Li Y, Dai J, Fan XL *et al.* I/O efficient early bursting cohesive subgraph discovery in massive temporal networks. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 37(6): 1337–1355 Nov. 2022. DOI 10.1007/s11390-022-2367-3

I/O Efficient Early Bursting Cohesive Subgraph Discovery in Massive Temporal Networks

Yuan Li[1] (李　源), *Member*, *CCF*, Jie Dai[1,2] (戴　捷), *Student Member*, *CCF*
Xiao-Lin Fan[1] (范晓林), *Student Member*, *CCF*, Yu-Hai Zhao[2,*] (赵宇海), *Senior Member*, *CCF*, and Guo-Ren Wang[3] (王国仁), *Senior Member*, *CCF*

[1] *School of Information Science and Technology, North China University of Technology, Beijing 100144, China*
[2] *School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China*
[3] *School of Computer, Beijing Institute of Technology, Beijing 100081, China*

E-mail: liyuan@ncut.edu.cn; {18151010205, 2019312170106}@mail.ncut.edu.cn; zhaoyuhai@mail.neu.edu.cn
wanggr@bit.edu.cn

Received March 30, 2022; accepted November 8, 2022.

Abstract Temporal networks are an effective way to encode temporal information into graph data losslessly. Finding the bursting cohesive subgraph (BCS), which accumulates its cohesiveness at the fastest rate, is an important problem in temporal networks. The BCS has a large number of applications, such as representing emergency events in social media, traffic congestion in road networks and epidemic outbreak in communities. Nevertheless, existing methods demand the BCS lasting for a time interval, which neglects the timeliness of the BCS. In this paper, we design an early bursting cohesive subgraph (EBCS) model based on the k-core to enable identifying the burstiness as soon as possible. To find the EBCS, we first construct a time weight graph (TWG) to measure the bursting level by integrating the topological and temporal information. Then, we propose a global search algorithm, called GS-EBCS, which can find the exact EBCS by iteratively removing nodes from the TWG. Further, we propose a local search algorithm, named LS-EBCS, to find the EBCS by first expanding from a seed node until obtaining a candidate k-core and then refining the k-core to the result subgraph in an optimal time complexity. Subsequently, considering the situation that the massive temporal networks cannot be completely put into the memory, we first design an I/O method to build the TWG and then develop I/O efficient global search and local search algorithms, namely I/O-GS and I/O-LS respectively, to find the EBCS under the semi-external model. Extensive experiments, conducted on four real temporal networks, demonstrate the efficiency and effectiveness of our proposed algorithms. For example, on the DBLP dataset, I/O-LS and LS-EBCS have comparable running time, while the maximum memory usage of I/O-LS is only 6.5 MB, which is much smaller than that of LS-EBCS taking 308.7 MB.

Keywords early bursting cohesive subgraph (EBCS), I/O efficient algorithm, semi-external model, temporal network

1 Introduction

Temporal networks are composed of nodes and edges associated with timestamps, where nodes and edges represent the entities and the relationships between entities, respectively. Moreover, the edges may have time-varying weight which reflects the strength of the connections at different timestamps. As such, temporal networks are becoming essential to describe the dynamics of complex matters in the real world [1].

Cohesive subgraph mining (CSM) in temporal networks is a fundamental problem, which aims to discover the densely-connected regions that meanwhile fulfill the temporal constraints of the network. According to different variations of temporal patterns, the studies can be broadly classified into three categories, including continuous [2–6], periodic [7, 8] and bursting [9, 10] cohesive subgraph mining. Among them, the burst-

Regular Paper
This work was supported by the National Natural Science Foundation of China under Grant Nos. 61902004, 61772124, 61732003, and 61977001, the Project of Beijing Municipal Education Commission under Grant No. KM202010009009, Innovative Talents of Higher Education in Liaoning Province under Grant No. LR2020076, and the Basic Research Operating Funds for National Defense Major Incubation Projects under Grant No. N2116017.
*Corresponding Author

ing cohesive subgraph (BCS) is a subgraph that can accumulate its cohesiveness at the fastest rate in temporal networks. It represents the sudden appearance of real-world matters and has many practical applications. For example, the BCS can represent an emergency disaster (e.g., earthquake [11]) that is suddenly and widely spread in social media, a traffic congestion where there is a sudden and dramatic increase in the number of vehicles waiting at intersections or along the streets in road networks [12], and an epidemic outbreak like COVID-19 having a rapid spread in the community-based network [13].

Motivation. Although the BCS has a large number of applications in real temporal networks, there are not many studies on the BCS. Qin *et al.* [9] proposed the (l, δ)-maximal dense core model, which is a temporal subgraph where each node has an average degree no less than δ in a time segment with a length no less than l. Chu *et al.* [10] proposed the density bursting subgraph (DBS), which is a subgraph that accumulates its density at the fastest speed in an arbitrarily long time duration. One commonality in the above two models is that they both require the subgraphs lasting for at least a time interval, e.g., l. However, this requirement neglects the timeliness and avoids the bursting subgraphs being discovered earlier.

Our Model. Intuitively, the earlier an emergency event is detected, the better it can be handled. Therefore, in this paper, we propose the early bursting cohesive subgraph (EBCS) model, which can support the discovery of bursting events from temporal networks in a timely manner. Specifically, an EBCS is defined as the maximal connected subgraph that meets both the cohesiveness and the burstiness criteria. Here, we use the commonly-used minimum degree [14, 15] to measure the cohesiveness of the EBCS. And for the measurement of the burstiness of the EBCS, we customize a subgraph burstiness (detailed in Definition 7), which is the smallest time weight of the node in the subgraph. And the time weight is computed as the product of the burst rate and the increment between two moments, i.e., $\frac{\triangle|\triangle|}{T}$, where $\triangle$ denotes the increment and T denotes the interval between two moments.

For example, Fig.1 shows a toy temporal road network, where the nodes represent locations and the edges represent roads connecting different locations. Edge

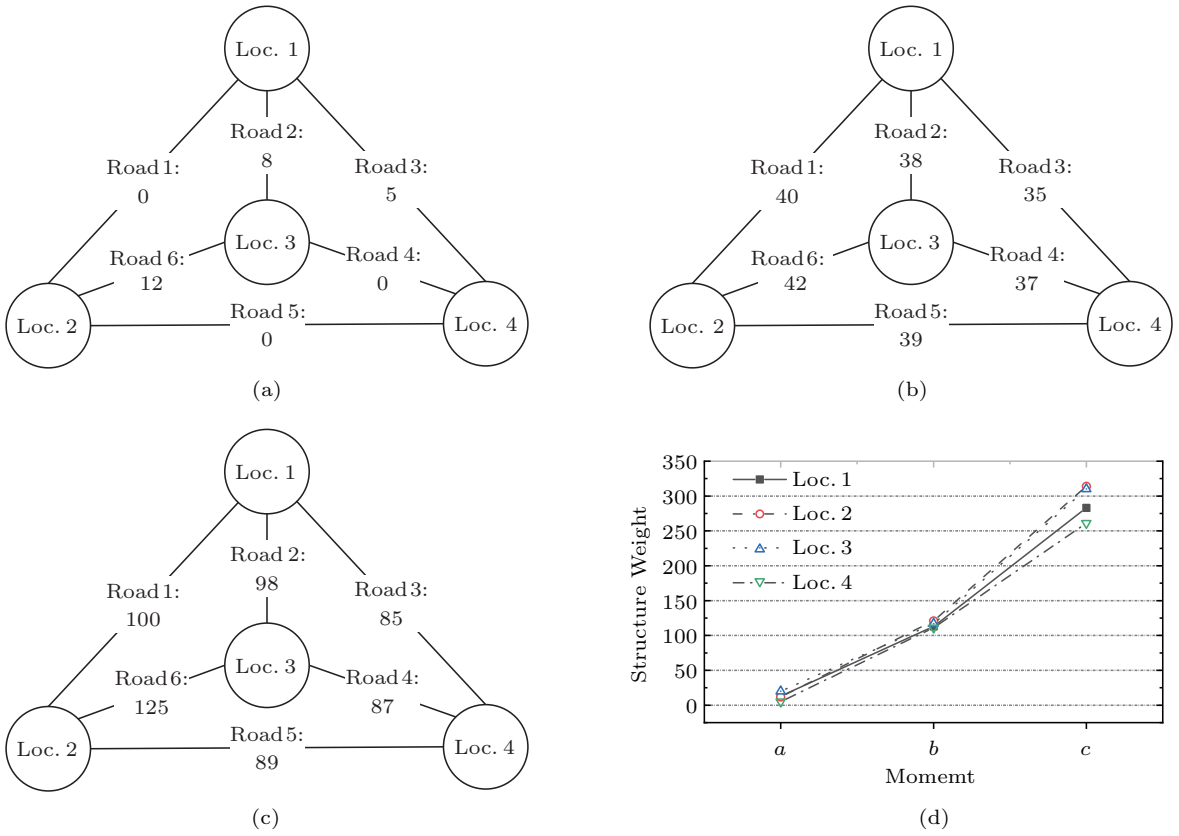


Fig.1. A toy example of the EBCS in a temporal road network. (a) Moment a. (b) Moment b. (c) Moment c. (d) Change rates in the structure weights of nodes. Loc.: location.

weights at different moments denote the number of vehicles waiting on the roads. Let us consider the graph S composed of {Loc. 1, Loc. 2, Loc. 3, Loc. 4}. The structure weights (detailed in Definition 4) of nodes corresponding to moments a and b are $\{13, 12, 20, 5\}$ and $\{113, 121, 117, 111\}$ respectively. And the structure weight of a node is calculated as the sum of its adjacent edge weights. Now, corresponding to moment a, the time weights of the nodes of S at moment b are $\{\frac{100^2}{b-a}, \frac{109^2}{b-a}, \frac{97^2}{b-a}, \frac{106^2}{b-a}\}$, and S is an EBCS with the minimum degree of 3 and burstiness of $97^2/(b-a)$, which indicates that a slight traffic jam has occurred on these roads. In other words, once we can detect the EBCS at moment b (shown in Fig.1(b)), we are able to take measures in time to avoid a worse traffic jam at moment c (shown in Fig.1(c)).

Challenges. Actually, finding the EBCS in massive temporal networks is not a trivial task, and we mainly face the following two challenges. 1) In temporal networks, edges and edge weights are always changing with time and the number of edges is massive. Thus, how to measure the burstiness of the subgraph in this variation is a big challenge. 2) Because the EBCS has an urgent need for timeliness to ensure the bursting events being identified as soon as possible, we have to design efficient algorithms to find the EBCS. Furthermore, the situation will be much worse when the massive temporal networks cannot be entirely loaded into the memory.

Our Solutions. Since nodes are more stable than the massive evolving edges in temporal networks, we first transform the edge weights in temporal networks into node weights, by accumulating the weights of edges to their connected nodes. At the same time, we define the structure weight of each node as its node weight, which is robust to outliers. Further, by exploiting the sliding-window technique, we can calculate the time weight (burstiness) of each node in the current moment network through the burst rate and the increment of the structure weight compared with the network in the most recent sliding-window. In this way, we can perceive the burstiness of the current moment network, named as time weight graph (TWG), in time. To find the EBCS in TWG, 1) we propose a global search algorithm, called GS-EBCS, which iteratively removes the nodes with the smallest time weight while maintaining the structure constraints. 2) We further propose a local search algorithm with the idea of expanding and refining [16], called LS-EBCS. Specifically, LS-EBCS first expands the subgraph from the node with the largest time weight and then refines the discovered subgraph to satisfy the constraints of the EBCS, which is optimal.

On top of our conference paper [16], we improve GS-EBCS and LS-EBCS. After the two algorithms find the k-core, instead of the original method that updates the degrees of the remaining nodes every time when a node is deleted, we develop a batch-deletion strategy that includes a multiplication operation and an in-half operation. In this way, the algorithms only update the degrees of the remaining nodes once after one batch. The details of this approach are described in Subsection 4.2. In addition, we further consider the situation that the massive temporal networks cannot be fully loaded into the memory. We design I/O efficient approaches to discover the EBCS based on the semi-external model, which only requires the node information to be loaded in the memory. To the best of our knowledge, this is the first work exploiting I/O approaches to solve the CSM problem in massive temporal networks. 1) To adapt to the semi-external model, we propose an I/O time weight graph transformation algorithm, whose I/O complexity is $O(\frac{2|V|+4|E|}{B})$, where $|V|$ denotes the number of nodes, $|E|$ is the number of edges, and B represents the disk block size. 2) To discover the EBCS, we propose the I/O-GS algorithm, which loads all the node information from the disk and finds the nodes that can form the maximum k-core based on this information and then discovers the EBCS among these nodes. The I/O complexity of I/O-GS is $O(\frac{(|V|+|E|)}{B})$. 3) To further reduce the I/O cost as well as the memory usage, we propose the I/O-LS algorithm. Due to I/O-LS preferentially loading nodes with the largest time weight, it rarely loads all the nodes into memory. The I/O complexity of I/O-LS is $O(\frac{\sum_{i=1}^{i=l}(|S(i)|+|E[S(i)]|)+|E[S(l)]|}{B})$, where l denotes the number of expansions, $S(i)$ denotes the number of nodes visited in the i-th expansion, and $E[S(i)]$ represents the edges induced by $S(i)$. In fact, except for the I/O time weight graph transformation algorithm, it is not necessary to load the adjacent edges of each node. Therefore, the actual I/O and memory consumption could be even smaller. For example, on the DBLP dataset, I/O-LS and LS have comparable running time, while the maximum memory usage of I/O-LS is only 6.5 MB, which is much smaller than that of LS, 308.7 MB.

Our contributions about this paper are as follows.

• We propose an early bursting cohesive subgraph (EBCS) model in massive temporal networks, which enables the discovery of bursting matters in a timely

manner.

• We develop two in-memory EBCS discovery algorithms, called GS-EBCS and LS-EBCS, respectively. GS-EBCS finds the EBCS by iteratively removing nodes from the graph, while the LS-EBCS algorithm exploits the idea of expanding and refining, and in general, does not need to visit the whole graph.

• We further propose the I/O time weight graph transformation algorithm, the I/O-GS algorithm and the I/O-LS algorithm based on the semi-external model. Among them, I/O-LS has the best performance and further reduces the I/O cost and memory usage. To the best of our knowledge, this is the first study to use I/O methods to deal with CSM in massive temporal networks.

In the end, extensive experiments are conducted on four real temporal networks to demonstrate the efficiency and effectiveness of our proposed algorithms. The organization is as follows. Section 2 introduces the related work. Section 3 gives the concepts and the problem definition. Sections 4 and 5 detail the in-memory and I/O efficient algorithms, respectively. Section 6 demonstrates the experiments. Finally, we conclude the paper in Section 7.

2 Related Work

Our work is related to the topics of cohesive subgraph mining (CSM) in weighted networks and temporal networks, and I/O efficient graph processing.

CSM in Weighted Networks. The weighted networks can be divided into node-weighted networks and edge-weighted networks. For CSM in node-weighted networks, Li *et al.*[17,18] proposed to find the top-r k-cores with the largest minimum node weight as the influential community. For the same problem, Chen *et al.*[19] proposed a more efficient method, which only needs to check the connected components relevant with the top-r results. Furthermore, from the local perspective, Bi *et al.*[20] proposed a progressive local search algorithm, which does not need to visit the whole structure of the network. For CSM in edge-weighted networks, Zheng *et al.*[21] developed a method to find the k-truss with the largest minimum edge weight. Sun *et al.*[22] proposed an index-based method to find the k-core with the smallest group weight. In this work, we transform the temporal networks with edge weights into time weight graphs with node weights, which are used to measure the burstiness of subgraphs.

CSM in Temporal Networks. Based on different temporal patterns, the work of CSM in temporal networks can be further categorized as continuous CSM, periodic CSM and bursting CSM. For persistent CSM, Li *et al.*[2] first proposed to find the maximum (θ,τ) persistent k-core in a temporal network, which can capture the persistence of a cohesive subgraph. Then, Li *et al.*[6] studied the personalized version of the (θ,τ) persistent k-core problem, which identifies the (θ,τ)-continual k-core containing one query node. Qin *et al.*[4] proposed a density-based clustering problem to detect stable communities in temporal graphs, which are required to appear for a certain time. Semertzidis *et al.*[3] proposed methods to find the densest set of nodes aggregated in at least k graph snapshots. To enhance the diversity of results, Lin *et al.*[5] formulated a diversified lasting cohesive subgraphs problem to find top-r maximal lasting (k,p)-cores with maximum coverage regarding the number of vertices and timestamps. For periodic CSM, Lahiri and Berger-Wolf[23] proposed to find periodic or near periodic subgraphs in temporal networks. Qin *et al.*[7,24] proposed to find the maximal σ-periodic k-clique, which is a clique with a size larger than k that appears at least σ times periodically in temporal graphs. In addition, Zhang *et al.*[8] considered the seasonal feature of the periodic subgraph. For bursting CSM, Chu *et al.*[10] proposed methods to find the density bursting subgraph in temporal networks. Qin *et al.*[9] proposed a maximal dense core model to represent the bursting cohesive subgraph in temporal networks. However, both of the existing bursting CSM methods required the bursting subgraphs lasting for a certain time interval. In this paper, we study the EBCS problem, which takes the timeliness of the bursting cohesive subgraph into account.

I/O Graph Processing. I/O is an effective way for massive graph data processing[25]. Cheng *et al.*[26] first proposed the external-memory algorithm for core decomposition in massive graphs. Sun *et al.*[27] proposed GraphMP to tackle big graph analysis on a single machine by reducing the disk I/O overhead. Recently, the semi-external model has been extensively studied for massive graph processing, due to its power on limiting memory properties. In the semi-external model, only nodes can be loaded in memory while edges are stored on disk. Wen *et al.*[28] studied the I/O efficient core decomposition following the semi-external model. Yuan *et al.*[29] proposed I/O efficient methods to compute k-edge connected components (k-ECC) decomposition via graph reduction. Zhang *et al.*[30] proposed I/O-efficient semi-external algorithms to find all strongly connected components (SCC) for a massive

directed graph. Sun *et al.*[27] proposed a new EP-SCC algorithm to further optimize the in-memory processes. Jiang *et al.*[31] designed a semi-external method to find k-turss communities in massive graphs. Li *et al.*[18] proposed a novel I/O-efficient algorithm to find the top-r k-influential communities. In this paper, we propose I/O efficient algorithms to find the EBCS by the semi-external model.

3 Problem Formulation

In this section, we introduce the relevant notations and definitions over the EBCS, and give the specific problem definition.

3.1 General Concepts

Given a temporal network, it consists of a set of snapshot graphs with continuous timestamps, and each snapshot graph can be abstractly represented as $G_x(V, E(x), A(x), t_x)$, where V represents the set of nodes, $|V|$ denotes the number of nodes in V, and each G_x has the same V. $t_x \in \{t_i, t_{i+1}, \ldots, t_j\}$ denotes the timestamps. At timestamp t_x, $e_{(u,v)}$ denotes the undirected edge between two nodes $u, v \in V$. $E(x)$ denotes the set of edges at timestamp t_x and $|E(x)|$ denotes the number of edges. $ew_{(u,v)} \geqslant 0$ denotes the edge weight between two nodes u, v, where $ew \in \boldsymbol{A}(x)$, and $\boldsymbol{A}(x)$ is a $|V| \times |V|$ matrix storing the weight between any two nodes. And $ew_{(u,v)} = 0$ means there is no edge between two nodes u, v. Therefore, it is also possible to represent the set of edges by $\boldsymbol{A}(x)$. For a node u of G_x, $nbr_{G_x}(u)$ denotes the set of neighbor nodes of node u in G_x, $deg_{G_x}(u)$ denotes the degree of node u in G_x, and $|nbr_{G_x}(u)| = deg_{G_x}(u)$. In addition, we use $size(G_x) = |V| + |E(x)|$ to represent the size of G_x.

Graph Storage. For each G_x, we use two tables for storage on disk. One is the node table, which stores the nodes and their corresponding degrees, and the other is the edge table, which stores the edges and edge weights. For example, for the edge table of Fig. 2(e), each row is shaped as $(v_1, v_2, v_3, v_4, 5, 7, 3)$. It represents the neighbor nodes of node v_1 and their corresponding edge weights.

Definition 1 (Induced Subgraph). *Given a set $S \subseteq V$, then the graph consisting of S and its connected edges in G_x is called the induced subgraph of G_x, denoted as $G_x[S]$. $E(x)[S]$ represents edges induced by S.*

Definition 2 (Subgraph Goodness). *For an induced subgraph $G_x[S]$, the subgraph goodness of $G_x[S]$ can be defined by the minimum degree*:

$$\delta(G_x[S]) = \min\left\{deg_{G_x[S]}(v) \mid v \in S\right\}.$$

Definition 3 (K-C Subgraph). *Given a positive integer k, if $G_x[S]$ is a connected subgraph and $\delta(G_x[S]) \geqslant k$, then $G_x[S]$ is called as the K-C subgraph, which is denoted as $G_x^k[S]$.*

To solve the challenges in Section 1, we consider reflecting the changes of edges to nodes, i.e., transforming an edge-weighted temporal network to a node-weighted temporal network, which exploits the topology of the network and historical temporal information. And this transformation is implemented in the following two definitions.

Definition 4 (Structure Weight). *For any node u in G_x, the structure weight of node u is the sum of the weights of the edges connected to u, i.e., $sum_x(u) = \sum_{v \in nbr_{G_x}(u)} ew_{(u,v)}$.*

Definition 5 (Time Weight). *Given a positive integer sg and two snapshot graphs G_x and G_{x-sg}, for any node u in G_x, $\triangle = sum_x(u) - sum_{x-sg}(u)$ represents the increment of u in the time interval sg. Thus, the time weight of u is defined as*:

$$twn_x(u) = \frac{\triangle|\triangle|}{sg}. \tag{1}$$

In Definition 5, we use the technique of sliding-window[32] to control the sensitivity of discovering the burst. sg represents the length of the sliding-window. And we make comparison between the snapshot graphs G_x and G_{x-sg}, corresponding to the timestamp t_x and

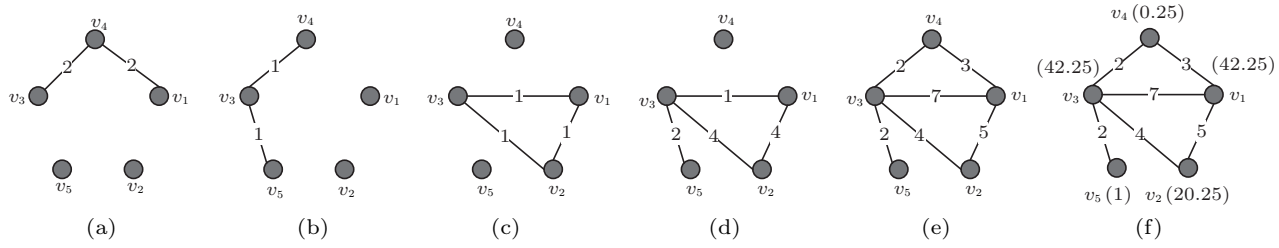


Fig. 2. Example of a temporal network and a TWG. (a) G_1. (b) G_2. (c) G_3. (d) G_4. (e) G_5. (f) $\widetilde{G}_5$ ($sg = 4$).

the timestamp sg moment before t_x, i.e., t_{x-sg}. Therefore, the smaller sg is, the smaller the time interval left for event bursting is, and the higher the sensitivity of discovering the burst is.

In addition, in (1), we use $\triangle/sg$ to represent the burst rate of increment $\triangle$. Then, the product of the burst rate and the increment, i.e., $twn_x(u)$, indicates the severity of the burst. Thus, a larger $twn_x(u)$ indicates a larger burst of u, and then u is of more concern. $twn_x(u)$ is also the final node weight of u.

Definition 6 (Time Weight Graph). *Let node set S contain all the nodes $u \in V$ with $twn_x(u) \geqslant 0$. Then, we call $G_x[S]$ as the time weight graph of G_x, or TWG for short. And TWG is denoted as $\widetilde{G}_x$.*

We use $V(x)$ and $\widetilde{E}(x)$ to denote the set of nodes and the set of edges of $\widetilde{G}_x$ respectively.

Definition 7 (Subgraph Burstiness). *For a $\widetilde{G}_x[M]$, the subgraph burstiness of $\widetilde{G}_x[M]$ is defined as the minimum time weight of the node in $\widetilde{G}_x[M]$, i.e.,*

$$sgb\left(\widetilde{G}_x[M]\right) = \min\left\{twn_x(u) \mid u \in M\right\}.$$

Definition 8 (EBCS). *Given a threshold φ, a positive interger k, and a $\widetilde{G}_x$, let $max\_weight = \max\{twn_x(u) \mid u \in V(x)\}$, and then we can find a node set $S \subseteq V(x)$ such that 1) $max\_weight \geqslant \varphi$; 2) $\widetilde{G}_x[S]$ is a K-C subgraph; 3) among all K-C subgraphs, $sgb\left(\widetilde{G}_x[S]\right)$ is the largest; 4) there exists no other subgraph $\widetilde{G}_x[M]$ such that $\widetilde{G}_x[M]$ is a supergraph of $\widetilde{G}_x[S]$ with $sgb\left(\widetilde{G}_x[M]\right) = sgb\left(\widetilde{G}_x[S]\right)$ and $\widetilde{G}_x[M]$ is also a K-C subgraph, and then $\widetilde{G}_x[S]$ is the EBCS.*

From condition 3, we can infer that the number of nodes of EBCS is the maximum among all subgraphs that satisfy conditions 1 and 2. For any node $v \in V(x)$ with $twn_x(v) = sgb\left(\widetilde{G}_x[S]\right)$, if $v \notin S$, then $deg_{\widetilde{G}_x[S\cup\{v\}]}(v)$ must not satisfy k.

Example 1. We set $k = 2$, $\varphi = 19$, and $sg = 4$. 1) Figs.2(a)–2(e) show a temporal network consisting of five snapshot graphs. 2) The node set $S = \{v_1, v_2, v_3\}$ and its edges in Fig. 2(e) form an induced subgraph $G_5[S]$, and its subgraph goodness is $2 \geqslant k$. Then, $G_5[S]$ is a K-C subgraph according to Definition 3. Actually, $G_5[S \cup \{v_4\}]$ is also a K-C subgraph, and it is the supergraph of $G_5[S]$. 3) According to Definitions 4–6, we transform G_5 into $\widetilde{G}_5$, shown in Fig. 2(f). In G_5 and G_1, the structure weights of v_1 are $3 + 7 + 5 = 15$ and 2 respectively, and thus, the time weight of v_1 in $\widetilde{G}_5$ is $(15 - 2)^2/4 = 42.25$. 4) According to conditions of the EBCS, the subgraph $\widetilde{G}_5[S]$ induced by the node set $S = \{v_1, v_2, v_3\}$ is the EBCS and $sgb(\widetilde{G}_5[S])$ is 20.25.

3.2 Problem Definition

Problem 1. Given an edge-weighted temporal network, a threshold φ, and two positive integers k and sg, we aim to discover the EBCS in $\widetilde{G}_x$ after transforming the snapshot graph G_x into the time weight graph $\widetilde{G}_x$ by integrating the topological and temporal information of the network.

4 In-Memory EBCS Discovery Algorithms

In this section, we introduce the solution when the whole graph can be completely loaded into memory. Specifically, we first introduce the transformation of TWG. Then, we design a global search based EBCS discovery algorithm. Futher, we propose a more efficient local search based EBCS discovery algorithm, in general, which avoids visiting all nodes.

4.1 In-Memory TWG Transformation Algorithm

The in-memory TWG transformation algorithm (IMTA) in Algorithm 1 implements the transformation of edge-weighted snapshot graphs to the TWG. Specifically, based on the topology of the graph, the algorithm first obtains the structure weight of each node at two different timestamps t_i and t_{i-sg}. Then, IMTA calculates the time weight of each node at t_i, which is the node weight, based on the given sg.

Algorithm 1. IMTA(sg, G_i, G_{i-sg})

```
1: Calculate sum_i(v) and sum_{i-sg}(v) for all v ∈ V;
2: for each v ∈ V do
3:     Calculate twn_i(v);
4:     if twn_i(v) < 0 then
5:         Remove v from G_i;
6: Sort V(i) in descending order according to twn_i(v) for all
   v ∈ V(i);
7: for each v ∈ V(i) do
8:     TWG node table ← v, deg_{G̃_i}(v), twn_i(v);
9:     TWG edge table ← v, nbr_{G̃_i}(v);
10: return TWG;
```

The IMTA algorithm first loads G_i and G_{i-sg} into memory and calculates the structure weights of nodes separately (line 1). Then, according to Definitions 5 and 6, the algorithm gets the time weights of nodes in G_i and $\widetilde{G}_i$ in turn (lines 2–5). After getting $\widetilde{G}_i$, we need to store it to disk using the node and edge tables (lines 7–9). Each line of the TWG node table includes node $v \in V(i)$, $deg_{\widetilde{G}_i}(v)$, and $twn_i(v)$, and each line of the TWG edge table includes v and $nbr_{\widetilde{G}_i}(v)$. Besides,

the order of the nodes in both tables is sorted by time weights of nodes from the largest to the smallest (line 6).

Theorem 1. *The time and space complexities of the IMTA algorithm are* $O(3|V|+2|E(i)|+|E(i-sg)|)$ *and* $O(2|V|+|E_i|+|E_{i-sg}|)$ *respectively.*

Proof. In terms of the processing time, the IMTA algorithm needs to visit each node $v \in V$, for each v, it needs to visit $deg_{G_i}(v)$ or $deg_{G_{i-sg}}(v)$ edges, and the algorithm also needs to visit all the nodes in G_i to calculate the time weight. After getting $\widetilde{G}_i$, IMTA needs to visit the whole $\widetilde{G}_i$ to put it into the disk, and thus, the time complexity is $O(3|V|+2|E(i)|+|E(i-sg)|)$. For space, IMTA needs to load G_i and G_{i-sg}, and therefore, the space complexity is $O(2|V|+|E_i|+|E_{i-sg}|)$. □

4.2 Global Search Based EBCS Discovery Algorithm

To discover the EBCS in $\widetilde{G}_i$, we develop the global search based EBCS discovery algorithm, i.e., GS-EBCS. According to the definition of EBCS, it is known that k-core is the main structure of EBCS, and another main feature is the constraint on the minimum time weight and the number of nodes. Therefore, the GS-EBCS algorithm first prunes $\widetilde{G}_i$ into a maximum k-core based on the global search. Next, the nodes with the smallest time weight will be continuously removed, and finally the $\widetilde{G}_i$ with the k-core structure is the EBCS.

Optimization. On top of our conference paper [16], instead of the original method that updates the degrees of the remaining nodes every time after a node is deleted, we develop a batch-deletion strategy to delete multiple nodes in one batch. In this way, we only need to update the degrees of the remaining nodes after one batch-deletion. An important issue in batch-deletion is to determine the number of deleted nodes in each batch. Here, we develop two operations, i.e., the multiplication operation and the in-half operation. For the multiplication operation, we twice the number of deleted nodes in the next batch, while for the in-half operation, we remove half of the number of the deleted nodes of this batch in the next batch.

Specifically, in the first batch, the number of deleted nodes is 1. And we say the batch-deletion is successful if the updated subgraph is still a k-core. From the first batch, if the deletion succeeds in each batch, we will continue to conduct the multiplication operation until we meet the first failure. When one batch-deletion fails, which means the remaining nodes fail to form a k-core after the node deletion, we need to recover the subgraph by adding the deleted nodes back and invoke the in-half operation. It is worth noting that, in order to prevent repeated invalid deletions, once the in-half operation is invoked for the first time, we will always conduct the in-half operation for each of the subsequent batch-deletions. In the end, when there is no node for deletion, the EBCS is found. The correctness of the batch-deletion strategy is proved in Theorem 2.

Theorem 2. *Using the batch-deletion approach can successfully find the EBCS.*

Proof. The discovery of the EBCS using the batch-deletion strategy can be divided into two cases. Firstly, the deletion fails in the first batch, i.e., *del_num* is 1. At this time, the EBCS is the k-core before performing the node deletion operation. Secondly, the deletion fails in a batch greater than 1. Failure is bound to occur because the multiplication operation cannot be executed all the time. And the EBCS must be included in the subgraph before the execution of this failed batch.

We assume that the deletion fails in the $(p+1)$-th batch, where $p \geqslant 1$, and then the number of deleted nodes in the $(p+1)$-th batch is 2^p, which means that after the p-th batch-deletion, we are able to find the EBCS by deleting a certain number of nodes between 0 and 2^p-1. Because subsequent batch-deletions will only perform the in-half operation, the number of deleted nodes since the $(p+2)$-th batch is $2^{(p-1)}, 2^{(p-2)}, 2^{(p-3)}, \ldots, 2^0, 0$, respectively. If we select at least one unduplicated element from them to sum, the result will be in the range $[0, 2^p-1]$, which indicates that we can delete any number of nodes between 0 and 2^p-1 in total in subsequent batch-deletions after the $(p+1)$-th batch-deletion. That is, we must be able to find the EBCS contained in the updated subgraph after the p-th deletion. Therefore, combining the analysis of the two situations described above, using the batch-deletion approach can successfully find the EBCS. □

Specifically, GS-EBCS first determines whether there is the possibility of the EBCS in $\widetilde{G}_i$ based on the condition 1 of the EBCS (line 2). If the condition is satisfied, *GetCore*() in Algorithm 2 prunes $\widetilde{G}_i$ to a maximum k-core. This function first removes all nodes in $\widetilde{G}_i$ whose degrees are less than k (lines 5–7 of Algorithm 2), and then iterates this process until $\delta(\widetilde{G}_i) \geqslant k$ (lines 1–8 of Algorithm 2). After getting the maximum k-core, GS-EBCS calls *DeleteNode*() several times to remove nodes with the smallest time weight until no node can be deleted under the condition of the k-core structure (lines 7–28 of Algorithm 3).

Algorithm 2. GetCore and DeleteNode

```
1: Procedure GetCore(k, G̃_i)
2: update ← true;
3: while uptade do
4:    update← false;
5:    for each v in G̃_i do
6:       if deg_{G̃_i}(v) < k then
7:          Remove v from G̃_i;
8:          update ← true;
9: Procedure DeleteNode(k, G̃_i, del_num, v_max, v_min)
10: if del_num ⩾ |V(i)| − k then
11:    return G̃_i ← ∅;
12: for u ← v_min to v_max do
13:    Remove u from G̃_i; remove all nodes that have the same
       twn_i(u) from G̃_i;
14:    if the number of removed nodes ⩾ del_num then
15:       break
16: GetCore(k, G̃_i);
17: return G̃_i;
```

Since the k-core must have at least $k+1$ nodes, it is necessary to determine in *DeleteNode*() whether $|V(i)|$ is less than $k+1$ after deleting *del_num* nodes (line 10 of Algorithm 2). When deleting node u, the function removes all nodes whose time weight is $twn_i(u)$ (lines 13 and 14 of Algorithm 2), which is to ensure that the EBCS has the maximum number of nodes (condition 3 of the EBCS).

For the process of the batch-deletion (lines 6–28), *del_num* indicates the number of deleted nodes, and each call of *DeleteNode*() by the algorithm is an execution of one batch. Line 22 of Algorithm 3 executes the multiplication operation, and lines 15 and 28 execute the in-half operation, where $flag\_del$ is false indicating that subsequent batches will only execute the in-half operation. When no nodes can be deleted (lines 17–19) or the number of deleted nodes becomes 1 again (lines 23–25), the EBCS is found.

Example 2. For $\widetilde{G}_5$ (Fig. 2(f)), $V(5) = \{v_1, v_3, v_2, v_5, v_4\}$. The execution process of GS-EBCS is as follows.

Step 1: *GetCore*(2, $\widetilde{G}_5$) $\rightarrow S = \{v_1, v_3, v_2, v_4\}$.

Step 2: *DeleteNode*(2, $\widetilde{G}_5[S]$, 1, v_1, v_4) $\rightarrow S = \{v_1, v_3, v_2\}$ (when deleting v_4, there are no nodes with time weight equal to $twn_5(v_4)$, and thus, no nodes need to be removed together).

Step 3: *DeleteNode*(2, $\widetilde{G}_5[S]$, 2, v_1, v_2) $\rightarrow S = \emptyset$.

Step 4: *DeleteNode*(2, $\widetilde{G}_5[S]$, 1, v_1, v_2) $\rightarrow S = \emptyset$.

Step 5: return $S = \{v_1, v_3, v_2\}$.

Theorem 3. *The time complexity of GS-EBCS is* $O(\frac{|V(i)|^2+6|V(i)|+5|\widetilde{E}(i)|-k^2-k}{2})$ *and the space complexity is* $O(|V(i)| + |\widetilde{E}(i)|)$.

Proof. The GS-EBCS algorithm takes $O(|V(i)| + |\widetilde{E}(i)|)$ time to load $\widetilde{G}_i$. *GetCore*(), in the worst case, needs to delete one node each loop until $k+1$ nodes remain to form the k-core, and thus, the function takes at most $O(\frac{(k+1+|V(i)|)(|V(i)|-k)+2|\widetilde{E}(i)|}{2})$. For *DeleteNode*(), it takes at most $O(|V(i)|)$. Therefore, the time complexity is $O(\frac{|V(i)|^2+6|V(i)|+5|\widetilde{E}(i)|-k^2-k}{2})$. Besides, GS-EBCS needs $O(|V(i)| + |\widetilde{E}(i)|)$ to store $\widetilde{G}_i$ in memory. □

4.3 Local Search Based EBCS Discovery Algorithm

Because the GS-EBCS algorithm is based on global search, it has to delete more nodes to find the target subgraph, especially when k is small or the whole graph is dense. This makes the GS-EBCS algorithm inefficient. In this subsection, we focus on the local search for the EBCS and propose the local search based EBCS algorithm, named as LS-EBCS. This algorithm is more efficient when the value of k is small or the graph is dense.

The LS-EBCS algorithm utilizes the idea of expanding and refining. It expands the subgraph from the nodes with the largest time weight until the algorithm finds the k-core or all nodes are visited, and then refines the discovered subgraph to satisfy all constraints of the EBCS, whose time complexity is optimal. Of course, if all nodes are visited, then the LS-EBCS algorithm degenerates to GS-EBCS. In specific, LS-EBCS starts from the first node in $V(i)$ ($V(i)$ is ordered), and keeps adding nodes in sequence until the subgraph induced by these nodes contains a k-core or all nodes are visited. After that, the algorithm keeps deleting nodes with the smallest time weight until the EBCS is found, and the deletion operation is the same as that in lines 6–28 of Algorithm 3.

In the implementation of Algorithm 4, S is composed of the first $k+1$ nodes with a degree no less than k (line 2). To ensure that the EBCS has the maximum number of nodes, the algorithm adds the nodes with the same $twn_i(u)$ and the degree no less than k into S, where u is the $(k+1)$-th node in the current S (lines 3 and 4). If $\widetilde{G}_i[S]$ fails to form a k-core (lines 5 and 6), then the algorithm continues to add nodes to S in order. Besides whenever the size of $\widetilde{G}_i[S]$ doubles (the proof that the time complexity of the algorithm can be minimized when the graph size is increased by a factor of 2 was shown in [20]) or all nodes are visited, the algorithm stops adding nodes, and again repeats the

operations in lines 3–5 (lines 8 and 9). If the expanded $\widetilde{G}_i[S]$ still does not contain the k-core, the algorithm iterates the above expansion process until S contains the k-core or all nodes are visited (lines 7–11). After getting the k-core, LS-EBCS performs the node deletion operation of Algorithm 3 (line 12).

Algorithm 3. GS-EBCS(k, φ, $\widetilde{G}_i$)

1: $v_{\max} \leftarrow$ first node in $V(i)$;
2: **if** $twn_i(v_{\max}) < \varphi$ **then**
3: **return** None;
4: $flag\_del \leftarrow$ **true**;
5: $GetCore(k, \widetilde{G}_i)$
6: $\widetilde{W}_i \leftarrow \widetilde{G}_i$; /*back up $\widetilde{G}_i$.*/
7: $del\_num \leftarrow 1$;
8: **while true do**
9: $v_{\max} \leftarrow$ first node in $V(i)$;
10: $v_{\min} \leftarrow$ last node in $V(i)$;
11: $\widetilde{G}_i \leftarrow DeleteNode(k, \widetilde{G}_i, del\_num, v_{\max}, v_{\min})$
12: **if** $\widetilde{G}_i = \emptyset$ and $del\_num > 1$ **then**
13: $\widetilde{G}_i \leftarrow \widetilde{W}_i$;
14: $flag\_del \leftarrow$ **false**;
15: $del\_num \leftarrow del\_num/2$;
16: **continue**;
17: **else if** $\widetilde{G}_i = \emptyset$ and $del\_num = 1$ **then**
18: $\widetilde{G}_i \leftarrow \widetilde{W}_i$;
19: **break**;
20: **if** $flag\_del$ **then**
21: $\widetilde{W}_i \leftarrow \widetilde{G}_i$;
22: $del\_num \leftarrow 2 \times del\_num$;
23: **else if** $del\_num = 1$ **then**
24: $\widetilde{G}_i \leftarrow \widetilde{W}_i$;
25: **break**;
26: **else**
27: $\widetilde{W}_i \leftarrow \widetilde{G}_i$;
28: $del\_num \leftarrow del\_num/2$;
29: **return** $V(i)$;

Algorithm 4. LS-EBCS(k, φ, $\widetilde{G}_i$)

1: Lines 1–3 of Algorithm 3;
2: $S \leftarrow$ first $k+1$ nodes that $deg_{\widetilde{G}_i}(v) \geqslant k$ from $V(i)$; /* $V(i)$ is sorted in descending order. */
3: $u \leftarrow$ the last added node;
4: Add all nodes that have the same $twn_i(u)$ and $deg_{\widetilde{G}_i}(u) \geqslant k$ to S from $V(i)$;
5: $GetCore(k, \widetilde{G}_i[S])$;
6: **if** $\widetilde{G}_i[S] = \emptyset$ **then**
7: **while true do**
8: Continue to add node $v \in V(i)$ that $deg_{\widetilde{G}_i}(v) \geqslant k$ to S **until** $size(\widetilde{G}_i[S])$ multiplication **or** no addable node;
9: Reapeat lines 3–5;
10: **if** $\widetilde{G}_i[S] \neq \emptyset$ or no addable node **then**
11: **break**;
12: Lines 6–28 of Algorithm 3; /* The processing object of LS-EBCS is $\widetilde{G}_i[S]$ and S. */

Example 3. Continuing with example 2, the steps of obtaining $\widetilde{G}_5[S]$ are as follows:

$V(5) = \{v_1, v_3, v_2, v_5, v_4\}$;

step 1: $S = \{v_1, v_3, v_2\}$ (no nodes with time weight equal to $twn_5(v_2)$ and $size(\widetilde{G}_5[S]) = 6$);

step 2: $GetCore(2, \widetilde{G}_5[S]) \rightarrow S = \{v_1, v_3, v_2\}$ (no need to expansion);

step 3: $DeleteNode(2, \widetilde{G}_5[S], 1, v_1, v_2) \rightarrow S = \emptyset$;

step 4: return $S = \{v_1, v_3, v_2\}$.

The biggest difference between LS-EBCS and GS-EBCS is the way of finding the k-core.

Theorem 4. *The time complexity of the LS-EBCS algorithm is* $O(\frac{l^2k^2+3(l^2k+lk^2)+2l^2-3kl-10l-4+7(|V(i)|+|\widetilde{E}(i)|)}{4})$. *And the space complexity is* $O(|V(i)| + |\widetilde{E}(i)|)$.

Proof. Let l denote the number of expansions and S' denote the set of nodes after one expansion. In the worst case, each expansion requires adding twice as many nodes of itself, and since these nodes still do not form a k-core, these nodes have at most $(k \times |S'|/2) - 1$ edges. Each time, $GetCore()$ visits all nodes and edges, and thus, it takes at most $O(\frac{l(k+1)(l+1)(k+2)-4l}{4})$ to execute $GetCore()$ l times. $DeleteNode()$ takes at most $O(l(k+1))$ time. And the algorithm takes $O(|V(i)| + |\widetilde{E}(i)|)$ time to load $\widetilde{G}_i$. Therefore, the time complexity is $O(\frac{l^2k^2+3(l^2k+lk^2)+2l^2-3kl-10l-4+8(|V(i)|+|\widetilde{E}(i)|)}{4})$. The space complexity is the same as that of the GS-EBCS. □

5 I/O Efficient EBCS Discovery Algorithms

The space complexity of both GS-EBCS and LS-EBCS is $O(|V(i)| + |\widetilde{E}(i)|)$. Both nodes and edges are loaded in memory. However, when dealing with massive graphs, the reality is that the memory space is limited, and the number of edges of the graph is generally much larger than the number of nodes. Therefore, in this section, to solve the problem that massive graphs cannot be completely put into memory, we propose the I/O algorithms based on the semi-external model. For all I/O algorithms, we set τ to denote the average time required to load or write $nbr_{\widetilde{G}_i}(v)$ ($nbr_{\widetilde{G}_{i-sg}}(v)$) and B to denote the disk block size.

The semi-external model in this paper allows only node information to be resident in memory, and only the connected edges of one node will be loaded at a time. This approach is very effective in reducing the memory usage of massive graphs with a space complexity of $O(|V|)$. In the following, we first introduce the TWG transformation algorithm using I/O in Subsection 5.1, and the I/O-GS algorithm based on the global idea is introduced in Subsection 5.2. In order to further reduce the I/O cost and memory occupation of I/O-GS

and improve the efficiency of discovering the EBCS in massive temporal networks, we propose the I/O-LS algorithm based on the idea of expanding and refining in Subsection 5.3.

5.1 I/O TWG Transformation Algorithm

For the transformation of the TWG using the semi-external model, we only load V from disk and only load the relevant information when the neighbor nodes of node $u \in V$ need to be loaded. Therefore, we develop the TWG transformation algorithm based on the semi-external model (SETA) in Algorithm 5.

Algorithm 5. SETA(sg, G_i on disk, G_{i-sg} on disk)

1: $V \leftarrow$ load node table of G_i from disk; /* All snapshot graphs have the same V. */
2: **for** each $v \in V$ **do**
3: Load $nbr_{\widetilde{G}_i}(v)$ and associated edge weights from disk;
4: Load $nbr_{\widetilde{G}_{i-sg}}(v)$ and associated edge weights from disk;
5: Calculate $sum_i(v)$ and $sum_{i-sg}(v)$;
6: Lines 2–6 of Algorithm 1;
7: **for** each $v \in V(i)$ **do**
8: Load $nbr_{\widetilde{G}_i}(v)$ from disk;
9: $nbr_{\widetilde{G}_i}(v) \leftarrow$ nodes of $nbr_{\widetilde{G}_i}(v) \in V(i)$;
10: TWG node table $\leftarrow v, |nbr_{\widetilde{G}_i}(v)|, twn_i(v)$;
11: TWG edge table $\leftarrow v, nbr_{\widetilde{G}_i}(v)$;
12: **return** TWG;

Since all snapshot graphs have the same V, although the transformation of TWG involves two snapshot graphs, memory only needs space for one $|V|$ (line 1). Specifically, the algorithm visits each $v \in V$ and loads $nbr_{G_i}(v)$ and $nbr_{G_{i-sg}}(v)$ from the disk along with the corresponding edge weight information (lines 2–4), and next, computes the structure weight (line 5). After that, the algorithm obtains $\widetilde{G}_i$ (line 6). When writing $\widetilde{G}_i$ to the disk, SETA also loads a node's neighbor nodes from the edge table of G_i when needed (line 8), and then, removes those nodes that are not in $\widetilde{G}_i$ (line 9). Similarly, the order of the nodes in both tables is sorted by the time weight of nodes from the largest to the smallest.

Theorem 5. *The time, I/O, and space complexities of the SETA algorithm are* $O(|V|(4\tau+3))$, $O(\frac{2|V|+4\max\{|E(i)|,|E(i-sg)|\}}{B})$ *and* $O(|V|)$ *respectively.*

Proof. SETA requires loading V and two snapshot graphs, which takes $O(|V|+2|V|\tau)$. Obtaining $\widetilde{G}_i$ takes $O(|V|)$ and writing $\widetilde{G}_i$ to disk takes at most $O(|V|+2|V|\tau)$. Therefore, the time complexity is $O(|V|(4\tau+3))$. For I/O complexity, the algorithm requires $O(\frac{(|V|+|E(i)|+|E(i-sg)|)}{B})$ when loading snapshot graphs. And it also needs to load neighbor nodes from disk first each time when it writes to disk, which takes $O(\frac{|V(i)|+2|\widetilde{E}(i)|}{B})$, and thus, the I/O complexity is no more than $O(\frac{2|V|+4\max\{|E(i)|,|E(i-sg)|\}}{B})$. Because the semi-external model only allows V in memory, the space complexity is $O(|V|)$. □

5.2 I/O Global Search Based Discovery Algorithm

In this subsection, we develop the I/O-GS algorithm. I/O-GS is also based on the idea of the global search for the maximum k-core and the batch-deletion. But there is only node information in memory, including node v, $deg_{\widetilde{G}_i(v)}$ and $twn_i(v)$.

The I/O-GS algorithm (Algorithm 6) contains three functions (lines 10–36). *UpdateDegree*() is to update the degrees of the nodes, and *GetCore*$^+$() has the same role as *GetCore*() to discover the k-core, and *DeleteNode*$^+$() is used to delete nodes with the smallest time weight.

Specifically, the I/O-GS algorithm loads all the node information from the disk into memory (line 1). Similarly, after determining whether $\widetilde{G}_i$ satisfies condition 1 of the EBCS (line 4), the algorithm calls *GetCore*$^+$() to find the maximum k-core.

In *GetCore*$^+$(), after each round of removing nodes with a degree less than k, the algorithm updates the remaining nodes in $\widetilde{G}_i$ with *UpdateDegree*() (lines 17–24). This function visits each deleted node v and then loads $nbr_{\widetilde{G}_i}(v)$ from disk (line 12). For each $u \in nbr_{\widetilde{G}_i}(v)$ and $u \in V(i)$, $deg_{\widetilde{G}_i}(u)$ is reduced by 1 due to the loss of a neighbor node v (lines 13–15). After getting the maximum k-core, the I/O-GS algorithm needs to call *DeleteNode*$^+$() to delete nodes with the smallest time weight and finds the EBCS. And the deletion is described in Algorithm 2 and Algorithm 3.

Theorem 6. *The time complexity of I/O-GS is* $O(\frac{|V(i)|^2+(5+2\tau)|V(i)|+2|\widetilde{E}(i)|-k^2-k-2\tau k}{2})$, *and the I/O and space complexities are* $O(\frac{|V(i)|+|\widetilde{E}(i)|}{B})$ *and* $O|V(i)|$ *respectively.*

Proof. The I/O-GS algorithm takes $O(|V(i)|)$ to load the node table of $\widetilde{G}_i$. Compared with the GS-EBCS algorithm, I/O-GS needs more time to load edges from disk in finding k-core and deleting nodes, and thus, the time complexity of I/O-GS does not exceed $O(\frac{|V(i)|^2+(5+2\tau)|V(i)|+2|\widetilde{E}(i)|-k^2-k-2\tau k}{2})$. Because the algorithm loads only $V(i)$ and needs to load $nbr_{\widetilde{G}_i}(u)$ from the disk for each removed node u, the I/O complexity is $O(\frac{|V(i)|+|\widetilde{E}(i)|}{B})$. Besides, it is clear that the space complexity of the I/O-GS is $O|V(i)|$. □

Algorithm 6. I/O-GS(k, φ, $\widetilde{G}_i$ on disk)

1: Load all nodes' information from disk; /* $V(i)$ is sorted in descending order. */
2: $twn_i(v)$, $deg_{\widetilde{G}_i}(v) \leftarrow$ for all $v \in V(i)$;
3: $v_{\max}$, $v_{\min} \leftarrow$ first node, last node in $V(i)$;
4: **if** $twn_i(v_{\max}) < \varphi$ **then**
5: **return** None;
6: $flag\_del \leftarrow$ **true**;
7: $GetCore^{+}(k, V(i), deg_{\widetilde{G}_i})$;
8: Lines 6–28 of Algorithm 3. /* The difference in this algorithm is that I/O-GS backs up the nodes. *DeleteNode*() is changed to $DeleteNode^{+}()$. */
9: **return** $V(i)$;
10: **Procedure** $UpdateDegree(k, S, V(i), deg_{\widetilde{G}_i})$
11: **for** each $v \in S$ **do**
12: Load $nbr_{\widetilde{G}_i}(v)$ from disk;
13: **for** each $u \in nbr_{\widetilde{G}_i}(v)$ **do**
14: **if** $u \in V(i)$ **then**
15: $deg_{\widetilde{G}_i}(u) \leftarrow deg_{\widetilde{G}_i}(u) - 1$;
16: **Procedure** $GetCore^{+}(k, V(i), deg_{\widetilde{G}_i})$
17: $update \leftarrow$ **true**;
18: **while** *uptade* **do**
19: $update \leftarrow$ **false**; $S \leftarrow \emptyset$;
20: **for** each $v \in V(i)$ **do**
21: **if** $deg_{\widetilde{G}_i}(v) < k$ **then**
22: Remove v from $V(i)$;
23: $update \leftarrow$ **true**; $S \leftarrow S \cup \{v\}$
24: $UpdateDegree(k, S, V(i), deg_{\widetilde{G}_i})$
25: **Procedure** $DeleteNode^{+}(k, V(i), deg_{\widetilde{G}_i}, twn_i, del\_num, v_{\max}, v_{\min})$
26: $S \leftarrow \emptyset$;
27: **if** $del\_num \geqslant |V(i)| - k$ **then**
28: **return** $V(i) \leftarrow \emptyset$;
29: **for** $u \leftarrow v_{\min}$ to $v_{\max}$ **do**
30: Remove u from $V(i)$; remove all nodes that have the same $twn_i(u)$ from $V(i)$;
31: $S \leftarrow$ deleted nodes;
32: **if** $|S| \geqslant del\_num$ **then**
33: **break**;
34: $UpdateDegree(k, S, V(i), deg_{\widetilde{G}_i})$;
35: $GetCore^{+}(k, V(i), deg_{\widetilde{G}_i})$;
36: **return** $V(i)$;

5.3 I/O Local Search Based Discovery Algorithm

Although I/O-GS can handle massive temporal networks, it still needs to load all node information into memory, and most of the nodes also need to load edges, which leads to high I/O cost. In fact, the definition of the EBCS indicates that when k is less than the maximum k of the graph and the graph is dense, the EBCS is among the top-ranked nodes of $V(i)$, and thus, it is unnecessary to load all the nodes for the I/O method. Therefore, in order to reduce the I/O cost and memory occupation of I/O-GS and improve the efficiency of discovering the EBCS, we propose the I/O-LS algorithm.

The I/O-LS algorithm follows the idea of expanding and refining, but the difference is that I/O-LS can load directly into memory starting from node v with the maximum time weight until T containing the k-core is found. This makes I/O-LS a great reduction in time, memory and I/O consumption.

Specifically, Algorithm 7 first loads the node information of the first row of the node table from disk, which includes the largest time weight (line 1). If the time weight of this node is less than φ (not satisfying condition 1 of the EBCS), the algorithm exits (lines 2 and 3). Instead, the algorithm gradually loads the nodes from disk and starts to expand.

Algorithm 7. I/O-LS(k, φ, $\widetilde{G}_i$ on disk)

1: $v \leftarrow$ load the first node from disk;
2: **if** $twn_i(v) < \varphi$ **then**
3: **return** None;
4: $T \leftarrow$ load the first $k+1$ nodes with a degree no less than k from disk;
5: $u \leftarrow$ the last loaded node;
6: Load all nodes that have the same $twn_i(u)$ and $deg_{\widetilde{G}_i}(u) \geqslant k$ to T from disk;
7: $GetCore^{+}(k, T, deg_{\widetilde{G}_i})$;
8: **if** $T = \emptyset$ **then**
9: **while true do**
10: Continue to load node v that $deg_{\widetilde{G}_i}(v) \geqslant k$ to T from disk **until** $size(\widetilde{G}_i[T])$ multiplication **or** no addable node;
11: $u \leftarrow$ the last loaded node;
12: Load all nodes to T from disk that have the same $twn_i(u)$;
13: $GetCore^{+}(k, T, deg_{\widetilde{G}_i})$;
14: **if** $T \neq \emptyset$ or no addable node **then**
15: **break**;
16: $flag\_del \leftarrow$ **true**;
17: Lines 8 and 9 of Algorithm 6;

Because the I/O-LS algorithm is consistent with LS-EBCS in terms of the execution method, the analysis of the specific expansion is shown in Algorithm 4, and the analysis of the batch-deletion is shown in Algorithm 6.

Theorem 7. *The time complexity of I/O-LS is* $O(\frac{l^2k^2+3(l^2k+lk^2)+2l^2(\tau k+\tau+1)+6l\tau(k+1)-3l(k+10)-4}{4})$, *and the I/O and space complexities of the I/O-LS algorithm are* $O(\frac{l^2k^2+3lk^2+l^2k+7lk-4}{4B})$ *and* $O(|lk+l|)$ *respectively.*

Proof. In the first $(l-1)$ expansions, $GetCore^{+}()$ deletes all nodes, and then loads the neighborhood nodes of each deleted node from disk and visits their connected edges by *UpdateDegree*(). After the l-th expansion, the algorithm keeps the EBCS containing at least $k+1$ nodes, and thus, the time complexity is $O(\frac{l^2k^2+3(l^2k+lk^2)+2l^2(\tau k+\tau+1)+6l\tau(k+1)-3lk-10l-4}{4})$.

For I/O complexity, the algorithm loads a node or deletes a node by loading its neighborhood nodes from

disk. In the first $(l-1)$ times, $GetCore^{+}()$ deletes all nodes of the expansion, and in the l-th, the algorithm deletes at most $l(k+1)$ nodes, and thus, the total I/O complexity is $O(\frac{l^2k^2+3lk^2+l^2k+7lk-4}{4B})$. The algorithm loads at most $l(k+1)$ nodes, and thus, it requires $O(lk+l)$ space. □

6 Performance Studies

In this section, we conduct extensive experiments and evaluate the efficiency and effectiveness of our proposed algorithms, i.e., IMTA, SETA, GS-EBCS, LS-EBCS, I/O-GS and I/O-LS. All algorithms are implemented in Python 3.8 under JetBrains PyCharm 2021. All the experiments are conducted on a computer with Windows 10 Professional operating system, Intel® Core™ i5-7200 CPU with 2.5 GHz, 12 G RAM, and 1 T hard disk.

We evaluate the in-memory algorithms and the I/O efficient algorithms with different settings. All experiments of the in-memory algorithms are performed in the above environment. In order to evaluate the I/O algorithms, we manually set a small memory capacity for the tests in the above environment, so that the tested graphs cannot be completely loaded into memory.

6.1 Experimental Setup

Datasets. We use four real datasets. The statistical information is shown in Table 1.

Table 1. Dataset Statistics

Dataset	$\|V\|$	$\|E\|$	SG	TS	MC
LKML①	27 937	1 096 440	98	Month	5
MCB②	76 556	856 956	139	Day	7
DBLP③	1 727 220	14 719 227	46	Year	250
YTB④	3 223 589	12 223 774	202	Day	500

Note: TS denotes the unit of the timestamp, SG denotes the number of snapshot graphs contained in a temporal network, and MC denotes the artificially set upper limit of the memory capacity (MB) for datasets in I/O experiments.

LKML① is a temporal communication network of the Linux kernel mailing list: each edge in the snapshot graph indicates that users v and u have contacted each other via email during this month, and the edge weight is the number of exchanged emails. The largest snapshot graph has 4 449 edges and the maximum k value for the k-core is 15.

Microblog② (MCB for short) is a temporal semantic network: each edge in the snapshot graph represents that words v and u, the nodes that the edge connects, occur together in a sentence, and the edge weight is the percentage of occurrence. The largest snapshot graph has 11 980 edges and the maximum k value for the k-core is 15.

DBLP③ is an author collaboration temporal network: each edge in the snapshot graph denotes that authors v and u, the nodes that the edge connects, have collaborated relation during this year, and the edge weight is the number of collaborations. The largest snapshot graph has 1 222 656 edges and the maximum k value for the k-core is 98.

YouTube④ (YTB for short) is a friendship network of users on YouTube: each edge in the snapshot graph denotes that users v and u, the nodes that the edge connects, interacted on this day. The largest snapshot graph has 1 368 340 edges and the maximum k value for the k-core is 12.

Parameters. In all experiments, the specific parameters include region value sg, core number k and threshold φ. sg denotes the value of the sliding-windows, which affects the number of TWGs in the dataset. In order to focus time on bursting information worthy of attention, we filter some unimportant TWGs by an artificially given threshold φ. The total number of TWGs and the number of filtered TWGs for different datasets under these two parameters are given in Table 2.

Table 2. Number of Filtered TWGs and TWGs

Dataset	sg	φ	Number of Filtered TWGs/ Number of All TWGs
LKML①	2	25 290.56	31/96
	3	14 324.57	23/95
MCB②	4	8.56×10^{-5}	50/135
	7	4.36×10^{-5}	34/132
DBLP③	2	294.15	13/44
	3	506.07	14/43
YTB④	4	33 384.87	56/198
	7	16 292.91	38/195

Memory Capacity. Under the assumption of the semi-external model, all I/O algorithms are performed

①http://konect.uni-koblenz.de, Jul. 2020.

②https://www.weibo.com, May 2020.

③http://dblp.uni-trier.de/xml/, Jul. 2020.

④http://snap.stanford.edu/data/index.html, Jul. 2020.

under the condition that the memory capacity is capped, which ensures that the node information of the tested graph can be fully loaded into memory, but not all edges. The MC in Table 1 gives specific information on the memory capacity set for each dataset.

6.2 Efficiency Testing

Exp-1: *Efficiency of IMTA and SETA.* We compare the two TWG transformation algorithms by the running time and the memory usage. All experiments were performed on the respective largest TWG of each dataset with maximum sg and $\varphi = 0$, where sg is shown in Table 2. The experimental results in Fig.3(a) show that the running time of the IMTA algorithm is lower than that of the SETA algorithm, because SETA involves many I/O read/write operations. However, on the memory usage shown in Fig.3(b), the consumption of the IMTA algorithm is too high when processing the massive snapshot graph. In contrast, the memory usage of the SETA algorithm is essentially half that of the IMTA algorithm under the same parameters, which demonstrates the effectiveness of designing algorithms using the semi-external model. In general, SETA is still more suitable for dealing with massive temporal networks.

Exp-2: *Efficiency of GS-EBCS and LS-EBCS.* In this experiment, we elaborate the efficiency of the two in-memory algorithms on the four datasets with different parameters. Table 3 presents the running time of the two algorithms. Since each dataset forms a varying number of TWGs at a given sg, the running time is the total time required by the algorithm to process all TWGs, where φ is 0 to indicate that no TWG is filtered. From the experimental results in Table 3, we can see that the LS-EBCS algorithm outperforms the GS-EBCS algorithm when k is far from the maximum k of the tested graph, especially in MCB. This is the result obtained by the idea of the LS-EBCS algorithm and the denseness of the TWG. When k is smaller and the TWG is denser, the LS-EBCS algorithm needs to expand only a few times to obtain the set of nodes containing the EBCS. In addition, we can see that the GS-EBCS algorithm is faster than the LS-EBCS algorithm when the value of k is very close to the maximum k of the TWG

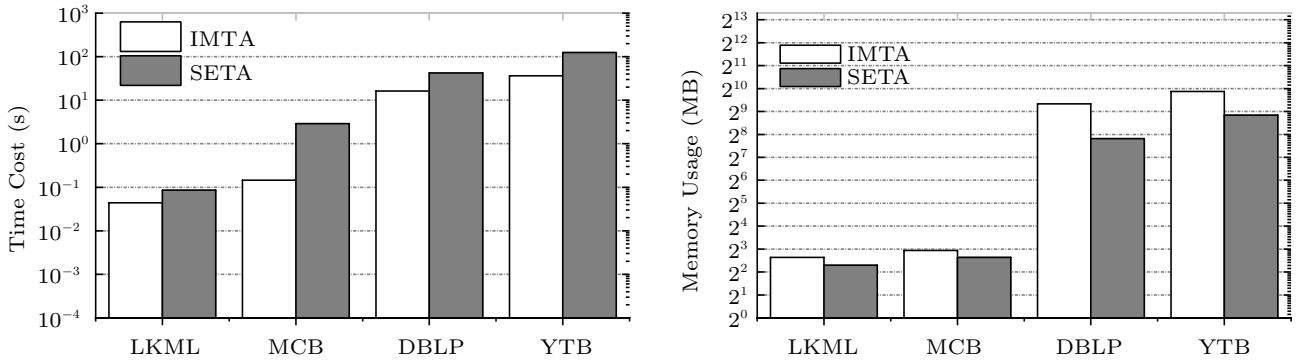


Fig.3. Efficiency results of IMTA and SETA. (a) Running time. (b) Memory usage.

Table 3. Running Time (s) of In-Memory Algorithms with Varying Parameters for Different Datasets

Dataset	sg	φ	$k=3$		$k=6$		$k=9$		$k=12$		$k=29$		$k=52$		$k=75$	
			LS	GS	LS	GS	LS	GS	LS	GS	LS	GS	LS	GS	LS	GS
LKML	2	0	**0.631**	2.643	**1.192**	1.511	1.530	**1.216**	1.342	**0.989**	-	-	-	-	-	-
	3	0	**0.769**	2.678	**1.184**	1.610	1.546	**1.229**	1.432	**1.001**	-	-	-	-	-	-
MCB	4	0	**1.127**	11.605	**1.754**	5.063	**2.903**	3.245	3.134	**2.627**	-	-	-	-	-	-
	7	0	**1.225**	16.345	**1.791**	5.399	**2.971**	3.458	3.293	**2.842**	-	-	-	-	-	-
YTB	4	0	**55.098**	100.742	**69.392**	69.806	73.174	**67.049**	70.974	**64.053**	-	-	-	-	-	-
	7	0	**65.279**	101.718	**75.144**	101.132	78.570	**73.502**	82.715	**69.311**	-	-	-	-	-	-
DBLP	2	0	-	-	**16.213**	107.533	-	-	-	-	**16.017**	28.892	17.936	**15.871**	19.080	**9.231**
	3	0	-	-	**16.851**	108.989	-	-	-	-	**18.536**	29.435	19.449	**15.120**	21.020	**11.066**

Note: The running time marked in bold indicates that the algorithm runs in a shorter time under the current parameters. Besides, LS and GS denote LS-EBCS and GS-EBCS, respectively.

or the TWG is sparse, i.e., the result on the LKML and YTB. This is because, in the above cases, the LS-EBCS algorithm has to increase the subgraph size by expanding frequently. In the worst case, the subgraph contains all nodes of the TWG, and then the LS-EBCS algorithm degenerates into the GS-EBCS algorithm. Thus, the whole process of LS-EBCS takes more time than that of the GS-EBCS algorithm. For YTB, the maximum k-core of the largest snapshot graph is 12, the maximum core of most snapshot graphs does not exceed 6, and thus, the whole network is very sparse. This leads to the result of the LS-EBCS algorithm for YTB in Table 3.

To show more clearly the comparison between two in-memory algorithms, we conduct experiments on the respective largest TWG for each dataset under Exp-1 conditions. As shown in Figs.4(a)–4(d), we can clearly see the variation of running time over two algorithms as the value of k. It is worth noting that the running time of the LS-EBCS algorithm does not increase linearly with k, for example, Fig.4(c).

This is because the running time of LS-EBCS is affected by the number of extensions. The fewer the extensions are needed, the fewer the nodes are added, and then the shorter the running time of the algorithm has. Also the number of expansions is affected by the initial number of nodes, for example, when k equals 6, the initial number of nodes is 7.

From Fig.4(a), we can see the effect of k close to the maximum k on the LS-EBCS algorithm. By comparing Figs.4(a)–4(d), we can also find the effect of the sparsity of the graph on LS-EBCS.

In general, the LS-EBCS algorithm is more efficient than the GS-EBCS algorithm in most cases. Only in extreme cases, for example, when k is very close to the maximum k of the graph or the graph is very sparse, the GS-EBCS algorithm outperforms LS-EBCS.

Exp-3: *Efficiency of I/O Algorithms.* We test the efficiency of the two I/O EBCS discovery algorithms in detail based on the parameters in Exp-2, mainly comparing the running time and I/O cost of the algorithms, where I/O cost refers to the number of load (write) operations executed by the algorithm. The specific experimental information is shown in Table 4 and Fig.5.

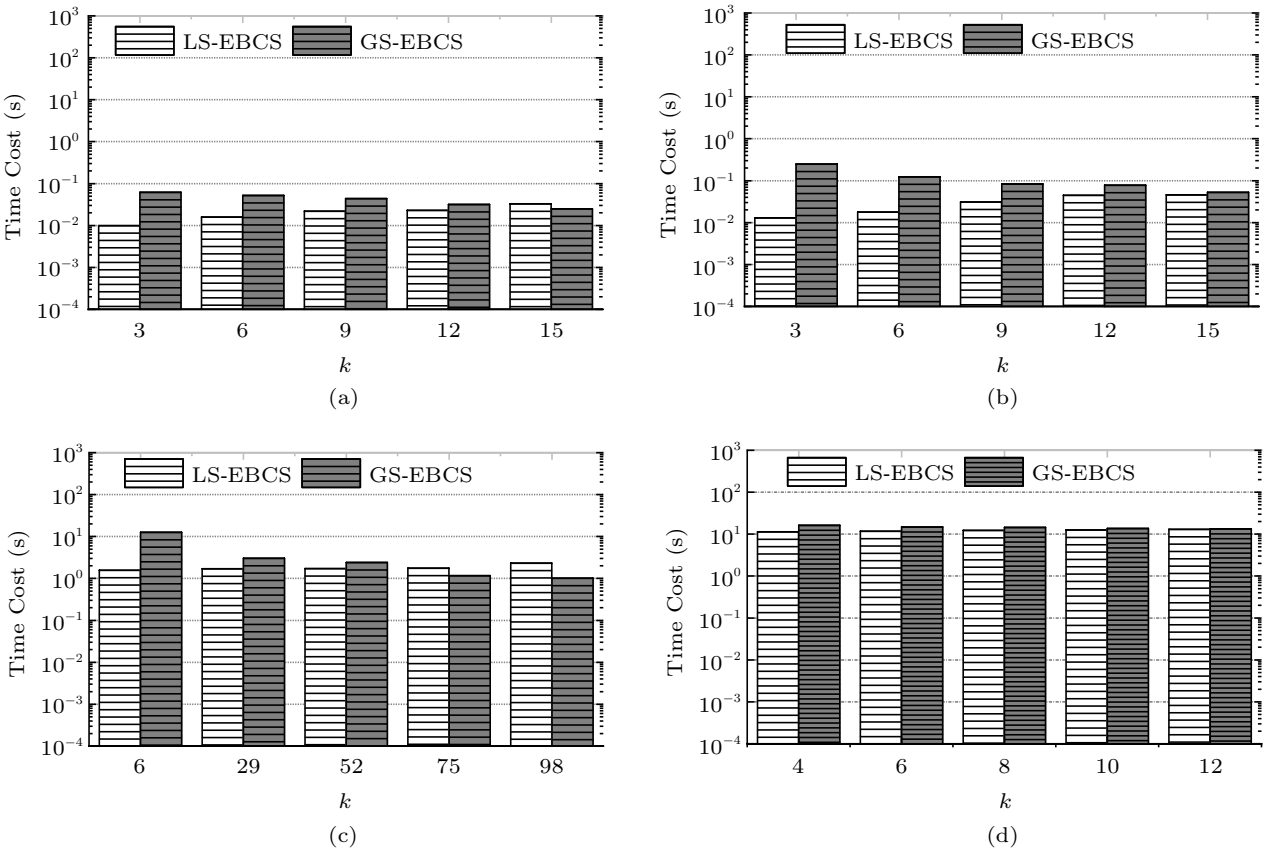


Fig.4. Efficiency results of two in-memory algorithms on the respective largest TWG for each dataset. (a) LKML. (b) MCB. (c) DBLP. (d) YTB.

Table 4. Running Time (s) of I/O Algorithms with Varying Parameters for Different Datasets

Dataset	sg	φ	$k=3$		$k=6$		$k=9$		$k=12$		$k=29$		$k=52$		$k=75$	
			I/O-LS	I/O-GS	I/O-LS	I/O-GS	I/O-LS	I/O-GS	I/O-LS	I/O-GS	I/O-LS	I/O-GS	I/O-LS	I/O-GS	I/O-LS	I/O-GS
LKML	2	0	**0.627**	5.098	3.426	**2.778**	6.605	**1.500**	7.001	**1.071**	-	-	-	-	-	-
	3	0	**0.745**	5.814	3.537	**2.899**	6.755	**1.649**	7.022	**1.891**	-	-	-	-	-	-
MCB	4	0	**0.511**	54.591	**2.388**	13.660	6.688	**4.710**	10.543	**2.540**	-	-	-	-	-	-
	7	0	**0.646**	55.801	**2.233**	15.827	7.754	**5.316**	10.909	**3.107**	-	-	-	-	-	-
YTB	4	0	**49.034**	147.029	240.981	**75.658**	327.860	**67.290**	363.931	**66.763**	-	-	-	-	-	-
	7	0	**53.097**	165.126	241.247	**76.360**	360.003	**70.942**	368.074	**70.322**	-	-	-	-	-	-
DBLP	2	0	-	-	**0.685**	195.527	-	-	-	-	**16.233**	35.546	45.328	**17.774**	89.071	**12.106**
	3	0	-	-	**1.024**	207.630	-	-	-	-	**19.140**	38.360	51.466	**18.518**	109.847	**13.398**

From the results, we can find that I/O algorithms generally consume more time than in-memory algorithms, especially I/O-GS. This is due to the fact that the I/O algorithms require constant loading from and writing to disk. Moreover, the results of Fig.5 clearly demonstrate the advantages of the I/O-LS algorithm over the I/O-GS algorithm in terms of the running time and the I/O cost when k is very small or the graph is dense. Because the main idea of the I/O algorithms is the same as that of the in-memory algorithms, the specific data analysis is not repeated.

To further demonstrate the advantages of I/O algorithms in terms of memory usage, we record the maximum memory usage of the in-memory EBCS discovery algorithms and the I/O EBCS discovery algorithms under Exp-1 conditions respectively. We also give the I/O cost of the SETA algorithm to to build the tested TWGs of Fig.5. And the experimental results are shown in Fig.6. From Fig.6(a), we can further see the advantages of the I/O algorithms, especially I/O-LS.

Discussion. There is an interesting phenomenon in the results of Table 3 and Table 4. That is, when k is small, the I/O-LS algorithm runs faster than the in-memory LS-EBCS algorithm. The reasons are two folds. Firstly, the I/O-LS algorithm loads only a very small number of nodes into memory in this case, and thus, the algorithm is processed efficiently. Secondly, LS-EBCS still needs to load all the nodes and edges into memory.

6.3 Effectiveness Testing

Exp-4: *Effectiveness of the EBCS.* We use the graph diameter metric [33] to test the effectiveness of the EBCS found by our proposed algorithms. The graph diameter is defined as the maximum value of the shortest path between any two nodes in the graph. From this definition, we can see that the smaller the diameter of the graph is, the denser the graph is.

Fig.7 gives the diameters of the found EBCS, and we can see that most of the results for the diameter are small, which indicates that the EBCS is cohesive and our algorithms are effective.

Exp-5: *Case Study on MCB.* Microblog[5] is a social platform for sharing real-time information. For MCB, this paper collects information posted by 128 trustworthy media agencies from December 1, 2019 to April 17, 2020. To illustrate the real-world application of the EBCS, we set $k=5$, $sg=7$ and $\varphi=0$, and then we find a real bursting event on the MCB that occurred on December 31, 2019. Fig.8 gives the EBCS model of this event in the temporal network, and from "image" we can see that the pneumonia of the unknown cause has appeared in Wuhan city. Our model represents this event in a timely and efficient manner, which indicates that it is significant to find the EBCS in temporal networks.

7 Conclusions

In this work, we studied the problem of identifying EBCS from temporal networks. Extensive experiments showed that the I/O-GS and I/O-LS algorithms had comparable runtime with that of GS-EBCS and LS-EBCS, respectively, while the maximum memory usage is much smaller. Further, we found that the global search based methods, GS-EBCS and I/O-GS are more suitable for the situation when the temporal networks are sparse and the value of k is large, while the local search based methods, LS-EBCS and I/O-LS, have better performance when the temporal networks are dense and the value of k is small. In the future, we plan to implement the discovery of the EBCS in the distributed frameworks such as Hadoop and Spark.

[5] https://www.weibo.com, Jul. 2021

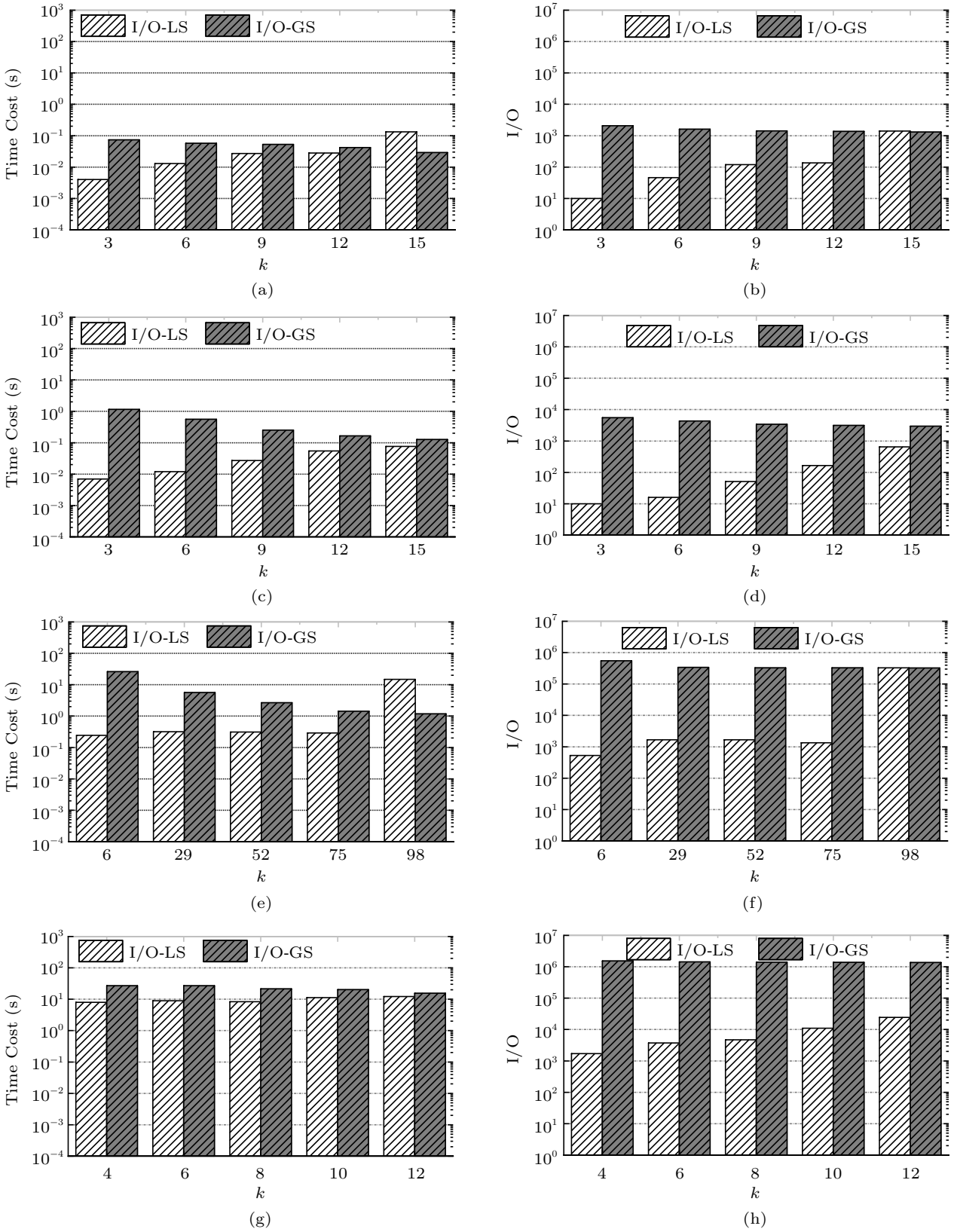


Fig.5. Efficiency results of I/O algorithms. (a), (c), (e), (g) Running time on LKML, MCB, DBLP and YTB respectively. (b), (d), (f), (h) I/O cost on LKML, MCB, DBLP and YTB respectively.

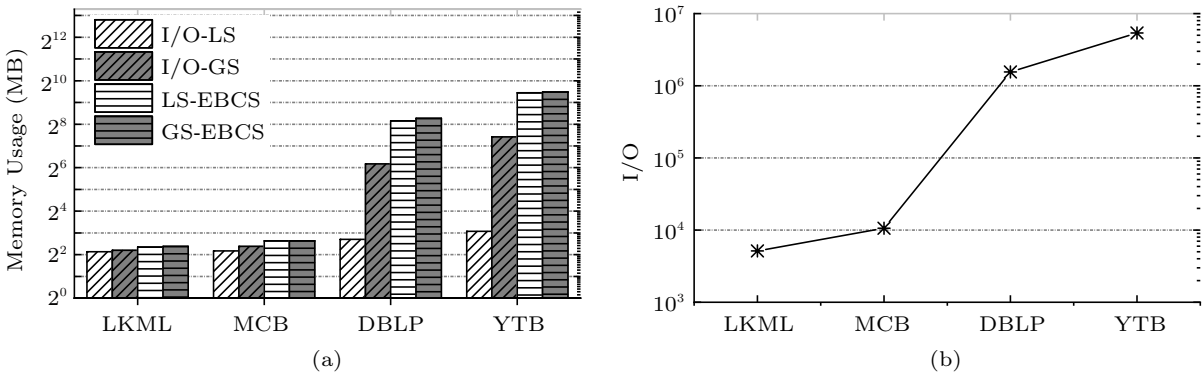


Fig.6. Maximum memory usage of four algorithms and the I/O cost of SETA. (a) Maximum memory usage. (b) I/O cost.

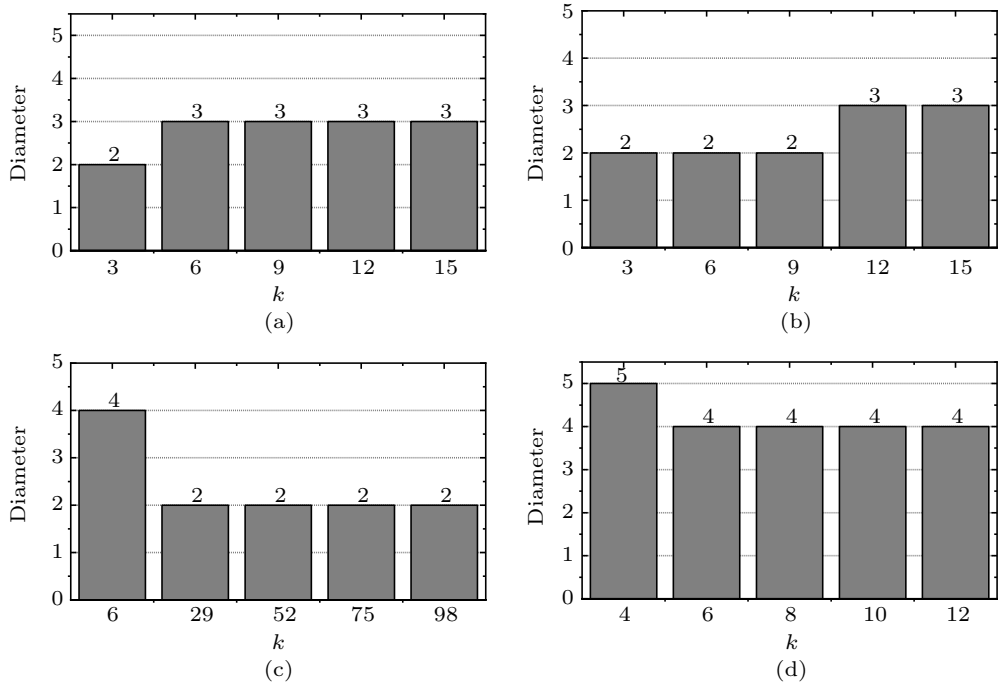


Fig.7. Graph diameters of the EBCS found by the algorithms on the respective largest TWG for each dataset. (a) LKML. (b) MCB. (c) DBLP. (d) YTB.

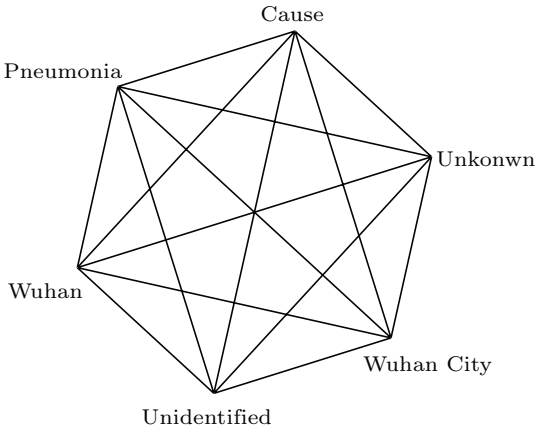


Fig.8. A real EBCS case in MCB.

References

[1] Holme P, Saramäki J. Temporal networks. *Physics Reports*, 2012, 519(3): 97-125. DOI: 10.1016/j.physrep.2012.03.001.

[2] Li R H, Su J, Qin L, Yu J X, Dai Q. Persistent community search in temporal networks. In *Proc. the 34th IEEE International Conference on Data Engineering*, Apr. 2018, pp.797-808. DOI: 10.1109/ICDE.2018.00077.

[3] Semertzidis K, Pitoura E, Terzi E, Tsaparas P. Finding lasting dense subgraphs. *Data Min. Knowl. Discov.*, 2019, 33(5): 1417-1445. DOI: 10.1007/s10618-018-0602-x.

[4] Qin H, Li R H, Wang G, Huang X, Yuan Y, Yu J X. Mining stable communities in temporal networks by density-based clustering. *IEEE Trans. Big Data*, 2022, 8(3): 671-684. DOI: 10.1109/TBDATA.2020.2974849.

[5] Lin L, Yuan P, Li R, Jin H. Mining diversified top-r lasting cohesive subgraphs on temporal networks. *IEEE Transactions on Big Data*. DOI: 10.1109/TBDATA.2021.3058294.

[6] Li Y, Liu J, Zhao H, Sun J, Zhao Y, Wang G. Efficient continual cohesive subgraph search in large temporal graphs. *World Wide Web*, 2021, 24(5): 1483-1509. DOI: 10.1007/s11280-021-00917-z.

[7] Qin H, Li R H, Wang G, Qin L, Cheng Y, Yuan Y. Mining periodic cliques in temporal networks. In *Proc. the 35th IEEE International Conference on Data Engineering*, Apr. 2019, pp.1130-1141. DOI: 10.1109/ICDE.2019.00104.

[8] Zhang Q, Guo D, Zhao X, Li X, Wang X. Seasonal-periodic subgraph mining in temporal networks. In *Proc. the 29th ACM International Conference on Information and Knowledge Management,* Oct. 2020, pp.2309-2312. DOI: 10.1145/3340531.3412091.

[9] Qin H, Li R H, Wang G, Qin L, Yuan Y, Zhang Z. Mining bursting communities in temporal graphs. arXiv:1911.02780, 2019. https://arxiv.org/abs/1911.02780, Jul. 2022.

[10] Chu L, Zhang Y, Yang Y, Wang L, Pei J. Online density bursting subgraph detection from temporal graphs. *Proc. VLDB Endow.*, 2019, 12(13): 2353-2365. DOI: 10.14778/3358701.3358704.

[11] Palen L, Hughes A L. Social media in disaster communication. In *Handbook of Disaster Research*, Rodríguez H, Donner W, Trainor J E (eds.), Springer Cham, 2018, pp.497-518. DOI: 10.1007/978-3-319-63254-4_24.

[12] Jain V, Sharma A, Subramanian L. Road traffic congestion in the developing world. In *Proc. the 2nd ACM Symposium on Computing for Development*, Mar. 2012, Article No. 11. DOI: 10.1145/2160601.2160616.

[13] Cooper I, Mondal A, Antonopoulos G C. A SIR model assumption for the spread of COVID-19 in different communities. *Chaos, Solitons & Fractals*, 2020, 139: Article No. 110057. DOI: 10.1016/j.chaos.2020.110057.

[14] Barbieri N, Bonchi F, Galimberti E, Gullo F. Efficient and effective community search. *Data Min. Knowl. Discov.*, 2015, 29(5): 1406-1433. DOI: 10.1007/s10618-015-0422-1.

[15] Cui W, Xiao Y, Wang H, Wang W. Local search of communities in large graphs. In *Proc. the 2014 ACM SIGMOD International Conference on Management of Data*, Jun. 2014, pp.991-1002. DOI: 10.1145/2588555.2612179.

[16] Dai J, Li Y, Fan X, Sun J, Zhao Y. Finding early bursting cohesive subgraphs in large temporal networks. In *Proc. the 2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation*, Oct. 2021, pp.264-271. DOI: 10.1109/SWC50871.2021.00044.

[17] Li R H, Qin L, Yu J X, Mao R. Influential community search in large networks. *Proc. VLDB Endow.*, 2015, 8(5): 509-520. DOI: 10.14778/2735479.2735484.

[18] Li R, Qin L, Yu J X, Mao R. Finding influential communities in massive networks. *VLDB J.*, 2017, 26(6): 751-776. DOI: 10.1007/s00778-017-0467-4.

[19] Chen S, Wei R, Popova D, Thomo A. Efficient computation of importance based communities in web-scale networks using a single machine. In *Proc. the 25th ACM International Conference on Information and Knowledge Management*, Oct. 2016, pp.1553-1562. DOI: 10.1145/2983323.2983836.

[20] Bi F, Chang L, Lin X, Zhang W. An optimal and progressive approach to online search of top-k influential communities. *Proc. VLDB Endow.*, 2018, 11(9): 1056-1068. DOI: 10.14778/3213880.3213881.

[21] Zheng Z, Ye F, Li R H, Ling G, Jin T. Finding weighted k-truss communities in large networks. *Inf. Sci.*, 2017, 417: 344-360. DOI: 10.1016/j.ins.2017.07.012.

[22] Sun L, Huang X, Li R, Choi B, Xu J. Index-based intimate-core community search in large weighted graphs. *IEEE Trans. Knowl. Data Eng.*, 2022, 34(9): 4313-4327. DOI: 10.1109/TKDE.2020.3040762.

[23] Lahiri M, Berger-Wolf T F. Mining periodic behavior in dynamic social networks. In *Proc. the 8th IEEE International Conference on Data Mining*, Dec. 2008, pp.373-382. DOI: 10.1109/ICDM.2008.104.

[24] Qin H, Li R, Yuan Y, Wang G, Yang W, Qin L. Periodic communities mining in temporal networks: Concepts and algorithms. *IEEE Trans. Knowl. Data Eng.*, 2022, 34(8): 3927-3945. DOI: DOI: 10.1109/TKDE.2020.3028025.

[25] Maheshwari A, Zeh N. A survey of techniques for designing I/O-efficient algorithms. In *Algorithms for Memory Hierarchies*, Meyer U, Sanders P, Sibeyn J (eds.), Springer, 2003, pp.36-61. DOI: 10.1007/3-540-36574-5_3.

[26] Cheng J, Ke Y, Chu S, Özsu M. Efficient core decomposition in massive networks. In *Proc. the 27th IEEE International Conference on Data Engineering*, Apr. 2011, pp.51-62. DOI: 10.1109/ICDE.2011.5767911.

[27] Sun P, Wen Y, Duong T N B, Xiao X. GraphMP: I/O-efficient big graph analytics on a single commodity machine. *IEEE Trans. Big Data*, 2020, 6(4): 816-829. DOI: 10.1109/TBDATA.2019.2908384.

[28] Wen D, Qin L, Zhang Y, Lin X, Yu J X. I/O efficient core graph decomposition at web scale. In *Proc. the 32nd IEEE International Conference on Data Engineering*, May 2016, pp.133-144. DOI: 10.1109/ICDE.2016.7498235.

[29] Yuan L, Qin L, Lin X, Chang L, Zhang W. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.*, 2017, 26(2): 275-300. DOI: 10.1007/s00778-016-0451-4.

[30] Zhang Z, Yu J X, Qin L, Chang L, Lin X. I/O efficient: Computing SCCs in massive graphs. *VLDB J.*, 2015, 24(2): 245-270. DOI: 10.1007/s00778-014-0372-z.

[31] Jiang Y, Huang X, Cheng H. I/O efficient k-truss community search in massive graphs. *VLDB J.*, 2021, 30(5): 713-738. DOI: 10.1007/s00778-020-00649-y.

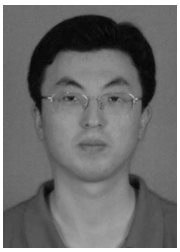
[32] Li Y, Wang G, Zhao Y, Zhu F, Wu Y. Towards k-vertex connected component discovery from large networks. *World Wide Web*, 2020, 23(2): 799-830. DOI: 10.1007/s11280-019-00725-6.

[33] Li Y, Sheng F, Sun J, Zhao Y, Wang G. A k-connected truss subgraph discovery algorithm in large scale dual networks. *Chinese Journal of Computers*, 2020, 43(9): 1721-1736. DOI: 10.11897/SP.J.1016.2020.01721. (in Chinese)

Yuan Li received his B.S., M.S., and Ph.D. degrees in computer science and technology from the Northeastern University, Shenyang, in 2009, 2011, and 2018, respectively. He is currently a lecture with the School of Information Science and Technology, North China University of Technology, Beijing. He is a member of CCF. His research interests include data management, data mining, and bioinformatics.

Jie Dai received his B.S. degree in computer science and technology from the North China University of Technology, Beijing, in 2022. He is currently a M.S. student at the School of Computer Science and Engineering, Northeastern University, Shenyang. His research interest is data mining.

Xiao-Lin Fan received her B.S. degree from Southwest University of Science and Technology, Mianyang, in 2018, and M.S. degree from North China University of Technology, Beijing, in 2022, both in software engineering. She is currently a Ph.D. student at Beihang University, Beijing. Her current research interests include data mining and knowledge graph.



Yu-Hai Zhao received his B.S., M.S., and Ph.D. degrees in computer science and technology from the Northeastern University, Shenyang, in 1999, 2004, and 2007, respectively. He is currently a professor with the School of Computer Science and Engineering, Northeastern University, Shenyang. He is a senior member of CCF. His current research interests include data management, data mining, and bioinformatics.

Guo-Ren Wang received his B.S., M.S., and Ph.D. degrees in computer science and technology from the Northeastern University, Shenyang, in 1988, 1991 and 1996, respectively. He is currently a professor with School of Computer, Beijing Institute of Technology, Beijing. He is a senior member of CCF. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and cloud data management. He has published more than 100 research papers.