TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 07/15 pp292–309
DOI: 10.26599/TST.2021.9010091
Volume 28, Number 2, April 2023

Core Decomposition and Maintenance in Bipartite Graphs

Dongxiao Yu, Lifang Zhang, Qi Luo*, Xiuzhen Cheng, and Zhipeng Cai

Abstract: The prevalence of graph data has brought a lot of attention to cohesive and dense subgraph mining. In contrast with the large number of indexes proposed to help mine dense subgraphs in general graphs, only very few indexes are proposed for the same in bipartite graphs. In this work, we present the index called $\alpha(\beta)$-core number on vertices, which reflects the maximal cohesive and dense subgraph a vertex can be in, to help enumerate the (α, β)-cores, a commonly used dense structure in bipartite graphs. To address the problem of extremely high time and space cost for enumerating the (α, β)-cores, we first present a linear time and space algorithm for computing the $\alpha(\beta)$-core numbers of vertices. We further propose core maintenance algorithms, to update the core numbers of vertices when a graph changes by avoiding recalculations. Experimental results on different real-world and synthetic datasets demonstrate the effectiveness and efficiency of our algorithms.

Key words: core decomposition; core maintenance; bipartite graph; dense subgraph mining

1 Introduction

Many real-world relationships among different entities are modeled as bipartite graphs, including collaboration networks[1], gene co-expression networks[2], and user-item networks[3]. The vertices of a bipartite graph $G = (U, V, E)$ are composed of two disjoint sets, i.e., U and V, and each edge connects a vertex in U to a vertex in V. Due to the wide range of applications of bipartite graphs, such as user recommendation[4], fraud identification[1], privacy-preserving[5–7], and genetic analysis[8], the problem of making better use of bipartite graph data and its analysis have been studied in recent years. Among them, the problem of identifying cohesive and dense subgraphs has drawn a great deal of attention from research communities and industry. The (α, β)-core is considered as an effective tool for mining dense subgraphs in bipartite graphs, and computing (α, β)-core for a given α and β has become an important research topic[4, 9].

The (α, β)-core is a maximal bipartite subgraph G' derived from two vertex sets $U' \subseteq U$ and $V' \subseteq V$, and it satisfies that all the vertices in U' and V' have a degree of at least α and β, respectively. According to the definition, the basic algorithm for computing an (α, β)-core when given α and β, was presented in Ref. [4]. In brief, the main idea is to iteratively remove vertices from U and V that have a degree less than α and β, respectively, until no more vertices can be deleted. The time complexity of computing an (α, β)-core is $O(m)$ in the worst case, where m is the number of edges in the bipartite graph. Therefore, when the scale of the graph is very large, implementing this method is not acceptable since it needs to traverse the whole graph which costs too much time. To solve this problem, a linear space index structure called BiCore-index is proposed in Ref. [9], which guarantees that an (α, β)-core can be calculated in the optimal time, i.e., linear time with respect to the result size. Specifically, it precomputes (α, β)-cores for all possible combinations of α and β and then makes use of a three-level tree structure to preserve them locally. As a result, when given a particular α and β combination, it figures out the (α, β)-core immediately by visiting the

• Dongxiao Yu, Lifang Zhang, Qi Luo, and Xiuzhen Cheng are with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China. E-mail: {dxyu, xzcheng}@sdu.edu.cn; {zhanglf, luoqi2018}@mail.sdu.edu.cn.
• Zhipeng Cai is with the Department of Computer Science, Georgia State University, Atlanta, GA 30303, USA. E-mail: zcai@gsu.edu.
∗ To whom correspondence should be addressed.
Manuscript received: 2021-10-12; revised: 2021-11-16; accepted: 2021-11-19

vertices stored in the corresponding location. However, even if the authors have optimized the index structure, it still takes up a lot of space.

Inspired by the concept of k-core in general graphs[10], we present the concept of $\alpha(\beta)$-core number in bipartite graphs, to solve the problem of high time and space costs in the computation of (α, β)-cores. Specifically, given an α (or β), the $\alpha(\beta)$-core number of a vertex v is defined as the maximum $\hat{\beta}$ such that v can be in an $(\alpha, \hat{\beta})$-core. After computing $\alpha(\beta)$-core numbers of all vertices for all combinations of α and β, it is easy to enumerate the (α, β)-cores in the bipartite graph. Meanwhile, comparing the index structure given in Ref. [9], the space for storing the index can be significantly reduced. More importantly, $\alpha(\beta)$-core exhibits its merit in handling dynamic graphs. As the graph change usually affects a small part of the graph, it only needs to update the core numbers of a small number of vertices that are affected, avoiding recomputation of all (α, β)-cores.

With the concept of $\alpha(\beta)$-core number, we first study the core decomposition problem, i.e., computing the core numbers of vertices. For any given α (or β), we propose an algorithm that can compute the $\alpha(\beta)$-core numbers of vertices in linear time and space. We further propose efficient algorithms for the core maintenance problem, i.e., updating the core numbers of vertices after the graph changes. We focus on the scenario of edge insertion/deletion since the vertex changes can be seen as edge changes[11].

We start from the single-edge insertion/deletion scenario. By proposing the concept of *pre-core*, it allows us to quantify the variance of the core number of every vertex and identify vertices whose core numbers change. Based on this observation, we propose core maintenance algorithms that can greatly reduce the number of vertices and edges visited during the process of updating, thus facilitating a quicker update of the core numbers of vertices. Experiments on real-world graphs show that the update time can be reduced by 80% to 90% compared with the recomputation of the core numbers using the core decomposition algorithm.

We further consider a general case where multiple edges are inserted/deleted into/from a graph. An intuition approach is to adopt the single-edge core maintenance algorithms to handle the inserted/deleted edges one by one. But this approach is obviously inefficient. To improve the efficiency of core maintenance, we propose a batch processing approach that can handle multiple edge insertions/deletions simultaneously. However, the core number change becomes much more complex when multiple edges are inserted/deleted together. This is because the change in the core number of a vertex can be affected by multiple inserted/deleted edges, which makes it hard to determine the exact change on the core number value. Furthermore, different from the single-edge scenario, it is difficult to identify the set of vertices that may change the core numbers. To overcome these difficulties, a new structure of inserted/deleted edges, called V-independent edge set (*VES*), is proposed to solve the challenge of determining the core number change of vertices. We show that the insertion/deletion of a *VES* can make the core number of each vertex change by at most 1. In this way, when inserting/deleting a V-independent edge set, the core maintenance task is simplified to identify the vertices whose core numbers change, such that multiple edges can be handled simultaneously. Then the inserted/deleted edges can be processed iteratively by splitting these edges into multiple disjoint *VES*s. With this batch processing approach, we propose core maintenance algorithms that can reduce the number of iterations for processing the inserted/deleted edges from the number of inserted/deleted edges to the maximum number of edges inserted/deleted to each vertex in V. The experiments on real-world graphs show that the batch processing algorithms can further reduce the updating time from 50% to 90%, comparing with the single-edge processing algorithms.

The rest of this paper is organized as follows: We first review the related work in Section 2 and then give some basic concepts in Section 3, including the definition of $\alpha(\beta)$-core number of each vertex in a bipartite graph. In Section 4, the specific algorithm for core decomposition is proposed. Based on the theoretical results, we give the single-edge processing and batch processing algorithms in Sections 5 and 6, respectively. In Section 7, the experimental results are reported. Finally, the paper is concluded in Section 8.

2 Related Work

Dense subgraph mining. With the prevalence of graph data, mining dense subgraphs has attracted a lot of attention[12, 13]. One fundamental structure of dense subgraph is clique in which every pair of vertices is connected. However, computing a clique is NP-Hard[14], so a lot of work has been done to come up with new structures to loosen up the clique. Among them, the most commonly used indexes to measure dense subgraphs include k-core[15–17], k-truss[18–20], k-plex[21, 22], etc. These indexes are all proposed for general graphs. In contrast, there are fewer indexes that are particularly presented for dense subgraph mining in bipartite graphs. In Ref. [23], (α, β)-core in bipartite graphs was first introduced and algorithms for computing (α, β)-core were proposed in Refs. [4, 9]. Moreover, a few other works tried to adapt some indexes of dense subgraphs in general graphs to bipartite graphs. For example, in Ref. [24], it changed the concept of k-clique to biclique and quasi-biclique was further proposed in Ref. [25].

Core maintenance. The core maintenance problem is to update the core numbers of partial vertices after the graph changes instead of recalculating the entire graph. In Ref. [26], it is proven that the core number of each vertex can change by at most 1 after inserting/deleting one edge and proposed an effective algorithm, called TRAVERSE, to identify the vertex set whose core numbers really change. In Ref. [27], Wang et al. proposed a structure called superior edge set and showed that inserting/deleting these edges into/from a graph changes the core number of each vertex by at most 1. Based on it, parallel algorithms were presented to simultaneously process multiple edges in each iteration. Reference [28] showed that if the inserted/deleted edges constitute a matching, the core number update with respect to each inserted/deleted edge can be handled in parallel. In Ref. [11], it further optimized the structure of edge set, which greatly improves the efficiency of core maintenance. In Ref. [29], an effective algorithm was proposed to improve the I/O efficiency of core maintenance. The core maintenance problem in the distributed environment was studied in Refs. [30, 31]. In Ref. [32], it presented a new definition of weighted coreness for vertices in a weighted graph. Reference [33] studied exact hypercore maintenance in large-scale dynamic hypergraphs. Although there have been many definitions of core numbers under different network models, none of them distinguishes between different types of vertices. In fact, in a bipartite graph, such as a user-item graphs, there should be different constraints on users and items. Therefore, core numbers on bipartite graph are necessary to be studied.

3 Definition

We use $G = (U, V, E)$ to represent a bipartite graph, where $U(G)$ and $V(G)$ denote two disjoint vertex sets and $E(G)$ denotes the edge set. The number of vertices in $U(G)$ ($V(G)$) is denoted by N_u (N_v), and let N and M be the total number of vertices and edges in G, respectively. For a bipartite graph, there is no edge between U or between V, i.e., $\forall e = (u, v) \in E(G)$, it satisfies that $u \in U(G)$ and $v \in V(G)$ or vice versa. We use $deg_G(v)$ and $neigh_G(v)$ to represent the degree and neighbors of the vertex v in graph G, respectively. When the context is clear, we can omit G in the symbolic representation for simplicity. Given two vertex sets $U' \subseteq U(G)$ and $V' \subseteq V(G)$, we can obtain a bipartite subgraph G' of G induced by U' and V' such that $U(G') = U'$, $V(G') = V'$, and $E(G') = E(G) \cap (U' \times V')$.

Definition 1 (**(α, β)-core**) Given a bipartite graph G=(U, V, E) and two integers α and β, the (α, β)-core is a maximal bipartite subgraph G' derived from two vertex sets U' and V' where $U' \subseteq U$ and $V' \subseteq V$, satisfying that the degree of each vertex is not less than α in U' and not less than β in V', i.e., $\forall u \in U'$, $deg_{G'}(u) \geqslant \alpha$ and $\forall v \in V'$, $deg_{G'}(v) \geqslant \beta$.

An example of a bipartite graph is shown in Fig. 1. The vertices in the graph are divided into two sets, represented by U (the top part) and V (the bottom part). The two endpoints of all edges are one from U and one from V. $\{u_3, u_4, u_5, v_2, v_3\}$ is a (3, 2)-score and $\{u_1, u_2, v_1\}$ is a (2, 1)-core. According to Definition 1, an (α, β)-core in a bipartite graph can be easily computed by iteratively removing vertices that do not satisfy the degree constraint, and its time complexity is $O(M)$ in the worst case[4]. However, the scale of the real graph is usually very large, so it will take a lot of time to calculate an (α, β)-core. In addition, the parameter α (β) may frequently change due to the graph evolution, so the computation algorithm is very expensive, since it needs to traverse the whole graph each time. In Ref. [9], an index structure called BiCore was proposed to prestore all (α, β)-cores, which guarantees to get an (α, β)-core

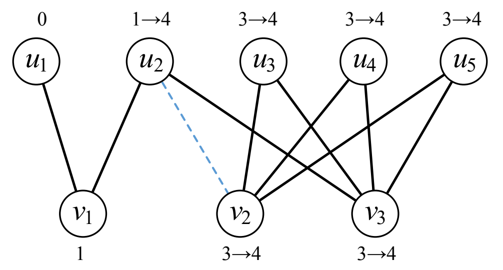


Fig. 1 Change in the core number of each vertex when an edge (u_2, v_2) is inserted.

efficiently. However, storing all (α,β)-cores would cost huge space. Combining these two methods, it makes us wonder whether we can compromise the cost of time and space, and solve the problem of computing (α,β)-core more effectively.

Inspired by the widely used k-core model, we find that the (α,β)-core has a similar nesting property with k-core. Specifically, it is known that a $(k+1)$-core must be a k-core, and similarly, an $(\alpha,\ \beta+1)$-core and an $(\alpha+1,\ \beta)$-core must be an (α,β)-core. Considering the core numbers of vertices in general graph, we can also define the core numbers of vertices in a bipartite graph. Specifically, two kinds of core numbers are defined for each vertex.

Definition 2 (**$\alpha(\beta)$-core number**) Given an integer α, if a vertex $w \in G$ can be in an $(\alpha,\ \beta)$-core but not in an $(\alpha,\ \beta+1)$-core, then the α-core number of w is β, denoted as $core_\alpha(w)=\beta$. Correspondingly, the β-core number of w is defined when β is given and we denote it by $core_\beta(w)$.

Notice that if there is not an $(\alpha,\ \beta)$-core including w given an integer α, then we say $core_\alpha(w)=0$. Similarly, when β is decided and w is not in any $(\alpha,\ \beta)$-core, then $core_\beta(w)=0$. Because $core_\alpha(*)$ and $core_\beta(*)$ are similar, we are going to focus on $core_\alpha(*)$ for the rest of the paper. Specifically, when α is explicit, we can directly represent $core_\alpha(w)$ as $core(w)$ for simplification and call it as the core number of w.

Example 1 In Fig. 1, we take $\alpha=2$ as an example to explain the core number of each vertex before inserting the edge $(u_2,\ v_2)$. Since the degree of u_1 is less than 2, it is not included in any $(2,*)$-core, and hence $core_2(u_1)=0$. For the other vertices, u_3, u_3, u_5, v_2, v_3 can be in a $(2,3)$-core together, so they all have $core_2(*)=3$. At last, because u_2 and v_1 are at most in a $(2,1)$-core, their core number is 1 when $\alpha=2$.

With the definition of $\alpha(\beta)$-core number, we only need to look at the core number of each vertex stored under all possible α (β) to check whether the vertex is in an (α,β)-core or not. For example, if we want to compute a $(2,3)$-core in Fig. 1, we can check the $core_2(*)$ of each vertex in the graph. All the vertices whose core numbers are not less than 3 are in the $(2,3)$-core. Hence, it is easy to figure out the $(2,3)$-core in Fig. 1 is $\{u_3,u_4,u_5,v_2,v_3\}$.

In subsequent sections, we will pay attention to two categories of core computation algorithms, core decomposition and core maintenance. The core decomposition problem is to compute the core number of each vertex in a given bipartite graph G, while the core maintenance problem is to maintain the core numbers of vertices when the graph is changed. In particular, we study the scenarios where a set of edges E_s are either inserted into or deleted from the bipartite graph G, and they are called the incremental and decremental core maintenance problems, respectively.

4 Core Decomposition

In this section, we first solve the core decomposition problem. Specifically, by adapting the algorithm for (α,β)-core computation given in Ref. [9], we present an algorithm for core decomposition in bipartite graphs in Algorithm 1.

For a bipartite graph $G=(U,V,E)$, the value of α ranges from 1 to $deg_{\max}(U)$, since the α value represents the degrees of vertices in U (Line 1). For each α, we calculate the corresponding $core_\alpha(*)$ for all vertices. For each fixed α, the vertices in U whose degrees are less than α are removed along with their adjacent edges. The core numbers of these vertices are set as 0 (Lines 3–5). Then we need to compute the core numbers of rest vertices in V and U, respectively.

In the remaining graph G', the vertices in V are handled in the order of increasing degrees. Let the minimum degree in $V(G')$ be $\beta_{\min}$, all vertices in V whose degrees are not greater than $\beta_{\min}$ and their adjacent edges are deleted. The core numbers of these vertices are set as $\beta_{\min}$ (Lines 7–10). For the vertices in U, once there is a vertex u whose degree is less than α, then we set $core_\alpha(u)=\beta_{\min}$ and remove u

Algorithm 1 ComputeCore (*G*)

```
   Input: An bipartite graph $G=(U,V,E)$
   Output: Core numbers of all vertices in $G$
1  for $\alpha = 1$ to $deg_{\max}(U)$ do
2      $G' \leftarrow G$;
3      while $\exists u \in U$ and $deg_{G'}(u) < \alpha$ do
4          $core_\alpha(u) \leftarrow 0$;
5          remove $u$ and its adjacent edges from $G'$;
6      while $G' \neq \varnothing$ do
7          $\beta_{\min} \leftarrow \min_{v\in G'}\{deg_{G'}(v)\}$;
8          while $\exists v \in V(G')$ and $deg_{G'}(v) \leqslant \beta_{\min}$ do
9              $core_\alpha(v) \leftarrow \beta_{\min}$;
10             remove $v$ and its adjacent edges from $G'$;
11             while $\exists u \in U(G')$ and $deg_{G'}(u) < \alpha$ do
12                 $core_\alpha(u) \leftarrow \beta_{\min}$;
13                 remove $u$ and its adjacent edges from $G'$;
14 return $core(U \cup V)$
```

and its adjacent edges from G', since u cannot be in any (α, β)-core with $\beta > \beta_{\min}$ (Lines 11–13). When G' becomes empty, i.e., all the vertices have been processed, then we obtain the core number of each vertex under the fixed α. While traversing through each possible value of α, the core number of each vertex under each possible α can be computed.

The correctness of Algorithm 1 can be easily obtained by the definition of α-core number. We next show the efficiency of the algorithm.

Theorem 1 Given a bipartite graph $G = (U, V, E)$, the time needed for core decomposition is $O((N + M) \times deg_{\max})$ and the space required to store the core numbers of vertices is $O(N \times deg_{\max})$, where N, M, and $deg_{\max} = \min\{deg_{\max}(U), deg_{\max}(V)\}$ are the total number of vertices, the total number of edges, and the smaller value between maximum degree of vertices in U and maximum degree of vertices in V, respectively.

Proof Each vertex and each edge only need to be traversed once, and hence the time for core decomposition is $O(N + M)$. To be more efficient, we can interchange U and V when $deg_{\max}(U) > deg_{\max}(V)$, then α can be at most $deg_{\max}$ and we will get the time complexity of the algorithm. As for the space required, for each vertex we need to store core numbers for at most $deg_{\max}$ values, and hence the total space used is $O(N \times deg_{\max})$. ■

In reality, the graph may change over time, hence the core numbers of vertices have to be updated after the graph changes. Intuitively, we can execute Algorithm 1 to recompute the core numbers of all vertices. However, though the computation time of core numbers for each fixed α is only linear, it is still unacceptable considering that the graph may have billions of vertices and edges. Hence, we next turn our attention to the core maintenance problem, i.e., updating the core numbers of vertices by avoiding recomputations. We will first present the scenario of a single edge insertion/deletion, and then propose a batch processing approach to further improve the efficiency of core maintenance.

5 Core Maintenance with Single-Edge Insertion/Deletion

In this section, we propose our core maintenance algorithms in a single-edge insertion/deletion scenario. Specifically, we will first present the theoretical basis for core number updates and then give the incremental and decremental core maintenance algorithms.

5.1 Theoretical basis

Given a bipartite graph $G = (U, V, E)$, we consider core maintenance under the scenario that a single edge e is inserted into/deleted from graph G. Before presenting the core maintenance algorithms, we need to solve two problems, quantifying the core number change of vertices and identifying the vertices which will change the core numbers after the insertion/deletion.

As for the k-core, it has been shown that after inserting/deleting an edge, the core number of every vertex can change by at most 1[26]. However, it becomes much more complex when considering bipartite graphs, where the α (β)-core number change is much more than 1. We use the graph in Fig. 1 as an example. When an edge (u_2, v_2) is inserted into the graph, the core number of u_2 changes from 1 to 4. This is because u_2 already has a neighbor v_3 that satisfies $core_2(v_3) > core_2(u_2)$. Hence when u_2 adds another new neighbor with a large core number, $core_2(u_2)$ increases by more than 1. Similarly, it is also possible that the core number of vertices may decrease by more than 1 after deleting an edge.

To overcome this difficulty, we propose the concept of *pre-core* on vertices in U. Interestingly, the core number of each vertex in U can change by at most 1 with respect to their *pre-core* values.

Definition 3 (*pre-core*) Let $G' = (U', V', E')$ be the graph obtained after inserting/deleting an edge $e = (u, v)$ into/from a bipartite graph $G = (U, V, E)$. For a fixed α, the *pre-core* of vertex $u \in U'$ is defined as $\arg\max_{\beta \geqslant 0}\{| \{v \in neigh_{G'}(u) \mid core_\alpha(v) \geqslant \beta\} | \geqslant \alpha\}$.

In Fig. 1, for $\alpha = 2$, because u_2 has 3 neighbors in G' whose core numbers are $core(v_1) = 1$, $core(v_2) = 3$, and $core(v_3) = 3$, so the *pre-core* of u_2 is 3. In subsequence, when we say that the core number of vertex $u \in U'$ changes, we mean it changes with respect to its *pre-core* value.

Lemma 1 Given a bipartite graph $G = (U, V, E)$, let G' be the graph obtained by inserting/deleting an edge $e = (u, v)$ into/from G. For a fixed α, the core number of every vertex changes by at most 1.

Proof We first analyze the case of edge insertion. For any $v_i \in V$, assuming that $core(v_i) = \beta$ in G and it increases by $x > 1$ after the insertion. By Definition 2, we know that v_i has at least $\beta + x$ neighbors in G' whose core numbers are not less than $\beta + x$. Because $core(v_i) = \beta$ before insertion, then v_i has at most β neighbors whose core numbers are not less than $\beta + x$

in G. However, since v_i adds at most one neighbor after inserting an edge (actually only v adds a neighbor), then v_i must need another neighbor $u_1 \in U$ whose core number increases from β to $\beta + x$. As for u_1, its core number increase requires another neighbor v_1 whose core number changes from β to $\beta + x$ (the vertices in U including u have no new neighbors because of the *pre-core*), because u_1 has at most $\alpha - 1$ neighbors in G whose core numbers are not less than $\beta + x$. However, the reason for the increase of $core(v_1)$ is the same as v_i. Since the size of the whole graph is finite, there must be a circle where the core number of vertex w increases because of itself, which is impossible. So the assumption is not correct, which means the core number of each vertex $v \in V$ changes by at most 1. As for any $u_i \in U$, assuming $core(u_i) = \beta'$ before insertion (for u, this core number refers to its *pre-core* value). Now that we know the core number of any vertex in V changes by at most 1, then u_i has at most α neighbors whose core numbers are not less than $\beta' + 1$ after inserting an edge. That is to say, $core(u_i) \leqslant \beta' + 1$ in G', which proves that the core number of any vertex in U increases by at most 1 after insertion.

For the deletion case, suppose there is a vertex w whose core number decreases by x, where $x > 1$. If we add (insert) the deleted edge back, then it causes $core(w)$ to increase by x. Since we have shown that this is not true, the core number of each vertex decreases by at most 1 after deleting an edge. ■

When inserting/deleting an edge $e = (u, v)$ into/from a graph $G = (U, V, E)$, given α, let r denote the vertex with a smaller core number between u and v and $k = core(r)$. If a path is rooted in r and each vertex w in the path satisfies $core(w) = k$, then we call it a C-path of edge e. With this definition, we have the following lemma.

Lemma 2 Given a bipartite graph $G = (U, V, E)$, if we insert/delete an edge $e = (u, v)$ into/from it and G becomes G', then only vertices w in a C-path of e can change their core numbers.

Proof Without loss of generality, we assume that $k = core(u) \leqslant core(v)$ and analyze how u affects other vertices. For each $w \in neigh_{G'}(u)$, we analyze the core number change of w in two cases. If $core(w) > k$, then u cannot be in an $(\alpha, k + 2)$-core with w, as its core number increases by at most 1 as stated in Lemma 1, so u can not help w to increase w's core number. If $core(w) < k$, then the inserted edge also cannot cause $core(w)$ to increase, because u has already been the vertex in G that supports the current core number of w. To sum up, only the vertices in the C-path of e are affected by u, which means their core numbers may increase.

Similarly, we can easily show that removing an edge only decreases the core numbers of those vertices in the C-path of e. ■

With Lemmas 1 and 2, we can start from either vertex u or v and identify those vertices whose core numbers may change when an edge is inserted/deleted. Once all these vertices have been identified, we will increase/decrease their core numbers by 1. Based on these observations, we next present core maintenance algorithms for scenarios of single-edge insertion and deletion, respectively.

5.2 Incremental core maintenance

Algorithm. The detailed algorithm to maintain the core number of each vertex after inserting an edge is given in Algorithm 2. After inserting (u, v) into G, it iterates over all values of α from 1 to $deg_{G'}(u)$, since the largest value of α that u can have in an (α, β)-core is its own degree (Lines 1 and 2). We set Q and C to preserve the vertices to be checked and the vertices whose core numbers might increase, respectively (Line 3). To prevent repetitive processing of a vertex, we set *visited*[ ] to mark the vertices that have been visited (Line 4). For each possible α, we aim to find the vertices in the graph whose core numbers change after inserting an edge and then increase their core numbers by 1.

According to Lemma 1, we need to compute the $pre\text{-}core_\alpha(u)$, to make sure that $core_\alpha(u)$ changes by at most 1 (Line 5). We will set r be the vertex with the smaller core number between u and v and add it to Q (Lines 6–8). When a vertex x is ejected from Q, we insert it to C and set *visited*[x] as *true* (Lines 9–11). For each vertex x in Q, we are going to check its each neighbor y, to see if u's core number is likely to increase. If $core_\alpha(y) > k$ or y is in C, then y must support the increase of $core(x)$, because y is already in an $(\alpha, k + 1)$-core or $core_\alpha(y)$ is supposed to increase from k to $k + 1$. Besides that, if $core_\alpha(y) = k$ and is not visited, then y is also likely to help $core_\alpha(x)$ to increase. So when the neighbor y of x satisfies one of these two conditions, $SN(x)$ increases by 1 (Lines 12–16). When $SN(x)$ is larger than the threshold k, it means that it is possible for $core_\alpha(x)$ to increase. Based on Lemma 2, each neighbor y of x with $core_\alpha(y)$= $core_\alpha(x)$ and not visited is pushed into Q for further

Algorithm 2 AnEdgeInsert $(G, e, core\ (U \cup V))$

```
Input: A bipartite graph $G = (U, V, E)$, an edge $e = (u, v)$
       to be inserted and $core(U \cup V)$ is the core number
       of each vertex under different $\alpha$
Output: The updated core number for each vertex
1  $G' \leftarrow$ insert $(u, v)$ into $G$;
2  for $\alpha = 1$ to $deg_{G'}(u)$ do
3      $Q \leftarrow$ empty queue; $C \leftarrow \varnothing$;
4      $visited[w] \leftarrow false$ and $SN(w) = 0$ for each
       $w \in (U(G') \cup V(G'))$;
5      $core_\alpha(u) \leftarrow pre\text{-}core_\alpha(u)$;
6      if $core_\alpha(u) \leqslant core_\alpha(v)$ then $r \leftarrow u$;
7      else $r \leftarrow v$;
8      $Q.push(r)$; $k = core_\alpha(r)$;
9      while $Q \neq \varnothing$ do
10         $x \leftarrow Q.pop()$; $Q_1 \leftarrow Q$;
11         $C.insert(x)$; $visited[x] \leftarrow true$;
12         for $y \in neigh_{G'}(x)$ do
13             if $core_\alpha(y) > k$ or $y \in C$ then $SN(x)++$;
14             else if $core_\alpha(y) = k$ and not $visited[y]$ then
15                 $SN(x)++$;
16                 if $y \notin Q_1$ then $Q_1.push(y)$;
17         if $x \in V(G')$ and $SN(x) > k$ or $x \in U(G')$ and
           $SN(x) \geqslant \alpha$ then
18             $Q \leftarrow Q_1$;
19         else
20             DfsDelete $(x, C, G', core_\alpha(*), \alpha)$;
21     for each $w \in C$ do
22         $core_\alpha(w)++$;
23 return $core(U' \cup V')$
24 Precedure DfsDelete $(x, C, G, core, \alpha)$:
25     $C$.remove$(x)$;
26     for $w \in neigh_G(x)$ do
27         if $w \in C$ and $core(w) = core(x)$ then
28             $SN(w)--$;
29             if $w \in V$ and $SN(w) \leqslant core(w)$ or $w \in U$ and
               $SN(w) < \alpha$ then
30                 DfsDelete $(w, C, G, core, \alpha)$;
```

checking (Lines 17 and 18). Once $SN(x)$ is not enough to support the increase in the core number of x, the procedure DfsDelete, which is DFS process, is invoked to delete the vertices from C that cannot increase their core numbers (Lines 19 and 20). While Q becomes empty, we have dealt with all the vertices whose core number might change. Finally, all the vertices in C that have not been deleted will increase their core numbers by 1 (Lines 21 and 22).

The procedure DfsDelete removes the vertex x from C, since it does not satisfy the increasing condition (Line 25). For each neighbor w of x, if it is in C and $core(w) = core(x)$, we will reduce the $SN(w)$ by 1, since y has lost a neighbor x to support its core number increase (Lines 26–28). When w does not have enough neighbors to hold its core number increase, DfsDelete will be invoked again to recursively remove w (Lines 29 and 30).

Performance analysis. We will analyze the correctness and efficiency of Algorithm 2. Firstly, some notations are defined, which will be used in measuring the time complexity of the algorithm.

When an edge $e=(u,v)$ is inserted into the graph $G=(U,V,E)$, for each α, let U_α^1 and V_α^1 be the set of vertices whose core numbers are equal to $core_\alpha(e)$ in U and V, respectively, where $core_\alpha(e)=\min\{core_\alpha(u), core_\alpha(v)\}$. Let $U_I = \max_{1\leqslant\alpha\leqslant deg_{G'}(u)} U_\alpha^1$ and $V_I = \max_{1\leqslant\alpha\leqslant deg_{G'}(v)} V_\alpha^1$. E_α^1 is the set of edges in the subgraph induced by U_α^1 to V_α^1. Let $E_I = \max_{1\leqslant\alpha\leqslant deg_{G'}(u)} E_\alpha^1$.

Let $N_U^\alpha = \max_{u_i\in U_\alpha^1}\{SN_0(u_i) - \alpha, 0\}$, where $SN_0(u_i)$ denotes the SN value of u_i when u_i is first processed. Let $N_U = \max_{1\leqslant\alpha\leqslant deg_{G'}(u)} N_U^\alpha$. Similarly, we define $N_V^\alpha = \max_{v_i\in V_\alpha^1}\{SN_0(v_i) - core_\alpha, 0\}$ and $N_V = \max_{1\leqslant\alpha\leqslant deg_{G'}(u)} N_V^\alpha$.

Theorem 2 When inserting an edge $e = (u, v)$ into graph G, Algorithm 2 can correctly update the core numbers of all vertices and the time complexity is $O(deg_{G'}(u) \times (E_I + U_I \times N_U + V_I \times N_V))$.

Proof For each α, Algorithm 2 first finds the vertices whose core numbers change and then increases their core numbers by 1. In the algorithm, the core numbers of u and v are compared and the one with smaller core number is labeled as r. The algorithm traverses from r and pushes all vertices w with $core_\alpha(w) = core_\alpha(r)$ that are reachable from r through a C-path to Q for further identification. According to Lemma 2, only those vertices may change their core numbers.

If $core_\alpha(w) = core_\alpha(r) = k$ and $core_\alpha(w)$ increases after insertion, then w must satisfy one of the following conditions: (1) w has a new neighbor whose core numbers are at least $k + 1$. (2) Some neighbors of w have their core numbers increased from k to $k + 1$. In the algorithm, $SN(w)$ records the number of these neighbors that support w's core number increase. If $SN(w) \leqslant core_\alpha(w)$, it is obvious that w's core number will not increase. Once a vertex is identified not to change its core number, the SN values of other vertices are updated by invoking DfsDelete. After all the vertices whose core numbers might change have been processed, the subgraph induced by the vertices in C that are not deleted constitutes an $(\alpha, k+1)$-core, as the *SN* value of these vertices is at least $k+1$. Then by Lemma 1, all these vertices will increase the core number by 1. Hence, the algorithm can correctly update the core numbers of vertices.

Next, we analyze the time complexity. Firstly, there are $deg_{G'}(u)$ iterations in the algorithm execution. For each iteration, since only the vertices w with $core(w) = k$ are reachable from r through a C-path, and will be processed, so the number of vertices and edges that the algorithm visits are at most $U_I + V_I$ and E_I, respectively. For each vertex, except for the first calculation of the *SN* value when ejected from Q, the subsequent visits will decrease its *SN* value by 1. Therefore, the vertex $u \in U$ $(v \in V)$ can be visited at most N_U^α (N_V^α) times for a certain α, since it will be removed when its SN is smaller than α $(core_\alpha)$. Then we can see that the vertex $u \in U$ $(v \in V)$ will be visited at most N_U (N_V) times in any iteration. To sum up, the time complexity of Algorithm 2 can be obtained. ■

Example 2 Here is an example to illustrate the execution of Algorithm 2 in Fig. 1 at $\alpha = 2$. We first insert (u_2, v_2) into the graph and compute $pre\text{-}core(u_2) = 3$ and $k = 3$. Since $core(u_2) = core(v_2)$, without loss of generality, let $r = u_2$. We push u_2 into Q, and then set $visited[u_2] = true$ to avoid repeated visits. We can easily get that $SN(u_2) = 2$, because the core numbers of its neighbors v_2 and v_3 are both 3 and not visited. Since $SN(u_2) \geqslant \alpha$ and $core(v_2) = core(v_3) = k$, we will next push v_2 and v_3 to Q. As for v_2, it can be obtained that $SN(v_2) = 4 > k$, so v_2 will not be deleted, and we are going to process v_2's neighbors. When all the vertices in Q are processed, we find that the vertices that are still in C are $\{u_2, v_2, u_3, v_3, u_4, u_5\}$. As a result, we increase their core numbers by 1, to give the value 4 and Algorithm 2 terminates.

5.3 Decremental core maintenance

Similar to Algorithm 2 of core maintainance after single-edge insertion, we here give the core maintenance algorithm after deleting an edge.

Algorithm. The detailed algorithm to update the core number of each vertex after removing an edge is given in Algorithm 3. After deleting (u, v) from G, it iterates over all possible values of α from 1 to $deg_G u$, since the largest value of α can affect the (α, β)-core while deleting edge (u, v) is the degree of u in G (Lines 1 and 2).

Algorithm 3 **AnEdgeDelete** $(G, e, core\ (U \cup V))$

```
Input: A bipartite graph $G = (U, V, E)$, an edge $e = (u, v)$
       to be deleted, and $core(U \cup V)$ is the core number of
       each vertex under different $\alpha$
Output: The updated core number for each vertex
1  $G' \leftarrow$ delete $(u, v)$ from $G$;
2  for $\alpha = 1$ to $deg_G(u)$ do
3      $Q \leftarrow$ empty queue; $C \leftarrow \varnothing$;
4      $visited[w] \leftarrow false$ and $SN(w) = 0$ for each
       $w \in (U(G') \cup V(G'))$ ;
5      if $core_\alpha(u) \leqslant core_\alpha(v)$ then
6          $Q.push(u)$; $k = core_\alpha(u)$;
7      Lines 6–7 by replacing $u$ with $v$;
8      while $Q \neq \varnothing$ do
9          $x \leftarrow Q.pop()$;
10         $visited[x] \leftarrow true$;
11         for each $y \in neigh_{G'}(x)$ do
12             if $core_\alpha(y) \geqslant k$ and $y \notin C$ then $SN(x)++$;
13         if $x \in V(G')$ and $SN(x) < k$ or $x \in U(G')$ and
           $SN(x) < \alpha$ then
14             DfsUpdate $(x, C, G', core_\alpha(*), \alpha, visited[], Q)$;
15     for each $w \in C$ do
16         $core_\alpha(w)--$;
17     if $core_\alpha(u) > pre\text{-}core_\alpha(u)$ then
18         $core_\alpha(u) \leftarrow pre\text{-}core_\alpha(u)$;
19 Procedure DfsUpdate $(x, C, G, core, \alpha, visited[], Q)$:
20 $C.insert(x)$;
21 for $w \in neigh_G(x)$ do
22     if not $visited[w]$ and $core(w) = core(x)$ and $w \notin Q$
       then $Q.push(w)$;
23     if $visited[w]$ and $w \notin C$ and $core(w) = core(x)$ then
24         $SN(w)--$;
25         if $w \in V$ and $SN(w) < core(x)$ or $w \in U$ and
           $SN(w) < \alpha$ then
26             DfsUpdate $(w, C, G, core, \alpha, visited[], Q)$
```

In each iteration of α, the algorithm process is similar to Algorithm 2. The difference is that it compares $core_\alpha(u)$ and $core_\alpha(v)$ to determine r instead of computing $pre\text{-}core_\alpha(u)$ (Lines 5–7). When dealing with each vertex x in Q, it computes $SN(x)$ value by counting x's neighbors whose core numbers are not less than k and are not in C (Lines 8–12). Once $SN(x)$ is not enough to keep the current core number of x, the procedure DfsUpdate is invoked to record the vertices whose core numbers are going to decrease (Lines 13 and 14). Finally, all the vertices in C that have not been deleted will decrease their core numbers by 1 (Lines 15 and 16). Specifically, if $core_\alpha(u)$ after deletion is greater than $pre\text{-}core_\alpha(u)$, we will set the $core_\alpha(u)$ as *pre-core*$_\alpha(u)$, since the $core(u)$ is at most *pre-core*(u) according to Definition 3 (Lines 17 and 18).

The DfsUpdate first adds x to C and then checks its each neighbor (Lines 20 and 21). For those vertices w with $core(w) = core(x)$, we will discuss them in two cases. If w is not visited, then it will be pushed into Q for further detection (Line 22). Otherwise, if it has been visited but not in C, we decrease $SN(w)$ by 1. Once $SN(w)$ is less than α or $core(x)$, the DfsUpdate will be recursively called (Lines 23–26).

Performance analysis. We will analyze the correctness and efficiency of Algorithm 3. Similar to Algorithm 2, we first give some useful notations.

Given a graph $G = (U, V, E)$, we delete an edge $e = (u, v)$ from it. For each α, let U_α^2 and V_α^2 be the set of vertices whose core numbers are equal to $core_\alpha(e)$ in U and V, respectively, where $core_\alpha(e) = \min\{core_\alpha(u), core_\alpha(v)\}$. Let $U_D = \max_{1 \leqslant \alpha \leqslant deg_G(u)} U_\alpha^2$ and $V_D = \max_{1 \leqslant \alpha \leqslant deg_G(v)} V_\alpha^2$. Let E_α^2 be the set of edges in the subgraph induced by U_α^2 to V_α^2. Let $E_D = \max_{1 \leqslant \alpha \leqslant deg_G(u)} E_\alpha^2$.

Let $\tilde{N}_U^\alpha = \max_{u_i \in U_\alpha^2}\{SN_0(u_i) - \alpha, 0\}$, where $SN_0(u_i)$ denotes the SN value of u_i when u_i is first processed. Let $\tilde{N}_U = \max_{1 \leqslant \alpha \leqslant deg_G(u)} \tilde{N}_U^\alpha$. Similarly, we define $\tilde{N}_V^\alpha = \max_{v_i \in V_\alpha^2}\{SN_0(v_i) - core_\alpha, 0\}$ and $\tilde{N}_V = \max_{1 \leqslant \alpha \leqslant deg_G(u)} \tilde{N}_V^\alpha$.

Using the notations above and an analysis similar to the single-edge insertion case, we can easily obtain the following theorem.

Theorem 3 When deleting an edge $e = (u, v)$ from graph G, Algorithm 3 can update the core numbers of vertices in $O(deg_G(u) \times (E_D + U_D \times \tilde{N}_U + V_D \times \tilde{N}_V))$ time.

6 Core Maintenance with Multiple-Edge Insertion/Deletion

In this section, we consider a more general scenario where multiple edges are inserted/deleted, and present batch processing algorithms to handle multiple-edge insertion/deletion.

As discussed before, the core number change becomes much more complex when multiple edges are inserted/deleted together, because the core number change of a vertex can be affected by multiple inserted/deleted edges. In the case of multiple-edge insertion/deletion, it is hard to determine the exact change of the core number value and to identify the set of vertices that may change the core numbers.

Let us consider core number of each vertex in Fig. 1 at $\alpha = 1$, assuming (u_2, v_2) has been inserted. It can be easily concluded that $core(u_1) = core(v_1) = 2$ and all the other vertices have core numbers of 4. If we remove the edge set $E_s = \{(u_2, v_1), (u_2, v_2), (u_3, v_2), (u_4, v_3), (u_5, v_3)\}$ from the graph, then it is difficult to tell which vertices have their core numbers changed. In fact, core numbers of all the vertices have changed. In addition, we can know that each vertex in U has *pre-core* value that is equal to its original core number. However, there are many vertices whose core numbers decrease by more than 1, such as $core(u_3)$, which goes from 4 to 2 after deletion.

To overcome these difficulties, we here propose a structure of inserted/deleted edges, called V-independent edge set, to solve the challenge of determining the core number change of vertices, by showing that the insertion/deletion of a *VES* can cause the core number of each vertex to change by at most 1. Based on this observation, we propose our batch processing algorithms, which divides the inserted/deleted edges into multiple *VES*s and handles these *VES*s iteratively.

In subsequence, we first introduce the structure of V-independent edge set and give some theoretical results, and then present the core maintenance algorithms for both incremental and decremental scenarios.

6.1 Theoretical basis

Given a set of edges $E_i = (e_1, e_2, ..., e_p)$, where $e_i = (u_i, v_i)$, let U_c denote the set of all vertices $u_i \in U$ connected to E_i.

Definition 4 (V-independent edge set (*VES*)) Given a bipartite graph $G = (U, V, E)$, let $G' = (U', V', E')$ be the graph after inserting/deleting a set of edges E_s into/from G. If an edge set $E_i \subseteq E_s$ satisfies that for any vertex in V', it has at most one neighbor in U_c, then we call E_i as a V-independent edge set.

Here we give an example to explain the definition of the V-independent edge set. A bipartite graph is shown in Fig. 2, where dotted lines represent

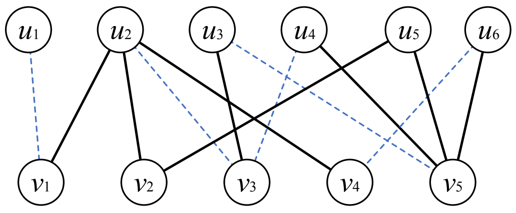


Fig. 2 An example for V-independent edge set.

edges in the graph that need to be updated (inserted/deleted). According to the definition above, $E_i = \{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_4, v_3), (u_6, v_4)\}$ and $U_c = \{u_1, u_2, u_3, u_4, u_6\}$. Let $E' = \{(u_1, v_1), (u_2, v_3), (u_3, v_5), (u_6, v_4)\}$, then it is easy to conclude that E' is a *VES* because each vertex in $\{v_1, v_3, v_4, v_5\}$ has at most one neighbor in U_c. Next, we prove that if inserting/deleting a *VES* into/from graph G, the core number of each vertex changes by at most 1 (for some vertices $u \in U$, their core numbers change with respect to their *pre-core* value).

Lemma 3 Let $G' = (U', V', E')$ be the graph obtained by inserting/deleting a V-independent edge set E_i into/from graph $G = (U, V, E)$. Then for each $w \in (U' \cup V')$, $core(w)$ can change by at most 1.

Proof We first analyze the insertion of edges, and discuss the vertices in U' and V', respectively. For each vertex $v \in V'$, we assume that $core(v)$ increases from β to $\beta + x$ after inserting a *VES*, where $x \geqslant 1$. According to Definition 4, v adds at most one new neighbor u_i in G'. So v still needs another neighbor u_1 whose core number goes from β to $\beta + x$ in G'. As for u_1, it has at most $\alpha - 1$ neighbors in G whose core numbers are not less than $\beta + x$ and has no new neighbors because of *pre-core*, so its increasing in core number requires another neighbor v_1 whose core number changes from β to $\beta + x$. However, the reason for the increase of $core(v_1)$ is the same as v_i. Similar to the analysis in Lemma 1, we can prove that this assumption is wrong, which means the core number of each vertex $v \in V$ changes by at most 1. As for any $u_i \in U'$, assume $core(u_i) = \beta'$ before insertion (for some vertices in U', it refers to its *pre-core* value). Since the core number of any vertex in V' changes by at most 1, then u_i has at most α neighbors whose core numbers are not less than $\beta' + 1$ after inserting an edge. That is to say, $core(u_i) \leqslant \beta' + 1$ in G', which proves that the core number of any vertex in U' increases by at most 1 after inserting a *VES* into the graph.

In the case of edge deletion, if there is a vertex w whose core number can reduce by x, where $x > 1$. When we add back the deleted edges to G', then $core(w)$ will increase by x after inserting a *VES*, which contradicts the previous conclusion. So it is easy to say that the core numbers of all vertices change by at most 1 when deleting a V-independent edge set. ■

6.2 Incremental core maintenance

Algorithm. The detailed algorithm to maintain each vertex's core number with the insertion of multiple edges is given in Algorithm 4. It is executed until all edges in E_s have been processed (Line 1). In each iteration, the algorithm calls the subroutine FindInsertEdges to find a V-independent edge set E_i and deletes it from E_s (Lines 2–4). For each possible α, the InsertChangedSet algorithm is invoked to find the vertices whose core numbers will increase by 1 after insertion and change their core numbers (Lines 5–9).

Algorithm 5 aims to find a V-independent edge set E_i from all unprocessed edges in E_s. Basically, it sets two parameters, *addN*[ ] marking the connection between a

Algorithm 4 MultiEdgesInsert ($G, E_s, core\ (U \cup V)$)

```
Input: A bipartite graph $G = (U, V, E)$, a set of edges $E_s$ to
       be inserted, and $core(U \cup V)$ is the core number of
       each vertex under different $\alpha$
Output: The updated core number for each vertex
1  while $E_s \neq \varnothing$ do
2      $E_i \leftarrow$ FindInsertEdges($G, E_s$);
3      $G' \leftarrow$ insert $E_i$ into $G$ ;
4      delete $E_i$ from $E_s$;
5      for $\alpha = 1$ to max$D_i$ do
6          $V_{changed} \leftarrow \varnothing$;
7          $V_{changed} \leftarrow$ InsertChangedSet($G', E_i, core_\alpha(*), \alpha$);
8          for each $w \in V_{changed}$ do
9              $core_\alpha(w) \leftarrow core_\alpha(w) + 1$;
```

Algorithm 5 FindInsertEdges (G, E_s)

```
Input: A bipartite graph $G = (U, V, E)$, a set of edges $E_s$ to
       be inserted
Output: A $V$-independent edge set of $E_s$
1  $E_i \leftarrow \varnothing$; $U_c \leftarrow \varnothing$;
2  for each $v \in V$ do
3      addN[$v$] $\leftarrow 0$;
4  flag = false;
5  for each $e = (u, v) \in E_s$ do
6      flag $\leftarrow$ true;
7      if $u \notin U_c$ then
8          for each $v_i \in neigh(u)$ and $v$ do
9              if addN[$v_i$] $\neq 0$ or addN[$v$] $\neq 0$ then
10                 flag = false;
11                 break;
12     else if addN[$v$] $\neq u$ and addN[$v$] $\neq 0$ then
13         flag = false;
14     if flag then
15         add $e$ into $E_i$;
16         if $u \notin U_c$ then
17             $U_c$.insert($u$); addN[$v$] = $u$;
18             set addN[$w$] = $u$ for each $w \in neigh(u)$;
19 return $E_i$
```

node in V and the selected edges, and *flag* indicating whether an edge is selected or not (Lines 2–4). For each endpoint u of edge $e = (u, v)$ in E_s, there are two cases for the algorithm execution. When u is not in U_c, then it checks if u's neighbors and v are already connected to other vertices in U_c (Lines 7 and 8). Once any vertex is found whose *addN*[ ] is not 0, the algorithm will set *flag* to *false* indicating this edge cannot be selected (Lines 9–11). Similarly, when u has been in U_c, the algorithm will check if v is already connected to other vertices in U_c (Lines 12 and 13). When *flag* is *true*, the edge e will be added into E_i (Lines 14 and 15). If u is not in U_c, it will add u to U_c and update $addN[\,] = u$ for each neighbor of u and v (Lines 16–18).

After the *VES* is found, Algorithm 6 is invoked to find all vertices whose core numbers will change

Algorithm 6 InsertChangedSet ($G, E_i, core, \alpha$)

```
Input: A bipartite graph $G = (U, V, E)$, a set of edges $E_i$ to
       be inserted, and core is the core number of each
       vertex under a particular $\alpha$
Output: The set of vertices whose core number changes
        after inserting the edge set $E_i$.
1  $Q \leftarrow$ empty queue;
2  $C \leftarrow \varnothing$;
3  if $Q \neq \varnothing$ then
4      for each $w \in (U \cup V)$ do
5          $visited[w] \leftarrow false$;
6          $SN(w) = 0$;
7  for each $(u_i, v_i) \in E_i$ do
8      $core(u_i) \leftarrow pre\text{-}core(u_i)$;
9      if $core(u_i) \leqslant core(v_i)$ and $u_i \notin Q$ then
10         $Q.push(u_i)$
11     else if $core(v_i) < core(u_i)$ and $v_i \notin Q$ then
12         $Q.push(v_i)$
13 while $Q \neq \varnothing$ do
14     $x \leftarrow Q.pop()$; $Q_1 \leftarrow Q$;
15     $C.insert(x)$;
16     $visited[x] \leftarrow true$;
17     for each $y \in neigh(x)$ do
18         if $core(y) > core(x)$ or $core(y) = core(x)$ and
           $y \in C$ then
19             $SN(x) + +$;
20         else if $core(y) = core(x)$ and not $visited[y]$ then
21             $SN(x) + +$;
22             if $y \notin Q_1$ then $Q_1.push(y)$;
23     if $x \in V$ and $SN(x) > core(x)$ or $x \in U$ and
       $SN(x) \geqslant \alpha$ then
24         $Q \leftarrow Q_1$;
25     else DfsDelete $(x, C, G, core, \alpha)$;
26 return $C$
```

after insertion. Similar to Algorithm 2, we first do the initialization (Lines 1–6). For each edge (u_i, v_i) to be inserted, it computes the *pre-core* of u_i and then adds the endpoint with the smaller core number to Q (Lines 7–12). When a vertex x is ejected from Q, the algorithm inserts it into C and sets *visited*[x] as *true* (Lines 14–16). For each vertex x in Q, each of its neighbors y is checked to see if x's core number is likely to increase. The calculation of *SN* value is the same as the case of single-edge insertion (Lines 17–22). If $SN(x)$ reaches the threshold, i.e., $SN(x)$ is greater than $core(x)$ when x in V or is not less than α when x in U, then each neighbor y of x which satisfies $core_\alpha(y) = core_\alpha(x)$ and is not visited is pushed into Q for further checking (Lines 23 and 24). Otherwise, the procedure DfsDelete is invoked to remove x from C, since it is impossible for $core(x)$ to increase (Line 25).

Performance analysis. To analyze the time complexity of the algorithm, we first provide some useful notations. Let $\varDelta_I$ be the maximum number of edges inserted for each vertex in V. By the definition of *VES*, it is easy to see that the inserted edges can be divided into $\varDelta_I$ V-independent edge sets. When a *VES* E_i is inserted into graph $G = (U, V, E)$, the maximum degree of all vertices in U_c is denoted by $\max D_i$ and $\max D_I = \max_{1 \leqslant i \leqslant \varDelta_I} \max D_i$. For each α, let U_α^3 and V_α^3 be the set of vertices whose core numbers are equal to $core(e_i)$ in U and V, respectively, where $e_i = (u_i, v_i) \in E_i$ and $core_\alpha(e_i) = \min\{core_\alpha(u_i), core_\alpha(v_i)\}$. Let $\tilde{U_I} = \max_{1 \leqslant \alpha \leqslant \max D_I} U_\alpha^3$ and $\tilde{V_I} = \max_{1 \leqslant \alpha \leqslant \max D_I} V_\alpha^3$. Let E_α^3 be the set of edges in the subgraph induced by U_α^3 to V_α^3, and let $\tilde{E}_I = \max_{1 \leqslant \alpha \leqslant \max D_I} E_\alpha^3$.

Let $M_U^\alpha = \max_{u_i \in U_\alpha^3}\{SN_0(u_i) - \alpha, 0\}$, where $SN_0(u_i)$ denotes the *SN* value of u_i when u_i is processed for the first time. Let $M_U = \max_{1 \leqslant \alpha \leqslant \max D_I} M_U^\alpha$. Similarly, we define $M_V^\alpha = \max_{v_i \in V_\alpha^3}\{SN_0(v_i) - core_\alpha, 0\}$ and $M_V = \max_{1 \leqslant \alpha \leqslant \max D_I} M_V^\alpha$.

The following theorem gives the time complexity of Algorithm 4 and proves its correctness.

Theorem 4 Algorithm 4 can correctly update the core numbers of all vertices after inserting an edge set E_s in $O(\varDelta_I \times (\max D_I \times ((\tilde{E_I} + \tilde{U_I} \times M_U + \tilde{V_I} \times M_V))))$ time.

Proof Algorithm 4 is executed in iterations and each iteration includes two parts. The first part finds a V-independent edge set from all unprocessed edges in E_s by executing Algorithm 5. According to Lemma 3, each vertex can change its core number by 1 after inserting a V-independent edge set into the graph.

As for the second part, we invoke Algorithm 6 to identify the vertices whose core number can increase after insertion for each α. It starts from the inserted edges and adds each endpoint with the smaller core number to Q for further identification. Similar to Algorithm 2, the algorithm uses $SN(w)$ to record the number of neighbors that support w's core number increase. If $SN(w) \leqslant core_\alpha(w)$, it is obvious that w's core number will not increase. Once a vertex is identified not to change its core number, the *SN* values of other vertices are updated by invoking DfsDelete. After all the vertices whose core numbers might change have been processed, the vertices in C that are not deleted will increase their core numbers, as these vertices have enough neighbors to support them in an (α, β)-core with a larger β value. When all edges in E_s are handled, the potential vertices are visited and the ones that cannot increase core numbers are removed. All of the above ensure the correctness of Algorithm 4.

As for the time complexity, it is similar to the case of single-edge insertion. The difference is that Algorithm 4 has to deal with multiple edges in Δ_I batches. For each batch, the algorithm needs to iterate at most max D_i times. Based on Lemma 2, when inserting an edge $e = (u, v)$ (assuming $core_\alpha(u) \leqslant core_\alpha(v)$), only the vertices w with $core(w) = core(u)$ are reachable from u through a C-path, and can change their core numbers. Therefore, when inserting a *VES* into the graph, the number of vertices and edges visited by Algorithm 4 is at most $U_I + V_I$ and E_I, respectively. For each vertex, except for the first *SN* value calculation when ejected from Q, all the subsequent visits will decrease its *SN* value by 1. So for the vertex $u \in U (v \in V)$, it can be visited by at most $N_u (N_v)$ times, since it will be removed from C when its *SN* is smaller than α ($core_\alpha$). Then we can know that the vertex $u \in U$ ($v \in V$) will be visited at most N_U (N_V) times in any iteration. Therefore, it can be concluded that the time complexity of the Algorithm 4 is the same as stated in Theorem 4. ■

Discussion. A point we would like to emphasize is that the batch processing algorithm can greatly reduce redundant computations. This is because when processing the insertion of a VES, once a vertex is determined to increase its core number, it is unnecessary to visit this vertex again, as it will not increase the core number anymore. We illustrate this observation with the graph in Fig. 1. Assuming that $E_s = \{(u_2, v_1), (u_2, v_2), (u_2, v_3)\}$ is the edge set that needs to deal with. If we insert these edges one by one using Algorithm 2, then we need to visit vertex u_2 three times, vertex u_1 and v_1 two times, and the rest vertices once. However, since the E_s satisfies the conditions of V-independent edge set, so we can process these edges using one iteration and visit all vertices in U and V only once. Thus, the time consumed is significantly reduced.

6.3 Decremental core maintenance

Algorithm. The detailed algorithm to maintain each vertex's core number when multiple edges are deleted is given in Algorithm 7. It is executed until all edges in E_s have been processed (Line 1). In each iteration, the algorithm calls the subroutine FindDeleteEdges to find a V-independent edge set E_i of E_s (Line 2). Then for each α, the DeleteChangedSet algorithm is invoked to find the vertices whose core numbers will decrease by 1 after deleting E_i and set their core numbers (Lines 5–9). Specifically, for those vertices in U connected to E_i, their current core numbers are compared with *pre-core*, because deleting E_i may cause their core numbers to change by more than 1 (Lines 10–12).

Algorithm 8 finds a *VES* from the unprocessed edges in E_s. The algorithm is similar to Algorithm 5 except that for each edge $e = (u, v)$, we need not to deal with v, because v is no longer a neighbor of u after deletion. When finding the vertices whose core number changes in Algorithm 9, the difference is that it compares $core_\alpha(u)$ with $core_\alpha(v)$ instead of computing $pre\text{-}core_\alpha(u)$ for each edge $e = (u, v)$ (Lines 3–6). When dealing with each vertex x in Q, the process of computing $SN(x)$ value is similar to Algorithm 3 (Lines 11–17). Once the $SN(x)$ is not enough to keep the current core number

Algorithm 7 MultiEdgesDelete (G, E_s, *core* ($U \cup V$))

```
Input: A bipartite graph $G = (U, V, E)$, a set of edges $E_s$ to
       be deleted, and core ($U \cup V$) is the core number of
       each vertex under different $\alpha$
Output: The updated core number for each vertex
1  while $E_s \neq \varnothing$ do
2      $E_i \leftarrow$ FindDeleteEdges($G, E_s$);
3      $G' \leftarrow$ delete $E_i$ from $G$;
4      delete $E_i$ from $E_s$;
5      for $\alpha = 1$ to max$D_d$ do
6          $V_{changed} \leftarrow \varnothing$;
7          $V_{changed} \leftarrow$ DeleteChangedSet ($G', E_i, core_\alpha(*), \alpha$);
8          for each $w \in V_{changed}$ do
9              $core_\alpha(w) \leftarrow core_\alpha(w) - 1$;
10         for each $u_i \in U_c$ do
11             if $core_\alpha(u_i) >$ $pre\text{-}core_\alpha(u_i)$ then
12                 $core_\alpha(u_i) = pre\text{-}core_\alpha(u_i)$;
```

Algorithm 8 FindDeleteEdges (G, E_s)

```
Input: A bipartite graph $G = (U, V, E)$, a set of edges $E_s$ to be deleted
Output: A neighbor edge set of $E_s$
1  $E_i \leftarrow \varnothing$; $U_c \leftarrow \varnothing$;
2  for each $v \in V$ do
3      $remN[v] \leftarrow 0$;
4  $flag = false$;
5  for each $e = (u, v) \in E_s$ do
6      $flag \leftarrow true$;
7      if $u \notin U_c$ then
8          for each $v_i \in neigh(u)$ do
9              if $remN[v_i] \neq 0$ then
10                 $flag = false$;
11                 break;
12     if $flag$ then
13         add $e$ into $E_i$;
14         if $u \notin U_c$ then
15             $U_c.insert(u)$;
16             set $remN[w] = u$ for each $w \in neigh(u)$;
17 return $E_i$
```

Algorithm 9 DeleteChangedSet ($G, E_i, core, \alpha$)

```
Input: A bipartite graph $G = (U, V, E)$, a set of edges $E_i$ to be deleted, and $core$ is the core number of all vertices under a particular $\alpha$
Output: The set of vertices whose core number changes after deleting the edge set $E_i$.
1  $Q \leftarrow$ empty queue;
2  $C \leftarrow \varnothing$;
3  for each $(u_i, v_i) \in E_i$ do
4      if $core(u_i) \leqslant core(v_i)$ and $u_i \notin Q$ then
5          $Q.push(u_i)$
6      Lines 4–5 by replacing $u_i$ with $v_i$;
7  if $Q \neq \varnothing$ then
8      for each $w \in (U \cup V)$ do
9          $visited[w] \leftarrow false$;
10         $SN(w) = 0$;
11 while $Q \neq \varnothing$ do
12     $x \leftarrow Q.pop()$;
13     $visited[x] \leftarrow true$;
14     for each $y \in neigh(x)$ do
15         if $core(y) > core(x)$ then $SN(x) + +$;
16         else if $core(y) = core(x)$ and $y \notin C$ then
17             $SN(x) + +$;
18     if $x \in V$ and $SN(x) < core(x)$ or $x \in U$ and SN$(x) < \alpha$ then
19         DfsUpdate $(x, C, G, core, \alpha, visited[], Q)$;
20 return $C$
```

of x, the procedure DfsUpdate is invoked to record $core_\alpha(x)$ that is going to decrease (Lines 18 and 19).

Performance analysis. Firstly, we will give some notations that are similar to the case of inserting multiple edges and then the correctness and time complexity of Algorithm 7 can be easily obtained.

Let $\varDelta_D$ be the maximum number of edges deleted from each vertex in V. Similarly, we can get that the number of VESs is $\varDelta_D$. When a VES E_i is deleted from graph $G = (U, V, E)$, the maximum degree of all vertices in U_c is denoted by $\max D_d$ and $\max D_D = \max_{1 \leqslant i \leqslant \varDelta_D} \max D_d$. For each α, let U_α^4 and V_α^4 be the set of vertices whose core numbers equal to $core(e_i)$ in U and V, respectively, where $e_i = (u_i, v_i) \in E_i$ and $core_\alpha(e_i) = \min\{core_\alpha(u), core_\alpha(v)\}$. Let $\tilde{U}_D = \max_{1 \leqslant \alpha \leqslant \max D_D} U_\alpha^4$ and $\tilde{V}_D = \max_{1 \leqslant \alpha \leqslant \max D_D} V_\alpha^4$. Let E_α^4 be the set of edges in the subgraph induced by U_α^4 to V_α^4 and let $\tilde{E}_D = \max_{1 \leqslant \alpha \leqslant \max D_D} E_\alpha^4$.

Let $\tilde{M}_U^\alpha = \max_{u_i \in U_\alpha^4}\{SN_0(u_i) - \alpha, 0\}$, where $SN_0(u_i)$ denotes the SN value of u_i when u_i is processed for the first time. Let $\tilde{M}_U = \max_{1 \leqslant \alpha \leqslant \max D_D} \tilde{M}_U^\alpha$. Similarly, we define $\tilde{M}_V^\alpha = \max_{v_i \in V_\alpha^4}\{SN_0(v_i) - core_\alpha, 0\}$ and $\tilde{M}_V = \max_{1 \leqslant \alpha \leqslant \max D_D} \tilde{M}_V^\alpha$.

Theorem 5 Algorithm 7 can correctly update the core numbers of all vertices after deleting an edge set E_s in $O(\varDelta_D \times (\max D_D \times ((\tilde{E}_D + \tilde{U}_D \times \tilde{M}_U + \tilde{V}_D \times \tilde{M}_V))))$ time.

7 Experiment

In this section, we conduct empirical studies to evaluate the performances of our proposed algorithms. We evaluate the algorithms on 5 real-world graphs and 2 synthetic graphs, real-world graphs are shown in Table 1. We first evaluate the running time of core decomposition, and then evaluate the efficiency of our proposed core maintenance algorithms on real graphs by computing the average processing time per edge while changing the number of edges inserted/deleted. We also compared the single-edge insertion/deletion algorithms and the multiple-edge insertion/deletion algorithms with the approach of recomputing the core numbers of all vertices, to assess the effectiveness of our core maintenance algorithm. Finally, we use synthetic graphs to evaluate the scalability of the proposed algorithms.

All programs were implemented in Java language and compiled with IntelliJ IDEA, and all experiments were performed on a machine with Intel Core i5-7500

Table 1 Real-world graph datasets and core decomposition time.

Dataset	N_u	N_v	M	$deg_{\max}(U)$	$deg_{\max}(V)$	*ComputeCore* (s)
OC (opsahl-collaboration)	1.7×10^4	2.2×10^4	0.59×10^6	116	18	0.7
DW (dbpedia-writer)	8.9×10^4	4.6×10^4	0.14×10^3	42	246	2.8
BC (BookCrossing)	0.4×10^6	0.1×10^6	1.2×10^6	13 601	2502	1737
BT (bibsonomy-2ti)	0.2×10^6	0.77×10^6	2.6×10^6	182 908	341	1017
WE (Wikipedia-en)	1.85×10^6	0.18×10^6	3.8×10^6	54	11 593	195

3.41 GHz and 8 GB DDR3-RAM in Windows 10. The graphs for the experimental section are generated by a python program.

Dataset. All the real-world graphs are downloaded from KONECT[†]. The opsahl-collaboration dataset is extracted from Openflights.org data, the dbpedia-writer dataset is extracted from DBpedia, the BookCrossing dataset contains information about books read by members of the BookCrossing community, the bibsonomy-2ti dataset is a bipartite tagCpublication network from BibSonomy, and the Wikipedia-en dataset is a bipartite edit network of the English Wikipedia. For the synthetic bipartite graphs, the basic idea is to randomly divide the vertices of a general graph into two groups and then delete the edges connecting the vertices in the same group. To generate general graphs, we use Stanford Network Analysis Platform system with two models: the Watts-Strogatz (WS) model[34], which is a random graph with small-world network properties such as clustering and short average path length, and the Gnm-Random (GR) model[35] that generates a random graph with a specified number of vertices and edges. All the synthetic graphs have vertices in the range of 2^{15} to 2^{19}, and the maximum degree is approximately 13 for GR graphs and 11 for WS graphs, respectively.

7.1 Performance evaluation

We first analyze the experimental results of the core decomposition algorithm and then evaluate the impact of the size of inserted/deleted edges on the single-edge and multiple-edge insertion/deletion algorithms, respectively. Finally, we analyze the influence of the graph size on the core maintenance algorithms. The first two experiments are conducted on real-world graphs, and the third one is on synthetic graphs.

The time of core decomposition on real graphs is given in Table 1. In general, the time of Algorithm 1 increases as the graph grows. But there are exceptions because the time complexity is also influenced by the vertex degree of the graph according to Theorem 1. For example, although the size of the graph BC is not very large, the core decomposition time is long due to its large vertex degree. On the other hand, since graph WE has a smaller vertex degree, the core decomposition is very fast.

We then evaluate the performance and the stability of our single-edge deletion/insertion algorithms. The evaluation is conducted on the five real-world graphs given in Table 1. In each graph, P_i edges are selected randomly, where $P_i = 10^i$ for $i = 0, 1, 2, 3, 4$. These edges are first deleted from the original graph to evaluate the deletion algorithm and then are inserted back to evaluate the insertion algorithm. The processing time per edge for the deletion and insertion cases are shown in Figs. 3a and 3b, respectively. It can be seen that the processing time per edge has a slight downward trend as the number of deleted/inserted edges increases. Theoretically, the execution time of each edge is nearly the same regardless of the number of changed edges. However, because of caching and other factors in the actual experiments, the execution time per edge will have a downward trend when we call the single-edge algorithms multiple times continuously. Furthermore, Fig. 3 also illustrates that the processing time of the single-edge insertion/deletion algorithms is much less, comparing with the recomputation of the core numbers of all vertices. This is because of the locality of core maintenance. Only a small part of vertices around the inserted/deleted edge may change the core number, and the core maintenance algorithms try to reduce the search range as much as possible.

We also evaluate the performance of multiple-edge insertion/deletion algorithms on real-world graphs. The experimental setting is similar to the single-edge case, and the results are demonstrated in Fig. 4. It can be seen that the processing time per edge of multiple-edge algorithms is much less than that of invoking the single-edge algorithm multiple times. The reason is that if there are a large number of updated edges, the multiple-edge insertion/deletion algorithm can handle multiple edges in one iteration, which can greatly reduce unnecessary duplicate accesses to vertices and edges. Furthermore, we can also find that the average

[†] http://konect.cc/networks/

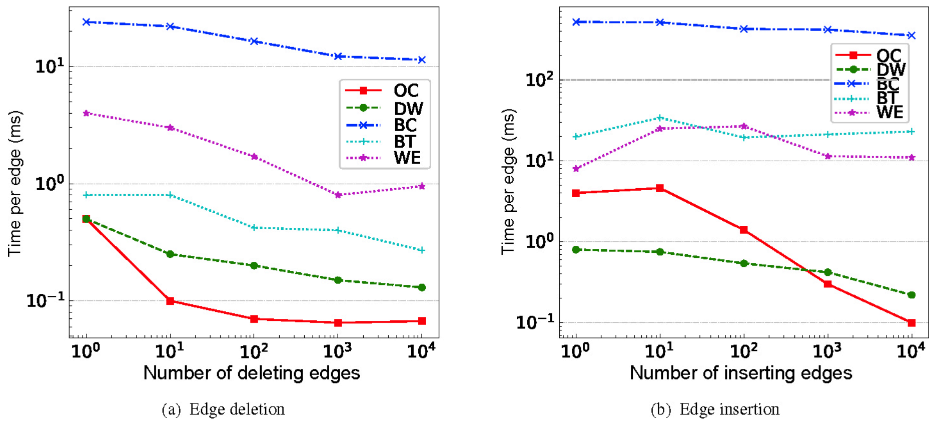


Fig. 3 Influence of the number of deleting/inserting edges on single-edge core maintenance algorithms.

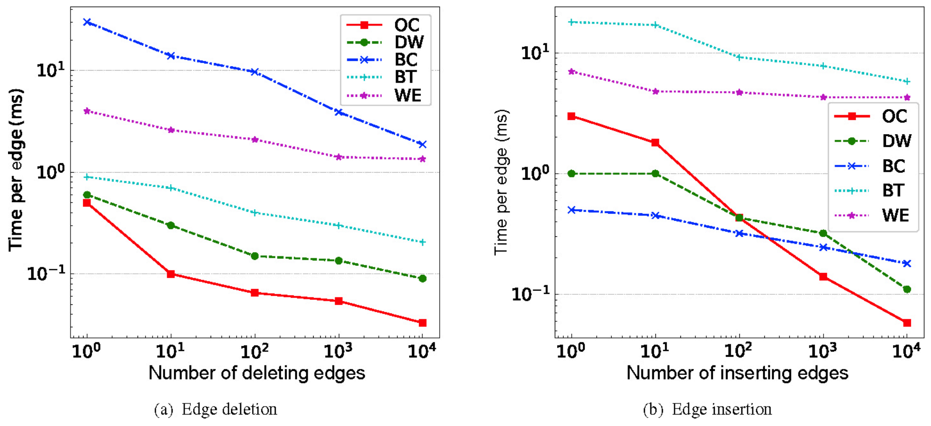


Fig. 4 Influence of the number of deleting/inserting edges on multiple-edge core maintenance algorithms.

processing time decreases as the number of updated edges increases, this is because when more edges are updated, they can be processed together using the multiple-edge insertion/deletion algorithms.

In Fig. 5, we demonstrate the acceleration ratio of the single-edge and multiple-edge core maintenance algorithms, comparing with recomputing the core numbers of all vertices when the graph changes. For each real-world graph, 10 000 edges are randomly selected as the update set. In Fig. 5, the x-axis represents the graphs and the y-axis is the ratio of the average time of processing an edge using core maintenance algorithms and the time of core decomposition. For the deletion scenario, the running time of a single-edge core maintenance algorithm for different graphs is less than 10% of the time for core decomposition. Meanwhile, the multiple-edge core maintenance algorithm takes less than 5% of the time for core decomposition. For graphs BC and BT, the speedup can be as much as 100 times. For the insertion case, the single-edge

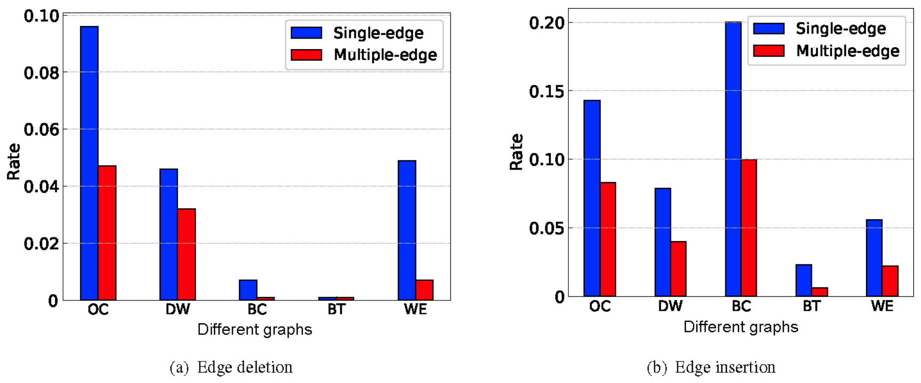


Fig. 5 Comparison of the efficiency of two core maintenance algorithms and the core decomposition algorithm.

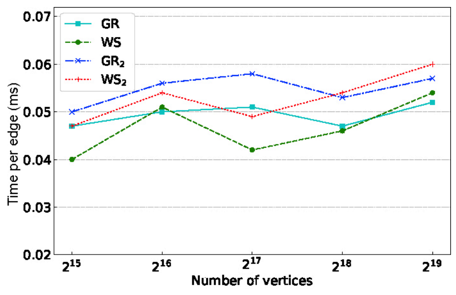
algorithm takes about 10% to 20% of the time for core decomposition, while the multiple-edge algorithm takes between 3% and 10% of the time for core decomposition. Hence, comparing with recomputing the core number, our core maintenance algorithms can greatly improve the efficiency of core numbers update.

7.2 Scalability evaluation

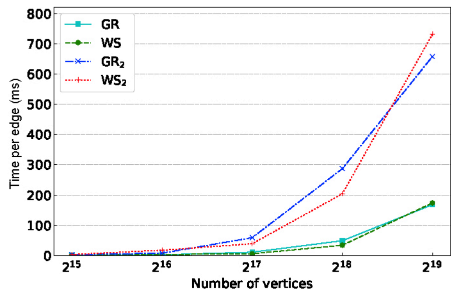
We finally evaluate the scalability of our algorithms in synthetic graphs, by letting the number of vertices scale from 2^{15} to 2^{19}. In the experiments, we randomly select 1000 edges as the update set for each graph. Here we compute the time per edge for single-edge algorithms (GR_2 and WS_2) and multiple-edge algorithms (GR and WS). As shown in Fig. 6, the former takes more time than the latter in almost all cases. Moreover, it also shows that though the graph size increases exponentially, the average processing time of the multiple-edge core maintenance algorithms increases almost linearly, which demonstrates our algorithms can work well in the graphs with large size. Especially, it is surprising to see that the average processing time of the deletion algorithms changes slightly as graph size increases exponentially.

8 Conclusion

We propose an index of $\alpha(\beta)$-core number in bipartite graphs that reflects the maximal cohesive and dense subgraphs a vertex can be in. Efficient algorithms for core decomposition and core maintenance are proposed to compute and update the core numbers of vertices. The core decomposition algorithm can compute the core number of every vertex in linear time and space, while by quantifying the core number change and accurately identifying the vertices whose core numbers may change after edge insertion/deletion, the core decomposition algorithms can update the core numbers in both the single-edge and multiple-edge insertion/deletion scenarios, visiting only a small number of vertices and edges, thus avoiding recomputations. Experimental results showed that the core maintenance algorithms can greatly reduce the core number update time comparing with recalculation.



(a) Edge deletion



(b) Edge insertion

Fig. 6 Impact of graph size; GR (GR_2) and WS (WS_2) refer to the multiple-edge (single-edge) core maintenance algorithms.

Acknowledgment

This work was supported by the National Key Research and Development Program of China (No. 2019YFB2102600), the National Natural Science Foundation of China (Nos. 62122042 and 61971269), and the Blockchain Core Technology Strategic Research Program of Ministry of Education of China (No. 2020KJ010301) fund.

References

[1] A. Beutel, W. Xu, V. Guruswami, C. Palow, and C. Faloutsos, Copycatch: Stopping group attacks by spotting lockstep behavior in social networks, in *Proc. 22nd International Conference on World Wide Web*, Rio de Janeiro, Brazil, 2013, pp. 119–130.

[2] M. Kaytoue, S. O. Kuznetsov, A. Napoli, and S. Duplessis, Mining gene expression data with pattern structures in formal concept analysis, *Inf. Sci.*, vol. 181, no. 10, pp. 1989–2001, 2011.

[3] J. Wang, A. P. de Vries, and M. J. T. Reinders, Unifying user-based and item-based collaborative filtering approaches by similarity fusion, in *Proc. 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, WA, USA, 2006, pp. 501–508.

[4] D. Ding, H. Li, Z. Huang, and N. Mamoulis, Efficient fault-tolerant group recommendation using alpha-beta-core, in *Proc. 2017 ACM on Conference on Information and Knowledge Management*, Singapore, 2017, pp. 2047–2050.

[5] Z. Cai, Z. He, X. Guan, and Y. Li, Collective data-sanitization for preventing sensitive information inference attacks in social networks, *IEEE Trans. Dependable Secur. Comput.*, vol. 15, no. 4, pp. 577–590, 2018.

[6] K. Li, G. Lu, G. Luo, and Z. Cai, Seed-free graph de-anonymiztiation with adversarial learning, in *Proc. 29th ACM International Conference on Information & Knowledge Management*, Virtual Event, Ireland, 2020, pp. 745–754.

[7] K. Li, G. Luo, Y. Ye, W. Li, S. Ji, and Z. Cai, Adversarial privacy-preserving graph embedding against inference attack, *IEEE Internet Things J.*, vol. 8, no. 8, pp. 6904–6915, 2021.

[8] K. Xu, R. Williams, S. H. Hong, Q. Liu, and J. Zhang, Semi-bipartite graph visualization for gene ontology networks, in *Proc. 17th international conference on Graph Drawing*, Chicago, IL, USA, 2009, pp. 244–255.

[9] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou, Efficient (α, β)-core computation in bipartite graphs, *VLDB J.*, vol. 29, no. 5, pp. 1075–1099, 2020.

[10] S. B. Seidman, Network structure and minimum degree, *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[11] Q. -S. Hua, Y. Shi, D. Yu, H. Jin, J. Yu, Z. Cai, X. Cheng, and H. Chen, Faster parallel core maintenance algorithms in dynamic graphs, *IEEE Trans. Parallel Distributed Syst.*, vol. 31, no. 6, pp. 1287–1300, 2020.

[12] Y. Zhang, B. Wu, Y. Liu, and J. Lv, Local community detection based on network motifs, *Tsinghua Science and Technology*, vol. 24, no. 6, pp. 716–727, 2019.

[13] L. Yu, B. Wu, and B. Wang. LBLP: Link-clustering-based approach for overlapping community detection, *Tsinghua Science and Technology*, vol. 4, pp. 387–397, 2013.

[14] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy, Approximating clique is almost NP-complete (preliminary version), in *Proc. 32nd Annual Symposium on Foundations of Computer Science,* San Juan, PR, USA, 1991, pp. 2–12.

[15] V. Batagelj and M. Zaversnik, An o(m) algorithm for cores decomposition of networks, arXiv preprint arXiv: cs/0310049, 2003.

[16] Q. Luo, D. Yu, F. Li, Z. Dou, Z. Cai, J. Yu, and X. Cheng, Distributed core decomposition in probabilistic graphs, in *Proc. 8th International Conference on Computational Data and Social Networks,* Ho Chi Minh City, Vietnam, 2019, pp. 16–32.

[17] D. Yu, L. Zhang, Q. Luo, X. Cheng, J. Yu, and Z. Cai, Fast skyline community search in multi-valued networks, *Big Data Mining Analytics*, vol. 3, no. 3, pp. 171–180, 2020.

[18] P. Chen, C. Chou, and M. Chen, Distributed algorithms for k-truss decomposition, in *Proc. 2014 IEEE International Conference on Big Data,* Washington, DC, USA, 2014, pp. 471–480.

[19] Q. Luo, D. Yu, X. Cheng, Z. Cai, J. Yu, and W. Lv, Batch processing for truss maintenance in large dynamic graphs, *IEEE Trans. Comput. Soc. Syst.*, vol. 7, no. 6, pp. 1435–1446, 2020.

[20] J. Wang and J. Cheng, Truss decomposition in massive networks, *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 812–823, 2012.

[21] B. Balasundaram, S. Butenko, and I. V. Hicks, Clique relaxations in social network analysis: The maximum k-plex problem, *Oper. Res.*, vol. 59, no. 1, pp. 133–142, 2011.

[22] M. Bentert, A. -S. Himmel, H. Molter, M. Morik, R. Niedermeier, and R. Saitenmacher, Listing all maximal k-plexes in temporal graphs, in *Proc. 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, Barcelona, Spain, 2018, pp. 41–46.

[23] A. Ahmed, V. Batagelj, X. Fu, S. Hong, D. Merrick, and A. Mrvar, Visualisation and analysis of the internet movie database, in *Proc. 2007 6th International Asia-Pacific Symposium on Visualization*, Sydney, Australia, 2007, pp.17–24.

[24] D. S. Hochbaum, Approximating clique and biclique problems, *Journal of Algorithms*, vol. 29, no. 1, pp. 174–200, 1998.

[25] K. Sim, J. Li, V. Gopalkrishnan, and G. Liu, Mining maximal quasi-bicliques to co-cluster stocks and financial ratios for value investment, in *Proc. 6th IEEE International Conference on Data Mining* (*ICDM 2006*), Hong Kong, China, 2006, pp. 1059–1063.

[26] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K. -L. Wu, and Ü. V. Çatalyürek, Incremental k-core decomposition: Algorithms and evaluation, *VLDB J.*, vol. 25, no. 3, pp. 425–447, 2016.

[27] N. Wang, D. Yu, H. Jin, C. Qian, X. Xie, and Q. -S. Hua, Parallel algorithms for core maintenance in dynamic graphs, arXiv preprint arXiv: 1612.09368, 2016.

[28] H. Jin, N. Wang, D. Yu, Q. -S. Hua, X. Shi, and X. Xie, Core maintenance in dynamic graphs: A parallel approach based on matching, *IEEE Trans. Parallel Distributed Syst.*, vol. 29, no. 11, pp. 2416–2428, 2018.

[29] A. E. Sariyüce, C. Seshadhri, A. Pinar, and Ü. V. Catalyurek, Finding the hierarchy of dense subgraphs using nucleus decompositions, in *Proc. 24th International Conference on World Wide Web*, Florence, Italy, 2015, pp. 927–937.

[30] H. Aksu, M. Canim, Y. -C. Chang, I. Korpeoglu, and Ö. Ulusoy, Distributed k-core view materialization and maintenance for large dynamic graphs, *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 10, pp. 2439–2452, 2014.

[31] S. Aridhi, M. Brugnara, A. Montresor, and Y. Velegrakis, Distributed k-core decomposition and maintenance in large dynamic graphs, in *Proc. 10th ACM International Conference on Distributed and Event-Based Systems*, Irvine, CA, USA, 2016, pp. 161–168.

[32] W. Zhou, H. Huang, Q. -S. Hua, D. Yu, H. Jin, and X. Fu, Core decomposition and maintenance in weighted graph, *World Wide Web*, vol. 24, no. 2, pp. 541–561, 2021.

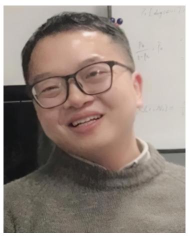
[33] Q. Luo, D. Yu, Z. Cai, X. Lin, and X. Cheng, Hypercore maintenance in dynamic hypergraphs, in *Proc. 2021 IEEE 37th International Conference on Data Engineering* (*ICDE*), Chania, Greece, 2021, pp. 2051–2056.

[34] D. J. Watts and S. H. Strogatz, Collective dynamics of small world networks, *Nature*, vol. 393, pp. 440–442, 1998.

[35] D. E. Knuth, *Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition*. Boston, MA, USA: Addison Wesley, 1997.

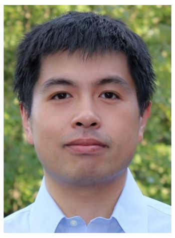
Dongxiao Yu received the BS degree from Shandong University in 2006 and the PhD degree from University of Hong Kong in 2014. He became an associate professor in the School of Computer Science and Technology, Huazhong University of Science and Technology in 2016. He is currently a professor in the School of Computer Science and Technology, Shandong University. His research interests include wireless networks, distributed computing, and graph algorithms.

Lifang Zhang received the BS degree from Shandong University in 2019. She is currently pursuing the master degree at Shandong University. Her research interests include graph analysis and data mining.



Qi Luo received the BS degree from Northeastern University in 2015, and the MS and PhD degrees in computer science from Shandong University in 2018 and 2022, respectively. His research interests include graph data mining and analysis.

Xiuzhen Cheng received the MS and PhD degrees in computer science from University of Minnesota Twin Cities in 2000 and 2002, respectively. She is a professor in the School of Computer Science and Technology, Shandong University, Qingdao, China. She has published more than 170 peer-reviewed papers. Her current research interests include cyber physical systems, wireless and mobile computing, sensor networking, wireless and mobile security, and algorithm design and analysis. She has served on the editorial boards of several technical journals and the technical program committees of various professional conferences/workshops. She also has chaired several international conferences. She worked as a professor in the Department of Computer Science, George Washington University, Washington, DC, USA from 2013 to 2017. She worked as a program director for the US National Science Foundation (NSF) from April to October in 2006 (full time), and from April 2008 to May 2010 (part time). She received the NSF CAREER Award in 2004. She is a fellow of IEEE and a member of ACM.



Zhipeng Cai is currently an associate professor in the Department of Computer Science at Georgia State University, USA. He received the PhD and MS degrees from University of Alberta, and BS degree from Beijing Institute of Technology. His research areas focus on wireless networking, Internet of Things, machine learning, cyber-security, and privacy and big data. He is the recipient of an NSF CAREER Award. He served as a steering committee co-chair and a steering committee member for WASA and IPCCC. He also served as a technical program committee member for more than 20 conferences, including INFOCOM, MOBIHOC, ICDE, and ICDCS. He has been serving as an associate editor-in-chief for *Elsevier High-Confidence Computing Journal* (*HCC*), and an associate editor for several international journals, such as *IEEE Internet of Things Journal* (*IoT-J*), *IEEE Transactions on Knowledge and Data Engineering* (*TKDE*), and *IEEE Transactions on Vehicular Technology* (*TVT*). He has published more than 70 papers in prestigious journals with more than 40 papers published in *IEEE/ACM Transactions*.