The VLDB Journal
DOI 10.1007/s00778-016-0423-8

REGULAR PAPER

Incremental *k*-core decomposition: algorithms and evaluation

Ahmet Erdem Sarıyüce[1] · Buğra Gedik[2] · Gabriela Jacques-Silva[3] · Kun-Lung Wu[3] · Ümit V. Çatalyürek[4]

Received: 13 October 2014 / Revised: 4 January 2016 / Accepted: 16 January 2016

Abstract A *k*-core of a graph is a maximal connected subgraph in which every vertex is connected to at least *k* vertices in the subgraph. *k*-core decomposition is often used in large-scale network analysis, such as community detection, protein function prediction, visualization, and solving NP-hard problems on real networks efficiently, like maximal clique finding. In many real-world applications, networks change over time. As a result, it is essential to develop efficient incremental algorithms for dynamic graph data. In this paper, we propose a suite of incremental *k*-core decomposition algorithms for dynamic graph data. These algorithms locate a small subgraph that is guaranteed to contain the list of vertices whose maximum *k*-core values have changed and efficiently process this subgraph to update the *k*-core decomposition. We present incremental algorithms for both insertion and deletion operations, and propose auxiliary vertex state maintenance techniques that can further accelerate these operations. Our results show a significant reduction in runtime compared to non-incremental alternatives. We illustrate the efficiency of our algorithms on different types of real and synthetic graphs, at varying scales. For a graph of 16 million vertices, we observe relative throughputs reaching a million times, relative to the non-incremental algorithms.

Keywords *k*-Core · Streaming graph algorithms · Dense subgraph discovery · Incremental graph algorithms

✉ Ahmet Erdem Sarıyüce
asariyu@sandia.gov

Buğra Gedik
bgedik@cs.bilkent.edu.tr

Gabriela Jacques-Silva
g.jacques@us.ibm.com

Kun-Lung Wu
klwu@us.ibm.com

Ümit V. Çatalyürek
umit@bmi.osu.edu

1 Sandia National Labs, Livermore, CA, USA

2 Bilkent University, Ankara, Turkey

3 IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

4 The Ohio State University, Columbus, OH, USA

1 Introduction

Relationships between people and systems can be captured as graphs where vertices represent entities and edges represent connections among them. In many applications, it is highly beneficial to capture this graph structure and analyze it. For instance, the graph may represent a social network, where finding communities in the graph [16] can facilitate targeted advertising. As another example, the graph may represent the web link structure, and finding densely connected regions in the graph [13] may help identify link spam [26]. In telecommunications, graphs are used to capture caller–callee relationships based on call detail records (CDRs) [24]. Such graphs can be used to locate closely connected groups of people for generating promotions. Graph structures are widely used in biological systems as well, such as in the study of proteins. Locating cliques in protein structures can be used for comparative modeling and prediction [27].

Many real-world graphs are highly dynamic. In social networks, users join/leave and connections are created/severed on a regular basis. In the web graph, new links are established and severed as a natural result of content update and creation. In customer call graphs, new edges are added as people extend their list of contacts. Furthermore, many applications require analyzing such graphs over a time window, as newly forming relationships may be more important than the existing ones. For instance, in customer call graphs, the historic calls are not too relevant for churn detection. Looking at a time window naturally brings removals as key operations, just like insertions. This is because as edges slide out of the time window, they have to be removed from the graph of interest. In summary, dynamic graphs where edges are added and removed continuously are common in practice and represent an important use case.

In this paper, we study the problem of incrementally maintaining the k-core decomposition of a graph. A k-core of a graph [29] is a maximal connected subgraph in which every vertex is connected to at least k other vertices. Finding k-cores in a graph is a fundamental operation for many graph algorithms. k-core is commonly used as part of community detection algorithms [18], as well as for finding dense components in graphs [3,5,21], as a filtering step for finding large cliques (as a k-clique is also a k-1-core), and for large-scale network visualization [2].

The k-core decomposition of a graph maintains, for each vertex, the max-k value: The maximum k value for which a k-core containing the vertex exists. This decomposition enables one to quickly find the k-core containing a given vertex for a given k. Algorithms for creating k-core decomposition of a graph in time linear to the number of edges in the graph exist [7]. For applications that manage dynamic graphs, applying such algorithms frequently is prohibitive in terms of performance and thus requires operating in large batches. However, the use of large batches takes away the ability to react to changes quickly—one of the key benefits of stream processing [31].

In this paper, we develop incremental algorithms for k-core decomposition of graphs. In particular, we develop algorithms to update the decomposition as edges are inserted into and removed from the graph (vertex additions and removals are trivial extensions). There are a number of challenges in achieving this. The first is a theoretical one: determining a small subset of vertices that are guaranteed to contain all vertices that may have their max-k values changed as a result of an insertion or removal. The second is a practical one: finding algorithms that can efficiently update the max-k values using this subset. Last but not the least, we have to understand the impact of the graph structure on the performance of such incremental algorithms.

We address these challenges by developing the first incremental k-core decomposition algorithm for dynamic graph data, where we efficiently process a small subgraph for each change. We focus on how to maintain the k-core decomposition when a single edge is inserted or removed. Single edge insertion and removal algorithms serve as a fundamental building block to handle dynamic graphs and can be used to efficiently process the updates coming in a batch manner. We develop a number of variations in our algorithm and empirically show that incremental processing provides a significant reduction in runtime compared to non-incremental alternatives, reaching 6 orders of magnitude faster solutions for a graph of size of around 16 million vertices. We showcase the efficiency of our algorithms on different types of real and synthetic graphs at varying scales and study the impact of graph structure on the performance of algorithm variations.

In summary, we make the following major contributions:

– We identify a small subset of vertices that have to be visited in order to update the max-k values (aka. the k-core decomposition) in the presence of edge insertions and deletions (Sect. 3).
– We develop a set of algorithms to update the k-core decomposition incrementally. To the best of our knowledge, these are the first such incremental algorithms (Sect. 4).
– We present a comparative experimental study that evaluates the performance of our algorithms on real-world and synthetic datasets (Sect. 6).

An earlier version of this paper has appeared in the VLDB 2013 conference [28]. This journal paper includes the following additional contributions that are not present in the conference paper:

– An incremental removal algorithm for incremental k-core decomposition, which is designed to be the dual of the *traversal* algorithm used for insertions (Sect. 4.3.3).
– A generalization of the *traversal* algorithm, which uses *multihop* residential core degrees (*RCD*s) (Sect. 4.5).
– A generalized *RCD* maintenance algorithm that can maintain the *RCD* values up to any given hop count, under insertions and removals ("Appendix").
– Additional evaluation of the incremental k-core decomposition algorithms, exploring the impact of graph structure on the optimal hop value for *RCD* maintenance (Sect. 6.5).

The rest of this paper is organized as follows. Section 2 gives the background on k-core decomposition of graphs. Section 3 introduces our theoretical findings that facilitate incremental k-core decomposition. Section 4 introduces several new algorithms for incremental maintenance of a graph's k-core decomposition. Section 5 provides discussions on implementation details. Section 6 gives a detailed experimental evaluation of our algorithms. Section 7 reports related work, and Sect. 8 concludes the paper. "Appendix" includes some of the pseudocodes and their detailed explanation.

2 Background

In this work, we focus on incremental maintenance of k-core decomposition of large networks modeled as undirected and unweighted graphs. Here, we start by giving several definitions that are used throughout the paper as part of our theorems and proofs.

Let G be an undirected and unweighted graph. For a vertex-induced subgraph $H \subseteq G$, $\delta(H)$ denotes the minimum degree of H, defined as the minimum number of neighbors a vertex in H has. That is to say, $\delta(H) = \min\{\delta_H(u): u \in H\}$, where $\delta_H(u)$ denotes the number of neighbors of a vertex u in H. As a result, any vertex in H is adjacent to at least $\delta(H)$ other vertices in H, and there is no other value larger than $\delta(H)$ that satisfies this property.

Definition 1 If H is a connected graph with $\delta(H) \geq k$, we say that H is a *seed* k-core of G. Additionally, if H is *maximal*, i.e., $\nexists H'$ s.t. $H \subset H' \wedge H'$ is a *seed* k-*core* of G, then we say that H is a k-core of G.

Observation 1 *Let H be a k-core that contains the vertex u. Then, H is unique in the sense that there can be no other k-core that contains u.*

k-cores cannot overlap partially, i.e., intersection of two k-cores is either empty or the k-core with smaller size. In other words, k-cores form a laminar family, which is a set system where all pairwise intersections are trivial (either empty or contains one of the sets). This is due to the maximality principle, pointed in Definition 1. We denote the unique k-core that contains u as H^u_k.

Definition 2 The *max-k*-core associated with a vertex u, denoted by H^u, is the k-core that contains u and has the largest $k = \delta(H^u)$, i.e., $\nexists H$ s.t. $u \in H \wedge H$ is an l-core $\wedge\, l > k$. The max-k-core number of u (also called the K *value of* u), denoted by $K(u)$, is defined as $K(u) = \delta(H^u)$.

Observation 2 *If H is a k-core in graph G, then there exists one and only one $(k-1)$-core $H' \supseteq H$ in G, since k-cores form a laminar family.*

Observation 3 *A vertex u with $K(u) = k$ takes part in cores $H^u_k \subseteq H^u_{k-1} \subseteq H^u_{k-2}, \ldots, \subseteq H^u_1$ by Observation 2.*

Building the core decomposition of a graph G is basically the same problem as finding the set of *max-k*-cores of all vertices in G. The following corollary shows that given the K values of all vertices, k-core of any vertex can be found for any k.

Corollary 1 *Given $K(v)$ for all vertices $v \in G$ and assuming $K(u) \geq k$, the unique k-core of a vertex u, denoted by H^u_k, consists of u as well as any vertex w that has $K(w) \geq k$ and is reachable from u via a path P such that $\forall_{v\in P},\ K(v) \geq k$. The unique k-core H^u_k can be found by traversing G starting at u and including each traversed vertex w to H^u_k if $K(w) \geq k$.*

```
Algorithm 1: findKCoreDecomposition($G(V, E)$)
Data: $G$: the graph
Compute $\delta_G(v)$ (i.e., the degree) for all vertices $v \in V$
Order the set of vertices $v \in V$ in non-decreasing order of $\delta_G(v)$
for each $v \in V$ do
    $K(v) \leftarrow \delta_G(v)$
    for each $(v, w) \in E$ do
        if $\delta_G(w) > \delta_G(v)$ then
            $\delta_G(w) \leftarrow \delta_G(w) - 1$
    Reorder the rest of $V$ accordingly
return $K$
```

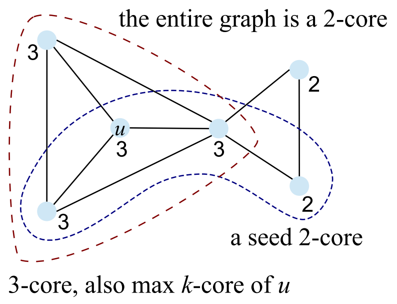

Fig. 1 Illustration of k-core concepts. The numbers adjacent to vertices are K values

Intuitively, in Corollary 1, all the traversed vertices are in H^u_k due to maximality property of k-cores, and all the vertices in H^u_k are traversed due to the connectivity property of k-cores, both based on Definition 1. Thus, the problem of maintaining the k-core decomposition of a graph is equivalent to the problem of maintaining its K values, by Corollary 1. The algorithm for constructing the k-core decomposition of a graph from scratch is based on the following property [29]: To find the k-cores of a graph, all vertices of degree less than k and their adjacent edges are recursively deleted. We provide its pseudocode in Algorithm 1 for completeness.

Corollary 1 is useful to model the incremental k-core decomposition problem as the maintenance of K values, which is a simple integer for each vertex. We build our theoretical findings and algorithms based on this, and deal with vertices instead of subgraphs, which is more convenient and also efficient to work on.

Figure 1 illustrates concepts related to k-core decomposition. In the sample graph, we see the K values of the vertices printed next to them, which is simply the k-core decomposition of the graph. We see a vertex labeled u. A seed 2-core that contains u is also shown. Moreover, the entire graph is the 2-core of u, i.e., $G = H^u_2$. The figure further shows a 3-core of u, that is H^u_3, which happens to be its max-k-core, that is $H^u_3 = H^u$. Note that $H^u_3 \subseteq H^u_2$.

3 Theoretical findings

In this section, we introduce our theoretical findings. These results facilitate incremental maintenance of the k-core decomposition of a graph. Since our incremental algorithms rely on finding a subgraph and processing it, we prove a number of theorems that can be used to find a small subgraph that is guaranteed to contain all the vertices whose K values change after an update.

Corollary 2 *Let $G = (V, E)$ be a graph and $u, v \in V$. If there is an edge $e \in E$ between u and v and if $K(u) > K(v)$, then $e \notin H^u$ and $e \in H^v$, by Corollary 1.*

In Corollary 1, it is stated for a vertex u, H^u is found by traversing the vertices with greater or equal K values. Thus, for two connected vertices u and v, if $K(u) > K(v)$, then H^u does not include v (and the edge between u and v).

Theorem 1 *If an edge is inserted to or removed from graph $G = (V, E)$, then the K value of vertex $u \in V$ can change by at most* 1.

Proof We first prove the insertion case. Assume that after the insertion of edge e, $K(u) = m$ is increased by n to $K_+(u) = m + n$, where $n > 1$. Let us denote the max-k-core of u after the insertion as H^u_+, and before insertion as H^u. It must be true that $e \in H^u_+$, as otherwise H^u_+ forms a seed $m + n$-core before the insertion as well, which is a contradiction. Let $Z = H^u_+ \backslash e$. If Z is not disconnected, then it must form an $m+n-1$-core, since the degree of its vertices can decrease by at most 1 due to the removal of a single edge. This leads to a contradiction since $m + n - 1 > m$ and H^u is maximal. In the disconnected case, each one of the resulting two connected components must be a seed $m + n - 1$-core as well, since the degree of a vertex can reduce by at most one in each component. Furthermore, since e is the only edge between the two disconnected components, the vertices must still have at least $m + n - 1$ neighbors in their respective components. One of these components must contain u, which is again a contradiction.

Next, we prove the removal case. Assume $K(u)$ is decreased by n after edge e is removed, where $n > 1$. Adding e back to the graph increases the K value of u by n, which is not possible, as shown in the first part of the proof, i.e., a contradiction.

Theorem 2 *If an edge (u, v) is inserted to or removed from $G = (V, E)$, where $u, v \in V$ and $K(u) < K(v)$, then $K(v)$ cannot change.*

Proof We first prove the insertion case. Assume that $K(v) = n$ increases and so becomes $K_+(v) = n + 1$ by Theorem 1. Then, we have $e \in H^v_+$ and consequently $u \in H^v_+$. However, $K(u) < n$ before insertion and $K_+(u)$ can be at most n after insertion (Theorem 1), implying that u cannot be in a seed $n + 1$-core, i.e., a contradiction.

For the removal case, assume that $K(v) = n$ decreases and becomes $K\_(v) = n - 1$ by Theorem 1. Inserting (u, v) back to the graph should increase the K value of v to $K(v) = n$. We must also have $e \in H^v$ and thus $u \in H^v$. But this is a contradiction due to Corollary 2, since $K(u) < K(v)$ and $u \notin H^v$.

From Theorem 2, we can say that when an edge (u, v) is inserted into or removed from the graph, $K(u)$ can change by at most 1 if $K(u) \leq K(v)$, or stay the same otherwise.

Theorem 3 *If an edge (u, v) is inserted into $G = (V, E)$, where $u, v \in V$, then all of the vertices whose K values have changed should form a connected subgraph $G^{'} \subset G \cup (u, v)$. Similarly, if an edge (u, v) is removed from $G = (V, E)$, where $u, v \in V$, then all the vertices whose K values have changed should form a connected subgraph $G^{''} \subset G$.*

Proof We prove the insertion case first. Assume that the updated vertices do not form a connected subgraph. Then, there are at least 2 non-overlapping subgraphs of updated vertices, S_1 and S_2. Since there is only one edge insertion, only one of these subgraphs, say S_1, can have a vertex who gets a new neighbor in G. Then, S_2 does not have any vertex that has its degree changed. This is a contradiction, because if a vertex has its K value increased, then it must have either gained a new neighbor (increased degree) or at least one of its existing neighbors must have its K value increased (Note that while it is *necessary* to satisfy one of these conditions, neither of them are *sufficient*. In other words, a vertex that has gained a new neighbor may not increase its K value, and a vertex with a neighbor whose K value has increased may not increase its K value). Applying this recursively, we must reach a vertex whose K value is increased due to gaining a new neighbor. However, for S_2, there is no such vertex since only reachable vertices whose K values have increased are in S_2, and none of them have their degrees changed.

For the removal case, assume that the updated vertices do not form a connected subgraph. Then, there are at least 2 non-overlapping subgraphs of updated vertices, S_1 and S_2. Since there is only one edge removal, only one of these subgraphs, say S_1, can have a vertex who loses a neighbor in G. Then, S_2 does not have any vertex that has its degree changed. This is a contradiction, because if a vertex has its K value decreased, then it must have either lost a neighbor (decreased degree) or at least one of its existing neighbors must have its K value decreased. Applying this recursively, we must reach a vertex whose K value is decreased due to losing an existing neighbor. However, for S_2, there is no such vertex since only vertices that can be reached and whose K value has decreased are in S_2, and none of them have their degrees changed.

Theorem 4 *Given a graph $G = (V, E)$, if an edge (u, v) is inserted (removed) and $K(u) \leq K(v)$, then only the vertices*

$w \in V$ that have $K(w) = K(u)$ and are reachable from u via a path that consists of vertices with K values equal to $K(u)$, ***may*** *have their K values incremented (decremented).*

Proof Before looking at the insertion and removal, we note that if the K value of any vertex in G increases (decreases) due to the insertion (removal) of (u, v), then $K(u)$ must have increased (decreased) as well. This follows from the recursive argument in Theorem 3, as otherwise none of the vertices that have their K values changed will have their degree changed.

For the insertion case, we first prove that for a vertex $w \in V$ such that $K(w) \neq K(u)$, $K(w) = m$ cannot change. We consider two cases: (i) where $K(w) > K(u)$ and (ii) where $K(w) < K(u)$.

For the $K(w) > K(u)$ case, assume $K(w)$ increases ($K_+(w) = m+1$). We must have $(u, v) \in H_+^w$, as otherwise H^w would not be a max-m-core before insertion. However, this is not possible since $K_+(w) > K_+(u)$, i.e., a contradiction due to Corollary 1.

For the $K(w) < K(u)$ case, assume $K(w)$ increases ($K_+(w) = m + 1$). Then, we have $(u, v) \in H_+^w$, as otherwise H^w would not be a max-m-core before insertion. We know that $m + 1 \leq K(u) \leq K(v)$, which implies $K_+(w) < K_+(u) \leq K_+(v)$. Removing (u, v) from H_+^w decreases the degrees of u and v by one, which can reduce their K value to at least $m + 1$. This means $H_+^w \backslash (u, v)$ is a seed $m+1$-core before the insertion, which is a contradiction.

We proved that only vertices with $K(w) = K(u)$, say $L \subseteq V$, *may* have their K values incremented. Furthermore, we know that all those vertices form a connected subgraph (Theorem 3). Since we have $u \in L$ as well, the insertion proof is complete.

We use similar arguments for the removal case. Again, we consider two cases.

For the $K(w) < K(u)$ case, assume $K(w)$ decreases ($K\_(w) = m - 1$). Say that we insert (u, v) back into the graph. The K value of w cannot increase in this case since $K\_(w) < K\_(u)$, and this is a contradiction, as shown in insertion part above.

For the $K(w) > K(u)$ case, assume $K(w)$ decreases ($K\_(w) = m-1$). We know that $(u, v) \notin H^w$ since $u \notin H^w$ due to $K(u) < K(w)$. Thus, H^w is still an m-core after the removal, creating a contradiction.

We proved that only the vertices that have $K(w) = K(u)$, say $L \subseteq V$, *may* have their K values decremented. Furthermore, by Theorem 3, we know that all those vertices form a connected subgraph. Since we have $u \in L$, the removal proof is complete.

Summary In this section, we showed that if an edge (u, v) is inserted into/removed from a graph, then the K value of u can change only if $K(u) \leq K(v)$. Let us call u the root. In case $K(u) = K(v)$, then either u or v is taken as the root. In addition, we showed that any vertex that may have its K value updated must have a K value that is equal to that of the root, and must be connected to the root via a path that contains only the vertices that have the same K value. We rely on these results in the next section.

4 Incremental algorithms

In this section, we introduce four algorithms to incrementally maintain the K values of vertices when a single edge is inserted or removed. The *subcore* (Sect. 4.1) and *purecore* (Sect. 4.2) algorithms are basic applications of the theoretical results given in the previous section, are easy to implement, and form a baseline for evaluating the performance of the *traversal* algorithm (Sect. 4.3). The traversal algorithm relies on additional ideas that aggressively cut the search space, but is more involved than the earlier two. For the edge insertion case, we also introduce the *generic multihop traversal* algorithm (Sect. 4.5), which generalizes the traversal algorithm to utilize multihop information.

4.1 The subcore algorithm

Our first algorithm for maintaining the K values of vertices when a single edge is inserted or removed is based on Theorem 4. We define a subgraph, called *subcore*, as follows:

Definition 3 Given a graph $G = (V, E)$ and a vertex $u \in V$, the *subcore* of u, also denoted as S_u, is a set of vertices $w \in V$ that have $K(w) = K(u)$ and are reachable from u via a path that consists of vertices with their K values equal to $K(u)$.

Given a graph $G = (V, E)$ and the K values of all $w \in V$, if an edge (u_1, u_2) is inserted to E, Algorithm 3 updates the K values. Similarly, if an edge (u_1, u_2) is removed from E, Algorithm 4 updates the K values. Both algorithms make use of Definition 3.

The basic idea is to locate the subcore of the root vertex and apply a process very similar to Algorithm 1 on the subcore. Algorithm 2 provides the pseudocode for finding the subcore. To find the subcore, we perform a BFS traversal and collect all vertices reachable from the root through vertices having the same K value as the root. During this process, we also collect the *current degree* (`cd`) values for each vertex in the subcore. In general, we use the *current degree* of a vertex throughout the paper to denote its degree in the new k-core after the edge insertion or removal operation. Depending on the context, it might be initialized with a different auxiliary information associated with a vertex. In Algorithm 2, *current degree* of a vertex is used to accumulate the degree of the vertex in its max-core and used to detect whether a vertex can change its K value or not. So, the `cd` of a vertex simply counts the number of its neighbors with a K value equal to or

```
Algorithm 2: findSubcore($G(V, E), K(), u$)
Data: $G$: the graph, $K$: max-$k$ values, $u$: the vertex
$H(V', E') \leftarrow$ empty graph; $Q \leftarrow$ empty queue
cd[$v$] = 0; visited[$v$] = false, $\forall v \in V$            ▷ Lazy init
$k \leftarrow K(u)$                          ▷ Remember $K$ value of the root
$Q$.push($u$); visited[$u$] $\leftarrow$ true
while not $Q$.empty() do
    $v \leftarrow Q$.pop(); $V'$.push($v$)
    for each $(v, w) \in E$ do
1       if $K(w) \geq k$ then
            cd[$v$] $\leftarrow$ cd[$v$] + 1
            if $K(w) = k$ and not visited[$w$] then
                $Q$.push($w$); $E'$.push(($v$, $w$))
                visited[$w$] $\leftarrow$ true
return $H$ and cd
```

```
Algorithm 3: SUBCORE: insertEdge($G(V, E), K(), u_1, u_2$)
Data: $G$: the graph, $K$: max-$k$ values, $(u_1, u_2)$: inserted edge
$r \leftarrow u_1$                                      ▷ Set the root
if $K(u_2) < K(u_1)$ then $r \leftarrow u_2$
$G \leftarrow G \cup (u_1, u_2)$                    ▷ Add the edge into $G$
$H$, cd $\leftarrow$ findSubcore($G, K, r$)            ▷ Find subcore
▷ Now, update the $K$ values of the vertices in $H$
$k \leftarrow K(r)$                           ▷ Remember $K$ value of the root
Sort cd values in non-decreasing order (using bucket sort)
for each $v \in H$ in order do
    if cd[$v$] $\leq k$ then             ▷ Cannot be part of a $k$+1-core
        for each $(v, w) \in H$ do
            if cd[$w$] > cd[$v$] then
                cd[$w$] $\leftarrow$ cd[$w$] − 1
                Reorder cd values accordingly
    else            ▷ All remaining vertices become part of $k$+1-core
        for each $w \in H$ do
            $K(w) \leftarrow k + 1$
        break
```

greater than the K value of the root. Degree of a vertex in its max-core helps us to eliminate vertices that cannot be part of a $k+1$ core, where k is the K value of the root. In particular, if the degree of a vertex in its max-core is not larger than k, we can eliminate the vertex from consideration. Once it is eliminated, it results in decrementing the current degree values of its neighbors in the subcore and the process can be repeated. Similar to Algorithm 1, this has to be performed in non-decreasing order of the current degree values.

Algorithm 3 shows how the subcore and the cd values are used to update the K values upon an edge insertion. We order the cd values of the vertices in the subcore in non-decreasing order. At each step, we pick the unprocessed vertex with the smallest cd value from the subcore. If it has a cd value less than or equal to the root's K value, say k, then it cannot be part of a $k+1$-core. Thus, for each of its neighbors in the subcore that have a higher cd, we decrement the neighbor's cd by 1, since the vertex being processed cannot be part of a higher core. We reorder the remaining vertices based on their updated cd values. Otherwise, that is if the current vertex has a cd value larger than k, all remaining vertices must also have their cd values larger than k, which means we can form a seed $k+1$ core with them. We increment their K values, completing the insertion.

Time complexity Algorithm 3 has two parts: (1) finding the subcore and cd values, and (2) processing them in a loop to find the new K values. In the worst case, regarding (1), we can end up traversing the entire graph and report it as the subcore, where all vertices have same K values in the graph. It will take $O(|E|)$ time. Regarding (2), we do essentially the same thing with Algorithm 1, which has $O(|E|)$ time complexity. Therefore, the worst-case time complexity of Algorithm 3 is $O(|E|)$. It is important to note that the algorithm is heuristic in nature, and we expect the size of the subcore to be much smaller than $O(|E|)$ in practice, which we verify in our experimental evaluation.

Space complexity In the worst case, the largest data structure used in Algorithm 3 is the graph H to store the output of findSubcore procedure. As mentioned above, we can end up traversing the entire graph to report it as a subcore, and we would need $O(|E|)$ space to store the graph in this case. Thus, the worst-case space complexity is $O(|E|)$.

Algorithm 4 shows how the subcore and the cd values are used to update the K values in the case of a removal. Unlike Algorithm 3, here we need to perform two subcore searches when the K values of the vertices incident upon the removed edge are the same, since the removal separates them. Once we locate the subcore, the process is very similar to that of the insertion. We pick the unprocessed vertex with the smallest cd value from the subcore and if it has a cd value less than the K value of the root, say k, then it cannot be part of a k-core anymore. As a result, we decrement its K value, and for each of its neighbors in the subcore that have a higher cd, we decrement the neighbor's cd by one, since the vertex currently being processed cannot be part of a higher core. After this, we reorder the remaining vertices based on their cd values. Otherwise, if the current vertex has a cd value larger than or equal to k, then all remaining vertices must also have their cd values larger than or equal to k, which means that we can still form a seed k-core with them. Thus, we stop processing and complete the removal.

Time complexity Algorithm 4 is quite similar to Algorithm 3 and only slightly differs in the second part, which does not affect the worst-case time complexity of $O(|E|)$.

Space complexity Just like the Algorithm 3, the worst-case space complexity is $O(|E|)$, for which the entire graph is reported as a subcore at the end of findSubcore procedure.

Algorithm 4: SUBCORE: REMOVEEDGE($G(V, E), K(), u_1, u_2$)

```
Data: $G$: the graph, $K$: max-$k$ values, $(u_1, u_2)$: removed edge
$r \leftarrow u_1$                                        ▷ Set the root
if $K(u_2) < K(u_1)$ then $r \leftarrow u_2$
$G \leftarrow G \setminus (u_1, u_2)$                     ▷ Remove the edge from $G$
if $K(u_1) \neq K(u_2)$ then
    $H, \mathtt{cd} \leftarrow$ findSubcore($G, K, r$)    ▷ Find subcore
else
    $H_1, \mathtt{cd}_1 \leftarrow$ findSubcore($G, K, u_1$)   ▷ Find subcore of $u_1$
    $H_2, \mathtt{cd}_2 \leftarrow$ findSubcore($G, K, u_2$)   ▷ Find subcore of $u_2$
    $H \leftarrow H_1 \cup H_2$; $\mathtt{cd} \leftarrow \mathtt{cd}_1 \cup \mathtt{cd}_2$
▷ Now, update the $K$ values of the vertices in $H$
$k \leftarrow K(r)$                                       ▷ Remember $K$ value of the root
Sort cd values in non-decreasing order (using bucket sort)
for each $v \in H$ in order do
    if $\mathtt{cd}[v] < k$ then                          ▷ Cannot be part of a $k$-core anymore
        $K(v) \leftarrow k - 1$
        for each $(w, v) \in H$ do
            if $\mathtt{cd}[w] > \mathtt{cd}[v]$ then
                $\mathtt{cd}[w] \leftarrow \mathtt{cd}[w] - 1$
                Reorder cd values accordingly
    else break;                                           ▷ All remaining vertices still in a $k$-core
```

4.2 The purecore algorithm

In Sect. 4.1, the subcore algorithm relied only on the K values of the vertices to locate a small subgraph that contains all the vertices that can have their K values changed. In this section, we look at the *purecore* algorithm that takes advantage of additional information about each vertex, so that a smaller set of candidate vertices can be located, reducing the overall cost of the algorithm. For this purpose, we define the *maximum-core degree* of a vertex.

Definition 4 The *maximum-core degree of a vertex* u, denoted as $MCD(u)$, is defined as the number of u's neighbors, w, such that $K(u) \leq K(w)$.

If the MCD value of a vertex is not greater than its K value and no new adjacent edge is inserted, then it is impossible for this vertex to increment its K value. This is simply because the number of neighbor vertices it has in a higher core will not be sufficient. Therefore, we can use the MCD value to test whether a vertex can increment its K value or not, upon a new edge insertion.

Observation 4 *For a given graph $G = (V, E)$ and a vertex $u \in V$, $MCD(u) \geq K(u)$.*

The observation follows simply from the definition of k-core, since $MCD(u) < K(u)$ would mean u cannot participate in a k-core with $K(u) = k$, leading to a contradiction. Note that $MCD(u)$ is simply an upper bound on $K(u)$.

We reduce the *subcore*, described in Definition 3, to a *purecore* by putting an extra condition regarding MCD values. The basic idea is that, if a vertex in the subcore does not have a MCD value greater than the K value of the root, it means that the vertex does not have enough neighbors that can participate in a higher core.

Definition 5 Given a graph $G = (V, E)$ and a vertex $u \in V$, the *purecore* of u, denoted as P_u, is the set of vertices $w \in V$ that have $K(w) = K(u)$ and $MCD(w) > K(u)$, and are reachable from u via a path that consists of vertices with K values equal to $K(u)$ and MCD values greater than $K(u)$.

Based on Definition 5, we give the following theorem.

Theorem 5 *Given a graph $G = (V, E)$, if an edge (u, v) is inserted and $K(u) \leq K(v)$, then only the vertices $w \in P_u$* ***may*** *have their K values incremented.*

Proof When an edge (u, v) is inserted to the graph and $K(u) \leq K(v)$, then the K value of a vertex $w \in S_u$, where $w \neq u$, cannot increment if $MCD(w) = K(w)$. Assume $K(w) = MCD(w) = k$ and $K(w)$ increments, becoming $k + 1$. Then, we have the new $MCD(w) \geq k + 1$ by Observation 4. From the initial assumption, we know that $MCD(w) = k$, i.e., w have k neighbors whose K values are greater or equal to k. Even if all those neighbors increment their K values, the new $MCD(w)$ can be at most k, which violates Observation 4, a contradiction.

With purecore, the algorithm to update the K values of vertices, when edge (u, v) is inserted, is the same as Algorithm 3, except that we use findPurecore procedure in place of Algorithm 2 (findSubcore). findPurecore procedure is same as the findSubcore, except line 1 (of Algorithm 2). Instead of checking for $K(w) \geq k$, findPurecore checks whether $K(w) > k$ *or* ($K(w) = k$ and $MCD(w) > k$). This is the condition needed to find the purecore of the root, as defined in Definition 5.

Time complexity The only difference with Algorithm 3 is the findPurecore procedure used in place of findSubcore. The worst case for findPurecore happens when we traverse all edges. Note that, for every edge (u, v), we need to visit all the neighbors of the vertex v in order to compute $MCD(v)$. This results in $O(|E| * (|E|/|V|)) = O(|E^2|/|V|)$ complexity. However, if we can check the MCD value of a vertex in constant time (which is possible by residential core degrees, explained in Sect. 4.3.1), total time complexity will be reduced to $O(|E|)$.

Space complexity As with Algorithm 3, in the worst case, the findPurecore procedure can return the entire graph, and thus, the space complexity is $O(|E|)$.
When an edge (u, v) is removed from the graph and $K(u) \leq K(v)$, then the K value of any vertex $w \in S_u$ can potentially decrement. Note that $MCD(w)$ can decrease if either w loses a neighbor, which is the case for u, or K value of some neighbor of w decrements, which is the case for neighbors

of u when $K(u)$ decrements. As a result, for removal, we do not rely on the purecore algorithm.

4.3 The traversal algorithm

We now present the traversal algorithm that visits an even smaller subgraph to update the k-core decomposition. First, we introduce an optimization to speedup the computation of the *MCD* values and then an additional metric to further scope the search.

4.3.1 Residential core degrees

In Sect. 4.2, we find a smaller set of candidate vertices to be updated by using more information about each vertex. Using more information, such as the *MCD* values, requires more computation in findPurecore procedure (Sect. 4.2). Thus, for a vertex u, when the size of P_u is *large* and close to the size of S_u, findPurecore procedure turns out to be more expensive than Algorithm 2. To alleviate this problem, we have two types of auxiliary information constantly reside in memory. We call these *residential core degrees*. Concretely, we maintain the *MCD* values, introduced in Definition 4, and the *PCD* values of vertices defined as follows:

Definition 6 The *purecore degree of a vertex u*, denoted as *PCD*(u), is the number of u's neighbors, w, such that *either* $K(u) = K(w) \wedge MCD(w) > K(u)$ *or* $K(u) < K(w)$.

For a vertex v, its purecore degree *PCD*(v) is the number of neighbors w it has that either has a higher K value than v or has the same K value but in turn has enough neighbors to potentially increase its K value (in case an insertion was made and the K values are to be updated). The *PCD* value of a vertex represents its potential number of neighbors in a next max-core. It is a stronger indicator than its *MCD* value for showing eligibility to increase the K value and also useful, because if *PCD*$(v) \leq k$ where k is the K value of the root, then v cannot increment its K value.

Maintaining the *MCD* and *PCD* values of vertices after each insertion and removal should be done efficiently. In general, the *MCD* value of a vertex is based on the K values of its neighbors, as seen from Definition 4, and the *PCD* value of a vertex is based on the K and *MCD* values of its neighbors, as described in Definition 4. Observation 5 gives a rule of thumb for *MCD* and *PCD* maintenance.

Observation 5 *For a graph $G = (V, E)$, when the K value of a vertex $u \in V$ changes, the MCD values of vertices u, v can change, where $(u, v) \in E$. When the K or MCD value of a vertex $u \in V$ changes, the PCD values of vertices v can change, where $(u, v) \in E$. As a result, when the K value of a vertex $u \in V$ changes, the PCD values of vertices u, v, w can change, where $(u, v), (v, w) \in E$.*

The observation is the direct result of Definitions 4 and 6. *MCD* of a vertex u is a function of $K(u)$ and the K values of its neighbors, say w. If $K(u)$ or any of the K value of a neighbor w changes, then *MCD*(u) may change. It implies that a change in K value of a vertex may change its own *MCD* value as well as its neighbors' *MCD* values. A similar argument can be said for *PCD* values. By Definition 6, *PCD* of a vertex u is a function of $K(u)$, and K and *MCD* values of its neighbors, say w. Therefore, if $K(u)$ or, K or *MCD* value of a neighbor w changes, then *PCD*(u) may change. This implies that a change in the K value of a vertex may change its own *PCD* value as well as its neighbors' *PCD* values. Note that this change may also affect *MCD* value of a neighbor, which in turn may affect *PCD* value of a neighbor of a neighbor. In summary, the observation says that a K value update can result in changes in the *MCD* values within the 1-hop neighborhood of the vertex, whereas changes in the *PCD* values can happen within the 2-hop neighborhood.

Based on Observation 5, when an edge (u, v) is inserted into or removed from a graph $G = (V, E)$, we first recompute the *MCD* value of the root vertex u and the *PCD* values of its neighbors. Next, we apply the algorithm to update the K values of vertices. Last, we do the following two operations to adjust the *MCD* and *PCD* values:

- Recomputing the *MCD* values of vertices $w, x \in V$ for which $K(w)$ is updated and $(w, x) \in E$.
- Recomputing the *PCD* values of vertices $w, x, y \in V$ for which $K(w)$ is updated and $(w, x), (x, y) \in E$

Further shortcuts are possible, based on the K and *MCD* values of the updated vertices, to minimize the number of *MCD* and *PCD* re-computations. We defer the details to "Appendix."

4.3.2 Root-aware edge insertion

So far, in all our incremental algorithms, we first find a subgraph and its corresponding `cd` values by a BFS traversal (phase 1). In a second phase, we process that subgraph by reordering the vertices with respect to their `cd` values and remove the vertex with the minimum `cd` at each step. Traversing the subgraph and computing the `cd` values should be done prior to the second phase, since we need all the vertex degrees in the subgraph. Theorem 4 points an interesting fact, saying that if the K value of some vertex changes, then the K value of at least one extremity of the inserted/removed edge, named as the root vertex (say u), must change. For the insertion algorithm, this fact suggests a *root-aware* approach, in which all vertices know whether the root still has a chance to change its K value. Additional operations are avoided once the algorithm detects that root is not going to change its K values. If *PCD*$(u) \leq K(u)$, then u cannot increment its K

value. This condition implies that there is no chance for the root to increase its K value.

We realize this root-aware approach by applying a depth-first search (DFS) with an *eviction* mechanism, where the vertices $v \in V$ are evicted if $PCD(v) \leq K(v)$. By doing that, we combine phases 1 and 2.

The root-aware insertion procedure does not need the `cd` values of all the vertices in the subgraph. As a result, we create the `cd` values for each vertex *on-the-fly* during DFS, avoiding the first phase of our previous algorithms completely. We leverage the *residential core degrees*, introduced in Sect. 4.3.1, to speed up the creation of `cd` values. On-the-fly creation of `cd` values makes the insertion algorithm more efficient.

Algorithm 5: TRAVERSAL: INSERTEDGE($G(V, E), K(), MCD(), PCD(), u_1, u_2$)

```
Data: G: the graph, K: max-k values, MCD: max-core degrees, PCD:
      purecore degrees, (u_1,u_2): inserted edge
  $r \leftarrow u_1$                                    ▷ Set the root
  if $K(u_2) < K(u_1)$ then $r \leftarrow u_2$
  $G \leftarrow G \cup (u_1, u_2)$                      ▷ Add the edge into G
1 PREPARERCDs    ▷ Prepares MCDs, PCDs of vertices around inserted
  edge
  ▷ Perform a traversal over vertices that have root's K value, while
    evicting the ones that cannot be a part of a k+1-core
  $S \leftarrow$ empty stack                            ▷ To perform DFS
  visited[$v$] = false, $\forall v \in V$              ▷ To perform DFS (lazy init)
  evicted[$v$] = false, $\forall v \in V$   ▷ To remember evicted vert. (lazy init)
  cd[$v$] = 0, $\forall v \in V$          ▷ To find vertices to be evicted (lazy init)
  $k \leftarrow K(r)$                         ▷ Remember the K value of the root
2 cd[$r$] $\leftarrow PCD(r)$                           ▷ Set cd of root
  S.push($r$); visited[$r$] $\leftarrow$ true
  while not S.empty() do                               ▷ Do a DFS traversal
      $v \leftarrow$ S.pop()
3     if cd[$v$] $> k$ then             ▷ Vertex is currently part of a k+1-core
          for each $(v, w) \in E$ do
              ▷ Neighboring vertex currently part of a k+1-core
4             if $K(w) = k$ and $MCD(w) > k$ and
                                             not visited[$w$] then
                  S.push($w$); visited[$w$] $\leftarrow$ true
                  ▷ Use + as cd[$w$] may be < 0 due to evictions
5                 cd[$w$] $\leftarrow$ cd[$w$] + $PCD(w)$
6     else                                ▷ Vertex cannot be part of a k+1-core
          if not evicted[$v$] then             ▷ Recursively perform eviction
              PROPAGATEEVICTION($G, K$, cd, evicted, $k, v$)
  for each $v$ s.t. visited[$v$] do                    ▷ Find visited vertices
      if not evicted[$v$] then                          ▷ If not evicted as well
          $K(v) \leftarrow K(v) + 1$                  ▷ The vertex is part of a k+1-core
7 RECOMPUTERCDs       ▷ Recomputes MCDs, PCDs of vertices with
  updated K values
```

Algorithm 5 updates the K values of vertices by utilizing Algorithm 6, when edge (u, v) is inserted into the graph $G = (V, E)$. We start with PREPARERCDs procedure, which prepares residential core degrees as explained in Sect. 4.3.1. Then, we do a DFS starting from the root, say r, and at each step, we pop the vertex v from the top of the stack and push some of its neighbors, say w, into the stack, if v and w are candidates to be in a $k+1$-core, where $k = K(r)$. If v cannot be in a $k+1$-core, then we mark it as evicted and initiate a recursive eviction from v. In a recursive eviction, the `cd` values of vertices x are decremented, for $(v, x) \in E$ and $K(x) = k$. If the `cd` value of x turns out to be equal to k and x is not already marked as evicted, then we start another eviction from x. When DFS finishes, we increment the K values of all vertices that were visited but not evicted. Last, we adjust the residential core degrees by RECOMPUTERCDs procedure as discussed in Sect. 4.3.1.

Algorithm 6: PROPAGATEEVICTION($G(V, E), K()$, cd[], evicted[], k, v)

```
Data: G: the graph, K: max-k values, cd: cd values, evicted:
      evicted values, k: max-k of root, v: evicted vertex
  evicted[$v$] $\leftarrow$ true
  for each $(v, w) \in E$ do
      if $K(w) = k$ then
1         cd[$w$] $\leftarrow$ cd[$w$] − 1
2         if cd[$w$] = $k$ and not evicted[$w$] then
              PROPAGATEEVICTION($G, K$, cd, evicted, $k, v$)
```

Theorem 6 *Algorithm 5 updates the K values of the vertices upon an edge insertion.*

Proof Algorithm 5 combines the first and the second parts of the FINDPURECORE procedure. In Algorithm 5, we apply two principles: (1) we only visit the vertices that are in a subset of the purecore of the root, thanks to line 4 of Algorithm 5 (see Definition 5), and (2) we mark the vertices recursively as evicted if their *PCD* values cannot exceed their K value. If a vertex is evicted, that means it cannot increase its K value, and we enforce this by line 6 of Algorithm 5 and line 2 of Algorithm 6. Furthermore, we propagate this eviction mechanism recursively by checking the `cd` value of neighbors.

Proof of (1) is by Theorem 5. Proof of (2) has two parts. In part (a), when an edge is inserted, if $PCD(u) \leq K(u)$ for a vertex $u \in V$, then u cannot increase its K value as shown in lines labeled 6 and 2 in Algorithms 5 and 6, respectively. Assume it does and say that $k = K(u)$. Then, after $K(u)$ increases, u must have at least $k+1$ neighbors with greater or equal K value, by Observation 4. However, at most k neighbors of u can have their K values greater than or equal to k after $K(u)$ increases, since $PCD(u) \leq K(u)$ before $K(u)$ is increased, i.e., a contradiction. In part (b), we prove that if $PCD(u) \leq K(u)$, where u is the visited vertex, then $PCD(w)$ must be decremented as shown in line labeled 1 in Algorithm 6, where w is a neighbor of u having K value of $K(u)$. Assume that $PCD(w)$ is not decremented. Then, u is supposed to be in the max-core of w, if w increases its K value. However, u cannot be in the max-core of w, since it cannot increase its K value as proved in the first paragraph of proof, i.e., a contradiction.

We traverse the graph starting from the root and evict some of the vertices during this process. Non-evicted and traversed vertices increment their K values at the end of the algorithm. This is because all visited vertices will have a positive cd value, and if a visited vertex is not evicted, that means its cd value is above k (by line 3 of Algorithm 5).

Time complexity Algorithm 5 is basically doing a depth-first traversal on vertices whose cd values are greater than the K value of the root, and evicting the vertices whose cd values are equal to K of the root. As a result, in the worst case, we will end up traversing the entire graph. PREPARERCDS and RECOMPUTERCDS procedures have $O(|E|)$ time complexities as well. More generic forms of maintaining *RCD* values are presented in the "Appendix," and in the worst case, they will end up traversing the entire graph for preparing/recomputing *RCD* values. In total, worst-case time complexity for Algorithm 5 is again $O(|E|)$.

Space complexity We maintain two auxiliary arrays, to maintain *MCD* and *PCD* values, with size $O(|V|)$. Thus, the space complexity for Algorithm 5 is $O(|V|)$.

4.3.3 Edge removal

Edge removal using the traversal algorithm employs a similar on-the-fly updating of the cd values. A key difference from the edge insertion algorithm is that the edge removal relies on a simple recursion on the vertices whose K values should be decremented.

The traversal algorithm for edge removal is presented in Algorithm 7, with the helper Algorithm 8. We start with preparing residential core degrees as explained in Sect. 4.3.1. Depending on the equality of K values of the edge extremities, i.e., u_1 and u_2, we apply one or two recursive propagation operations to correctly calculate the K values. In the propagation operation, if the cd value of v turns out to be below its K value (i.e., K needs to be decremented), we perform a recursive *dismissal* operation starting from v, which is given in Algorithm 8. In the recursive dismissal operation, we decrement $K(v)$ and the cd values of vertices w, where $(v, w) \in E$, $K(w) = k$, and k is the K value of the root. If w gets a smaller cd value than k and $K(w)$ has not decremented yet, then we start another recursive dismissal, but this time from w. When the recursion completes, we adjust the residential core degrees as discussed in Sect. 4.3.1.

Theorem 7 *Algorithm 7 updates the K values of the vertices upon an edge removal.*

Proof Proof relies on Theorem 4 and Observation 4. Algorithm 7 is basically finding the vertices that have the same

Algorithm 7: TRAVERSAL: REMOVEEDGE($G(V, E)$, $K()$, $MCD()$, $PCD()$, u_1, u_2)

```
Data: G: the graph, K: max-k values, MCD: max-core degrees, PCD:
      purecore degrees, (u_1,u_2): removed edge
$r \leftarrow u_1$                                          ▷ Set the root
if $K(u_2) < K(u_1)$ then $r \leftarrow u_2$
$G \leftarrow G \setminus (u_1, u_2)$                       ▷ Remove the edge from G
PREPARERCDs  ▷ Prepares MCDs, PCDs of vertices around removed edge
▷ Perform a DFS traversal over vertices that have root's K value,
  while dismissing the ones that cannot be a part of a k-1-core
visited[$v$] = false, $\forall v \in V$          ▷ To perform DFS (lazy init)
dismissed[$v$] = false, $\forall v \in V$    ▷ To remember dis. vertices (lazy init)
cd[$v$] = 0, $\forall v \in V$       ▷ To find vertices to be dismissed (lazy init)
$k \leftarrow K(r)$                          ▷ Remember the K value of the root
if $K(u_1) \neq K(u_2)$ then
    visited[$r$] ← true
1   cd[$r$] ← $MCD(r)$
2   if cd[$r$] < $k$ then
        PROPAGATEDISMISSAL($G$, $K$, $MCD$, cd, dismissed,
        visited, $k$, $r$)
else
    visited[$u_1$] ← true
3   cd[$u_1$] ← $MCD(u_1)$
4   if cd[$u_1$] < $k$ then
        PROPAGATEDISMISSAL($G$, $K$, $MCD$, cd, dismissed,
        visited, $k$, $u_1$)
    visited[$u_2$] ← true
5   cd[$u_2$] ← $MCD(u_2)$
6   if not dismissed[$u_2$] and cd[$u_2$] < $k$ then
        PROPAGATEDISMISSAL($G$, $K$, $MCD$, cd, dismissed,
        visited, $k$, $u_2$)
RECOMPUTERCDs    ▷ Recomputes MCDs, PCDs of vertices with updated K values
```

K value as the root (by Theorem 4), detecting the ones contradicting Observation 4, and decrementing their K values. Throughout the removal process, we make sure that the vertices that have the same K value as the root follows Observation 4. To do that, we maintain current degree (cd) values of vertices by initializing them via *MCD* values at the beginning (lines 1, 3 and 5 of Algorithm 7), increasing when we first visit (line 1 of Algorithm 8), and decrementing when K value of a neighbor changes (line 2 of Algorithm 8). Lines 2, 4, and 6 of Algorithm 7, and line 3 of Algorithm 8 checks if *MCD* of a vertex is less than its K value (Observation 4), and decrements its K value if so.

Time complexity Algorithm 7 is quite similar to Algorithm 5 in that it is a depth-first search traversal on vertices whose cd values are less than the K value of the root. In the worst case, we will end up traversing the entire graph. PREPARERCDS and RECOMPUTERCDS procedures have $O(|E|)$ time complexities, and in total, the worst-case time complexity for Algorithm 7 is $O(|E|)$.

Space complexity Similar to Algorithm 5, we maintain two auxiliary arrays, for *MCD* and *PCD* values, each with size

Algorithm 8: propagateDismissal($G(V, E)$, $K()$, $MCD()$, cd, dismissed, visited, k, v)

```
Data: G: the graph, K: max-k values, MCD: max-core degrees, cd:
      cd values, dismissed: dismissed values, visited: visited
      values, k: max-k of root, v: dismissed vertex
dismissed[$v$] ← true
$K(v) \leftarrow K(v) - 1$                    ▷ The vertex is part of a k-1-core
for each $(v, w) \in E$ do
    if $K(w) = k$ then
        if not visited[$w$] then
1           cd[$w$] ← cd[$w$] + $MCD(w)$
            visited[$v$] ← true
2       cd[$w$] ← cd[$w$] − 1
3       if cd[$w$] < $k$ and not dismissed[$w$] then
            propagateDismissal($G$, $K$, $MCD$, cd, dismissed,
            visited, $k$, $w$)
```

$O(|V|)$. The space complexity for Algorithm 7 is $O(|V|)$. In the removal algorithm, we do not need to use *PCD* values. *PCD* of a vertex gives an estimate about how likely the K value of a vertex can increase. However, it does not say anything about the decrease in a K value, since there is no relationship between K and *PCD* value of a vertex that can be used in place of Observation 4.

4.4 Illustrative example

Figure 2 illustrates the subcore, purecore, and traversal algorithms using a sample graph. The edge drawn using a dashed bold line is the one that is being inserted into the graph. The vertex shown in black is the root vertex. The graph shows the K values and the *MCD* values for each vertex before the insertion. The set of vertices visited by each one of the subcore, purecore, and the traversal algorithms, for the purpose of updating the K values, is shown in the figure. The subcore algorithm visits the vertices with K value of 2, which are reachable from the root. The purecore algorithm visits the vertices with K value of 2 and *MCD* value of greater than 2 that are reachable from the root.

The traversal algorithm starts by updating the *MCD* value of the root to 5, due to the new edge. Then, DFS starts and pushes the root to the stack. When the root is popped from the stack, its two neighbors with (K, *MCD*) values of (2, 3) are pushed to the stack (*MCD* values greater than K value of the root, indicating that they can potentially be part of a larger core). Say that those vertices are x at the top and y at the bottom in Fig. 2. Based on Definition 6, the cd values of x and y are updated to 2 since their *PCD* values are 2. After that, we move to the next iteration and pop vertex x from the stack. The cd value of x is 2, which is not greater than the K value of the root. This means that it cannot participate in a higher core. As a result, no neighbors of x are visited and propagateEviction is initiated for x. In propagateEviction, x is evicted and the cd values of all neighbors of x are

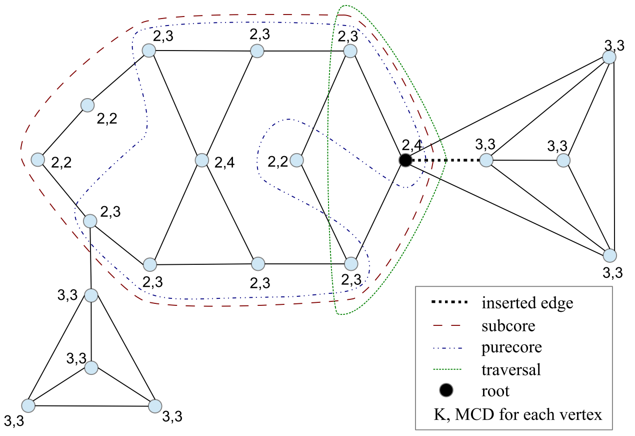

Fig. 2 Illustration of the vertices visited by the subcore, purecore, and the traversal algorithms

decremented, since all neighbors have a K value of 2 (same as root). Furthermore, propagateEviction is not initiated for any neighbor of x, since the cd value of the root (one of x's neighbors) becomes 4, and the cd value of other two neighbors of x becomes -1, all of which are different than the K value of the root.

In the next step, the DFS pops vertex y from the stack. Similar to x, the cd value of y is 2, which is not greater than the K value of the root. As a result, no neighbors of y are visited and propagateEviction is initiated for y. In propagateEviction, y is evicted and the cd values of all neighbors of y are decremented, since they have a K value of 2 (same as root). Furthermore, propagateEviction is not initiated for any neighbor of y, since the cd value of y's neighbors differs from the K value of the root. After these operations, the stack is empty, and the only vertex that is visited but not evicted is the root. As a result, the K value of the root is incremented. As the last step, the *MCD* and *PCD* values of vertices are updated as explained in Sect. 4.3.1.

We can easily see that the set of vertices visited by the subcore algorithm is larger than that of the purecore algorithm, whereas the traversal algorithm visits the smallest number of vertices compared to the other two.

4.5 Generic multihop traversal algorithm for insertion

The traversal algorithm that handles edge insertions, presented in Sect. 4.3, makes use of the *MCD* and *PCD* values of the vertices. *MCD* value of a vertex contains information from the 1-hop neighborhood, whereas *PCD* value contains information from the 2-hop neighborhood. However, the traversal algorithm can be generalized to utilize multihop information (greater than 2-hops). Higher hop counts enable faster detection of vertices that cannot appear in a larger core, yet increase the time spent to maintain the residential core degrees. As such, it involves a trade-off. Yet, in order to investigate this trade-off, we need to support using information

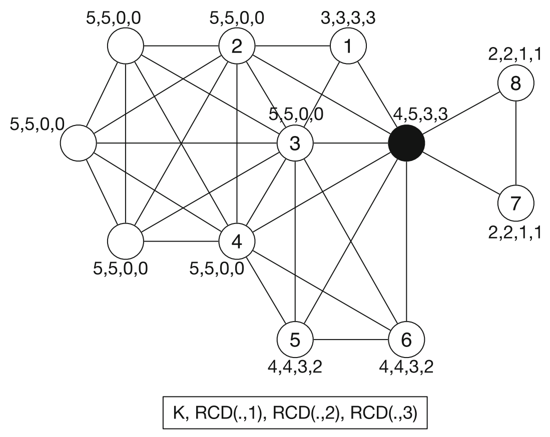

Fig. 3 Illustration of RCD values of the vertices in the sample graph

from arbitrary number of hops. Accordingly, in this section, we present the generic traversal algorithm for edge insertion, which leverages the multihop residential information of vertices for potentially faster calculation of K values.

First, we present the generic definition for n-hop residential core degrees.

Definition 7 The n-core degree of a vertex u, denoted as $RCD(u, n)$ where $n \geq 0$, is defined in terms of the number of u's neighbors, w, such that *either* $K(u) = K(w) \wedge RCD(w, n-1) > K(u)$ *or* $K(u) < K(w)$. When $n = 0$, $RCD(\cdot, n)$ of a vertex is defined to be ∞.

For a vertex v, its n-core degree is defined recursively. It is simply the number of neighbors w it has with either higher K value than v's K value or has equal K value and higher $(n-1)$-core degree than v's K value. With higher values of n, $RCD(\cdot, n)$ value of a vertex becomes a stronger indicator of eligibility to increase its K value. Value of n implies the extent of neighborhood information being used. For $n = 1$, only the information on 1-hop neighbors is used, and for $n = 2$, the information on hop-1 and hop-2 neighbors is utilized. Note that, when $n = 1$, $RCD(\cdot, n)$ definition reduces to MCD (maximum-core degree), given in Definition 4. Also, when $n = 2$, it has the same definition as the PCD (pure-core degree), given in Definition 6.

Observation 6 *For a given graph $G = (V, E)$ and a vertex $u \in V$, $RCD(u, n) \geq RCD(u, n+1)$ for $n \geq 0$.*

The observation is a direct result of Definition 7. Increasing n values decreases the number of neighbors that can satisfy $K(u) = K(w)$ and $RCD(w, n-1) > K(u)$.

Figure 3 shows an example graph to illustrate the $RCD(u, n)$ definition. K, $RCD(\cdot, 1)$ (MCD), $RCD(\cdot, 2)$ (PCD), and $RCD(\cdot, 3)$ values of vertices are shown next to them. For example, the $RCD(\cdot, 3)$ value of the black vertex is computed as follows: There are three neighbors (vertices 2, 3, and 4) with a K value of 5, which is greater than the K value of black vertex. The $RCD(\cdot, 3)$ value is then incremented by 3. Vertices 1, 7, and 8 have smaller K values than the black vertex; thus, they are not counted. The K value of vertices 5 and 6 is equal to the K value of the black vertex, and therefore, we check whether their $RCD(\cdot, 2)$ values are greater than their K values. However, it is not the case, since $RCD(\cdot, 2)$ value of both vertices is 3 and both vertices have a K value of 4. As a result, $RCD(\cdot, 3)$ value of the black vertex is set to 3.

The generalized traversal algorithm for insertion, MultihopTraversalInsertEdge, which utilizes the multihop information based on a given hop distance n (where $n > 1$), is quite similar to Algorithm 5. The main difference in the multihop traversal algorithm is that we use $RCD(\cdot, n)$ values instead of PCD values and $RCD(\cdot, n-1)$ values in place of MCD values. Differences between Algorithm 5 and MultihopTraversalInsertEdge are on lines 1, 2, 4, 5, and 7 (of Algorithm 5).

Instead of lines 1 and 7, we use multihopPrepareRCDsInsertion and multihopRecomputeRCDs procedures, respectively, for the generalized version of RCD maintenance for multihop residential core degrees. The details of generalized RCD maintenance are given in "Appendix." In lines 2, 4, and 5, RCD values of hop n are used to reduce the traversal space. We use $RCD(r, n)$ in place of $PCD(r)$ in line 2, where n is the number of hops and parameter of the algorithm. Likewise, $RCD(w, n-1)$ is put in place of $MCD(w)$ in line 4 and $RCD(w, n)$ to replace $PCD(w)$ in line 5. As stated earlier, RCDs with increasing hop values become stronger indicators of whether the K value of a vertex will increase or not. By Observation 6, we may have lower values for $RCD(\cdot, n)$ for higher n values. Keeping $RCD(\cdot, n)$ values low will help us to terminate MultihopTraversalInsertEdge algorithm earlier, due to the condition in line 3.

Time complexity The only difference between MultihopTraversalInsertEdge and Algorithm 5 is the use of $RCD(\cdot, n)$ values in place of MCD and PCD, and the generic multihopPrepareRCDsInsertion and multihopRecomputeRCDs procedures. Using $RCD(\cdot, n)$ values does not bring any additional complexity. The generic versions for preparing and recomputing RCDs has $O(h \cdot |E|)$ complexity, and since h (number of hops) is small, total worst-case complexity of insertion is $O(|E|)$.

Space complexity Different than Algorithm 5, we maintain h number of auxiliary arrays, for RCD values, where each has a size of $O(|V|)$. Thus, the space complexity for Algorithm 7 is $O(h \cdot |V|)$. However, since h is small, the overall space complexity is given as $O(|V|)$.
We expect that the traversal algorithm will explore a smaller space with higher n values. However, higher n values result in increased RCD maintenance cost. We experimentally evaluated our multihop traversal algorithm for different n values to find the optimal hop value. As discussed later in Sect. 6.5, the optimal value changes based on the dataset.

5 Implementation

In this section, we provide details about efficient implementations of the incremental algorithms presented. In particular, we discuss two main issues: the lazy initialization of arrays used in the algorithms, and the repeated sorting of the `cd` (current degree) arrays.

5.1 Lazy arrays

The non-incremental algorithms for computing the k-core decomposition perform work that is proportional to the size of the graph. As a result, our incremental algorithms should avoid any operation that requires work in the order of the size of the graph. However, several of our algorithms include arrays like `visited`, `evicted`, `cd` that are initialized to a default value and accessed using vertex indices. For these, we use lazy arrays to avoid allocations and initializations in the order of the graph size.

A lazy array employs a hash map-based data structure to implement a sparse array. For a given vertex, if its value is not currently being stored in the hash map, it is assumed to have the designated default value. When a different value for the vertex needs to be stored, the entry for it is created in the hash map.

Since hash maps provide constant lookup time, using lazy arrays achieves significant speedup when the number of vertices visited by the incremental algorithms is smaller than the graph size. On the other hand, when the number of vertices visited gets large, relative to the graph size, lazy arrays start performing worse, since the constant overhead of accessing a data item in a hash map is significantly higher than that of regular arrays. We checked the existing implementations for hash maps and used dense_hash_map library[1] for better performance.

Given that our algorithms locate a small subset of vertices for updating the k-core decomposition of a graph, the use of lazy arrays is almost always beneficial. For graphs that have very large subcores, relative to the graph size (which we show to be an uncommon occurrence in practice), an implementation of lazy arrays that switches to a dense representation when the occupation percentage of the array gets larger can be an effective solution, even though we do not implement that variation in this study.

5.2 Bucket sort

Several of our algorithms require reordering the set of unprocessed vertices in a subgraph (such as a subcore or a purecore) based on their `cd` values. In the worst case, this subgraph could be as large as the graph itself (again, this is uncommon in real-world graphs). To perform this re-sorting efficiently, we use bucket sort. Note that the `cd` values have a very small range, and thus bucket sort not only provides $O(N)$ sort time for the initial sort (where N is the subcore or purecore size), but it also enables $O(1)$ updates when a vertex changes its `cd` value (in our case, the values only decrease). We use a bucket data structure that relies on linked lists for storing its bucket contents and on a hash map to quickly locate the link list entry of any given vertex.

6 Experimental evaluation

In this section, we evaluate how the proposed algorithms behave under different scenarios. The first set of experiments shows the scalability of our best performing algorithm by studying its runtime performance as the size of the synthetic datasets increases. The second set of experiments compares the performance of our incremental algorithms with respect to each other on real datasets. The third experiment investigates the performance variation depending on the K values of u and v, when an edge (u, v) is inserted/removed. The last set of experiments examines the performance trade-offs associated with the multihop traversal insertion and generic *RCD* maintenance algorithms.

Our algorithms are implemented in C++ and compiled with `gcc 4.4.4` at –O2 optimization level. All experiments are executed sequentially on a Linux operating system running on a machine with two Intel Xeon E5520 2.27 GHz CPUs, with 48 GB of RAM.

6.1 Datasets

Our dataset includes synthetic and real-world graphs. For synthetic graphs, we use the SNAP library [30] to generate networks following three different models. The first is the Erdős–Renyi (ER) model, which generates random graphs [15]. The second is the Barabasi–Albert (BA) preferential attachment model [6], which follows a power law for the vertex degree distributions. The third model, generated with SNAP's R-MAT generator [9], follows a power law vertex degree distribution and also exhibits small-world properties. We set the partition probabilities as [0.45, 0.25, 0.20, 0.10], to approximate the k-core distribution of real citation graphs in our dataset. For all synthetic graphs, we specify the average degree as 8 so that different synthetic graphs with same number of vertices also have the same number of edges.

Figures 4 and 5 show the cumulative distribution of K values and purecore sizes (i.e., number of edges of the purecore subgraph of each vertex in the graph) for the synthetic datasets with 2^{24} vertices. For a graph $G = (V, E)$, we calculate the purecore of each vertex $u \in V$ by using

1 https://github.com/sparsehash/sparsehash.

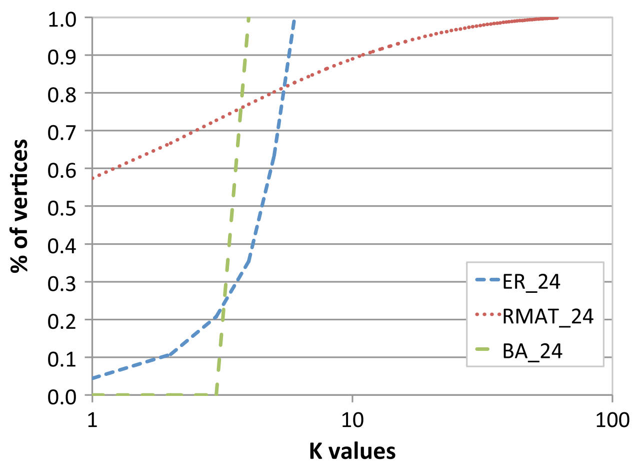

Fig. 4 Cumulative K value distribution for synthetic graphs

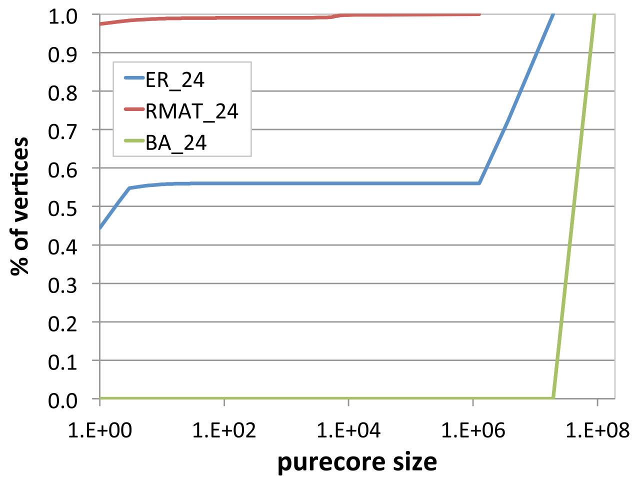

Fig. 5 Cumulative purecore size distribution for synthetic graphs

FINDPURECORE procedure, explained in Sect. 4.2. These figures reveal the structure of the generated graphs and how it impacts the incremental k-core decomposition performance. The K value distribution is an indication of the connectivity of the graph, while the purecore size is an indication of the potential runtime of our incremental algorithms when an edge incident upon a given vertex is inserted/removed.

As shown in Fig. 4, the graph based on the Barabasi–Albert model (BA_24) has 100 % of its vertices with $K = 4$. In addition, all of its vertices result in a purecore size of over 100 million edges. These properties of the BA graphs are due to the graph generation algorithm of the BA model, where newly inserted edges are likely to connect high-degree vertices. As we will see shortly, real-world graphs do not follow such properties, and the figure shows that the BA model is very poor in approximating real-world graphs in terms of the K value distribution. The RMAT-generated graph (RMAT_24) has nearly 60 % of its vertices with very low K values. As the K value increases, the percentage of vertices with that K value decreases. Furthermore, 98 % of its vertices have very small purecore sizes. The ER-generated graph (ER_24) has K values up to 6, and as the K value increases, the percentage of vertices with that K value also increases. The latter behavior is unlike the RMAT-generated graph. As we will see shortly, most real-world graphs of interest behave more closely to the RMAT-generated graphs with respect to their K value distribution.

The real graphs we use are from the 10th DIMACS Graph Partitioning and Graph Clustering Implementation Challenge repository [11] and include Internet router level and European domain computer network graphs (caidaRouterLevel and eu-2005), co-author and citation network graphs (citationCiteseer, coAuthorsCiteseer, coAuthorsDBLP, coPapersCiteseer), condensed matter collaboration network graphs (cond-mat), power grid network graphs (power), and protein interaction network graphs (protein interaction 1). Table 1 provides the details about each used graph, including their vertex and edge set size, maximum and average degrees, and their maximum k value. All graphs are undirected.

Figure 6 shows the K value distribution for all graphs in Table 1. The figure shows that the vertices of both coPapersCiteseer and eu-2005 have highly variant K values. Figure 7 shows the purecore size distribution for our real datasets. The data indicate that all of the graphs have at least 80 % of their vertices with corresponding purecore sizes of less than 100. This is an indication that our incremental algorithms are expected to perform well on these graphs.

As all our graphs are originally static, we emulate an incremental algorithm by considering that the whole set of edges and vertices constitutes a *sliding window* snapshot. For evaluating algorithm execution, we first evict a random edge from the current graph in the window. This emulates the behavior of a full sliding window, which must open space for inserting a new data item. We then insert a new edge between two random vertices. We also evaluated worst-case execution times by inserting and removing edges from vertices that have top *purecore* sizes. Such results had similar trends as the random insertion case and are omitted for brevity. Note that we do not assume any specific data distribution with respect to which edges get inserted or deleted. In addition, we make no assumptions regarding edge arrival rates. Instead, we evaluate the performance of our algorithms by determining the maximum update rate they support. For all experiments, we remove a randomly selected edge from the graph, and then insert the same edge to make a fair comparison between insertion and removal behavior. We repeat these removals and insertions 500 times.

6.2 Scalability

In this experiment, we evaluate the performance of the traversal algorithm (Sect. 4.3) as the size of the synthetic graphs

Table 1 Real-world graph datasets and their properties

Graph file	Number of vertices	Number of edges	Maximum degree	Average degree	Max *k*
caidaRouterLevel	192,244	609,066	1071	6.336	32
eu-2005	862,664	16,138,468	68,963	37.415	388
citationCiteseer	268,495	1,156,647	1318	8.616	15
coAuthorsCiteseer	227,320	814,134	1372	7.163	86
coAuthorsDBLP	299,067	977,676	336	6.538	114
coPapersCiteseer	434,102	16,036,720	1188	73.885	844
cond-mat	16,726	47,594	107	5.691	17
power	4941	6594	19	2.669	5
protein interaction 1	9673	37,081	270	7.667	14

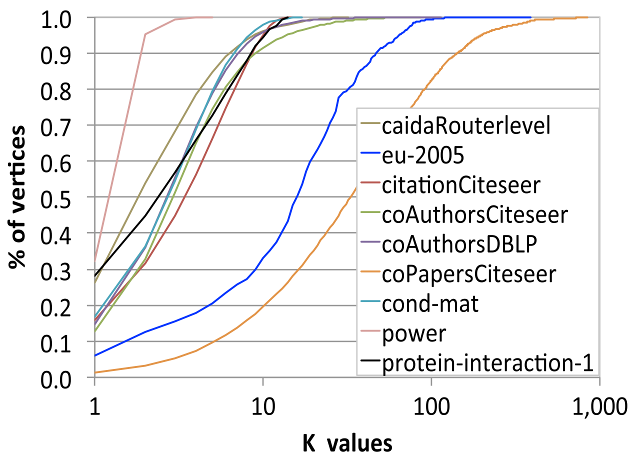

Fig. 6 Cumulative K value distribution for real-world graphs

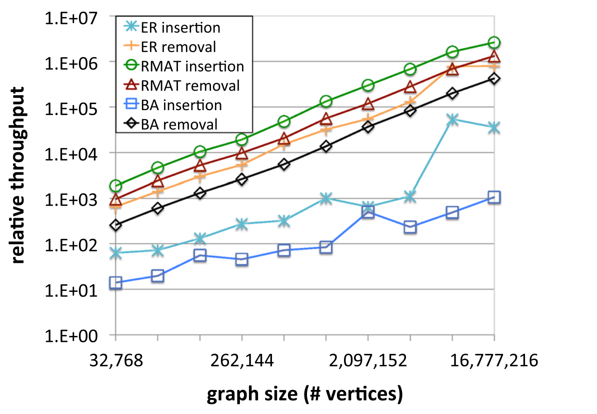

Fig. 8 Relative throughput of incremental insertion and removal algorithms for synthetic graphs when varying the graph size from 2^{15} to 2^{24}. Relative throughput is the ratio of the non-incremental algorithm runtime to the given incremental algorithm runtime. Removal scales better than insertion, reaching around 10^6 relative throughput

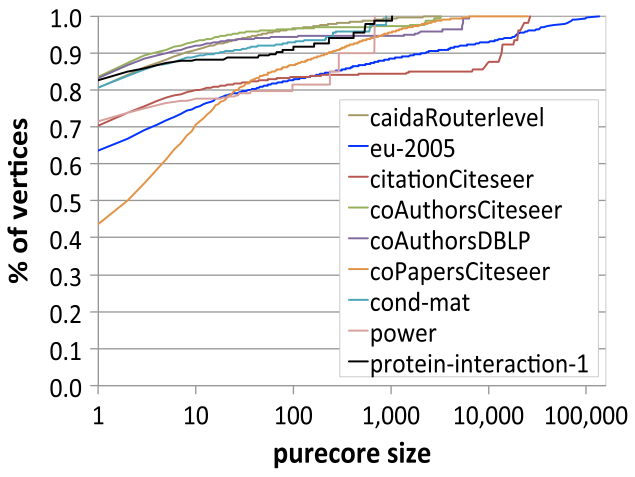

Fig. 7 Cumulative purecore size distribution for real-world graphs

increase. We first report relative throughput numbers, which are obtained by comparing the traversal runtimes with our baseline—the non-incremental version of k-core decomposition (Algorithm 1)—and then present the update rates, which show the number of edge removals/insertions processed per second. Testing the algorithm under different graph sizes emulates the scenario where an incremental algorithm uses different sliding window sizes.

Figure 8 shows the relative throughput of our incremental insertion and removal algorithms when the number of vertices from the graph ranges from 2^{15} to 2^{24}. For the insertion algorithm, the RMAT graph shows the best scalability, with relative throughputs ranging from 1865× to 2,624,690× (6 order of magnitude). This drastic relative throughput is because the K values of the vertices in the graph have high variability and majority of the vertices have very small purecore sizes, as shown in Figs. 4 and 5 for the RMAT graph with size 2^{24}. Such factors result in very fast insertions. The insertion of edges into the graph following the Erdős–Renyi model (ER) shows relative throughputs ranging from 63× to 53,968×. Although it also scales well with the size of the graph, the relative throughputs are not as high as the ones observed for the RMAT graph. This behavior can be explained by the fact that the ER graph has a more uniform K value distribution when compared to the RMAT graph. Furthermore, when the graph has size of 2^{24}, over

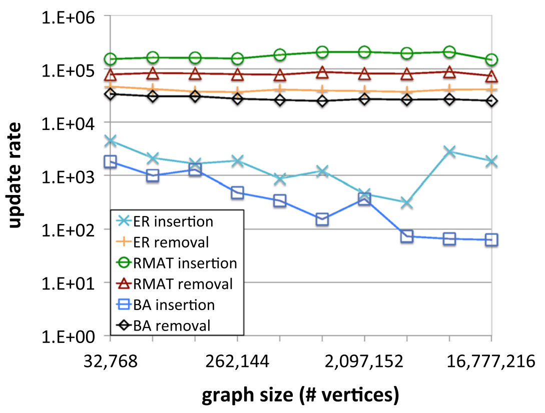

Fig. 9 Update rates of incremental insertion and removal algorithms for synthetic graphs when varying the graph size from 2^{15} to 2^{24}

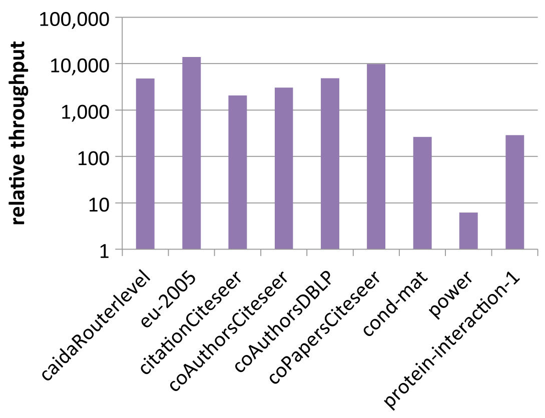

Fig. 10 Subcore algorithm relative throughput for real datasets when compared to the baseline. Our incremental algorithm runs up to 14,000× faster than the non-incremental algorithm

40 % of its insertions may result in touching purecores of over 1 million edges. When inserting edges into graph based on the Barabasi–Albert model (BA), our incremental algorithm is only slightly better than the non-incremental one. As we discussed earlier, in these graphs all vertices have the same K value initially, resulting in subcore sizes that are almost equal to the graph size. In this case, the incremental algorithm does not provide much benefit on top of the base one, yet brings additional computation overheads (such as due to lazy arrays). As we will show shortly, this nature of the BA graphs is not found in real-world graphs.

The removal algorithm scales for all three synthetic graphs, where the relative throughput ranges from 259× to 1,321,200×. For the ER and BA graphs, the removal algorithm scales better than the insertion one because it has much lower cost (see Sect. 4.3). At large scales, we notice that the use of incremental algorithms becomes even more critical, since the cost of the baseline is linear in the size of the graph.

The scalability experiments indicate how good our incremental algorithm can perform for different graph sizes when there are k-core decomposition queries (*read queries*) interspersed with edge insertion and removal (*write queries*). An example scenario is when the updates are coming in a batch, and the k-core decomposition is requested after each batch, we say $b =$ write/read, where b is the batch size. Taking the RMAT graph with size of 2^{24} vertices as an example, we can see that if the write/read ratio (batch size) is less than 1,972,945 (the average relative throughput of one removal and one insertion), it is better to use the incremental algorithm than to compute the k-core decomposition from scratch after inserting new edges and removing the oldest ones from the graph (sliding window scenario).

Figure 9 shows the update rates, i.e., number of edges processed per second, for our incremental insertion and removal algorithms when the number of vertices in the graph ranges from 2^{15} to 2^{24}. For RMAT graphs, both insertion and removal rates reach up to 205,000 and 87,000 updates/sec, and more importantly, update rates do not change when the graph size increases. ER graphs have lower update rates for both insertion and removal. Removal rates for ER graphs stay stable as the graph size increases and insertion rates only decrease by a factor of 3 (from 4478 to 1867) when the graph size increases from 2^{15} to 2^{24}. For BA graphs, update rates for removal decreases from 33,547 to 25,222 when the graph size increases. Insertion rate has a similar decreasing behavior with the graph size. However, the rates are lower—starting from 1809 and decreasing to 62 when the graph size gets bigger. The decreasing trend for the BA graphs is due to the large purecore sizes that are proportional to the graph size. Again, we will show that real-world graphs do not exhibit this behavior.

6.3 Performance comparison

In this experiment, we analyze how our three incremental algorithms perform when processing one edge removal and one edge insertion (i.e., one sliding window operation) on the real datasets described in Table 1. This helps us to see whether the algorithm that is expected to give the best results, traversal, shows the best performance for all the real datasets we have.

Figure 10 shows the performance of the subcore algorithm (Sect. 4.1) considering the average time taken by one graph update. The performance is shown in terms of the relative throughput provided by the incremental algorithm compared to the non-incremental one. The relative throughputs vary from 6.2× to 14,000×. The datasets in which the incremental algorithm performs the best are the eu-2005 and coPapersCiteseer. Similar to the results obtained in the synthetic graphs, the performance of the subcore algorithm benefits from the high variability in the K value distribution of the graph. The dataset in which the subcore algorithm performs the worst is power. This is because 63.19 % of the vertices

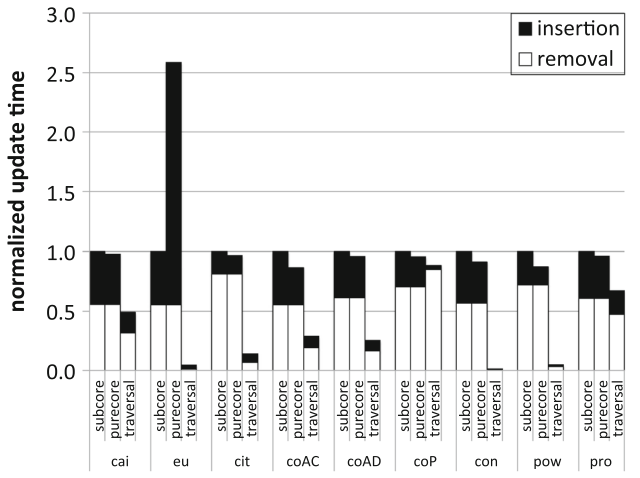

Fig. 11 Average update time comparison of incremental algorithms when processing real datasets. Times are normalized by the average update time of the subcore algorithm. Traversal algorithm shows the best performance for all datasets

in the power graph have the same K value, yielding large subcore sizes.

Figure 11 shows the average update time of each algorithm normalized by the update time of the subcore algorithm. Each group of three columns shows the results for a given dataset. For each group, the results are displayed in the following order: subcore (Sect. 4.1), purecore (Sect. 4.2), and traversal (with residential core degrees) (Sect. 4.3). The stacked columns represent the update time attributed to the removal (bottom) and insertion operations (top).

The results show that the purecore algorithm can perform worse than the subcore one for some datasets (eu-2005) even though the purecore of a vertex is always smaller than or equal to the subcore of a vertex. This is due to the additional work performed to locate a smaller subgraph. This additional work is not always worth it if the purecore is not sufficiently small compared to the subcore. The figure also displays that the traversal algorithm shows the best performance for all datasets, being up to 73× better than the subcore algorithm. The traversal algorithm shows dramatic improvement compared to subcore when processing eu-2005 and cond-mat graphs. Our results also show that the traversal algorithm has the most efficient removal for all datasets.

We also investigate the impact of residential core degrees on synthetic graphs generated using the Erdős–Renyi model. Table 2 shows the average time in seconds spent for one edge removal plus one edge insertion with the traversal algorithm. For each graph, we ran the traversal algorithm with and without the Residential Core Degrees. The results show that using Residential Core Degrees provides up to 81 % reduced running time. The results for RMAT show less improvement. We expect that this is due to their small-world property, which

Table 2 Average runtimes (s) for one edge removal plus one edge insertion with traversal algorithm on Erdős–Renyi graphs

Graph scale	With RCD (ratio)	Without RCD
16	0.019 (28 %)	0.067
18	0.082 (24 %)	0.335
20	0.396 (19 %)	2.047
22	2.039 (24 %)	8.600
24	1.914 (21 %)	8.991

Ratio shows with RCD runtimes relative to without

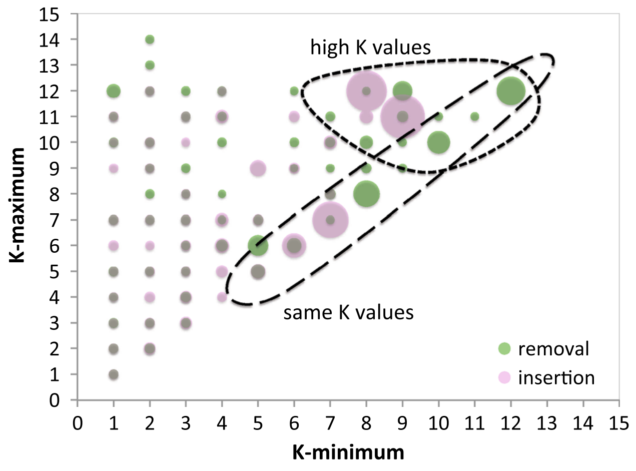

Fig. 12 Edge insertion and removal execution times of the traversal algorithm for different K values. Runtime shows low variability in general with higher runtimes when K values are high and/or equal

creates an advantage for manually computing *MCD*/*PCD* values.

6.4 Performance variation

In this section, we evaluate the performance of the traversal algorithm when inserting and removing random edges into vertices with varying K values. The objective was to understand how the execution time varies as edge insertions and removals are performed on different parts of the graph with different connectivity characteristics. For instance, performance implications of adding an edge between a vertex that has a high K value and one with a low K value versus between two vertices having close K values.

Figure 12 shows the performance results for the citationCiteseer graph, which serves as a good representative for our real dataset. This graph has vertices with K values varying from 1 to 15. A bubble in the graph indicates the time taken to insert or remove an edge between two random vertices u and v. If $K(u) \leq K(v)$, $K(u)$ is displayed on the x axis, while $K(v)$ is displayed on the y-axis. The size of the bubble indicates the average execution time for the insertion (pink) and removal (green) of an edge. The larger the bubble is, the greater the execution time is.

The graph shows that the runtime of the traversal algorithm has low variability. This is a good property, as it means that the algorithm is able to locate a small subgraph to traverse irrespective of the properties of the neighborhoods of the two vertices u and v. Our algorithm shows low runtime variability, as we consistently traverse subgraphs using the vertex with the lowest K value as root.

Execution times vary more when the K values of the different vertices are the same (diagonal). The reason is that the traversal algorithm visits the subgraphs associated with both vertices affected by the new edge, resulting in longer execution times. We also see that insertions between vertices with large K values have large execution times. In general, the execution times we see are proportional to the sizes of the subcores and *not* to the max-cores. In other words, what affects the execution time are the sizes of the subgraphs with the same K value. For small K values, such subgraphs are small, because they are bounded by higher K-valued vertices, which in turn belong to their max-core. For large K values, subcores are bigger, because large K-valued vertices tend to be close to each other due to the definition of k-core. Although their max-core sizes are small relative to that of small K-valued vertices, their subcore sizes turn out to be larger.

6.5 Multihop performance

In this set of experiments, we evaluate the impact of hop distance on the performance of multihop traversal insertion algorithm, given in Sect. 4.5, on real-world networks. We also evaluate the *RCD* maintenance algorithms. The goal was to observe how the maintenance times are affecting the total runtime for different hop distances and how the traversal space and time are reduced with increasing hop counts.

We show the normalized maintenance (black) and traversal (white) times for different hop counts (x axis) in Fig. 13. Normalization is done with respect to 2-hop results for each graph. We refer to the processing time taken by MULTIHOP-PREPARERCDsINSERTION and MULTIHOPRECOMPUTERCDs procedures of MULTIHOPTRAVERSALINSERTEDGE as "maintenance" time and to the time taken by the rest of the algorithm as "traversal" time. For a given graph, we inserted the same set of 500 edges for each hop count, which are randomly selected at the start, so that the comparison is fair. Note that the bars corresponding to 2-hops represent the traversal-based insertion algorithm, given in Sect. 4.3.

From the figure, we observe that the maintenance times are increasing with higher hop counts. For instance, 3-hop maintenance time is 2.4× more than the one for 2-hops, whereas the 4-hop and 5-hop times are 5.3× and 11.1× larger.

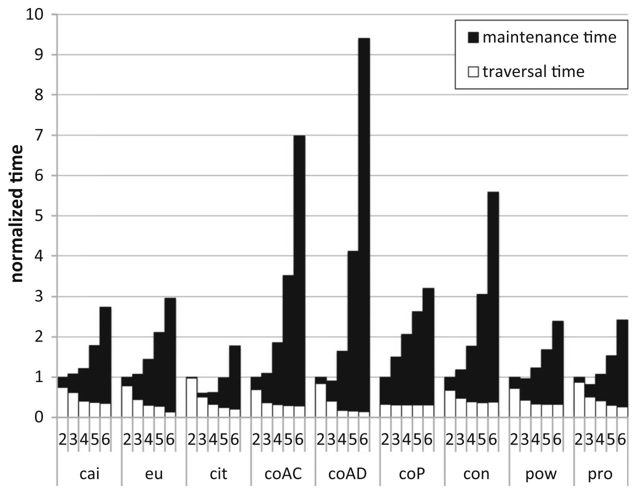

Fig. 13 Maintenance times increase with the higher hop counts, yet the traversal times decrease in general. When the running time of the 2-hop variant is dominated by the traversal time, increasing hop counts brings significant improvement in terms of the traversal times. 3-hop variants are shown to give the best overall performance for 4 of the graphs, out of 9 total

On the other hand, traversal times, shown with white bars, present a different picture. Higher number of hops results in reduced traversal spaces and thus lower running times. On average, 3-hop and 4-hop traversals are .16× and 2.4× faster than the 2-hop one. We observe that if the traversed graph space is high for the 2-hop algorithm (which is indicated by the long traversal times), it is likely to get better relative throughput with higher hop counts. For example, citationCiteseer (cit) graph requires large traversals when the 2-hop algorithm is applied. On average, the 2-hop traversal algorithm visits 340.9 vertices, and this number goes down to 169.2 and 101.3 with 3 and 4 hops, respectively. On the other hand, caidaRouterLevel (cai) graph requires 4.4 vertex traversals on average for the 2-hop algorithm, and this number is going down to 1.5 with higher hop counts. Therefore, if the 2 hop algorithm traverses significant space, then there is room for improvement, and this opportunity is leveraged well by the higher hop count algorithms. Otherwise, maintenance times become the bottleneck.

If we look at the total times, we see that 4 of the 9 graphs benefit from the higher hop counts, and best performance is obtained by 3-hop algorithms for those graphs. One distinguishing feature of these four graphs is that their 2-hop traversal times are more than 85% of the total time, which means that there is a significant room for improvement over the 2-hop variant.

To understand the running time changes with varying number of hops in a better way, we plotted the maximum, 90 %, and median times for the real-world graphs in Fig. 14. We normalized all the times with respect to the 2-hop median times. Median times for all graphs are around 0.01 ms, so they

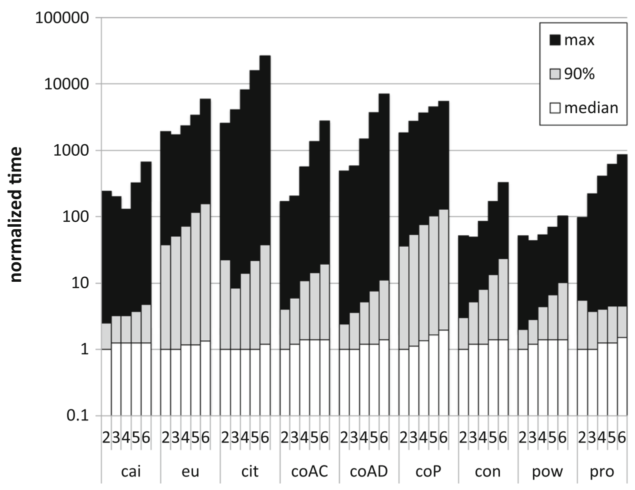

Fig. 14 Detailed running time comparison for varying hop counts. Given 500 edge insertions, *max bar* shows the longest time taken by an edge insertion, whereas *median bar* shows the median of the insertion times. 90 % *bar* shows the running time value such that 90 percentile of the edge insertions takes at most that much time

do not differ significantly for different hop counts. When we look at the 90 percentile bars, citationCiteseer (cit) and protein interaction 1 (pro) graphs show that 3-hop variant is superior to the 2-hop variant. Maximum times are ranging from 0.2 to 132 ms. The interesting thing is that maximum time bars for caidaRouterLevel (cai), eu-2005 (eu), and cond-mat (con) show that 3-hop and 4-hop variants result in better running times compared to the 2-hop variants, which cannot be observed for the total times shown in Fig. 13. This means that the larger hop counts reduce the variance in the edge insertion times, as they prevent very large traversals that the 2-hop variant sometimes encounters. These results confirm the fact that if there is a significant amount of work to do in order to adjust the *k*-core decomposition, then higher hops will provide better running times. Overall, we suggest to use 3-hops or 4-hops when the graph dataset used results in large traversals.

7 Related work

k-core concept is first introduced by Erdős and Hajnal [14] as the “degeneracy number” of graph, which is the largest *k* for which there is a non-empty *k*-core. The definition of *k*-core subgraph is first introduced by Seidman [29] to characterize the cohesive regions of graphs. Batagelj and Zaversnik [7] developed an efficient algorithm to find the *k*-core decomposition of a graph. In our work, we build upon these works to develop *k*-core decomposition algorithms that are incremental in nature, making it possible to apply these algorithms in dynamic settings where edge insertions and removals happen frequently, such as maintaining a recent history of a dynamic graph.

There are many application areas of *k*-core decomposition including but not limited to social networks [19,32], visualization of large networks [2,17,34], and protein interaction networks analysis [4,33]. In social network analysis, *k*-cores have been used for community detection [19], clustering [32], and criminal network detection [25].

Thanks to its well-defined structure, *k*-cores have been used extensively to analyze the structure of certain types of networks [12,23] and to generate graphs with specific properties [8]. Many graph problems such as maximal clique finding [5], dense subgraph discovery [3], and betweenness approximation [20] use *k*-core decomposition as a subroutine.

In terms of algorithms specific to finding *k*-core decompositions, an external memory algorithm for *k*-core decomposition is introduced in [10]. Authors propose several heuristics to bound K values of vertices and shares similar ideas with our *RCD* definitions. There are also studies about *k*-core decomposition on directed [18] and weighted [19] networks. As an effort to incremental *k*-core decomposition, Aksu et al. [1] introduced dense *k*-core subgraph maintenance algorithms in distributed settings. In their work, they materialized the *k*-core subgraph with large K values and maintain them for dynamic graphs. However, they ignore the maintenance of small K-valued dense subgraphs. In this respect, our work is unique in the sense that we provide maintenance of *all* dense subgraphs in a given graph.

Concurrently with our first work [28], Li et al. [22] published a report on incremental algorithms for core decomposition. Our algorithms differ from theirs in two important aspects: (1) They propose quadratic complexity incremental algorithms, whereas our algorithms have linear complexity. (2) The relative throughput results achieved by our algorithm outperform theirs. For instance, their best algorithm has 6.3× relative throughput on the cond-mat graph, while our best algorithm (traversal) achieves a relative throughput of 19,272×, which is more than three orders of magnitude higher.

8 Conclusion

In this paper, we have introduced incremental algorithms for *k*-core decomposition of graphs. The key feature of these algorithms is their incremental nature—the ability to update the *k*-core decomposition quickly when a new edge is inserted or removed, without having to traverse the entire graph. Our experimental evaluation shows that these incremental algorithms can perform significantly better than their batch alternatives, where the relative throughput in execution time increases with the increasing graph size. Given the importance of k-core decomposition in detection of dense regions and communities, maximum clique finding, and graph visualization, we believe these incremental algorithms will serve as a fundamental building block for future incremental solutions for other graph problems, where the updates are coming in a batch.

Acknowledgments This work is partially sponsored by the US Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program (Agreement No. W911NF-12-C-0028). The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the US Government.This work is also partially sponsored by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant EEEAG #112E271.

Appendix: Generic RCD maintenance

As stated earlier in Sect. 4.3.1, maintaining *RCD* values is a non-trivial operation. Yet, it is critical in reducing the scopes of the traversals, potentially bringing down the cost of edge modifications. Overall, efficient mechanisms for maintaining *RCD* values is needed. Here, we introduce the generic versions of the *RCD* maintenance algorithms, which update the *RCD* values of vertices up to the given hop count n. In other words, given the number of hops, n, the proposed algorithms maintain the *RCD* values for $n, n-1, \ldots, 1$.

multihopPrepareRCDsInsertion (Algorithm 9) is used at the beginning of the multihop traversal-based edge insertion algorithm, explained in Sect. 4.5. It prepares the *RCD* values before the multihop traversal operation is performed for the inserted edge, (u_1, u_2), and the given hop distance, n. This preparation is needed as the *RCD* values of the root(s) may have changed due to the updated degrees, and this change may have propagated to *RCD* values of other vertices. The preparation phase is performed assuming that the K values are intact. Those will be updated during the traversal, and a re-computation of *RCD*s would be required at the end (multihopRecomputeRCDs procedure).

The preparation starts with determining the *root* vertices based on their K values. If the K values of the extremities of the inserted edge are not equal, we increment the $RCD(r, h)$ value of root for all $h \leq n$. The rationale behind this is that the root vertex gains a new neighbor with a higher K value, and by Definition 7, it increases all *RCD* values of root by one. Following this increment operation, we check whether the $RCD(r, h)$ has exceeded k, because this implies further changes in $RCD(\cdot, h+1)$ values of r's neighbor vertices (by Definition 7). In the preparation phase, $RCD(\cdot, n)$ of a vertex only changes when $RCD(\cdot, n-1)$ of a neighbor changes and that is what we are checking for. Remember that $RCD(u, n)$ is the number of u's neighbors, w, where either $K(u) < K(w)$ or $K(u) = K(w)$ and $RCD(w, n-1) > K(u)$. Throughout the algorithm, we accumulate the vertices whose $RCD(\cdot, h)$ values just exceed k in the next *frontiers* set, where h is the

Algorithm 9: MULTIHOP RCD MAINTENANCE: multihopPrepareRCDsInsertion $(G(V, E), K(), RCD(,), n, u_1, u_2)$

```
Data: $G$: the graph, $K$: max-$k$ values, $RCD$: residential core degrees, $n$:
      number of hops ($> 1$), $(u_1,u_2)$: inserted edge
$r \leftarrow u_1$                                          ▷ Set the root
if $K(u_2) < K(u_1)$ then $r \leftarrow u_2$
$k \leftarrow K(r)$                            ▷ Remember the $K$ value of the root
$roots \leftarrow$ empty set
$frontiers \leftarrow n$ number of empty sets
if $K(u_1) \neq K(u_2)$ then
    for each $h \in [1..n]$ do                                  ▷ For each hop
        ▷ $r$ gets a new neig. with higher K value
        $RCD(r, h) \leftarrow RCD(r, h) + 1$
        if $h < n$ and $RCD(r, h) = k + 1$ then
            ▷ If the RCD value exceeds k, it is pushed to frontiers. RCD of
              neigs will be updated in the next iteration
            $frontiers[h + 1]$.push($r$)
        if $h > 1$ then
            ▷ Neigs of the vertices in frontiers are explored to update their
              RCD values
            for each $v \in frontiers[h]$ do
                for each $(v, w) \in E$ do
                    if $K(w) = k$ then
                        ▷ RCD value of every neig, with same K value, is
                          incremented
                        $RCD(w, h) \leftarrow RCD(w, h) + 1$
                        if $h < n$ and $RCD(w, h) = k + 1$ then
                            ▷ If the RCD value exceeds k, neigs will be
                              updated in the next iteration
                            $frontiers[h + 1]$.push($w$)
else
    for each $h \in [1..n]$ do                                  ▷ For each hop
        if $h = 1$ then                                     ▷ MCD computation
            ▷ $u_1$ and $u_2$ get a new neig with equal K value
            ▷ If the RCD value exceeds k, it is pushed to frontiers. RCD of
              neigs will be updated in the next iteration
            $RCD(u_1, h) \leftarrow RCD(u_1, h) + 1$
            if $RCD(u_1, h) = k + 1$ then
                $frontiers[h + 1]$.push($u_1$)
            $RCD(u_2, h) \leftarrow RCD(u_2, h) + 1$
            if $RCD(u_2, h) = k + 1$ then
                $frontiers[h + 1]$.push($u_2$)
        else
            ▷ Handle the newly inserted edge
            if $RCD(u_2, h - 1) > k$ then
                $RCD(u_1, h) \leftarrow RCD(u_1, h) + 1$
                if $h < n$ and $RCD(u_1, h) = k + 1$ then
                    $frontiers[h + 1]$.push($u_1$)
            if $RCD(u_1, h - 1) > k$ then
                $RCD(u_2, h) \leftarrow RCD(u_2, h) + 1$
                if $h < n$ and $RCD(u_2, h) = k + 1$ then
                    $frontiers[h + 1]$.push($u_2$)
            ▷ Neigs of the vertices in frontiers are explored to update their
              RCD values
            for each $v \in frontiers[h]$ do
                for each $(v, w) \in E$ do
                    ▷ Exclude the newly inserted edge
                    if not $(v = u_1$ and $w = u_2)$ and
                    not $(v = u_2$ and $w = u_1)$ and
                    $K(w) = k$ then
                        $RCD(w, h) \leftarrow RCD(w, h) + 1$
                        if $h < n$ and $RCD(w, h) = k + 1$ then
                            $frontiers[h + 1]$.push($w$)
```

hop number. We avoid this accumulation operation if the last hop number h is being processed, since there is no need for further processing in that case. When the hop number h is greater than 1, we process the neighbors (with the same K value) of the vertices in the current *frontiers* set by incrementing their $RCD(\cdot, h)$ values. We also perform checks to see whether k is exceeded and accordingly populate the next *frontiers* set.

If the K values of the extremities of the inserted edge are equal, we do different operations for $h = 1$ and $h > 1$, where h is the current hop number. For $h = 1$, where $RCD(u, 1)$ is actually equal to $MCD(u)$, we just increment the $RCD(\cdot, 1)$ values of both extremities of the inserted edge (by Definition 7) and perform checks to see whether k is exceeded and accordingly populate the next *frontiers* set. If $h > 1$, we need to handle the new inserted edge separately. Let us say u_1 and u_2 are the extremities of the inserted edge. We first check the $RCD(u_1, h-1)$ [and dually $RCD(u_2, h-1)$] is greater than k. If so, we increment the $RCD(u_2, h)$ [and dually $RCD(u_1, h)$] and perform the k value checks to populate the next frontier as needed. After that, we process the neighbors (with the same K value) of the vertices in current *frontiers* set. One important difference in this step is that we exclude the edge between u_1 and u_2, because that edge is already handled.

Multihop algorithms are only applicable for the edge insertion operation. For removal, using 1-hop information (*MCD* values) is necessary and sufficient, as stated in the last paragraph of Sect. 4.3.3. Therefore, going for multihop information does not bring any additional benefit in terms of the running time. However, given that we are interested in sliding window scenarios, where removals happen together with insertions, we need to accommodate the maintenance of *RCD* values when there is an edge removal. For this purpose, we develop the MULTIHOPPREPARERCDSREMOVAL method. Detailed pseudocode and explanation can be found in "RCD maintenance for edge removal" of appendix.

After the multihop traversal, if the K values of some vertices are incremented, then this will create a cascading effect on *RCD* values of the vertices around. Efficiently handling the cascades and doing the update operations is again of great importance. Algorithm 10 finds those vertices whose *RCD* values need to be updated and efficiently updates these *RCD* values. It has two main parameters: the set of vertices whose K values are updated (`changed`), and the hop distance until which *RCD* values are to be updated (n). We start the algorithm by marking the `changed` vertices as *visited*. Throughout the algorithm, we mark the vertices via the visited array to prevent duplicates during the update procedure. In the main for loop (the second one), we process the updates for each hop, in order. At each iteration, we populate the `changed` set with the updated vertices and then update the *RCD* values of the vertices in `changed`. Cascading effect propagates by a single hop neighborhood at each iteration. In other words, if we assume that a vertex u has its K value updated, and we want 3-hop distance *RCD* values to be updated; $RCD(1)$, $RCD(2)$ and $RCD(3)$ of u will be updated. Furthermore, $RCD(2)$ and $RCD(3)$ values of *some* vertices in u's hop-1 neighborhood will be updated, and $RCD(3)$ values of *some* vertices in u's hop-2 neighborhood will be updated.

Algorithm 10: MULTIHOP RCD MAINTENANCE: MULTIHOPRECOMPUTERCDS $(G(V,E), K(), RCD(,), n, \texttt{changed})$

```
Data: G: the graph, K: max-k values, RCD: residential core degrees,
      n: number of hops (> 1), changed: set of vertices with
      updated K value
visited[v] = false, ∀v ∈ V                                  ▷ Lazy init
for each v ∈ changed do
    visited[v] = true
for each h ∈ {1...n} do
    updated ← empty set
    for each v ∈ changed do
        for each (v, w) ∈ E do
            if not visited[w] and
            ▷ For insertion
                      (K(w) = K(v) or K(w) = K(v) − 1) then
                ▷ For removal
                ▷          (K(w) = K(v) or K(w) = K(v) + 1)
                updated.push(w)
                visited[w] = true
    for each v ∈ updated do
        changed.push(v)
    for each v ∈ changed do
        RCD(h)(v) ← COMPUTERCD (v, K, RCD, h)
```

Pruning the vertices in the neighborhood is critical in making the procedure efficient. For the edge insertion case, given vertex v, we prune the neighborhood vertices by checking whether they are visited previously and whether the K value of the neighbor vertex is either equal to K value of v or equal to K value of v minus 1 (plus 1 for the edge removal case). The reason behind this check is based on Definition 7. $RCD(n)$ of a neighbor vertex may change iff the K values are equal. For the edge insertion case, given that there are also some vertices whose K values are incremented, we need to consider them as well by checking the neighbor vertices with one less K value. Likewise, for the edge removal, we check the neighbor vertices with one more K value, as stated with comments in the pseudocode of Algorithm 10. We accumulate the vertices to be updated in `changed` set and update their *RCD* values for the hop distance at that iteration. COMPUTERCD procedure at the end finds the *RCD* values for all hop numbers up to h. It basically makes use of Definition 7. In summary, we handle the cascading effect of *RCD* maintenance efficiently by the aforementioned pruning techniques.

RCD maintenance for edge removal

Algorithm 11 adjusts the *RCD* values when there is an edge removal and is very similar to Algorithm 9. One important difference is that, instead of incrementing the *RCD* values, we need to decrement them whenever necessary. We also check whether the $RCD(\cdot, h)$ value goes below $k+1$, which implies changes in $RCD(\cdot, h+1)$ values of neighbor vertices. Another difference between Algorithms 11 and 9 exists when the K values of the removed edge extremities are equal. In this case, we need to remember the *RCD* values for all hop numbers before the edge removal operation. This enables us to process the hop numbers $h > 1$.

Algorithm 11: MULTIHOP RCD MAINTENANCE: MULTIHOPPREPARERCDSREMOVAL $(G(V, E), K(), RCD(,), n, u_1, u_2)$

```
Data: G: the graph, K: max-k values, RCD: residential core degrees,
      n: number of hops (> 1), (u_1,u_2): inserted edge
$r \leftarrow u_1$                                          ▷ Set the root
if $K(u_2) < K(u_1)$ then $r \leftarrow u_2$
$k \leftarrow K(r)$                         ▷ Remember the $K$ value of the root
$roots \leftarrow$ empty set
$frontiers \leftarrow n$ number of empty sets
if $K(u_1) \neq K(u_2)$ then
    for each $h \in [1..n]$ do                              ▷ For each hop
        $RCD(r, h) \leftarrow RCD(r, h) - 1$
        if $h < n$ and $RCD(r, h) = k$ then
            $frontiers[h+1]$.push($r$)
        if $h > 1$ then
            for each $v \in frontiers[h]$ do
                for each $(v, w) \in E$ do
                    if $K(w) = k$ then
                        $RCD(w, h) \leftarrow RCD(w, h) - 1$
                        if $h < n$ and $RCD(w, h) = k$ then
                            $frontiers[h+1]$.push($w$)
else
    ▷ Remember the old RCD values of $u_1$ and $u_2$
    $old\_RCD \leftarrow$ empty set for $u_1$ and $u_2$
    for each $h \in [1..n]$ do                              ▷ For each hop
        $old\_RCD(u_1, h) \leftarrow RCD(u_1, h)$
        $old\_RCD(u_2, h) \leftarrow RCD(u_2, h)$
    for each $h \in [1..n]$ do                              ▷ For each hop
        if $h = 1$ then
            $RCD(u_1, h) \leftarrow RCD(u_1, h) - 1$
            if $RCD(u_1, h) = k$ then
                $frontiers[h+1]$.push($u_1$)
            $RCD(u_2, h) \leftarrow RCD(u_2, h) - 1$
            if $RCD(u_2, h) = k$ then
                $frontiers[h+1]$.push($u_2$)
        else
            if $old\_RCD(u_2, h-1) > k$ then
                $RCD(u_1, h) \leftarrow RCD(u_1, h) - 1$
                if $h < n$ and $RCD(u_1, h) = k$ then
                    $frontiers[h+1]$.push($u_1$)
            if $old\_RCD(u_1, h-1) > k$ then
                $RCD(u_2, h) \leftarrow RCD(u_2, h) - 1$
                if $h < n$ and $RCD(u_2, h) = k$ then
                    $frontiers[h+1]$.push($u_2$)
            for each $v \in frontiers[h]$ do
                for each $(v, w) \in E$ do
                    if not $(v = u_1$ and $w = u_2)$ and
                    not $(v = u_2$ and $w = u_1)$ and
                    $K(w) = k$ then
                        $RCD(w, h) \leftarrow RCD(w, h) - 1$
                        if $h < n$ and $RCD(w, h) = k$ then
                            $frontiers[h+1]$.push($w$)
```

References

1. Aksu, H., Canim, M., Chang, Y., Korpeoglu, I., Ulusoy, O.: Distributed-Core View Materialization and Maintenance for Large Dynamic Graphs. Knowl Data Eng. IEEE Trans. **26**(10), 2439–2452 (2014)
2. Alvarez-Hamelin, J.I., Dall'Asta, L., Barrat, A., Vespignani, A.: *k*-Core decomposition: a tool for the visualization of large scale networks. In: The Computing Research Repository (CoRR), arXiv:abs/cs/0504107 (2005)
3. Andersen, R., Chellapilla, K.: Finding dense subgraphs with size bounds. In: Workshop on Algorithms and Models for the Web Graph (WAW), pp. 25–37 (2009)
4. Bader, G.D., Hogue, C.W.V.: An automated method for finding molecular complexes in large protein interaction networks. BMC Bioinf. **4**(1), 1–27 (2003). doi:10.1186/1471-2105-4-2
5. Balasundaram, B., Butenko, S., Hicks, I.: Clique relaxations in social network analysis: the maximum *k*-plex problem. Oper. Res. **59**, 133–142 (2011)
6. Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
7. Batagelj, V., Zaversnik, M.: An O(m) algorithm for cores decomposition of networks. In: The Computing Research Repository (CoRR), arXiv:cs.DS/0310049 (2003)
8. Baur, M., Gaertler, M., Görke, R., Krug, M., Wagner, D.: Augmenting k-core generation with preferential attachment. Netw. Heterog. Media **3**(2), 277–294 (2008)
9. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: SIAM International Conference on Data Mining (SDM) (2004)
10. Cheng, J., Ke, Y., Chu, S., Ozsu, M.T.: Efficient core decomposition in massive networks. In: IEEE International Conference on Data Engineering (ICDE), pp. 51–62 (2011)
11. DIMACS. 10th DIMACS Implementation Challenge. http://www.cc.gatech.edu/dimacs10
12. Dorogovtsev, S.N., Goltsev, A.V., Mendes, J.F.F.: *k*-core organization of complex networks. Phys. Rev. Lett. **96**(4), 040601 (2006)
13. Dourisboure, Y., Geraci, F., Pellegrini, M.: Extraction and classification of dense communities in the web. In: World Wide Web Conference (WWW), pp. 461–470 (2007)
14. Erdős, P., Hajnal, A.: On chromatic number of graphs and set-systems. Acta Math. Hung. **17**, 61–99 (1966)
15. Erdős, P., Rényi, A.: On the Evolution of Random Graphs, pp. 17–61. Institute of Mathematics, Hungarian Academy of Sciences, Budapest, Hungary (1960)
16. Fortunato, S.: Community detection in graphs. Phys. Rep. **483**(3–5), 75–174 (2009)
17. Gaertler, M.: Dynamic analysis of the autonomous system graph. In: International Workshop on Inter-domain Performance and Simulation (IPS), pp. 13–24 (2004)
18. Giatsidis, C., Thilikos, D.M., Vazirgiannis, M.: D-cores: Measuring collaboration of directed graphs based on degeneracy. In: IEEE International Conference on Data Mining (ICDM), pp. 201–210 (2011)

19. Giatsidis, C., Thilikos, D.M., Vazirgiannis, M.: Evaluating cooperation in communities with the *k*-core structure. In: International Conference on Advances in Social Network Analysis and Mining (ASONAM), pp. 87–93 (2011)
20. Healy, J., Janssen, J., Milios, E., Aiello, W.: Characterization of graphs using degree cores. In: Workshop on Algorithms and Models for the Web Graph (WAW), pp. 137–148 (2006)
21. Kortsarz, G., Peleg, D.: Generating sparse 2-spanners. J. Algorithms **17**(2), 222–236 (1994)
22. Li, R.-H., Yu, J.X.: Efficient Core Maintenance in Large Dynamic Graphs. CoRR, arXiv:1207.4567 (2012)
23. Luczak, T.: Size and connectivity of the *k*-core of a random graph. Discrete Math. **91**(1), 61–68 (1991)
24. Nanavati, A.A., Siva, G., Das, G., Chakraborty, D., Dasgupta, K., Mukherjea, S., Joshi, A.: On the structural properties of massive telecom call graphs: findings and implications. In: ACM International Conference on Information and Knowledge Management (CIKM), pp. 435–444 (2006)
25. Ozgul, F., Erdem, Z., Bowerman, C., Atzenbeck, C.: Comparison of feature-based criminal network detection models with k-core and n-clique. In: International Conference on Advances in Social Network Analysis and Mining (ASONAM), pp: 400–401 (2010)
26. Saito, H., Toyoda, M., Kitsuregawa, M., Aihara, K.: A large-scale study of link spam detection by graph algorithms. In: International Workshop on Adversarial Information Retrieval on the Web (AIRWeb), pp. 45–48 (2007)
27. Samudrala, R., Moult, J.: A graph-theoretic algorithm for comparative modeling of protein structure. J. Mol. Biol. **279**(1), 287–302 (1998)
28. Sarıyüce, A.E., Gedik, B., Jacques-Silva, G., Wu, K.-L., Çatalyürek, Ü.V.: Streaming algorithms for *k*-core decomposition. In: Proceedings of the Very Large Data Bases Conference (PVLDB), pp. 433–444 (2013)
29. Seidman, S.B.: Network structure and minimum degree. Soc. Netw. **5**(3), 269–287 (1983)
30. SNAP. Stanford network analysis package. http://snap.stanford.edu/snap
31. Turaga, D., Andrade, H., Gedik, B., Venkatramani, C., Verscheure, O., Harris, J.D., Cox, J., Szewczyk, W., Jones, P.: Design principles for developing stream processing applications. Softw. Pract. Exp. **40**(12), 1073–1104 (2010)
32. Verma, A., Butenko, S.: Network clustering via clique relaxations: a community based approach. In: 10th DIMACS Implementation Challenge (2011)
33. Wuchty, S., Almaas, E.: Peeling the yeast protein network. Proteomics **5**(2), 444–449 (2005)
34. Zhang, Y., Parthasarathy, S.: Extracting analyzing and visualizing triangle *k*-core motifs within networks. In: IEEE International Conference on Data Engineering (ICDE), pp. 1049–1060 (2012)