Efficient Core Maintenance in Large Dynamic Graphs

Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao

Abstract—The k-core decomposition in a graph is a fundamental problem for social network analysis. The problem of k-core decomposition is to calculate the core number for every node in a graph. Previous studies mainly focus on k-core decomposition in a static graph. There exists a linear time algorithm for k-core decomposition in a static graph. However, in many real-world applications such as online social networks and the Internet, the graph typically evolves over time. In such applications, a key issue is to maintain the core numbers of nodes when the graph changes over time. A simple implementation is to perform the linear time algorithm to recompute the core number for every node after the graph is updated. Such simple implementation is expensive when the graph is very large. In this paper, we propose a new efficient algorithm to maintain the core number for every node in a dynamic graph. Our main result is that only certain nodes need to update their core numbers when the graph is changed by inserting/deleting an edge. We devise an efficient algorithm to identify and recompute the core numbers of such nodes. The complexity of our algorithm is independent of the graph size. In addition, to further accelerate the algorithm, we develop two pruning strategies by exploiting the lower and upper bounds of the core number. Finally, we conduct extensive experiments over both real-world and synthetic datasets, and the results demonstrate the efficiency of the proposed algorithm.

Index Terms—Core maintenance, k-core decomposition, dynamic graphs

1 Introduction

In the last decade, online social network analysis has become an important topic in both research and industry communities due to a huge number of applications. A crucial issue in social network analysis is to identify the cohesive subgroups of users in a network. The cohesive subgroup denotes a subset of users who are well-connected to one another in a network [1]. In the literature, there are a number of metrics for measuring the cohesiveness of a group of users in a social network. Examples include cliques, n-cliques, n-clans, k-plexes, k-core, k-trusses and so on [2].

For most of these metrics except k-core, the computational complexity is typically NP-hard or at least quadratic. k-core, as an exception, is a well-studied notion in graph theory and social network analysis [3]. Through-out the paper, we will interchangeably use graph and network. Given a graph G, the k-core is the largest subgraph of G such that all the nodes in such a subgraph have degree of at least k. For each node v in G, the core number of v denotes the largest k such that a k-core exists and contains v. The k-core decomposition in a graph G is the computation of the core number for every node in G. There is a linear time algorithm, devised by Batagelj and Zaversnik [4], to compute the k-core decomposition in a graph.

Besides the analysis of cohesive subgroup, k-core decomposition has been recognized as a powerful tool to analyze the structure and function of a network, and it has many applications. For example, the k-core decomposition has been applied to visualize large networks [5], [6], to map, model and analyze the topological structure of the Internet [7], [8], to predict the function of protein in protein-protein interaction network [9]–[11], to identify influential spreader in complex networks [12], as well as to study percolation on complex networks [13].

From the algorithmic perspective, efficient and scalable algorithms for k-core decomposition in a static graph already exist [4], [14], [15]. However, in many real-world applications, such as online social network and the Internet, the network evolves over time. In such dynamic networks, many applications require to maintain the core number for every node online, given the network changes over time. For example, in an application of k-core-based interactive graph visualization [16], the graph is typically changed by the users (insert or delete some edges). In such a case, the visualization algorithm needs to update the core numbers of all the nodes online so that the algorithm can dynamically adjust the layout of the graph. In addition, for the applications of social network analysis, the algorithms of maintaining the core numbers online can be used to monitor the dynamics of cohesive subgroups. However, in a dynamic network, it is difficult to update the core numbers of nodes. The reason is as follows. An edge insertion/deletion results in that the degree of two end-nodes of the edge increase/decrease by 1. This may lead to the updates of the core numbers of the end-nodes. Such

• *R.-H. Li and R. Mao are with the Guangdong Province Key Laboratory of Popular High Performance Computers, Shenzhen University, Shenzhen 518060, China. E-mail: lironghuascut@gmail.com; mao@szu.edu.cn.*
• *J. X. Yu is with the Chinese University of Hong Kong, Hong Kong, China. E-mail: yu@se.cuhk.edu.hk.*

Manuscript received 30 Nov. 2012; revised 9 Aug. 2013; accepted 21 Aug. 2013. Date of publication 26 Sep. 2013; date of current version 29 Aug. 2014. Recommended for acceptance by J. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier 10.1109/TKDE.2013.158

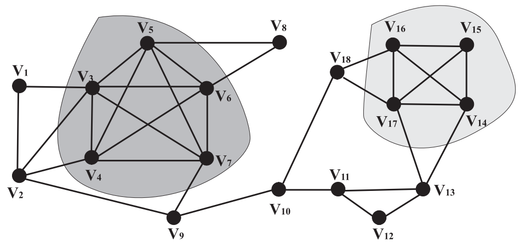


Fig. 1. Example graph.

updates may affect the core numbers of the neighbors of the end-nodes which may need to be updated. In other words, the update of the core numbers of the end-nodes may *spread* across the network. For example, in Fig. 1, if we insert an edge (v_8, v_{10}) into the graph, then the degree of v_8 and v_{10} increase by 1. Suppose that the core numbers of v_8 and v_{10} increase by 1, then we can see that such core number update leads to the core numbers of v_{10}'s neighbors (v_9, v_{18}, v_{11}) that may need to be updated. And then the update of core numbers of v_{10}'s neighbors will result in the update of core numbers of v_{10}'s neighbors' neighbors. This update process may *spread* over the network. Therefore, it is hard to determine which node in a network should update its core number given the network changes.

To update the core number for every node in a dynamic graph, in [17], Miorandi and Pellegrini propose to use the linear algorithm given in [4] to recompute the core number for every node in a graph. Obviously, such an algorithm is expensive when the graph is very large. In this paper, we propose an efficient algorithm to maintain the core number for each node in a dynamic network. Our algorithm is based on the following key observation. We find that only a certain number of nodes need to update their core numbers when a graph is updated by inserting/deleting an edge. Reconsider the example in Fig. 1. After inserting an edge (v_8, v_{10}), we can observe that only the core numbers of the nodes $\{v_8, v_{10}, v_{18}, v_9, v_2\}$ updates, while the core numbers of the remaining nodes does not change. The key challenge is how to identify the nodes whose core numbers need to be updated. To tackle this problem, we propose a three-stage algorithm to update the core numbers of the nodes. First, we prove that the core numbers of the nodes who simultaneously satisfy the following two conditions may need to be updated: (1) the nodes are reachable from the end-nodes of the inserted/deleted edge, and (2) their core numbers equal to the minimal core number of the end-nodes of the inserted/deleted edge. Based on this, we propose a coloring algorithm to find such nodes whose core numbers may need to be updated. Second, from the nodes found by the coloring algorithm, we propose a recoloring algorithm to identify the nodes whose core numbers definitely need to be updated. Third, we update the core numbers of such nodes by a linear algorithm. The major advantage of our algorithm is that its time complexity is independent of the graph size, and it depends on the size of the nodes found by the coloring algorithm. To further accelerate our algorithm, we develop two pruning techniques to reduce the size of the nodes found by the coloring algorithm. In addition, it is worth mentioning that the proposed algorithm can also be used to handle a batch of edge insertions and deletions by processing the edges one by one. Also, the proposed technique can be applied to process node insertions and deletions, because node insertions and deletions can be simulated by a sequence of edge insertions and deletions respectively. Finally, we perform extensive experiments to evaluate our algorithms over both real-world and synthetic datasets, and the results demonstrate the efficiency of the algorithm. For example, in the real-world datasets, our algorithm reduces the average update time of the baseline algorithm up to 10231 times for handling a single edge update.

The rest of this paper is organized as follows. We give the problem statement in Section 2. We propose our basic algorithm as well as the pruning strategies in Section 3. Extensive experimental studies are reported in Section 4, and the related work is discussed in Section 5. We conclude this work in Section 6.

2 Preliminaries

Consider an undirected and unweighted graph $G = (V, E)$, where V denotes a set of nodes and E denotes a set of undirected edges between the nodes. Let $n = |V|$ and $m = |E|$ be the number of nodes and the number of edges in G, respectively. A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. We give the definition of the k-core [3] as follows.

Definition 1. *Given a nonnegative integer k and a graph G. The k-core (or a core of order k) is the largest subgraph G' of G where each node in G' has at least a degree k.*

The core number of a node v is defined as the largest k such that the k-core exists and contains this node. We denote the core number of node v as C_v. Note that the node with a large core number is also in the low order core. That is to say, the cores are nested. For example, assuming a node v is in a 3-core, then node v is also in 2-core, 1-core and 0-core.

Given a graph G, the problem of k-core decomposition is to calculate the core number for every node in G. The following example illustrates the concept of k-core decomposition in graph.

Example 1. Fig. 1 shows a graph G that contains 18 nodes, i.e., $v_1, \ldots, v_{18}$. By Definition 1, we can find that the nodes $v_3, \ldots, v_7$ form a 4-core. The reason is because the induced subgraph by the nodes $v_3, \ldots, v_7$ is the largest subgraph in which the degrees of nodes are no less than 4. Similarly, the subgraph induced by the nodes $v_3, \ldots, v_7, v_{14}, \ldots, v_{17}$ is a 3-core, and the whole graph G is a 2-core. Here we can find that the nodes $v_3, \ldots, v_7$ are also in the 3-core and 2-core.

It is well known that the k-core decomposition in a static graph can be calculated by an $O(n + m)$ algorithm [4]. In this paper, we consider the core maintenance problem in dynamic graphs defined below.

Problem definition: Given a graph $G = (V, E)$, and the core numbers of all the nodes in G. The goal of the core maintenance problem is to update the core numbers of all the nodes in G when the graph G is changed by inserting or deleting an edge.

The challenge in the above problem is that an edge insertion or deletion may result in the core numbers of a number of nodes that need to be updated. Previous solution for this problem [17] is to perform the $O(n+m)$ core decomposition algorithm to re-compute the core number for every node in the updated graph. Clearly, such algorithm is expensive when the graph is very large. In the following, we shall devise an efficient algorithm for this problem. Note that although the proposed algorithm is mainly addressed to the core maintenance problem given the graph is updated by one edge insertion or deletion, it can also be used to process a batch of edge updates. Moreover, since node insertions and deletions can be easily simulated as a sequence of edge insertions and edge deletions respectively, our algorithm can also be applied to handle node updates.

3 The Proposed Algorithm

Let $N(v)$ be the set of neighbor nodes of node v, D_v be the degree of node v, i.e., $D_v = |N(v)|$. Then, we give two important quantities associated with a node v as follows. Specifically, we define X_v as the number of v's neighbors whose core numbers are greater than or equal to C_v, and define Y_v as the number of v's neighbors whose core numbers are strictly greater than C_v. Formally, for a node v, we have $X_v = |\{u{:}u \in N(v), C_u \geq C_v\}|$ and $Y_v = |\{u{:}u \in N(v), C_u > C_v\}|$. In effect, by definition, X_v denotes the degree of node v in the C_v-core. It is important to note that unless otherwise specified, here the definitions of D_v, C_v, X_v, and Y_v are based on the graph before updating. The following lemma shows that C_v is bounded by Y_v and X_v.

Lemma 1. *For every node v of a graph G, we have $Y_v \leq C_v \leq X_v \leq D_v$.*

Below, we define the notion of induced core subgraph.

Definition 2. *Given a graph $G = (V, E)$ and a node v, the induced core subgraph of node v, denoted as $G_v = (V_v, E_v)$, is a connected subgraph which contains node v. Moreover, the core numbers of all the nodes in G_v are equivalent to C_v.*

By Definition 2, the induced core subgraph of node v includes the nodes such that they are reachable from v and their core numbers are equal to C_v. Based on Definition 2, we define the union of two induced core subgraphs.

Definition 3. *For two nodes u and v and their corresponding induced core subgraph $G_u = (V_u, E_u)$ and $G_v = (V_v, E_v)$, the union of G_u and G_v is defined as $G_{u\cup v} = (V_{u\cup v}, E_{u\cup v})$, where $V_{u\cup v} = V_v \bigcup V_u$ and $E_{u\cup v} = \{(v_i, v_j) | (v_i, v_j) \in E, v_i \in V_{u\cup v}, v_j \in V_{u\cup v}\}$.*

It is worth mentioning that the union of two induced core subgraphs is not necessarily connected.

Based on Definition 2 and 3, we present a k-core update theorem as follows.

Theorem 1 (k-core update theorem). *Given a graph $G = (V, E)$ and two nodes u and v. Then, after inserting or deleting an edge (u, v) in G, we have the following results.*

- *If $C_u > C_v$, only the core numbers of nodes in the induced core subgraph of node v, i.e., G_v, may need to be updated.*
- *If $C_u < C_v$, only the core numbers of nodes in the induced core subgraph of node u, i.e., G_u, may need to be updated.*
- *If $C_u = C_v$, only the core numbers of nodes in the union of two induced core subgraphs G_u and G_v, i.e., $G_{u\cup v}$, may need to be updated.*

To prove Theorem 1, we first give some useful lemmas as follows. The proofs can be easily obtained, thus we omit them for brevity.

Lemma 2. *If we insert (delete) an edge (u, v) in a graph G, the core number of any node in G increases (decreases) by at most 1.*

Lemma 3. *Given a graph G and two nodes u and v such that $C_u = C_v$. If we insert an edge (u, v) in G, then either C_u and C_v increase by 1 or C_u and C_v do not change.*

Lemma 4. *Given a graph G and an edge (u, v). Suppose that G is updated by inserting or deleting an edge (u, v). Then, for any node w in G, if the core number of w (C_w) needs to be changed, such change only affects the core numbers of nodes in G_w. If C_w does not change, then it does not affect the core number of the nodes in G.*

Armed with the above lemmas, we prove the k-core update theorem as follows.

Proof of Theorem 1. For the insertion of an edge (u, v), we consider three different cases: (1) $C_u > C_v$, (2) $C_u < C_v$, and (3) $C_u = C_v$. For $C_u > C_v$, we know that node u is in a higher order core than node v. By Definition 1, adding a neighbor v with a small core number to a node u does not affect C_u. Since C_u does not change, node u will not affect the core numbers of the nodes in G (by Lemma 4). Consequently, we only need to update the core numbers of the nodes that are affected by node v. By Lemma 4, if C_v changes, only the core numbers of nodes in G_v may need to be updated. If C_v does not change, then no node's core number needs to be updated. This proves the case (1). Symmetrically, we can use the similar arguments to prove the case (2). For case (3), after inserting an edge (u, v), by Lemma 3, either C_u and C_v increase by 1 or C_u and C_v do not change. If C_u and C_v do not change, by Lemma 4, we conclude that no node's core number needs to be updated. If C_u and C_v increase by 1, by Lemma 4, the core numbers of the nodes in G_u and G_v may need to be updated. That is to say, the core numbers of the nodes in $G_{u\cup v}$ may need to be updated.

Similarly, for the deletion of an edge (u, v), we also consider three different cases: (1) $C_u > C_v$, (2) $C_u < C_v$, and (3) $C_u = C_v$. The proofs of the first two cases are very similar to those of the edge-insertion case, thereby we omit them for brevity. For $C_u = C_v$, after deleting an edge (u, v), if C_u and C_v do not change, we conclude that no node's core number needs to be updated according to Lemma 4. If C_u changes, by Lemma 4, the core numbers of nodes in G_u may need to be updated. Likewise, if C_v changes, the core numbers of nodes in G_v may need to be updated. To summarize, after removing an edge (u, v), only the core numbers of the nodes in $G_{u\cup v}$ may need to be updated. This completes the proof. □

Algorithm 1 Insertion(G, u, v)

```
Input:  Graph $G = (V, E)$ and an edge $(u, v)$
Output: the updated core numbers of the nodes
 1: Initialize visited($w$) $\leftarrow 0$ for all node $w \in V$;
 2: Initialize color($w$) $\leftarrow 0$ for all node $w \in V$;
 3: $V_c \leftarrow \emptyset$;
 4: if $C_u > C_v$ then
 5:     $c \leftarrow C_v$;
 6:     Color($G$, $v$, $c$);
 7:     RecolorInsert($G$, $c$);
 8:     UpdateInsert($G$, $c$);
 9: else
10:     $c \leftarrow C_u$;
11:     Color($G$, $u$, $c$);
12:     RecolorInsert($G$, $c$);
13:     UpdateInsert($G$, $c$);
```

Algorithm 2 void **Color**(G, u, c)

```
1: Initialize a queue $Q$;
2: $Q.enqueue(u)$; visited($u$) $\leftarrow 1$;
3: while $Q$ is not empty do
4:     $u \leftarrow Q.dequeue()$;
5:     for each node $w \in N(u)$ do
6:         if visited($w$) $= 0$ and $C_w = c$ then
7:             $Q.enqueue(w)$; visited($w$) $\leftarrow 1$;
8:     if color($u$) = 0 then
9:         $V_c \leftarrow V_c \cup \{u\}$; color($u$) = 1;
```

Algorithm 3 void **RecolorInsert**(G, c)

```
 1: flag $\leftarrow 0$;
 2: for each node $u \in V_c$ do
 3:     if color($u$) = 1 then
 4:         $\tilde{X}_u \leftarrow 0$;
 5:         for each node $w \in N(u)$ do
 6:             if (color($w$) = 1) or ($C_w > c$) then
 7:                 $\tilde{X}_u \leftarrow \tilde{X}_u + 1$;
 8:         if $\tilde{X}_u \leq c$ then
 9:             color($u$) $\leftarrow 0$;
10:             flag $\leftarrow 1$;
11: if flag = 1 then
12:     RecolorInsert($G$, $c$);
```

3.1 The Basic Algorithm

Based on the k-core update theorem, we present a basic algorithm for core maintenance in a graph given that the graph is updated by an edge insertion/deletion. Below, we detail the algorithms for edge insertion and deletion respectively.

Algorithm for edge insertion: Our main algorithm for edge insertion consists of three steps. After inserting an edge (u, v), by the k-core update theorem, only the core numbers of nodes in the induced core subgraph (G_u, or G_v, or $G_{u \cup v}$) may need to be updated. Therefore, the first step of our algorithm is to identify the nodes in the induced core subgraph. Let V_c be the set of nodes found in the first step. In other words, $V_c = V_u$, or $V_c = V_v$, or $V_c = V_{u \cup v}$. Then, the second step of our algorithm is to determine those nodes in V_c whose core numbers definitely need to be updated. Finally, the third step of our algorithm is to update the core numbers of such nodes.

The main algorithm for edge insertion, called **Insertion**, is outlined in Algorithm 1. Algorithm 1 includes three sub-algorithms, namely **Color**, **RecolorInsert**, and **UpdateInsert**, which correspond to the first, the second, and the third step of the main algorithm respectively. Before we proceed further, we define an important concept "color" which is frequently used in our algorithm. A color is a 0/1 value which is utilized to distinguish the nodes whose core numbers need to be updated from those nodes whose core numbers are unchanged. Initially, all the nodes are assigned a color 0 (line 2 in Algorithm 1). Then, the algorithm invokes **Color** to color all the nodes in V_c by a color 1 (line 6 and 11 in Algorithm 1), denoting that the core numbers of those nodes may need to be updated. Subsequently, the algorithm invokes **RecolorInsert** to recolor the nodes in V_c whose core numbers are definitely unchanged by a color 0 (line 7 and 12 in Algorithm 1). After that, we can get that the core numbers of the nodes in V_c with color 1 has to be updated. Finally, the algorithm invokes **UpdateInsert** to update the core numbers of such nodes (line 8 and 13 in Algorithm 1). Recall that by the k-core update theorem, the algorithm has to process three different cases: $C_u > C_v$, $C_u < C_v$, and $C_u = C_v$ (line 4 and 9 in Algorithm 1). However, in Algorithm 1, the last two cases can be merged (line 9-13 in Algorithm 1), and we will interpret this point later. Below, we detail the sub-algorithms, **Color**, **RecolorInsert**, and **UpdateInsert**.

To simplify our description, we mainly focus on describing the sub-algorithms under the case $C_u = C_v$, and similar description can be used for other cases ($C_u < C_v$ and $C_u > C_v$). Suppose that node u and v have core number $C_u = C_v = c$. In this case, we have $V_c = V_{u \cup v}$. By Definition 3, finding the nodes in $V_{u \cup v}$ can be done by a Breadth-First Search (BFS) algorithm. **Color** depicted in Algorithm 2 is indeed such a BFS algorithm. By **Color**, all the nodes in V_c are colored by a color 1. To find all the nodes in $V_{u \cup v}$, we can invoke **Color**(G, u, c). Note that after inserting edge (u, v), the nodes that are reachable from v can also be found by **Color**(G, u, c). Therefore, the coloring process under the case $C_u = C_v$ is the same as the coloring process under the case $C_u < C_v$, where **Color**(G, u, c) is invoked to find all the nodes in V_u. Since the subsequent processes in Algorithm 1 are the same for all three cases, we can merge these two cases (line 9-13 in Algorithm 1).

RecolorInsert described in Algorithm 3 is used to identify the nodes in V_c whose core numbers are definitely unchanged. Specifically, Algorithm 3 recursively recolors the nodes whose core numbers do not change by a color 0. Let $\tilde{X}_u$ be the sum of the number of node u's neighbors whose core numbers are larger than c and the number of u's neighbors whose color is 1. In each recursion, the algorithm recomputes $\tilde{X}_u$ for each node u in V_c (line 4-7 in Algorithm 3). For a node u, if the current $\tilde{X}_u$ is smaller than or equal to c, then the algorithm recolors it by 0 (line 8-10 in Algorithm 3). The rationale is as follows. If $\tilde{X}_w \leq c$, w has at most c neighbors whose core numbers are larger than c after inserting an edge (u, v). As a result, C_w cannot be updated, and thus the algorithm recolors it by 0. It is important to note that this recoloring process may affect the color of w's neighbors. This is because before recoloring w, w may contribute to calculate $\tilde{X}_z$, where z is a neighbor

Algorithm 4 void **UpdateInsert**(G, c)

```
1: for each node $w \in V_c$ do
2:     if color($w$) = 1 then
3:         $C_w \leftarrow c + 1$;
```

of w. Consequently, the algorithm has to recursively recolor the nodes in V_c. The recursion is terminated until no node needs to be recolored. Note that Algorithm 3 is recursively invoked at most $|V_c| + 1$ times, because the algorithm at least recolors one node in one recursion in the worse case. The following theorem shows that after Algorithm 3 terminates, a node with a color 1 is a sufficient and necessary condition for updating its core number.

Theorem 2. *Under the case of insertion of an edge (u, v), the core number of a node needs to be updated if and only if its color is 1 after Algorithm 3 terminates.*

Proof. First, we prove that if the core number of a node w needs to be updated, then its color is 1 after Algorithm 3 terminates. We focus on the case of $C_u = C_v = c$, similar proof can be used to prove the other two cases. By our assumption and Lemma 4, we have $w \in V_c$, where $V_c = V_{u \cup v}$. Then, by Lemma 2, after inserting an edge (u, v), the core number of the nodes in V_c increases by at most 1. Therefore, if C_w needs to be updated, then the updated core number of w must be $c + 1$. That is to say, node w must have $c+1$ neighbors whose core numbers are larger than or equal to $c + 1$. Now assume that the color of node w is 0. This means that $\tilde{X}_w \leq c$ when Algorithm 3 terminates. This result implies that node w has at most c neighbors whose core numbers are larger than c, which is a contradiction.

Second, we prove that if a node has a color 1 after Algorithm 3 terminates, then the core number of this node must be updated. Let V_1 be a set of nodes with color 1 after Algorithm 3 terminates, and $V_{>c}$ be a set of nodes whose core numbers are greater than c. Denote by $G^\star = (V_1 \cup V_{>c}, E^\star)$ an subgraph induced by the nodes $V_1 \cup V_{>c}$. Consider a node w in such an induced subgraph $G^\star$. Clearly, if w has a color 1 (i.e., $w \in V_1$), then it has $\tilde{X}_w$ ($\tilde{X}_w > c$) neighbors in $G^\star$. If w has a color 0 (i.e., $w \in V_{>c}$), then its core number C_w is larger than c. By Definition 1, the induced subgraph $G^\star$ belongs to the $(c+1)$-core. Therefore, the core number of a node w with color 1 is at least $c + 1$. By Lemma 2, after inserting an edge (u, v), the core number of any nodes in graph G increases by at most 1. Consequently, the core number of the nodes with color 1 increases by 1. □

UpdateInsert outlined in Algorithm 4 increases the core numbers of the nodes in V_c with color 1 to $c + 1$, because only the core numbers of those nodes increase by 1 after the coloring and recoloring processes. The correctness of our algorithm for edge insertion can be guaranteed by Theorem 1 and Theorem 2. The following example explains how the **Insertion** algorithm works.

Example 2. Let us consider the same graph given in Fig. 1. Assume that we insert an edge (v_8, v_{10}), which results in a graph given in Fig. 2. In Fig. 2, the dashed line denotes the inserted edge. Since $C_{v_8} = C_{v_{10}} = c = 2$, the **Insertion** algorithm first invokes **Color**(G, v_8, 2). After this process, we can get that $V_c = \{v_8, v_{10}, v_9, v_2, v_1, v_{18}, v_{11}, v_{12}, v_{13}\}$. And all the nodes in V_c are colored by 1 and the other nodes are colored by 0. Then, the algorithm invokes **RecolorInsert**(G, 2). At the first recursion, we can find that $\tilde{X}_{v_1} = 2$, thereby it is recolored by 0. Also, the node V_{12} is recolored by 0, because $\tilde{X}_{v_{12}} = 2$. At the second recursion, we can find that the nodes v_{11} and v_{13} are recolored by 0. At the third recursion, no node needs to be recolored, the algorithm therefore terminates. After invoking **RecolorInsert**(G, 2), the nodes $\{v_8, v_{10}, v_9, v_2, v_{18}\}$ are colored by 1, thereby their core numbers must increase to 3 by Theorem 2. Finally, the **Insertion** algorithm invokes **UpdateInsert**(G, 2) to update the core numbers of such nodes. As a consequence, the core numbers of the nodes $\{v_8, v_{10}, v_9, v_2, v_{18}\}$ is increased to 3.

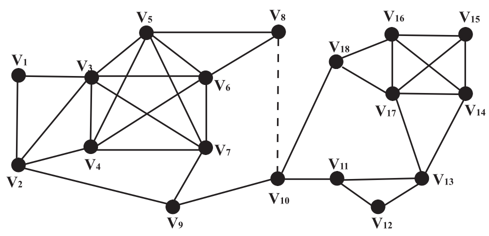


Fig. 2. Graph after inserting an edge (v_8, v_{10}).

We analyze the time complexity of the **Insertion** algorithm as follows. First, the **Color** algorithm takes $O(\sum_{u \in V_c} D_u)$ time complexity. Second, the **RecolorInsert** algorithm takes $O(|V_c| \sum_{u \in V_c} D_u)$ time complexity in the worse case, as the algorithm is recursively invoked at most $O(|V_c|)$ times and each recursion takes $O(\sum_{u \in V_c} D_u)$ time complexity. Finally, the **UpdateInsert** algorithm takes $O(|V_c|)$ time complexity. Put it all together, the time complexity of the **Insertion** algorithm is $O(|V_c| \sum_{u \in V_c} D_u)$ in the worse case, which is independent of the graph size. However, in practice, the algorithm is more efficient than such worse-case time complexity. The reasons are twofold. On the one hand, $|V_c|$ is typically not very large w.r.t. the number of nodes of the graph (we will show this in the experiments). On the other hand, very often, the **RecolorInsert** algorithm terminates very fast.

Algorithm for edge deletion: The main algorithm for edge deletion, namely **Deletion**, is outlined in Algorithm 5. Similar to the edge insertion case, **Deletion** also includes three sub-algorithms: **Color**, **RecolorDelete**, and **UpdateDelete**. Here **Color** is used to find the nodes in the induced core subgraph, **RecolorDelete** is utilized to identify the nodes whose core numbers need to be updated, and **UpdateDelete** is applied to update the core numbers of the nodes identified by **RecolorDelete**. The detailed description of **Deletion** is given as follows.

Similarly, let V_c be a set of nodes whose core numbers may need to be updated. First, Algorithm 5 initializes the color of all the nodes to 0 and V_c to an empty set. Likewise, under the edge deletion case, we also have to consider three cases. That is, $C_u > C_v$, $C_u < C_v$, and $C_u = C_v$. Here we only focus on the $C_u = C_v$, because the other two cases are very similar to the edge-insertion cases. For the $C_u = C_v$ case, the algorithm first invokes **Color**(G, u, c) to find the nodes in G_u

Algorithm 5 Deletion(G, u, v)

```
Input:   Graph $G = (V, E)$ and an edge $(u, v)$
Output: the updated core numbers of the nodes
1: Initialize visited($w$) $\leftarrow 0$ for all node $w \in V$;
2: Initialize color($w$) $\leftarrow 0$ for all node $w \in V$;
3: $V_c \leftarrow \emptyset$;
4: if $C_u > C_v$ then
5:     $c \leftarrow C_v$;
6:     Color($G$, $v$, $c$);
7:     RecolorDelete($G$, $c$);
8:     UpdateDelete($G$, $c$);
9: if $C_u < C_v$ then
10:    $c \leftarrow C_u$;
11:    Color($G$, $u$, $c$);
12:    RecolorDelete($G$, $c$);
13:    UpdateDelete($G$, $c$);
14: if $C_u = C_v$ then
15:    $c \leftarrow C_u$;
16:    Color($G$, $u$, $c$);
17:    if color($v$) $= 0$ then
18:        Initialize visited($w$) $\leftarrow 0$ for all node $w \in V$;
19:        Color($G$, $v$, $c$);
20:        RecolorDelete($G$, $c$);
21:        UpdateDelete($G$, $c$);
22:    else
23:        RecolorDelete($G$, $c$);
24:        UpdateDelete($G$, $c$);
```

(line 16 in Algorithm 5). Then, the algorithm has to handle two different cases. First, if u can reach v, then the coloring algorithm can also find the nodes in G_v (in this case, v's color is 1, line 22 in Algorithm 5). Second, if u cannot reach v (v's color is 0), then the algorithm invokes **Color**(G, v, c) to find the nodes in G_v (line 19 in Algorithm 5). After this process, all the node in $G_{u \cup v}$ are recorded in V_c. Then, we can invoke **RecolorDelete**(G, c) and **UpdateDelete**(G, c) algorithms to update the core numbers of the nodes in V_c. Below, we give the detailed descriptions of **RecolorDelete** and **UpdateDelete** respectively.

RecolorDelete recursively recolors the nodes whose core numbers need to be updated by a color 0. In each recursion, the algorithm calculates $\tilde{X}_w$ for every node w in V_c. Similarly, here $\tilde{X}_w$ denotes the sum of the number of w's neighbors whose color is 1 and the number of w's neighbors whose core numbers are larger than c, where $c = \min\{C_u, C_v\}$. For a node $w \in V_c$, if $X_w < c$, then the algorithm colors w by a color 0. The algorithm terminates if no node needs to be recolored. Clearly, the algorithm is invoked at most $|V_c|$ times. The following theorem shows that a node in V_c with a color 0 after Algorithm 6 terminates is a sufficient and necessary condition for updating its core number.

Theorem 3. *Under the case of deletion of an edge (u, v), a node in V_c whose core number needs to update if and only if its color is 0 after Algorithm 6 terminates.*

Proof. First, we prove that if a node w in V_c whose core number needs to be updated, then its color is 0 after Algorithm 6 terminates. By our assumption and Lemma 2, after deleting an edge (u, v), C_w decreases by 1. This means that C_w decreases to $c - 1$. That is to say, w has $c - 1$ neighbors whose core numbers are no less

Algorithm 6 void **RecolorDelete**(G, c)

```
1: flag $\leftarrow 0$;
2: for each node $u \in V_c$ do
3:     if color($u$) = 1 then
4:         $\tilde{X}_u \leftarrow 0$;
5:         for each node $w \in N(u)$ do
6:             if (color($w$) = 1) or ($C_w > c$) then
7:                 $\tilde{X}_u \leftarrow \tilde{X}_u + 1$;
8:         if $\tilde{X}_u < c$ then
9:             color($u$) $\leftarrow 0$;
10:            flag $\leftarrow 1$;
11: if flag = 1 then
12:    RecolorDelete($G$, $c$);
```

Algorithm 7 void **UpdateDelete**(G, c)

```
1: for each node $w \in V_c$ do
2:     if color($w$) $= 0$ then
3:         $C_w \leftarrow c - 1$;
```

than $c - 1$. Suppose that the color of w is 1 after the algorithm terminates. This implies that $X_w \geq c$. Recall that X_w denotes the sum of the number of w's neighbors whose core numbers are larger than c and the number of w's neighbors whose color is 1. Note that a node with color 1 suggests that its core number equals to c. As a result, w has at least c neighbors whose core numbers are larger than or equal to c, which is a contradiction.

Second, we prove that if a node w in V_c is recolored by 0 after Algorithm 6 terminates, then C_w must be updated. Let $V_{>c}$ be a set of nodes whose core numbers are larger than c. Then, after deleting an edge (u, v), we construct an induced subgraph by the nodes in $V_c \cup V_{>c}$, which is denoted as $G^\star = (V_c \cup V_{>c}, E^\star)$. Note that the core numbers of the nodes in $V \backslash \{V_c \cup V_{>c}\}$ are smaller than c. Therefore, they do not affect the core numbers of the nodes in $V_c \cup V_{>c}$. If a node $w \in V_c$ with a color 0 after Algorithm 6 terminates, then $X_w < c$. This suggests that the node w in $G^\star$ has at most $c - 1$ neighbors. By Definition 1, $G^\star$ at most belongs to the $(c - 1)$-core. By Lemma 2, the core number of any nodes in G decreases by at most 1 after deleting an edge. Therefore, the core numbers of the nodes in V_c with color 0 decrease by 1. This completes the proof. □

UpdateDelete which is depicted in Algorithm 7 is used to update the core numbers of the nodes in V_c with color 0 to $c - 1$, because only the core numbers of those nodes need to decrease by 1 after the coloring and recoloring steps. The correctness of **Deletion** can be guaranteed by Theorem 1 and Theorem 3. By a similar analysis as the edge insertion case, the time complexity of **Deletion**(G, u, v) is $O(|V_c| \sum_{u \in V_c} D_u)$. The following example explains how **Deletion** works.

Example 3. Let's consider the graph depicted in Fig. 2. Suppose that we delete the edge (v_8, v_{10}). Since $C_{v_8} = C_{v_{10}} = c = 3$, the **Deletion** algorithm first invokes **Color**(G, v_8, 3), which results in $V_c = \{v_8\}$. Clearly, the color of v_{10} is 0 after this process ends. Hence, the algorithm invokes **Color**(G, v_{10}, 3), which leads to $V_c = \{v_8, v_{10}, v_9, v_2, v_{18}\}$. After this process, all the nodes in V_c are colored by 1 and other nodes are colored by

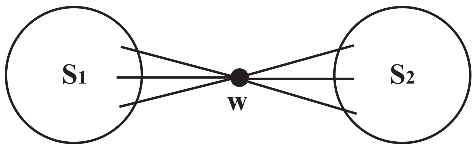


Fig. 3. Toy induced core subgraph.

0. Then, the algorithm invokes **RecolorDelete**(G, 3). At the first recursion, since $\tilde{X}_{v_8} = 2$, v_8 is recolored by 0. Similarly, v_{10}, v_9, v_2, and v_{18} will be recolored by 0 at the first recursion. At the second recursion, the algorithm terminates because no node needs to be recolored. Therefore, all the nodes in V_c are recolored by 0. Finally, the algorithm invokes **UpdateDelete**(G, c) to decrease the core numbers of all the nodes in V_c to 2.

3.2 Pruning Strategies

As analysis in the previous subsection, the time complexity of the **Insertion** and **Deletion** algorithms depend on the size of V_c. In this subsection, to further accelerate our algorithms, we devise two pruning techniques, namely X-pruning and Y-pruning, to remove the nodes in V_c whose core numbers are definitely unchanged given the graph is updated.

X**-pruning:** By Lemma 1, for a node w, X_w is an upper bound of C_w. Here we make use of such upper bound to develop pruning technique. We refer to it as X-pruning. Below, we discuss the X-pruning technique over the edge insertion and edge deletion cases, respectively.

First, we consider the insertion case. Assume that we insert an edge (u_0, v_0). Also, we need to consider three cases, $C_{u_0} > C_{v_0}$, $C_{u_0} < C_{v_0}$, $C_{u_0} = C_{v_0}$. Below, we mainly focus on describing the X-pruning rule under the case of $C_{u_0} = C_{v_0}$, and similar descriptions can be used for the other two cases. For a node w in V_c, after inserting an edge (u_0, v_0), if X_w equals to c, then C_w cannot increase to $c+1$. As a result, we can safely prune w. For example, consider an graph in Fig. 2. Assume that we insert an edge (v_8, v_{10}). Then, for the node v_1, we have $X_{v_1} = 2$. Clearly, C_{v_1} cannot increase to 3, thereby we can prune v_1.

In effect, after removing w, for the nodes that cannot be reachable from u_0 and v_0 in the induced core subgraph can also be pruned. Let us consider a toy induced core subgraph shown in Fig. 3. Suppose that the induced core subgraph can be partitioned into three parts, S_1, w, and S_2. Further, we assume that both u_0 and v_0 are in S_1, and $X_w = c$. Recall that after inserting an edge (u_0, v_0), if $X_w = c$, then C_w is unchanged. By Lemma 4, w will not affect the core numbers of the nodes in S_2. As a consequence, the core numbers of the nodes in S_2 cannot be increased, and we can safely prune all the nodes in S_2. More formally, we give a pruning theorem as follows. The proof can be easily obtained, we omit it for brevity.

Theorem 4. *Given a graph G and an edge (u_0, v_0). After inserting an edge (u_0, v_0) in G, for a node $w \in V_c$ and $X_w < c+1$, we have the following pruning rules.*

- *If $C_{u_0} > C_{v_0}$ (i.e., $V_c = V_{v_0}$), then for any node $u \in V_c$ that every path from v_0 to u in G_{v_0} must go through w can be pruned.*
- *If $C_{u_0} < C_{v_0}$ (i.e., $V_c = V_{u_0}$), then for any node $u \in V_c$ that every path from u_0 to u in G_{u_0} must go through w can be pruned.*
- *If $C_{u_0} = C_{v_0}$ (i.e., $V_c = V_{u_0 \cup v_0}$), then for any node $u \in V_c$ that every path either from u_0 to u or from v_0 to u in $G_{u_0 \cup v_0}$ must go through w can be pruned.*

Algorithm 8 void **XPruneColor**(G, u, c)

```
1: Initialize a queue $Q$;
2: $Q.enqueue(u)$; visited($u$) $\leftarrow 1$;
3: while $Q$ is not empty do
4:     $u \leftarrow Q.dequeue()$;
5:     Compute $X_u$;
6:     if $X_u > c$ then
7:         for each node $w \in N(u)$ do
8:             if visited($w$) $= 0$ and $C_w = c$ then
9:                 $Q.enqueue(w)$; visited($w$) $\leftarrow 1$;
10:        if color($u$) = 0 then
11:            $V_c \leftarrow V_c \cup \{u\}$; color($u$) = 1;
```

Based on Theorem 4, we can prune certain nodes in the coloring procedure (the **Color** algorithm). We present our new coloring algorithm with X-pruning in Algorithm 8. The new coloring algorithm is still a BFS algorithm. The algorithm first calculates X_u when it visits a node u (line 4-5 in Algorithm 8). If $X_u \leq c$, the BFS algorithm does not need to traverse u by Theorem 4. If $X_u > c$, the algorithm visits u's neighbors to check whether they are in V_c or not (line 6-9 in Algorithm 8), and then the algorithm adds node u into V_c and color it by 1 (line 10-11 in Algorithm 8). To implement this pruning strategy, we can replace the **Color** algorithm with the **XPruneColor** algorithm in Algorithm 1.

Second, we consider the edge deletion case. Suppose that we delete an edge (u_0, v_0) from graph G and the core numbers of all the nodes in V_c are c. We consider three different cases: (1) $C_{u_0} > C_{v_0}$, (2) $C_{u_0} < C_{v_0}$, and (3) $C_{u_0} = C_{v_0}$. For $C_{u_0} > C_{v_0}$, we only need to find the nodes in G_{v_0}, because the deletion of edge (u_0, v_0) does not affect the core numbers of the nodes in G_{u_0}. Recall that after deleting an edge, the core numbers of the nodes in V_c decrease by at most 1. Therefore, after deleting an edge (u_0, v_0), if $X_{v_0} \geq c$, then v_0's core number will not be changed. This is because $X_{v_0} \geq c$ implies v_0 has at least c neighbors whose core numbers are larger than or equal to c. That is to say, the core number of node v_0 is still c. Since v_0's core number does not change, we do not need to update the core numbers of the nodes in G_{v_0}. As a result, under the case of $C_u > C_v$ in Algorithm 5 (line 4 in Algorithm 5), we can first compute X_v. If $X_v \geq c$, we do nothing. Symmetrically, for $C_{u_0} < C_{v_0}$, we have a similar pruning rule as the case of $C_{u_0} > C_{v_0}$. For $C_{u_0} = C_{v_0}$, we first compute X_{u_0} and X_{v_0}. If $X_{u_0} < c$, then we need to update the core numbers of the nodes in G_{u_0}. Also, if $X_{v_0} < c$, we update the core numbers of the nodes in G_{v_0}. For the case that $X_{u_0} \geq c$ and $X_{v_0} \geq c$, we do nothing, because no node's core number needs to be updated. It is worth mentioning that X_{u_0} and X_{v_0} are computed based on the core numbers of the nodes that have not been updated. The detailed algorithm with X-pruning for the edge deletion case can be easily implemented, we thus omit it for brevity.

Y**-pruning:** For a node w, Y_w is a lower bound of C_w by Lemma 1. Here we develop a pruning technique using such lower bound, and we refer to this pruning technique as Y-pruning.

To illustrate our idea, let us reconsider the toy induced core subgraph shown in Fig. 3 which includes three parts, S_1, w, and S_2. Suppose that we insert or delete an edge (u_0, v_0). Below, we focus on the case of $C_{u_0} = C_{v_0} = c$, and similar descriptions can be used for other two cases. Further, we assume that both u_0 and v_0 are in S_1, and $Y_w = c$. First, we consider the insertion case, i.e., an edge (u_0, v_0) insertion. In this case, we claim that the core numbers of the nodes in S_2 are unchanged. The reason is as follow. Let u in S_2 be a neighbor node of w. Then, for any neighbor u, we have $Y_u < c$ (if not, u and w will be in a $(c+1)$-core). This implies that for each neighbor of w in S_2, the core number cannot increase to $c+1$ after inserting (u_0, v_0). As a result, the core numbers of all the nodes in S_2 will not change after inserting (u_0, v_0). Second, for the deletion case, if we delete an edge (u_0, v_0), C_w is still equal to c because w has c neighbors whose core numbers are larger than c ($Y_w = c$). Clearly, the core numbers of the nodes in S_2 are also unchanged. Put it all together, under both edge insertion and edge deletion cases, the core numbers of all the nodes in S_2 will not change, and thereby we can safely prune the nodes in S_2. Formally, for Y-pruning, we have the following theorem.

Theorem 5. *Given a graph G and an edge (u_0, v_0). After inserting/deleting an edge (u_0, v_0) in G, for a node $w \in V_c$, if $Y_w = c$, then we have the following pruning rules.*

- *If $C_{u_0} > C_{v_0}$ (i.e., $V_c = V_{v_0}$), then for any node $u \in V_c$ and $u \neq w$ that every path from v_0 to u in G_{v_0} must go through w can be pruned.*
- *If $C_{u_0} < C_{v_0}$ (i.e., $V_c = V_{u_0}$), then for any node $u \in V_c$ and $u \neq w$ that every path from u_0 to u in G_{u_0} must go through w can be pruned.*
- *If $C_{u_0} = C_{v_0}$ (i.e., $V_c = V_{u_0 \cup v_0}$), then for any node $u \in V_c$ and $u \neq w$ that every path either from u_0 to u or from v_0 to u in $G_{u_0 \cup v_0}$ must go through w can be pruned.*

Proof. We prove this theorem under the case $C_{u_0} = C_{v_0}$, and for other cases, we have similar proofs. Below, we discuss the proofs for the edge insertion and edge deletion cases, respectively.

First, we prove the edge insertion case. Let $V_{>c}$ be a set of nodes whose core numbers are larger than c. Assume that we remove w from V_c. Then, after removing w, we denote a set of nodes in V_c that cannot be reachable either from u_0 or from v_0 as V_1. Then, after inserting an edge (u_0, v_0), we consider two cases: (1) w's core number will not change, and (2) w's core number increases by 1. The first case suggests that w is still in the c-core, and we can safely remove w from V_c. Therefore, for the nodes in V_1, we can also remove them from V_c, because only the core numbers of those nodes that are reachable from u_0 or v_0 may need to be updated. Second, we consider the case that w's core number increases by 1 after inserting an edge (u_0, v_0). We denote a subset of nodes in V_c whose core numbers increase by 1 as $\tilde{V}_c$ after inserting an edge (u_0, v_0). Further, we denote a subset of nodes in V_1 whose core numbers need to increase by 1 as V_2. In other words, $V_2 = V_1 \bigcap \tilde{V}_c$. Clearly, the theorem holds if $V_2 = \emptyset$. Now we prove this by contradiction. Specifically, we assume that $V_2 \neq \emptyset$. By definition, after inserting an edge (u_0, v_0), the induced subgraph by the nodes in $\tilde{V}_c \bigcup V_{>c}$ forms a $(k+1)$-core. We denote such subgraph as $G' = (V', E')$, where $V' = \tilde{V}_c \bigcup V_{>c}$. Clearly, all the nodes in G' has at least a degree $c+1$. Now consider a subgraph $G^\star$ induced by the nodes in $V_2 \bigcup \{w\} \bigcup V_{>c}$. We claim that all the nodes in $G^\star$ has at least a degree $c+1$. First, for the nodes in $V_{>c}$, their degree is obviously greater than $c+1$ w.r.t. $G^\star$. Second, we consider the nodes in V_2. By definition, in graph G', there is no edge between the nodes in V_2 and the nodes in $\tilde{V}_c \backslash \{V_2 \bigcup \{w\}\}$. Since the nodes in V_2 have at least a degree $c+1$ w.r.t. graph G', they also have at least a degree $c+1$ w.r.t. graph $G^\star$. Third, we consider the node w. On the one hand, we claim that w has at least one neighbor in V_2. Suppose w has no neighbor in V_2, then the nodes in V_2 whose core numbers cannot increase to $c+1$ after inserting an edge (u_0, v_0) by the k-core update theorem, which contradict to our assumption. Hence, w has at least one neighbor in V_2. On the other hand, since $Y_w = c$, w has c neighbors whose core numbers are larger than c. As a result, w has at least a degree $c+1$ w.r.t. graph $G^\star$. Put it all together, all the nodes in $G^\star$ have at least a degree $c+1$. Note that by our definition the induced subgraph $G^\star$ does not contain node u_0 and v_0. Consequently, before inserting the edge (u_0, v_0), the core numbers of the nodes in $G^\star$ are at least $c+1$. That is to say, the nodes in V_2 has core number $c+1$ before inserting the edge (u_0, v_0), which is a contradiction. This completes the proof for the edge insertion case.

For the edge deletion case, after deleing an edge (u_0, v_0), the core numbers of all the nodes in V_c decrease by at most 1 according to Lemma 2. Hence, if a node $w \in V_c$ has $Y_w = c$, then w's core number will not decrease. Similarly, let $V_{>c}$ be a set of nodes whose core numbers are larger than c. And assume that we remove w from V_c. Then, after removing w, we denote a set of nodes in V_c that cannot be reachable either from u_0 or from v_0 as V_1. Now consider a subgraph $G^\star$ induced by the nodes $V_1 \bigcup \{w\} \bigcup V_{>c}$. We claim that all the nodes in such subgraph have at least a degree c. First, for the nodes in $V_{>c}$, their degree is clearly larger than c w.r.t. $G^\star$ because their core numbers are larger than c. Second, w's degree is at least c w.r.t. $G^\star$, because w has c neighbors whose core numbers are larger than c. Third, for the nodes in V_1, their degree is also at least c w.r.t. $G^\star$. The rationale is as follows. By definition, no edge in G goes through the nodes in $V_c \backslash \{V_1 \cup \{w\}\}$ and the nodes in V_1. Since the core numbers of the nodes in V_1 are c, the nodes in V_1 has at least c neighbors w.r.t. $G^\star$. Consequently, the core numbers of the nodes in $G^\star$ are still c after removing the edge (u_0, v_0). This implies that the nodes in V_1 can be pruned, which completes the proof for the edge deletion case. □

Based on Theorem 5, we can implement the Y-pruning strategy in the coloring procedure. We present our new coloring algorithm with Y-pruning in Algorithm 9, which is

Algorithm 9 void **YPruneColor**(G, u, c)

```
1:  Initialize a queue $Q$;
2:  $Q.enqueue(u)$; visited($u$) $\leftarrow 1$;
3:  while $Q$ is not empty do
4:      $u \leftarrow Q.dequeue()$;
5:      Compute $Y_u$;
6:      if $Y_u < c$ then
7:          for each node $w \in N(u)$ do
8:              if visited($w$) $= 0$ and $C_w = c$ then
9:                  $Q.enqueue(w)$; visited($w$) $\leftarrow 1$;
10:     if color($u$) = 0 then
11:         $V_c \leftarrow V_c \cup \{u\}$; color($u$) = 1;
```

also a BFS algorithm. In particular, when the algorithm visits a node u, it first calculates Y_u (line 4-5 in Algorithm 9). If $Y_u < c$, the algorithm visits u's neighbors to check whether they are in V_c or not (line 6-9 in Algorithm 9) or not. If $Y_u = c$, the algorithm does not need to traverse u's neighbors by Theorem 5. Under both cases (i.e., $Y_u < c$ and $Y_u = c$), the algorithm adds u into V_c, and color it by 1 (line 10-11 in Algorithm 9). Below, we discuss how to integrate the **YPruneColor** algorithm into the **Insertion** and **Deletion** algorithm.

First, to integrate the **YPruneColor** algorithm into the **Insertion** algorithm, we need to replace the **Color** algorithm with the **YPruneColor** algorithm as well as handle the following special case. That is, if $C_{u_0} = C_{v_0} = c$, $Y_{u_0} = c$ and $Y_{v_0} < c$, we need to invoke **YPruneColor**(G, v_0, c). If $C_{u_0} = C_{v_0} = c$, $Y_{u_0} < c$ and $Y_{v_0} = c$, we have to invoke **YPruneColor**(G, u_0, c). The reason is because we need to allow the BFS algorithm to go through the edge (u_0, v_0) in order to add both u_0 and v_0 into V_c. If $C_{u_0} = C_{v_0} = c$ and $Y_{u_0} = Y_{v_0} = c$, then we have to invoke both **YPruneColor**(G, u_0, c) and **YPruneColor**(G, v_0, c) so as to add both u_0 and v_0 into V_c. Second, to integrate the **YPruneColor** algorithm into the **Deletion** algorithm, we only need to replace the **Color** algorithm with the **YPruneColor** algorithm.

Combination of X-pruning and Y-pruning: Here we discuss how to combine both X-pruning and Y-pruning for edge insertion case and edge deletion case, respectively. For edge insertion case, we can integrate both X-pruning and Y-pruning into the coloring procedure. Specifically, in the coloring procedure, when the BFS algorithm visits a node u, we calculate both X_u and Y_u. Then, we use the X-pruning rule to determine the color of node u, and make use of both X-pruning and Y-pruning rules to determine whether the algorithm visits u's neighbors or not. For the edge deletion case, we can easily integrate both X-pruning and Y-pruning via the following two steps. First, we replace the **Color** algorithm in **Deletion** with the **YPruneColor** algorithm. Second, we integrate the X-pruning rule into the **Deletion** algorithm.

3.3 Discussions

Here we present an extension of the proposed algorithm with a cache strategy to handle very large graphs which cannot fit into internal memory. We assume that the internal memory can hold $\Theta(n)$ data, but cannot hold all the edges of the graph. Note that this assumption typically holds for most publicly available real-world networks. For example, a very common PC with 4G internal memory can hold a network with 100 million nodes. Under this assumption, the core number of each node and all the other arrays with size $\Theta(n)$ in our algorithm (e.g., the visited array, the color array, and so on) can be stored in internal memory. Further, we assume that the graph is represented by adjacency lists. By our assumptions, the adjacency list of each node can be loaded in internal memory by constant I/O. Indeed, if we set the buffer size as $\Theta(n)$, each node's adjacency list can be loaded in memory by one I/O. In the worse case, it is easy to derive that the coloring algorithm (Algorithms 2,8, and 9) takes at most $O(|V_c|)$ I/O. Notice that with both X-pruning and Y-pruning, $|V_c|$ is very small for most real-world graphs as observed in the experiments, thus the coloring algorithm is very efficient. To further reduce the I/O cost of the coloring algorithm, we can maintain the low-degree nodes' adjacency lists in the internal memory when the memory is sufficient. Since most nodes in the real-world networks are low-degree nodes, we can cache a large number of low-degree nodes' adjacency lists, thus cutting the I/O cost.

For the recoloring algorithm (Algorithms 3 and 6), we adopt the following cache strategy. For a node u in V_c, we cache u's neighbors with core number larger than c. Note that this can be done by a simple modification of the coloring algorithm. That is to say, we cache several *partial* adjacency lists of the nodes in $|V_c|$. In effect, since $|V_c|$ is very small, all such *partial* adjacency lists of the nodes in $|V_c|$, in many cases, can be cached in internal memory. Under these cases, the recoloring algorithm will not take any I/O cost. However, when the internal memory cannot cache all the *partial* adjacency lists, we apply the well-known LRU (least recently used) replacement strategy to replace the blocks. In addition, we use the following strategy to further reduce the I/O cost. In each iteration of the recoloring algorithm, when a node is recolored, we can safely delete this node's *partial* adjacency list from the internal memory. For the core number updating algorithms (Algorithms 4 and 7), no I/O cost is taken, as V_c can hold in internal memory.

TABLE 1
Summary of the Datasets

Name	#. of nodes	#. of edges	Ref.
EmailEnron	36,692	367,662	[18]
Slashdot	82,168	867,372	[18]
Gowalla	196,591	1,900,654	[18]
Pokec	1,632,803	61,245,128	[18]
Orkut	3,072,441	117,185,083	[18]
Wikipedia	15,172,739	260,332,502	[19]
Flickr	2,302,925	33,140,018	[20]
Youtube	3,223,643	18,524,095	[21]

4 Experiments

Different algorithms: We evaluate five different algorithms. The baseline algorithm is the algorithm that invokes the $O(n + m)$ algorithm to update the core numbers when the internal memory is sufficient [17]. However, when the memory is insufficient, the baseline algorithm invokes the external k-core decomposition algorithm proposed in [14]

TABLE 2
Average Update Time of Different Algorithms

Time	Average deletion time					Average insertion time					Average update time				
	B	N	X	Y	XY	B	N	X	Y	XY	B	N	X	Y	XY
EmailEnron	10.8	2.4	0.9	1.8	**0.9**	10.6	2.9	2.7	2.8	**2.7**	10.7	2.7	1.8	2.3	**1.8**
Slashdot	24.4	4.9	1.6	2.1	**1.5**	22.3	2.9	1.8	2.1	**1.6**	23.3	3.7	1.7	2.1	**1.6**
Gowalla	108.2	2.1	1.1	1.8	**0.9**	107.5	1.7	1.5	1.6	**1.2**	107.9	1.9	1.3	1.7	**1.1**
Pokec	68.3*s*	12.1	8.3	10.5	**6.5**	72.8*s*	12.4	8.6	11.3	**7.4**	70.6*s*	12.3	8.5	10.9	**6.9**
Flickr	55.6*s*	9.2	7.6	8.3	**5.3**	57.3*s*	10.4	8.1	9.0	**5.8**	56.5*s*	9.8	7.9	8.7	**5.6**
Youtube	49.3*s*	10.3	8.2	9.6	**5.9**	51.5*s*	10.6	8.6	10.2	**6.3**	50.4*s*	10.5	8.4	9.9	**6.1**

The default time unit is millisecond, and 's' denotes second.

to update the core numbers. We denote the baseline algorithm as algorithm B. For the proposed algorithms, we have four variants. The first algorithm, denoted as algorithm N, is the basic algorithm without pruning strategy. The other three algorithms, denoted as algorithm X, Y, and XY, are the basic algorithm with X-pruning, Y-pruning, and both X and Y-pruning respectively.

Datasets: In the experiments, we collect eight real-world datasets which are summarized in Table 1. The first four datasets (Slashdot, Gowalla, WebGoogle, and Pokec) are static networks datasets which can be holden in internal memory. The Orkut and Wikipedia are two massive static social network datasets which cannot be completely holden in internal memory, but the nodes of these networks can be stored in memory. The last two datasets (Flickr and Youtube) are real-world dynamic social network datasets where the temporal information of each edge is available. We get the first five datasets from Stanford network data collections [18], the Wikipedia dataset from [19], and the last two datasets from [20] and [21] respectively.

Experimental environment: We conduct the experiments on a Windows Server 2007 with 4xDual-Core Intel Xeon 2.66 GHz CPU, and 8G memory. All algorithms are implemented in C++.

4.1 Experimental Results

Without specifically stated, in all the experiments, we randomly delete and insert 500 edges for the original static datasets. For each dynamic dataset (Flickr and Youtube), we randomly choose a time point t to extract two snapshots, say G_1 and G_2. Note that G_1 is a subgraph of G_2, i.e., G_2 includes edges that are inserted after time t. Since the original dynamic datasets only record the edge insertion time [20], [21], for a fair comparison with the static networks, we consider the first 500 edge insertions in G_2 (w.r.t. G_1), and then randomly delete them to generate 500 edge deletions. For all the datasets, we invoke five different algorithms to update the core numbers of the nodes after an edge update. For all algorithms, we record three quantities, namely average insertion time, average deletion time, and average update time, to measure the efficiency. We calculate the average insertion (deletion) time by the average core number update time of different algorithms over all the edge insertions (deletions). The average update time is the mean of average insertion time and average deletion time. To evaluate the efficiency of our algorithms (algorithm N, algorithm X, algorithm Y, algorithm XY), we compare them with the baseline algorithm (algorithm B) according to the average insertion/deletion/update time.

We first discuss the case that the internal memory is sufficient to hold the graph. The results under this case are reported in Table 2. From Table 2, we can see that all of our algorithms (algorithm N, algorithm X, algorithm Y, algorithm XY) perform much better than the baseline algorithm (algorithm B) over all the datasets used. The best algorithm is the algorithm XY, which is the basic algorithm with both X-pruning and Y-pruning, followed by algorithm X, algorithm Y, algorithm N, and algorithm B. Over all the datasets used, the maximal speedup of our algorithms is achieved in Pokec dataset (the fourth row in Table 2). Specifically, in Pokec dataset, algorithm XY, algorithm X, algorithm Y and algorithm N reduce the average update time of algorithm B by 10231, 8035, 6724, and 5739 times respectively. In general, we find that the speedup of our algorithms increases as the graph size increases. This is because the time complexity of the baseline algorithm is linear w.r.t. the graph size for handling each edge insertion/deletion. Instead, the time complexity of our algorithms is independent of the graph size, and it only depends on $|V_c|$, which is the size of the induced core subgraph. Moreover, we find that our basic algorithm with pruning techniques is significantly more efficient than the basic algorithm without pruning technique. In addition, we can observe that the performance of our algorithms in real-world dynamic graphs (Flickr and Youtube) is desired. For example, the XY algorithm cuts the average update time of the baseline algorithm by 10089 and 8262 times in Flickr and Youtube datasets respectively.

Second, we consider the case that the entire graph cannot stored in internal memory. In this experiment, we use 6G memory to cache the low degree nodes' adjacency lists. The results are described in Table 3. Here we only report the average update time, and similar results can be observed for the average insertion/deletion time. In Table 3, the first and third rows denote the total running time of different algorithms, while the second and fourth rows denote the

TABLE 3
Average Update Time of Different Algorithms in Disk-Resident Graphs (*s*: second, *m*: minute)

	B	N	X	Y	XY
Orkut	13.2*m*	5.8*s*	3.2*s*	4.5*s*	**2.8***s*
(I/O)	7.3*m*	≈5.8*s*	≈3.2*s*	≈4.5*s*	≈**2.8***s*
Wikipedia	19.6*m*	1.2*m*	43.5*s*	57.3*s*	**35.2***s*
(I/O)	10.3*m*	≈1.2*m*	≈43.5*s*	≈57.3*s*	≈**35.2***s*

TABLE 4
Average $|V_c|$ of Different Algorithms

Name	N	X	Y	XY
EmailEnron	69.1	36.7	60.9	**33.6**
Slashdot	256.4	126.0	231.0	**113.1**
Gowalla	116.1	50.3	50.4	**28.2**
Pokec	1054.1	707.6	966.3	**605.4**
Orkut	828.3	513.4	664.5	**414.3**
Wikipedia	968.3	635.6	737.2	**537.5**
Flickr	823.5	610.7	705.9	**401.6**
Youtube	906.5	623.2	813.8	**456.3**

I/O time of different algorithms. As can be seen, all of our algorithms outperform the baseline algorithm. Similar to the previous results, the best algorithm is the XY algorithm, followed by the algorithm X, Y, N, and B. The XY algorithm improves the total running time of algorithm B by 283 and 33 times in Orkut and Wikipedia datasets respectively. Moreover, we can observe that the running time of our algorithm is dominated by the I/O time. This is because our algorithms only work on the nodes in V_c which is very small w.r.t. the graph size, thus the CPU time is very small regarding to the I/O time. Additionally, we can find that the performance of our algorithms in Orkut dataset is better than those in Wikipedia dataset. The reason is that the size of the Orkut dataset is significantly smaller than that of the Wikipedia dataset, thus we can maintain much more low-degree nodes' adjacency lists in internal memory to reduce the I/O cost.

The size of V_c: Table 4 reports the average $|V_c|$ of the proposed algorithms in all the datasets used. As can be seen from Table 4, algorithm XY has the smallest average $|V_c|$. The average $|V_c|$ of algorithm X is smaller than that of algorithm Y over all the datasets, indicating that the X-pruning strategy is more powerful than the Y-pruning strategy. As desired, algorithm X, Y, and XY has smaller $|V_c|$ than algorithm N. In general, the average $|V_c|$ for all the proposed algorithms is very small w.r.t. the graph size. Moreover, we can find that $|V_c|$ is independent of the graph size. For example, in Slashdot dataset (the second row), which is a relatively small dataset, the average $|V_c|$ of algorithm XY is 113.1. However, in Gowalla dataset (the third row), which is a relatively large dataset, the average $|V_c|$ of algorithm XY is only 28.2. These results confirm the efficiency of our algorithms as observed in the previous experiment.

The effect of pruning: Here we investigate the effective of the pruning techniques. From Tables 2 and 3, over all the datasets, we can see that the X-pruning strategy (algorithm X) is more effective than the Y-pruning strategy (algorithm Y) according to the average update time. For example, in EmailEnron (row 1 in Table 2) and Orkut datasets (row 1 in Table 3), algorithm X reduces the average update time of algorithm N by 33.3% and 44.8% respectively. However, in the same datasets, algorithm Y reduces the average update time of algorithm N by 14.8% and 22.4% respectively. This result indicates that the condition of the Y-pruning is stronger than the condition of the X-pruning in many real-world graphs. Recall that by Theorem 5, if there is at least one node u with core number C_u and $Y_u = C_u$ in the induced core subgraph, then the Y-pruning strategy may prune some nodes. The condition of Y-pruning strategy ($Y_u = C_u$) is strong, because if a node has C_u neighbors whose core numbers are larger than C_u, then this node may have "another additional neighbor" whose core number is larger than C_u, thus resulting in that the node u is in a (C_u+1)-core. Instead, indicating by the experimental result, the condition of the X-pruning strategy ($X_u \leq C_u + 1$) may be easily satisfied in real-world graphs. This result also implies that the lower bound of the core number in Lemma 1 (Y_v) is typically very loose for many nodes in real-world graphs. In addition, we can see that the algorithm with both X and Y-pruning strategies is more efficient than the algorithm with only one pruning strategy over all the datasets.

TABLE 5
Average $|V_c|$ vs Edge-Updating Strategy

Name	N	X	Y	XY
Random	116.1	50.3	50.4	**28.2**
LCN	152.3	76.2	78.8	**53.1**
HCN	109.6	45.9	48.1	**26.1**

Effect of edge-updating strategy: Here we show how the edge-updating strategy affects the performance of the proposed algorithms. Since the efficiency of our algorithms depends on $|V_c|$, we study $|V_c|$ under three different edge-updating strategies which are random edge-updating, low-core-numbers (HCN) edge updating, and high-core-numbers (HCN) edge updating. Here an LCN (HCN) edge denotes an edge whose endpoints have low (high) core numbers. For all these strategies, we pick 500 edges for insertion and deletion respectively. Table 5 shows the results in Gowalla dataset, and similar results can also be observed from other datasets. As can be seen, by the LCN edge-updating strategy, our algorithms achieve the largest average $|V_c|$ while by the HCN edge-updating strategy, our algorithms get the lowest $|V_c|$. These results indicate that the proposed algorithms work well when the underlying edge-updating distribution favors the HCN edges. In contrast, when such a distribution favors the LCN edges, the performance of our algorithms might be not very good.

Results of a batch of edge updates: In previous experiments, we have shown that the proposed algorithms are very efficient for the core maintenance problem. These algorithms are extremely useful to incrementally update the core numbers of the nodes when the graph evolves over time. Besides the graph with a single edge update, here we show the performance of our algorithms in a dynamic graph given a batch of edge updates. Suppose that the graph has r edge updates at a time interval Δt. To maintain the core numbers of the nodes, we has to sequentially invoke our algorithms r times. For the baseline algorithm (algorithm B), however, we can invoke it once to recompute the core numbers of all nodes. Since algorithm XY is the best algorithm for a single edge update, we only compare algorithm XY with algorithm B. We use the *relative running time* as a metric to evaluate algorithm XY. In particular, let t be the running time of algorithm XY, t_B be the running time of the baseline algorithm (algorithm B). Then, the relative running time of algorithm XY is defined by $\delta = t/t_B$. Obviously, the smaller δ the algorithm achieves, the more

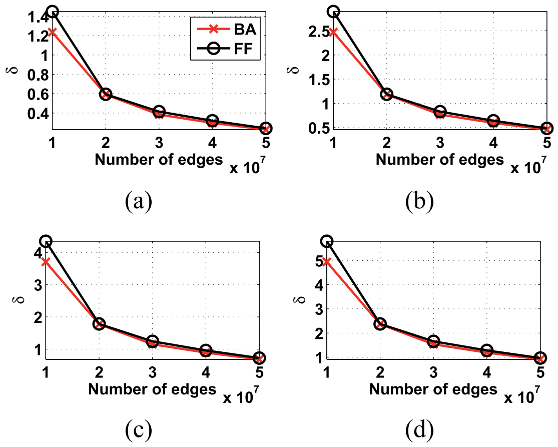


Fig. 4. Efficiency of algorithm XY under various r.

efficient the algorithm is. In addition, it is worth noting that if $\delta < 1$, algorithm XY is better than the baseline algorithm, otherwise it is worse than the baseline algorithm.

To evaluate the effect of r w.r.t. the graph size, we generate ten large synthetic graphs, where five graphs are based on the Barabasi-Albert (BA) random graph model [22] and another five graphs based on the forest fire (FF) model [23]. Specifically, for each random graph model, we produce five synthetic graphs $G_1, \ldots, G_5$ with G_i has i million nodes and $10 \times i$ million edges for $i = 1, \ldots, 5$. Note that here we only consider the graphs that can be fitted into internal memory, as our algorithms are very efficient in those graphs. The results of algorithm XY under different r over all the synthetic graphs are shown in Fig. 4. From Fig. 4, we can see that algorithm XY exhibits comparable performance under both BA and FF random graph models. In general, δ decreases as the graph size increases, and δ increases as the increasing r. In addition, we can observe that algorithm XY significantly outperforms the baseline algorithm when $r = 2500$ and the number of edges is no less than 10 million. Moreover, when $r = 10000$, our algorithm is still better than the baseline algorithm when the number of edges is no less than 50 million. That is to say, if such a graph has $r = 10000$ edge-updates in a time interval Δt, our algorithm outperforms the baseline algorithm. Below, we study the effect of Δt in real-world dynamic graphs.

To evaluate the performance of algorithm XY under different Δt, we perform experiments in two real-world dynamic datasets (Flickr and Youtube). Specifically, we extract the snapshots of the dynamic graph based on the time unit 'hour'. Since the time-granularity of the original dynamic datasets is 'day', we uniformly partition the set of edge-updates in a day into 24 subsets. After extracting the snapshots, the average number of edge-updates in one hour in Flickr and Youtube datasets are 3923 and 625 respectively. Fig. 5 reports the efficiency of algorithm XY under different Δt. As can be seen, algorithm XY is more efficient than the baseline algorithm (algorithm B) when $\Delta t \leq 2$ and $\Delta t \leq 12$ in Flickr and Youtube datasets respectively. Clearly, our algorithm performs much better in Youtube dataset, because this dataset evolves slower than the Flickr dataset. These results indicate that for a batch of edge updates, algorithm XY is more efficient than algorithm B when the graph is very large and evolves slowly.

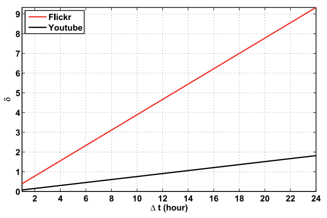


Fig. 5. Efficiency of algorithm XY under different Δt.

5 Related Work

The k-core decomposition in networks has been extensively studied in the literature. In [3], Seidman first introduced the concept of k-core to measure the group cohesion in a network. The cohesion of k-core increases as k increases. Recently, the k-core decomposition in graphs has been used in many applications. Notable examples include visualization of large complex networks [5], [6], [24], [25], uncovering the topological structure of the Internet [7], [8], analysis of the structure of biological networks [9]–[11], studying percolation in random graph [13], [26], and identifying influential spreader in networks [12].

From an algorithmic perspective, Batagelj and Zaversnik proposed an $O(n+m)$ algorithm for k-core decomposition in general graphs [4]. Their algorithm recursively deletes the node with the lowest degree and uses the bin-sort algorithm to maintain the order of the nodes. However, this algorithm has to randomly access the graph, thus it could be inefficient for the disk-resident graphs. To overcome this issue, Cheng *et al.* [14] proposed an efficient k-core decomposition algorithm for disk-resident graphs. Their algorithm works in a top-to-down manner to calculate k-core. To make the k-core decomposition more scalable, Montresor *et al.* [15] proposed a distributed algorithm for k-core decomposition by exploiting the locality property of k-core. All the mentioned algorithms are focus on k-core decomposition in static graph except for [17]. For the dynamic graph, in [17], Miorandi and Pellegrini applied the $O(n+m)$ algorithm [4] to recompute the core numbers of the nodes when the graph is updated, which is inefficient in large graphs. In this paper, we propose a more efficient algorithm than their algorithm to maintain the core numbers of the nodes in a dynamic graph online.

6 Conclusion

In this paper, we propose an efficient algorithm for maintaining the core numbers of nodes in dynamic graphs. For a node u, we define a notion of induced core subgraph G_u, which contains the nodes that are reachable from u and have the same core number as u. Given a graph G and an edge (u, v), we find that only the core numbers of nodes in G_u or G_v or $G_{u \cup v}$ may need to be updated after inserting/deleing the edge (u, v). Based on this, first, we introduce a coloring algorithm to identify all of these nodes. Second, we devise a recoloring algorithm to determine the nodes whose core numbers definitely need to be updated. Finally, we update the core numbers of such nodes by a linear algorithm. In addition, we develop two pruning strategies, namely X-pruning and Y-pruning, to further accelerate the algorithm. We perform extensive experiments to evaluate the proposed algorithm, and the results demonstrate the efficiency of our algorithm.

ACKNOWLEDGMENTS

This work was supported in part by the grant of the Research Grants Council of Hong Kong SAR, China CUHK/ 418512, and in part by NSFC project 61170076, and China-863 project 2012AA01A309.

REFERENCES

[1] R. A. Hanneman and M. Riddle. (2005). *Introduction to Social Network Methods* [Online]. Available: http://faculty.ucr.edu/~hanneman/nettext

[2] J. Cohen, “Trusses: Cohesive subgraphs for social network analysis,” National Security Agency, Fort Meade, MD, USA, Tech. Rep., 2005.

[3] S. B. Seidman, “Network structure and minimum degree,” *Soc. Netw.*, vol. 5, no. 3, pp. 269–287, 1983.

[4] V. Batagelj and M. Zaversnik. (2003). An O(m) algorithm for cores decomposition of networks. CoRR, vol. cs.DS/0310049 [Online]. Available: http://arxiv.org/abs/cs.DS/0310049

[5] V. Batagelj, A. Mrvar, and M. Zaversnik, “Partitioning approach to visualization of large graphs,” in *Proc. Graph Drawing*, 1999, pp. 90–97.

[6] J. I. Alvarez-Hamelin, L. Dall’Asta, A. Barrat, and A. Vespignani, “Large scale networks fingerprinting and visualization using the k-core decomposition,” in *Proc. NIPS*, 2005.

[7] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, and E. Shir, “A model of internet topology using k-shell decomposition,” in *Proc. Nat. Acad. Sci. USA*, vol. 104, no. 27, pp. 11150–11154, 2007.

[8] J. I. Alvarez-Hamelin, L. Dall’Asta, A. Barrat, and A. Vespignani, “K-core decomposition of internet graphs: Hierarchies, self-similarity and measurement biases,” *Netw. Heterogeneous Media*, vol. 3, no. 2, pp. 371–393, 2008.

[9] M. Kitsak *et al.*, “An automated method for finding molecular complexes in large protein interaction networks,” *BMC Bioinform.*, vol. 4, Article 2, Jan. 2003.

[10] M. Altaf-Ul-Amin *et al.*, “Prediction of protein functions based on k-cores of protein-protein interaction networks and amino acid sequences,” *Genome Inform.*, vol. 14, pp. 498–499, 2003.

[11] S. Wuchty and E. Almaas, “Peeling the yeast protein network,” *Proteomics*, vol. 5, no. 2, pp. 444–449, 2005.

[12] M. Kitsak *et al.*, “Identification of influential spreaders in complex networks,” *Nature Phys.*, vol. 6, pp. 888–893, Aug. 2010.

[13] A. V. Goltsev, S. N. Dorogovtsev, and J. F. F. Mendes. (2006). k-core (bootstrap) percolation on complex networks: Critical phenomena and nonlocal effects. CoRR, vol. abs/cond-mat/0602611 [Online]. Available: http://arxiv.org/abs/cond-mat/0602611

[14] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, “Efficient core decomposition in massive networks,” in *Proc. IEEE 27th ICDE*, Hanover, Germany, 2011.

[15] A. Montresor, F. D. Pellegrini, and D. Miorandi, “Distributed k-core decomposition,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2, pp. 288–300, Feb. 2013.

[16] T. von Landesberger *et al.*, “Visual analysis of large graphs: State-of-the-art and future research challenges,” *Comput. Graph. Forum*, vol. 30, no. 6, pp. 1719–1749, 2011.

[17] D. Miorandi and F. D. Pellegrini, “K-shell decomposition for dynamic complex networks,” in *Proc. 8th Int. Symp. WiOpt*, Avignon, France, 2010.

[18] J. Leskovec. (2010). *Standford Network Analysis Project* [Online]. Available: http://snap.standford.edu

[19] S. Auer *et al.*, “DBpedia: A nucleus for a web of open data,” in *Proc. Int. Semantic Web Conf.*, Busan, Korea, 2008, pp. 722–735.

[20] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, “Growth of the Flickr social network,” in *Proc. Workshop Online Social Netw.*, Washington, DC, USA, 2008, pp. 25–30.

[21] A. Mislove, “Online social networks: Measurement, analysis, and applications to distributed information systems,” Ph.D. dissertation, Rice Univ., Houston, TX, USA, 2009.

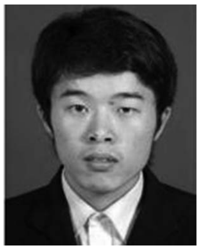
[22] A.-L. Barabasi and R. Albert, “Emergence of scaling in random networks,” *Sci.*, vol. 286, no. 5439, pp. 509–512, 1999.

[23] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, “Graphs over time: Densification laws, shrinking diameters and possible explanations,” in *Proc. 11th ACM SIGKDD Int. Conf. KDD*, Chicago, IL, USA, 2005, pp. 177–187.

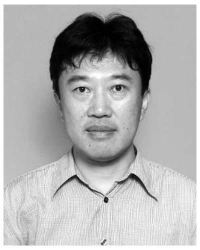
[24] M. Baur, U. Brandes, M. Gaertler, and D. Wagner, “Drawing the AS graph in 2.5 dimensions,” in *Proc. Graph Drawing*, New York, NY, USA, 2004, pp. 43–48.

[25] Y. Zhang and S. Parthasarathy, “Extracting, analyzing and visualizing triangle k-core motifs within networks,” in *Proc. IEEE 28th ICDE*, Washington, DC, USA, 2012.

[26] S. N. Dorogovtsev, A. V. Goltsev, and J. F. F. Mendes, “k-core organization of complex networks,” *Phys. Rev. Lett.*, vol. 96, no. 4, Article 040601, 2006.



Rong-Hua Li received the PhD degree from the Chinese University of Hong Kong in 2013. He is currently an Assistant Professor at Shenzhen University, China. His research interests include algorithmic aspects of social network analysis, big graph data management and mining.



Jeffrey Xu Yu received the B.E., M.E., and Ph.D. degrees in computer science from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. He has held teaching positions at the Institute of Information Sciences and Electronics, University of Tsukuba, and at the Department of Computer Science, Australian National University, Australia. Currently, he is a Professor in the Department of Systems Engineering and Engineering Management, the Chinese University of Hong Kong, Hong Kong. He is serving as a *VLDB Journal* Editorial Board Member. His current research interests include graph database, graph mining, keyword search in relational databases, and social network analysis.

Rui Mao received the Ph.D. degree in computer science from the University of Texas at Austin, TX, USA, in 2007. He is currently an Associate Professor at Shen Zhen University, China. His current research interests include big data analysis and management, content-based similarity query of multimedia and biological data, data mining, and machine learning.