applied sciences

MDPI

Article

Cohesive Subgraph Identification in Weighted Bipartite Graphs

Xijuan Liu and Xiaoyang Wang *

School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China; liuxijuan@zjgsu.edu.cn

* Correspondence: xiaoyangw@zjgsu.edu.cn

Abstract: Cohesive subgraph identification is a fundamental problem in bipartite graph analysis. In real applications, to better represent the co-relationship between entities, edges are usually associated with weights or frequencies, which are neglected by most existing research. To fill the gap, we propose a new cohesive subgraph model, (k, ω)-core, by considering both subgraph cohesiveness and frequency for weighted bipartite graphs. Specifically, (k, ω)-core requires each node on the left layer to have at least k neighbors (cohesiveness) and each node on the right layer to have a weight of at least ω (frequency). In real scenarios, different users may have different parameter requirements. To handle massive graphs and queries, index-based strategies are developed. In addition, effective optimization techniques are proposed to improve the index construction phase. Compared with the baseline, extensive experiments on six datasets validate the superiority of our proposed methods.

Keywords: bipartite graph; cohesive subgraph; (k, ω)-core; index construction

Citation: Liu, X.; Wang, X. Cohesive Subgraph Identification in Weighted Bipartite Graphs. *Appl. Sci.* **2021**, *11*, 9051. https://doi.org/10.3390/app11199051

Academic Editor: Ugo Vaccaro

Received: 8 September 2021
Accepted: 24 September 2021
Published: 28 September 2021

1. Introduction

Bipartite graphs are widely used in many real-world applications to model the complex relationships across different types of entities, such as customer–product network and author–paper collaboration network [1–4]. A bipartite graph $G = (L, R, E)$ consists of two sets of disjoint nodes, i.e., L and R. Only nodes from different sets can be connected. For example, Figure 1 shows an example of customer–product bipartite network, where edges represent the purchase relationships. The left layer L is a set of customers and the right layer R consists of a set of products purchased. There is no edge between the customers L (resp. products R).

As a fundamental problem in graph analysis, cohesive subgraph identification is widely studied in the literature (e.g., [5–8]). For bipartite graphs, a variety of cohesive subgraph models have been proposed to identify important structures, such as (α, β)-core [3], bitruss [9] and biclique [10]. Biclique is the most cohesive model, which requires the nodes inside to be fully connected. However, the computation complexity, i.e., NP-hard, makes it hard to apply in many time-efficient applications. Bitruss [11] adopts the butterfly motif (i.e., a $(2, 2)$-biclique) to investigate the cohesiveness of bipartite graphs. The (α, β)-core of bipartite graphs, which can be computed in linear time, has attracted great attention recently [3,12,13]. However, the model still has a drawback.

Motivations: Given a bipartite graph $G = (L, R, E)$, (α, β)-core is the maximal subgraph, where each node in L has at least α neighbors in R while each node in R has at least β neighbors in L. It can be computed in linear time by iteratively deleting the node with a degree less than α or β. For instance, in Figure 1, the subgraph consisting of nodes $\{u_2, u_3, v_2, v_3, v_4\}$ is a (2,1)-core. However, in the (α, β)-core, it only emphasizes the engagement of each node, i.e., each node has a sufficient number of neighbors in the subgraph and treats each edge equally. However, in real applications, edges usually tend to have quite different weights. For example, in the customer–product network (e.g., Figure 1), each edge is assigned a weight, which reflects the frequency between a customer and a product. The frequency denotes the number of times the customer has bought the product.

To make sense of the weight information, we propose a novel model, (k, ω)-core, to detect the densely frequent communities, which ensures that the nodes in the left layer have a sufficient number of neighbors and the nodes in the right layer have enough weights. Given a bipartite graph, the (k, ω)-core is the maximal subgraph where each node in L (resp. R) has at least k neighbors (resp. ω weight). The weight of a node is the sum of the weights of each adjacent edge. For instance, reconsidering the graph in Figure 1, the weights of products $\{v_1, v_2, v_3, v_4, v_5\}$ are $\{1, 5, 5, 3, 1\}$, and the subgraph consisting of $\{u_1, u_2, u_3, v_2, v_3, v_4\}$ is a (2,2)-core. The nodes $\{u_4, v_1, v_5\}$ are excluded from the (2,2)-core, since customer $\{u_4\}$ has not bought a sufficient number of distinct products while products $\{v_1, v_5\}$ have not been purchased enough times.

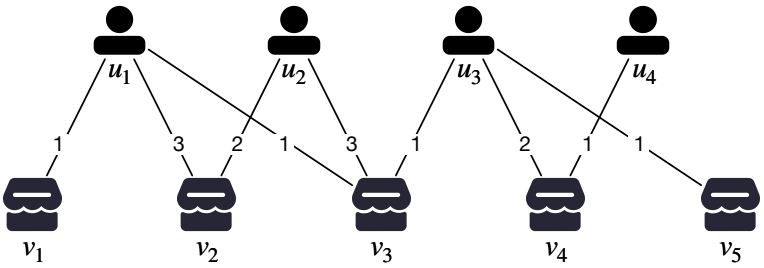


Figure 1. A weighted bipartite graph of the customer–product network (the weight on the edge denotes the number of times that the customer has bought the product).

Applications: The proposed (k, ω)-core model can be used in many real-world applications, such as product recommendation and fraud detection.

- Product recommendation: In a product–customer network (e.g., Figure 1), a (k, ω)-core means a group of users with sufficient common tastes. Then, we can use the group information for product recommendation. For example, in Figure 1, $\{u_1, u_2, u_3, v_2, v_3, v_4\}$ is the (2,2)-core. Then, we can recommend product v_2 to user u_3, since u_3 shares many common interests with u_1 and u_2.
- Fraud detection: For an online shopping website, fraudsters use a larger number of accounts to frequently purchase some selected products in order to boost the ranking of these products. This behavior can be modeled with a (k, ω)-core. By carefully selecting the parameters, we can use the detected (k, ω)-core to narrow down the searching space of fraudster accounts.

In real-life applications, the value of k (resp. ω) is determined by users based on their own requirements. The two parameters provide more flexibility when adjusting the resulting communities. As observed, the (α, β)-core is a special case of the (k, ω)-core when all the weights in the graph equal 1. Naively, we can extend the solution of computing (α, β)-core by iteratively deleting the nodes violating the constraints. The time complexity is linear to the input graph. However, in real applications, the graph size is usually large, which means algorithms that are linear to the input graph size are also not affordable [14]. In addition, different users may have different requirements of the input parameters k and ω, which can lead to a large amount of queries. Therefore, more efficient methods are expected to handle the massive graphs and queries.

In this paper, we resort to index-based approaches. A straightforward solution is to compute all possible (k, ω)-cores and maintain all the results. However, it will cause a huge computational cost by visiting the same subgraph multiple times. Thus, the time cost of computing all (k, ω)-cores becomes unaffordable on large graphs. To reduce the cost, we propose different index construction strategies to ensure a balance between building space-efficient indexes and supporting efficient-scalable query processing. Our major contributions are summarized as follows:

- We propose a new cohesive subgraph model (k, ω)-core on weighted bipartite graphs by considering both density and frequency of the subgraph.

- To efficiently handle massive graphs and queries, we develop three advanced index construction strategies, i.e., RowIndex, OptionIndex and UnionIndex, to reduce index construction cost. In addition, the corresponding querying algorithms by using the three index structures are provided.
- We validate the advantages of the proposed algorithms through extensive experiments on real-world datasets. The results show that the index-based algorithms outperform the baselines significantly. Moreover, users can make a trade-off between the time and space cost when selecting from the three strategies.

Roadmap: The rest of the paper is organized as follows. In Section 2, we introduce the (k, ω)-core model and formulate our problem. Section 3 introduces the naive online algorithm. Section 4 presents the index-based algorithms and advanced index structures. We report our experimental results in Section 5 and review the related work in Section 6. Finally, we present the conclusion and future work in Section 7.

2. Preliminaries

We use $G = (L, R, E, W)$ to denote a weighted bipartite graph, where nodes in G are partitioned into two disjoint sets L and R, such that each edge from $E \subseteq L \times R$ connects two nodes from L and R, respectively. We use $n = |L| + |R|$ and $m = |E|$ to denote the number of nodes and edges, respectively. $N(u)$ is the set of adjacent nodes of u in G, which is also called the neighbor set of u in G. The degree of a node $u \in L$, denoted by $d(u)$, is the number of neighbors of u in G. For each edge $e(u, v)$, we assign it a positive weight $w(u, v) \in W$, defined as the frequency of edge $e(u, v)$. The weight of a node $v \in R$, denoted by $w_t(v) = \sum_{u \in N(v)} w(u, v)$, is the sum of weights of each adjacent edge. We use $k_{max} = max\{d(u)|u \in L\}$ and $\omega_{max} = max\{\omega(v)|v \in R\}$ to denote the maximum degree and weight for nodes in G, respectively. For a bipartite graph G and two node sets $L' \subseteq L$ and $R' \subseteq R$, the bipartite subgraph induced by L' and R' is the subgraph G' of G such that $E' = E \cap (L' \times R')$. To evaluate the cohesiveness and frequency of communities in weighted bipartite subgraphs, we resort to the minimum degree for node set L and minimum weight for node set R. In detail, for an induced subgraph, we request that nodes in L' have a degree of at least k and nodes in R' have a weight no less than ω.

Definition 1 ((k, ω)-core)**.** *Given a weighted bipartite graph $G = (L, R, E, W)$ and two query parameters k and ω, the induced subgraph $S = (L', R', E', W')$ is the (k, ω)-core of G, denoted by $C_{k,\omega}$, if S satisfies:*

- *Degree constraint. For each node $u \in L'$, it has degree at least k, i.e., $d(u, S) \geq k$;*
- *Weight constraint. For each node $v \in R'$, it has weight no less than ω, i.e., $\omega_t(v, S) \geq \omega$;*
- *Maximal. Any supergraph $S' \supset S$ is not a (k, ω)-core.*

Example 1. *Figure 1 is a toy weighted bipartite graph for modeling the customer–product affiliations. It consists of two layers of nodes, i.e., the four nodes in the left layer denote the customers and five nodes in the right layer denote the products. The edges between nodes represent the purchase relationships and the weight of edges reflects the purchase frequency. Given the query parameters $k = 2$ and $\omega = 4$, we can obtain $C_{2,4}$ consisting of nodes $\{u_1, u_2, v_2, v_3\}$.*

For simplicity, we refer to a weighted bipartite graph as a graph, and omit G, S in the notations if the context is self-evident. In the following lemma, we show that (k, ω)-core has the nested property. It is easy to verify the correctness of the lemma based on the definition. Thus, we omit the proof here.

Lemma 1. *Given a weighted bipartite graph G, the (k', ω')-core is nested to the (k, ω)-core, i.e., $C_{k',\omega'} \subseteq C_{k,\omega}$, if $k' \geq k$ and $\omega' \geq \omega$.*

Example 2. *As shown in Example 1, $C_{2,4}$ consists of nodes $\{u_1, u_2, v_2, v_3\}$. Suppose $k = 2$ and $\omega = 2$. We can find that $C_{2,2}$ contains $C_{2,4}$, i.e., $C_{2,2} = \{u_1, u_2, u_3, v_2, v_3, v_4\} \supseteq C_{2,4}$.*

Problem 1. *Given a weighted. bipartite graph G and two query parameters k and ω, we aim to design algorithms to compute the (k, ω)-core correctly and efficiently.*

3. Online Solution

Before introducing the detailed algorithms, Figure 2 shows the general framework of the proposed techniques in this paper. To identify the (k, ω)-core, an online solution is first developed in Section 3. To efficiently handle large networks and different input parameters, an index-based solution is further proposed in Section 4. The index-based solution consists of two phases: an index construction phase and query phase. In addition, different optimization techniques are proposed to ensure a balance between the index construction time and index space.

For the online solution, we introduce a baseline algorithm, named GCORE, by extending the solution for (α, β)-core computation. The main idea of GCORE is to iteratively remove nodes with a degree less than k in L and a weight less than ω in R. GCORE terminates until the size of G stays unchanged, i.e., there is no node that violates the constraints. Then, we output the remaining graph as (k, ω)-core. The details are shown in Algorithm 1. In Lines 2–5, we check the degree constraint for nodes in L. For each node $u \in L$ with $d(u) < k$, we remove it with its adjacent edges. Then, we update the weight of node v in $N(u)$, i.e, subtract the weight of corresponding removed edge $e(u, v)$ from the total weight $w_t(v)$. In Lines 6–9, we examine the weight constraint for nodes in R. For each node v with $w_t(v) < \omega$, we remove it with its incident edges. Accordingly, we decrease the degree of u by 1 for each u in $N(v)$, which may cause the node to violate the degree constraint. The algorithm terminates until both constraints are satisfied and finally returns the (k, ω)-core of G.

Algorithm 1: GENERATE (k, ω)-CORE

Input : Bipartite graph: $G = (L, R, E, W)$, degree constraint: k, weight constraint: ω
Output: The (k, ω)-core of G

```
1  while ∃u ∈ L with d(u) < k ∨ ∃v ∈ R with w_t(v) < ω do
2      for each u ∈ L ∧ d(u) < k do
3          for each v ∈ N(u) do
4              w_t(v) ← w_t(v) − w(u, v);
5          L.remove(u);
6      for each v ∈ R ∧ w_t(v) < ω do
7          for each u ∈ N(v) do
8              d(u) ← d(u) − 1;
9          R.remove(v);
10 return G
```

Discussion: The time complexity of Algorithm 1 is linear to the size of the graph. However, as discussed in the introduction, the method is still not affordable, especially for massive graphs and queries.

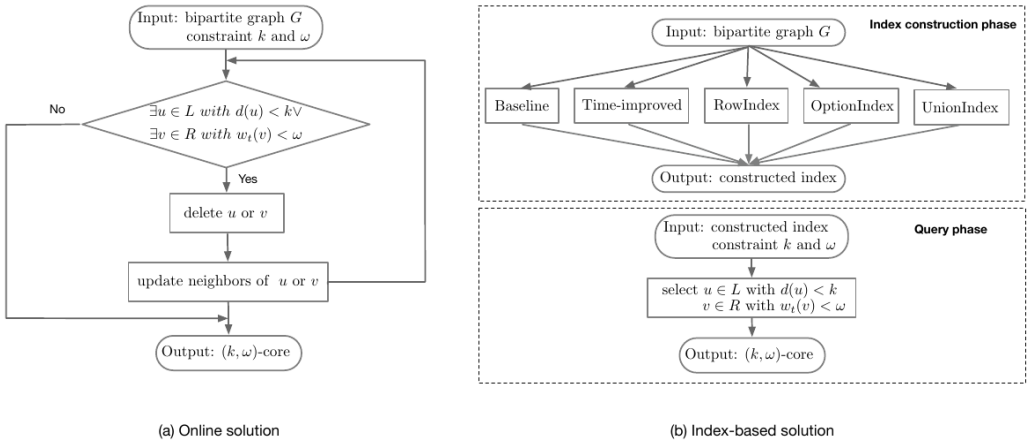


Figure 2. General framework of the online and index-based solutions.

4. Index-Based Solution

For each input parameter, Algorithm 1 has to compute the (k, ω)-core from scratch, which is time-consuming and cannot support a large number of queries. To tackle the challenges, in this section, index-based algorithms are developed. The main idea is that we effectively organize all the (k, ω)-cores in the index, so that a query could be efficiently answered. Firstly, a baseline solution is presented. To speed up the processing of the baseline, we devise a time-improved solution. Then, several novel index structures are developed to shrink the storage space.

4.1. Baseline Solution

Intuitively, the naive index-based algorithm is to compute all the (k, ω)-cores by repeatedly using the GCORE algorithm and then storing all of them in the index. As a result, we can quickly return the (k, ω)-core for any given query parameters. In details, we organize all the (k, ω)-cores in a two-dimensional index. That is, the nodes in (k, ω)-core are all stored in (k, ω)-cell, where (k, ω)-cell is in the k-th row and ω-th column $(0 \leq k \leq k_{max}, 0 \leq \omega \leq \omega_{max})$ of the index. The procedure terminates until all the possible (k, ω)-cores are found. As a result, we can immediately obtain (k, ω)-core for any given pair of parameters k and ω, according to the two-dimensional locations of cells. Table 1 shows the index for the graph in Figure 1. For example, the set of nodes in the $(1, 1)$-core, i.e., $\{u_1, u_2, u_3, u_4, v_1, v_2, v_3, v_4, v_5\}$, are all stored in the $(1, 1)$-cell. If querying the $(1, 1)$-core, we only need to visit the $(1, 1)$-cell. Hence, $Q_{1,1}$ can be easily solved in optimal time, with $O(1)$ time complexity.

Table 1. Index construction example.

k/ω	1	2	3	4	5
1	u_1, u_2, u_3, u_4 v_1, v_2, v_3, v_4, v_5	u_1, u_2, u_3, u_4 v_2, v_3, v_4	u_1, u_2, u_3, u_4 v_2, v_3, v_4	u_1, u_2, u_3 v_2, v_3	u_1, u_2, u_3 v_2, v_3
2	u_1, u_2, u_3 v_1, v_2, v_3, v_4, v_5	u_1, u_2, u_3 v_2, v_3, v_4	u_1, u_2 v_2, v_3	u_1, u_2 v_2, v_3	
3	u_1, u_3 v_1, v_2, v_3, v_4, v_5				

4.2. Time-Improved Method

The baseline index method is time-consuming, since we need to compute all the possible (k, ω)-cores one by one. Due to the nested property of (k, ω)-core, many subgraphs will be computed multiple times. To reduce the time consumption, we resort to the time-improved solution by escaping the unnecessary (k, ω)-core computations. Before

going to the detailed method, we first introduce the concept of $\omega_{max,k}(u)$ to help present the algorithm.

Definition 2 ($\omega_{max,k}(u)$)**.** *Given a weighted bipartite graph $G = (U, E, W)$, where $U = R \cup L$, and a specific value k, for each node $u \in U$, $\omega_{max,k}(u)$ is the maximum value of ω for which there exists a (k, ω)-core that contains u.*

For a node $u \in U$ and a specific value k, we know that the $(k, \omega_{max,k}(u))$-core contains u by Definition 2. According to the nested property of (k, ω)-core by Lemma 1, we can infer that the $(k, \omega_{max,k}(u))$-core is also contained in (k, ω_i)-cores of G, where ω_i is no larger than $\omega_{max,k}(u)$. Thus, there are many redundant computations in the process of constructing index structure. To address the above concerns, we devise an improved index-based algorithm. Given a graph G and an integer k, we first compute $\omega_{max,k}(u)$ for each node $u \in U$ and then store u in the (k, ω)-cells where $0 \leq \omega \leq \omega_{max,k}(u)$. Note that we store all nodes in *row* for a specific input k. The details are shown in Algorithm 2.

Algorithm 2: ComputeRow(k, G)

```
   Input  : Bipartite graph: G = (L, R, E, W)
   Output : Constructed index
 1 Initialize row[ ] ← ∅, ω ← 1;
 2 G' = (L', R', E', W') ← Generate (k, 0)-core;
 3 while G' is not empty do
 4     while ∃u ∈ L' : d(u) < k ∨ ∃v ∈ R' : w_t(v) < ω do
 5         for each u ∈ L' ∧ d(u) < k do
 6             for each v ∈ N(u) do
 7                 w_t(v) ← w_t(v) − w(u, v);
 8             L'.remove(u);
 9             for i = 0 to ω − 1 do
10                 Put u into row[i];
11         for each v ∈ R' ∧ w_t(v) < ω do
12             for each u ∈ N(v) do
13                 d(u) ← d(u) − 1;
14             R'.remove(v);
15             for i = 0 to ω − 1 do
16                 Put v into row[i];
17     ω ← ω + 1;
18 return row
```

In Algorithm 2, we first initialize *row* as empty and ω as 1 (Line 1). Then, we generate the $(k, 0)$-core as the candidate subgraph by using the GCore algorithm. In Lines 5–10, if node $u \in L'$ violates the degree constraint, we remove it with its adjacent edges and update the weight of the node $v \in R'$ which is also included in the neighbor set of u. After obtaining $\omega_{max,k}(u)$, we put node u into $row[i]$ where $0 \leq i \leq \omega_{max,k}(u) - 1$. Similarly, we check the weight constraint. In Lines 11–16, if node $v \in R'$ dissatisfies the weight constraint, we decrease the degree of the node u inside the neighbor set of v by 1. Then, we obtain $\omega_{max,k}(v)$ and put node v into $row[i]$, where $0 \leq i \leq \omega_{max,k}(v) - 1$. We continue the iteration until all the nodes are removed from G'. Finally, we return *row* as the resulting index for a given specific k. Note that we can obtain the index structure for the whole graph by repeatedly invoking Algorithm 2 with different input values of k.

Discussion: Although the time-improved method can speedup the processing, it is prohibitive for large graphs due to the large index storage cost. This is because a node can

be stored in multiple cells due to the nested property. For instance, given a fixed $k = 1$, the nodes in the $(1,3)$-cell will also be stored in $(1,1)$-cell, $(1,2)$-cell and $(1,3)$-cell. Similarly, for a specific ω, the same problem still exists when computing the column index.

4.3. Advanced Index Structures

As discussed, the baseline index method suffers from storage issues. To shrink the index space without sacrificing much efficiency, we introduce three novel index structures, i.e., (1) RowIndex: by utilizing the nested property of (k,ω)-core, we compress each *row* of the index; (2) OptionIndex: by comparing the shrink size of compression in *row* and *column*, we select the better compression direction; (3) UnionIndex: by considering both *row* and *column* compression, we conduct the union operations on cells of the index. In addition, the corresponding query algorithms are presented.

4.3.1. Rowindex

According to the nested property in Lemma 1, we know that $C_{k,\omega}$ is always a subset of $C_{k,\omega-1}$. Thus, we resort to the RowIndex by compressing *row* of the index, since it can avoid storing a single node many times. Given a specific k, we say that all the $(k,*)$-cells are in the k-th row, where the symbol "$*$" represents any possible value of ω. The main difference between RowIndex and the index structure proposed above is that we only store each node $u \in U$ in the $(k,\omega_{max,k}(u))$-cell, instead of putting it into (k,ω_i)-cells where $0 \leq \omega_i \leq \omega_{max,k}(u)$. Thus, we only need to deposit each node at most once in each *row* of the index, which can save space from the redundant copies of nodes. Meanwhile, we also record the shrink direction (i.e., "$\rightarrow$") in the *shrink*, which is a direction table. As the procedure of RowIndex is easy to understand, we omit its pseudo-codes in the context.

RowIndex Query Algorithm: Given query parameters k and ω, we first locate the (k,ω)-cell. Then, we collect all the nodes contained in the (k,ω_i)-cell where $\omega \leq \omega_i \leq \omega_{max}$, and output them together as the resulting (k,ω)-core.

Example 3. *As shown in Table 1, for $k = 1$, the $(1,1)$-cell containing nodes $\{u_1,u_2,u_3,u_4,v_1,v_2,v_3,v_4,v_5\}$ can be compressed to the $(1,3)$-cell and the $(1,5)$-cell. That is, nodes u_4,v_4 only need to be saved in the $(1,3)$-cell and nodes u_1,u_2,u_3,v_2,v_3 only need to be stored in the $(1,5)$-cell. Thus, only the remaining nodes v_1 and v_5 are stored in the $(1,1)$-cell. Obviously, RowIndex saves a lot of space. When querying the $(2,3)$-core, we first locate the $(2,3)$-cell and output nodes in the $(2,3)$-cell and $(2,4)$-cell together. Thus, we have $C_{2,3} = \{u_1,u_2,v_2,v_3\}$.*

4.3.2. OptionIndex

As discussed above, RowIndex utilizes the nested property to reduce the redundant storage for each node in each *row* of the index. Similarly, we can construct ColumnIndex to compress each *column* of the index in the same manner, which also enjoys the same space cost. Naturally, it is possible that certain cells may compress more storage by ColumnIndex than RowIndex. That is, *column* compression may contribute more to space saving for some cells. Motivated by this, we devised the OptionIndex structure, which is constructed by traversing all cells one by one. Specifically, when visiting a specific cell, we first compared the compression size of different compression directions, i.e., RowIndex or ColumnIndex, and then selected the better one to reduce more space. For example, in Table 1, the compression size is 7 if we use RowIndex to shrink the $(1,1)$-cell to the $(1,2)$-cell with shrink direction "$\rightarrow$". Additionally, the compression size is 8 if we use ColumnIndex to shrink the $(1,1)$-cell to the $(2,1)$-cell with shrink direction "$\downarrow$". Since ColumnIndex shrinks more than the RowIndex for the $(1,1)$-cell, we chose ColumnIndex and shrank $(1,1)$-cell to the $(2,1)$-cell. Similarly, we chose RowIndex for the $(1,4)$-cell, as RowIndex saves a space of five nodes while ColumnIndex saves four. The details of the construction procedure for OptionIndex are shown in Algorithm 3.

Algorithm 3: OPTIONINDEX CONSTRUCTION ALGORITHM

Input : Bipartite graph: $G = (L, R, E, W)$
Output: Constructed index

```
1  Initialize index[ ][ ] ← ∅, shrink[ ][ ] ← ∅;
2  cRow[ ] ← ComputeRow(0, G);                                  /* Algorithm 2 */
3  for k = 0 to k_max-1 do
4      nRow[ ] ← ComputeRow(k + 1, G);
5      for ω = 0 to ω_max do
6          rs ← +∞, cs ← +∞;
7          if ω + 1 ≤ ω_max then
8              rs ← cRow[ω].size − cRow[ω + 1].size;
9          if k + 1 ≤ k_max then
10             cs ← cRow[ω].size − nRow[ω].size
11         if rs ≤ cs then
12             index[k][ω] ← cRow[ω] − cRow[ω + 1];
13             Put → into shrink[k][ω];
14         else
15             index[k][ω] ← cRow[ω] − nRow[ω];
16             Put ↓ into shrink[k][ω];
17     cRow ← nRow;
18 Shrink the last row of the index;
19 return index and shrink
```

In Algorithm 3, we first initialize the *index* and *shrink* as empty (Line 1). In Line 2, the algorithm computes $(0, \omega)$-core as the initialization of the current processing row *cRow* and deals with each row in the main loop (Lines 4–17). We set the row next to the *cRow* as *nRow* at Line 4. Then, we compressed the storage space for all possible (k, ω)-cores in *cRow* (Lines 5–16). In each inner iteration, we first initialized both of the resulting sizes of the (k, ω)-cell after *row* shrink (*rs*) and *column* shrink (*cs*) as positive infinity. In Lines 7–8, we use RowIndex to shrink the (k, ω)-cell to the $(k, \omega + 1)$-cell and the resulting size of the (k, ω)-cell is reserved in *rs*. Meanwhile, in Lines 9–10, we utilize ColumnIndex to shrink the (k, ω)-cell to the $(k + 1, \omega)$-cell and the resulting size of the (k, ω)-cell is reserved in *cs*. It is obvious that smaller the resulting size of the (k, ω)-cell is, the better the result of compression. Hence, in Lines 11–16, for a specific (k, ω)-cell, if the value of *rs* is no larger than *cs*, we choose RowIndex to compress and put the nodes contained in the (k, ω)-core but not in the $(k, \omega + 1)$-core into (k, ω)-cell, with the corresponding direction "$\rightarrow$" recorded in *shrink*. Otherwise, we select ColumnIndex to compress, and put the nodes contained in the (k, ω)-core but not in the $(k + 1, \omega)$-core into (k, ω)-cell, with the corresponding direction "$\downarrow$" reserved in *shrink*. We deal with each cell the same way one by one. Finally, we shrink the last *row* of the *index* in Line 18 by using Algorithm 2 and then return the resulting OptionIndex with its corresponding direction table *shrink* in Line 19.

OptionIndex Query Algorithm: Based on the pre-computed OptionIndex, we devised an efficient option query algorithm, and the details are shown in Algorithm 4. In Line 1, we first initialize the (k, ω)-core Q as empty. In Lines 2–3, for given k and ω, we locate index[k][ω] and then add the nodes contained in the (k, ω)-cell to Q. At the same time, we obtain the shrink direction d from *shrink*[k][ω] (Line 4). In Lines 6–9, if the direction is "$\rightarrow$", it implies the current (k, ω)-cell adopts the *row* compression. Then, we add the nodes contained in the $(k, \omega + 1)$-cell to Q and then turn to the $(k, \omega + 1)$-cell. In Lines 10–13, if the direction is "$\downarrow$", it suggests that the shrink direction is down the column. Accordingly, the nodes stored in the (k, ω)-cell are added into Q, and then we turn to $(k + 1, \omega)$-cell for

the next iteration. The procedure terminates until the shrink direction is null and finally we return Q as the resulting (k, ω)-core.

Algorithm 4: OPTIONINDEX BASED QUERY ALGORITHM

Input : Bipartite graph: $G = (L, R, E, W)$, degree constraint: k, weight constraint: ω
Output: The (k, ω)-core of G

```
1  Q ← ∅;
2  Locate index[k][ω];
3  Q ← Q ∪ index[k][ω];
4  d ← shrink[k][ω];
5  while d is not null do
6      if d = "→" then
7          ω ← ω + 1;
8          Q ← Q ∪ index[k][ω];
9          d ← shrink[k][ω];
10     else if d = "↓" then
11         k ← k + 1;
12         Q ← Q ∪ index[k][ω];
13         d ← shrink[k][ω];
14 return Q
```

4.3.3. Unionindex

To further reduce index cost, we propose the UnionIndex. The main difference between UnionIndex and OptionIndex is that we compress certain cells both in *row* and *column* directions at the same time to narrow more space. For example, recall that in Table 1, the $(1, 1)$-cell can be shrunk to the $(1, 2)$-cell with compression size 7, or to the $(2, 1)$-cell with compression size 8. However, the compression size can be up to 9 (i.e., all nodes in the graph) if we shrink the $(1, 1)$-cell to both the $(1, 2)$-cell and $(2, 1)$-cell simultaneously with shrink directions "$\rightarrow$" and "$\downarrow$". Thus, we chose both of the two directions to shrink space storage. In detail, we deposited the nodes contained in the $(1, 1)$-core but not in the union set of $(1, 2)$-core and $(2, 1)$-core into the $(1, 1)$-cell with shrink directions "$\rightarrow$" and "$\downarrow$" recorded simultaneously in the direction table.

The pseudo-codes to construct UnionIndex are presented in Algorithm 5. Since the UnionIndex structure is similar to the OptionIndex structure, we only demonstrate the difference from Algorithm 3 for simplicity. In Lines 6–8, for a specific k, if the $(k+1, \omega)$-core is nested to the $(k, \omega+1)$-core, it indicates that the compression size of RowIndex is larger than that of ColunmIndex. Thus, we put the nodes contained in the (k, ω)-core but not in the $(k, \omega+1)$-core into the (k, ω)-cell with shrink direction "$\rightarrow$". On the contrary, if the $(k+1, \omega)$-core contained the $(k, \omega+1)$-core, we deposited the nodes included in the (k, ω)-core but not in the $(k+1, \omega)$-core into the (k, ω)-cell with shrink direction "$\downarrow$" in Lines 9–11. Otherwise, in Lines 12–14, we compress the (k, ω)-cell to the $(k+1, \omega)$-cell and the $(k, \omega+1)$-cell at the same time with shrink directions "$\rightarrow$" and "$\downarrow$" recorded in the direction table, by avoiding the redundant storage of nodes in the union set of the $(k+1, \omega)$-core and the $(k, \omega+1)$-core. Finally, the algorithm returns UnionIndex with its corresponding direction table *shrink* in Line 17.

Algorithm 5: UNIONINDEX CONSTRUCTION ALGORITHM

Input : Bipartite graph: $G = (L, R, E, W)$
Output: Constructed index

Initialize $index[\,][\,] \leftarrow \emptyset$, $shrink[\,][\,] \leftarrow \emptyset$;
$cRow[\,] \leftarrow$ COMPUTEROW$(0, G)$;
for k = 0 *to* k_{max}-1 **do**
  $nRow[\,] \leftarrow$ COMPUTEROW$(k+1, G)$;
  for ω = 0 *to* ω_{max} **do**
    if $nRow[\omega] \subset cRow[\omega+1]$ **then**
      $index[k][\omega] \leftarrow cRow[\omega] - cRow[\omega+1]$;
      Put "$\rightarrow$" into $shrink[k][\omega]$;
    else if $cRow[\omega+1] \subset nRow[\omega]$ **then**
      $index[k][\omega] \leftarrow cRow[\omega] - nRow[\omega]$;
      Put "$\downarrow$" into $shrink[k][\omega]$;
    else
      $index[k][\omega] \leftarrow cRow[\omega] - (cRow[\omega+1] \cup nRow[\omega])$;
      Put "$\rightarrow$" and "$\downarrow$" into $shrink[k][\omega]$;
  $cRow \leftarrow nRow$;
Compress the last *row* of the *index*;
return *index and shrink*

UnionIndex Query Algorithm: The procedure for querying UnionIndex is simple and the details are shown as follows. When given two query parameters k and ω, we first locate the (k, ω)-cell and collect the nodes stored inside it. Then, we obtain the corresponding shrink direction in the direction table *shrink*, which is obtained with Algorithm 5. If the direction is only "$\rightarrow$" (resp. "$\downarrow$"), then we locate the $(k, \omega+1)$-cell (resp. $(k+1, \omega)$-cell) and collect the nodes contained inside it. Particularly, if there are two shrink directions "$\rightarrow$" and "$\downarrow$" recorded in the *shrink*, we visit the $(k+1, \omega)$-cell and the $(k, \omega+1)$-cell at the same time, collecting their nodes together without duplications. We did the same for all visited cells until the current shrink direction was null and finally we outputted all the collected nodes as the resulting (k, ω)-core.

5. Experiments

In this section, we detail experiments over six real-life networks to verify the performance of the proposed methods.

5.1. Experiment Setup

Algorithms: In the experiments, we implemented and evaluated the algorithms as follows.

- GCore. The baseline algorithm i.e., Algorithm 1.
- BL. The baseline index-based solution.
- TI. The time-improved index algorithm.
- TI+Row. The time-improved algorithm that is integrated with the RowIndex structure.
- TI+Option. The time-improved algorithm that is integrated with the OptionIndex structure.
- TI+Union. The time-improved algorithm that is integrated with the UnionIndex structure.

Datasets: We employed six real-life networks, i.e., Pedia, Movielens, News, Quote, Books and Citeulike, which have been widely used in previous studies (e.g., [2,3,15]) and are publicly available at http://konect.cc/networks/ (accessed on 20 May 2021). Table 2 provides the statistical details of the datasets. For a query with a given pair of parameters k

and ω, we ran the algorithms over each dataset 200 times and reported the average value. All the programs were implemented in C++ and the experiments were performed on a PC with an Intel Xeon 3.2 GHz CPU and 32 GB RAM.

Table 2. Statistics of datasets.

Dataset	L Layer	R Layer	Edges
Pedia	6325	24,952	25,039
Movielens	7588	16,528	63,135
News	1408	25,138	105,039
Quote	21,607	94,756	232,924
Books	32,583	134,942	432,092
Citeulike	22,715	791,763	1,531,769

5.2. Performance Evaluation

To evaluate the efficiency, we compare the response time and space storage of the algorithms on the datasets as follows.

Efficiency of the time-improved algorithm: We firstly compared the baseline solution (BL) with the time-improved algorithm (TI) over all the datasets for index construction. The results are shown in Figure 3. It is obvious that TI runs much faster than BL, since BL needs to compute each subgraph from scratch. In particular, TI significantly outperforms BL in large graphs. For instance, in the Books dataset, TI can achieve a speed that is up to 42X faster.

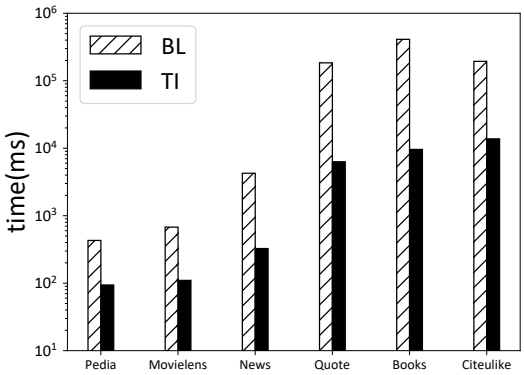
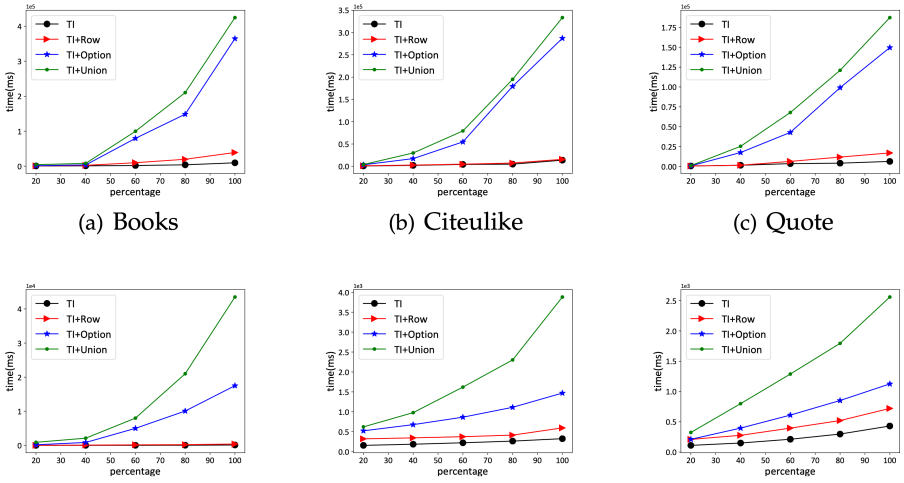


Figure 3. Efficiency evaluation of the time-improved algorithm.

Evaluation of index-construction time: To evaluate the performance of different strategies, in Figure 4, we report the index construction time of TI, TI+Row, TI+Option and TI+Union. We vary the percentage of nodes selected as the input graph. As expected, more nodes will lead to a higher index construction time. TI+Union is slower than the other methods, since it is the most complex method for index construction. The rank of the time costs for the three index construction methods is: RowIndex < OptionIndex < UnionIndex. Note that the highest time cost gap is in the order of seconds, which is tolerable for many applications.

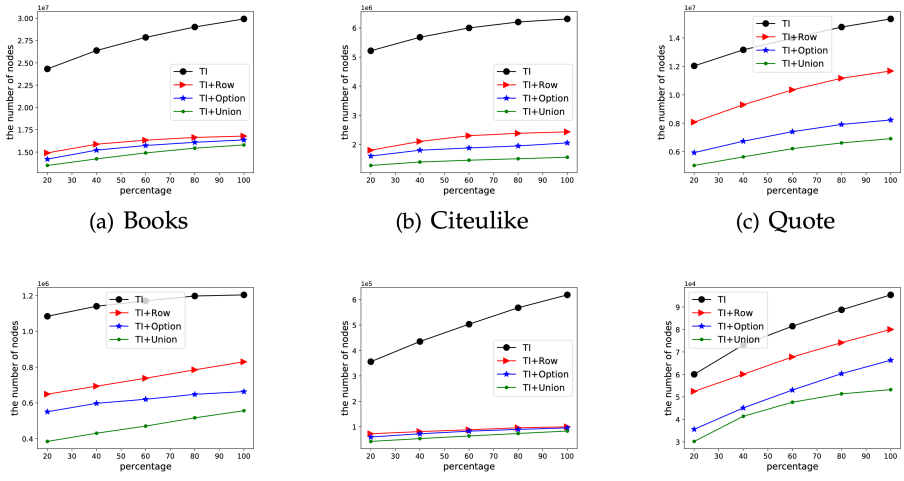


(a) Books (b) Citeulike (c) Quote

(d) Movielens (e) News (f) Pedia

Figure 4. Evaluation of index-construction time.

Evaluation of the index space: In this experiment, the space costs are compared among the four index-construction algorithms, i.e., TI, TI+Row, TI+Option and TI+Union. Note that the space storage is measured by the number of nodes stored inside the index. Similarly, we vary the percentage of nodes in each dataset. The results are shown in Figure 5. As observed, with the increase in nodes involved, more index space is required for all the algorithms. Obviously, the space cost of TI+Union is much less than that of TI, which can save up to 7X space in the News dataset. As expected, TI+Union greatly outperforms TI+Row and TI+Option, for it can omit the largest number of unpromising copies of nodes. The rank of the space cost for these methods is: UnionIndex < OptionIndex < RowIndex.

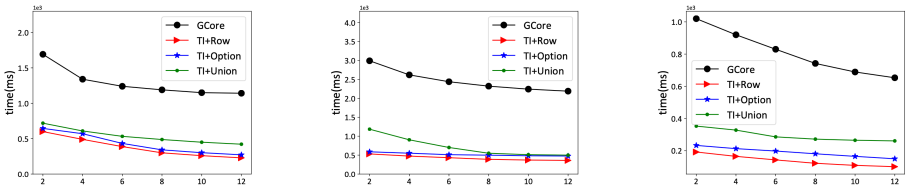


(a) Books (b) Citeulike (c) Quote

(d) Movielens (e) News (f) Pedia

Figure 5. Evaluation of the index space.

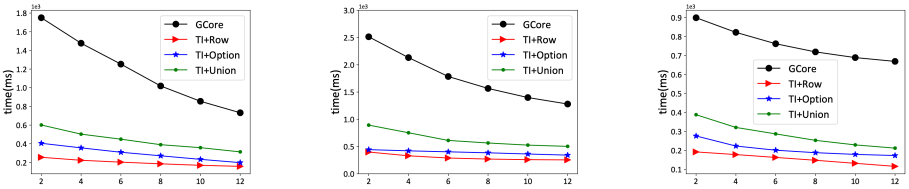
Effect of k in (k, ω)-core queries: To evaluate the querying performance of proposed techniques, we report the response time of GCore, TI+Row, TI+Option and TI+Union algorithms on the three largest datasets by varying k. The results are shown in Figure 6. As shown, with the increase in k, the response time of each algorithm decreases. This is because the returned densely frequent community size become smaller when the degree constraint k becomes tighter. Moreover, there is no doubt that all the index-based query algorithms run much faster than GCore for all k values. The main reason is that the index-based algorithms pre-compute the (k, ω)-core information, so that we can quickly obtain the related nodes when querying any (k, ω)-core.



(a) Books(vary k) (b) Citeulike(vary k) (c) Quote(vary k)

Figure 6. Effect of k in (k, ω)-core queries.

Effect of ω in (k, ω)-core queries: In Figure 7, we report the response time of GCore, TI+Row, TI+Option and TI+Union by varying ω. With the increase in ω, the response time decreases for all the algorithms, since the size of the detected cohesive subgraph decreases accordingly. The index-based solutions are much faster than the online solution, i.e., GCore. As shown, more complex index structures, such as TI+Union, will lead to a higher computation cost. Therefore, users can make a trade-off between the querying time and space cost when selecting the index strategies.



(a) Books(vary ω) (b) Citeulike(vary ω) (c) Quote(vary ω)

Figure 7. Effect of ω in (k, ω)-core queries.

Discussion: According to the results of the experiment, we can find that the index construction time grows with the increase in dataset size. However, the increase rates of OptionIndex and UnionIndex grow much faster than RowIndex. This is because, for larger datasets, it usually means larger k_{max} and ω_{max}. Therefore, with the increase in k_{max} and ω_{max}, OptionIndex and UnionIndex need to take more time to decide the best direction for index construction in order to shrink the index space. When selecting the appropriate solution, in addition to the index space issue, users should pay more attention to k_{max} and ω_{max} of the used networks.

6. Related Work

Graphs are widely used to model the complex relationships between entities [16]. As a special graph, many real-life systems are modeled in bipartite graphs, such as author–paper networks [17], customer–product networks [18] and gene co-expression networks [19]. Bipartite graph analysis is of great importance and has attracted great attention in the literature. Guillaume et al. show that all complex networks can be viewed as bipartite structures sharing some important statistics, such as degree distributions [20]. In [21], Kannan et al. utilize simple Markov chains for the problem of generating labeled bipartite graphs with a given degree sequence. Borgatti et al. present and discuss ways of applying and interpreting traditional network analysis techniques to two-mode data [22].

Cohesive subgraph identification is a fundamental problem in graph analysis, and different models are proposed, such as k-core [23], k-truss [24] and clique [25]. Due to the unique properties of bipartite graphs, many studies are conducted to design and investigate the cohesive subgraph models for bipartite graphs, such as (α, β)-core, bitruss and biclique. Ahmed et al. [26] are the first to formally propose and investigate the (α, β)-core model. The authors of [3] further extend the linear k-core mining algorithm to compute the (α, β)-core. In [4], the authors combine the influence property with (α, β)-core for community detection. Considering the structure properties, Zou et al. [9] propose the bitruss model, where each edge in the community is contained in at least k butterflies. To further study the clustering ability in bipartite graphs, Flajolet et al. [27] use the ratio of the

number of butterflies to the number of three paths for modeling the cohesiveness of the graph. In [28], Robins et al. resort to the $(2,2)$-biclique to model the cohesion. In [10], a progressive method is proposed to speed up the computation of biclique. As we can see, the previous studies do not consider the weight factor for cohesive subgraph identification. Thus, in this paper, we propose (k,ω)-core to capture the weight property for bipartite network analysis. Even though we can extend the computation procedure of (α,β)-core for (k,ω)-core identification (i.e., online solution), it cannot handle large graphs and different parameters efficiently. Therefore, in this paper, we propose index-based solutions with different optimization strategies to deal with this issue.

7. Conclusions and Future Work

In this paper, we introduce a novel cohesive subgraph model (k,ω)-core for weighted bipartite graph analysis. A baseline online solution is first presented by extending the method for (α,β)-core computation. To handle massive graphs and queries, index-based strategies are developed by using the nested property. To balance the query performance and space cost, three advanced index structures are further introduced. Finally, we conduct extensive experiments on real-world datasets to evaluate the performance of the proposed techniques. In future work, we will consider the external algorithms or distributed solutions for (α,β)-core identification in order to support larger networks.

Author Contributions: Conceptualization, X.L.; methodology, X.L.; software, X.W.; validation, X.L.; formal analysis, X.W.; investigation, X.L.; resources, X.W.; data curation, X.L.; writing—original draft preparation, X.W.; writing—review and editing, X.W.; visualization, X.L.; supervision, X.W.; project administration, X.W. Both authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by NSFC 61802345, ZJNSF LQ20F020007, ZJNSF LY21F020012 and Y202045024.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Publicly available datasets were used in this study. The data can be downloaded from: http://konect.cc/networks/ (accessed on 20 May 2021).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Wang, H.; Lian, D.; Liu, W.; Wen, D.; Chen, C.; Wang, X. Powerful graph of graphs neural network for structured entity analysis. *World Wide Web* **2021**. [CrossRef]
2. Chen, C.; Zhu, Q.; Wu, Y.; Sun, R.; Wang, X.; Liu, X. Efficient critical relationships identification in bipartite networks. *World Wide Web* **2021**. [CrossRef]
3. Liu, B.; Yuan, L.; Lin, X.; Qin, L.; Zhang, W.; Zhou, J. Efficient (α, β)-core computation: An index-based approach. In Proceedings of the World Wide Web Conference, San Francisco, CA, USA, 13–17 May 2019; pp. 1130–1141.
4. Wang, K.; Zhang, W.; Lin, X.; Zhang, Y.; Qin, L.; Zhang, Y. Efficient and effective community search on large-scale bipartite graphs. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; pp. 85–96.
5. Chen, C.; Wu, Y.; Sun, R.; Wang, X. Maximum Signed Θ-Clique Identification in Large Signed Graphs. *IEEE Trans. Knowl. Data Eng.* **2021**. [CrossRef]
6. Sun, R.; Chen, C.; Wang, X.; Zhang, Y.; Wang, X. Stable community detection in signed social networks. *IEEE Trans. Knowl. Data Eng.* **2020**. [CrossRef]
7. Zhao, J.; Sun, R.; Zhu, Q.; Wang, X.; Chen, C. Community identification in signed networks: A k-truss based model. In Proceedings of the 29th ACM International Conference on Information & Knowledge Management, Turin, Italy, 22–26 October 2020.
8. Sun, R.; Chen, C.; Wang, X.; Wu, Y.; Zhang, M.; Liu, X. The art of characterization in large networks: Finding the critical attributes. *World Wide Web* **2021**. [CrossRef]
9. Zou, Z. Bitruss decomposition of bipartite graphs. In Proceedings of the International Conference on Database Systems for Advanced Applications, Dallas, TX, USA, 16–19 April 2016; pp. 218–233.
10. Lyu, B.; Qin, L.; Lin, X.; Zhang, Y.; Qian, Z.; Zhou, J. Maximum biclique search at billion scale. In Proceedings of the VLDB Endowment, Tokyo, Japan, 31 August–4 September 2020.

11. Poernomo, A.K.; Gopalkrishnan, V. Towards efficient mining of proportional fault-tolerant frequent itemsets. In Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, 28 June–1 July 2009; pp. 697–706.
12. Cerinšek, M.; Batagelj, V. Generalized two-mode cores. *Soc. Netws* **2015**, *42*, 80–87. [CrossRef]
13. Ding, D.; Li, H.; Huang, Z.; Mamoulis, N. Efficient fault-tolerant group recommendation using alpha-beta-core. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, Singapore, 6–10 November 2017; pp. 2047–2050.
14. Bi, F.; Chang, L.; Lin, X.; Zhang, W. An Optimal and Progressive Approach to Online Search of Top-K Influential Communities. In Proceedings of the VLDB Endowment, Rio de Janeiro, Brazil, 27 August 2018; Volume 11.
15. Zhu, Q.; Zheng, J.; Yang, H.; Chen, C.; Wang, X.; Zhang, Y. Hurricane in Bipartite Graphs: The Lethal Nodes of Butterflies. In Proceedings of the 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, 7–9 July 2020; pp. 1–4.
16. Zhao, X.; Xiao, C.; Lin, X.; Zhang, W.; Wang, Y. Efficient structure similarity searches: A partition-based approach. *VLDB J.* **2018**, *27*, 53–78. [CrossRef]
17. Beutel, A.; Xu, W.; Guruswami, V.; Palow, C.; Faloutsos, C. Copycatch: Stopping group attacks by spotting lockstep behavior in social networks. In Proceedings of the 22nd International Conference on World Wide Web, Rio de Janeiro, Brazil, 13–17 May 2013; pp. 119–130.
18. Wang, J.; De Vries, A.P.; Reinders, M.J. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, New York, NY, USA, 6–11 August 2006; pp. 501–508.
19. Kaytoue, M.; Kuznetsov, S.O.; Napoli, A.; Duplessis, S. Mining gene expression data with pattern structures in formal concept analysis. *Inf. Sci.* **2011**, *181*, 1989–2001. [CrossRef]
20. Guillaume, J.L.; Latapy, M. Bipartite structure of all complex networks. *Inf. Process. Lett.* **2004**, *90*, 215–221. [CrossRef]
21. Kannan, R.; Tetali, P.; Vempala, S. Simple Markov-chain algorithms for generating bipartite graphs and tournaments. *Random Struct. Algorithms* **1999**, *14*, 293–308. [CrossRef]
22. Borgatti, S.P.; Everett, M.G. Network analysis of 2-mode data. *Soc. Netw.* **1997**, *19*, 243–269. [CrossRef]
23. Chen, C.; Zhu, Q.; Sun, R.; Wang, X.; Wu, Y. Edge Manipulation Approaches for K-core Minimization: Metrics and Analytics. *IEEE Trans. Knowl. Data Eng.* **2021**. [CrossRef]
24. Chen, C.; Zhang, M.; Sun, R.; Wang, X.; Zhu, W.; Wang, X. Locating pivotal connections: The K-Truss minimization and maximization problems. *World Wide Web* **2021**. [CrossRef]
25. Sun, R.; Zhu, Q.; Chen, C.; Wang, X.; Zhang, Y.; Wang, X. Discovering cliques in signed networks based on balance theory. In Proceedings of the International Conference on Database Systems for Advanced Applications, Jeju, Korea, 24–27 September 2020; pp. 666–674.
26. Ahmed, A.; Batagelj, V.; Fu, X.; Hong, S.H.; Merrick, D.; Mrvar, A. Visualisation and analysis of the Internet movie database. In Proceedings of the International Asia-Pacific Symposium on Visualization, Sydney, NSW, Australia, 5–7 February 2007; pp. 17–24.
27. Lind, P.G.; Gonzalez, M.C.; Herrmann, H.J. Cycles and clustering in bipartite networks. *Phys. Rev. E* **2005**, *72*, 056127. [CrossRef] [PubMed]
28. Robins, G.; Alexander, M. Small worlds among interlocking directors: Network structure and distance in bipartite graphs. *Comput. Math. Organ. Theory* **2004**, *10*, 69–94. [CrossRef]