2022 IEEE 38th International Conference on Data Engineering (ICDE) | 978-1-6654-0883-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDE53745.2022.00033

# Efficient Top-$k$ Ego-Betweenness Search

Qi Zhang[†], Rong-Hua Li[†], Minjia Pan[†], Yongheng Dai[‡], Guoren Wang[†], Ye Yuan[†]

[†]*Beijing Institute of Technology, Beijing, China;* [‡]*Diankeyun Technologies Co. , Ltd.;*
qizhangcs@bit.edu.cn; lironghuabit@126.com; panminjia_cs@163.com;
toyhdai@163.com; wanggrbit@126.com; yuan-ye@bit.edu.cn

***Abstract*—Betweenness centrality, measured by the number of times a vertex occurs on all shortest paths of a graph, has been recognized as a key indicator for the importance of a vertex in the network. However, the betweenness of a vertex is often very hard to compute because it needs to explore all the shortest paths between the other vertices. Recently, a relaxed concept called ego-betweenness was introduced which focuses on computing the betweenness of a vertex in its ego network. In this work, we study a problem of finding the top-$k$ vertices with the highest ego-betweennesses. We first develop two novel search algorithms equipped with a basic upper bound and a dynamic upper bound to efficiently solve this problem. Then, we propose local-update and lazy-update solutions to maintain the ego-betweennesses for all vertices and the top-$k$ results when the graph is updated by an edge insertion and deletion, respectively. In addition, we also present two efficient parallel algorithms to further improve the efficiency. The results of extensive experiments on five large real-life datasets demonstrate the efficiency, scalability, and effectiveness of our algorithms.**

## I. Introduction

Betweenness centrality is a fundamental metric in network analysis [1], [2]. The betweenness centrality of a vertex $v$ is the sum of the ratio of the shortest paths that pass through $v$ between other vertices in a graph. Such a centrality metric has been successfully used in a variety of network analysis applications, such as social network analysis [3], biological network analysis [4], communication network analysis [5] and so on. More specifically, in social networks, a vertex with a high betweenness centrality is plausibly an influential user who can decide whether to share information or not [3]. In protein interaction networks, the high-betweenness proteins represent important connectors that link some modular organizations [4]. In communication networks, the nodes with higher betweennesses might have more control over the network, thus attacking these nodes may cause severe damage to the network [5].

Although betweenness centrality plays a critical role in network analysis, computing betweenness scores for all vertices is notoriously expensive because it requires exploring the shortest paths between all vertices in a graph. The state-of-the-art algorithm for betweenness computation is the Brandes' algorithm [6] which takes $O(nm)$ time. Such a time complexity is acceptable only in small graphs with a few tens of thousands of vertices and edges, but it is prohibitively expensive on modern networks with millions of vertices and tens of millions of edges.

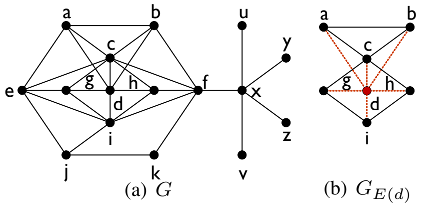
To avoid the high computational cost problem, Everett *et al.* [7] introduced a relaxed concept called ego-betweenness centrality which focuses on computing a vertex's betweenness in its ego network, where the ego network of a vertex $u$ is the subgraph induced by $u$ and $u$'s neighbors. More specifically, the ego-betweenness of a vertex $u$ is measured by the sum of the ratio of the shortest paths that pass through $u$ between $u$'s neighbors in the ego network. Everett *et al.* showed that the ego-betweenness centrality is highly correlated with the traditional betweenness centrality in networks, thus it can be considered as a good approximation of the traditional betweenness. In addition, ego-betweenness can also measure the importance of a node [7], [8]. Unlike traditional betweenness, which plays the role of "bridge" in a network, ego-betweenness focuses on the links existing from the perspective of a node and appraises the ability of a node as a "center" in its ego network. With this property, ego betweenness can be useful in the networks where global topology knowledge is inaccessible or the network presents small-world features, such as social networks [9], wireless sensor networks [10], mobile ad-hoc networks [11], and vehicular ad-hoc networks [12]. Moreover, real-life applications often require retrieving the top-$k$ vertices with the highest ego-betweenness scores, rather than the exact ego-betweenness scores for all vertices. Motivated by this, we in this paper study the problem of identifying the top-$k$ vertices in a graph with the highest ego-betweennesses.

To solve the top-$k$ ego-betweenness search problem, a straightforward algorithm is to calculate the ego-betweennesses for all vertices and then select the top-$k$ results. However, such a straightforward algorithm is very costly for large graphs, because the total cost for constructing the ego network for each vertex is very expensive in large graphs. To efficiently compute the top-$k$ vertices, the general idea of top-$k$ search frameworks [13]–[15] can be used, which explores the vertices based on a predefined ordering and then applies some upper-bounding rules to prune the unpromising vertices. Inspired by these algorithms, we first derive a basic upper bound and a dynamic upper bound of ego-betweenness. Then, we develop two top-$k$ search algorithms with those bounds to efficiently solve the top-$k$ ego-betweenness search problem. To handle dynamic graphs, we present local-update solutions to maintain ego-betweennesses for all vertices, and also develop lazy-update techniques to maintain the top-$k$ results. Additionally, we propose two efficient parallel algorithms to improve the efficiency of ego-betweenness computation. In summary, we make the following contributions.

**Top-$k$ search algorithms.** We develop a basic algorithm with a static upper bound and an improved algorithm with a tighter and dynamically-updating upper bound to find the top-$k$ vertices with the highest ego-betweennesses. The two algorithms consume $O(\alpha m d_{\max})$ and $O(\alpha m d_{\max} + m \log n)$ time using $O(m d_{\max})$ space in the worst case respectively. Here $\alpha$ is the arboricity of the graph [16] which is typically very small in real-life graphs [17]. We show that both algorithms can significantly prune the vertices that are definitely not contained in the top-$k$ results. Moreover, the improved algorithm can achieve more effective pruning performance on real-world graphs due to the tighter and dynamically-updating upper bound.

**Ego-betweenness maintenance and parallel algorithms.** To handle dynamic graphs, we present local-update algorithms to maintain the ego-betweennesses of all vertices when the graph is updated by inserting or deleting an edge. We also propose lazy-update techniques to maintain the top-$k$ results for the updates of an edge. To further improve the efficiency, we present two efficient parallel algorithms to compute all

2375-026X/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDE53745.2022.00033



(a) $G$ (b) $G_{E(d)}$

Fig. 1. Running example

vertices' ego-betweennesses. Compared with the sequential algorithms, our parallel solutions can achieve a high degree of parallelism, thus improving the efficiency of ego-betweenness computation significantly.

**Extensive experiments.** We conduct comprehensive experimental studies to evaluate the proposed algorithms using five large real-world datasets. The results show that 1) our improved algorithm with a dynamic upper bound is roughly 5-23 times faster than the basic algorithm with a static upper bound; 2) our maintenance algorithms can maintain the top-$k$ results in less than 12 seconds in a large graph with 58,655,848 vertices and 261,321,033 edges; 3) our best parallel algorithm can achieve near 8 speedup ratio when using 16 threads; 4) the top-$k$ results of ego-betweenness are highly similar to the top-$k$ results of traditional betweenness. Thus, our results indicate that the ego-betweenness metric can be seen as a very good approximation of the traditional betweenness metric, but it is much cheaper to compute by utilizing the proposed algorithms.

**Reproducibility.** For reproducibility, the source code of this paper is released at github: https://github.com/QiZhang1996/egobetweenness.

## 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 $N(u, v)$, are the vertices that are adjacent to both $u$ and $v$, i.e., $N(u, v) = \{w \in V|(u, w) \in E, (v, w) \in E\}$. Denote by $d_{\max}$ the maximum degree of the vertices in $G$. 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 $E_S = \{(u, v)|u, v \in S, (u, v) \in E\}$.

We define a total order $\prec$ on $V$ as follows. For vertices $u$ and $v$ in $V$, we say $u \prec v$, if and only if 1) $d(u) > d(v)$ or 2) $d(u) = d(v)$ and $u$ has a larger ID than $v$. Based on such a degree ordering $\prec$, we can construct a directed graph $G^+$ from $G$ by orientating each undirected edge $(u, v) \in G$ to respect the total order $u \prec v$. We denote the out-neighborhood of $u$ in $G^+$ as $N^+(u) = \{v \in V|(u, v) \in E^+\}$.

We give an essential concept, called *ego network*, as follows.

*Definition 1:* (Ego network) For vertex $p$ in $G = (V, E)$, the *ego network* of $p$, denoted by $G_{E(p)}$, is a subgraph of $G$ induced by the vertex set $N(p) \cup \{p\}$.

Given a vertex $p$ in $G$ and its ego network $G_{E(p)}$, for $u, v \in N(p)$, let $g_{uv}$ be the number of the shortest paths connecting $u$ and $v$ in $G_{E(p)}$ and $g_{uv}(p)$ be the number of those shortest paths that contain vertex $p$. Note that in $G_{E(p)}$, $g_{uv}(p)$ is either 0 or 1. Denote by $b_{uv}(p) = g_{uv}(p)/g_{uv}$ the probability that a randomly selected shortest path connecting $u$ with $v$ contains $p$ in $G_{E(p)}$. Based on the above notions, the definition of *ego-betweenness* is given as follows.

*Definition 2:* (Ego-betweenness [7]) For a vertex $p$ in $G$ , the ego-betweenness of $p$, denoted by $C_B(p)$, is defined as $C_B(p) = \sum_{u \prec v} b_{uv}(p), u, v \in N(p)$. □

*Example 1:* Consider a graph $G$ in Fig. 1(a) and a vertex $d \in G$ with the ego network $G_{E(d)}$ illustrated in Fig. 1(b). For vertices $c$ and $i$, there are three shortest paths connecting $c$ and $i$ in $G_{E(d)}$, namely, $c \to g \to i$, $c \to h \to i$, and $c \to d \to i$, thus $g_{ci} = 3$ and $g_{ci}(d) = 1$, further $b_{ci}(d) = g_{ci}(d)/g_{ci} = 1/3$ holds. Analogously, we have $b_{hg}(d) = 1/3$, $b_{ga}(d) = b_{gb}(d) = b_{ha}(d) = b_{hb}(d) = 1/2$, $b_{ia}(d) = b_{ib}(d) = 1$ and the probabilities for other vertex pairs in $G_{E(d)}$ are equal to 0, thus $C_B(d) = 14/3$. □

**Problem definition.** Given a graph $G$ and an integer $k$, the top-$k$ ego-betweenness search problem is to identify the $k$ vertices in $G$ with the highest ego-betweenness scores.

In addition, real-world networks undergo dynamically updates. To this end, we also investigate the problem of top-$k$ ego-betweenness maintenance when the graph is updated.

**Challenges.** To solve the top-$k$ ego-betweenness search problem, a straightforward algorithm is to compute the ego-betweenness for each vertex, and then pick the top-$k$ vertices as the answers. Such an approach, however, is costly for large graphs. This is because the algorithm needs to explore the ego network $G_{E(p)}$ to compute the ego-betweenness for each vertex $p$. The total size of all ego networks could be very large, thus the straightforward algorithm might be very expensive for large graphs. Since we are only interested in the top-$k$ results, we do not need to compute all vertices' ego-betweenness scores exactly. The challenges of the problem are: 1) how to efficiently prune the vertices that are definitely not contained in the top-$k$ results; 2) how to efficiently compute the ego-betweenness for each vertex; 3) how to maintain the top-$k$ vertices with the highest ego-betweennesses in dynamic networks. To tackle these challenges, we will develop two new online search algorithms with two non-trivial punning techniques to efficiently search the top-$k$ ego-betweenness vertices. Then, we also design local update techniques and lazy update techniques to handle frequent updates and maintain the top-$k$ results.

## III. Top-$k$ ego-betweenness search

### A. *The* BaseBSearch *algorithm*

Given a graph $G = (V, E)$ and a vertex $p \in V$. We use $\bar{S}_{E(p)}$ to store the edges between the neighbors of $p$, i.e., $\bar{S}_{E(p)} = \{(u, v)|u, v \in N(p), (u, v) \in E\}$. For vertices $u, v \in N(p)$ and $(u, v) \notin E$, we suppose that $u \prec v$. Let $\hat{S}_p(u, v)$, which does not include $p$, be the set of vertices that connect $u$ and $v$ in $G_{E(p)}$, i.e., $\hat{S}_p(u, v) = \{w|u, v, w \in N(p), (u, v) \notin E, (u, w) \in E, (v, w) \in E\}$. If $p$ is the only one vertex $p$ that links $u$ and $v$ in $G_{E(p)}$, we add the pair $(u, v)$ into the set $\ddot{S}_{E(p)}$, i.e., $\ddot{S}_{E(p)} = \{(u, v)|u, v, w \in N(p), (u, v) \notin E, \nexists w \in N(p), (u, w) \in E, (v, w) \in E\}$. Denote by $\hat{S}_{E(p)}$ the collection of all $\hat{S}_p(u, v)$s. Based on these notations, all the vertex pairs are divided into three categories, i.e., $\bar{S}_{E(p)}$, $\hat{S}_{E(p)}$ and $\ddot{S}_{E(p)}$. We use $\bar{C}_p$ to represent the size of $\bar{S}_{E(p)}$, i.e., $\bar{C}_p = |\bar{S}_{E(p)}|$. Similarly, we denote $\hat{C}_p = |\hat{S}_{E(p)}|$ and $\ddot{C}_p = |\ddot{S}_{E(p)}|$.

*Example 2:* Consider the ego-network $G_E(d)$ in Fig. 1(b). The vertex pair $(a, b)$ belongs to $\bar{S}_{E(d)}$ because the edge $(a, b)$ exists in $G_E(d)$. For the pair $(a, g)$, we have $\hat{S}_p(a, g) = \{c\}$ as it maintains the vertices that can connect $a$ and $g$ in $G_{E(d)}$ but does not include the ego vertex $d$. While for the pair $(a, i)$, there is no vertex can link $a$ and $i$ expect $d$, thus we add $(a, i)$ into $\ddot{S}_{E(p)}$. Generally, we have $\hat{S}_{E(p)} = \{((a, g), \{c\}), ((a, h), \{c\}), ((b, g), \{c\}), ((b, h), \{c\}), ((c, i), \{g,$

$h\}),((g,h),\{c,i\})\}$, $\ddot{S}_{E(p)} = \{(a,i),(b,i)\}$ and the other pairs are in $\bar{S}_{E(p)}$. □

Before introducing the BaseBSearch algorithm, we first give some useful lemmas which lead to an upper bound of ego-betweenness for pruning search space in BaseBSearch.

*Lemma 1:* For any vertex $p$ in $G$, we have $\bar{C}_p + \hat{C}_p + \ddot{C}_p = \frac{d(p)*(d(p)-1)}{2}$.

*Proof:* Clearly, the vertex pairs between vertex $p$'s neighbors are divided into three categories, namely, $\bar{S}_{E(p)}$, $\hat{S}_{E(p)}$ and $\ddot{S}_{E(p)}$. Therefore, the sum of $\bar{C}_p$, $\hat{C}_p$ and $\ddot{C}_p$ is the number of all vertex pairs between $N(p)$, i.e., $\bar{C}_p + \hat{C}_p + \ddot{C}_p = \frac{d(p)*(d(p)-1)}{2}$. □

*Lemma 2:* For any vertex $p$ in $G$, $C_B(p) = \frac{d(p)*(d(p)-1)}{2} - \bar{C}_p - \hat{C}_p + \sum_{(u,v)\in\hat{S}_{E(p)}} \frac{1}{|\hat{S}_p(u,v)|+1} \leq \overline{\mathsf{ub}}(p) = \frac{d(p)*(d(p)-1)}{2}$ holds.

*Proof:* Based on Definition 2, $C_B(p)$ is closely related to the number of shortest paths between $u$ and $v$ in $G_{E(p)}$. First, for each $(u,v) \in \ddot{S}_{E(p)}$, there is only one vertex $p$ that can link $u$ and $v$, so $b_{uv}(p)$ is equal to 1. Thus, $\ddot{C}_p$ is a part of $C_B(p)$ which equals $\frac{d(p)*(d(p)-1)}{2} - \bar{C}_p - \hat{C}_p$ according to Lemma 1. Second, for every vertex pair $(u,v) \in \hat{S}_{E(p)}$, $\hat{S}_p(u,v)$ is the set of vertices connecting $u$ with $v$ in $G_{E(p)}$ but does not include $p$, thus the probability $b_{uv}(p)$ is equal to $\frac{1}{|\hat{S}_p(u,v)|+1}$. To sum up, $C_B(p) = \frac{d(p)*(d(p)-1)}{2} - \bar{C}_p - \hat{C}_p + \sum_{(u,v)\in\hat{S}_{E(p)}} \frac{1}{|\hat{S}_p(u,v)|+1}$. When $\hat{S}_{E(p)}$ is not empty, we have $\hat{C}_p = |\hat{S}_{E(p)}| = \sum_{(u,v)\in\hat{S}_p(u,v)} 1 > \sum_{(u,v)\in\hat{S}_{E(p)}} \frac{1}{|\hat{S}_p(u,v)|+1}$. Otherwise, $\hat{C}_p = \sum_{(u,v)\in\hat{S}_{E(p)}} \frac{1}{|\hat{S}_p(u,v)|+1} = 0$. Since $\bar{C}_p$ is no less than 0, thus $C_B(p) \leq \overline{\mathsf{ub}}(p)$ holds. □

Equipped with Lemma 2, we present a basic search approach, called BaseBSearch, which computes the vertices' ego-betweennesses in non-increasing order of their upper bounds. The main idea of BaseBSearch is that a vertex with a large upper bound may have a high chance contained in the top-$k$ results. Based on this idea, the exact computations for the vertices with small upper bounds will be postponed or even avoided, thus BaseBSearch can significantly improve the efficiency compared with the algorithm calculating all ego-betweennesses.

The pseudo-code of BaseBSearch is outlined in Algorithm 1. For each vertex $u$, $S_u$ is a map to maintain the number of the shortest paths that do not go through $u$ for all neighbor pairs. Algorithm 1 works as follows. It first calculates the upper bound $\overline{\mathsf{ub}}(u)$ for each vertex $u$ based on Lemma 2 and initializes $C_B(u)$ as $\overline{\mathsf{ub}}(u)$ (lines 1-2). Then, it sorts the vertices in non-increasing order with respect to their upper bounds, and picks an unexplored vertex $u$ with the maximum $\overline{\mathsf{ub}}(u)$ to calculate $C_B(u)$ until the top-$k$ vertices are found (lines 6-19). During the processing of vertex $u$, if the result set $R$ has $k$ vertices and the $\min_{v\in R} C_B(v) \geq \overline{\mathsf{ub}}(u)$ holds, the algorithm terminates (line 7). Otherwise, BaseBSearch computes $C_B(u)$ and identifies whether $u$ should be added into the answer set $R$ (lines 8-18). For vertex $u$, we explore the number of shortest paths between $u$'s neighbors by enumerating the triangles including $u$ and maintain them in the hash map $S_u$. In $S_u$, we always keep a vertex pair $(i,j)$ with $val = 0$ if $i$ and $j$ are connected in $G_{E(u)}$; on the other hand, $val$ records the number of vertices that link $i$ and $j$ but not contain $u$. When a $\triangle_{(u,v,w)}$ is found, we update the hash maps for $u$, $v$ and $w$ (lines 12-13). Note that BaseBSearch processes vertices in the order of the upper bounds (i.e., the total order), all triangles containing $u$ can be touched without omission after handling $u$ and $S_u$ maintains the number of the shortest paths correctly. Further, the algorithm calculates $C_B(u)$ according to Lemma 2 and updates $R$ (lines 15-18). Finally, BaseBSearch outputs the answer set $R$.

```
Algorithm 1: BaseBSearch $(G, k)$
   Input: $G = (V, E)$, an integer $k \geq 1$.
   Output: The top-$k$ vertex set $R$.
1  for $u \in V$ do
2      $\overline{\mathsf{ub}}(u) \leftarrow \frac{d(u)*(d(u)-1)}{2}$; $C_B(u) \leftarrow \overline{\mathsf{ub}}(u)$;
3  $R \leftarrow \emptyset$;
4  Construct the oriented graph $G^+ = (V, E^+)$ of $G$;
5  Initialize an array $B$ with $B(i) = false, 0 \leq i < n$;
6  for $u \in V$ according to the total order do
7      if $|R| = k$ and $\min_{v\in R} C_B(v) \geq \overline{\mathsf{ub}}(u)$ then break;
8      for $v \in N^+(u)$ do $B(v) \leftarrow true$;
9      for $v \in N^+(u)$ do
10         for $w \in N^+(v)$ do
11             if $B(w) = true$ then
12                 UptSMap$(S_u, v, w)$; UptSMap$(S_v, u, w)$;
13                 if $\nexists S_w(u,v)$ then $S_w$.insert$((u,v),0)$;
14     for $v \in N^+(u)$ do $B(v) \leftarrow false$;
15     for $((i,j), val) \in S_u$ do
16         $C_B(u) \leftarrow C_B(u) - 1$;
17         if $val \neq 0$ then $C_B(u) \leftarrow C_B(u) + \frac{1}{val+1}$;
18     Update $R$ based on $u$ and $C_B(u)$;
19 return $R$;
20 Procedure UptSMap$(S_u, v, w)$
21 for $x \in N(u)$ do
22     if $(x,v) \in E$ and $\nexists S_u(x,v)$ then $S_u$.insert$((x,v),0)$;
23     if $(x,w) \in E$ and $\nexists S_u(x,w)$ then $S_u$.insert$((x,w),0)$;
24     if $(x,v) \in E$ and $(x,w) \notin E$ then
25         if $\nexists S_u(x,w)$ then $S_u$.insert$((x,w),1)$;
26         else if $S_u(x,w).val \neq 0$ then $S_u(x,w).val$++;
27     if $(x,v) \notin E$ and $(x,w) \in E$ then
28         Update $S_u$ and $S_w$ as lines 25-26;
```

| $v$ | c | i | f | d | x | e | h | g | b | a |
|---|---|---|---|---|---|---|---|---|---|---|
| $\overline{ub}$ | 21 | 15 | 15 | 15 | 10 | 10 | 6 | 6 | 6 | 6 |
| $C_B(v)$ | 41/6 | 8 | 11 | 14/3 | 10 | 4.5 | 2/3 | 2/3 | 1 | 1 |
| $R$ | {c} | {i, c} | {f, i, c} | {f, i, c, d} | {f, x, i, c, d} | | | | | |

Fig. 2. The running process of BaseBSearch on $G$

*Example 3:* Consider a graph $G$ as shown in Fig. 1(a) and an integer $k = 5$. The running process of Algorithm 1 on this graph is illustrated in Fig. 2. The algorithm computes the ego-betweennesses of $c, i, f, d, x, e, h, g, b, a$ in turn based on their upper bounds (i.e., the total order). After computing $C_B(a)$, the largest upper bound among the remaining vertices: $j, k, u, v, x, y, z$ is $\overline{\mathsf{ub}}(j) = 3 < C_B(d) = 14/3$ ($d$ is the 5-th element in $R$), thus Algorithm 1 terminates. Compared with calculating the ego-betweennesses of all vertices, BaseBSearch can save 6 ego-betweenness computations by utilizing the upper bound $\overline{\mathsf{ub}}$. □

### B. The OptBSearch algorithm

BaseBSearch may not be very efficient for top-$k$ search because the upper bound $\overline{\mathsf{ub}}$ is not very tight. To further improve the efficiency, we propose the OptBSearch algorithm with a dynamic upper bound $\widetilde{\mathsf{ub}}$ which is tighter than $\overline{\mathsf{ub}}$.

Recall that we calculate $C_B(u)$ with the information of the shortest paths which is derived by touching the triangles including vertex $u$. In this processing, some useful information about the number of shortest paths for $u$'s neighbors can also be obtained. We refer to those information as identified in-

**Algorithm 2:** OptBSearch $(G, k, \theta)$

```
Input: $G = (V, E)$, an integer $k \geq 1$, a gradient ratio $\theta \geq 1$.
Output: The top-$k$ vertex set $R$.
1  $H \leftarrow \emptyset$; $R \leftarrow \emptyset$;
2  Initialize an array $B$ with $B(i) = false, 0 \leq i < n$;
3  for $v \in V$ do
4      $\widetilde{\mathsf{ub}}(v) \leftarrow \frac{d(v)*(d(v)-1)}{2}$; $C_B(v) \leftarrow \widetilde{\mathsf{ub}}(v)$; $H.push(v, \widetilde{\mathsf{ub}}(v))$;
5  while $H \neq \emptyset$ do
6      $(v^*, \widehat{\mathsf{tb}}) \leftarrow H.pop()$;
7      Compute $\widetilde{\mathsf{ub}}(v^*)$ according to Lemma 3;
8      if $\theta \cdot \widetilde{\mathsf{ub}}(v^*) < \widehat{\mathsf{tb}}$ then
9          if $|R| < k$ or $\widetilde{\mathsf{ub}}(v^*) > \min_{v \in R} C_B(v)$ then
10             $H.push(v^*, \widetilde{\mathsf{ub}}(v^*))$;
11         continue;
12     if $|R| = k$ and $\widehat{\mathsf{tb}} \leq \min_{v \in R} C_B(v)$ then break;
13     EgoBWCal $(G, v^*, B)$;
14     if $|R| < k$ then $R \leftarrow R \cup \{v^*\}$;
15     else if $C_B(v^*) > \min_{v \in R} C_B(v)$ then
16         $u \leftarrow \arg\min_{v \in R} C_B(v)$; $R \leftarrow (R - \{u\}) \cup \{v^*\}$;
17     $B(v^*) \leftarrow true$;
18 return $R$;
```

**Algorithm 3:** EgoBWCal $(G, u, B)$

```
Input: $G = (V, E)$, vertex $u$, an array $B$.
Output: $C_B(u)$.
1  Initialize $DN$ and $EN$ according to $B$;
2  Initialize an array $V_{is}$ with $V_{is}(i) = false, 0 \leq i < n$;
3  for $i \in N(u)$ do $rd(i) \leftarrow \emptyset$;
4  for $((i, j), val) \in S_u$ do
5      if $val = 0$ then
6          $rd(i) \leftarrow rd(i) \cup \{j\}$; $rd(j) \leftarrow rd(j) \cup \{i\}$;
7  for $i \in DN$ do
8      for $p \in rd(i)$ do $V_{is}(p) \leftarrow true$;
9      for $j \in DN - \{i\}$ do
10         if $V_{is}(j) = false$ then
11             for $p \in rd(j)$ do
12                 if $V_{is}(p) = true$ and $B(p) = false$ then
13                     $S_u(i, j).val$++; $S_p(i, j).val$++;
14 for $i \in EN$ do
15     for $j \in EN - \{i\}$ do
16         if $(i, j) \in E$ then
17             $S_u$.insert$((i, j), 0)$; $S_i$.insert$((u, j), 0)$;
18             $S_j$.insert$((u, i), 0)$;
19             for $k \in rd(j)$ do
20                 if $\nexists S_u(i, k)$ then
21                     $S_u$.insert$((i, k), 1)$; $S_j$.insert$((i, k), 1)$;
22                 else if $S_u(i, k).val \neq 0$ then
23                     $S_u(i, k).val$++; $S_j(i, k).val$++;
24             Update $S_u, S_i$ by $rd(i)$ as lines 19-23;
25             $rd(i) \leftarrow rd(i) \cup \{j\}$; $rd(j) \leftarrow rd(j) \cup \{i\}$;
26 Calculate $C_B(u)$ as lines 15-17 of Algorithm 1;
27 return $C_B(u)$;
```

formation which include some vertex pairs and edges. Below, we will use these identified information to derive a tighter and dynamically-updated upper bound of ego-betweenness.

Given a vertex $p$, let $*\bar{S}_{E(p)}$ be the collection of identified edges in $G_{E(p)}$ and $*\hat{S}_{E(p)}$ be the set of the currently identified vertex pairs whose property is the same as the pairs in $\hat{S}_{E(p)}$. For a vertex pair $(u, v)$ in $*\hat{S}_{E(p)}$, denote by $*\hat{S}_{p(u,v)}$ the set of identified vertices that link $u$ and $v$ but does not contain $p$. Let $*\bar{C}_p$ and $*\hat{C}_p$ be the size of $*\bar{S}_{E(p)}$ and $*\hat{S}_{E(p)}$, respectively. We develop a tighter upper bound of ego-betweenness $\widetilde{\mathsf{ub}}$ in Lemma 3.

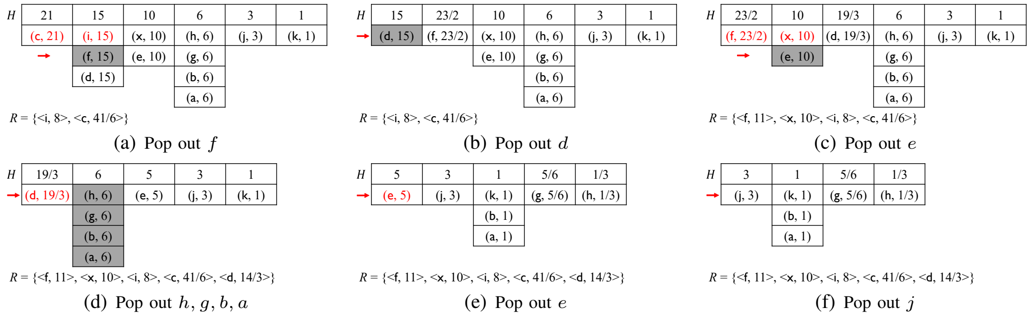
*Lemma 3:* For a vertex $p$ in $G$, $C_B(p) \leq \widetilde{\mathsf{ub}}(p) = \frac{d(p)*(d(p)-1)}{2} - *\bar{C}_p - *\hat{C}_p + \sum_{(u,v) \in *\hat{S}_{E(p)}} \frac{1}{|*\hat{S}_{p(u,v)}|+1}$ holds.

*Proof:* By definition, we have $*\bar{C}_p \leq \bar{C}_p$, $*\hat{C}_p \leq \hat{C}_p$ and $| * \hat{S}_{p(u,v)}| \leq |\hat{S}_{p(u,v)}|$. Further, $\sum_{(u,v) \in *\hat{S}_{E(p)}} \frac{1}{|*\hat{S}_{p(u,v)}|+1} \geq \sum_{(u,v) \in \hat{S}_{E(p)}} \frac{1}{|\hat{S}_{p(u,v)}|+1}$ holds. According to Lemma 2, we can obtain $C_B(p) \leq \widetilde{\mathsf{ub}}(p) = \frac{d(p)*(d(p)-1)}{2} - *\bar{C}_p - *\hat{C}_p + \sum_{(u,v) \in *\hat{S}_{E(p)}} \frac{1}{|*\hat{S}_{p(u,v)}|+1} \leq \overline{\mathsf{ub}}(p)$. □

Note that the upper bound $\widetilde{\mathsf{ub}}(p)$ in Lemma 3 will be dynamically updated during the execution of the top-$k$ search algorithm, because $*\bar{C}_p$, $*\hat{C}_p$ and $| * \hat{S}_{p(u,v)}|$ will be updated when calculating vertices' ego-betweennesses exactly. The OptBSearch framework with such a dynamic upper bound is depicted in Algorithm 2. It first calculates $\widetilde{\mathsf{ub}}(v)$ and $C_B(v)$ for each vertex $v$, and pushes $v$ with the initial bound $\widetilde{\mathsf{ub}}(v)$ into a sorted list $H$ (lines 3-4). Then, the OptBSearch iteratively finds the top-$k$ results (lines 5-17). It pops the vertex $v^*$ with the largest upper bound value $\widehat{\mathsf{tb}}$ from $H$. As the number of shortest paths between $v^*$'s neighbors may be updated, the algorithm calculates $\widetilde{\mathsf{ub}}(v^*)$ based on Lemma 3. OptBSearch then compares $\widetilde{\mathsf{ub}}(v^*)$ with the old bound $\widehat{\mathsf{tb}}$ by employing a parameter $\theta \geq 1$ to avoid frequently calculating the upper bounds and updating $H$. When $\theta \cdot \widetilde{\mathsf{ub}}(v^*) < \widehat{\mathsf{tb}}$, that means $\widetilde{\mathsf{ub}}(v^*)$ is substantially smaller than $\widehat{\mathsf{tb}}$. If $|R| < k$ or $\widetilde{\mathsf{ub}}(v^*) > \min_{v \in R} C_B(v)$, we push $v^*$ to $H$ again with the tighter bound $\widetilde{\mathsf{ub}}(v^*)$ (line 10). Otherwise, $v^*$ does not belong to the top-$k$ answers and thus can be pruned. In both cases, the algorithm needs to pop the next vertex from $H$. If the early termination condition (line 12) is not satisfied, the algorithm performs EgoBWCal to compute $C_B(v^*)$ exactly and updates $R$ based on $C_B(v^*)$ (lines 13-17). Note that we use an array $B$ to record the vertices whose ego-betweennesses have been calculated, which can reduce redundant computations in the EgoBWCal procedure.

Algorithm 3 outlines the EgoBWCal procedure. Like BaseBSearch, a key issue is maintaining the number of the shortest paths in $S_u$ correctly by finding the triangles containing $u$. To avoid reduction, a simple but efficient approach is to record those enumerated triangles and update $S_u$ by deriving the shortest paths from these triangles. To this end, for each neighbor $i$ of $u$, Algorithm 3 uses $rd(i)$ to store such vertices that are contained in the touched triangles $\triangle_{(i,*,u)}$. It first initializes $rd(i)$ for every $i \in N(u)$ with the current $S_u$ as $S_u(i, j).val$ equals 0 indicates a visited triangle $\triangle_{(i,j,u)}$ (lines 3-6). Then, the procedure handles $u$'s neighbors to maintain $S_u$ according to whether they have been processed (lines 7-25). Specifically, if $B(i) = true$, we put $i$ into the set $DN$ and call it a processed vertex; otherwise, $i$ is added into the set $EN$ where stores the vertices to be processed. For the vertices $i, j \in DN$, EgoBWCal finds their common neighbors (denoted by $p$) based on $rd(i)$ and $rd(j)$ and updates the number of the shortest paths between $i$ and $j$ for $S_u$ and $S_p$ (lines 7-13). On the other hand, given $i, j \in EN$, the procedure enumerates new triangles and maintains related hash maps with $rd(i)$ and $rd(j)$ (lines 14-25). Note that with the discovery of new triangles, EgoBWCal also updates the related $rd(i)$s to avoid reduction (line 25). Finally, EgoBWCal calculates $C_B(u)$ with the same method as used in BaseBSearch.

*Example 4:* Reconsider the graph $G$ in Fig. 1(a). Suppose that $k = 5$ and $\theta = 1$. The running process of Algorithm 2 is illustrated in Fig. 3. The vertices colored red are computed their ego-betweennesses exactly and the vertices in gray grids



(a) Pop out $f$

(b) Pop out $d$

(c) Pop out $e$

(d) Pop out $h, g, b, a$

(e) Pop out $e$

(f) Pop out $j$

Fig. 3. The running process of $\mathsf{OptBSearch}$ on $G$

need to update their upper bounds and push back into $H$ again. The algorithm pushes all vertices with the initial upper bounds into $H$ and then processes them based on $H$. First, it pops $c$ with the largest upper bound $\widehat{\mathsf{tb}} = 21$ and calculates $\widetilde{\mathsf{ub}}(c)$ and $C_B(c)$. Due to $R = \emptyset$, $c$ is added into $R$ and $\mathsf{OptBSearch}$ does the same operation for $i$. Then, $f$ is popped with $\widehat{\mathsf{tb}} = 15$ and Algorithm 2 calculates $\widetilde{\mathsf{ub}}(f)$ as shown in Fig. 3(a). Since $\widetilde{\mathsf{ub}}(f) = 23/2$ is substantially smaller than $\widehat{\mathsf{tb}}$ based on $\theta = 1$, we push $(f, 23/2)$ into $H$ again and pops $d$ as the next processing vertex in Fig. 3(b). The tighter bound $\widetilde{\mathsf{ub}}(d) = 19/3$ is less than 15, thus $\mathsf{OptBSearch}$ pushes $d$ into $H$ again with $\widetilde{\mathsf{ub}}(d)$. In the following three iterations, $\mathsf{OptBSearch}$ computes $C_B(f)$ and $C_B(x)$ and adds them into $R$, and then processes $e$ as shown in Fig. 3(c). $e$ is pushed into $H$ with $\widetilde{\mathsf{ub}}(e) = 4$ and the algorithm pops $d$ to calculate $C_B(d)$ and adds $d$ into $R$ in Fig. 3(d). Due to $\widehat{\mathsf{tb}} = 6 > C_B(d)$, $h$ is popped and we calculate $\widetilde{\mathsf{ub}}(h)$ to update $H$. Similarly, we push $g, b, a$ into $H$ again with $\widetilde{\mathsf{ub}}(g), \widetilde{\mathsf{ub}}(b), \widetilde{\mathsf{ub}}(a)$ as shown in Fig. 3(e). When $e$ is processed, $C_B(e) = 9/2 < C_B(d)$ and $|R| = k = 5$ hold, thus $e$ is not an answer of top-$k$ results. When pops $j$ in Fig. 3(f), the algorithm safely prunes $j$ since $\widetilde{\mathsf{ub}}(j) < C_B(d)$. Obviously, the remaining vertices can also be pruned. In $\mathsf{OptBSearch}$, we invoke $\mathsf{EgoBWCal}$ six times to calculate the ego-betweennesses, while $\mathsf{BaseBSearch}$ performs ten ego-betweenness computations. □

### C. Analysis of the proposed algorithms

Below, we mainly analyze the correctness of Algorithm 1. The correctness analysis of Algorithm 2 is similar to that of Algorithm 1, thus we omit it for brevity.

*Theorem 1:* Given a graph $G = (V, E)$ and an integer $k$, Algorithm 1 correctly computes the top-$k$ vertices with the highest ego-betweennesses.

*Proof:* Recall that Algorithm 1 iteratively processes the vertices based on their upper bounds (Lemma 2). When a vertex $u$ is handled, if the answer set $R$ has $k$ vertices and $\min_{v \in R} C_B(v) \geq \widetilde{\mathsf{ub}}(u)$, then $C_B(u) \leq \widetilde{\mathsf{ub}}(u) \leq \min_{v \in R} C_B(v)$ holds. For any vertex $w \in V$ with a smaller degree, we have $C_B(w) \leq \widetilde{\mathsf{ub}}(w) \leq \widetilde{\mathsf{ub}}(u) \leq \min_{v \in R} C_B(v)$. Therefore, the algorithm can safely prune the remaining vertices and terminate, thereby the set $R$ exactly contains the top-$k$ answers. □

Before analyzing the time complexity, we give the concept of arboricity of a graph $G$ as follows [18].

*Definition 3:* (Arboricity) Given a graph $G = (V, E)$ with $n \geq 2$, the arboricity $\alpha$ 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 $\alpha$ is an important metric to measure the sparsity of a graph which is typically very small for most real-world graphs [18]. It was widely used to bound the time complexity of many graph analysis algorithms [14], [15], [17], [19]–[21]. Below, we also analyze the time complexity of Algorithm 1 and Algorithm 2 based on the parameter $\alpha$.

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

*Proof:* In lines 6-18 of Algorithm 1, the algorithm needs to enumerate each triangle once which takes $O(\alpha m)$ time. Note that when a triangle $\triangle_{(u,v,w)}$ is enumerated, the algorithm requires to maintain $S_u, S_v, S_w$. The time overhead of the update operator can be bounded by $O(d(u)) \leq O(d_{\max})$. Hence, the time complexity of Algorithm 1 is $O(\alpha m d_{\max})$. Second, we analyze the space complexity of Algorithm 1. Clearly, the space overhead is dominated by the size of the map structure $S_u$. For $u \in V$, the map structure $S_u$ contains $O(d(u)^2)$ vertex pairs, thus the space complexity is $O(\sum_{u \in V} d(u)^2) \leq O(d_{\max} m)$. □

*Theorem 3:* In the worst case, Algorithm 2 takes $O(\alpha m d_{\max} + m \log n)$ time using $O(d_{\max} m)$ space.

*Proof:* Algorithm 2 also enumerates each triangle once, thus it takes $O(\alpha m)$ time in the worst case. When invoking Algorithm 3 to calculate $C_B(u)$ for vertex $u$, the maps of $u$'s neighbors are updated, causing the re-calculations of the upper bound in Lemma 3 and the maintenance of the priority queue $H$ for those neighbors. Thus, the total time for updating the new bounds and $H$ can be bounded by $O(\sum_{u \in V} d(u) \log n) \leq O(m \log n)$. To sum up, the time complexity of Algorithm 2 is $O(\alpha m d_{\max} + m \log n)$. Similar to that of Algorithm 1, the space overhead of Algorithm 2 is dominated by the size of the map structure $S_u$ which is $O(\sum_{u \in V} d(u)^2) \leq O(d_{\max} m)$. □

Note that compared to Algorithm 1, the time complexity of Algorithm 2 has an additional term $O(m \log n)$. However, such an additional term is often dominated by $O(\alpha m d_{\max})$, because $O(\log n)$ is often smaller than $O(\alpha d_{\max})$ in real-world graphs. As shown in our experiments, Algorithm 2 is much more efficient than Algorithm 1 on real-world graphs due to the dynamic and tight upper bound.

## IV. The Update Algorithms

Real-world networks are often frequently updated. In this section, we develop local update algorithms to maintain the ego-betweennesses for all vertices when the graph is updated. We also propose lazy update techniques to efficiently maintain the top-$k$ results. We mainly focus on the cases of edge insertion and deletion, as vertex insertion and deletion can be seen as a series of edge insertions and deletions.

Our update algorithms are based on the following key observation.

*Observation 1:* After inserting/deleting an edge $(u, v)$ into/from $G$, the ego-betweennesses of the vertices in

**Algorithm 4:** LocalInsert

```
   Input: $G = (V, E)$, ego-betweenness array $C_B$, an inserted edge $(u, v)$.
   Output: the updated $\overline{C}_B$.
1  Insert $(u, v)$ into $G$;
2  $\mathcal{S} \leftarrow$ LocalUptSMap($G, (u, v)$);
3  $L \leftarrow N(u) \cap N(v)$; $\overline{C}_B \leftarrow C_B$;
4  for $(x, y) \in S_u$ do
5      if $x = v$ or $y = v$ then
6          $\overline{C}_B(u) \leftarrow \overline{C}_B(u) + 1/(S_u(x, y) + 1)$;
7      else
8          $\overline{C}_B(u) \leftarrow \overline{C}_B(u) + 1/(S_u(x, y) + 1) - 1/S_u(x, y)$;
9  Update $\overline{C}_B(v)$ as lines 4-8;
10 for $x \in L$ do
11     for $(y, z) \in S_x$ do
12         if ($y = u$ and $z = v$) or ($y = v$ and $z = u$) then
13             $\overline{C}_B(x) \leftarrow \overline{C}_B(x) - 1/(S_x(y, z) + 1)$;
14         else
15             $\overline{C}_B(x) \leftarrow \overline{C}_B(x) + 1/(S_x(y, z) + 1) - 1/S_x(y, z)$;
16 return $\overline{C}_B$;
```

$N(u,v) \cup \{u,v\}$ need to be updated, and the ego-betweennesses of the vertices that are not in $N(u,v) \cup \{u,v\}$ remain unchanged.

*Proof:* Here, we prove the edge insertion case and the proof for edge deletion is similar. The insertion of $(u,v)$ causes the insertions of vertex $v/u$ and a series of edges $\{(v,w)|w \in N(u,v)\}/\{(u,w)|w \in N(u,v)\}$ into $u/v$'s ego network $G_{E(u)}/G_{E(v)}$, thus the ego-betweennesses of $u$ and $v$ need to be updated. In addition, for a common neighbor $w \in N(u,v)$, there is a new edge $(u,v)$ in $G_{E(w)}$, thus the ego-betweenness of $w$ should be re-computed. □

### A. Local-update for edge insertion

We present the update rules for the vertices $u$, $v$ and $w \in N(u,v)$ when inserting an edge $(u,v)$. For brevity, let $L = N(u,v)$ denote the common neighbors of $u$ and $v$ and $S_u(x,y)$ be the number of vertices that link $x$ and $y$ but does not include $u$. Unless otherwise specified, $S_u(x,y)$ represents the value after inserting the edge. Denote by $\overline{C}_B(u)$ the ego-betweenness of $u$ after an edge insertion and $C_B(u)$ is the value before inserting an edge.

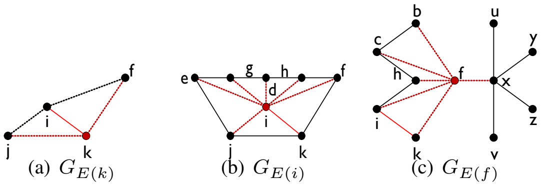
*Lemma 4:* Consider an inserted edge $(u,v)$, the updated ego-betweenness of $u$ is: $\overline{C}_B(u) = C_B(u) + \sum_{x,y \in L,(x,y) \notin E}(1/(S_u(x,y)+1) - 1/S_u(x,y)) + \sum_{x \in N(u),x \notin L} 1/(S_u(v,x)+1)$. The calculation of $\overline{C}_B(v)$ is similar.

*Proof:* For vertex $u$, after inserting an edge $(u,v)$ into $G$, $v$ is a new neighbor and is added into $G_E(u)$. For $x, y \in L$ and $(x,y) \notin E$, $C_B(u)$ has included the contribution of vertex pair $(x,y)$, thus we should update this part. $v$ is a new vertex that connects $x$ and $y$, and the number of the shortest paths between $x$ and $y$ only adds 1, thus we can calculate $S_u(x,y)$ and reveal the previous contribution to calculate $\overline{C}_B(u)$, i.e., $\overline{C}_B(u) = C_B(u) + 1/(S_u(x,y)+1) - 1/S_u(x,y)$. In addition, for $x \in L$, $x$ and $v$ are connected, thus it does not contribute to $\overline{C}_B(u)$. For $x \notin L$, $(v,x)$ is a new vertex pair which makes $u$'s ego-betweenness increase, thus we need to compute $S_u(v,x)$ and calculate $\overline{C}_B(u)$ by adding $1/(S_u(v,x)+1)$. □

*Lemma 5:* Consider an inserted edge $(u,v)$, the updated ego-betweenness of $w \in L$ is: $\overline{C}_B(w) = C_B(w) - 1/(S_w(u,v)+1) + \sum_{x \in N(w) \cap N(u)-\{v\},(x,v) \notin E}(1/(S_w(x,v)+1) - 1/S_w(x,v)) + \sum_{x \in N(w) \cap N(v)-\{u\},(x,u) \notin E}(1/(S_w(x,u)+1) - 1/S_w(x,u))$.

**Algorithm 5:** LocalUptSMap

```
   Input: $G = (V, E)$, an edge $(u, v)$.
   Output: The map set $\mathcal{S}$.
1  $L \leftarrow N(u) \cap N(v)$;
2  for $x \in N(u) \backslash L$ do $S_u$.insert($(x, v), 0$);
3  for $x \in N(v) \backslash L$ do $S_v$.insert($(x, u), 0$);
4  for $x \in L$ do $S_x$.insert($(u, v), 0$);
5  for $p \in L$ do
6      for $x \in N(u) \cap N(p)$ do
7          if $(x, v) \notin E$ and $x \neq v$ then
8              $S_u(x, v)$++;
9              for $y \in N(x) \cap N(v) \cap N(p)$ do
10                 if $\nexists S_p(x, v)$ then $S_p$.insert($(x, v), 0$);
11                 $S_p(x, v)$++;
12     Update $S_v$, $S_p$ as lines 6-11;
13     for $q \in L$ do
14         if $(p, q) \notin E$ and $q \prec p$ then
15             for $y \in N(u) \cap N(p) \cap N(q)$ do
16                 if $\nexists S_u(p, q)$ then $S_u.insert((p, q), 0)$;
17                 $S_u(p, q)$++;
18             Update $S_v$ as lines 15-17;
19         if $(p, q) \in E$ and $p \prec q$ then
20             $S_p(u, v)$++; $S_q(u, v)$++;
21 $\mathcal{S} \leftarrow \{S_x | x \in L \cup \{u, v\}\}$;
22 return $\mathcal{S}$;
```



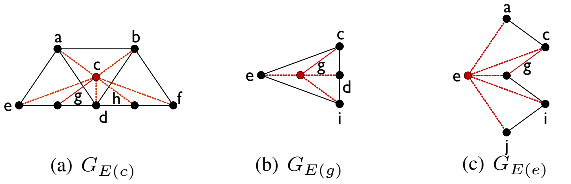
(a) $G_{E(k)}$ (b) $G_{E(i)}$ (c) $G_{E(f)}$

Fig. 4. Running example

*Proof:* For vertex $w \in L$, the insertion of $(u,v)$ causes the direct connection between $u$ and $v$ in $G_E(w)$ which makes $w$'s ego-betweenness decrease. We need to compute $S_w(u,v)$ before the insert operation and calculate $\overline{C}_B(w)$ as: $\overline{C}_B(w) = C_B(w) - 1/(S_w(u,v)+1)$. In addition, for $x \in N(w) \cap N(v)$ and $(u,x) \notin E$, $v$ now is a new vertex that connects $u$ and $x$, thus we calculate $S_w(u,x)$ and $\overline{C}_B(w)$ as: $\overline{C}_B(w) = C_B(w) + 1/(S_w(u,x)+1) - 1/S_w(u,x)$. Analogously, for $x \in N(w) \cap N(u)$ and $(v,x) \notin E$, $u$ is a new vertex that links $v$ and $x$, we calculate $\overline{C}_B(w)$ as the above operation. □

Equipped with the above lemmas, we propose a local update algorithm, called LocalInsert, to maintain the ego-betweennesses for handling edge insertion. The pseudo-code of LocalInsert is illustrated in Algorithm 4. LocalInsert first inserts the edge $(u,v)$ into $G$ (line 1). Then, it invokes the LocalUptSMap (Algorithm 5) to recompute the number of shortest paths of the affected vertex pairs in the ego networks of $u, v$ and their common neighbors (Observation 1). Finally, LocalInsert updates the ego-betweennesses for affected vertices. For the endpoints $u, v$ of the inserted edge, we calculate $\overline{C}_B(u)$ and $\overline{C}_B(v)$ based on Lemma 4 (lines 4-9); On the other hand, for the common neighbor $w$, LocalInsert computes $\overline{C}_B(w)$ according to Lemma 5 (lines 10-15).

*Example 5:* Reconsider the graph $G$ in Fig. 1(a). Suppose that we insert an edge $(i,k)$ into $G$. Clearly, the ego-betweennesses of $i, k$ and their common neighbors change based on Observation 1. Fig. 4(a) and Fig. 4(b) depict the ego networks of $k$ and $i$, respectively. In Fig. 4(a), the new pairs, i.e., $(f,i)$ and $(j,i)$, are generated due to the connection of $i$ and $k$, thus $C_B(k)$ changes. According to Lemma 4, the new $\overline{C}_B(k)$ is $\overline{C}_B(k) = C_B(k) + 1/(S_k(f,j)+1) - 1/S_k(f,j) = 1 + 1/(1+1) - 1/1 = 1/2$. Similarly, we can easily check



(a) $G_{E(c)}$ (b) $G_{E(g)}$ (c) $G_{E(e)}$

Fig. 5. Running example

that the updated ego-betweenness of $i$ is $\overline{C}_B(i) = 10.5$ from $G_{E(i)}$. For the common neighbor $f$, its ego network $G_{E(f)}$ is shown in Fig. 4(c). After the insertion of $(i,k)$, the $\overline{C}_B(f)$ is 9.5 which equals 11 before. This is because $i$ is now a neighbor of $k$ and they no longer need intermediate vertices to reach each other. In addition, the shortest paths for some vertex pairs may pass through $i$ or $j$ which causes the number of shortest paths to increase, thus making the ego-betweenness of $f$ decrease. □

### B. Local update for edge deletion

Here we consider the case of deleting an edge $(u,v)$ from $G$. When $(u,v)$ is deleted, only the vertices in $L \cup \{u,v\}$ need to update their ego-betweennesses according to Observation 1. Below we introduce the update rules for $u$, $v$ and $w \in L$. Since the proofs of the following lemmas are similar to that of Lemma 4 and Lemma 5, we omit them due to the space limitation. Like edge insertion, we use $C_B(u)$ and $\overline{C}_B(u)$ to denote the ego-betweenness of $u$ before and after deleting an edge respectively.

*Lemma 6:* Consider a deleted edge $(u,v)$, the updated ego-betweenness of $u$ is: $\overline{C}_B(u) = C_B(u) + \sum_{x,y\in L,(x,y)\notin E}(1/S_u(x,y) - 1/(S_u(x,y)+1)) - \sum_{x\in N(u),x\notin L} 1/(S_u(v,x)+1)$. The case of updating $\overline{C}_B(v)$ is similar.

*Lemma 7:* Consider a deleted edge $(u,v)$, the updated ego-betweenness of $w \in L$ is: $\overline{C}_B(w) = C_B(w) + 1/(S_w(u,v)+1) + \sum_{x\in N(w)\cap N(u)-\{v\},(x,v)\notin E}(1/S_w(x,v) - 1/(S_w(x,v)+1)) + \sum_{x\in N(w)\cap N(v)-\{u\},(x,u)\notin E}(1/S_w(x,u) - 1/(S_w(x,u)+1))$.

Note that $S_u(x,y)$ in Lemma 6 and Lemma 7 represents the value before deleting the edge. In particular, $S_w(u,v)$ in Lemma 7 is the value after the deleting update. Based on these lemmas, we present a local update algorithm, called LocalDelete, to maintain the ego-betweennesses when an edge $(u,v)$ is deleted. The framework of LocalDelete is similar to that of LocalInsert. We only need to make the following minor changes. For $u$ and $v$, LocalDelete modifies line 6 and line 8 of Algorithm 4 to $\overline{C}_B(u) \leftarrow \overline{C}_B(u) - 1/(S_u(x,y)+1)$ and $\overline{C}_B(u) \leftarrow \overline{C}_B(u) + 1/S_u(x,y) - 1/(S_u(x,y)+1)$ based on Lemma 6. According to Lemma 7, LocalDelete calculates $\overline{C}_B(x)$ for a common neighbor $x$ as $\overline{C}_B(x) \leftarrow \overline{C}_B(x) + 1/(S_x(y,z)+1)$ and $\overline{C}_B(x) \leftarrow \overline{C}_B(x) + 1/S_x(y,z) - 1/(S_x(y,z)+1)$ corresponding to line 13 and line 15 of Algorithm 4. Note that LocalDelete first performs LocalUptSMap (Algorithm 5) before deleting $(u,v)$ and then updates the ego-betweennesses of the affected vertices. Finally, it removes $(u,v)$ from $G$ and terminates. We omit the pseudo-code of LocalDelete due to the space limit.

*Example 6:* Reconsider the graph $G$ in Fig. 1(a). Suppose that we delete an edge $(c,g)$ from $G$. The ego networks of $c, g$ and their common neighbors change; further their ego-betweennesses need to be updated. For vertices $c$ and $g$, their ego networks are depicted in Fig. 5(a) and Fig. 5(b). Since $c$ and $g$ are disconnected, the pair $(c,i)$ in Fig. 5(b) no longer exists and the number of shortest paths for the vertex pair $(e,d)$ changes. According to Lemma 6, $\overline{C}_B(g)$ should be calculated as $\overline{C}_B(g) = C_B(g) - 1/(S_g(c,i)+1) + 1/S_g(e,d) - 1/(S_g(e,d)+1) = 2/3 - 1/3 + 1/2 - 1/3 = 1/2$. Analogously, the $\overline{C}_B(c)$ is 55/6 which can be easily checked from Fig. 5(a). For the common neighbor $e$, its ego network $G_{E(e)}$ is shown in Fig. 5(c). After deleting $(c,g)$, $\overline{C}_B(e)$ is still equal to 4.5 according to Lemma 7. □

### C. Updating the top-$k$ results

Here we present lazy update techniques to maintain the top-$k$ results when the graph is updated. The lazy-update techniques for edge insertion and edge deletion are designed by maintaining a sorted list of the vertices. Specifically, the sorted list, denoted by $H$, contains all vertices in $G$. For each vertex $u$ in $H$, $u$ associates with two variables, namely, $H(u).C_B$ and $H(u).F_G$, which represent the ego-betweenness $C_B(u)$ and the update state of $u$. If $H(u).F_G$ equals true, that means $H(u).C_B$ is not the exact value and should be re-calculated. Otherwise, $H(u).C_B$ is accurate. We calculate $H(u).C_B$ for each vertex $u$ and initialize $H(u).F_G$ as false, and then sort all vertices in non-increasing order of their ego-betweennesses to obtain $H$. Equipped with $H$, the lazy update techniques for edge insertion and edge deletion are as follows.

**Lazy update for edge insertion.** Consider an insertion edge $(u,v)$ and a common neighbor $w \in N(u)\cap N(v)$. The calculations of $\overline{C}_B(u)$, $\overline{C}_B(v)$ and $\overline{C}_B(w)$ are described in Lemma 4 and Lemma 5, respectively. Obviously, $S_*(*,*)+1 > S_*(*,*)$ holds, thus we have $1/(S_{u*}(*,*)+1) < 1/S_*(*,*)$. For vertex $w$, the parts $\sum_{x\in N(w)\cap N(u)-\{v\},(x,v)\notin E}(1/(S_w(x,v)+1) - 1/S_w(x,v))$ and $\sum_{x\in N(w)\cap N(v)-\{u\},(x,u)\notin E}(1/(S_w(x,u)+1) - 1/S_w(x,u))$ are both less than 0, and $1/(S_w(u,v)+1)$ is subtracted from $C_B(w)$, thus $C_B(w)$ tends to decrease. However, for vertex $u$ (as well as $v$), the part $\sum_{x\in N(u),x\notin L} 1/(S_u(v,x)+1)$ increases, but the part $\sum_{x,y\in L,(x,y)\notin E}(1/(S_u(x,y)+1) - 1/S_u(x,y))$ decreases, thus the changes of $C_B(u)$ and $C_B(v)$ are unclear. Nevertheless, an interesting finding is that with the insertion operation, the degrees of $u$ and $v$ increase and the upper bounds of $C_B(u)$ and $C_B(v)$ also increase. Based on these findings, we can implement a lazy update rule to maintain the top-$k$ results for edge insertion.

The lazy update algorithm to handle edge insertion, called LazyInsert, is shown in Algorithm 6. For the endpoint $u$ of the inserted edge, LazyInsert first identifies whether $u$ is included in the top-$k$ result set $R$. If $u \in R$, it calculates the ego-betweenness $H(u).C_B$ and sets $H(u).F_G$ to false to indicate the correctness of $H(u).C_B$. As $H(u).C_B$ is updated, we need to determine whether $u$ still belongs to $R$. If $H(u).C_B >= \min_{p\in R\setminus\{u\}} C_B(p)$ holds, $u$ is still included in the top-$k$ result set $R$. On the other hand, LazyInsert compares $H(u).C_B$ with the ego-betweenness of the $(k+1)$-th element in the sorted list $H$ (lines 4-8). Let $y$ denote the $(k+1)$-th vertex in $H$. If $H(y).F_G$ is false, that means $y$ is the vertex with the highest ego-betweenness that is not contained in $R$. LazyInsert compares $H(u).C_B$ with $H(y).C_B$ and maintains $R$ (lines 6-7). If $H(y).F_G = true$ holds, LazyInsert computes $H(y).C_B$ and updates $H(y).F_G$, and then performs the next loop (line 8). While $u \notin R$, we derive the new upper bound $\mathsf{ub}(u)$ of $H(u).C_B$ to determine whether $H(u).C_B$ needs to be computed exactly (lines 10-16). If $\mathsf{ub}(u) <= \min_{p\in R} C_B(p)$ holds, it means that $H(u).C_B$ is not greater than $\min_{p\in R} C_B(p)$, thus $u$ is still not an answer of the top-$k$ results and LazyInsert can avoid calculating the

**Algorithm 6:** LazyInsert

```
   Input: $G = (V, E)$, $H$, an inserted edge $(u, v)$, top-$k$ result set $R$.
   Output: the updated $R$.
1  if $u \in R$ then
2      Compute $H(u).C_B$; $H(u).F_G \leftarrow false$;
3      if $H(u).C_B < \min_{p \in R \setminus \{u\}} C_B(p)$ then
4          while $true$ do
5              $y \leftarrow \arg\max_{p \in H-R} H(p).C_B$;
6              if $H(y).F_G = false$ and $H(u).C_B < H(y).C_B$ then
7                  $R \leftarrow (R - \{u\}) \cup \{y\}$; break;
8              else Compute $H(y).C_B$; $H(y).F_G \leftarrow false$;
9  else
10     $\overline{\mathsf{ub}}(u) \leftarrow \frac{d(u)*(d(u)-1)}{2}$;
11     if $\overline{\mathsf{ub}}(u) > \min_{p \in R} C_B(p)$ then
12         Compute $H(u).C_B$; $H(u).F_G \leftarrow false$;
13         if $H(u).C_B > \min_{p \in R} C_B(p)$ then
14             $y \leftarrow \arg\min_{p \in R} C_B(p)$;
15             $R \leftarrow (R - \{y\}) \cup \{u\}$;
16     else $H(u).F_G \leftarrow true$;
17 Update $H$ and $R$ according to $v$ as lines 1-16;
18 $L \leftarrow N(u) \cap N(v)$;
19 for $x \in L$ do
20     if $x \in R$ then Update $H$ and $R$ according to $x$ as lines 2-8;
21     else $H(x).F_G \leftarrow true$;
22 return $R$;
```

correct $H(u).C_B$ and only updates $H(u).F_G$ to true (line 16). Otherwise, LazyInsert calculates $H(u).C_B$ and identifies whether $u$ should be inserted into $R$ (lines 12-15). Likewise, we perform the same operation for the other endpoint $v$ (line 17). Then, LazyInsert handles the common neighbors of $u$ and $v$ (lines 18-21). For vertex $x \in L$, the algorithm judges whether $x$ is included in $R$. If yes, it updates $H$ and $R$ as the operations of $u$ (line 20). On the other hand, because $C_B(x)$ is decreasing, $x$ is still not in the top-$k$ result set and thus LazyInsert avoids computing the exact $H(x).C_B$ and only sets $H(x).F_G$ to true (line 21). Note that the ego-betweennesses of the vertices in $R$ are always correct. Finally, LazyInsert returns the top-$k$ vertices with the highest ego-betweennesses correctly.

*Example 7:* Reconsider the graph $G$ in Fig. 1(a). Before inserting the edge $(i, k)$ into $G$, we have $C_B(i) = 8$ and $C_B(k) = 1$. After the insertion, $C_B(i)$ is equal to 10.5 and $C_B(k)$ is 0.5. For the common neighbor $f$, $C_B(f)$ decreases from 11 to 9.5. Clearly, the change of the ego-betweennesses for the ends of the insertion edge is uncertain while it is decreasing for the common neighbors. Suppose that $k = 1$ and the current result set is $R = \{f\}$. For vertex $k$, it is not included in $R$ and its new bound is $(3 * 2)/2 = 3 < C_B(f) = 11$, thus the calculation of $C_B(k)$ can be skipped and we only set $H(k).FG$ to true. For vertex $i$, the new bound is $(7 * 6)/2 = 21 > C_B(f) = 11$, thus we need to calculate the new $C_B(i) = 10.5$ and update $R = \{i\}$. In this case, consider the common neighbor $f$, it is not included in $R$. Since $C_B(f)$ is not incremental, it definitely not in the top-1 result after inserting $(i, k)$, thus we can avoid calculating $C_B(f)$ and updating the results $R$. □

Below, we analyze the correctness of Algorithm 6.

*Theorem 4:* Given a graph $G = (V, E)$, $H$ and the top-$k$ results $R$, Algorithm 6 correctly maintains the set $R$ when inserting an edge $(u, v)$.

*Proof:* As shown in Lemma 5, the ego-betweenness of a common neighbor $w$ tends to decrease. Thus, if $w$ is not a top-$k$ answer, Algorithm 6 avoids calculating $\overline{C}_B(w)$ because $\overline{C}_B(w) < C_B(w) \leq \min_{p \in R} C_B(p)$ holds. Otherwise, Algorithm 6 computes $\overline{C}_B(w)$ to determine whether it is still in the top-$k$ results. For the end-vertices of the inserted edge $(u, v)$, we show the correctness by taking $u$ as an example. If $u$ is a top-$k$ answer, Algorithm 6 computes $\overline{C}_B(u)$ and compares it with the minimal ego-betweenness in $R$ to determine whether $u$ still belongs to $R$. Otherwise, with the edge insertion, the degree of $u$ increases and the algorithm derives a new upper bound $\overline{\mathsf{ub}}(u) =$ for $\overline{C}_B(u)$. If $\overline{\mathsf{ub}}(u) \leq \min_{p \in R} C_B(p)$ holds, we have $\overline{C}_B(u) \leq \overline{\mathsf{ub}}(u) \leq \min_{p \in R} C_B(p)$, thus $u$ is still not in the top-$k$ results and can be safely pruned. On the other hand, Algorithm 6 calculates $\overline{C}_B(u)$ and identifies whether $u$ should be inserted into $R$. Therefore, the algorithm can correctly maintain the top-$k$ vertices. □

**Lazy update for edge deletion.** Consider the deletion edge $(u, v)$ and a common neighbor $w \in N(u) \cap N(v)$. Like the edge insertion, the changes of $C_B(u)$, $C_B(v)$ and $C_B(w)$ are as follows. $C_B(w)$ is definitely non-decreasing while $C_B(u)$ and $C_B(v)$ are uncertain. Fortunately, after deleting $(u, v)$, the degrees of $u$ and $v$ decrease and also the upper bounds of $C_B(u)$ and $C_B(v)$ decrease. Based on this, we can implement a lazy update algorithm which is very similar to edge insertion.

Our lazy update algorithm for handling edge deletion, called LazyDelete, can be easily devised by slightly modifying Algorithm 6. Like lines 14-15 of Algorithm 6, LazyDelete needs to find the vertex $y$ with the lowest ego-betweenness in the top-$k$ results. Armed with our lazy update technique, the ego-betweennesses of the vertices in $R$ are not all correct, thus LazyDelete must find $y \in R$ with the lowest ego-betweenness and $H(y).F_G = false$. The other steps of LazyDelete are similar to those of LazyInsert. Due to the space limit, the pseudo-code of LazyDelete is omitted.

*Example 8:* Let us still consider the graph $G$ in Fig. 1(a). Before deleting the edge $(c, g)$ from $G$, we have $C_B(c) = 41/6$, $C_B(g) = 2/3$ and $C_B(e) = 9/2$. After the deletion, the new ego-betweennesses for $c$, $g$, and $e$ are $55/6, 1/2, 9/2$, respectively. Obviously, the change of the ego-betweennesses for the ends of the deletion edge is uncertain while it is non-decreasing for the common neighbors. Suppose that $k = 1$ and we can check that the current $R = \{f\}$. For vertex $g$, its new bound is equal to $(3 * 2)/2 = 3 < C_B(f) = 11$ and $g \notin R$, thus we do not need to calculate the new ego-betweenness for $g$ and only set $H(g).FG$ to true. For vertex $c$, its new bound is $(6 * 5)/2 = 15 > C_B(f) = 11$, thus we calculate the new $C_B(c) = 55/6$ and the top-1 answer is still $f$. When $k = 12$, the top-$k$ results before deleting the edge $(c, g)$ is the set $V - \{u, v, y, z\}$. In this case, the common neighbor $e$ is included in $R$. Since $C_B(e)$ is non-decreasing after deleting $(c, g)$ , it is definitely still contained in the top-12 results, thus we can avoid updating the answer set $R$. □

## V. The parallel algorithms

### A. A vertex-based parallel algorithm

The ego-betweenness of a vertex is defined on its ego network which can be calculated independently, thus a straightforward parallel solution is to process each vertex in parallel. However, such a simple solution may be inefficient, especially for large graphs. When processing each vertex independently, we need to construct its ego network and explore the *diamond* structures (a diamond denotes two triangles that have a common edge), which makes the same *diamond* enumerated multiple times, resulting in repetitive calculations. To solve this problem, we propose a vertex-based parallel algorithm as follows.

As can be seen from Algorithm 1 and Algorithm 2, we explore the *diamond* structures by searching triangles, thus we can employ a parallel triangle enumeration to calculate the ego-betweennesses for all vertices. The main idea is that

TABLE I
DATASETS

| Dataset | $n$ | $m$ | $d_{\max}$ | Desecription |
|---|---|---|---|---|
| WikiTalk | 2,394,385 | 4,659,565 | 100,029 | Communication network |
| DBLP | 1,843,617 | 8,350,260 | 2,213 | Collaboration network |
| Pokec | 1,632,803 | 22,301,964 | 14,854 | Social network |
| LiveJournal | 3,997,962 | 34,681,189 | 14,815 | Social network |
| Sina | 58,655,848 | 261,321,033 | 278,489 | Social network |

every triangle in $G$ has a unique orientation based on the total ordering, and only be enumerated when processing the highest-ranked vertex in this triangle. When a triangle $\triangle_{(u,v,w)}$ is found, we utilize it to explore *diamond*s and maintain $S_u$, $S_v$, and $S_w$ which record the number of shortest paths between their neighbors. Note that we should lock the map $S$ when it is updated to ensure the correctness of the parallel algorithm. To avoid frequent locking operations, we employ the idea of Algorithm 2 to divide the neighbors of a vertex into the in-neighbors and out-neighbors for delaying the updates of $S$, which can also search a triangle once. As all triangles are enumerated, that is, the information of the number of shortest paths is correctly maintained in the maps, we calculate the ego-betweenness for each vertex in parallel according to Lemma 2. We refer to this parallel implementation as VertexPEBW and omit the pseudo-code due to the space limit.

### B. An edge-based parallel algorithm

In practice, VertexPEBW might still 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 triangles 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 parallelism of the algorithm. We refer to such an edge-parallel algorithm as EdgePEBW. In the experiments, we will compare the efficiency of VertexPEBW and EdgePEBW.

## VI. EXPERIMENTS

In this section, we conduct extensive experiments to evaluate the efficiency and effectiveness of the proposed algorithms. We implement the proposed top-$k$ ego-betweenness search algorithms, namely, BaseBSearch and OptBSearch (Algorithm 1 and Algorithm 2). For comparison, we implement a baseline algorithm proposed in [7], called Baseline, which computes the ego-betweenness based on a matrix multiplication technique. Note that it is very hard to integrate our upper-bounding techniques into the matrix multiplication procedure, thus Baseline first computes the ego-betweenness for all vertices using the matrix multiplication technique, and then picks the top-$k$ vertices. To maintain the ego-betweennesses for all vertices, we implement LocalInsert (Algorithm 4) and LocalDelete for handling edge insertion and edge deletion, respectively. We also implement LazyInsert (Algorithm 6) and LazyDelete to maintain the top-$k$ results for edge insertion and edge deletion, respectively. In addition, we implement two parallel algorithms, VertexPEBW and EdgePEBW, to calculate ego-betweennesses of all vertices using OpenMP. All algorithms are implemented in C++. All experiments are conducted on a PC with 2.10GHz CPU and 256GB memory running Red Hat 4.8.5. We set the time limit for all algorithms to 3 days, and use the symbol "INF" to denote that the algorithm cannot terminate within 3 days.

**Other related algorithms.** Note that this work focuses on the ego-betweenness computation problem, thus we mainly compare the performance of different ego-betweenness computation algorithms in efficiency testings, i.e., the Baseline and the proposed algorithms. All the existing betweenness computation methods, such as [6], [22], [23], are precluded from our comparison in efficiency testings. For effectiveness testings, we aim to observe whether the top-$k$ ego-betweenness vertices are similar to the top-$k$ betweenness vertices. To compute the top-$k$ betweenness vertices, we make use of the state-of-the-art Brandes' algorithm [6] to calculate the exact betweenness for each vertex and then identify the top-$k$ vertices with the highest betweennesses. In addition, we also use the sampling-based approximation betweenness algorithm [24], [25] to compute the top-$k$ vertices for comparison. For brevity, we refer to the exact Brandes' algorithm as TopBW, the approximation betweenness algorithm as TopABW, and our OptBSearch algorithm as TopEBW. The top-$k$ results yielded by TopBW, TopABW and TopEBW are denoted as BW, ABW and EBW respectively.

TABLE II
THE NUMBER OF VERTICES FOR EXACT COMPUTATION

| Dataset | $k = 500$ | | $k = 1000$ | | $k = 2000$ | |
|---|---|---|---|---|---|---|
| | BaseBS | OptBS | BaseBS | OptBS | BaseBS | OptBS |
| WikiTalk | 527 | **508** | 1052 | **1013** | 2098 | **2013** |
| DBLP | 557 | **550** | 1499 | **1160** | 3060 | **2491** |
| Pokec | 567 | **552** | 1230 | **1168** | 2498 | **2367** |
| LiveJournal | 791 | **615** | 1723 | **1282** | 3406 | **2413** |
| Sina | - | **500** | - | **1000** | - | **2000** |

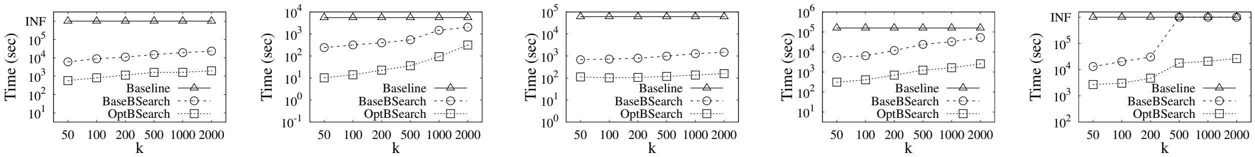
**Datasets.** We use 5 different types of real-life networks in the experiments, including social networks, communication networks and collaboration networks. The detailed statistics of the datasets are summarized in Table I. In Table I, $d_{\max}$ denotes the maximum degree of the graph. The dataset Sina is downloaded from https://networkrepository.com/soc.php, and the others are downloaded from snap.stanford.edu.

**Parameters.** The parameter $k$ in our algorithms is chosen from the set $\{50, 100, 200, 500, 1000, 2000\}$ with a default value of $k = 500$. The parameter $\theta$ in OptBSearch is selected from the set $\{1.05, 1.10, 1.15, 1.20, 1.25, 1.30\}$ with a default value 1.05. We will study the performance of our algorithms with varying $k$ and $\theta$. Unless otherwise specified, the value of a parameter is set to its default value when varying another parameter.

### A. Efficiency testing

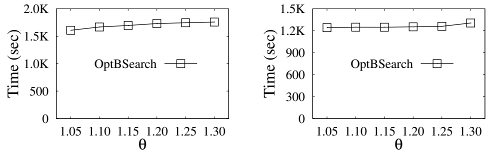
**Exp-1: Runtime of different algorithms.** Fig. 6 shows the runtime of Baseline, BaseBSearch and OptBSearch with varying $k$ on all datasets. As can be seen, BaseBSearch and OptBSearch are around 3-90 times and at least one order of magnitude faster than Baseline, and OptBSearch is roughly 5-23 times faster than BaseBSearch within all parameter settings. For example, on DBLP, BaseBSearch and OptBSearch take 240.482 seconds and 10.198 seconds to retrieve the top-50 results, while Baseline consumes 5527.398 seconds which is roughly 23 times and 540 times slower than BaseBSearch and OptBSearch. When $k = 200$ on LiveJournal, the runtime of Baseline, BaseBSearch and OptBSearch is 158,253.133 seconds, 11,858.172 seconds, and 702.529 seconds, respectively. This is because the Baseline needs to compute all vertices' ego-betweennesses to select the top-$k$ results, while our BaseBSearch and OptBSearch equipped with upper bounds only require to calculate the ego-betweennesses of a part of vertices. Moreover, the dynamic upper bound in OptBSearch is tighter than the static upper bound in BaseBSearch, thus it is more effective to prune unpromising vertices that are definitely not contained in the top-$k$ results.

In addition, we also record the number of vertices whose ego-betweennesses are computed exactly in BaseBSearch and OptBSearch. For brevity, we refer to BaseBSearch and OptBSearch as BaseBS and OptBS. Table II illustrates the



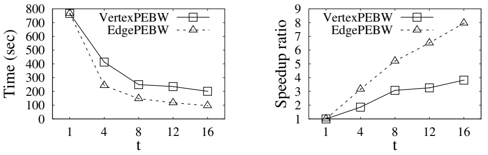
(a) WikiTalk (vary $k$) (b) DBLP (vary $k$) (c) Pokec (vary $k$) (d) LiveJournal (vary $k$) (e) Sina (vary $k$)

Fig. 6. Comparisons of Baseline, BaseBSearch and OptBSearch on various datasets



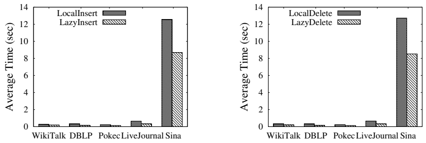
(a) WikiTalk (vary $\theta$) (b) LiveJournal (vary $\theta$)

Fig. 7. Evaluation of OptBSearch with varying $\theta$



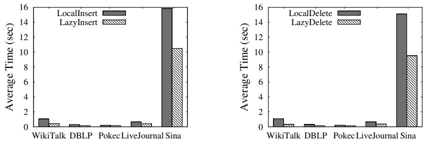
(a) Pokec (b) Pokec

Fig. 11. Evaluation of the parallel algorithms



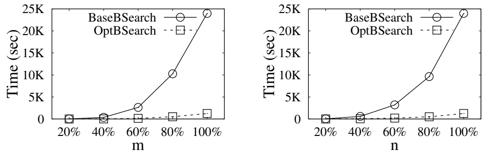
(a) edge insertion (b) edge deletion

Fig. 8. Average runtime of the algorithms for random edge updates



(a) edge insertion (b) edge deletion

Fig. 9. Average runtime of the algorithms for adversarial edge updates



(a) LiveJournal (vary $m$) (b) LiveJournal (vary $n$)

Fig. 10. Scalability of BaseBSearch and OptBSearch

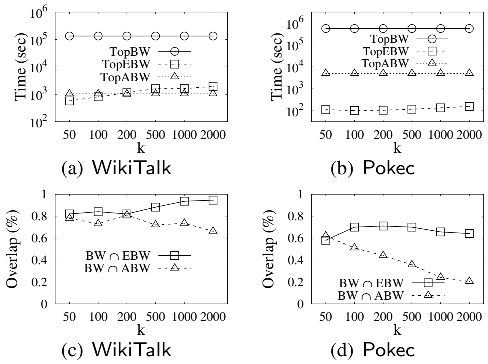
results of $k = 500, 1000, 2000$ on all datasets. Similar results can be observed for other $k$ values. As can be seen, the number of vertices computed by OptBSearch is significantly less than that computed by BaseBSearch on all datasets. For example, to obtain the top-2000 results on LiveJournal, OptBSearch only needs to compute the ego-betweennesses for 2,413 vertices, while BaseBSearch has to compute 3,406 vertices. These results further confirm our theoretical analysis in Section III.

**Exp-2: The effect of $\theta$.** Fig. 7 reports the effect of parameter $\theta$ in OptBSearch on WikiTalk and LiveJournal. The results on the other datasets are consistent. As can be seen, the runtime of OptBSearch varies slightly with different $\theta$ values. In general, OptBSearch performs slightly better with a relatively small $\theta$. For example, with $\theta = 1.05$, OptBSearch consumes the lowest runtime on both WikiTalk and LiveJournal. Note that a large $\theta$ may increase the cost of computing the exact ego-betweennesses, while a small $\theta$ may increase the cost of updating the upper bounds in $H$. These results indicate that when $\theta = 1.05$, OptBSearch can achieve a good tradeoff between these two costs.

**Exp-3: Evaluation of the updating algorithms.** To evaluate the performance of our updating algorithms, we randomly select 1,000 edges for insertion and deletion on each dataset, which we call random edge updates. We also randomly generate 1,000 edges whose end-vertices are the vertices in the top-$k$ results on each dataset for comparison, which we call adversarial edge updates. Fig. 8 and Fig. 9 show the average runtime of LocalInsert, LocalDelete, LazyInsert and LazyDelete on all datasets for random edge updates and adversarial edge updates, respectively. In general, the runtime of all algorithms for adversarial edge updates is slightly higher than the runtime of all algorithms for random edge updates. Moreover, the update time of LazyInsert is lower than that of LocalInsert for both random and adversarial edge updates as expected. For example, when inserting random edges, LocalInsert consumes 12.555 seconds to maintain ego-betweennesses for all vertices on Sina, while LazyInsert takes 8.690 seconds for updating the top-$k$ results. In the case of adversarial edge updates, LocalInsert and LazyInsert take 15.837 seconds and 10.497 seconds to maintain ego-betweennesses for all vertices and the top-$k$ results on Sina, respectively. Similar results can also be observed for LocalDelete and LazyDelete. In addition, the average runtime of LocalInsert (LazyInsert) and LocalDelete (LazyDelete) is almost the same. Note that the runtime of our updating algorithms is less than 16 seconds for all datasets, and even less than 1.0 seconds for relatively small datasets expect Sina. These results indicate that the proposed updating algorithms are very efficient on large real-life graphs even for adversarial edge updates.

**Exp-4: Scalability testing.** Here we evaluate the scalability of BaseBSearch and OptBSearch. To this end, we generate four subgraphs for each dataset by randomly picking 20%-80% of the edges (vertices), and evaluate the runtime of BaseBSearch and OptBSearch on these subgraphs. Fig. 10 illustrates the results on LiveJournal. The results on the other datasets are similar. As can be seen, the runtime of OptBSearch increases very smoothly with increasing $m$ or $n$, while the runtime of BaseBSearch increases more sharply. Again, we can see that OptBSearch is significantly faster than BaseBSearch with all parameter settings, which is consistent with our previous findings.

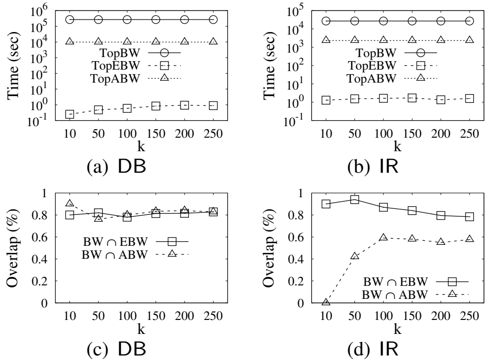
**Exp-5: Evaluation of parallel algorithms.** We vary the number of threads $t$ from 1 to 16, and evaluate two parallel algorithms, i.e., VertexPEBW and EdgePEBW, with an increasing $t$. We run OptBSearch with the parameter $k = n$ to compute ego-betweennesses as baseline for $t = 1$. Fig. 11 shows the results of runtime and speedup ratio on Pokec. From Fig. 11, we can see that both VertexPEBW and EdgePEBW achieve very good speedup ratios. The runtime of EdgePEBW



(a) WikiTalk (b) Pokec

(c) WikiTalk (d) Pokec

Fig. 12. Comparisons of TopBW, TopABW and TopEBW



(a) DB (b) IR

(c) DB (d) IR

Fig. 13. Comparisons of TopBW, TopABW and TopEBW on DBLP

is lower than VertexPEBW with all parameter settings. For example, the running time of OptBSearch to calculate ego-betweennesses for all vertices is 766.652 seconds. When $t = 16$, VertexPEBW takes 200.763 seconds and EdgePEBW consumes 96.260 seconds to compute the results. The speedup ratios of VertexPEBW and EdgePEBW are roughly equal to 4 and 8, respectively. These results indicate that our parallel algorithms are very efficient on real-life graphs.

### *B. Effectiveness testing*

**Exp-6: Comparisons of TopBW, TopABW and TopEBW.** We compare TopBW, TopABW, and TopEBW on WikiTalk and Pokec with $k \in \{50, 100, 200, 500, 1000, 2000\}$. The results on the other datasets are consistent. Note that to speed up the betweenness computation, we also implement a parallel version of TopBW for comparison. The runtime of TopBW with 64 threads, TopABW and TopEBW is shown in Fig. 12(a-b). Clearly, TopEBW is at least two orders of magnitude faster than the parallel TopBW within all parameter settings. Compared to TopABW, TopEBW is at least one order of magnitude faster on Pokec with varying $k$. Specifically, on Pokec, TopEBW takes 106.276 seconds, while TopABW and TopBW consume 5,038.103 seconds and 559,322.062 seconds to output the top-200 results. Fig. 12(c-d) report the overlap of the top-$k$ results obtained by TopBW, TopABW and TopEBW. As can be seen, the overlap between BW and EBW is generally higher than 60% on all datasets. Particularly, on WikiTalk, the overlap is even more than 80%. Moreover, we can see that the overlap between BW and EBW is generally higher than the overlap between BW and ABW. These results indicate that for approximating the top-$k$ betweenness, our algorithm is much better than the sampling-based approximation betweenness computation algorithm in terms of both runtime and accuracy.

**Remark.** The worst-case time complexity of the Brandes' algorithm, i.e., TopBW, is $O(mn)$ which calculates betweennesses for all vertices. The BaseBSearch and OptBSearch (i.e., TopEBW) consume $O(\alpha m d_{max})$ and $O(\alpha m d_{\max} + m \log n)$ time to compute all vertices' ego-betweennesses in the worst case, respectively. Since $d_{max}$ is often much smaller than $n$ and $\alpha$ is a small constant in most large real-life graphs, the worst-case time complexity of our algorithms is expected to be lower than $O(mn)$. In our experiments, we can compare the running time of TopBW and TopEBW by calculating the betweennesses and ego-betweennesses for all vertices. The running time of TopEBW and TopBW on Pokec is illustrated in Fig. 11(a) (i.e., $t = 1, k = n$) and Fig. 12(b) (the TopBW algorithm), respectively. Similar results can also be observed on the other datasets. As can be seen, TopEBW takes 766.652 seconds to obtain the ego-betweennesses for all vertices in Pokec, while TopBW consumes 559,322.062 seconds to compute all vertices' betweennesses, which is roughly three orders of magnitude slower than TopEBW. These results confirm that the practical performance of our algorithms is significantly better than that of TopBW [6] when computing the (ego) betweennesses for all vertices.

TABLE III
TOP-10 SCHOLARS IN DB

| Top-10 EBW | $d$ | $C_B$ | Top-10 BW | $d$ | $B_T$ |
|---|---|---|---|---|---|
| ***Jiawei Han** | 412 | 73,928.5 | ***Philip S. Yu** | 360 | 50,320,100 |
| ***Philip S. Yu** | 360 | 58,834.1 | ***Jiawei Han** | 412 | 50,059,900 |
| ***Christos Faloutsos** | 337 | 52,192.9 | ***Christos Faloutsos** | 337 | 46,340,200 |
| ***Jian Pei** | 215 | 20,531.1 | ***Gerhard Weikum** | 213 | 26,232,700 |
| ***Gerhard Weikum** | 213 | 19,238.3 | ***Beng Chin Ooi** | 205 | 22,376,200 |
| ***Michael J. Franklin** | 220 | 17,867.5 | ***Jian Pei** | 215 | 21,470,900 |
| Michael Stonebraker | 210 | 16,081.4 | ***Michael J. Franklin** | 220 | 20,809,000 |
| ***Raghu Ramakrishnan** | 210 | 15,930.1 | ***Raghu Ramakrishnan** | 210 | 18,481,900 |
| ***Beng Chin Ooi** | 205 | 14,848.2 | Haixun Wang | 183 | 17,062,500 |
| Hector Garcia-Molina | 197 | 14,664.8 | H. V. Jagadish | 178 | 16,144,700 |

TABLE IV
TOP-10 SCHOLARS IN IR

| Top-10 EBW | $d$ | $C_B$ | Top-10 BW | $d$ | $B_T$ |
|---|---|---|---|---|---|
| ***Jeffrey P. Bigham** | 2441 | 1.4846e+06 | ***Taesup Moon** | 2318 | 1.33948e+07 |
| ***Alex D. Wade** | 2510 | 1.46767e+06 | ***Jeffrey P. Bigham** | 2441 | 1.1711e+07 |
| ***Adam Sadilek** | 1993 | 1.30844e+06 | ***Alex D. Wade** | 2510 | 1.10161e+07 |
| ***Taesup Moon** | 2318 | 1.25722e+06 | ***Adam Sadilek** | 1993 | 9.49158e+06 |
| ***Antonio Gulli** | 1951 | 1.16136e+06 | ***Antonio Gulli** | 1951 | 9.44098e+06 |
| ***Henry A. Kautz** | 1731 | 882,981 | ***Bob Boynton** | 1618 | 7.00364e+06 |
| ***Bob Boynton** | 1618 | 844,761 | ***Henry A. Kautz** | 1731 | 6.82747e+06 |
| ***Padmini Srinivasan** | 1541 | 822,131 | Linchuan Xu | 1834 | 6.79258e+06 |
| ***Yelena Mejova** | 1210 | 580,116 | ***Padmini Srinivasan** | 1541 | 6.41121e+06 |
| Raymie Stata | 796 | 224,422 | ***Yelena Mejova** | 1210 | 5.82391e+06 |

**Exp-7: Case study on DBLP.** We extract two subgraphs, namely, DB and IR, 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 (i.e., SIGMOD, SIGKDD, SIGIR, VLDB, ICDE, PODS, KDD, ICDM, SDM, EDBT). The DB subgraph contains 37,177 vertices and 131,715 edges. The IR subgraph contains the authors who had published at least one paper in the information retrieval related conferences (i.e., SIGIR, CIKM, WSDM) with 13,445 vertices and 37,428 edges. We invoke TopBW, TopABW, and TopEBW to find the top-$k$ highest (ego-)betweennesses scholars on DB and IR with the parameter $k \in \{10, 50, 100, 150, 200, 250\}$. The results are shown in Fig. 13. Consistent with previous findings, the running time of TopEBW is significantly faster than TopBW and TopABW. Moreover, we can see that the overlap between BW and EBW is higher than 80% on both DB and IR. Although TopABW can achieve comparable effectiveness on DB, it performs much worse than our algorithm on IR. These results further confirm that our solutions are significantly better than the sampling-based betweenness approximation algorithm [24], [25] in terms of both efficiency and effectiveness.

We also illustrate the top-10 scholars on DB and IR in Table III and Table IV. In both Table III and Table IV, $d$

denotes the number of co-authors of a scholar; $C_B$ and $B_T$ denote the ego-betweenness and betweenness of a scholar respectively. Clearly, the overlaps of the top-10 results are 80% and 90% on DB and IR respectively. Moreover, we can see that the top-10 scholars with the highest ego-betweennesses are the most influential in the database, data mining, and information retrieval communities. Such scholars may play a bridge role in connecting different research groups. For example, in Table III, Professor Jiawei Han has 412 co-authors and maintains connections with many different research groups. Similarly, in Table IV, Taesup Moon is interested in diverse areas such as information retrieval, statistical machine learning, information theory, signal processing and so on, thus he plays an important role in promoting the interactions between different research communities. These results indicate that our algorithms can be used to find high influential vertices in a network that act as network bridges.

**Exp-8: Case studies on Dolphins and TrainBombing.** Here we conduct case studies on two small datasets, Dolphins and TrainBombing, to evaluate the effectiveness of our solutions. Dolphins (62 nodes and 159 edges) is a social network of dolphins living in New Zealand, and TrainBombing (64 nodes and 243 edges) contains contacts between suspected terrorists involved in the train bombing of Madrid on 2004. Both Dolphins and TrainBombing can be downloaded from https://konect.cc/networks. We invoke the TopBW and TopEBW algorithms to find the top-1 vertices with the highest betweenness and ego-betweenness on Dolphins and TrainBombing. The results are depicted in Fig. 14 in which the vertices colored blue and red are the vertices with the highest betweennesses and ego-betweennesses respectively. From Fig. 14(a), we can see that the blue vertex plays the role of a "bridge" in Dolphins which connects two different tightly-connected communities. Moreover, such a vertex has few neighbors and its ego-betweenness is not very large. While the vertex colored red with the highest ego-betweenness lies in the center of a community, indicating that it is an important vertex in Dolphins. Meanwhile, the number of neighbors associated with the red vertex is larger compared with that of the blue vertex. In Fig. 14(b), similar results can also be observed on TrainBombing. These results suggest that the semantics of ego-betweenness is different from that of betweenness. In particular, our algorithm can find vertices that act as "centers" in a community which often exhibit strong relationships with other vertices. However, the vertices with high betweennesses tend to connect different communities as "bridges" and are likely associated with weak links.

## VII. Related work

**Betweenness centrality.** Our work is closely related to betweenness centrality [6], [26]. Betweenness centrality is an important measure of centrality in a graph based on the shortest path, which has been applied to a wide range of applications in social networks [3], biological networks [4], computer networks [5], road networks [2] and so on. The best-known algorithm for betweenness computation, proposed by Brandes [6], runs $O(nm)$ time complexity for unweighted networks. Measuring the betweenness centrality scores of all vertices is notoriously expensive, thus many parallel and approximate algorithms have been developed to reduce the computation cost [27]–[31]. Fan *et al.* proposed an efficient parallel GPU-based algorithm for computing betweenness centrality in large weighted networks and integrated the work-efficient strategy to address the load-imbalance problem [27]. Furno *et al.* studied the performance of a parametric two-level clustering algorithm for computing approximate value of betweenness with an ideal speedup with respect to Brandes' algorithm [30]. In this paper, we focus on the ego-betweenness centrality which is first proposed by Everett *et al.* [7] as an approximation of betweenness centrality. Ego-betweenness centrality has gained recognition in its own right as a natural measure of a node's importance as a network bridge [32]. To the best of our knowledge, our work is the first to study the problem of finding top-$k$ ego-betweenness vertices in graphs.

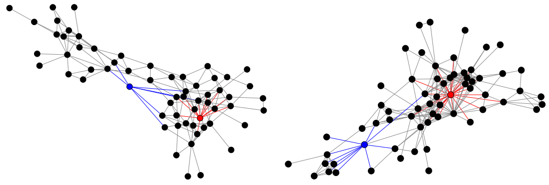


Fig. 14. Case studies on Dolphins and TrainBombing (blue vertex: the top-1 betweenness vertex; red vertex: the top-1 ego-betweenness vertex)

**Top-$k$ retrieval.** Our work is also related to the top-$k$ retrieval problem, which aims to find $k$ results with the largest scores/relevances based on a pre-defined ranking function [33]. The general framework for answering top-$k$ queries is to process the candidates according to a heuristic order and prune the search space based on some carefully-designed upper bounds. An excellent survey can be found in [33]. There are many studies on top-$k$ query processing for heterogeneous applications, such as processing distributed preference queries [34], keyword queries [35], set similarity join queries [36]. An influential algorithm was proposed by Fagin *et al.* [37], [38], which considers both random access and/or sequential access of the ranked lists. Recently, some studies take diversity into consideration in the top-$k$ retrieval in order to return diversified ranking results [39]–[43]. For instance, Li *et al.* proposed a scalable algorithm to achieve near-optimal top-$k$ diversified ranking with linear time and space complexity with respect to the graph size. Some studies have also been done which focus mainly on exploring influential communities, individuals, and relationships in different networks [13]–[15], [44], [45]. For example, the study [45] investigated an instance-optimal algorithm, which runs in linear time complexity without indexes, for computing the top-$k$ influential communities. In this paper, we develop two search frameworks to identify the top-$k$ vertices with the highest ego-betweennesses and propose efficient techniques to maintain top-$k$ results when the graph is updated.

## VIII. Conclusion

In this paper, we study a problem of finding the top-$k$ vertices in a graph with the highest ego-betweennesses. To solve this problem, we first develop two top-$k$ search frameworks with a static upper bound and a novel dynamic upper bound, respectively. Then, we propose efficient local maintenance algorithms to maintain the ego-betweenness for each vertex when an edge is inserted and deleted. We also present lazy-update techniques to maintain the top-$k$ results in dynamic graphs. We conduct extensive experiments using five real-life datasets to evaluate the proposed algorithms. The results demonstrate the efficiency, scalability and effectiveness of our algorithms. Also, the results show that the top-$k$ ego-betweenness results are highly similar to the top-$k$ betweenness results, but they are much cheaper to compute by our algorithms.

**Acknowledgement.** This work was partially supported by (i) National Key Research and Development Program of China 2020AAA0108503, (ii) NSFC Grants 62072034, U1809206, and 61772346. Guoren Wang and Rong-Hua Li are the corresponding authors of this paper.

# References

[1] L. C. Freeman, “A set of measures of centrality based on betweenness,” *Sociometry*, pp. 35–41, 1977.

[2] M. E. Newman, “The mathematics of networks,” *The new palgrave encyclopedia of economics*, vol. 2, no. 2008, pp. 1–12, 2008.

[3] D. A. Ostrowski, “An approximation of betweenness centrality for social networks,” in *ICSC*, pp. 489–492, 2015.

[4] H. Jeong, S. P. Mason, A.-L. Barabási, and Z. N. Oltvai, “Lethality and centrality in protein networks,” *Nature*, vol. 411, no. 6833, pp. 41–42, 2001.

[5] L. Baldesi, L. Maccari, and R. L. Cigno, “On the use of eigenvector centrality for cooperative streaming,” *IEEE Communications Letters*, vol. 21, no. 9, pp. 1953–1956, 2017.

[6] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[7] M. Everett and S. P. Borgatti, “Ego network betweenness,” *Social networks*, vol. 27, no. 1, pp. 31–38, 2005.

[8] L. C. Freeman, “Centered graphs and the structure of ego networks,” *Math. Soc. Sci.*, vol. 3, no. 3, pp. 291–304, 1982.

[9] B. Guidi, M. Conti, A. Passarella, and L. Ricci, “Distributed protocols for ego betweenness centrality computation in dosns,” in *2014 IEEE International Conference on Pervasive Computing and Communication Workshops, PerCom 2014 Workshops, Budapest, Hungary, March 24-28, 2014*, pp. 539–544, IEEE Computer Society, 2014.

[10] A. Cuzzocrea, A. Papadimitriou, D. Katsaros, and Y. Manolopoulos, “Edge betweenness centrality: A novel algorithm for qos-based topology control over wireless sensor networks,” *J. Netw. Comput. Appl.*, vol. 35, no. 4, pp. 1210–1217, 2012.

[11] E. M. Daly and M. Haahr, “Social network analysis for routing in disconnected delay-tolerant manets,” in *Proceedings of the 8th ACM Interational Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc 2007, Montreal, Quebec, Canada, September 9-14, 2007*, pp. 32–40, ACM, 2007.

[12] A. T. Akabane, R. Immich, R. W. Pazzi, E. R. M. Madeira, and L. A. Villas, “Distributed egocentric betweenness measure as a vehicle selection mechanism in vanets: A performance evaluation study,” *Sensors*, vol. 18, no. 8, p. 2731, 2018.

[13] 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.

[14] L. Chang, C. Zhang, X. Lin, and L. Qin, “Scalable top-k structural diversity search,” in *ICDE*, pp. 95–98, 2017.

[15] Q. Zhang, R.-H. Li, Q. Yang, G. Wang, and L. Qin, “Efficient top-k edge structural diversity search,” in *ICDE*, pp. 205–216, 2020.

[16] N. Chiba and T. Nishizeki, “Arboricity and subgraph listing algorithms,” *SIAM Journal on computing*, vol. 14, no. 1, pp. 210–223, 1985.

[17] 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.

[18] 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.

[19] J. Wang and J. Cheng, “Truss decomposition in massive networks,” *PVLDB*, vol. 5, no. 9, pp. 812–823, 2012.

[20] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, “Querying k-truss community in large and dynamic graphs,” *SIGMOD*, 2014.

[21] 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.

[22] D. Prountzos and K. Pingali, “Betweenness centrality: algorithms and implementations,” in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, Shenzhen, China, February 23-27, 2013*, pp. 35–46, ACM, 2013.

[23] C. Daniel, A. Furno, L. Goglia, and E. Zimeo, “Fast cluster-based computation of exact betweenness centrality in large graphs,” *J. Big Data*, vol. 8, no. 1, p. 92, 2021.

[24] M. Riondato and E. Upfal, “ABRA: approximating betweenness centrality in static and dynamic graphs with rademacher averages,” in *KDD* (B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, eds.).

[25] M. Riondato and E. M. Kornaropoulos, “Fast approximation of betweenness centrality through sampling,” *Data Min. Knowl. Discov.*, vol. 30, no. 2, pp. 438–475, 2016.

[26] L. C. Freeman, “Centrality in social networks conceptual clarification,” *Social networks*, vol. 1, no. 3, pp. 215–239, 1978.

[27] R. Fan, K. Xu, and J. Zhao, “A gpu-based solution for fast calculation of the betweenness centrality in large weighted networks,” *PeerJ Comput. Sci.*, vol. 3, p. e140, 2017.

[28] R. K. Behera, D. Naik, D. Ramesh, and S. K. Rath, “MR-IBC: mapreduce-based incremental betweenness centrality in large-scale complex networks,” *Soc. Netw. Anal. Min.*, vol. 10, no. 1, p. 25, 2020.

[29] M. H. Chehreghani, “An efficient algorithm for approximate betweenness centrality computation,” *Comput. J.*, vol. 57, no. 9, pp. 1371–1382, 2014.

[30] A. Furno, N.-E. El Faouzi, R. Sharma, and E. Zimeo, “Reducing pivots of approximated betweenness computation by hierarchically clustering complex networks,” in *COMPLEX NETWORKS*, pp. 65–77, 2017.

[31] P. Crescenzi, P. Fraigniaud, and A. Paz, “Simple and fast distributed computation of betweenness centrality,” in *INFOCOM*, pp. 337–346, 2020.

[32] P. V. Marsden, “Egocentric and sociocentric measures of network centrality,” *Social networks*, vol. 24, no. 4, pp. 407–422, 2002.

[33] 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.

[34] K. C.-C. Chang and S.-w. Hwang, “Minimal probing: supporting expensive predicates for top-k queries,” in *SIGMOD*, pp. 346–357, 2002.

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

[36] C. Xiao, W. Wang, X. Lin, and H. Shang, “Top-k set similarity joins,” in *ICDE*, pp. 916–927, 2009.

[37] R. Fagin, “Combining fuzzy information from multiple systems,” *Journal of computer and system sciences*, vol. 58, no. 1, pp. 83–99, 1999.

[38] R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware,” *Journal of computer and system sciences*, vol. 66, no. 4, pp. 614–656, 2003.

[39] 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.

[40] R.-H. Li and J. X. Yu, “Scalable diversified ranking on large graphs,” *IEEE TKDE*, vol. 25, no. 9, pp. 2133–2146, 2013.

[41] A. Angel and N. Koudas, “Efficient diversity-aware search,” in *Proceedings of SIGMOD*, pp. 781–792, 2011.

[42] X. Zhu, J. Guo, X. Cheng, P. Du, and H.-W. Shen, “A unified framework for recommending diverse and relevant queries,” in *Proceedings of WWW*, pp. 37–46, 2011.

[43] R. Agrawal, S. Gollapudi, A. Halverson, and S. Ieong, “Diversifying search results,” in *Proceedings of WSDM*, pp. 5–14, 2009.

[44] 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.

[45] F. Bi, L. Chang, X. Lin, and W. Zhang, “An optimal and progressive approach to online search of top-k influential communities,” *VLDB*, vol. 11, no. 9, pp. 1056–1068, 2018.