The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-06190-x

Simplified algorithms for order-based core maintenance

Bin Guo[1] · **Emil Sekerinski**[2]

Accepted: 3 May 2024

Abstract

Graph analytics attract much attention from both research and industry communities. Due to its linear time complexity, the *k*-core decomposition is widely used in many real-world applications such as biology, social networks, community detection, ecology, and information spreading. In many such applications, the data graphs continuously change over time. The changes correspond to edge insertion and removal. Instead of recomputing the *k*-core, which is time-consuming, we study how to maintain the *k*-core efficiently. That is, when inserting or deleting an edge, we need to identify the affected vertices by searching for more vertices. The state-of-the-art order-based method maintains an order, the so-called *k*-order, among all vertices, which can significantly reduce the searching space. However, this order-based method is complicated to understand and implement, and its correctness is not formally discussed. In this work, we propose a simplified order-based approach by introducing the classical Order Data Structure to maintain the *k*-order, which significantly improves the worst-case time complexity for both edge insertion and removal algorithms. Also, our simplified method is intuitive to understand and implement; it is easy to argue the correctness formally. Additionally, we discuss a simplified batch insertion approach. The experiments evaluate our simplified method over 12 real and synthetic graphs with billions of vertices. Compared with the existing method, our simplified approach achieves high speedups up to 7.7× and 9.7× for edge insertion and removal, respectively.

Keywords Graph · *k*-core maintenance · Order-based

✉ Bin Guo
binguo@trentu.ca

Emil Sekerinski
emil@mcmaster.ca

[1] Department of Computing & Information Systems, Trent University, 1600 West Bank Drive, Peterborough, ON K9L0G2, Canada

[2] Department of Computing and Software, McMaster University, Main St. W., Hamilton, ON L8S4L8, Canada

Published online: 28 May 2024

1 Introduction

Given an undirected graph $G = (V, E)$, the k-core decomposition is to identify the maximal subgraph G' in which each vertex has a degree of at least k; the core number of each vertex u is defined as the maximum value of k such that u is contained in the k-core of G [1, 2]. It is well-known that the core numbers can be computed with linear running time $O(|V| + |E|)$ [1]. Due to the linear time complexity, the k-core decomposition is easily and widely used in many real-world applications. In [2], Kong et al. summarize a large number of applications in biology, social networks, community detection, ecology, information spreading, etc. Especially in [3], Lesser et al. investigate the k-core robustness in ecological and financial networks.

In a survey [4], Malliaros et al. summarize the main research work related to k-core decomposition from 1968 to 2019. In static graphs, the computation of the core numbers has been extensively studied [1, 5–8]. However, in many real-world applications, such as determining the influence of individuals in spreading epidemics in dynamic complex networks [9] and tracking the actual spreading dynamics in dynamic social media networks [10], the data graphs continuously change over time. The changes correspond to the insertion and deletion of edges, which may have an impact on the core numbers of some vertices in the graph. Graphs of this kind are called *dynamic graphs*. When inserting or removing an edge, it is time-consuming to recalculate the core numbers of all vertices; a better approach is first to find the affected vertices and then to update their corresponding core numbers. The problem of maintaining the core numbers for dynamic graphs is called *core maintenance*. To the best of our knowledge, little work has been done on the k-core maintenance [11–14].

In this work, we focus on core maintenance. More formally, given an undirected dynamic graph $G = (V, E)$, after inserting an edge into or removing an edge from G, the problem is how to efficiently update the core number for the affected vertices. To do this, we first need to identify a set of vertices whose core numbers need to be updated (denoted as V^*) by traversing a possibly larger set of vertices (denoted as V^+). Then, it is easy to re-compute the new core numbers of vertices in V^*. In practice, an edge removal algorithm for core maintenance [11, 12] is easy to devise; but for edge insertion, it is challenging. Clearly, an efficient edge insertion algorithm should have a small cost for identifying V^*, which means a small ratio $|V^+|/|V^*|$. In this work, we mainly discuss the edge insertion algorithms for core maintenance.

In [11], Sariyüce et al. propose a *traversal* algorithm. This insertion algorithm searches V^* only in a local region near the edge that is inserted, which can be much faster than recomputing the core numbers for the whole graph. However, this insertion algorithm has a high variation in terms of performance due to the high variation of the ratio $|V^+|/|V^*|$. In [12], Zhang et al. propose an *order-based* algorithm, which is the state-of-the-art method for core maintenance. The main idea is that an *k-order* of all vertices is explicitly maintained. Here, the k-order is an order of vertices whose core number is determined by the core decomposition

algorithm like the BZ algorithm [1]. When a new edge (u, v) is inserted, the potentially affected vertices are checked with such k-order, by which numerous vertices are avoided to be checked. In this case, the size of V^+ is greatly reduced, so the ratio $|V^+|/|V^*|$ is typically much smaller and has less variation compared to the traversal algorithm. Thus, the computation time is significantly improved.

However, this order-based approach has two drawbacks. First, the order-based edge insertion algorithm is so complicated that it is not intuitive to understand easily. This complexity further brings difficulties to the correctness and implementation; in fact, the proof of correctness for the edge-insert algorithm is not formally discussed in [12]. Second, the k-order of the vertices in a graph is maintained by two specific data structures: (1) $\mathcal{A}$ (double linked lists combined with balanced binary search trees) for operations like inserting, deleting, comparing the order of two vertices, all of which requires worst-case $O(\log |V|)$ time; and (2) $\mathcal{B}$ (double linked lists combined with heaps) for searching the ordered vertices by jumping unnecessary ones, which requires worst-case $O(\log |V|)$ time; both data structures are complicated to implement.

In this work, we overcome the above drawbacks in [12] by proposing our simplified order-based approach. There are three main contributions summarized below:

- First, we introduce a well-known *Order Data Structure* [15, 16] to efficiently maintain the k-order of vertices in a graph G, and thus, our method has improved time complexities. Specifically, this Order Data Structure only requires amortized $O(1)$ time for order operations, including inserting, deleting, and comparing the order of two vertices; this is faster than the $\mathcal{A}$ data structure in [12], especially when data graphs are large. In addition, priority queues can be introduced to maintain each affected vertex in k order in the worst case $O(log|E^+|)$ time ($|E^+|$ is the number of edges adjacent to vertices in V^+); this is also faster than the $\mathcal{B}$ data structure in [12] since we normally have $|E^+| \ll |V|$ in real data graphs.
- Second, compared to the method in [12], when introducing Order Data Structures, the $\mathcal{A}$ and $\mathcal{B}$ data structures can be abandoned so that the order-based core maintenance approach can be significantly simplified. We also formally prove the correctness of our method.
- Third, our simplified order-based insertion algorithm can be easily extended to handle a batch of insertion edges without difficulties since it is common that a great number of edges are inserted or removed simultaneously; by doing this, the vertices in $V^+ \backslash V^*$ are possibly avoided to be repeatedly traversed so that the total size of V^+ is smaller compared to unit insertion.

The rest of this paper is organized as follows. Related work is discussed in Sect. 2. The preliminaries are given in Sect. 3. The original order-based algorithm is reviewed in Sect. 4. Our simplified order-based insertion and removal algorithms are proposed in Sect. 5. Our simplified order-based batch insertion is proposed in Sect. 6. We report on extensive performance studies in Sect. 7 and conclude in Sect. 8.

2 Related work

2.1 Core decomposition

In [17], Seidman first introduces k-core to analyze the density of social networks. In [5], Cheng et al. propose an external memory algorithm, so-called EMcore, which runs in a top-down manner such that the whole graph does not have to be loaded into memory. In [8], Wen et al. provide a semi-external algorithm, which requires $O(n)$ size memory to maintain the information of vertices. In [6], Khaouid et al. investigate the core decomposition in a single PC on large graphs using the `GraphChi` and `WebGraph` models. In [7], Montresoret et al. consider the core decomposition in a distributed system. In addition, parallel computation of core decomposition in multicore processors is first investigated in [18], where the ParK algorithm was proposed. Based on the main idea of ParK, a more scalable PKC algorithm has been reported in [19]. In [20], Chan et al. design distributed algorithms for approximate core decomposition.

2.2 Core maintenance

In [21], Zhang et al. prove that the core maintenance is asymmetric: the edge removal is bounded for $V^* = V^+$, but the edge insertion is unbounded for $V^* \subseteq V^+$. In other words, to identify V^*, the edge removal only needs to traverse V^*; however, the edge insertion can traverse a much larger set of vertices than V^*.

In [22], an algorithm that is similar to the traversal algorithm [11] is given, but this solution has quadratic time complexity. In [8], a semi-external algorithm for core maintenance is proposed in order to reduce the I/O cost, but this method is not optimized for CPU time. In [23, 24], parallel approaches for core maintenance are proposed for both edge insertion and removal. There exists some research based on core maintenance. In [25], the authors study computing all k-cores in the graph snapshot over the time window. In [26], Sun et al. design algorithms to maintain approximate core numbers in dynamic hypergraphs. In [27], Liu et al. propose a parallel and batch-dynamic algorithm for approximate k-core decomposition and maintenance.

Furthermore, in [28], the authors explore the hierarchical core maintenance. In [29], the core maintenance problem is explored in edge-weighted graphs using the order-based approach. In [30], a *spatial k-core* is proposed to model the rapidly growing amount of spatial data. Given a set of 2-dimensional data nodes, the spatial k-core is the maximal subset of nodes, where each node has at least k close neighbors; two nodes are close if their spatial distance is not larger than a given threshold.

3 Preliminaries

Let $G = (V, E)$ be an undirected unweighted graph, where $V(G)$ denotes the set of vertices and $E(G)$ represents the set of edges in G. When the context is clear, we will use V and E instead of $V(G)$ and $E(G)$ for simplicity, respectively. Note that, as G is an undirected graph, an edge $(u, v) \in E(G)$ is equivalent to $(v, u) \in E(G)$. We denote the number of vertices and edges of G by n and m, respectively. We define the set

of neighbors of a vertex $u \in V$ as $u.adj$, formally $u.adj = \{v \in V : (u, v) \in E\}$. We denote the degree of u in G as $u.deg = |u.adj|$. In addition, to analyze the time complexity, we denote the maximal degree among all vertices in G as $Deg(G) = \max\{v \in V(G) : v.deg\}$. We say that a graph G' is a subgraph of G, denoted as $G' \subseteq G$, if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Given a subset $V' \subseteq V$, the subgraph induced by V', denoted as $G(V')$, is defined as $G(V') = (V', E')$ where $E' = \{(u, v) \in E : u, v \in V'\}$.

Definition 1 (k-Core) Given an undirected graph $G = (V, E)$ and an integer k, a subgraph G_k of G is called a k-core if it satisfies the following conditions: (1) for $\forall u \in V(G_k)$, $u.deg \geq k$; (2) G_k is a maximal subgraph.

Here, we have $G_{k+1} \subseteq G_k$, for all $k \geq 0$, and G_0 is just G.

Definition 2 (Core Number) Given an undirected graph $G = (V, E)$, the core number of a vertex $u \in G(V)$, denoted as $u.core$, is defined as $u.core = max\{k : u \in V(G_k)\}$.

In other words, $u.core$ is the largest k such that there exists a k-core containing u.

Definition 3 (Subcore) Given an undirected graph $G = (V, E)$, a maximal set of vertices $S \subseteq V$ is called a k-subcore if and only if (1) $\forall u \in S, u.core = k$; (2) the induced subgraph $G(S)$ is connected. The subcore that contains vertex u is denoted as $\mathtt{sc}(u)$.

3.1 Core decomposition

Given a graph $G = (V, E)$, the problem of computing the core number for each $u \in V(G)$ is called core decomposition. In [1], Batagelj and Zaversnik propose an algorithm with a linear running time of $O(m + n)$, the so-called BZ algorithm. The general idea is the *peeling process*. That is, to compute the k-core G_k of G, the vertices (and their adjacent edges) whose degrees are less than k are repeatedly removed. When there are no more vertices to remove, the resulting graph is the k-core of G.

Algorithm 1 BZ algorithm for core decomposition

```
   input  : an undirected graph G = (V, E)
   output: the core number u.core for each u ∈ V
1  for u ∈ V do  u.d ← |u.adj|; u.core = ∅
2  Q ← a min-priority queue by u.d for all u ∈ V
3  while Q ≠ ∅ do
4  |  u ← Q.dequeue()
5  |  u.core ← u.d; remove u from G
6  |  for v ∈ u.adj do
7  |  |  if u.d < v.d then v.d ← v.d − 1
8  |  update Q
```

Algorithm 1 shows the steps of the BZ algorithm. In initialization, for each vertex $u \in V$, the auxiliary degree $u.d$ is set to $|u.adj|$ and the core number $u.core$ is not identified (line 1). The postcondition is that for each vertex $u \in V$, the $u.d$ equals the core number, formally $u.d = u.core$. We state informally lines 3 - 8 as a loop invariant: (1) the vertex u always has the minimum degree $u.d$ since u is removed from the min-priority queue Q (line 4); and (2) if u obtains its core number, $u.core$ equals to $u.d$ (line 5). The key step is updating $v.d$ for all $v \in u.adj$. That is, $v.d$ are decremented by 1 if $u.d$ is smaller than $v.d$ (lines 6 and 7). In this algorithm, the min-priority queue Q can be efficiently implemented by bucket sorting [1], by which the total running time is optimized to linear $O(m + n)$.

3.2 Core maintenance

The problem of maintaining the core numbers for dynamic graphs G is called core maintenance, when edges are inserted into and removed from G continuously. The insertion and removal of vertices can be simulated as a sequence of edge insertions and removals. Hence, in this paper, we focus on maintaining the core numbers when an edge is inserted into or removed from a graph G.

Definition 4 (Candidate Set V^* and Searching Set V^+) Given an undirected graph $G = (V, E)$, when an edge is inserted or removed, a candidate set of vertices, denoted as V^*, have to be computed so that the core numbers of all vertices in V^* must be updated. In order to identify V^*, a minimal set of searching vertices, denoted as V^+, is traversed by repeatedly accessing their adjacent edges.

Definition 4 says that V^* is identified by traversing all vertices in V^+, so that V^* has to belong to V^+, denoted as $V^* \subseteq V^+$. Further, the vertices in $V^+ \setminus V^*$ are traversed but not candidate vertices. Efficient core maintenance algorithms should have a small ratio of $|V^+|/|V^*|$ in order to minimize the cost of computing V^*. After V^* is identified, the core number of vertices in V^* can be updated accordingly.

Example 1 Consider the graph G in Fig. 1. The numbers inside the vertices are the core numbers. Three vertices, v_1 to v_3, have same core numbers of 2; the other vertices, u_1 to u_{1000}, have same core numbers of 1. The whole graph G is the 1-core since each vertex has a degree of at least 1; the subgraph induced by $\{v_1, v_2, v_3\}$ is the 2-core since each vertex in this subgraph has a degree of at least 2. After inserting an edge, for example, (u_1, u_{500}), we observe that the core numbers of all vertices do not change according to the peeling process. In this case, the candidate set $V^* = \emptyset$. However, the searching set V^+ is different for different edge insertion methods, e.g., the order-based algorithm may have $V^+ = \{u_1, u_2, u_3\}$ and the traversal algorithm traverse all vertices in $\mathtt{sc}(u_1)$ with $V^+ = \{u_1, u_2, \ldots, u_{1000}\}$.

We present two theorems given in [11, 12, 22] that are useful for discussing the correctness of our insertion and removal algorithms.

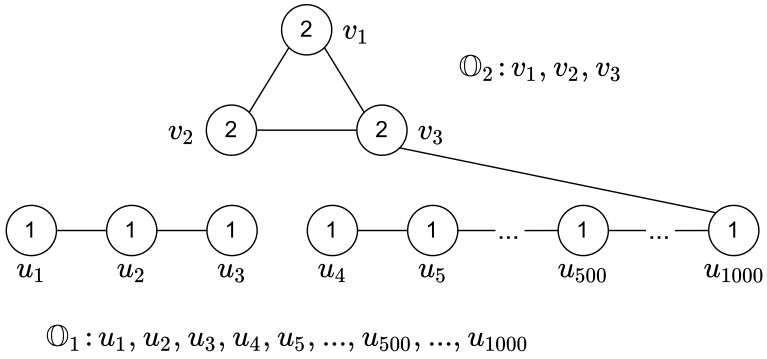


Fig. 1 A sample graph G with $\mathbb{O} = \mathbb{O}_1\mathbb{O}_2$ in k-order

Theorem 1 [11, 12, 22] *After inserting an edge into or removing an edge from* $G = (V, E)$, *the core number of a vertex* $u \in V^*$ *increases or decreases by at most* 1, *respectively*.

Theorem 2 [11, 12, 22] *Suppose an edge* (u, v) *with* $K = u.core \leq v.core$ *is inserted in* (*resp. removed from*) G. *Suppose* V^* *is nonempty. We have the following*: (1) *if* $u.core < v.core$, *then* $u \in V^*$ *and* $V^* \subseteq \mathtt{sc}(u)$ (*as in Definition* 3); (2) *if* $u.core = v.core$, *then both vertices* u *and* v *are in* V^* (*resp. at least one of* u *and* v *is in* V^*) *and* $V^* \subseteq \mathtt{sc}(u) \cup \mathtt{sc}(v)$; (3) *the induced subgraph of* $V^* \in G \cup \{(u, v)\}$ *is connected*.

Theorem 2 suggests that: (1) V^* only includes the vertices $u \in V$ with $u.core = K$; (2) V^* can be searched in a small local region near the inserted or removed edge rather than in a whole graph. That is, to identify V^*, all vertices in V^+ are located in the subcores containing u and v.

3.3 Order Data Structure

The well-known *Order Data Structure* [15, 16] maintains a total order of items, denoted as $\mathbb{O}$, with $O(1)$ running time and space. It includes three operations:

- $\mathtt{ORDER}(\mathbb{O}, x, y)$: determine if x precedes y in the total order $\mathbb{O}$.
- $\mathtt{INSERT}(\mathbb{O}, x, y)$: insert a new item y after x in the total order $\mathbb{O}$.
- $\mathtt{DELETE}(\mathbb{O}, x)$: remove an item x from the total order $\mathbb{O}$.

The main idea is that each item x in the total order $\mathbb{O}$ is assigned a label to indicate the order. In this way, an ORDER operation only requires $O(1)$ time for label comparisons, and a DELETE operation only requires $O(1)$ time for directly removing one item

without affecting the labels of other items. Significantly, an INSERT($\mathbb{O}, x, y$) is complicated: 1) if there exists a valid label between two items x and x's successor, the new item y can be inserted between them by assigning a new label, which requires $O(1)$ time; 2) or else, a *relabel operation* is triggered to rebalance the labels for adjacent items, which requires $O(1)$ amortized running time.

In this work, our simplified order-based core maintenance algorithms are based on this Order Data Structure. Our time complexity analysis is based on the $O(1)$ time for the above three order operations.

4 The order-based algorithm

In this section, we discuss the state-of-the-art order-based core maintenance approach in [12]. This algorithm is based on the k-order, which can be generated by the BZ algorithm for core decomposition [1] as in Algorithm 1. The k-order is defined as follows.

Definition 5 (k-Order $\preceq$ [12]) Given a graph G, the k-order $\preceq$ is defined for any pair of vertices u and v over the graph G as follows: (1) when $u.core < v.core$, $u \preceq v$; (2) when $u.core = v.core$, $u \preceq v$ if u's core number is determined before v's by BZ algorithm (Algorithm 1, line 5).

A k-order $\preceq$ is an instance of all the possible vertex sequences produced by Algorithm 1. When generating the k-order, there may be multiple vertices $v \in Q$ that have the same value of $u.d$ and can be popped out from Q at the same time together (Algorithm 1, line 4). When dealing with these vertices with the same value of d, different sequences generate different instances of correct k-order for all vertices. There are three heuristic strategies, "small degree first," "large degree first," and "random." The experiments in [12] show that the "small degree first" consistently has the best performance over all tested real graphs.

For the k-order, transitivity holds, that is, $u \preceq v$ if $u \preceq w \wedge w \preceq v$. For each edge insertion and removal, the k-order will be maintained. Here, $\mathbb{O}_k$ denotes the sequence of vertices in k-order whose core numbers are k. A sequence $\mathbb{O} = \mathbb{O}_0\mathbb{O}_1\mathbb{O}_2 \cdots$ over $V(G)$ can be obtained, where $\mathbb{O}_i \preceq \mathbb{O}_j$ if $i < j$. It is clear that $\preceq$ is defined over the sequence of $\mathbb{O} = \mathbb{O}_0\mathbb{O}_1\mathbb{O}_2 \cdots$. In other words, for all vertices in the graph, the sequence $\mathbb{O}$ indicates the k-order $\preceq$.

Example 2 Continually consider the graph G in Fig. 1. The numbers inside the vertices are the core numbers. The k-order of G is shown by $\mathbb{O}_1$ and $\mathbb{O}_2$, which is the order of core numbers determined by the BZ algorithm (Algorithm 1 line 5); also, $\mathbb{O}_1$ is determined before $\mathbb{O}_2$ so that we have $\mathbb{O}_1 \preceq \mathbb{O}_2$.

4.1 The order-based insertion

The key step for the insertion algorithm is to determine V^*. To do this, two degrees, $u.d^+$ and $u.d^*$, for each vertex $u \in V(G)$ are maintained in order to identify whether u can be added into V^* or not:

- Remaining degree $u.d^+$: the number of neighbors after the vertex u in $\mathbb{O}$ that can potentially support the increment of the current core number, defined as

$$u.d^+ = |\{w \in u.adj \mid u \preceq w \wedge w \notin V^+ \setminus V^*\}|$$

- Candidate degree $u.d^*$: the number of neighbors before the vertex u in $\mathbb{O}$ that can potentially have their core number increased, defined as

$$u.d^* = |\{w \in u.adj \mid w \preceq u \wedge w \in V^*\}|$$

Assume that an edge (u, v) is inserted with $K = u.core \leq v.core$. The intuition behind the order-based insertion algorithm is as follows. Starting from u, all affected vertices with the same core number K (Theorem 2) are traversed in $\mathbb{O}$. For each visited vertex $w \in V^+$, the value of $w.d^* + w.d^+$ is maximal as w is visited by k-order. In this case, w will be added to V^* if $w.d^* + w.d^+ > K$; otherwise, w is impossible in V^*, which may repeatedly cause other vertices to be removed from V^*. When all vertices with core number K are traversed, this process terminates and V^* is identified. Finally, the core numbers for all the vertices in V^* are updated by increasing by 1 (Theorem 1). Obviously, for all vertices $u \in V$, the order $\mathbb{O}$ along with $u.d^+$ and $u.d^*$ must be maintained accordingly.

Compared to the Traversal Insertion Algorithm [11], the benefit of traversing with k-order is that a large number of unnecessary vertices in $V^+ \setminus V^*$ can be avoided. This is why the order-based insertion algorithm is generally more efficient.

The order-based insertion algorithm is not easy to implement as it needs to traverse the vertices in $\mathbb{O}$ efficiently. There are three cases. First, given a pair of vertices $u, v \in \mathbb{O}_k$, the order-based insertion algorithm needs to test whether $u \preceq v$ or not efficiently. For this, $\mathbb{O}_k$ is implemented as a double-linked list associated with a data structure $\mathcal{A}_k$, which is a binary search tree and each tree node holds one vertex. For all $u, v \in \mathbb{O}_k$, we can test the order $u \preceq v$ in $O(\log |\mathbb{O}_k|)$ time by using $\mathcal{A}_k$. Second, the order-based insertion algorithm needs to efficiently "jump" over a large number of non-affected vertices that have $u.d^* = 0$. To do this, $\mathbb{O}_k$ is also associated with a data structure $\mathcal{B}$, which is a min-heap. Here, $\mathcal{B}$ supports finding a affected vertex u with $u.d^* > 0$ sequentially in $\mathbb{O}_k$ with $O(1)$ time; but it requires $O(\log |\mathbb{O}_k|)$ time to maintain the min-heap. Therefore, when maintaining $\mathbb{O}$, both $\mathcal{A}$ and $\mathcal{B}$ require to update accordingly, which requires the worst-case $O(|V^+| \cdot \log |\mathbb{O}_k| + O(|V^*|) \log |\mathbb{O}_{k+1}|)$ time for removing $v \in V^*$ from $\mathbb{O}_k$ and then inserting $v \in V^*$ at the head of $\mathbb{O}_{k+1}$.

As we can see, the $\mathcal{A}$ and $\mathcal{B}$ data structures are complicated, which complicates understanding and implementation. Additionally, the operations on $\mathcal{A}$ and $\mathcal{B}$ are time-consuming, especially when handling a data graph with large sizes of $\mathbb{O}_k$ or $\mathbb{O}_{k+1}$.

4.2 The order-based removal

The order-based removal algorithm adopts the same routine used in the traversal removal algorithm [11] to compute V^*. This order-based removal algorithm is based on the *max-core degree*.

Definition 6 (max-core degree *mcd*) [11, 12] Given a graph $G = (V, E)$, for each vertex $v \in V$, the max-core degree, $v.mcd$, is the number of v's neighbors w such that $w.core \geq v.core$, defined as $v.mcd = |\{w \in v.adj : w.core \geq v.core\}|$.

As discussed, edge removal is much simpler than edge insertion since edge removal is bounded for $V^* = V^+$. Assuming that an edge (u, v) is removed from the graph, both $u.mcd$ and $v.mcd$ are updated accordingly. This may repeatedly affect other adjacent vertices' *mcd*. When the process terminates, all affected vertices u that have $u.mcd < u.core$ can be added to V^*, and then their core numbers are off by 1. Obviously, for all vertices $u \in V^*$, the sequence $\mathbb{O}$ along with $u.mcd$ must be maintained accordingly.

Compared with the traversal removal algorithm, the difference is that the order-based removal algorithm needs to maintain $\mathbb{O}$ for all vertices in V^*. That is, all vertices in V^* with the core number k are deleted from $\mathbb{O}_k$ and then appended to $\mathbb{O}_{k-1}$ in the corresponding k-order. Recall that two associated data structures, $\mathcal{A}$ and $\mathcal{B}$, are used for the order-based insertion algorithm. Both $\mathcal{A}$ and $\mathcal{B}$ must be updated accordingly, which requires worst-case $O(|V^*| \cdot (\log |O_k| + \log |O_{k-1}|))$ time for removing $v \in V^*$ from $\mathbb{O}_k$ and appending $v \in V^*$ at the tail of $\mathbb{O}_{k-1}$. Analogously to the order-based insertion, the operations on $\mathcal{A}$ and $\mathcal{B}$ are time-consuming when handling a data graph with a large size of $\mathbb{O}_k$ or $\mathbb{O}_{k-1}$.

5 The simplified order-based algorithm

The main reason for the order-based algorithm being complicated and inefficient is that two data structures, $\mathcal{A}$ and $\mathcal{B}$, are used to maintain $\mathbb{O}$ in k-order for all vertices in a graph. In this section, we adopt the Order Data Structure [15, 16] to maintain the k-order for all vertices. There are two benefits: one is that the k-order operations, such as inserting, deleting, and comparing the order of two vertices, can be optimized to $O(1)$ amortized running time; the other is that the original order-based method [12] can be simplified, which makes it easier to implement and to discuss the correctness.

Before introducing the new method, we propose a *constructed directed acyclic graph (DAG)* to simplify the statement of our algorithms. Given an undirected graph $G = (V, E)$ with $\mathbb{O}$ in k-order, each edge $(u, v) \in E(G)$ can be assigned a direction such that $u \preceq v$. By doing this, a *directed acyclic graph* (DAG) $\vec{G} = (V, \vec{E})$ can be constructed where each edge $u \mapsto v \in \vec{E}(\vec{G})$ satisfies $u \preceq v$. Of course, the k-order of G is the *topological order* of $\vec{G}$. The *post* of a vertex v in $\vec{G}(V, \vec{E})$ is all its successors (outgoing edges), defined by $u(\vec{G}).post = \{v \mid u \mapsto v \in \vec{E}(\vec{G})\}$; the

pre of a vertex v in $\vec{G}(V, \vec{E})$ is all its predecessors (incoming edges), defined by $u(\vec{G}).pre = \{v \mid v \mapsto u \in \vec{E}(\vec{G})\}$. When the context is clear, we use $u.post$ instead of $u(\vec{G}).post$ and $u.pre$ instead of $u(\vec{G}).pre$.

In other words, the constructed DAG $\vec{G} = (V, \vec{E})$ is equivalent to the undirected graph $G(V, E)$ by associating the direction for each edge in k-order. This newly defined constructed DAG $\vec{G}$ is convenient for describing our simplified order-based insertion algorithm.

Lemma 1 *Given a constructed DAG $\vec{G} = (V, \vec{E})$, for each vertex $v \in V$, the out-degree $|v.post|$ is not greater than the core number, $|v.post| \leq v.core$.*

Proof Since the topological order of $\vec{G}$ is the k-core of G, when removing the vertex v by executing the BZ algorithm (Algorithm 1, line 5) all the vertices in $v.pre$ are already removed. In such a case, the out-degree of v is its current degree. If there exists $|v.post| > v.core$, the value $v.core$ should be equal to $|v.post|$, which leads to a contradiction. □

If inserting an edge into a constructed DAG $\vec{G}$ does not violate Lemma 1, no maintenance operations are required. Otherwise, $\vec{G}$ has to be maintained to re-establish Lemma 1.

Table 1 summarizes the notations frequently used when describing the algorithm.

5.1 The simplified order-based insertion

5.2 Theory background

With the concept of the constructed DAG $\vec{G}$, we can introduce our simplified insertion algorithm to maintain the core numbers after an edge is inserted into $\vec{G}$. For convenience, based on the constructed DAG $\vec{G}$, we first redefine the candidate degree and the remaining degree as in [12].

Definition 7 (Remaining Out-Degree) Given a constructed DAG $\vec{G}(V, \vec{E})$, the remaining out-degree $v.d^+_{out}$ is the total number of its successors w without the ones that are confirmed not in V^* due to $w.d^*_{in} + w.d^+_{out} \leq K$, denoted as

$$v.d^+_{out} = |\{w \in v.post \,:\, w \notin V^+ \setminus V^*\}|$$

Definition 8 (Candidate In-Degree) Given a constructed DAG $\vec{G}(V, \vec{E})$, the candidate in-degree $v.d^*_{in}$ is the total number of its predecessors located in V^*, denoted as

$$v.d^*_{in} = |\{w \in v.pre \,:\, w \in V^*\}|$$

In other words, assuming that $K = v.core$, the candidate in-degree $v.d^*_{in}$ counts the number of predecessors that are already in the new $(K+1)$-core; $v.d^+_{out}$ counts the

Table 1 Notations

Notation	Description
$G = (V, E)$	An undirected graph
$\vec{G} = (V, \vec{E})$	An constructed DAG by the k-order
$u \mapsto v \in \vec{E}(\vec{G})$	A directed edge in an constructed DAG
$\mathbb{O} = \mathbb{O}_0 \mathbb{O}_1 \dots \mathbb{O}_k$	A sequence indicates the k-order $\preceq$
$u(\vec{G}).d^*_{in}$	The remaining in-degree of u
$u(\vec{G}).d^+_{out}$	The candidate out-degree of u
$u(\vec{G}).post$	The successors of u in $\vec{G}$
$u(\vec{G}).pre$	The predecessor of u in $\vec{G}$
$u.mcd$	The max-core degree of u
$u.core$	The core number of u
V^*	Candidate set
V^+	Searching set
$\Delta\vec{G} = (V, \Delta\vec{E})$	An inserted graph

number of successors that can be in the new $(K + 1)$-core. Therefore, $v.d^*_{in} + v.d^+_{out}$ upper bounds the number of v's neighbors in the new $(K + 1)$-core.

Theorem 3 *Given a constructed DAG $\vec{G} = (V, \vec{E})$ by inserting an edge $u \mapsto v$ with $K = u.core \leq v.core$, the candidate set V^* includes all possible vertices that satisfy*: 1) *their core numbers are equal to K, and* 2) *their total numbers of candidate in-degree and remaining out-degree are greater than K, denoted as*

$$\forall w \in V : w \in V^* \equiv (w.core = K \ \wedge \ w.d^*_{in} + w.d^+_{out} > K)$$

Proof According to Theorem 1 and Theorem 2, for all vertices in V^*, we have 1) their core numbers equal to K, and 2) their core numbers will increase to $K + 1$ and they can be added to the new $(K + 1)$-core. By the definition of k-core, for a vertex $v \in V^*$, v must have at least $K + 1$ adjacent vertices that can be in the new $(K + 1)$ -core. As $v.d^*_{in} + v.d^+_{out}$ is the number of v's adjacent vertices that can be in the new $(K + 1)$-core, we get $v.d^*_{in} + v.d^+_{out} > K$ for all vertices $v \in V^*$. □

Theorem 4 *Given a constructed DAG $\vec{G} = (V, \vec{E})$ by inserting an edge $u \mapsto v$ with u in $\mathbb{O}_K$, all affected vertices w are after u in $\mathbb{O}_K$. Starting from u, when w is traversed in $\mathbb{O}_K$ and the V^+, V^*, $w.d^*_{in}$, $w.d^+_{out}$ are updated accordingly, each time the value of $w.d^*_{in} + w.d^+_{out}$ is maximal.*

Proof For all the vertices in the constructed DAG $\vec{G}$, $\mathbb{O}$ is the topological order in $\vec{G}$ according to the definition of $\vec{G}$. When traversing affected vertices w in G in such topological order, each time for w all affected predecessors must have been traversed, so that we get the value of $w.d^*_{in}$ is maximal; also, all the related successors

are not yet traversed, so that the value of $w.d^+_{out}$ is also maximal. Therefore, the total value of $w.d^*_{in} + w.d^+_{out}$ is maximal. □

In other words, when traversing the affected vertices w in $\mathbb{O}$, $w.d^*_{in} + w.d^+_{out}$ is the upper-bound. That means, when traversing the vertices after w in $\mathbb{O}$, $w.d^*_{in} + w.d^+_{out}$ only can be decreased as some vertices can be removed from V^*. In this case, we can safely remove w from V^* if $w.d^*_{in} + w.d^+_{out} \leq K$, since w is impossible in V^* according to Theorem 3. This is the key idea behind the order-based insertion algorithm.

5.3 The algorithm

Algorithm 2 shows the detailed steps when inserting an edge $u \mapsto v$. An issue is the implementation of traversing the vertices in $\mathbb{O}_k$. We propose to use a Min-Priority Queue combined with the Order Data Structure (line 4). The idea is as follows: 1) $\mathbb{O}_k$ is maintained by the Order Data Structure [15, 16], by which each vertex is assigned a label (an integer number) to indicate the order, and 2) all adjacent vertices are added into a Min-Priority Queue by using such labels as their keys. By doing this, we can dequeue a vertex from the Min-Priority Queue each time to "jump" over unaffected vertices efficiently. In addition, three colors are used to indicate the different statuses for each vertex v in a graph:

- `white`: v has initial status, $v \notin V^* \wedge v \notin V^+$.
- `black`: v is traversed and identified as a candidate vertex, $v \in V^* \wedge v \in V^+$.
- `gray`: v is traversed and identified as impossible to be a candidate vertex, $v \notin V^* \wedge v \in V^+ \equiv v \in V^+ \setminus V^*$.

Before executing, we assume that for all vertices $v \in V(\vec{G})$ their d^+_{out} and d^*_{in} are correctly maintained, that is $v.d^+_{out} = |v.post| \ \wedge \ v.d^*_{in} = 0$. Initially, both V^* and V^+ are empty (all vertices are `white`) and K is initialized to $u.core$ since $u \preceq v$ for $u \mapsto v$ (line 1). After inserting an edge $u \mapsto v$ with $u \preceq v$ in $\mathbb{O}$, we have $u.d^+_{out}$ increase by one (line 2). The algorithm will terminate if $u.d^+_{out} \leq u.core$ as Lemma 1 is satisfied (line 3). Otherwise, u is added into the Min-Priority Queue Q (line 4) for propagation (line 5 to 8). For each w removed from Q (line 6), we check the value of $w.d^*_{in} + w.d^+_{out}$. That is, if $w.d^*_{in} + w.d^+_{out} > K$, vertex w can be added to V^* and may cause other vertices added in V^*, which is processed by the `Forward` procedure (line 7). Otherwise, w cannot be added to V^*, which may cause some vertices to be removed from V^* processed by the `Backward` procedure (line 8). Here, $w.d^*_{in} > 0$ means that w is affected, or else w can be omitted since w has no predecessors in V^* (line 8). When Q is empty, this process terminates, and V^* is obtained (line 5). At the ending phase, for all vertices V^*, their core numbers are increased by one (by Theorem 1), and their d^*_{in} are reset (line 9). Finally, the $\mathbb{O}$ is maintained (line 10).

Algorithm 2 EdgeInsert($\vec{G}, \mathbb{O}, u \mapsto v$)

input : A DAG $\vec{G}(V, \vec{E})$; the corresponding $\mathbb{O}$; an edge $u \mapsto v$ to be inserted.
output: An updated DAG $\vec{G}(V, \vec{E})$; the updated $\mathbb{O}$.

```
1  V*, V+, K ← ∅, ∅, u.core                          // all vertices are white
2  insert u ↦ v into G with u.d+out ← u.d+out + 1
3  if u.d+out ≤ K then return
4  Q ← a min-priority queue by 𝕆; Q.enqueue(u)
5  while Q ≠ ∅ do
6  |  w ← Q.dequeue()
7  |  if w.d*in + w.d+out > K then Forward(w, Q, K, V*, V+)
8  |  else if w.d*in > 0 then Backward(w, 𝕆, K, V*, V+)
   // Ending Phase
9  for w ∈ V* do w.core ← K + 1; w.d*in ← 0
10 for w ∈ V* do remove w from 𝕆_K and insert w at the beginning of 𝕆_{K+1} in
     k-order (the order w added into V*)
```

The detailed steps of the `Forward` procedure are shown in Algorithm 3. At first, u is added to V^* and V^+ (set from `white` to `black`) since u has $u.d^*_{in} + u.d^+_{out} > K$ (line 1). Then, for each u's successor v whose core number equals K (by Theorem 2), $v.d^*_{in}$ increases by one (lines 2 and 3). In this case, v is affected and must be added to Q for subsequent propagation (line 4). Note that a vertex v can only be added to Q since only the successors of v are enqueued.

Algorithm 3 Forward(u, Q, K, V^*, V^+)

```
1 V* ← V* ∪ {u}; V+ ← V+ ∪ {u}                      // u is white to black
2 for v ∈ u.post : v.core = K do
3 |  v.d*in ← v.d*in + 1
4 |  if v ∉ Q then Q.enqueue(v)
```

The detail steps of the `Backward` procedure are shown in Algorithm 4. In the `DoPre`(u) procedure, for all u's predecessors v that are located in V^* (line 11), $v.d^+_{out}$ is decreased by one since u is set to `gray` and cannot be added into V^* any more (line 12); in this case, v has to be added into R for propagation if $v.d^*_{in} + v.d^+_{out} \leq K$ (line 13). Similarly, in the `DoPost`(u) procedure, for all u's successors v that have $v.d^*_{in} > 0$ (line 15), $v.d^*_{in}$ is decreased by one (line 16) and added into R for propagation if $v.d^*_{in} + v.d^+_{out} \leq K$ (lines 17 and 18).

The detailed steps of the `Backward` procedure are shown in Algorithm 4. The queue R is used for propagation (line 2). The `DoPre`(u) procedure updates the graph when setting u from `white` to `gray` or from `black` to `gray`, that is, for all u's predecessors in V^*, all d^+_{out} are off by 1 and then added to R for propagation, if its $d^*_{in} + d^+_{out} \leq K$ since they cannot be in V^* any more (lines 10 - 13). Similarly, the `DoPost` procedure updates the graph when setting u from `black` to `gray`, that is, for all u's successors with $d^*_{in} > 0$, all d^*_{in} are off by 1 and then added to R for

propagation if it is in V^* and its $d^*_{in} + d^+_{out} \leq K$ (lines 14 - 18). Now, we explain the algorithm step by step. At first, w is just added to V^+ (set from `white` to `gray`) since w has $w.d^*_{in} + w.d^+_{out} \leq K$ (line 1). The queue R is initialized as empty for propagation (line 2) and w is propagated by the `DoPre` procedure. Of course, w's d^+_{out} and d^*_{in} are updated (line 3) since all `black` vertices causing $w.d^*_{in}$ increased will be moved after w in $\mathbb{O}$ eventually. All the vertices in R are `black` waiting to be propagated (lines 4 to 9). For each $u \in R$, vertex u is removed from R (line 5) and removed from V^*, which sets u from `black` to `gray` (line 6). This may require d^*_{in} and d^+_{out} of adjacent vertices to be updated, which is done by the procedures `DoPre` and `DoPost`, respectively (line 7). To maintain $\mathbb{O}_K$, u is first removed from $\mathbb{O}_K$ and then inserted after p in $\mathbb{O}_K$, where p initially is w or the previous moved vertices in $\mathbb{O}_K$ (line 8). Of course, u's d^+_{out} and d^*_{in} are updated (line 3) since all `black` vertices causing $u.d^*_{in}$ increased will be moved after w in $\mathbb{O}$ eventually. This process is repeated until R is empty (lines 4 to 9).

Algorithm 4 Backward($w, \mathbb{O}, K, V^*, V^+$)

```
1  V+ ← V+ ∪ {w}; p ← w                                   // w is white to gray
2  R ← an empty queue; DoPre(w)
3  w.d+out ← w.d+out + w.d*in; w.d*in ← 0
4  while R ≠ ∅ do
5  |  u ← R.dequeue()
6  |  V* ← V* \ {u}                                      // u is black to gray
7  |  DoPre(u); DoPost(u)
8  |  DELETE (O_K, u); INSERT(O_K, p, u); p ← u
9  |  u.d+out ← u.d+out + u.d*in; u.d*in ← 0
10 procedure DoPre(u)
11 |  for v ∈ u.pre : v ∈ V* do
12 |  |  v.d+out ← v.d+out − 1
13 |  |  if v.d*in + v.d+out ≤ K ∧ v ∉ R then R.enqueue(v)
14 procedure DoPost(u)
15 |  for v ∈ u.post : v.d*in > 0 do
16 |  |  v.d*in ← v.d*in − 1
17 |  |  if v ∈ V* ∧ v.d*in + v.d+out ≤ K ∧ v ∉ R then
18 |  |  |  R.enqueue(v)
```

Example 3 Consider inserting an edge to a constructed graph in Fig. 2 obtained from Fig. 1. The numbers inside the vertices are the core numbers, and the two numbers beside the vertices u_1, u_2, u_3 and u_{500} are their $d^*_{in} + d^+_{out}$. Initially, we have the min-priority queue $Q = \emptyset$ and $K = 1$. In Fig. 2(a), after inserting an edge $u_1 \mapsto u_{500}$, we get $u_1.d^+_{out} = 2 > K$ and therefore u_1 is added to Q as $Q = \{u_1\}$. We begin to propagate Q. First, in Fig. 2(a), u_1 is removed from Q to do the `Forward` procedure since $u_1.d^+_{out} + u_1.d^*_{in} = 0 + 2 > K$, by which u_1 is colored by `black`, all $u_1.post$'s d^*_{in} add

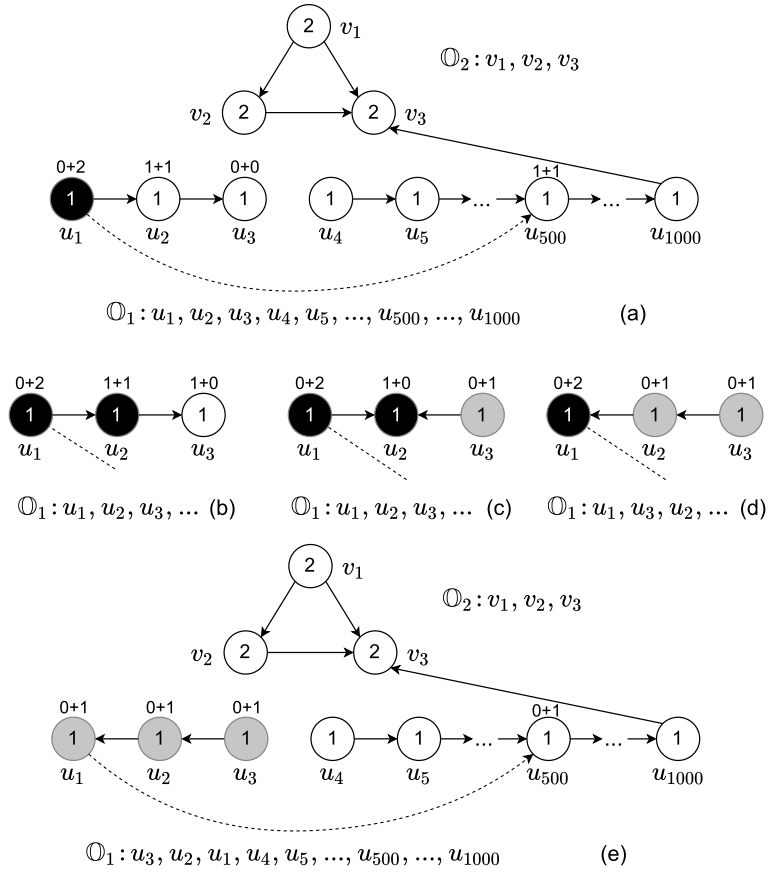


Fig. 2 Insert one edge $u_1 \mapsto u_{500}$ to a constructed graph $\vec{G}$ obtained from Fig. 1

by 1, and all $u_1.post$ are put into Q as $Q = \{u_2, u_{500}\}$. Second, in Fig. 2(b), u_2 is removed from Q to do the `Forward` procedure since $u_2.d^+_{out} + u_2.d^*_{in} = 1 + 1 > K$, by which u_2 is colored by `black`, all $u_2.post$'s d^*_{in} add by 1, and all $u_1.post$ are added into Q as $Q = \{u_3, u_{500}\}$. Third, in Fig. 2(c), however, u_3 is removed from Q to do the `Backward` procedure since $u_3.d^*_{in} + u_3.d^+_{out} = 1 + 0 \leq K$, by which u_3 is colored by `gray` and we have $Q = \{u_{500}\}$.

The `Backward` procedure continues. In Fig. 2(d), we get $u_2.d^+_{out}$ off by 1 and $u_2.d^*_{in} + u_2.d^+_{out} = 1 + 0 \leq K$, so that u_2 is set to `gray`, by which u_2 is moved after u_3 in $\mathbb{O}_1$. In Fig. 2(e), we get $u_1.d^+_{out}$ off by 1 and $u_1.d^*_{in} + u_1.d^+_{out} = 0 + 1 \leq K$, so that u_1 is also set to `gray`, by which u_1 is moved after u_3 in $\mathbb{O}_1$; also, we get $u_{500}.d^*_{in}$ off by 1 and the `Backward` procedure terminate. Finally, we still need to check the last u_{500} in Q, which can be safely omitted since its d^*_{in} is 0. In this simple example, we have $V^* = \emptyset \wedge V^+ = \{u_1, u_2, u_3\}$ and only 4 vertices added to Q. A large number of vertices in $\mathbb{O}_1$, e.g., $u_4 \ldots u_{1000}$, are avoided to be traversed.

5.4 Correctness

The key issue of Algorithm 2 is to identify the candidate set V^*. For correctness, the algorithm has to be sound and complete. The soundness implies that all the vertices in V^* are correctly identified,

$$sound(V^*) \equiv \forall v \in V : v \in V^* \Rightarrow v.d^*_{in} + v.d^+_{out} > K \wedge v.core = K$$

The completeness implies that all possible candidate vertices are added into V^*,

$$complete(V^*) \equiv \forall v \in V : v.d^*_{in} + v.d^+_{out} > K \wedge v.core = K \Rightarrow v \in V^*$$

The algorithm has to ensure both soundness and completeness

$$sound(V^*) \wedge complete(V^*),$$

which is equivalent to

$$\forall v \in V : v \in V^* \equiv v.d^*_{in} + v.d^+_{out} > K \wedge v.core = K$$

To argue the soundness and completeness, we first define the vertices in $V(G)$ to have the correct candidate in-degrees and remaining out-degrees as

$$in^*(V) \equiv \forall v \in V : v.d^*_{in} = |\{w \in v.pre : w \in V^*\}|$$
$$out^+(V) \equiv \forall v \in V : v.d^+_{out} = |\{w \in v.post : w \notin V^+ \setminus V^*\}|$$

We also define the sequence $\mathbb{O}$ for all vertices in V are in k-order as

$$\forall v_i \in V : \mathbb{O}(V) = (v_1, v_2, \ldots, v_i) \Rightarrow v_1 \preceq v_2 \preceq \cdots \preceq v_i$$

Theorem 5 (soundness and completeness) *For any constructed graph* $\vec{G}(V, \vec{E})$, *the while-loop in Algorithm* 2 (5 *to* 8) *terminates with* $sound(V^*)$ *and* $complete(V^*)$.

Proof The invariant of the outer while-loop (lines 5 to 8 in Algorithm 2) is that all vertices in V^* are sound, but adding a vertex to V^* (`white` to `black`) may lead to its successors to be incomplete; for all vertices, their d^*_{in} and d^+_{out} counts are correctly maintained. All vertices in Q have their core numbers as K, and their d^*_{in} must be greater or equal to 0, and all vertices $v \in V$ must be greater than 0 if v is located in Q; also, the k-order for all the vertices not in V^* is correctly maintained:

$$\begin{aligned} & sound(V^*) \wedge complete(V^* \setminus Q) \wedge in^*(V) \wedge out^+(V) \\ & \wedge (\forall v \in Q : v.core = K \wedge v.d^*_{in} \geq 0) \\ & \wedge (\forall v \in V : v.d^*_{in} > 0 \Rightarrow v \in Q) \\ & \wedge \mathbb{O}(V \setminus V^*) \end{aligned}$$

The invariant initially holds as $V^* = \emptyset$ and for all vertices their d^*_{in}, d^+_{out} and k-order are correctly initialized; also u is first add to Q for propagation only when

$u.core = K \wedge u.d^+_{out} > K \wedge u.d^*_{in} = 0$. We now argue that the while-loop preserves this invariant:

- *sound*(V^*) is preserved as $v \in V$ is added to V^* only if $v.d^*_{in} + v.d^+_{out} > K$ by the `Forward` procedure; also, v is safely removed from V^* if $v.d^*_{in} + v.d^+_{out} \leq K$ by the `Backward` procedure according to Theorem 4.
- *complete*($V^* \setminus Q$) is preserved as all the affected vertices v, which may have $v.d^*_{in} + v.d^+_{out} > K$, are added to Q by the `Forward` procedure for propagation.
- $in^*(V)$ is preserved as each time when a vertex v is added to V^*, all its successors' d^*_{in} are increased by 1 in the `Forward` procedure; also each time when a vertex v cannot be added to V^*, the $\mathbb{O}$ may change `Backward` procedure.
- $out^*(V)$ is preserved as each time when a vertex v cannot be added to V^*, the $\mathbb{O}$ may change, and the corresponding d^+_{out} are correctly maintained by the `Backward` procedure.
- $(\forall v \in Q : v.core = K \wedge v.d^*_{in} \geq 0)$ is preserved as in the `Forward` procedure, the vertices v are added in Q only if $v.core = K$ with $v.d^*_{in}$ add by 1; but $v.d^*_{in}$ may be reduced to 0 in the `Backward` procedure when some vertices cannot in V^*.
- $(\forall v \in V : v.d^*_{in} > 0 \Rightarrow v \in Q)$ is preserved as v can be added in Q only after adding $v.d^*_{in}$ by 1 in the `Forward` procedure.
- $\mathbb{O}(V \setminus V^*)$ is preserved as the k-order of all vertices $v \in V^+ \setminus V^*$ is correctly maintained by the `Backward` procedure and the k-order of all the other vertices $v \in V \setminus V^+$ is not affected.

We also have to argue the invariant of the inner while-loop in the `Backward` procedure (lines 4 to 9 in Algorithm 4). The additional invariant is that all vertices in R have to be located in V^* but not sound as their $d^*_{in} + d^+_{out} \leq K$:

$$
\begin{aligned}
& sound(V^* \setminus R) \ \wedge \ complete(V^* \setminus Q) \ \wedge \ in^*(V) \ \wedge \ out^+(V) \\
& \wedge \ (\forall v \in R : v.core = K \wedge v \in V^* \wedge v.d^*_{in} + v.d^+_{out} \leq K) \\
& \wedge \ (\forall v \in Q : v.core = K \wedge v.d^*_{in} \geq 0) \\
& \wedge \ (\forall v \in V : v.d^*_{in} > 0 \Rightarrow v \in Q) \\
& \wedge \ \mathbb{O}(V \setminus V^*)
\end{aligned}
$$

The invariant initially holds as for w, all its predecessors' d^+_{out} are off by 1 and added in R if their $d^*_{in} + d^+_{out} \leq K$ since w is identified in $V^+ \setminus V^*$ (`gray`). We have $w \preceq$ all vertices in V^* in $\mathbb{O}$, denoted as $w \preceq V^*$, as 1) v can be moved to the head of $\mathbb{O}_{K+1}$ and $v.core$ is add by 1 if v is still in V^* when the outer while-loop terminated, and 2) v is removed from V^* and moved after w in O_K. In this case, $w.d^+_{out}$ and $w.d^*_{in}$ are can be correctly updated to $(w.d^+_{out} + w.d^*_{in})$ and 0, respectively. We now argue that the while-loop preserves this invariant:

- $sound(V^* \setminus R)$ is preserved as all $v \in V^*$ are added to R if $v.d^*_{in} + v.d^+_{out} \leq K$.
- $in^*(V)$ is preserved as each time for a vertex $u \in R$ setting from `black` to `gray`, for all its affected successor, which have $d^*_{in} > 0$, their d^*_{in} are off by 1; also, $u.d^*_{in}$ is set to 0 when setting from `black` to `gray` since $u \preceq$ all vertices in V^* in the changed $\mathbb{O}$.
- $out^*(V)$ is preserved as each time for a vertex $u \in R$ setting from `black` to `gray`, for all its affected predecessor, which are in V^*, their d^+_{out} are off by 1; also, $u.d^+_{out}$ is set to $u.d^+_{out} + u.d^*_{in}$ since $u \preceq$ all vertices in V^* in the changed $\mathbb{O}$.
- $(\forall v \in R : v.core = K \wedge v \in V^* \wedge v.d^*_{in} + v.d^+_{out} \leq K)$ is preserved as each time for a vertex $v \in V^*$, v is checked when $v.d^*_{in}$ or $v.d^+_{out}$ is off by 1, and v is added to R if $v.d^*_{in} + v.d^+_{out} \leq K$.
- $\mathbb{O}(V \setminus V^*)$ is preserved as each vertex v that removed from V^* by pealing are moved following p in O_K, where p is w or the previous vertex removed from V^*.

At the termination of the inner while-loop, we get $R = \emptyset$. At the termination of the outer while-loop, we get $Q = \emptyset$. The postcondition of the outer while-loop is $sound(V^*) \wedge complete(V^*)$. □

At the ending phase of Algorithm 2, the core numbers of all vertices in V^* are added by 1, and $\mathbb{O}$ in k-order is maintained. On the termination of Algorithm 2, the core numbers are correctly maintained and also $\mathbb{O}(V) \wedge in^*(V) \wedge out^+(V)$, which provides the correct initial state for the next edge insertion.

5.5 Complexity

Theorem 6 *The time complexity of the simplified order-based insertion algorithm is* $O(|E^+| \cdot \log |E^+|)$ *in the worst case, where* $|E^+|$ *is the number of adjacent edges for all vertices in* V^+ *defined as* $|E^+| = \sum_{v \in V^+} v.deg$.

Proof As the definition of V^+, it includes all traversed vertices to identify V^*. In the `Forward` procedure, the vertices in V^+ are traversed at most once, so do in the `Backward` procedure, which requires worst-case $O(|E^+|)$ time. In the while-loop (Algorithm 2 lines 5 - 10), the min-priority queue Q includes at most $|E^+|$ vertices since each related edge of vertices in V^+ is added into Q at most once. The min-priority queue can be implemented by min-heap, which requires worst-case $O(|E^+| \cdot \log |E^+|)$ time to dequeue all the values. All the vertices in $\mathbb{O}$ are maintained with Order Data Structure so that manipulating the order of one vertex requires amortized $O(1)$ time; there are totally at most $|V^+|$ vertices whose order are manipulated, which requires worst-case $O(|V^+|)$ amortized time. Therefore, the total worst-case time complexity is $O(|E^+| + |E^+| \cdot \log |E^+| + |V^+|) = O(|E^+| \cdot \log |E^+|)$. □

Theorem 7 *The space complexity of the simplified order-based insertion algorithm is* $O(n)$ *in the worst case*.

Proof Each vertex v is assigned three counters that are $v.core$, $v.d^*_{in}$ and $v.d^+_{out}$, which requires $O(3n)$ space. Both Q and R have at most n vertices, respectively, which require worst-case $O(2n)$ space together. Two arrays are required for V^+ and V^*, which requires worst-case $O(2n)$ space. All vertices in $\mathbb{O}$ are maintained by Order Data Structure. For this, all vertices are linked by double-linked lists, which require $O(2n)$ space; also, vertices are assigned labels (typically 64 bits integer) to indicate the order, which requires $O(2n)$ space. Therefore, the total worst-case space complexity is $O(3n + 2n + 2n + 2n + 2n) = O(n)$. □

5.6 The simplified order-based removal

Our simplified order-based removal Algorithm is mostly the same as the original order-based removal Algorithm in [11, 12], so the details are omitted in this section. The only difference is that our simplified order-based removal algorithm adopts the Order Data Structure to maintain $\mathbb{O}$, instead of the complicated $\mathcal{A}$ and $\mathcal{B}$ data structures [12]. In this case, the worst-case time complexity can be improved as the Order Data Structure only requires amortized $O(1)$ time for each order operation.

5.7 Complexities

Theorem 8 *The time complexity of the simplified order-based removal algorithm is* $O(Deg(G) + |E^*|)$ *in the worst case, where* $|E^*| = \sum_{w \in V^*} w.deg$.

Proof Typically, the data graph G is stored by adjacent lists. For removing an edge (u, v), all edges of the vertex u and v are sequentially traversed, which requires at most $O(Deg(G))$ time. We know that V^+ includes all traversed vertices to identify the candidate set V^* and $V^* = V^+$ in this algorithm. The vertices in V^* are traversed at most once, which requires worst-case $O(|E^*|)$ time. All vertices in V^* are removed from the $\mathbb{O}_K$ and appended to $\mathbb{O}_{K-1}$ in k-order, which requires $O(|V^*|)$ time as each insert or remove operation only needs amortized $O(1)$ time by the Order Data Structure. Since it is possible that $Deg(G) > |E^*|$ in some cases like $V^* = \emptyset$, the total worst-case time complexity is $O(Deg(G) + |E^*| + |V^*|) = O(Deg(G) + |E^*|)$. □

Theorem 9 *The space complexity of the simplified order-based removal algorithm is* $O(n)$ *in the worst case*.

Proof For each vertex v in the graph, $v.mcd$ is used to identify the V^*, which requires $O(n)$ space. All vertices in $\mathbb{O}$ are maintained by Order Data Structure, which requires $O(4n)$ space. A queue is used for the propagation, which requires worst-case $O(n)$

space. One array is required for V^*, which requires $O(n)$ space. Therefore, the total worst-case space is $O(n + 4n + n + n) = O(n)$. □

6 The simplified order-based batch insertion

In practice, it is common for a batch of edges to be inserted into a graph together. If multiple edges are inserted one by one, the vertices in $V^+ \setminus V^*$ may be repeatedly traversed. Instead of inserting one by one, we can handle the edge insertion in batch. In this section, we extend our simplified order-based unit insertion algorithm to batch insertion.

Let $\Delta G = (V, \Delta E)$ be an inserted graph to a constructed DAG $\vec{G}$. That is, $\Delta E(\Delta G)$ contains a batch of edges that will be inserted to $\vec{G}$. Each edge $u \mapsto v \in \Delta E$ satisfies $u \preceq v$ in the k-order of $\vec{G}$.

Theorem 10 *After inserted a graph $\Delta G = (V, \Delta E)$ to constructed DAG $\vec{G} = (V, \vec{E})$, the core number of a vertex $v \in V(\vec{G})$ increases by at most 1 if v satisfies $|v.post| \leq v.core + 1$.*

Proof For each $v \in V(\vec{G})$, Lemma 1 proves that the out-degree of v satisfies $|v.post| \leq v.core$. Analogies, when inserting ΔG into $\vec{G}$ with $|v.post| \leq v.core + 1$, the core number can be increased by at most 1, as after inserting the new graph has to satisfy $v.d_{out} \leq v.core$ for all vertices $v \in V(\vec{G})$. □

Theorem 10 suggests that in each round we can insert multiples edges $u \mapsto v \in \Delta E$ into $\vec{G}$ only if $|u(\vec{G}).post| \leq u(\vec{G}).core + 1$; otherwise, $u \mapsto v$ has to be inserted in next round until all edges are inserted. In the worst case, there are $Deg(\Delta G)$ round required if each edges $u \mapsto v \in \Delta E$ satisfy $u(\vec{G}).d^+_{out} = u(\vec{G}).core$.

6.1 The algorithm

Algorithm 5 shows the detailed steps. A batch of edges $u \mapsto v \in \Delta E$ can be inserted into $\vec{G}$ only if $u.d^+_{out} \leq u.core$ (lines 3 and 4). When $u.d^+_{out} = u.core + 1$, we can put u into the Min-Priority Queue Q for propagation (line 5). Of course, the inserted edges are removed from ΔG (line 6). After all possible edges are inserted, the propagation is the same as in lines 5-10 of Algorithm 2 (line 7), where K is the core numbers of local k-subcore with $K = u.core \leq v.core$ for an inserted edge $u \mapsto v$. This process repeatedly continues until the ΔG becomes empty (line 1).

Algorithm 5 BatchEdgeInsert($\vec{G}, \mathbb{O}, \Delta G$)

input : A constructed DAG $\vec{G} = (V, \vec{E})$; the corresponding $\mathbb{O}$; An inserted graph $\Delta G = (V, \Delta E)$.
output: A updated DAG $\vec{G}(V, \vec{E})$; A updated $\mathbb{O}$.

1 **while** $\Delta G \neq \emptyset$ **do**
2 $V^*, V^+, Q \leftarrow \emptyset, \emptyset$, a min-priority queue by $\mathbb{O}$
3 **for** $u \mapsto v \in \Delta E(\Delta G) : u.d^+_{out} \leq u.core$ **do**
4 insert $u \mapsto v$ into $\vec{G}$ with $u.d^+_{out} \leftarrow u.d^+_{out} + 1$
5 **if** $u.d^+_{out} = u.core + 1$ **then** $Q.enqueu(u)$
6 remove $u \mapsto v$ from ΔG
7 same code as lines 5 - 10 in Algorithm 2 with K as the core number of local subcore

Example 4 Consider inserting two edges in the constructed graph in Fig. 3. Initially, the Min-Priority Queue Q is empty, and K is the core number of the corresponding k-subcore. In Fig. 3(a), after inserting two edges, $u_1 \mapsto v_2$ and $u_2 \mapsto v_2$, we get $u_1.d^+_{out} = K + 1 = 2$ and $u_2.d^+_{out} = K + 1 = 2$, so that these two edges can be inserted in batch and we put u_1 and u_2 in Q as $Q = \{u_1, u_2\}$. We begin to propagate Q. First, in Fig. 3(a), u_1 is removed from Q to do the `Forward` procedure since $u_1.d^*_{in} + u_1.d^+_{out} = 0 + 2 > K = 1$, by which u_1 is colored by `black`; within subcore $\mathtt{sc}(u_1)$, all $u_1.post$'s d^*_{in} are added by 1, and all $u_1.post$ are put in Q as $Q = \{u_2\}$. Second, in Fig. 3(b), u_2 is removed from Q to do the `Forward` procedure since $u_2.d^*_{in} + u_2.d^+_{out} = 0 + 2 > K = 1$, by which u_2 is colored by `black`; within subcore $\mathtt{sc}(u_2)$, all $u_2.post$'s d^*_{in} are added by 1, and all $u_2.post$ are put in Q as $Q = \{u_3\}$. Third, in Fig. 3(c), u_3 is removed from Q to do the `Backward` procedure since $u_3.d^*_{in} + u_3.d^+_{out} = 1 + 0 \leq K = 1$, by which u_3 is colored by `black` and $u_2.d^+_{out}$ off by 1; however, since $u_2.d^*_{in} + u_2.d^+_{out} = 1 + 1 > K = 1$, we have u_2 still `black` and the `Backward` procedure terminates. Finally, in Fig. 3(d), two `black` vertices, u_1 and u_2, have increased core numbers as 2; then, they are removed from $\mathbb{O}_1$ and inserted before the head of $\mathbb{O}_2$ to maintain the k-order.

In this example, we have $V^* = \{u_1, u_2\} \wedge V^+ = \{u_1, u_2, u_3\}$ by batch inserting two edges together. If we insert $u_1 \mapsto v_2$ first and then insert $u_2 \mapsto v_2$ second, the final V^* is same; but V^+ is $\{u_1, u_2, u_3\}$ and $\{u_2, u_3\}$ for two inserted edges, respectively. In this case, both u_2 and u_3 are repeatedly traversed, which can be avoided by batch insertion.

6.2 Correctness

For each round of the while-loop (lines 2 - 7), the correctness argument is totally the same as the single edge insertion in Algorithm 2.

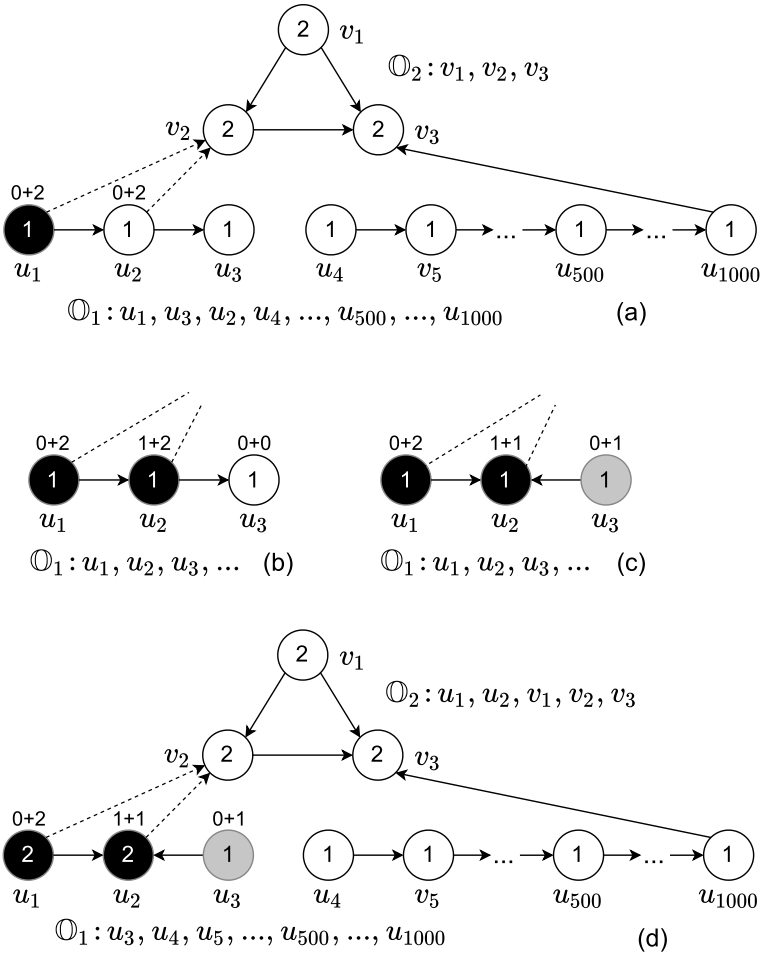


Fig. 3 Insert a batch of two edges $u_1 \mapsto v_2$ and $u_2 \mapsto v_2$ to a constructed graph $\vec{G}$ obtained from Fig. 1

6.3 Complexities

The total worst-case running time of line 7 is the same as in Algorithm 2, which is $O(|E^+| \cdot \log |E^+|)$. Typically, for Q, the running time of enqueue and dequeue is larger than in Algorithm 2 since each time numerous vertices can be initially added into Q for propagation (line 5). The outer while-loop (line 1) runs at most ΔE rounds, so that ΔE is checked at most $O(Deg(\Delta G) \cdot |\Delta E|)$ round as $\mathbb{O}$ can be changed and thus the directions of edges in ΔE can be changed. Typically, the majority of edges can be inserted in the first round of the while-loop. Therefore, the time complexity of Algorithm 5 is $O(|E^+| \cdot \log |E^+| + Deg(\Delta G) \cdot |\Delta E|)$ in the worst case.

The space complexity of Algorithm 5 is the same as Algorithm 2.

7 Experiments

In this section, we conduct experimental studies using 12 real and synthetic graphs and report the performance of our algorithm by comparing it with the original order-based method:

- The order-based algorithm [12] with unit edge insertion (`I`) and edge removal (`R`); Before running, we execute the initialization (`Init`) step.
- Our simplified order-based with unit edge insertion (`OurI`) and edge removal (`OurR`); Before running, we execute the initialization (`OurInit`) step.
- Our simplified order-based batch edge insertion (`OurBI`).

Note that, the other important compared method, traversal algorithm [11], is omitted in our experiments. The reason is that the traversal algorithm is experimentally well-studied in [12], and the result shows that the order-based algorithm significantly outperforms the traversal algorithm.

7.1 Experiment setup

The experiments are performed on a desktop computer with an Intel CPU (4 cores, 8 hyperthreads, 8 MB of last-level cache) and 16 GB of main memory. The machine runs the Ubuntu Linux (18.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 7.3.0 with the -O3 option. All implementations and results are available at github.[1]

7.2 Tested graphs

We evaluate the performance of different methods over a variety of real-world and synthetic graphs, which are shown in Table 2. For simplicity, directed graphs are converted to undirected ones in our testing; all of the self-loops and repeated edges are removed. That is, a vertex cannot connect to itself, and each pair of vertices can connect with at most one edge. The *livej*, *patent*, *wiki-talk*, and *roadNet-CA* graphs are obtained from SNAP.[2] The *dbpedia*, *baidu*, *pokec* and *wiki-talk-en wiki-links-en* graphs are collected from the KONECT[3] project. The *ER*, *BA*, and *RMAT* graphs are synthetic graphs; they are generated by the SNAP[4] system using Erdös–Rényi, Barabasi–Albert, and the R-MAT graph models, respectively. For these generated graphs, the average degree is fixed to 8 by choosing 1,000,000 vertices and 8,000,000 edges.

In Table 2, for the columns, *AvgDeg* is the average degree for all vertices, *Max-k* is the maximum value of k for all vertices, *Diameter* is the longest shortest path between each pair of vertices, and *AvgCC* is the average clustering coefficient (the

[1] https://github.com/Itisben/SimplifiedCoreMaint.git.

[2] http://snap.stanford.edu/data/index.html.

[3] http://konect.cc/networks/.

[4] http://snap.stanford.edu/snappy/doc/reference/generators.html.

Table 2 Tested real and synthetic graphs

Graph	$n = \|V\|$	$m = \|E\|$	AvgDeg	Max-k	Diameter	AvgCC
livej	4,847,571	68,993,773	14.23	372	16	0.274,2
patent	6,009,555	16,518,948	2.75	64	22	0.075,7
wikitalk	2,394,385	5,021,410	2.10	131	9	0.052,6
roadNet-CA	1,971,281	5,533,214	2.81	3	849	0.046,4
dbpedia	3,966,925	13,820,853	3.48	20	67	0.000,143,386
baidu	2,141,301	17,794,839	8.31	78	20	0.002,448,50
pokec	1,632,804	30,622,564	18.75	47	162	0.046,821,2
wiki-talk-en	2,987,536	24,981,163	8.36	210	9	0.002,203,51
wiki-links-en	5,710,993	130,160,392	22.79	821	12	0.014,303,2
ER	1,000,000	8,000,000	8.00	11	–	–
BA	1,000,000	8,000,000	8.00	8	–	–
RMAT	1,000,000	8,000,000	8.00	237	–	–

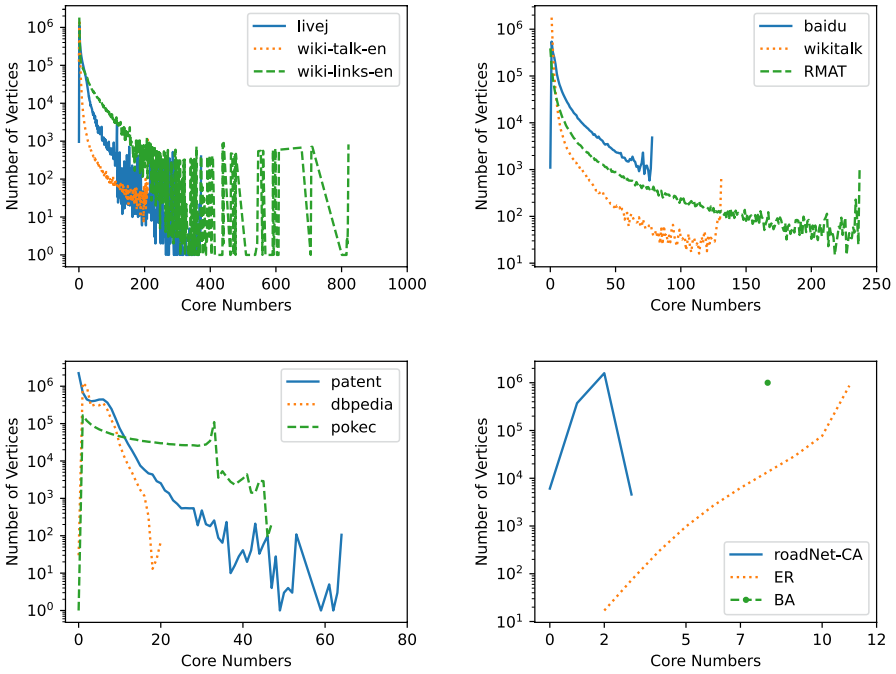


Fig. 4 The distribution of core numbers

probability that a pair of randomly chosen connected edges is completed by a third edge to form a triangle [31]). We can see that all graphs have millions of edges, their average degrees ranged from 2.1 to 22.8, and their maximal core numbers range from 3 to 821. For each tested graph, the distribution of the core numbers for all the vertices is shown in Fig. 4, where the x-axis is the core numbers and the y-axis is the

number of the vertices with the corresponding core numbers. For most graphs, many vertices have small core numbers, and few have large core numbers. Specifically, *wiki-link-en* has the maximum core numbers up to 821, so that for most of its $\mathbb{O}_k$, the sizes are around 1000; *BA* has a single core number as 8 so that all vertices are in the single $\mathbb{O}_k$. Since all vertices with core number k in $\mathbb{O}_k$ are maintained in k-order, the size of $\mathbb{O}_k$ is related to the performance of different methods.

7.3 Running time evaluation

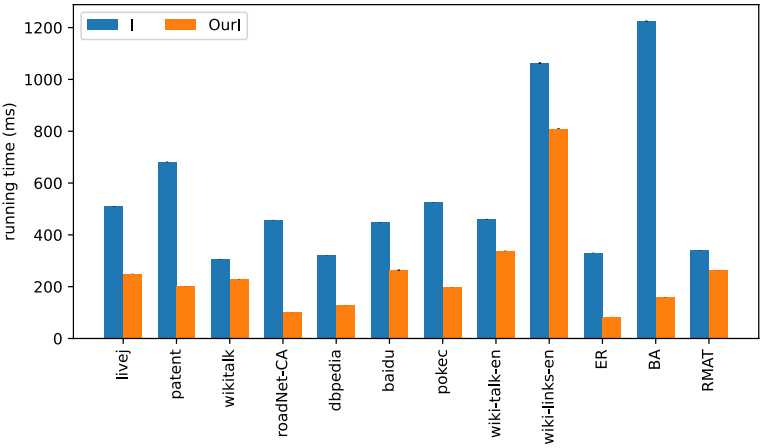
In this experiment, we compare the performance of our simplified order-based method (`OurI` and `OurR`) with the original order-based method (`I` and `R`). For each tested graph, we first randomly select 100,000 edges out of each tested graph. For each graph, we measure the accumulated time for inserting or removing these 100,000 edges. Each test runs at least 50 times, and we calculate the means with 95% confidence intervals.

The results for edge insertion are shown in Fig. 5(a). We can see `OurI` outperforms `I` over all tested graphs. Specifically, Table 3 shows the speedups of `OurI` vs. `I`, which ranges from 1.29 to 7.69. The reason is that the sequence $\mathbb{O}_k$ in k-order is maintained separately for each core number k. Each time insert v into or remove v from $\mathbb{O}_k$, `OurI` only requires worst-case $O(1)$ amortized time while `I` requires worst-case $O(\log |\mathbb{O}_k|)$ time. Therefore, over *BA* we can see `OurI` gains the largest speedup as 7.69 since all vertices have single one core number with $|\mathbb{O}_8| = 8,000,000$; over *wiki-links-en* we can see `OurI` gains the smallest speedup as 1.29 since vertices have core numbers ranging from 0 to 821 such that a large portion of order lists has $|\mathbb{O}_k|$ around 1000.

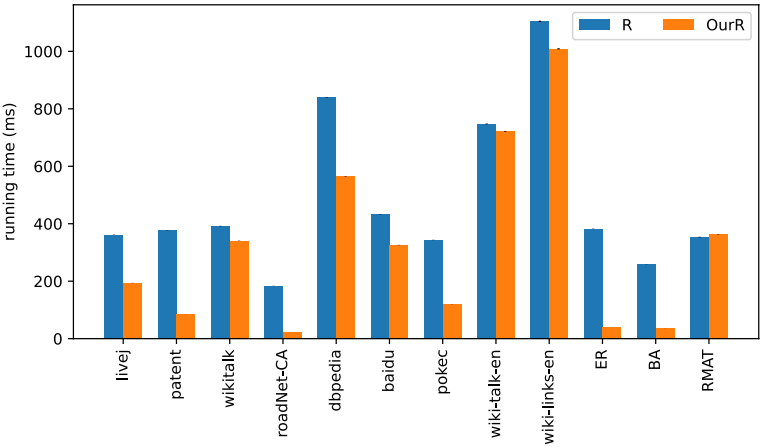
Similarly, we can see that the edge removal has almost the same trend of speedups in Fig. 5b, which ranges from 1.16 to 5.26 in Table 3. However, we observe that the speedups of removal may be less than the insertion over most graphs. The reason is that edge removal requires fewer order operations compared with edge insertion. That is, unlike the edge insertion, it is not necessary to compare the k-order for two vertices by $\texttt{ORDER}(\mathbb{O}, x, y)$ when reversing the vertices. The main order operations are $\texttt{REMOVE}(\mathbb{O}, x)$ and then $\texttt{INSERT}(\mathbb{O}, x, y)$, when the core numbers of vertices $x \in V^*$ are off by 1.

In Table 3, for the batch insertion, we can see the speedups of `OurBI` vs. `I` are much less than the speedups of `OurI` vs. `I`, although `OurBI` may have smaller size of V^+ than `OurI`. One reason is that `OurBI` has to traverse inserted graph ΔG at most $Deg(\Delta G)$ round, which is the maximum degree of the inserted graph ΔG. The other reason is that compared with `OurI`, `OurBI` has a larger size of priority queue Q, which `OurBI` requires more running time on enqueue and dequeue operations.

In Table 3, we also observe that the speedups of `OurInit` vs. `Init` are a little larger than 1. The reason is that for initialization, most of the running time is spent on computing the core number for all vertices by the BZ algorithm. After running the BZ algorithm, `OurInit` assigns labels for all vertices to construct $\mathbb{O}$ in



(a) Edge Insertion



(b) Edge Removal

Fig. 5 Compare the running times of two methods

k-order, which requires worst-case $O(n)$ time. However, `Init` has to add all vertices to binary search trees, which requires worst-case $O(n \log n)$ time.

7.4 Index space and creation evaluation

For the implementation, `OurI` and `OurR` need about $15n$ space, while `I` and `R` need about $23n$ space. That is, our method only requires about 65% space of compared methods. Also, for creating the index (computing the initial core numbers and constructing the k-order for all vertices), our method is roughly 10% faster than the compared method. This is because `OurI` and `OurR` use OM data structures to maintain

Table 3 Compare the speedups of our method for all graphs

Graph	OurI vs I	OurBI vs I	OurR vs R	OurInit vs Init
livej	2.04	1.66	1.87	1.02
patent	3.37	2.68	4.41	1.04
wikitalk	1.34	1.63	1.15	1.26
roadNet-CA	4.51	2.95	8.56	1.17
dbpedia	2.49	2.14	1.49	1.08
baidu	1.70	1.68	1.33	1.04
pokec	2.67	2.37	2.87	1.03
wiki-talk-en	1.36	1.45	1.04	1.20
wiki-links-en	1.31	1.16	1.09	1.02
ER	3.97	2.76	9.72	1.08
BA	7.69	5.26	7.42	1.15
RMAT	1.29	1.31	0.97	1.09

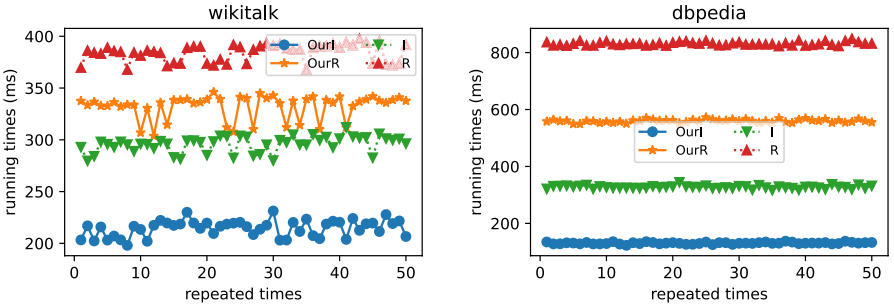


Fig. 6 The stability of all methods over selected graphs

the k-order of all vertices, which has improved time and space complexities compared with double-linked lists combined with binary search trees and heads used in `I` and `R`.

7.5 Stability evaluation

We test the stability of different methods over two selected graphs, i.e., *wikitalk* and *dbpedia* as follows. First, we randomly sample 5, 000, 000 edges and partition them into 50 groups, where each group has totally different 100, 000 edges. Second, for each group, we measure the accumulated running time of different methods. That is, the experiments run 50 times, and each time has totally different inserted or removed edges.

Figure 6 shows the results over two selected graphs. We can see that `OurI` and `OurR` outperform `I` and `R`, respectively. More importantly, the performance of `OurI` and `OurR` is as well-bounded as `I` and `R`, respectively. The reason is that `I` and `R` are well-bounded as the variation of V^+ is small for different inserting or

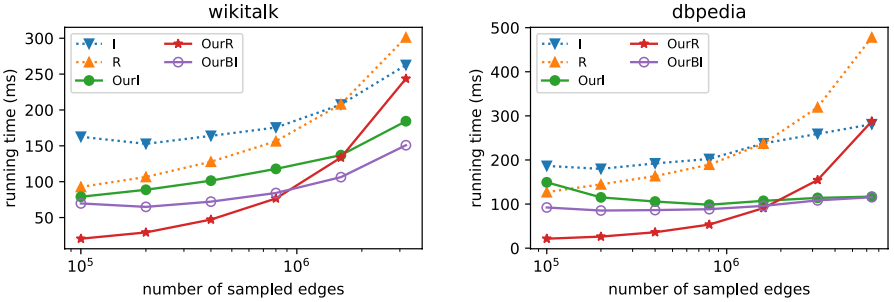


Fig. 7 The scalability of all methods over selected graphs

removal edges; also, `OurI` and `OurR` have the same size of traversed vertices V^+ and thus have similar well-bounded performance.

7.6 Scalability evaluation

We test the scalability of different methods over two selected graphs, i.e., *wikitalk* and *dbpedia*. We vary the number of edges exponentially by randomly sampling from 100,000 to 200,000, 400,000, 800,000, 1,600,000, etc. We keep incident vertices of edges for each sampling to generate the induced subgraphs. Over each subgraph, we further randomly selected 100,000 edges for insertion or removal. For example, over *wikitalk*, the first subgraph has 100,000 edges, all of which can be inserted or removed; the last subgraph has 3,200,000 edges, only 100,000 of which can be inserted or removed. Over each subgraph, we measure the accumulated time for the insertion or removal of these 100,000 edges. Each test runs at least 50 times, and we calculate the average running time.

We show the result in Fig. 7, where the x-axis is the number of sampled edges in subgraphs increasing exponentially, and the y-axis is the running times (ms) for different methods by inserting or removing 100,000 edges. Table 4 shows the details of scalability evaluation, where m' is the number of edges in subgraphs, *#lb* is the number of updated labels used by the Order Data Structures of our methods, and *#rp* is the number of outer while-loop repeated rounds for `OurBI`. We make several observations as follows:

- In Fig. 7, a first look reveals that the running time of `OurI` grows more slowly compared with `I`. The reason is that `OurI` improves the worst-case running time of each order operation of $\mathbb{O}_k$ from $O(\log |\mathbb{O}_k|)$ to $O(1)$. In this case, the larger sampled graphs have a larger size of $\mathbb{O}_k$, which can lead to higher speedups. However, we can see that the running time of `OurR` always grows with a similar trend compared with `R`. The reason is that a large percentage of the running time is spent on removing edges from the adjacent lists of vertices, which requires traversing all the corresponding edges. Because of this, even `OurR` has more efficient order operations for $\mathbb{O}$ than `R`, the speedups are not obvious.

Table 4 The details of scalability evaluation by varying the number of sampled edges over *wikitalk* and *dbpedia*

	OurI			OurBI				OurR	
m'	$\|V^*\|$	$\|V^+\|$	*#lb*	$\|V^*\|$	$\|V^+\|$	*#lb*	*#rp*	$\|V^*\|$	*#lb*
0.1M	107K	131K	1.5M	107K	123K	116K	10	107K	107K
0.2M	101K	118K	1.4M	101K	109K	111K	11	101K	101K
0.4M	101K	116K	1.3M	101K	107K	107K	9	101K	101K
0.8M	101K	113K	1.3M	101K	106K	104K	10	101K	101K
1.6M	100K	110K	1.2M	100K	106K	103K	11	100K	100K
3.2M	101K	110K	1.1M	101K	106K	103K	9	101K	101K
0.1M	146K	149K	2.6M	146K	151K	149K	4	146K	146K
0.2M	129K	136K	2.1M	129K	136K	132K	3	129K	129K
0.4M	117K	131K	1.8M	117K	130K	124K	3	117K	117K
0.8M	109K	127K	1.6M	109K	126K	118K	3	109K	109K
1.6M	105K	127K	1.4M	105K	125K	116K	4	105K	105K
3.2M	102K	124K	1.3M	102K	122K	113K	4	102K	102K
6.4M	100K	124K	1.2M	100K	122K	112K	4	100K	100K

- In Fig. 7, we observe that `OurBI` sometimes runs faster than `OurI`. The reason is as follows. From Table 4, compared with `OurI`, `OurBI` has less traversed vertices (V^+), as some repeated traversed vertices can be avoided; also, `OurBI` has less number of updated labels (*#lb*), as the number of relabel process can be reduced. However, compared with `OurI`, `OurBI` may add much more vertices into priority queue Q, which costs more the running time of enqueue and dequeue. Also, `OurBI` may require several times of repeated rounds (*#rp*), which may cost extra running time. Typically, this extra running time is acceptable as most of the edges can be inserted in the first round, e.g., for *dbpedia* with 6.4M sampled edges, the number of batch inserted edges is 100000, 1555, 12, and 0 in four rounds, respectively. This is why `OurBI` sometimes runs faster but sometimes slower compared with `OurI`.
- In Fig. 7, we observe that `OurR` is always faster than `OurI`. The reason is as follows. From Table 4, compared with `OurI`, `OurR` has less number of traversed vertices (V^*), as `OurR` has $V^* = V^+$; `OurR` has less number of updated labels (*#lb*), as vertices are removed from O_K and then appended after O_{K-1} and thus the relabel process is not always triggered.

8 Conclusion and future work

In this work, we study maintaining the k-core of graphs when inserting or removing edges. We simplify the state-of-the-art core maintenance algorithm and also improve its worst-case time complexity by introducing the classical Order Data Structure. Our simplified approach is easy to understand, implement, and argue the

correctness. The experiments show that our approach significantly outperforms the existing methods.

The big advantage of our proposed simplified approach is that it is easy to be parallelized [32]. In other words, our approach is a preparation step for parallel k-core maintenance. Also, the same methodology can be applied to other graphs, e.g., weighted graphs and probability graphs. Our approach can be extended to other graph algorithms, e.g., maintaining the k-truss in dynamic graphs. Additionally, the maintenance of the hierarchical k-core, which involves maintaining the connections among different k-cores in the hierarchy, can benefit from our result.

Author contributions Bin Guo writes the paper and does the experiments. As the supervisor, Emil Sekerinski reviews the paper.

Funding NSERC (Natural Sciences and Engineering Research Council of Canada) discovery.

Data availability All real graphs are downloaded from KONECT (http://konect.cc/networks/) and SNAP (http://snap.stanford.edu/data/index.html). All synthetic graphs are generated by using the graph generator of SANP (http://snap.stanford.edu/snappy/doc/reference/generators.html). The source code for this paper is available on GitHub (https://github.com/Itisben/SimplifiedCoreMaint.git).

Declarations

Conflict of interest Not applicable.

Ethical approval Not applicable.

References

1. Batagelj V, Zaversnik M (2003). An o(m) algorithm for cores decomposition of networks. CoRR cs.DS/0310049
2. Kong YX, Shi GY, Wu RJ, Zhang YC (2019) k-core: theories and applications. Tech Rep. https://doi.org/10.1016/j.physrep.2019.10.004 (http://doc.rero.ch)
3. Burleson-Lesser K, Morone F, Tomassone MS, Makse HA (2020) K-core robustness in ecological and financial networks. Sci Rep 10(1):1–14
4. Malliaros FD, Giatsidis C, Papadopoulos AN, Michalis V (2020) The core decomposition of networks: theory, algorithms and applications. VLDB J 29, 61–92. https://doi.org/10.1007/s00778-019-00587-4
5. Cheng J, Ke Y, Chu S, Özsu M.T (2011). Efficient core decomposition in massive networks. In: 2011 IEEE 27th international conference on data engineering, 51–62 . IEEE
6. Khaouid W, Barsky M, Srinivasan V, Thomo A (2015) K-core decomposition of large networks on a single pc. Proceed VLDB Endowment 9(1):13–23
7. Montresor A, De Pellegrini F, Miorandi D (2012) Distributed k-core decomposition. IEEE Trans Parallel Distrib Syst 24(2):288–300
8. Wen D, Qin L, Zhang Y, Lin X, Yu J.X (2016). I/o efficient core graph decomposition at web scale. In: 2016 IEEE 32nd international conference on data engineering (ICDE), 133–144 . IEEE
9. Miorandi D, De Pellegrini F (2010) K-shell decomposition for dynamic complex networks. In: 8th international symposium on modeling and optimization in mobile, Ad Hoc, and wireless networks, 488–496 . IEEE
10. Pei S, Muchnik L, Andrade JS Jr, Zheng Z, Makse HA (2014) Searching for Superspreaders of information in real-world social media. Sci Rep 4:5547
11. Sarıyüce AE, Gedik B, Jacques-Silva G, Wu K-L, Çatalyürek ÜV (2016) Incremental k-core decomposition: algorithms and evaluation. VLDB J 25(3):425–447

12. Zhang Y, Yu J.X, Zhang Y, Qin L (2017). A fast order-based approach for core maintenance. In: Proceedings—international conference on data engineering, 337–348 . https://doi.org/10.1109/ICDE.2017.93
13. Wu H, Cheng J, Lu Y, Ke Y, Huang Y, Yan D, Wu H (2015). Core decomposition in large temporal graphs. In: 2015 IEEE international conference on big data (Big data), 649–658 . IEEE
14. Saríyüce AE, Gedik B, Jacques-Silva G, Wu K-L, Çatalyürek ÜV (2013) Streaming algorithms for k-core decomposition. Proceed VLDB Endowment 6(6):433–444
15. Dietz P, Sleator D (1987) Two algorithms for maintaining order in a list. In: Proceedings of the nineteenth annual ACM symposium on theory of computing, 365–372
16. Bender M.A, Cole R, Demaine E.D, Farach-Colton M, Zito J (2002) Two simplified algorithms for maintaining order in a list. In: European symposium on algorithms, 152–164 . Springer
17. Seidman SB (1983) Network structure and minimum degree. Soc Netw 5(3):269–287
18. Dasari N.S, Desh R, Zubair M (2014). Park: An efficient algorithm for k-core decomposition on multicore processors. In: 2014 IEEE international conference on big data (Big Data), 9–16. IEEE
19. Kabir H, Madduri K (2017) Parallel k-core decomposition on multicore platforms. In: 2017 IEEE international parallel and distributed processing symposium workshops (IPDPSW), 1482–1491. IEEE
20. Chan THH, Sozio M, Sun B (2021) Distributed approximate k-core decomposition and min-max edge orientation: Breaking the diameter barrier. J Parallel Distrib Comput 147:87–99
21. Zhang Y, Yu JX (2019) Unboundedness and efficiency of truss maintenance in evolving graphs. In: Proceedings of the ACM SIGMOD international conference on management of data, 1024–1041. Association for computing machinery. https://doi.org/10.1145/3299869.3300082
22. Li RH, Yu JX, Mao R (2013) Efficient core maintenance in large dynamic graphs. IEEE Trans Know Data Eng 26(10):2453–2465
23. Wang N, Yu D, Jin H, Qian C, Xie X, Hua QS (2017) Parallel algorithm for core maintenance in dynamic graphs. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS), 2366–2371. IEEE
24. Jin H, Wang N, Yu D, Hua Q.S, Shi X, Xie X (2018). Core maintenance in dynamic graphs: a parallel approach based on matching. IEEE Trans Parallel Distrib Syst 29(11), 2416–2428 https://doi.org/10.1109/TPDS.2018.2835441arXiv:1703.03900
25. Yu M, Wen D, Qin L, Zhang Y, Zhang W, Lin X (2021) On querying historical k-cores. Proceed VLDB Endowment 14(11):2033–2045
26. Sun B, Chan THH, Sozio M (2020) Fully dynamic approximate k-core decomposition in hypergraphs. ACM Trans Know Discov Data (TKDD) 14(4):1–21
27. Liu Q.C, Shi J, Yu S, Dhulipala L, Shun J (2021). Parallel batch-dynamic algorithms for *k*-core decomposition and related graph problems. arXiv preprint arXiv:2106.03824
28. Lin Z, Zhang F, Lin X, Zhang W, Tian Z (2021) Hierarchical core maintenance on large dynamic graphs. Proceed VLDB Endowment 14(5):757–770
29. Liu B, Liu Z, Zhang F (2021). An order approach for the core maintenance problem on edge-weighted graphs. In: Algorithmic aspects in information and management: 15th international conference, AAIM 2021, virtual event, 2021, Proceedings, 426–437. Springer
30. Yang H, Wang K, Sun R, Wang X (2020) Fast algorithms for spatial k-core discovery and maintenance. In: 2020 IEEE 22nd international conference on high performance computing and communications; IEEE 18th international conference on smart city; IEEE 6th international conference on data science and systems (HPCC/SmartCity/DSS), 841–846. IEEE
31. Watts DJ, Strogatz SH (1998) Collective dynamics of small-world networks. Nature 393(6684):440–442
32. Guo B, Sekerinski E (2022) Parallel order-based core maintenance in dynamic graphs. arXiv preprint arXiv:2210.14290