm 5. First, for each $w \in N(uv)$, if it forms an isolated component in (u, v)'s ego-network, the algorithm deletes the set of v (u) in M_{uw} (M_{vw}), because v (u) is no longer a common neighbor between u (v) and w (lines 6-9). Then, for a 4-clique in $\hat{G}_{N(uv)}$, the algorithm needs to update M_{xy} for each edge (x, y) in that 4-clique except M_{uv} (lines 10-18). Note that in the Update procedure (lines 24-35), we only need to explore the vertices in the connected component containing u or v which is maintained by a set S. Subsequently, the algorithm iteratively updates $C_{w_1w_2}$ and H for each edge $(w_1, w_2) \in \tilde{G}_{N(uv)}$ using similar methods as used in Algorithm 3 and Algorithm 2 (lines 19-21). Finally, the algorithm deletes M_{uv}

Algorithm 5: Deletion

Input: $G=(V,E)$, H, M, $C_{w_1w_2}$ for each edge (w_1,w_2), and a deleted edge (u,v)
Output: the updated H, M, and $C_{w_1w_2}$ for each edge (w_1,w_2)

1 $N(uv) \leftarrow N(u) \cap N(v)$; $\tilde{G}_{N(uv)} \leftarrow \emptyset$;
2 $flag(w) \leftarrow 0$ for each $w \in V$;
3 **for** *each* $w \in N(uv)$ **do**
4 Add vertex w into $\tilde{G}_{N(uv)}$; $flag(w) \leftarrow 1$;
5 Add edges (u,w) and (v,w) into $\tilde{G}_{N(uv)}$;
6 **if** $M_{uw}[v].root = v$ *and* $M_{uw}[v].count = 1$ **then**
7 Delete $M_{uw}[v]$;
8 **if** $M_{vw}[u].root = u$ *and* $M_{vw}[u].count = 1$ **then**
9 Delete $M_{vw}[u]$;
10 **for** *each* $w_1 \in N(uv)$ **do**
11 **for** *each* $w_2 \in N^+(w_1)$ **do**
12 **if** $flag(w_2) = 1$ **then**
13 Add edge (w_1,w_2) into $\tilde{G}_{N(uv)}$;
14 $M_{w_1w_2}$.Update(u,w_1,w_2,u,v);
15 M_{uw_1}.Update(v,u,w_1,u,v);
16 M_{vw_1}.Update(u,v,w_1,u,v);
17 M_{uw_2}.Update(v,u,w_2,u,v);
18 M_{vw_2}.Update(u,v,w_2,u,v);
19 **for** *each edge* $(w_1,w_2) \in \tilde{G}_{N(uv)}$ **do**
20 Using lines 18-21 of Algorithm 3 to update $C_{w_1w_2}$;
21 Using lines 7-15 of Algorithm 2 to update H;
22 Delete M_{uv}; Delete $\tilde{G}_{N(uv)}$;
23 Delete (u,v) in H and G;
24 **Procedure** Update(z,w_1,w_2,u,v)
25 **if** $M_{w_1w_2}$ *has already been updated* **then return**;
26 Delete (u,v) in G; $S \leftarrow \emptyset$; $T_{w_1w_2} \leftarrow \emptyset$; $r \leftarrow M_{w_1w_2}[z].root$;
27 **for** *each* $w \in N(w_1w_2)$ **do**
28 **if** $M_{w_1w_2}[w].root = r$ **then**
29 $S \leftarrow S \cup \{w\}$;
30 $T_{w_1w_2}[w].root \leftarrow w$; $T_{w_1w_2}[w].count \leftarrow 1$;
31 **for** *each* $x \in S$ **do**
32 **for** *each* $y \in S$ *and* $y \in N^+(x)$ **do**
33 $T_{w_1w_2}$.Union(w_1,w_2,x,y);
34 $M_{w_1w_2} \leftarrow T_{w_1w_2}$;
35 Insert (u,v) in G;



(a) $\tilde{G}_{N(uk)}$ (b) $G_{N(jk)}$, before delete (c) $G_{N(jk)}$, after delete

Fig. 4. Illustration of the edge deletion algorithm

and (u,v) in the ESDIndex H (lines 22-23). The following example illustrates the key idea of the edge deletion algorithm.

Example 7: Reconsider the graph G in Fig. 1(a). Suppose that we delete an edge (u,k) from G. Then, the subgraph $\tilde{G}_{N(uk)}$ is shown in Fig. 4(a). According to the Observation 3, the ego-network of each edge in $\tilde{G}_{N(uk)}$ needs to be updated. Let us consider the edge (j,k) as an example. The ego-network of (j,k) before removing (u,k) is shown in Fig. 4(b) (shaded area). After deleting (u,k), the 4-clique $\{j,k,u,v\}$ is broken. As a result, the algorithm re-constructs $G_{N(jk)}$ and re-computes the structural diversity of (j,k). The updated ego-network of (j,k) is shown in Fig. 4(c) (the shaded area). As can be seen, the algorithm needs to create a new list $H(3)$, because there is a connected component with size $c=3$. After that, the algorithm inserts (j,k) into $H(3)$. □

The time complexity of Algorithm 5 is analyzed in the following theorem.

Theorem 9: The worst-case time complexity of Algorithm 5 for handling a deletion of (u,v) is $O(\alpha^2 m' + (\alpha\gamma(n) + \log m)m_{uv})$.

Proof: Clearly, the most time-consuming steps of Algorithm 5 are lines 10-18 and lines 19-21. In lines 10-18, the algorithm takes $O(\alpha^2 m')$ to list all 4-cliques in (u,v)'s ego-network. For each edge (w_1,w_2) in the those 4-cliques, the algorithm only updates $M_{w_1w_2}$ once. To update $M_{w_1w_2}$, the algorithm needs to traverse the ego-network of (w_1,w_2) and performs the Union operator. Thus, the total cost can be bounded by $O(\sum_{(w_1,w_2)\in G_{N(uv)}}(\min\{d_{w_1},d_{w_2}\}\gamma(n))) \leq O(\alpha\gamma(n)m_{uv})$. By a similar analysis in Theorem 8, the time cost taken in lines 19-21 is $O((\alpha+\log m)m_{uv})$. □

TABLE I
DATASETS

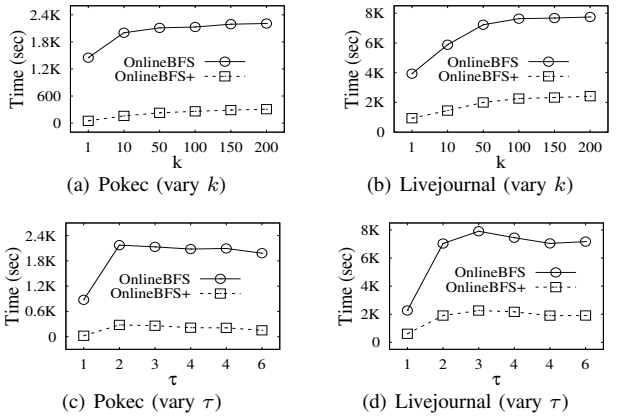
Dataset	n	m	$d_{\max}$	δ
Youtube	1,134,890	2,987,624	28,754	51
WikiTalk	2,394,385	4,659,565	100,029	131
DBLP	1,843,617	8,350,260	2,213	279
Pokec	1,632,803	22,301,964	14,854	47
LiveJournal	3,997,962	34,681,189	14,815	360

VI. EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the efficiency and effectiveness of the proposed algorithms. We implement the *dequeue-twice* online search framework (Algorithm 1) with two upper-bounding rules. Specifically, we refer to the *dequeue-twice* framework with min-degree and common-neighbor upper bounds as OnlineBFS and OnlineBFS+, respectively. Since there is no existing algorithm that can be used to compute the top-k edge structural diversities, we make use of our online search algorithms as baselines. For comparison, we implement the index-based algorithm and refer to it as IndexSearch. To construct the ESDIndex, we implement Algorithm 2 and Algorithm 3, denoted by ESDIndex and ESDIndex+, respectively. A parallel version of ESDIndex+, denoted by PESDIndex+, is also implemented. In addition, we implement Insertion (Algorithm 4) and Deletion (Algorithm 5) to maintain the ESDIndex in dynamic graphs. All algorithms are implemented in C++. All experiments are conducted on a PC with 2.40GHz Intel Xeon E52620 (6-core) CPU and 32GB memory running Ubuntu 16.04.1 LTS (64-bit). In all experiments, both the graph and the ESDIndex are resident in the main memory.

Datasets. We make use of five large real-life networks in the experiments. The detailed statistics of the datasets are summarized in Table I. In Table I, $d_{\max}$ and δ denote the maximum degree and the degeneracy of the graph, respectively. Youtube, Pokec, and LiveJournal are online social networks, WikiTalk is a communication network, and DBLP is a scientific collaboration network. All these datasets are downloaded from http://snap.stanford.edu/data/index.html.

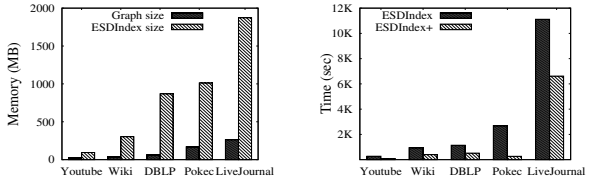
Parameters. There are two parameters in our algorithms: τ and k. The parameter τ is selected from the interval $[1,6]$ with a default value of $\tau = 3$; k is chosen from the set $\{1,10,50,100,150,200\}$ with a default value of $k = 100$.



(a) Pokec (vary k) (b) Livejournal (vary k)

(c) Pokec (vary τ) (d) Livejournal (vary τ)

Fig. 5. Runtime of OnlineBFS and OnlineBFS+ with varying parameters



(a) Memory overhead (b) Indexing time

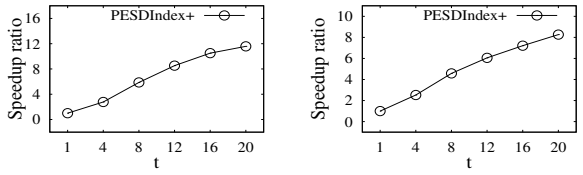
Fig. 6. Evaluation of the ESDIndex

Unless otherwise specified, the values of the other parameters are set to their default values when varying a parameter.

A. *Efficiency testing*

Exp-1: Comparison between OnlineBFS **and** OnlineBFS+**.** Fig. 5 shows the runtime of OnlineBFS and OnlineBFS+ with varying k and τ on Pokec and LiveJournal datasets. Similar results can also be observed on the other datasets. As expected, the runtime of both OnlineBFS and OnlineBFS+ increases as k increases. In general, both OnlineBFS and OnlineBFS+ achieve the minimum runtime at $\tau = 1$, and then the runtime of both OnlineBFS and OnlineBFS+ decreases when τ increases. This is because the structural diversities of many edges are 0 for a large τ, thus reducing the computational costs. As can be seen, OnlineBFS+ is significantly faster than OnlineBFS with all parameter settings. For example, on Pokec, OnlineBFS+ takes 261.7 seconds, while OnlineBFS consumes 2120.5 seconds to output the top-100 results. The reason is that the common-neighbor upper bound is tighter than the min-degree upper bound, thus it is more effective to prune the search space. These results indicate that the pruning benefits of OnlineBFS+ dominate the computational costs for the common-neighbor upper bounds.

Exp-2: Evaluation of the ESDIndex**.** We build the index structure for five datasets using both ESDIndex and ESDIndex+. Fig. 6(a) reports the index size and the graph size. As can be seen, the index size is generally 4-8 times larger than the graph size, which confirms the theoretical analysis in Section IV-A. From Fig. 6(b), we can see that ESDIndex+ is 2 to 10 times faster than ESDIndex over all datasets. The reason is as follows. ESDIndex applies BFS to compute edge structural diversities which traverses each 4-clique six times, while ESDIndex+ only needs to explore each 4-clique once. Moreover, we can see that ESDIndex+ performs very well on small-degeneracy graphs, this is because the time complexity



(a) Pokec (vary threads t) (b) LiveJournal (vary threads t)

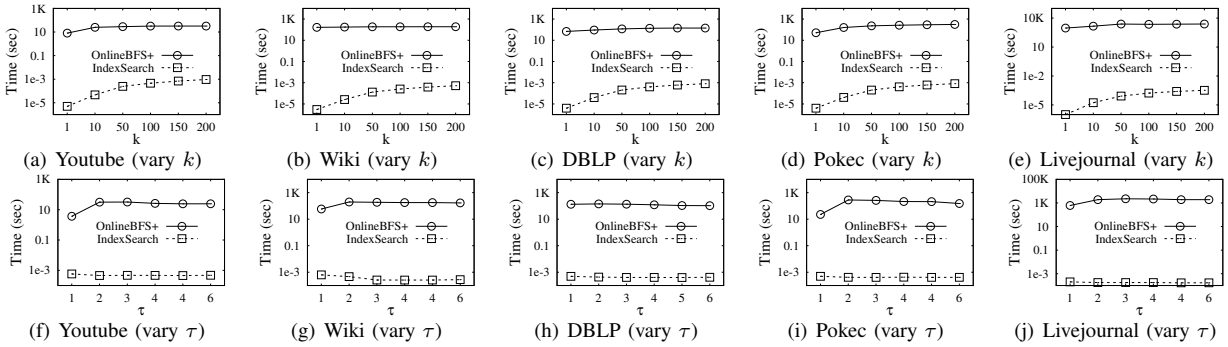
Fig. 7. Speedup ratio of the PESDIndex+ algorithm

of ESDIndex+ depends on the arboricity of the graph which can be well approximated by the degeneracy. For example, on a small-degeneracy graph Pokec, ESDIndex+ only consumes 266.3 seconds to create the index. However, on the same dataset, ESDIndex takes 2694.3 seconds to construct index. These results are consistent with our theoretical analysis shown in Section IV.

Exp-3: Parallel index construction. Here we evaluate the speedup ratio of the parallel index construction algorithm, i.e., PESDIndex+. To this end, we vary the number of threads, t, from 1 to 20 and evaluate the runtime of PESDIndex+ with an increasing t. Fig. 7 reports the results on Pokec and LiveJournal. Similar results can also be observed on the other datasets. From Fig. 7, we can see that PESDIndex+ achieves linear speedup ratios on both Pokec and LiveJournal. For example, when $t = 20$ the speedup ratio of PESDIndex+ is roughly equal to 12 on Pokec. These results indicate that our parallel index construction algorithm is very efficient on real-life graphs.

Exp-4: Comparison between OnlineBFS+ **and** IndexSearch**.** Fig. 8 shows the runtime of OnlineBFS+ and IndexSearch on different datasets with varying parameters. From Figs. 8(a-e), we can see that the runtime of both OnlineBFS+ and IndexSearch increases with increasing k. Generally, on all datasets, IndexSearch is at least four orders of magnitude faster than OnlineBFS+ with all parameter settings. For example, on Pokec, when $k = 100$, IndexSearch takes 0.4 milliseconds and OnlineBFS+ takes 261.7 seconds to output the top-k results. This is because IndexSearch can answer the top-k edge structural diversity query in near-optimal time ($O(k \log m + \log n)$), while OnlineBFS+ needs to compute the structural diversities online which is costly. Similarly, as shown in Figs. 8(f-j), the runtime of IndexSearch is also at least four orders of magnitude faster than OnlineBFS+ with varying τ. Moreover, we can see that the runtime of IndexSearch is robust with respect to (w.r.t.) the parameter τ, because IndexSearch can directly output the top-k results based on the ESDIndex which is independent of τ. These results confirm the theoretical analysis presented in Sec. IV.

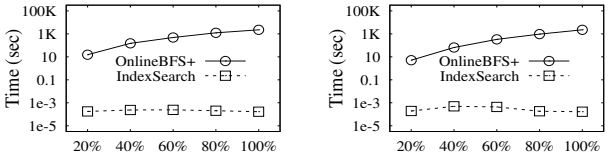
Exp-5: Scalability testings. Here we evaluate the scalability of the proposed algorithms. To this end, we generate four subgraphs for each dataset by randomly picking 20%-80% of the edges (vertices), and evaluate the runtime of OnlineBFS+ and IndexSearch on these subgraphs. Fig. 9 shows the results on LiveJournal, and similar results can also be obtained on the other datasets. We can clearly see that OnlineBFS+ achieves linear scalability. The runtime of OnlineBFS+ increases smoothly as the graph size increases. For the index-based algorithm, IndexSearch, its runtime keeps stable with varying m or n. Again, IndexSearch is four orders



(a) Youtube (vary k) (b) Wiki (vary k) (c) DBLP (vary k) (d) Pokec (vary k) (e) Livejournal (vary k)

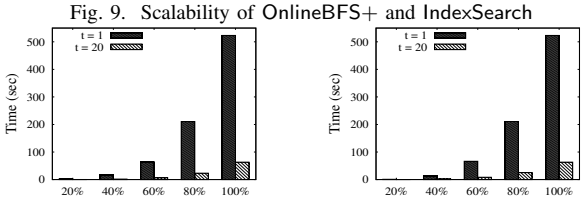
(f) Youtube (vary τ) (g) Wiki (vary τ) (h) DBLP (vary τ) (i) Pokec (vary τ) (j) Livejournal (vary τ)

Fig. 8. Comparison of runtime between OnlineBFS+ and IndexSearch on different datasets



(a) Vary m (b) Vary n

Fig. 9. Scalability of OnlineBFS+ and IndexSearch



(a) Vary m (b) Vary n

Fig. 10. Scalability of PESDIndex+ (with 1 and 20 threads)

of magnitude faster than OnlineBFS+ with all parameter settings, which is consistent with our previous findings. We also evaluate the scalability of our parallel index construction algorithm PESDIndex+ on LiveJournal. The results are shown in Fig. 10. As can be seen, the runtime of PESDIndex+ increases smoothly w.r.t. the graph size. The speedup ratio of PESDIndex+ with 20 threads is between 6 and 9 on all subgraphs. These results are consistent with our previous results.

Exp-6: Index maintenance testings. To evaluate the performance of our index maintenance algorithms, we randomly select 1000 edges for insertion and deletion on each dataset. The average runtime of our Insertion and Deletion algorithms over the 1000 edge insertions and deletions respectively is shown in Fig. 11. As expected, the update time of both Insertion and Deletion increases when the graph size and index size increase. The average insertion time is lower than the average deletion time on all datasets, because Deletion involves a more expensive Update procedure. These results are consistent with our analysis in Section V-B. In addition, we can see that both the insertion and deletion costs are much lower than the index construction cost. For example, on WikiTalk, the insertion and deletion time is 4.6 and 8.9 seconds respectively, while the index construction is 400.7 seconds. These results indicate that our index maintenance algorithms are very efficient on real-life graphs.

B. Effectiveness testing

We conduct two case studies to evaluate the effectiveness of the proposed algorithms.

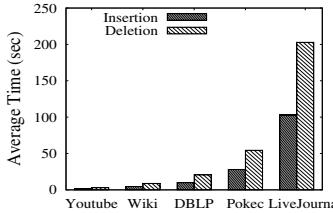
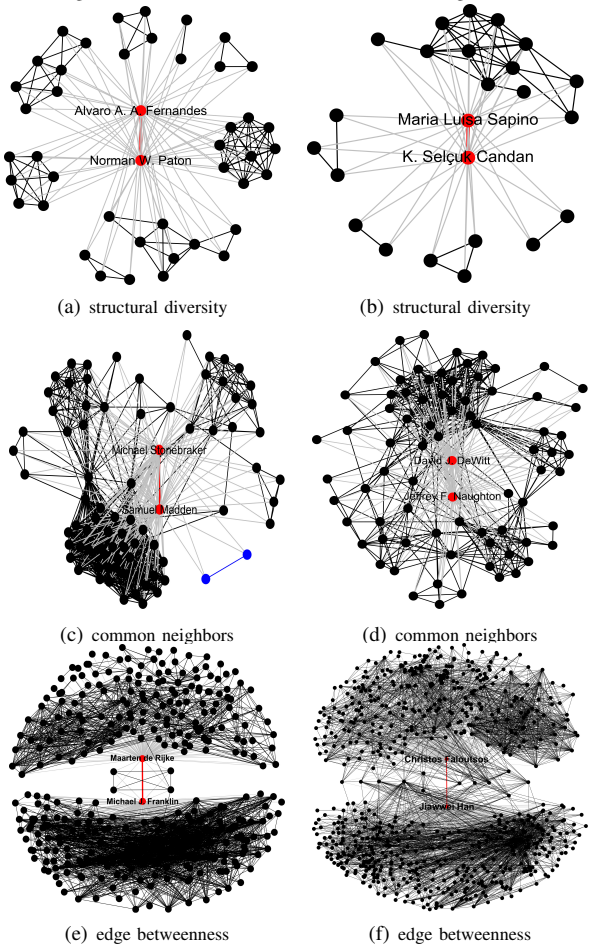


Fig. 11. Runtime of the index maintenance algorithms



(a) structural diversity (b) structural diversity

(c) common neighbors (d) common neighbors

(e) edge betweenness (f) edge betweenness

Fig. 12. Case Study on the DB subgraph of DBLP ($\tau = 2$)

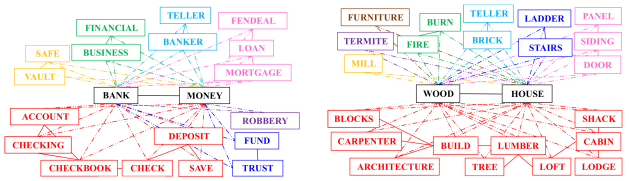
Exp-7: Case Study on DBLP**.** We extract a subgraph, namely DB, from DBLP for case study. DB contains the authors in DBLP who had published at least one paper in the database and data mining related conferences. The DB subgraph con-

tains 37,177 vertices and 131,715 edges. We set the parameter $\tau = 2$ to compute the edge structural diversity, and then use our algorithm to find the top-k edges on DB that have the highest structural diversities. We implement two baselines for comparison: one is based on the common neighbors (CN) and the other is based on the betweenness (BT) of an edge. Specifically, we compute the number of common neighbors (betweenness) for each edge, and then identify the top-k edges on DB with the largest number of common neighbors (betweenness). Figs. 12(a-b) show two edge ego-networks selected from the top-5 results obtained by our algorithm. Figs. 12(c-d) and Figs. 12(e-f) show the ego-networks of the top-2 edges obtained by CN and BT respectively.

Compared to CN, each edge ego-network obtained by our algorithm contains many connected components, while the edge ego-networks generated by CN have at most 2 connected components. Moreover, for our algorithm, the authors of each top-k edge tend to have diverse research interests. Such edges may play the role of a "bridge" in connecting different research communities. However, for CN, the authors of the top-k edges typically work only in the same area. For example, in Fig. 12(a), professor Norman W. Paton works in both database and bio-informatics areas. Similarly, in Fig. 12(b), professor K. Selcuk Candan's research is also cross two areas including database and multimedia. However, as illustrated in Figs. 12(c-d), all the four professors focus mainly on the database area. These results indicate that our algorithm could be used to find the edges that maintain the connections with diverse structural contexts. Such edges may play important roles to promote the interactions between different communities in a network.

As shown in Figs. 12(e-f), the ego-networks obtained by BT exhibit a *barbell* shape. The two authors of the edge obtained by BT link two different tightly-connected subgraphs and they typically share few common neighbors. Moreover, the two authors in Fig. 12(e) (Fig. 12(f)) co-author few papers, indicating a weak relationship. However, for the top-k edges obtained by our algorithm, the two authors of an edge usually have many common neighbors and also co-author many papers, which suggests a strong relationship. For example, in Fig. 12(e), professors Maarten de Rijke and Michael J. Franklin co-author only one paper, and professors Christos Faloutsos and Jiawei Han co-author two papers according to the results in DBLP. However, in Fig. 12(a) and Fig. 12(b), there are 102 and 56 papers that are co-authored by professors Norman W. Paton and Alvaro A. A. Fernandes and by professor K. Selcuk Candan and professor Maria Luisa Sapino, respectively. These results indicate that the semantics of structural diversity is totally different from that of betweenness. In particular, our algorithm can find edges that connect diverse social contexts and often exhibit strong social relationships. However, the edges derived by BT tend to connect two different communities and are likely weak links (the end nodes share few common neighbors).

In addition, we also evaluate the performance of our algorithm when the parameter $\tau \geq 3$. We find that when $\tau \geq 3$, the structural diversity scores of most edges in DBLP are no larger than 3, which results in that the top-k edges may have small structural diversity values. As a consequence, the ego-networks of some of the top-k edges may not reveal diverse structural contexts, which reduces the effectiveness of our algorithm. To avoid the top-k edges having small structural diversity values, we recommend to set τ as a small constant (e.g., $\tau = 2$) for practical applications.



(a) top-1 edge ego-network (b) top-2 edge ego-network

Fig. 13. Case Study on the word association network ($\tau = 2$)

Exp-8: Case Study on a word association network. We use a word association network downloaded from http://w3.usf.edu/FreeAssociation/ for case study. This network contains 5,040 vertices and 55,258 edges, where each vertex denotes a word and each edge connects two words indicating that they are related or strongly associated. We set $\tau = 2$ and $k = 2$, and find the top-k edges with the highest structural diversities. The results are shown in Fig. 13. The highest structural diversity edge is ("bank", "money"). We can see that there are 6 different connected components in its ego-network and each component represents a certain meaning of the two words "bank" and "money". The largest connected component (red part) contains 6 words, and those words are tightly related to the bank-account business. For the connected component colored in pink, the words {"loan", "mortagage", "federal"} are closely related to bank-lending business. For the other components, each component contains at most 2 words, and represents a distinct context of words associated with "bank" and "money". For the edge ("wood", "house"), similar results can be obtained. These results indicate that our top-k edge structural diversity search can be applied to find different meanings for a pair of words, which is a fundamental issue in natural language understanding. Additionally, we also evaluate CN and BT on this word association network. Due to the space limit, we do not show the ego-networks obtained by both CN and BT. We find that the results are consistent with our previous results on DBLP. Specifically, each top-k edge obtained by CN can reflect a strong relationship between two words, but it cannot reveal diverse meanings associated with these two words. Similarly, each top-k edge derived by BT generally connects two tightly-connected subgraphs, but the two nodes of this edge often have few common neighbors, suggesting a weak association relationship.

VII. Related Work

Structural diversity on graphs. The concept of structural diversity of a vertex was first introduced by Ugander et al. [1]. They showed that the user recruitment probability in an online social network (*Facebook*) is determined by the number of connected components in a user's ego-network. Also, several recent studies further confirmed that the structure of a user's ego-network does influence its social behavior in a social network [17], [18]. Instead of studying the structural diversity of a vertex, Dong et al. [3] introduced a concept of structural diversity over a pair of vertices (u, v), in which the structural diversity is defined as the number of connected components of the subgraph induced by the common neighbors of (u, v). They empirically showed that if a pair of vertices (u, v) has a high structural diversity, then (u, v) has a high probability to

be connected. Unlike their work, our work focuses mainly on how to efficiently compute the top-k edges with the highest structural diversities. From a computational viewpoint, Huang et al. [2] proposed several efficient online search algorithms to find the top-k vertices with the highest structural diversities in a network [2]. Chang et al. [4] proposed an improved algorithm for the same problem based on a carefully-designed vertex ordering. All the above mentioned techniques, however, only focus on the structural diversities of vertices, and they cannot be directly used for finding the top-k edges with the highest structural diversities.

Listing all 4-cliques. Our work is closely related to the 4-clique listing problem, where the goal is to list all 4-cliques in a graph. In [5], Chiba and Nishizeki proposed a sequential algorithm with the running time in $O(k \cdot m \cdot \alpha^{k-2})$ to list all k-cliques in sparse graphs, where α denotes the arboricity of the graph. Recently, Danisch et al. [19] proposed an improved algorithm with time complexity $O(k \cdot m \cdot \alpha(\frac{\delta}{2})^{k-2})$ to enumerate all k-cliques, where δ ($\delta \leq 2\alpha$) denotes the degeneracy of the graph [6]. In this work, we show a close connection between our problem and the 4-clique listing problem, and we also make use of the 4-clique listing algorithm to construct an index to support top-k edge structural diversity search.

Top-k query processing. Our work is also related to the top-k query processing techniques, where the goal is to find k objects with the highest rank based on some predefined ranking function [20]. There are many studies on top-k query processing for different kinds of applications, such as processing distributed preference queries [21], keyword queries [22], set similarity join queries [23], [24] and so on [25]–[29]. An excellent survey on this topic was given in [20]. In general, the key idea of many existing top-k query processing techniques is that they process the candidates according to a particular order and prune the search space based on some carefully-designed upper bounds. Inspired by this general idea, we develop a new *dequeue-twice* online search framework to identify the top-k edges with the highest structural diversities.

VIII. Conclusion

In this paper, we study the top-k edge structural diversity search problem, where the structural diversity of an edge is measured by the number of connected components in its ego-network. To solve our problem, we first propose a new *dequeue-twice* online search algorithm with two upper-bounding rules. Then, we propose a new index structure, called ESDIndex, to efficiently support the top-k edge structural diversity search. We show that the top-k edge structural diversity search can be processed in $O(k \log m + \log n)$ time with the ESDIndex, thus it is near-optimal. We also show that ESDIndex uses $O(\alpha m)$ space and can be created in $O((\alpha\gamma(n) + \log m)\alpha m)$ time, where α denotes the arboricity of the graph. Since α is typically very small in real-life sparse graphs, the ESDIndex based solution is very efficient in both time and space overheads. Additionally, we also develop efficient index maintenance techniques to handle dynamic graphs. We conduct extensive experiments using five large real-life networks, and the results demonstrate the efficiency, scalability, and effectiveness of the proposed solutions.

Acknowledgement. This work was partially supported by (i) NSFC Grants 61772346, 61732003, U1809206, 61836005, 61672358; (ii) National Key R&D Program of China 2018YFB1004402; (iii) Beijing Institute of Technology Research Fund Program for Young Scholars; (iv) ARC Discovery Project Grant DP160101513. Guoren Wang is the corresponding author of this paper.

References

[1] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg, "Structural diversity in social contagion," *Proceedings of the National Academy of Sciences*, vol. 109, no. 16, pp. 5962–5966, 2012.
[2] X. Huang, H. Cheng, R. Li, L. Qin, and J. X. Yu, "Top-k structural diversity search in large networks," *VLDB Journal*, vol. 24, no. 3, pp. 319–343, 2015.
[3] Y. Dong, R. A. Johnson, J. Xu, and N. V. Chawla, "Structural diversity and homophily: A study across more than one hundred big networks," in *KDD*, pp. 807–816, 2017.
[4] L. Chang, C. Zhang, X. Lin, and L. Qin, "Scalable top-k structural diversity search," in *ICDE*, pp. 95–98, 2017.
[5] N. Chiba and T. Nishizeki, "Arboricity and subgraph listing algorithms," *SIAM Journal on computing*, vol. 14, no. 1, pp. 210–223, 1985.
[6] D. Eppstein, M. Löffler, and D. Strash, "Listing all maximal cliques in large sparse real-world graphs," *ACM Journal of Experimental Algorithmics*, vol. 18, 2013.
[7] M. C. Lin, F. J. Soulignac, and J. L. Szwarcfiter, "Arboricity, h-index, and dynamic algorithms," *Theor. Comput. Sci.*, vol. 426, pp. 75–90, 2012.
[8] M. Ortmann and U. Brandes, "Triangle listing algorithms: Back from the diversion," in *ALENEX*, 2014.
[9] C. S. J. A. Nash-Williams, "Decomposition of finite graphs into forests," *Journal of the London Mathematical Society*, vol. 39, no. 1, pp. 12–12, 1964.
[10] J. Wang and J. Cheng, "Truss decomposition in massive networks," *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.
[11] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," *SIGMOD*, 2014.
[12] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *PVLDB*, vol. 8, no. 5, pp. 509–520, 2015.
[13] R. Li, L. Qin, J. X. Yu, and R. Mao, "Finding influential communities in massive networks," *VLDB J.*, vol. 26, no. 6, pp. 751–776, 2017.
[14] R.-H. Li, Q. Dai, L. Qin, G. Wang, X. Xiao, J. X. Yu, and S. Qiao, "Efficient signed clique search in signed networks," in *ICDE*, 2018.
[15] R.-H. Li, Q. Dai, G. Wang, Z. Ming, L. Qin, and J. X. Yu, "Improved algorithms for maximal clique search in uncertain networks," in *ICDE*, pp. 1178–1189, 2019.
[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.
[17] Z. Fang, X. Zhou, J. Tang, W. Shao, A. C. M. Fong, L. Sun, Y. Ding, L. Zhou, and J. Luo, "Modeling paying behavior in game social networks," in *CIKM*, pp. 411–420, 2014.
[18] H. Ma, "On measuring social friend interest similarities in recommender systems," in *SIGIR*, pp. 465–474, 2014.
[19] M. Danisch, O. Balalau, and M. Sozio, "Listing k-cliques in sparse real-world graphs," in *WWW*, pp. 589–598, 2018.
[20] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys*, vol. 40, no. 4, p. 11, 2008.
[21] K. C.-C. Chang and S.-w. Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD*, pp. 346–357, 2002.
[22] Y. Luo, X. Lin, W. Wang, and X. Zhou, "Spark: top-k keyword query in relational databases," in *SIGMOD*, pp. 115–126, 2007.
[23] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-k set similarity joins," in *ICDE*, pp. 916–927, 2009.
[24] Y. Kim and K. Shim, "Parallel top-k similarity join algorithms using mapreduce," in *ICDE*, pp. 510–521, 2012.
[25] F. Bi, L. Chang, X. Lin, and W. Zhang, "An optimal and progressive approach to online search of top-k influential communities," *Proceedings of the VLDB Endowment*, vol. 11, no. 9, pp. 1056–1068, 2018.
[26] R.-H. Li and J. X. Yu, "Scalable diversified ranking on large graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 9, pp. 2133–2146, 2013.
[27] L. Qin, J. X. Yu, and L. Chang, "Diversifying top-k results," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1124–1135, 2012.
[28] Y. Wang, G. Cong, G. Song, and K. Xie, "Community-based greedy algorithm for mining top-k influential nodes in mobile social networks," in *KDD*, pp. 1039–1048, 2010.
[29] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum, "Efficient top-k querying over social-tagging networks," in *SIGIR*, pp. 523–530, 2008.