Federated k-Core Decomposition: A Secure Distributed Approach

Bin Guo[1], Emil Sekerinski[2], and Lingyang Chu[2]

[1] Trent University, Department of Computer Science, Peterborough, ON K9L 0G2, Canada, binguo@trentu.ca
[2] McMaster University, Department of Computing & Software, Hamilton, ON L8S 4L8, Canada, emil@mcmaster.ca, chul9@mcmaster.ca

Abstract. As one of the most well-studied cohesive subgraph models, the k-core is widely used to find graph nodes that are "central" or "important" in many applications, such as biological networks, social networks, ecological networks, and financial networks. For distributed networks, e.g., Decentralized Online Social Networks (DOSNs) such that each vertex is a client as a single computing unit, the distributed k-core decomposition algorithms are already proposed. However, current distributed approaches fail to adequately protect privacy and security. In this work, we are the first to propose the secure version of the distributed k-core decomposition.

Keywords: graph, k-core decomposition, distributed, privacy

1 Introduction

Graphs are important data structures that can represent complex relations in many real applications, such as social networks, communication networks, biological networks, hyperlink networks, and model checking networks.

Given an undirected graph G, the *k-core decomposition* is to identify the maximal subgraph G' in which each vertex has a degree of at least k; the *core number* of each vertex u is defined as the maximum value of k such that u is contained in the k-core of G [3, 10]. It is well known that core numbers can be computed with linear running time $O(n+m)$ using the BZ algorithm [3], where n is the number of vertices and m is the number of edges in G. Due to the linear time complexity, the k-core decomposition is easily and widely used in many real-world applications.

In [10], Kong et al. summarize a large number of applications for k-core decomposition in biology, social networks, community detection, ecology, information spreading, etc. Especially in [4], Lesser et al. investigate the k-core robustness in ecological and financial networks. In a survey [14], Malliaros et al. summarize the main research work related to k-core decomposition from 1968 to 2019.

In today's data-driven world, data graphs tend to grow continuously and must be distributed rather than stored in a single machine. In particular, data privacy and security have attracted more and more attention [6,12,17]. Our work is based on the model for multiple *clients* [15], where each client is a single computational unit like a single machine with shared memory and different clients communicate by asynchronous network.

1.1 Data Privacy for *k*-Core Decomposition

Specifically, for k-core decomposition, *privacy* can be defined as follows: a) each client knows its own information; b) each client knows the ID of its directly connected neighbors; c) besides the above two, each client does not know any other information, such as the core numbers of neighbors and the connections of neighbors; d) there does not exist a centralized server for synchronization and storing the information of all clients.

After the k-core decomposition, each vertex successfully obtains its core number. The other important question is how to release the core numbers. Since core numbers can be private information for clients, it is not safe that all vertices report their core numbers when there are queries. We observe that many core number analytics in real graphs [4] only calculate the distribution of core numbers, instead of knowing the specific core number for each vertex. That is, each vertex is assigned a label that indicates its special property. We can receive a query, for example, how many vertices have both the label A and the maximum core number 10, denoted $A10$. In this case, essentially, we only need to release the total number of vertices with $A10$, and thus we can hide the real core number of each vertex.

However, existing traditional distributed k-core decomposition approaches [2, 13, 15] inadequately consider the data privacy and security for three problems. First, a centralized server connects to all clients to detect the termination k-core decomposition, by which the server may know all the IDs of the clients. Second, a client has to store all the core numbers of neighbors, which explicitly exposes its core numbers to neighbors. Third, there are no strategies to safely release the core numbers of vertices after core decomposition. In this work, we try to overcome these problems.

1.2 Our Contributions

In this paper, we improve the existing distributed k-core decomposition algorithm in [15] by preserving data privacy and security on "Secure-One-to-One", the so-called *federated k-core decomposition*. The general framework is shown in Fig. 1. Three approaches are proposed to improve privacy and security, which are our main contributions.

– We propose Homomorphic Encryption (HE) [1] to compare a pair of core numbers for two vertices without leaking the value of the core numbers to other vertices.
– We propose decentralized termination detection to identify whether the computation is complete or not.

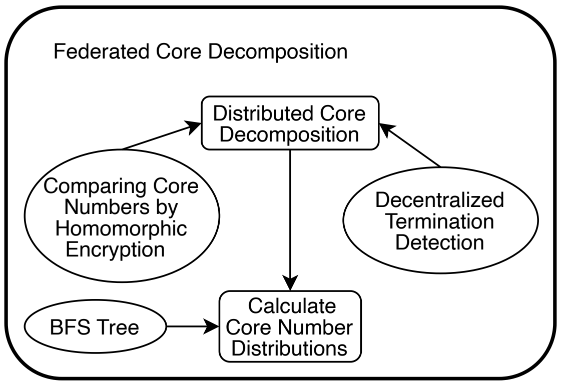

Federated Core Decomposition
Distributed Core Decomposition
Comparing Core Numbers by Homomorphic Encryption
Decentralized Termination Detection
BFS Tree
Calculate Core Number Distributions

Fig. 1: The framework of Our Federated k-Core Decomposition.

- After termination, all the core numbers of vertices have already been calculated. We propose a method to safely release the distribution of the core numbers for the vertices with the same labels, instead of the exact values of the core numbers.

2 Related Work

In [3], Batagelj et al. propose a linear time $O(m + n)$ algorithm for k-core decomposition, the so-called BZ algorithm. In [15], Montresor et al. first propose a distributed k-core decomposition algorithm that can handle distributed graphs. In [13], Luo et al. extend distributed k-core decomposition algorithms to probabilistic graphs. In [5], Chan et al. study the approximate distributed k-core decomposition. In [11], Liao et al. explore the distributed D-core decomposition specifically in directed graphs; here, D-core, also called (k, l)-core, is a directed version of k-core, the maximal directed subgraph such that every vertex has at least k in-neighbors and l out-neighbors. These traditional distributed approaches never consider privacy and security during computation.

3 Preliminary

Let $G = (V, E)$ be an undirected unweighted graph, where $V(G)$ denotes the set of vertices and $E(G)$ represents the set of edges in G. When the context is clear, we will use V and E instead of $V(G)$ and $E(G)$ for simplicity, respectively. As G is an undirected graph, an edge $(u, v) \in E(G)$ is equivalent to $(v, u) \in E(G)$. The set of neighbors of a vertex $u \in V$ is defined by $u.adj = \{v \in V : (u, v) \in E\}$. The degree of a vertex $u \in V$ is denoted by $u.deg = |u.adj|$. We denote the number of vertices and edges of G by n and m, respectively, in the context of analyzing time, space, and message complexities. We denote a set of messages for the communication of vertices by $M = \{m_1, m_2, m_3, ...\}$.

Furthermore, each vertex $u \in V$ has a label to indicate the local feature, which is defined by $u.lb$. The set $\mathcal{L}$ includes all the possible labels that $u \in V$ may have, denoted $\mathcal{L} = \{u.lb \mid u \in V\}$.

Definition 1 (k-Core). *Given an undirected graph $G = (V, E)$ and a natural number k, a induced subgraph G_k of G is called a k-core if it satisfies: (1) for $\forall u \in V(G_k)$, $u.deg \geq k$, and (2) G_k is maximal. Moreover, $G_{k+1} \subseteq G_k$, for all $k \geq 0$, and G_0 is just G.*

Definition 2 (Core Number). *Given an undirected graph $G = (V, E)$, the core number of a vertex $u \in G(V)$, denoted $u.core$, is defined as $u.core = max\{k : u \in V(G_k)\}$. That means $u.core$ is the largest k such that there exists a k-core containing u.*

Definition 3 (k-Core Decomposition). *Given a graph $G = (V, E)$, the problem of computing the core number for each $u \in V(G)$ is called k-core decomposition.*

3.1 Distributed k-Core Decomposition Algorithm

Our federated core decomposition is based on the distributed core decomposition algorithm in [15].

Theorem 1 (Locality [15]). *For all $u \in V$, its core number, $u.core$, is the largest value k such that u has at least k neighbors that have core numbers not less than k. Formally, we define $u.core = k$, where $k \leq |\{v \in u.adj : v.core \geq k\}|$ and $(k + 1) > |\{v \in u.adj : v.core \geq (k + 1)\}|$.*

Theorem 1 shows that the vertex u is sufficient to calculate its core number from the neighbor's information. The algorithm is reorganized in Algorithm 1.

1. For the initialization stage (lines 1 to 4), each $u \in V$ has its *estimate core numbers* initialized as its degree (line 2). Each u will maintain a set of estimated core numbers for all neighbors $u.adj$ (line 3). Then, each u sends its estimated core numbers $u.core'$ to all neighbors (line 4), where $\texttt{Send}_u(v, \langle ... \rangle)$ procedure (executed on the vertex u) is to send the message $\langle ... \rangle$ to the target vertex v (lines 4 and 11).
2. Then u will receive the estimated core numbers $v.core'$ from its neighbors $v \in u.adj$ (lines 5 to 11). The received $core'$ will be stored in the array $u.A$ (line 6). If $u.A$ does not include all the estimated core numbers of $u.A$, it will return directly since u has not yet received all the neighbors' messages (line 7); otherwise, u will get the new core number k by executing `GetCore` (line 8). If k is not equal to $u.core'$, u will update its core number and send it to all neighbors (lines 9 - 11). The `GetCore` procedure will check the core number with Theorem 1; if it is not satisfied, it will decrease the core number k until Theorem 1 is satisfied (lines 12 - 15).
3. Each vertex $u \in V$ executes the above distributed algorithm in parallel. The termination condition is that all vertices $u \in V$ satisfy Theorem 1 and

thus stop decreasing the estimated core numbers $u.core'$ at the same time. Finally, we obtain the final calculated core numbers $u.core = u.core'$ for all $u \in V$.

Algorithm 1: Distributed Core Decomposition on Each Vertex $u \in V$ in Parallel

```
procedure Initialize_u()
    $u.core' \leftarrow u.deg$
    $u.A \leftarrow$ an empty array storing neighbors' core numbers
    for $v \in u.adj$ do Send_u($v, \langle u, u.core' \rangle$)

procedure Receive_u($\langle v, core' \rangle$)
    $u.A[v] \leftarrow core'$
    if $|u.A| < u.deg$ then return
    $k \leftarrow$GetCore($u.A, u.core'$)
    if $u.core' \neq k$ then
        $u.core' \leftarrow k$
        for $v \in u.adj$ do Send_u($v, \langle u, k \rangle$)

procedure GetCore($A, k$)
    while $|\{i \in A : i \geq k\}| < k$ do
        $k \leftarrow k - 1$;
    return $k$
```

Example 1. In Fig. 2, we show the k-core decomposition in an example graph. Fig. 2(a) shows that the estimated core number of each vertex is initialized as its degree. For example, the vertex $d.core$ is set to 6 since it has 6 neighbors and its degree is 6. Fig. 2(b) shows that in the first round the core numbers for the vertices b, d, g, f and j are active and decrease to 2, 3, 3, 3 and 1 (colored red with bold circles), respectively. For example, the vertex $d.core$ decreases from 6 to 3, as $d.adj$ has a set of core numbers $\{2, 3, 3, 3, 4\}$ and $d.core$ can be at most 3 since there are at most 3 neighbors that have core numbers equal to or greater than 3. Fig. 2(c) shows that in the second round the core numbers for the vertices g and i decrease to 2 and 1 (colored purple with bold circles), respectively. Fig. 2(d) shows that in the third round the core number of the vertex h decreases to 1 (colored green). Finally, the algorithm terminates, since all vertices are inactive and cannot continually decrease their core numbers. As we can see, the core numbers are calculated in three rounds, and the vertex g updates the core number twice.

Message Complexities. Since most of the running time is spent on the message passing for the distributed k-core decomposition algorithm, we should analyze *message complexities* rather than the time complexities, where the message complexities is to count the number of messages passing between different vertices. We analyze the message complexities of Algorithm 1 in the standard *work-depth* model [9]. The *work*, denoted as $\mathcal{W}$, is the total number of operations that the algorithm uses. The *depth*, denoted as $\mathcal{D}$, is the longest chain of sequential operations.

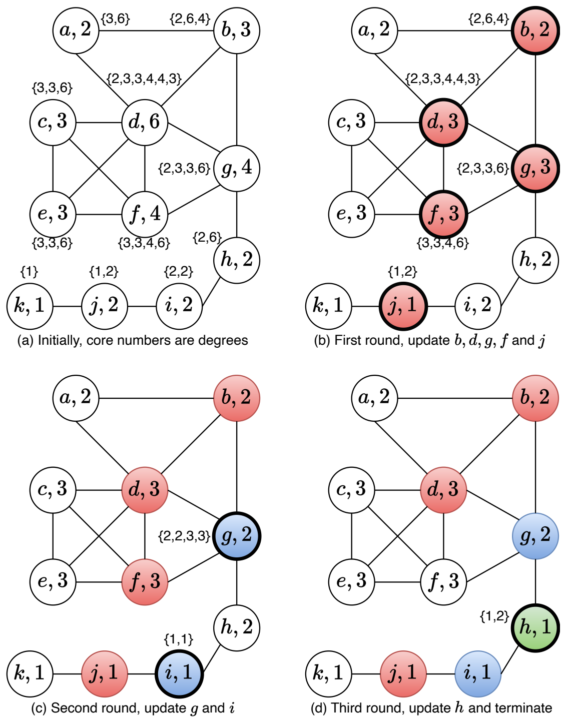


Fig. 2: An example graph executes distributed k-core decomposition. Inside the circles, the letters are the vertices' IDs and the numbers are the core numbers. The bold circles show that the vertex is active for calculating its core number. The colored circles show that the core numbers have been updated in different rounds. The sets of numbers beside some vertices are their neighbors' core numbers.

In Algorithm 1, the work $\mathcal{W}$ is the total number of messages that the degree reduces to the core number. Each vertex must send messages to notify all neighbors when each time its degree decreases by one until its degree is reduced to its core number. So, we have the work denoted as $\mathcal{W} = O[\sum_{u \in V} u.deg \cdot (u.deg - u.core)]$.

In the worst case, the process can be reduced to sequential running, e.g., a chain graph. In other words, the whole process needs the worst-case n round to converge one by one in a chain. We suppose that each vertex is a client and only has one worker, so it cannot send messages to all neighbors in parallel. Therefore, the depth $\mathcal{D}$ is equal to the work $\mathcal{W}$. However, real graphs, for example, social networks and communication networks, are not chain graphs and tend to be *small-world* graphs. They have the property that most vertices are reachable from any other node through a small number of steps, even if the network is large [16], which exhibits two key properties, high clustering and a short average path length. Therefore, these graphs have a set of vertices D in the longest chain, which is much smaller than the number of vertices n, denoted as $|D| \ll |n|$. For example, in small-world social networks we always have $|D|$ less than 100 and n

can be large as millions. Different chains can execute in parallel. Therefore, on average, the depth $\mathcal{D} = O[\sum_{u \in D} u.deg \cdot (u.deg - u.core)]$

4 Our Technical Details

In this section, we describe the details of the parts in the framework shown in Fig. 1.

4.1 Comparing Core Numbers by HE

We use asymmetric HE to compare the core numbers of two vertices.

Algorithm. Given an edge $(u, v) \in E(G)$, we have a *source* vertex u and a *target* vertex v. The vertex u wants to compare its core number with v. In other words, u wants to acquire the result of $u.core > v.core$. The whole process is to extend the core numbers comparing in secure (line 13 in Algorithm 1), and the array $u.A$ will store the result of core number comparison instead of the value of neighbors' core numbers (lines 3 and 6 in Algorithm 1). The secure comparison algorithm has four steps:

1. The target vertex v send the message m_1 to notify u that $v.core$ is decreased (or new assigned for initialization, line 4 in Algorithm 1), where m_1 includes the ID of v denoted as $m_1 = \langle v \rangle$.
2. The source vertex u generates a pair of keys (pri, pub); we define the public encrypt function $E_{pub}()$ and the private decrypt function $D_{pri}()$. Then, the source vertex u sends the message m_2 to v, where m_2 includes the public key and its encrypted core number denoted as $m_2 = \langle pub, E_{pub}(u.core) \rangle$.
3. The target vertex v receives the message m_2 from u. Then, v generates $E_{pub}(v.core)$. According to the definition of HE, we have $E_{pub}(u.core) > E_{pub}(v.core) = E_{pub}(u.core > v.core)$. Thus, v can reply the message m_3 to u, where m_3 includes the encrypted comparison result denoted as $m_3 = \langle E_{pub}(u.core > v.core) \rangle$.
4. The vertex u receives m_3 from v. Then u can decrypt the message $D_{pri}(m_3)$ with its private key pri and get the result of $u.core > v.core$.

Example 2. In Fig. 3, we show an example of how to compare the core numbers for the source vertex a and the target vertex b. The vertices a and b have the core numbers 2 and 3 initialized by their degree, respectively. First, b sends m_1 to a, indicating that $b.core$ is updated. Second, a generates a pair of public and private keys; a sends the public key pub and the encrypted core number 2 to b. Third, b send the encrypted result for the core number comparing $2 > 3$, to a. Finally, a decrypts the result and gets the result to be `false`, which is stored in the array A for later updating $a.core$.

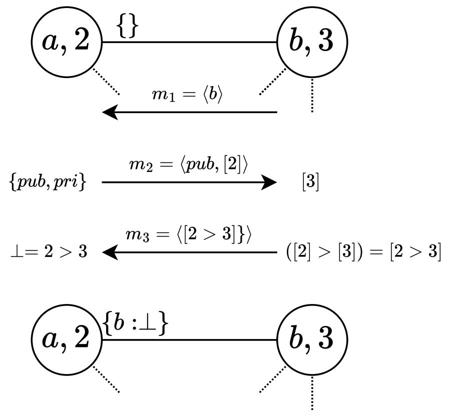


Fig. 3: An example of comparing core numbers between the vertices a and b in Fig. 2(a). The arrows are the direction for passing messages between two vertices. The [2] is the encrypted value 2 with the public key pub.

Privacy Security Analysis. In the end, only the source vertex u knows the result of $u.core > v.core$, but v does not know this result; also, u cannot know the value of $v.core$, and vice versa. However, there are two cases where the values of core numbers are leaked.

First, given $v.core$ is in the range $[0, k_{max}]$, u can compare $\lceil \log k_{max} \rceil$ times to acquire the value of $v.core$, by using a binary search. That is, at first, u can pretend that its core number is $k_{max}/2$, compare with $v.core$, and identify the half range in which $v.core$ is. This process continues until u acquires the value of $v.core$. To solve this problem, a straightforward solution is that the target vertex v can limit the number of comparisons for the source vertex u. In this case, we can ensure that u only compares the core number with v once, with minimal leaking the value of $v.core$ to u.

Second, a special case is that if we have $u.core = 2 \ \wedge \ u.core > v.core$ as true, we can infer that $v.core = 1$, since v must be connected to u and $v.core$ cannot be 0. In this case, the source vertex u can infer that $v.core = 1$, which is the minimal leaking of the value of $v.core$ to u.

Message Complexities. The whole process involves three messages, m_1, m_2, and m_3, so that the source vertex u can compare the core number with the target vertex v. Obviously, the length of m_1 is bounded by $O(\log n)$ since it only includes the ID of the vertex. The length of m_2 is bounded by $O(\log n)$, since the encrypted Boolean value always has a fixed length. The public and private keys have fixed length, e.g., the RSA typically ranges from 1024 bits to 4096 bits. So, m_2 includes the public key pub, where the length is bound by $O(\log n)$. Therefore, m_1, m_2, and m_3 need not be split, and three messages are required for our one-time secure comparison of the core numbers.

Compared with our method, in Algorithm 1, v only needs to send one single message that includes $v.core$ to u when $v.core$ is updated, where the core number must be bounded by $O(\log n)$. In other words, our secure algorithm spends triple messages to ensure the protection of the core numbers.

4.2 Decentralized Termination Detection

We describe how to detect the termination of the distributed core decomposition algorithm without using a centralized server, that is, a decentralized approach. Our approach is based on the Feedback BFS tree and heartbeats.

We first define that each vertex u needs to maintain a status, denoted $u.s$, which has two values:

- `Live`: the vertex u actively receive the messages from neighbors $u.adj$ and process these messages; or u actively send the messages to neighbors $u.adj$. Initially, the vertex u must be `Live` since it begins to send $u.core$ to all neighbors in $u.adj$.
- `Dead`: the vertex u stops calculating the core number since the Locality of u (Theorem 1) is satisfied; and u is not receiving or sending messages.

In Algorithm 1, before lines 2 and 6, we insert $u.s \leftarrow$ `Live`, respectively; after lines 4 and 11, we insert $u.s \leftarrow$ `Dead`, respectively. This is to explicitly show the status of the current vertex.

Theorem 2 (Termination Condition). *The distributed k-core decomposition is terminated if the status $u.s$ for all vertices $u \in V$ is* `Dead` *simultaneously within a period that must be greater than L_{max}.*

Proof. From Algorithm 1, we can see that the termination condition is that all vertices stop calculation, and also stop sending and receiving messages. It is possible that the vertex u has sent the message to v, but v has not yet received the message due to latency up to L_{max}. Since such a latency can be long, we must consider it when detecting the termination. In other words, the distributed algorithm is not terminated if there exists at least one `Live` vertex for a period greater than L_{max}.

Feedback BFS Tree. Given a graph G, we first select a root vertex r to build a Feedback BFS tree. The process is straightforward with four steps:

1. Starting from the root vertex r, it sends the message m_1 to all neighbors $r.adj$, where m_1 includes a version number ver in case there are multiple BFS trees generated on the same graph, denoted $m_1 = \langle ver \rangle$. Here, the root vertex has children without parent.
2. The vertex v receives the message m_1 from a neighbor u'. If v is the first time receiving m_1 (the parent $v.pr = u'$), it forwards m_1 to other neighbors $v.adj \setminus u'$ who have not received m_1 (the children $v.ch$). This process will be repeated until v is a leaf vertex that does not have children.
3. The leaf vertices w ($w.ch = \emptyset$) receive m_1 and will reply the message m_2 to its parent w', where m_2 includes an opposite version number $-ver$ in m_1 (indicate that m_2 is a corresponding replied message), denoted $m_2 = \langle -ver \rangle$. Then, w' will reply to its parent $w'.pr$ at once if all children $w'.ch$

have received m_2. This process will be repeated until the root vertex r receives the message m_2.

4. Finally, we obtain the Feedback BFS duration $\overline{T}$, which is the time period between the root vertex r sending m_1 and receiving m_2 (see Definition 4).

Definition 4 (Feedback BFS Duration $\overline{T}$). *Given a root vertex r with a BFS tree generated in the graph G, r sends messages to its children to generate the BFS tree at time t_1. Then r receives all the feedback messages from the children at time t_2. Here, a vertex v in the graph can send a feedback message to the parent only if v receives all feedback messages from the children recursively. The BFS duration is denoted as $\overline{T} = t_2 - t_1$ in a round way. That is, any vertex can receive the message from the root r within $\overline{T}/2$ in one way.*

We can see that this BFS tree generates a fastest path from the root to all the other vertices. Since message passing is slow, the first message that a vertex v can receive must come from the fastest path. In other words, the duration $\overline{T}$ means the round time between sending messages to the furthest vertex and receiving the feedback. Our termination detection is based on this BFS tree.

Theorem 3 (Reachable in $\overline{T}$). *Given a generated Feedback BSF tree on a distributed graph G, we can select any pair of vertices (u, v). The vertex u can broadcast the message m_1 starting u, and v can always receive m_1 within the duration $\overline{T}$.*

Proof. Suppose that the root vertex is r. The vertex u can broadcast the message m_1 to r within $\overline{T}/2$; and the r can continuously forward the message m_1 to v within $\overline{T}/2$. Therefore, the total time for u can broadcast m_1 to v cost at most $\overline{T}$ time.

Example 3. An example is shown in Fig. 4. A BFS tree is built by choosing the vertex a as the root. The vertex c, e, f, and k are leaves without children; the vertex d has a parent a and children c, e, f and g; the root a does not have a parent but have two children b and d.The BFS duration $\overline{T}$ is the round time of the message passing between a and the furthest leaf vertex k, where we have $20 \times 6 = 120$ ms for one way and 240 ms for the round way. We observe that there are totally 11 edges given 12 vertices in this BFS tree.

Heartbeat-Based Approach. After building the BFS tree for a given graph, each vertex sends the heartbeat message to notify other vertices that it is `Live`. Each vertex will send the heartbeat to other vertices in the BFS tree. We first define two attributes for the heartbeat:

– *Heartbeat Interval I.* Since our system can accept the latency of detecting termination, the heartbeat interval can be large enough to ensure low network resource consumption; but the it also should be much less than the total running time.
– *Timeout Duration T.* Typically, T is set to a multiple of the interval I (e.g., 2 - 3 times the interval). This gives some leeway for occasional delays in heartbeat delivery. For example, we can set I as 10 s giving T as 30 s.

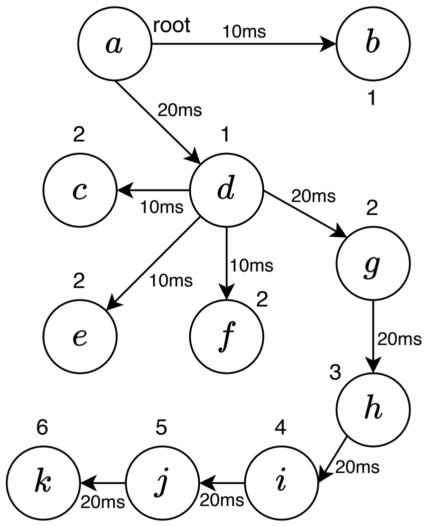


Fig. 4: An example of generate BFS tree. The root vertex is a. The numbers beside the circles are the depth of vertices from the root. The arrows of edges indicate the direction of broadcast the messages, which can only from the parent to the child. The time beside edges is the latency of the message passing via edges.

Theorem 4. *Given the Feedback BFS duration $\overline{T}$, we can choose the Timeout Duration $T = 3\overline{T}/2$, and the Heartbeat interval $I = T/3$.*

Proof. We choose the value of T for two reasons. First, we must consider the time that the message passes through networks, that is, a heartbeat can be received by other vertices using at most $\overline{T}$ time. Second, we must have some leeway for the timeout duration for measuring the time, e.g. $1/2$ of $\overline{T}$.

Base on the determined T and I, we define a second status $u.s'$ for each vertex u to detect termination, which includes two values:

- `Active`: the vertex u always receives at least one heartbeat within the Timeout Duration T; or u is computing, that is $u.s = $ `Live`.
- `Inactive`: the vertex u does not receive the heartbeat within the Timeout Duration T; and u is not computing, that is $u.s = $ `Dead`.

Only the `Live` vertices u that are performing computations with $u.s = $ `Live` can generate and send heartbeats. The detailed process is as follows:

1. Initially, all vertices $u \in V$ are set to `Live` which have the $u.s'$ set to `Active`.
2. For each vertex $u \in V$, if u is executing the computation with $u.s = $ `Live`, $u.s'$ is set to `Active`; then, u will continuously send heartbeats to neighbors in the BFS tree; otherwise, $u.s'$ is `Dead` and stop sending heartbeats.

3. For the vertex v, if v receive the heartbeats from u within T, $v.s'$ is set to `Active`; then, v will continuously forward heartbeats to other neighbors in the BFS tree except u; otherwise, $v.s'$ is set to `Inactive` and stop forwarding heartbeats.
4. The Steps 2 and 3 will repeat until the heartbeats broadcast to all vertices in the graph using the BFS tree.

By doing this, we only need to check the status of $u.s'$ for any vertices $u \in V$. If $u.s'$ is `Inactive`, we know that the distributed core decomposition is terminated; otherwise, the algorithms is still executing. In other words, any vertices u always have the status $u.s'$ indicating termination.

Example 4. As an example BFS tree shown in Fig. 4, we suppose that only the vertex h is `Live` and all other vertices are `Dead` as shown in Fig. 2(d).

At the beginning stage, all vertices are set to `Active`. Only the vertex h is doing the computation, so h is `Live`; it immediately generates heartbeats and sends them to all its BFS neighbors, g and i. These heartbeats will be repeatedly forwarded to all other vertices, which are set to `Active`. For example, g is set to `Active` when receiving heartbeats from h after 20 ms, then g immediately send heartbeats to d and d is set to `Active` when receiving them after 40 ms. In this case, the fast vertex b will receive the heartbeats and set to `Active`after 70 ms. Finally, all vertices can receive the heartbeat and set to `Active`. That is, the `Live` vertices can only generate and send heartbeats; the `Dead` vertices can only forward the heartbeats generated by the `Live` vertices.

We choose the Timeout Duration $T = 360$ms, larger than the BFS Duration $\overline{T} = 240$ms. So, we choose the Heartbeat Interval $I = 360/3 = 120$ ms. We can test the termination with any vertices, e.g. b; that is, if b cannot receive heartbeats within T, b is set to `Inactive`. We detect that the algorithm has been terminated.

Privacy Security Analysis. For the whole process, the root vertex is selected arbitrarily and any vertex can be the root, which is not centralized. Each vertex u must record the information of the neighbors $u.adj$ for the parent and children in the BFS tree; and u does not know any other connections of the neighbors. Finally, the root vertex can know the BFS duration $\overline{T}$, which is the information of the whole graph; but the latency for a specific edge cannot be inferred from $\overline{T}$.

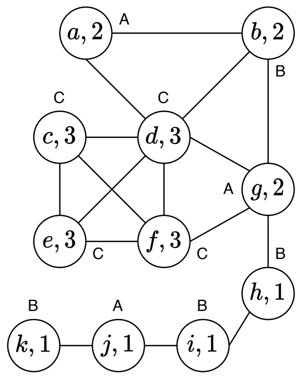
Message Complexities. We only analyze the Feedback BFS tree. Initially, the vertices send totally m messages to build the BFS tree. The BFS tree has $n-1$ edges, so the vertices respond with $n-1$ feedback messages. Therefore, in the worst case, the total number of messages is $\mathcal{W} = O(m+n)$, and the depth is the longest chain in the graph, denoted as $\mathcal{D} = O(|D|)$.

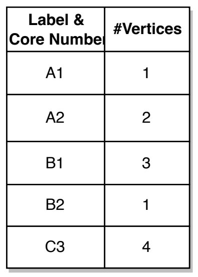
4.3 Calculate Core Number Distribution

After the termination of the distributed k-core decomposition, we need to release the core numbers as computation results to minimize privacy leakage.

Algorithm. Given a graph G, we assume that the BFS tree is already built with the root vertex r, which is used for decentralized termination detection. Then, for the vertex u, we select a label and a core number, denoted $u.(lb, k)$, to count the number of such vertices, denoted $n_{(lb,k)}$. The detailed process is as follows:

1. Starting from the root r, it generates a pair of keys (pri, pub) with the public encrypt function E_{pub} and the private decrypt function D_{pri}. It sends the message m_1 to all children $u \in r.ch$, where m_1 includes the public key, a pair of the label and the core number, and a version number ver in case of multiple instances of the BFS tree, denoted $m_1 = \langle pub, E_{pub}(lb, k), ver\rangle$.
2. The vertex u receives the message m_1. If u has children $u.ch$, u will forward m_1 to all $u.ch$. This process will repeat until all vertices $u \in V$ receive m_1.
3. The leaf vertices u ($u.ch = \emptyset$) receive m_1 and will reply the message m_2 to its parent $u.pr$. If the vertex u has the same label and core number as in m_1, denoted $E_{pub}(u.(lb, k)) = E_{pub}(lb, k) \in m_1$, we reply m_2 to $u.pr$, where m_2 includes the encrypted counting number 1 and opposite version number $-ver$ (indicate m_2 is a corresponding replied message), denoted $m_2 = \langle E_{pub}(1), -ver\rangle$; otherwise, the replied message $m_2 = \langle E_{pub}(0), -ver\rangle$. In other words, we the leaf vertices start to count.
4. The non-leaf vertices u ($u.ch \neq \emptyset$) receive the message m_2. If all children $v \in u.ch$ receive m_2 together, u will add all $E_{pub}(n_{(lb,k)})$ of children together and reply to $u.pr$ with the message $m_2 = \langle E_{pub}(n_{(lb,k)}), -ver\rangle$. Here, HE is used for the encrypted add operation like $E_{pub}(1) + E_{pub}(2) = E_{pub}(3)$.
5. This step (3) and (4) will repeat until all children of the root vertex $u \in r.ch$ receive the message m_2. Finally, the root r will obtain the complete number of vertices with the corresponding (lb, k). With its local private key, the root r can decrypt the counting, denoted $n_{(lb,k)} = D_{pri}(E_{pub}(n_{(lb,k)}))$. Now, we obtain the result of $n_{(lb,k)}$.





Label & Core Number	#Vertices
A1	1
A2	2
B1	3
B2	1
C3	4

Fig. 5: An example of calculate core number distribution. The capital letters beside the circles are the label of vertices. The right table shows the result of core number distribution, where the first column shows the labels and core numbers and the the second column shows the number of vertices.

Example 5. In Fig. 5, we show an example of calculating the distribution of the core numbers. After core decomposition, we observe that the vertices have core numbers and different labels, A, B and C. We want to release the number of vertices with the label and B and the core number 1, which is shown in the right table $B1$ with 3.

Using the BFS tree shown in Fig. 4, the root a sends the message m_1 to notify all vertices. The leaf vertices, b, c, f and k, will first reply the message m_2 to their parent vertices, a, d and j. This process repeats until the root a receives all the messages m_2 from its children, b and d, where b has counter 0 and d has counter as 3. Therefore, we obtain the final counter 3 for $B1$ as the result.

During this process, we can see that $n_{(B,1)}$ in m_2 are encrypted by HE. Thus, only the root vertex a can acquire the corresponding values and all other vertices only can perform the calculation without knowing the values.

Privacy Security Analysis. For the entire process, the root vertex is selected arbitrarily and any vertex can be the root, which is not centralized. Besides the root r, each vertex u performs the add operation on the number of vertices with $E_{pub}(n_{(lb,k)})$ for his children, but u does not know the specific value due to encryption. In addition, u does not know if it is counted or not in $n_{(lb,k)}$, since $u.(lb, k)$ is encrypted for comparison. Therefore, u only knows whether the calculation of the core number distribution is executed or not. But u does not know which root vertex r launches this calculation and which pair of labels and core numbers (lb, k) is counted. Only the root r has the private key for decryption and can get the final result of $n_{(lb,k)}$, so that all the other vertices cannot know such a result.

Message Complexities. When releasing one result for one pair given a label and a core number, since the BFS tree has $n - 1$ edges, the total number of messages is $\mathcal{W} = O(2(n - 1)) = O(n)$. The depth is the longest chain in the graph, denoted as $\mathcal{D} = O(D)$.

For releasing the results for all pairs given labels and core number, in the worst case, we can have a maximum number for the combination of labels and core numbers up to $|\mathcal{L}| \cdot k_{max}$. So, the algorithms must run at most $|\mathcal{L}| \cdot k_{max}$ to obtain all of the core number distributions. Typically, we can choose different root vertices to start the calculation of the core numbers at the same time. It can be executed in parallel with high probability, as most of the running time is spent on the message passing, and the calculations on clients are much faster than the latency of messages.

5 Conclusion and Future Work

In this work, we summarize the classical distributed k-core decomposition algorithm. Then, we adequately solve the privacy and security problems in terms of comparing core numbers, decentralized termination, and release core number distributions. In the future, we can apply our methodology to other distributed algorithms, e.g. distributed k-core maintenance [7, 8], to protect privacy and security.

References

1. Acar, A., Aksu, H., Uluagac, A.S., Conti, M.: A survey on homomorphic encryption schemes: Theory and implementation. ACM Computing Surveys (Csur) **51**(4) (2018) 1–35
2. Aridhi, S., Brugnara, M., Montresor, A., Velegrakis, Y.: Distributed k-core decomposition and maintenance in large dynamic graphs. In: Proceedings of the 10th ACM international conference on distributed and event-based systems. (2016) 161–168
3. Batagelj, V., Zaversnik, M.: An o(m) algorithm for cores decomposition of networks. CoRR **cs.DS/0310049** (2003)
4. Burleson-Lesser, K., Morone, F., Tomassone, M.S., Makse, H.A.: K-core robustness in ecological and financial networks. Scientific reports **10**(1) (2020) 1–14
5. Chan, T.H.H., Sozio, M., Sun, B.: Distributed approximate k-core decomposition and min–max edge orientation: Breaking the diameter barrier. Journal of Parallel and Distributed Computing **147** (2021) 87–99
6. Chen, C., Cui, J., Liu, G., Wu, J., Wang, L.: Survey and open problems in privacy preserving knowledge graph: Merging, query, representation, completion and applications. arXiv preprint arXiv:2011.10180 (2020)
7. Guo, B., Sekerinski, E.: Simplified algorithms for order-based core maintenance. arXiv preprint arXiv:2201.07103 (2022)
8. Guo, B., Sekerinski, E.: Parallel order-based core maintenance in dynamic graphs. In: Proceedings of the 52nd International Conference on Parallel Processing. (2023) 122–131
9. JéJé, J.: An introduction to parallel algorithms. Reading, MA: Addison-Wesley (1992)
10. Kong, Y.X., Shi, G.Y., Wu, R.J., Zhang, Y.C.: k-core: Theories and applications. Technical report (2019)
11. Liao, X., Liu, Q., Jiang, J., Huang, X., Xu, J., Choi, B.: Distributed d-core decomposition over large directed graphs. arXiv preprint arXiv:2202.05990 (2022)
12. Luo, G., Fang, Z., Zhao, X., Chen, M.: A survey of graph federation learning for data privacy security scenarios. (2023)
13. Luo, Q., Yu, D., Li, F., Cheng, X., Cai, Z., Yu, J.: Distributed core decomposition in probabilistic graphs. Asia-Pacific Journal of Operational Research **38**(05) (2021) 2140008
14. Malliaros, F.D., Giatsidis, C., Papadopoulos, A.N., Michalis Vazirgiannis, ·.: The core decomposition of networks: theory, algorithms and applications. The VLDB Journal **29** (2020) 61–92
15. Montresor, A., De Pellegrini, F., Miorandi, D.: Distributed k-core decomposition. IEEE Transactions on Parallel and Distributed Systems **24**(2) (2013) 288–300
16. Newman, M.E.: The structure and function of complex networks. SIAM review **45**(2) (2003) 167–256
17. Terzi, D.S., Terzi, R., Sagiroglu, S.: A survey on security and privacy issues in big data. In: 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST), IEEE (2015) 202–207