2017 IEEE 33rd International Conference on Data Engineering

Fast Computation of Dense Temporal Subgraphs

Shuai Ma, Renjun Hu, Luoshu Wang, Xuelian Lin, Jinpeng Huai
SKLSDE Lab, Beihang University, Beijing, China
Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China
{mashuai, hurenjun, wangluoshu, linxl, huaijp}@buaa.edu.cn

***Abstract*—Dense subgraph discovery has proven useful in various applications of temporal networks. We focus on a special class of temporal networks whose nodes and edges are kept fixed, but edge weights constantly and regularly vary with timestamps. However, finding dense subgraphs in temporal networks is non-trivial, and its state of the art solution uses a filter-and-verification framework, which is not scalable on large temporal networks. In this study, we propose a data-driven approach to finding dense subgraphs in large temporal networks with T timestamps. (1) We first develop a data-driven approach employing hidden statistics to identifying k time intervals, instead of $T * (T + 1)/2$ ones (k is typically much smaller than T), which strikes a balance between quality and efficiency. (2) After proving that the problem has no constant factor approximation algorithms, we design better heuristic algorithms to attack the problem, by building the connection of finding dense subgraphs with a variant of the Prize Collecting Steiner Tree problem. (3) Finally, we have conducted an extensive experimental study to demonstrate the effectiveness and efficiency of our approach, using real-life and synthetic data.**

I. Introduction

Today *dynamic* has become an apparent feature of many data analytic systems and applications that could be modeled as graphs or networks, such as social network analysis, biological data analysis, recommendation systems and route planning [2], [19]. Hence, it is not surprising that dynamic networks have drawn significant attentions from both industry and academic communities. In fact, various alternative terms of dynamic networks are commonly used, such as temporal networks, dynamic graphs, evolutionary networks, evolving networks, time-dependent graphs and graph streams [2], [6]–[8], [10], [12], [14], [17], [20], [21], [26]–[28], [32], [33].

Dense subgraph discovery and analysis have been widely studied in static networks [3], [4], [13], [15], [16], [22], [23], [31], such as finding maximal cliques, k-core analyses and the Prize Collecting Steiner Tree problem. It is worth pointing out that dense subgraphs are a very general concept, and their concrete semantics highly depend on the studied problems and applications. How to properly transfer or define their semantics over to temporal networks is still in the early stage, *e.g.,* heavy subgraphs [7], hotspots [33], anomalies [6] and regions [8], not to mention effective and efficient algorithms.

In this study, we investigate a special class of temporal networks (see a recent survey [19]) such that their nodes and edges are kept fixed, but their edge weights constantly and regularly vary with timestamps. Essentially, a temporal network with T timestamps can be viewed as T snapshots of a static network such that the network nodes and edges are kept the same among these T snapshots, but the edge weights may be different in different network snapshots. Road traffic networks typically fall into this category [7], [14], [27], [32], and road traffic analyses are of particular importance for large cities, such as Beijing, New York, London and Paris, that are facing with heavy traffic congestions, one of the great challenges of urban computing [6], [7], [27], [34].

We also focus on a certain form of dense temporal subgraphs, which was initially studied in [7]. Formally speaking, a temporal subgraph corresponds to a connected subgraph measured by the sum of all its edge weights in a time interval, *i.e.,* a continuous sequence of timestamps. Intuitively, a dense subgraph that we consider corresponds to a collection of connected highly slow or jam roads (*i.e.,* a jam area) in road networks, lasting for a continuous sequence of snapshots. We next use an example to illustrate its basic idea, such the real-time Beijing traffic status snapshot report generated by a Baidu product [1], and is regularly updated every couple of minutes. Given a sequence of such snapshots in a day period, dense temporal subgraphs help to analyze which areas during what time periods are in jam conditions at Beijing.

Challenges and limitations. However, the problem of finding dense subgraphs in temporal networks is non-trivial, and it is already NP-complete even for a temporal network with a single snapshot and with $+1$ or -1 edge weights only, as observed in [7]. Even worse, it remains hard to approximate for temporal networks with single snapshots (Section IV). Moreover, given a temporal network with T timestamps, there are a total number of $T * (T + 1)/2$ time intervals to consider, which further aggravates the difficulty. Finally, the state of the art solution MEDEN [7] adopts a filter-and-verification framework that *even if a large portion of time intervals are filtered, there often remain a large number of time intervals to verify*. Hence, MEDEN is not scalable when temporal networks have a large number of nodes/edges or a large number T of timestamps.

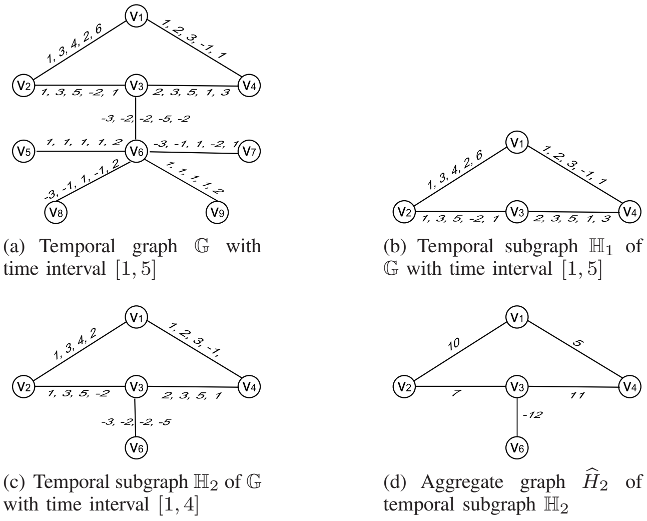
Contributions. To this end, we propose a highly efficient data-driven approach, instead of filter-and-verification, that employs hidden data statistics to find dense subgraphs in large temporal networks in an efficient and effective way.

(1) We first develop a data-driven approach to identifying k time intervals from $T * (T + 1)/2$ time intervals (Section III), striking a balance between quality and efficiency, in which T is the number of snapshots and k is a small constant, *e.g.,* 10. This is achieved by exploring the characteristics of time intervals involved with dense subgraphs based on a novel *evolving convergence phenomenon*.

(2) We then design an algorithm for computing dense subgraphs, given a time interval (Section IV). After showing that the problem has no constant factor approximation algorithms, we develop a heuristic algorithm (by proving the equivalence

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.95

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.



(a) Temporal graph $\mathbb{G}$ with time interval $[1, 5]$

(b) Temporal subgraph $\mathbb{H}_1$ of $\mathbb{G}$ with time interval $[1, 5]$

(c) Temporal subgraph $\mathbb{H}_2$ of $\mathbb{G}$ with time interval $[1, 4]$

(d) Aggregate graph $\widehat{H}_2$ of temporal subgraph $\mathbb{H}_2$

Fig. 1. Running example

of finding dense subgraphs and finding a maximum net worth subtree, a variant of the Prize Collecting Steiner Tree problem [13], [22]) and three optimization techniques to improve the efficiency while retain a high quality.

(3) Using BJDATA and SYNDATA, we conduct an extensive test (Section V). (a) We find that our method FIDES is $2,980$ and $1,079$ times on average faster than the state of the art solution MEDEN [7] on BJDATA and SYNDATA, respectively. (b) The dense subgraphs found by FIDES have a very high quality, *i.e.,* about 100.28% and 99.84% on average of those found by MEDEN on BJDATA and SYNDATA, respectively. (c) Finally, MEDEN already ran out of memory for temporal graphs with $150,000$ nodes and 2,000 snapshots.

II. PRELIMINARY

In this section, we introduce the basic definitions of temporal graphs and the problem to be investigated.

A. Basic Concepts

We first introduce basic concepts of temporal graphs.

Temporal graphs. A *temporal graph* $\mathbb{G}(V,\ E,\ F,\ T_b,\ T_e)$ is a weighted undirected graph with edge weights varying with timestamps, where (1) V is a finite set of nodes, (2) $E \subseteq V \times V$ is a finite set of edges, in which (u, v) or $(v, u) \in E$ denotes an undirected edge between nodes u and v, (3) for each timestamp $t \in [T_b, T_e]$, $F^t()$ is a total weight function that maps each edge in E to a *positive or negative* rational number, and (4) $[T_b, T_e]$ is a time interval representing $(T_e - T_b + 1)$ timestamps, in which $T_b \leq T_e$ are the beginning and ending timestamps (positive integers), respectively. When it is clear from the context, we simply use $\mathbb{G}(V,\ E,\ F)$ to denote temporal graphs for clarity.

Observe that we are considering a special class of temporal networks (see a recent survey [19]) such as road networks and communication networks, in which graph nodes and edges are kept fixed, but the edge weights vary with respect to timestamps. Intuitively, (1) a temporal graph $\mathbb{G}(V,\ E,\ F)$ essentially denotes a sequence $< G_1(V, E, F^1)$, …, $G_T(V, E, F^T) >$ of $T = T_e - T_b + 1$ standard graphs, and (2) the edge weights $F^t(e)$ specify the distances, communication latencies or travelling duration [7], [14], [19], [27], [32], or the affinity or collaborative compatibility [15] between the two corresponding nodes of edges e at timestamps t. Essentially, positive/negative edge weights model relationships opposed in nature, *e.g.,* fast/congested traffic and friend/foe relationships.

We also say that $G_i(V, E, F^i)$ $(i \in [1, T])$ is a *snapshot* of temporal graph $\mathbb{G}(V,\ E,\ F)$ at timestamp $T_b + i - 1$.

Temporal subgraphs. Temporal graph $\mathbb{H}(V_s, E_s, F_s, T_{b_s}, T_{e_s})$ is a *subgraph* of temporal graph $\mathbb{G}(V, E, F, T_b, T_e)$, denoted by $\mathbb{G}[V_s,\ E_s, T_{b_s}, T_{e_s}]$, if $V_s \subseteq V$, $E_s \subseteq E$, $T_{b_s}, T_{e_s} \in [T_b,\ T_e]$, and $F^t_s(e) = F^t(e)$ for any $t \in [T_{b_s}, T_{e_s}]$ and $e \in E_s$.

That is, subgraph $\mathbb{G}[V_s, E_s, T_{b_s}, T_{e_s}]$ only contains a subset of nodes and edges of graph $\mathbb{G}$, and it is restricted within the time interval $[T_{b_s}, T_{e_s}]$ that falls into $[T_b, T_e]$.

When $V_s = V$ and $E_s = E$, we also simply use $\mathbb{G}[T_{b_s}, T_{e_s}]$ to denote temporal subgraph $\mathbb{G}[V_s, E_s, T_{b_s}, T_{e_s}]$ for clarity.

Aggregate graphs. Given a temporal graph $\mathbb{G}(V, E, F, T_b,\ T_e)$, its *aggregate graph* $\widehat{G}(V, E, f)$ is a standard undirected weighted graph that has the same sets of nodes and edges as $\mathbb{G}$, and for each edge $e \in E$, its weight $f(e)$ is the sum of weights $F^t(e)$ with $t \in [T_b, T_e]$, *i.e.,* $f(e) = \sum_{t=T_b}^{T_e} F^t(e)$.

Cohesive density. The *cohesive density* of an aggregate graph $\widehat{G}(V, E, f)$, denoted by $\mathsf{cdensity}(\widehat{G})$, is equal to the sum of all the edge weights, *i.e.,* $\mathsf{cdensity}(\widehat{G}) = \sum_{e \in E} f(e)$.

Positive cohesive density. The *positive cohesive density* of an aggregate graph $\widehat{G}(V, E, f)$, denoted by $\mathsf{cdensity}^+(\widehat{G})$, is equal to the sum of all the positive edge weights, *i.e.,* $\mathsf{cdensity}^+(\widehat{G}) = \sum_{e \in E, f(e)>0} f(e)$.

The (positive) cohesive density of a temporal graph $\mathbb{G}$ is simply equal to the (positive) cohesive density of its corresponding aggregate graph $\widehat{G}$.

Dense subgraphs. Given a temporal graph $\mathbb{G}(V, E, F)$, its *dense subgraph* is a *connected temporal subgraph* $\mathbb{G}[V_s,\ E_s, i, j]$ with the *greatest cohesive density*.

We next illustrate these concepts with an example.

Example 1: (1) Figure 1(a) depicts a temporal graph $\mathbb{G}$ with 9 nodes, 9 edges and T = 5 timestamps.

(2) Figure 1(b) shows a temporal subgraph $\mathbb{H}_1$ of $\mathbb{G}$ with 4 nodes, 4 edges and time interval $[1, 5]$, and Fig. 1(c) shows a temporal subgraph $\mathbb{H}_2$ of $\mathbb{G}$ with 5 nodes, 5 edges and time interval $[1, 4]$, respectively.

(3) Figure 1(d) shows the aggregate graph $\widehat{H}_2$ of temporal subgraph $\mathbb{H}_2$ whose cohesive density $\mathsf{cdensity}(\widehat{H}_2)$ = 21 and positive cohesive density $\mathsf{cdensity}^+(\widehat{H}_2)$ = 33, respectively.

(4) One can verify that the temporal subgraph $\mathbb{H}_1$ in Fig. 1(b) is indeed the only dense subgraph of $\mathbb{G}$ with the greatest cohesive density $\mathsf{cdensity}(\widehat{H}_1)$ = 44. □

Remarks. In a road network snapshot generated by a Baidu product [1], red, yellow and green colored roads denote the traffic congestion, slow traffic and fast traffic conditions, respectively. If we replace red, yellow and green colors with +2, +1 and −1, respectively, we have a road network snapshot

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

```
Algorithm basicMEDEN
Input: Temporal graph $\mathbb{G}(V, E, F)$.
Output: $\mathbb{G}[V_s, E_s, i, j]$, a solution of FDS.
1. for each time interval $[i,j]$ $(i \le j \in [T_b, T_e])$ do
2.     let $\widehat{G}_{[i,j]}(V, E, f)$ be the aggregate graph of $\mathbb{G}(V, E, F, i, j)$;
3.     $\mathsf{UB}_{\mathsf{sop}}[\mathsf{i}, \mathsf{j}] := \mathsf{cdensity}^{+}(\widehat{G}_{[i,j]})$;
4. Estimate a lower bound LB for the solution of FDS;
5. for each time interval $[i,j]$ $(i \le j \in [T_b, T_e])$ do
6.     Prune $[i,j]$ if $\mathsf{UB}_{\mathsf{sop}}[\mathsf{i}, \mathsf{j}] \le \mathsf{LB}$;
7. for each not pruned time interval $[i,j]$ $(i \le j \in [T_b, T_e])$ do
8.     $\widehat{G}'_{[i,j]} := \mathsf{topDown}(\widehat{G}_{[i,j]})$ ;
9. $\mathbb{G}[V_s, E_s, i, j]$ := a subgraph with the greatest cohesive density;
10.return $\mathbb{G}[V_s, E_s, i, j]$.
```

Fig. 2. Algorithm basicMEDEN

with both positive and negative edge weights. Moreover, it is quite obvious that dense subgraphs correspond to jam regions.

B. Finding Dense Subgraphs

We next present the problem statement and its baseline solutions [7]. Given a temporal graph $\mathbb{G}(V, E, F)$, the problem of finding dense subgraphs (referred to as FDS) is to find a *connected temporal subgraph* $\mathbb{H} = \mathbb{G}[V_s, E_s, i, j]$ whose aggregate graph has the *greatest* cohesive density $\mathsf{cdensity}(\widehat{H})$.

Intractability. It is already known that the FDS problem is intractable, as observed in [7].

Proposition 1: *The FDS problem is NP-complete, even for a temporal network with a single snapshot and with $+1$ or -1 edge weights only [7].* □

Baseline solutions. We next present the details of algorithm basicMEDEN developed in [7], shown in Fig. 2. Here procedure topDown aims to find a subgraph of an aggregate graph with a higher cohesive density.

Algorithm basicMEDEN. Given a temporal graph $\mathbb{G}$, it returns a solution of FDS. It first computes an upper bound of the positive cohesive density $\mathsf{UB}_{\mathsf{sop}}[\mathsf{i}, \mathsf{j}]$ for each $[i, j]$ of the $T * (T+1)/2$ time intervals ($i \le j \in [T_b, T_e]$ and $T = T_e - T_b + 1$) (lines 1–3). It then uses procedure topDown to compute the solutions for the k time intervals $[i, j]$ that have the top–k highest $\mathsf{UB}_{\mathsf{sop}}[\mathsf{i}, \mathsf{j}]$, and sets LB to be the highest cohesive density of the k computed dense subgraphs (line 4). Using LB and $\mathsf{UB}_{\mathsf{sop}}[\mathsf{i}, \mathsf{j}]$, the algorithm prunes time intervals (lines 5–6), and uses topDown again to compute the solutions for all not pruned time intervals (lines 7–8). The subgraph found with the greatest cohesive density is finally returned (lines 9-10).

Remarks. (1) The state of the art solution MEDEN was proposed in [7], which adopted a filter-and-verification framework. For clarity, here we only present the basic version basicMEDEN. The sophisticated version MEDEN incorporates a more scalable filtering technique by grouping time intervals, and it was reported that MEDEN achieved an order of magnitude performance improvement over basicMEDEN [7]. (2) We will compare our approach with the sophisticated version MEDEN in the experimental study (Section V).

III. Identifying Time Intervals

Our data-driven approach to finding dense subgraphs in temporal graphs consists of two key components: (1) identifying k time intervals and (2) finding a dense subgraph for a given time interval. We first introduce how to identify k time intervals based on a novel evolving convergence phenomenon and an independent and identically distributed (i.i.d.) weight assumption. We consider a temporal graph $\mathbb{G}(V, E, F, T_b, T_e)$, and *w.l.o.g.* we assume $T_b = 1$ and $T_e = T$ in the sequel.

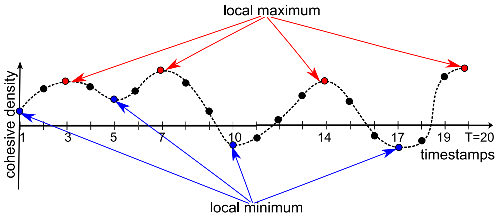


Fig. 3. Local minima and maxima

Observe that there are $T * (T+1)/2$ time intervals in total, and even if T is not a large number, *e.g.,* 2,000, there are more than 2×10^6 time intervals to investigate, which involves with too much computational cost. Even though MEDEN filters a large portion of unnecessary time intervals, say 99% [7], there remain a large number of time intervals to verify, *e.g.,* 2×10^4 in the above case. Recall that the FDS problem remains NP-complete for single snapshots (Proposition 1, Section II-B). Hence, we develop a data-driven approach, instead of filter-and-verification, to exploring k time intervals only, aiming at striking a balance between quality and efficiency. Here k is typically a small constant, *e.g.,* 10.

A. Characteristics of Time Intervals

We first present the key characteristics of the time intervals involved with dense subgraphs.

Cohesive density curves. Given a temporal graph $\mathbb{G}(V, E, F)$, its *cohesive density curve* is a function $y = W(x)$ such that for each $x \in [1, T]$, $W(x)$ is the cohesive density of the aggregate graph of temporal subgraph $\mathbb{G}[x, x]$.

Local maximum and minimum. We consider a cohesive density curve $y = W(x)$ of temporal graph $\mathbb{G}(V, E, F)$.

The curve y is said to have a *local maximum* at a point x^* if there exists some positive integer δ such that $W(x^*) \ge W(x)$ for all x with $|x - x^*| \le \delta$.

Similarly, y is said to have a *local minimum* at a point x^* if there exists some positive integer δ such that $W(x^*) \le W(x)$ for all x with $|x - x^*| \le \delta$.

Example 2: Consider the cohesive density curve $y = W(x)$ of temporal graph $\mathbb{G}$ in Fig. 3, in which $W(x)$ is the cohesive density of the aggregate graph of temporal subgraph $\mathbb{G}[x, x]$ ($x \in [1, 20]$). Here the curve y has a local maximum at the points $x = 3, 7, 14, 20$ and a local minimum at the points $x = 1, 5, 10, 17$, respectively. □

We now present the first key observation inspired by the convergent evolution in evolutionary biology, based on which we prove a very important characteristic of the time intervals involved with dense subgraphs.

Evolving convergence phenomenon. Consider a sequence $< \widehat{G}_1(V, E, f_1), \ldots, \widehat{G}_T(V, E, f_T) >$ of the T corresponding aggregate graphs of the temporal subgraphs $\mathbb{G}[1, 1]$, …, $\mathbb{G}[T, T]$ of temporal graph $\mathbb{G}$.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

The evolving convergence phenomenon asserts that if there exists an edge in $\widehat{G}_i$ ($i \in [1, T-1]$) whose weight is no less (respectively no greater) than its weight in $\widehat{G}_{i+1}$, then for all edges, their weights in $\widehat{G}_i$ are no less (respectively no greater) than their corresponding weights in $\widehat{G}_{i+1}$. Intuitively, this says that all edges evolve in a convergent manner, *i.e.,* the increase of one edge weight indicates that all the remaining edge weights do not decrease, and vice versa.

Proposition 2: *To find the dense subgraph, we only need to consider the time intervals $[i, j]$ such that the cohesive density curve has a local maximum at certain point between i and j under the evolving convergence phenomenon.* □

The evolving convergence phenomenon assures the correctness of Proposition 2 that gives a precise characterization of the time intervals involved with dense subgraphs, and may not completely hold. However, this phenomenon remains effective to a large extent in practice, and it is indeed a statistical characteristic to capture the general tendency of edge weight changes on certain temporal graphs.

Considering the Beijing road network for instance, the phenomenon *almost holds*. There are morning and evening peaks at Beijing, and it is typically common that a majority of roads become slow/jam during the peak time, and enter a faster traffic condition after the peak time ends, although individual roads may be *physically isolated*. In this case, it is easy to see that it is very likely that the dense subgraph lies in a time interval containing peaks.

Moreover, it is trivial to verify the following two characteristics, which further help us to identify the time intervals involved with dense subgraphs.

Fact 1: *All dense subgraphs have a non-negative cohesive density.* □

Intuitively, *Fact 1* tells us that dense subgraphs are more concerned with the positive weight edges, and the positive cohesive density may give better estimations for the potential time intervals, compared with the cohesive density.

Fact 2: *Temporal subgraph $\mathbb{G}[i, j]$ ($i \le j \in [1, T]$) with a higher positive cohesive density has a higher probability of containing a dense subgraph under the assumption of i.i.d. edge weights within single snapshots $\mathbb{G}[i, i]$ ($i \in [1, T]$).* □

Moreover, *Fact 2* tells us that temporal subgraph with the highest positive cohesive density has *a very high probability* of containing the dense subgraph which we are looking for. However, the i.i.d. edge weight assumption may not hold completely in practice. Hence, we compute the time intervals whose corresponding temporal subgraphs have the top–k highest positive cohesive densities, instead of the one with the highest positive cohesive density only, from those time intervals $[i, j]$ having a local maximum in the curve by Proposition 2.

Remarks. As will be shown by the experimental study (Section V), the three characteristics (*i.e.,* Proposition 2, Facts 1 and 2) together assure a pretty good estimation of the time intervals involved with dense subgraphs, even when the phenomenon does not hold completely, but almost holds.

```
Algorithm maxTInterval
Input: Temporal graph $\mathbb{G}(V, E, F)$, a positive rational $\xi$.
Output: $k/2$ aggregate graphs with largest positive cohesive densities.
1.  let $y = W(x)$ be the cohesive density curve of $\mathbb{G}$;
2.  let $x_1 < \ldots < x_h$ be the local maxima of $y$;
3.  for each $i \in [1, h]$ do
4.      $x_i.l$ := the largest $x$ with $x \le x_i$ and $|W(x) - W(x_i)| \ge \xi$;
5.      $x_i.u$ := the smallest $x$ with $x \ge x_i$ and $|W(x) - W(x_i)| \ge \xi$;
6.  while there are local maxima $x_i, x_j$ with $x_i.l \le x_j.l \le x_i.u$ do
7.      $x_i.u = \max(x_i.u, x_j.u)$;        $h := h - 1$;
8.      Remove local maximum $x_j$;   /*merging overlapped intervals*/
9.  $S := \{[x_i.l, x_j.u] \mid i \le j$ and $i, j \in [1, h]\}$;
10. $S'$ := the top $k/2$ intervals in $S$ whose aggregate graphs have
           the largest positive cohesive densities;
11. for each time interval $[l, u] \in S'$ do
12.     Find the least $l_s \le l$ and largest $u_s \ge u$ such that
        cdensity$^+(\mathbb{G}[l_s, u])$, cdensity$^+(\mathbb{G}[l, u_s])>$ cdensity$^+(\mathbb{G}[l, u])$;
13.     Replace $[l, u]$ with $[l_s, u_s]$;                  /*enlarge intervals*/
14. $R := \{\mathbb{G}[l, u] \mid [l, u] \in S'\}$;
15. return the aggregate graphs of temporal graphs in $R$.
```

Fig. 4. Algorithm maxTInterval using local maxima

B. Computing Top-K Time Intervals

We finally present our methods to estimate the top–k time intervals whose aggregate graphs have the greatest positive cohesive densities, following the analysis in Section III-A.

Algorithm maxTInterval uses local maxima to identify the time intervals, and is presented in Fig. 4. Given a temporal graph $\mathbb{G}(V, E, F)$ and a positive rational ξ, it outputs a set of $k/2$ aggregate graphs with the largest positive cohesive densities. It first computes the h local maxima of the cohesive density curve of $\mathbb{G}$ (lines 1–2). For each local maximum x_i ($i \in [1, h]$), it finds the largest timestamp $x_i.l \in [1, T]$ such that $x_i.l \le x_i$ and $|W(x_i.l) - W(x_i)| \ge \xi$, and the smallest timestamp $x_i.u \in [1, T]$ such that $x_i.u \ge x_i$ and $|W(x_i.u) - W(x_i)| \ge \xi$ (lines 3–5). Here constant $\xi = \sum_{i=1}^{T-1} |W(i+1) - W(i)|/(T-1)$ is the average density change of the cohesive density curve of $\mathbb{G}$. It then repeatedly merges overlapping time intervals by removing local maxima. For instance, for local maxima x_i and x_j, it removes x_j if $[x_i.l, x_i.u]$ and $[x_j.l, x_j.u]$ overlap (lines 6–8). The intuition behind these is that close maxima can be treated as a large one. Using $x_i.l$ and $x_i.u$ ($i \in [1, h]$), it generates a set S of $h * (h+1)/2$ time intervals (line 9), and computes a subset S' of $k/2$ intervals whose resulting aggregate graphs have the top–k/2 largest positive cohesive densities (line 10). After this, each time interval $[l, u] \in S'$ is further enlarged to produce an aggregate graph with a higher positive cohesive density (lines 11–13). To speed up the process, the decrement of l_s and increment of u_s are first initialized to 1, and are doubled every 4 successful tries. It finally computes and returns the $k/2$ aggregate graphs of temporal graphs for the time intervals in S' (lines 14–15).

Algorithm minTInterval uses local minima to identify the remaining top–k/2 time intervals that contain local maxima, which is along the same lines as algorithm maxTInterval except the following: It (1) computes h local minima $x_1 < \ldots < x_h$, instead of local maxima (line 2), (2) produces a set S of $h * (h-1)/2$ time intervals in the form of $\{[x_i.u, x_j.l] \mid i < j$ and $i, j \in [1, h]\}$ (line 9), and (3) shrinks the intervals, instead of enlarging intervals (lines 11–13).

We next use an example to illustrate how to generate time

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

intervals using local maxima and minima.

Example 3: Consider the curve $y = W(x)$ in Fig. 3 again.

(1) Assume without loss of generality that $3.l$ = 2, $7.l$ = 6, $14.l$ = 13, $20.l$ = 19 and $3.u$ = 4, $7.u$ = 8, $14.u$ = 15, $20.u$ = 20 for the local maximum at points x = 3, 7, 14, 20, respectively. Here no local maxima can be merged. Then the set S of time intervals at line 9 of maxTInterval is $\{[2, 4]$, $[2, 8]$, $[2, 15], [2, 20]$, $[6, 8]$, $[6, 15]$, $[6, 20]$, $[13, 15], [13, 20]$, $[19, 20]\}$. If the positive cohesive density of $\mathbb{G}[1, 8]$ is larger than $\mathbb{G}[2, 8]$, then maxTInterval replaces $[2, 8]$ with $[1, 8]$.

(2) Assume without loss of generality that $1.l$ = 1, $5.l$ = 4, $10.l$ = 9, $17.l$ = 16 and $1.u$ = 2, $5.u$ = 6, $10.u$ = 11, $17.u$ = 18 for the local minimum at points x = 1, 5, 10, 17, respectively. Again, no local minima can be merged. Then the set S of time intervals at line 9 of algorithm minTInterval is $\{[2, 4], [2, 9]$, $[2, 16], [6, 9], [6, 16], [11, 16]\}$. If the positive cohesive density of $\mathbb{G}[6, 15]$ is larger than $\mathbb{G}[6, 16]$, then minTInterval replaces $[6, 16]$ with $[6, 15]$. □

Time complexity analysis. Algorithms maxTInterval and minTInterval both run in $O((T + h^2) \cdot |E|)$ time, in which h is the number of local maxima or minima.

Observe the following. (1) It first takes $O(T \cdot |E|)$ time to generate the cohesive density curve (line 1), and $O(T)$ time to find the local maxima (line 2). (2) Then it takes $O(h \cdot \log h)$ time to merge local maxima (lines 3–8), and $O(h^2)$ time to generate the time intervals of the temporal subgraphs (line 9). (3) When computing the positive cohesive densities of the aggregate graphs, it takes $O(h^2 \cdot |E|)$ time. For each edge e and timestamp t, let $AF(e, t)$ be $\sum_{i=1}^{t} F^i(e)$. Thus the positive cohesive density of each aggregate graph can be computed in $O(|E|)$ time using $AF(e, t)$. The top–k/2 intervals are retrieved in $O(h^2 \cdot \log k)$ time (lines 10). (4) When tuning the top–k/2 time intervals (lines 11–13), it takes $O(k \cdot \log T \cdot |E|)$ time since each interval can be updated at most $O(\log T)$ times and each update needs to recompute the positive cohesive density of a new aggregate graph. Note that here k is a small constant, *e.g.,* 10 or 15, and h is typically much smaller than T. Putting these together, algorithm maxTInterval takes $O((T + h^2) \cdot |E|)$ time in total. And, it is similar to show that algorithm minTInterval runs in $O((T + h^2) \cdot |E|)$ time as well.

IV. Computing Dense Subgraphs

We now explain how to compute the dense subgraph for a given time interval. This reduces to the problem of finding the subgraph of an aggregate graph with the highest cohesive density, which remains NP-hard as observed in [7]. We first show that the problem has no constant factor approximation algorithms, and then establish the connection between the problem of finding the dense subgraph in an aggregate graph and the *Net Worth Maximization problem* (NWM), a variant of the Prize Collecting Steiner Tree problem [13], [22]. We then develop algorithm heuristics to attack the problem. Finally, we present our complete solution FIDES for finding dense subgraphs in temporal networks.

A. Approximation Hardness

The hardness is verified by a reduction from *the Net Worth Maximization optimization problem* (NWM), a variant of the Prize Collecting Steiner Tree problem [13], [22]. Given an undirected graph $G(V, E)$, a non-negative edge weight $w(e)$ for each edge $e \in E$ and a non-negative node weight $p(v)$ for each node $v \in V$, the NWM problem is to find a subtree $ST(V_{st}, E_{st})$ that maximizes its net worth $NW(ST)$ = $\sum_{v \in V_{st}} p(v) - \sum_{e \in E_{st}} w(e)$. It is known that the NWM problem is NP-complete, and is hard to approximate: it is NP-hard to approximate the optimum Net Worth within any constant factor [13], [22].

```
Input: Aggregate graph $\widehat{H}(V, E, f)$.
Output: Converted graph $\vec{H}(V', E', p, w)$.
$\widehat{H}^+$ := $\widehat{H}$ with non-negative edges only;
Compute the connected components $\mathrm{CC}_1, \ldots, \mathrm{CC}_l$ of $\widehat{H}^+$ ;
let $V'$ := $\{u_1, \ldots, u_l\}$;
for each $i \in [1, l]$ do $p(u_i)$ := the total edge weight of $\mathrm{CC}_i$;
if there are negative edges between $\mathrm{CC}_i$ and $\mathrm{CC}_j$ $(i, j \in [1, l])$;
then $E'$ := $E' \cup \{(u_i, u_j)\}$;
        $w(u_i, u_j)$ := |the largest negative edge weight|;
return $\vec{H}$.
```

Fig. 5. Procedure convertAG

To show the approximation hardness, we use *approximation factor preserving reduction* (AFP-reduction) [11], [30], a certain form of reduction that retains approximation bounds. Let Π_1 and Π_2 be two maximization optimization problems. An AFP-reduction from Π_1 to Π_2 is a pair of PTIME functions $(h,\ g)$ that satisfies the following conditions:

(1) for any instance I_1 of Π_1, $I_2 = h(I_1)$ is an instance of Π_2 such that $\mathsf{opt}_2(I_2) \geq \mathsf{opt}_1(I_1)$, where $\mathsf{opt}_1(I_1)$ (respectively $\mathsf{opt}_2(I_2)$) is the value of an optimal solution to I_1 (respectively I_2), and

(2) for any feasible solution s_2 to I_2, $s_1 = g(s_2)$ is a feasible solution to I_1 such that $\mathsf{obj}_1(s_1) \geq \mathsf{obj}_2(s_2)$, where $\mathsf{obj}_1()$ (respectively $\mathsf{obj}_2()$) is a function measuring the value of a solution to I_1 (respectively I_2).

Finding an optimal dense subgraph of an aggregate graph is non-trivial, as shown below.

Theorem 3: *The cohesive density achieved by an optimal subgraph of an aggregate graph is* NP*-hard to approximate within any constant factor.* □

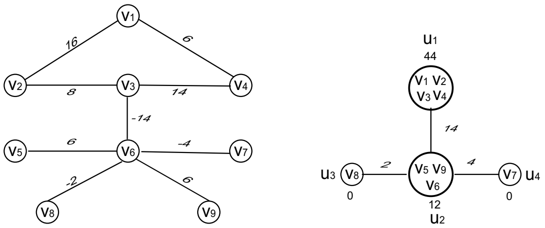
Proof Sketch: We show that there exists an AFP-reduction (h, g) from the NWM problem to the problem of finding the dense subgraph of an aggregate graph, from which the conclusion follows since the NWM problem is NP-hard to approximate within any constant factor [13], [22]. □

B. Connections with the NWM Problem

Theorem 3 tells us that heuristic algorithms are essentially the practical solutions on which we should focus, as its counterpart the NWM problem does [22]. We shall reduce the problem of finding the dense subgraph in an aggregate graph with *positive or negative* edge weights to the NWM problem, based on a notion of *converted graphs* that are undirected graphs with *non-negative* node and edge weights.

We next present the details of procedure convertAG in Fig. 5, which takes as input an aggregate graph $\widehat{H}(V, E, f)$, and returns its converted graph $\vec{H}(V', E', p, w)$, an undirected graph with non-negative node and edge weights.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.



(a) Aggregate graph $\widehat{G}$ (b) Converted graph $\vec{G}$

Fig. 6. Example of converted graphs

Procedure convertAG first generates graph $\widehat{H}^+$ by removing all the edges in $\widehat{H}$ with negative weights (line 1), and then computes the connected components of $\widehat{H}^+$ (line 2). For each connected component CC_i ($i \in [1, l]$), there is a corresponding node u_i in $\vec{H}$, whose weight $p(u_i)$ is equal to the total edge weight of CC_i (lines 3-4). An edge (u_i, u_j) $(i, j \in [1, l])$ is included in $\vec{H}$ if there are negative edges between CC_i and CC_j in $\widehat{H}$, and the edge weight $w(u_i, u_j)$ is the absolute value of the largest negative edge weight among all negative edges between CC_i and CC_j (lines 5-7). Finally, the converted graph $\vec{H}(V', E', p, w)$ is returned (line 8).

Example 4: Consider the temporal graph $\mathbb{G}$ in Fig. 1(a) again, and its aggregate graph $\widehat{G}$ in Fig. 6(a). The converted graph $\vec{G}$ of $\widehat{G}$ computed by convertAG is shown in Fig. 6(b). □

With a close look at the above procedure convertAG, it is easy to verify the following, which establishes the connection between the dense subgraphs in an aggregate graph and the maximum net worth subtrees in the converted graph.

Proposition 4: *Finding a dense subgraph in an aggregate graph $\widehat{H}$ is equivalent to finding a maximum net worth subtree in the converted graph* convertAG($\widehat{H}$). □

Remarks. (1) It is worth mentioning that procedure convertAG *reduces the size of aggregate graphs*, which further helps to improve the efficiency of finding the dense subgraphs, as shown by Example 4. (2) Different from aggregate graphs, converted graphs have only *non-negative* node and edge weights.

C. Algorithm Optimizations

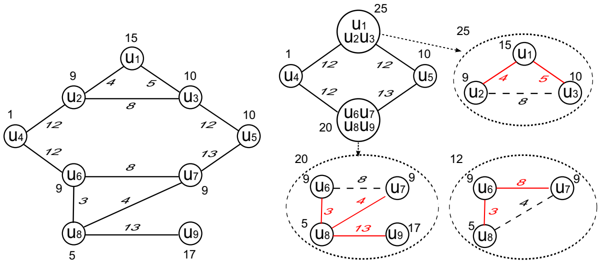
Proposition 4 tells us that the algorithm of the NWM problem [22] provides us a basic solution. We next investigate optimization techniques that could be employed to improve the performance for finding the dense subgraphs, in which we also incorporate the strong pruning technique that has been proven effective for the NWM problem [22].

(1) Strong merging. After having a converted graph, we repeatedly merge two neighboring nodes such that if one of them belongs to an optimal maximum net worth subtree, then the other must belong to as well. As a side benefit, this further reduces the size of the converted graph.

We next present the details of a basic version of procedure strongMerging in Fig. 7. It takes as input a converted graph $\vec{H}$, and returns its merged converted graph $\vec{H}'$. It repeatedly merges neighboring nodes until there are no changes, and two nodes u, v are merged if both their node weights are equal to or larger than the edge weight $w(u, v)$ (lines 1-7). Finally, it returns the merged converted graph $\vec{H}'$ (line 8).

```
Input: Converted graph $\vec{H}$.
Output: Merged converted graph $\vec{H}'$.
1. while there are changes do
2.   for any nodes $u, v$ with $p(v) \geq w(u, v)$ and $p(u) \geq w(u, v)$ do
3.     Merge $u$ and $v$ into a single node $x$;
4.     $p(x) := p(v) + p(u) - w(u, v)$;
5.     Remove edge $(u, v)$ from $\vec{H}$;
6.     Replace all edges $(v, y)$ and $(u, y)$ with $(x, y)$;
7.     $w(x, y) := \min(w(u, y), w(v, y))$;
8. return $\vec{H}'$.
```

Fig. 7. Procedure strongMerging



(a) Converted graph $\vec{H}$ (b) Merged converted graph $\vec{H}'$

Fig. 8. Example for strong merging

However, there may exist different ways to compute the weights of the merged nodes, as shown below.

Example 5: (1) Consider the converted graph $\vec{H}$ in Fig. 8(a). It is easy to verify that nodes u_1, u_2, u_3 are merged by procedure strongMerging. However, there are 2 different ways to merge u_1, u_2 and u_3 by strongMerging, each of which results in a different node weight. More specifically, when merging using (a) edges (u_1, u_2) and (u_2, u_3) and (b) edges (u_1, u_2) and (u_1, u_3), the merged node has weights $15 + 9 + 10 - 4 - 8 = 22$ and $15 + 9 + 10 - 4 - 5 = 25$, respectively. Note that edges (u_1, u_3) and (u_2, u_3) could not be used to merge nodes together, as (u_2, u_3) has a larger weight than (u_1, u_2), and hence will be discarded after merging nodes u_1 and u_3 (at lines 6–7 in Fig. 7).

(2) One can easily check that (a) when nodes u_6, u_7, u_8 are merged using edges (u_6, u_7) and (u_6, u_8), node u_9 cannot be merged with these nodes; (b) When nodes u_6, u_7, u_8 are merged using edges (u_6, u_8) and (u_7, u_8), node u_9 can be further merged with nodes u_6, u_7, u_8. That is, the way how nodes are merged also has effects on the merging process.

(3) Note that the converted graph $\vec{G}$ shown in Fig. 6(b) cannot be further merged. □

To address these, we maintain a minimum spanning tree for each (merged) node in the process of procedure strongMerging, which leads to the need of merging two minimum spanning trees when merging nodes. Hence, procedure strongMerging further uses Sleator–Tarjan dynamic trees [29] to achieve a good performance, as shown below.

Proposition 5: *For a converted graph $\vec{H}(V_H, E_H)$, the extra cost of maintaining minimum spanning trees for procedure* strongMerging *is bounded by $O(|E_H| \log |V_H|)$.* □

Proof Sketch: Let T be the merged minimum spanning subtree of two minimum spanning subtrees $T_1(V_1, E_1)$ and $T_2(V_2, E_2)$ in the process of strongMerging. We first show that it suffices to consider a T consisting of edges in E_1, E_2

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

```
Input: Minimum spanning tree $T$ of $\vec{H}'$.
Output: An optimal subtree $ST$ of $T$.
1. Randomly select a node as the root of $T$;
2. for each node $u$ in $T$ do $nw(u) := p(u)$;
3. for all nodes $u$ in $T$ in a bottom-up fashion do
4.   for each child node $v$ of $u$ do
5.     if $nw(v) < w(u, v)$ then remove edge $(u, v)$;
6.     else $nw(u) := nw(u) + nw(v) - w(u, v)$;
7. $u_r := \mathsf{argmax_u}\{nw(u)\}$;
8. return the subtree $ST$ rooted at $u_r$.
```

Fig. 9. Procedure strongPruning

and those between V_1 and V_2 only. Second, it is easy to see that merging T_1 and T_2 is equivalent to maintaining the minimum spanning tree in a graph $T_1 \cup T_2$ after inserting the edges between V_1 and V_2. Using Sleator–Tarjan dynamic trees, it takes $O(\log(|V_1| + |V_2|))$ time to deal with an inserted edge. From these, we have the conclusion. □

(2) Strong pruning. Strong pruning is an effective technique for solving the NWM problem developed in [22], [25] for finding an optimal net worth subtree that contains a specified root node. Hence, we revise and utilize the improved strong pruning technique that eliminates the restriction of containing a specified root node [25].

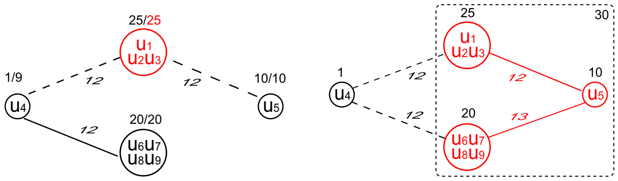
Given a minimum spanning tree T of the merged converted graph, it produces a subtree ST of T that maximizes its net worth $NW(ST)$ among all subtrees of T.

We next present the details of procedure strongPruning in Fig. 9 that given a minimum spanning tree T of a merged converted graph generated by procedure strongMerging, returns the maximum net worth subtree ST of T. It first reconstructs T by randomly choosing a root node (line 1). It then initializes the $nw(u)$ of each node u as its $p(u)$ (line 2). All nodes u are further updated in a bottom-up manner, *i.e.,* u is updated if and only if all its child nodes have already been updated. When it updates node u, each of the child nodes of u is processed independently: (a) If $nw(v) \geq w(u, v)$, it replaces $nw(u)$ with $nw(u) + nw(v) - w(u, v)$, and (b) otherwise, it removes edge (u, v) from T (lines 3–6). Finally, the subtree rooted at the node u_r with the maximum $nw(u_r)$ in the remaining T is returned as ST (lines 7-8).

In the following, we use an example to illustrate the process of procedure strongPruning.

Example 6: Consider the minimum spanning tree T of the merged converted graph $\vec{H}'$ in Fig. 8(b), obtained by removing the edge between nodes $\{u_5\}$ and $\{u_6, u_7, u_8, u_9\}$ from $\vec{H}'$. Note that here we simply use a set to denote a node in $\vec{H}'$. The optimal subtree ST produced by strongPruning is shown in Fig. 10(a), in which x and y of labels x/y denote the $p(u)$ and $nw(u)$ of nodes u, respectively, and dashed edges denote removed edges in the process.

Procedure strongPruning first randomly selects $\{u_1, u_2, u_3\}$ as the root of T. For each node u in T, its net worth $nw(u)$ is initialized with $p(u)$. The net worth $nw(\{u_5\})$ and $nw(\{u_6, u_7, u_8, u_9\})$ need not to be updated as both of them have no child nodes. The first modification is on node $\{u_4\}$, whose weight $nw(\{u_4\})$ is replaced by $nw(\{u_4\}) + nw(\{u_6, u_7, u_8, u_9\})-$ $w(\{u_4\},$ $\{u_6, u_7, u_8, u_9\})$. Then for node $\{u_1, u_2, u_3\}$, its attached two edges are removed since $nw(\{u_4\}) < w(\{u_1, u_2, u_3\}, \{u_4\})$ and $nw(\{u_5\}) < w(\{u_1, u_2, u_3\}, \{u_5\})$. Finally, the subtree rooted at $\{u_1, u_2, u_3\}$ is returned as ST, which is indeed $\{u_1, u_2, u_3\}$ itself, as its net worth $nw(\{u_1, u_2, u_3\})$ is the maximum one. □



(a) Subtree ST by strong pruning (b) Subtree ST' by bounded probing

Fig. 10. Examples for strong pruning and bounded probing

Based on [25], one can easily check the following result.

Corollary 6: *Given a minimum spanning tree $T(V, E)$, procedure* strongPruning *takes $O(|V|)$ time to find a subtree ST of T that maximizes its net worth $NW(ST)$ among all possible subtrees of T.* □

(3) Bounded probing. For the subtree ST found by procedure strongPruning, we propose a bounded probing technique to further maximize its net worth. We first illustrate this with an example below.

Example 7: Consider the subtree ST, *i.e.,* the single node $\{u_1, u_2, u_3\}$, found by procedure strongPruning in Fig. 10(a), where all nodes can no longer be merged. But, as shown in Fig. 10(b), after adding nodes $\{u_5\}$ and $\{u_6, u_7, u_8, u_9\}$ along the single path connecting them, we indeed have a subtree with a higher net worth $25 + 10 + 20 - 12 - 13 = 30$. □

We next introduce the main idea of bounded probing, which basically probes the set B of nodes in the merged converted graph $\vec{H}'$, but not in the subtree ST found by strongPruning, that can reach certain nodes in ST within r hops. We first greedily choose a path for each node in B that connects it with ST. Then we sort those paths that may increase the net worth of ST, and choose a set of disjoint paths. We extend ST with these disjoint paths, and repeat this process r times.

D. Algorithm for Aggregate Graphs

We now present the algorithm to compute the *Aggregate graph's Dense Subgraphs* (referred to as computeADS). A basic version of algorithm computeADS is shown in Fig. 11, which takes as input an aggregate graph $\widehat{H}$, and returns the dense subgraph of $\widehat{H}$ based on the three optimizations.

Algorithm computeADS first transforms aggregate graph $\widehat{H}$ into its converted graph $\vec{H}$ (line 1), and then produces a merged converted graph $\vec{H}'$ using the strong merging technique (line 2). A minimum spanning tree T of $\vec{H}'$ is computed (line 3), and is optimized to ST using the strong pruning technique (line 4). Tree ST is then optimized to ST' using the bounded probing technique (line 5), and ST' is further optimized to ST'' by computing a minimum spanning subtree of $\vec{H}$ that has the same set of nodes as ST' (line 6), and finally the subgraph of $\widehat{H}$ corresponding to the minimum spanning tree ST'' is returned (line 7).

We next show how algorithm computeADS finds the dense subgraph of an aggregate graph with an example below.

Example 8: (1) Consider the temporal graph $\mathbb{G}$ in Fig. 1(a) and its aggregate graph $\widehat{G}$ in Fig. 6(a) again. Algorithm

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

```
Algorithm computeADS
Input: Aggregate graph $\widehat{H}(V, E, f)$.
Output: Subgraph of $\widehat{H}$ with cohesive density as large as possible.
1. $\vec{H}(V', E', p, w)$ := convertAG($\widehat{H}$);
2. $\vec{H}'$ := strongMerging($\vec{H}$);
3. $T$ := a minimum spanning tree of $\vec{H}'$;
4. $ST$ := strongPrunning ($T$);
5. $ST'$ := boundedProbing ($ST$, $\vec{H}'$);
6. $ST''$ := a minimum spanning subtree of $\vec{H}$ using $ST'$;
   /*$ST''$ and $ST'$ have the same set of nodes */
7. return the subgraph of $\widehat{H}$ corresponding to $ST''$.
```

Fig. 11. Algorithm computeADS

computeADS first computes the converted graph $\vec{G}$ of $\widehat{G}$, shown in Fig. 6(b). In this case, as $\vec{G}$ is already a non-mergeable tree, computeADS simply uses the strong pruning technique to produce an optimized subtree, which contains the single node u_1 of $\vec{G}$. Finally, the subgraph with nodes v_1, v_2, v_3, v_4 in $\widehat{G}$ that corresponds to the node u_1 of $\vec{G}$ is returned.

(2) Consider the converted graph $\vec{H}$ shown in Fig. 8(a). Algorithm computeADS computes its merged converted graph $\vec{H}'$ shown in Fig. 8(b). The minimum spanning tree T of $\vec{H}'$ is then computed, as shown in Fig. 10(a). With strongPruning, it finds the optimal subtree of T, *i.e.,* the single $\{u_1, u_2, u_3\}$ in T. Using bounded probing, it finds another subtree ST', having nodes $\{u_1, u_2, u_3\}$, $\{u_5\}$ and $\{u_6, u_7, u_8, u_9\}$, with a higher net worth, as shown in Fig. 10(b). This tree is indeed the minimum spanning tree in the entire converted graph, and finally, the subgraph corresponding to ST' is returned. □

Recall that the strong merging technique directly merges two nodes using the edge between them. This might give worse results when there exist paths that are better than using the edge between them. Hence, algorithm computeADS further computes dense subgraphs without using strong merging (lines 1, 3–5 and 7 in Fig. 11), and finally returns the better solution of these two methods.

Time complexity analysis. Algorithm computeADS runs in $O(|V|+|E|+|E'|^2+(|E'|+|V'|)\cdot\log|V'|)$ time, in which $|E'|$ and $|V'|$ are the numbers of edges and nodes in the converted graph $\vec{H}$, respectively.

Observe the following. (1) Given an aggregate graph $\widehat{H}(V, E, f)$, it takes procedure convertAG $O(|V| + |E|)$ time to produce its converted graph $\vec{H}(V', E', p, w)$, as finding all connected components can be done in linear time [9] (line 1). (2) Procedure strongMerging can be done in $O(|E'|^2 + |E'|\log|V'|)$ time to generate the merged converted graph $\vec{H}'$ of $\vec{H}$ (line 2). (3) A minimum spanning tree can be computed in $O(|E'|\log|V'|)$ time [9] (lines 3, 6). (4) Strong pruning can be done in $O(|V'|)$ time [22] (line 4). Finally, (5) bounded probing takes $O(r^2 \cdot |E'| + r \cdot |V'|\log|V'|)$ time (line 5). Note that here $\vec{H}$ is typically much smaller than $\mathbb{G}$, $\vec{H}'$ is smaller than $\vec{H}$, and r is a small constant, *e.g.,* 3 and 4.

E. FIDES: The Complete Solution

We finally present the complete data-driven approach to FInding DEnse Subgraphs, referred to as FIDES, in temporal graphs, which combines algorithm computeADS above and the algorithms of identifying time intervals in Section III.

```
Algorithm FIDES
Input: Temporal graph $\mathbb{G}(V, E, F)$, positive integer $k$.
Output: Subgraph of $\mathbb{G}$ with cohesive density as large as possible.
1. Identifying $k/2$ time intervals using maxTInterval;
2. Identifying $k/2$ time intervals using minTInterval;
3. for each $[i, j]$ of the $k$ time intervals do
4.     compute the dense subgraph of $\mathbb{G}[i, j]$ using computeADS;
5. return the subgraph with the largest cohesive density.
```

Fig. 12. Algorithm FIDES: a data-driven approach

Algorithm FIDES is presented in Fig. 12, which takes as input a temporal graph $\mathbb{G}(V, E, F)$ and a positive integer k, and outputs the dense subgraph of $\mathbb{G}$ with the largest possible cohesive density. It first computes k time intervals using algorithms maxTInterval and minTInterval (lines 1-2). Among these k time intervals, it finds and returns the subgraph of $\mathbb{G}$ with the largest possible cohesive density, using algorithm computeADS (lines 3-5).

Time complexity analysis. By the complexity analyses of algorithms maxTInterval, minTInterval and computeADS, it is easy to know that given a temporal graph $\mathbb{G}(V, E, F)$ and a positive integer k, algorithm FIDES runs in $O((T + h^2) \cdot |E| + k \cdot (|V| + |E'|^2 + (|E'| + |V'|) \cdot \log|V'|))$ time.

Space complexity analysis. The space complexity of algorithm FIDES is $O(2 \cdot T \cdot |E|)$: (1) the storage of the temporal graph costs $O(T \cdot |E|)$ space, (2) we pre-compute $AF(e, t)$ for each edge e and timestamp t, which costs another $O(T \cdot |E|)$ space, and (3) each step of computeADS is basically based on the converted graph $\vec{H}$, with the space complexity being the size of the converted graph, *i.e.,* $O(|V'| + |E'|)$.

Note that here (1) h is the number of local maxima or minima, (2) T is the total number of snapshots, and (3) $|E'|$ and $|V'|$ are the largest numbers of edges and nodes in all converted graphs $\vec{H}$ in algorithm computeADS, which are typically much smaller than $|E|$ and $|V|$, respectively.

V. EXPERIMENTAL STUDY

Using both real-life and synthetic data, we conduct an extensive experimental study of our data-driven approach FIDES to finding dense subgraphs in large temporal networks, compared with the state of the art method MEDEN [7].

A. Experimental Settings

We first introduce the settings of our experimental study.

Datasets. We chose two datasets to test our approach.

(1) BJDATA is a real-life dataset that records the dynamic traffic condition of the road network in Beijing. Its road traffic conditions (+2: congestion, +1: slow, and −1: fast) were collected using Taxies equipped with GPS sensors, and were updated every 5 minutes. Here we consider day level temporal data with 289 snapshots in total. Hence, BJDATA is very large, and has $23, 724, 877$ nodes and $31, 280, 782$ edges.

(2) SYNDATA is produced by the synthetic data generator developed in [7]. Using random graphs as the underlying graph structure, the generator first produces a temporal graph with n nodes, m edges and T snapshots. Initially, all edges are assigned with negative weights. The generator activates a seed edge at random by assigning it a positive edge weight. After this, its neighboring edges are activated based on a probability

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

np_r. An activated edge also has a probability tp_r, to activate its copy in the next snapshot. Later activated edges will perform the same activation process, with decayed np_r and tp_r. The process is repeated until the graph reaches a fixed activation density ad_r, the percentage of activated edges in all snapshots. Rates np_r, tp_r and ad_r are fixed to 0.3, 0.9 and 0.3 by default, respectively, and the number of edges m is fixed to $2 \cdot n$.

Algorithms and implementation. Algorithms were all implemented with Java, including the state of the art algorithm MEDEN and the synthetic data generator [7] that are available at http://www.cs.ucsb.edu/d̃bl/software.php.

All experiments were run on a PC with 2 Intel Xeon E5–2630 2.4GHz CPUs and 64 GB of memory. The usage of virtual memory was forbidden in all our tests. When quantity measures are evaluated, the test was repeated over 5 times and the average is reported here.

B. Experimental Results

We tested the evolving convergence phenomenon proposed in Section III, and the effectiveness and efficiency of our data-driven approach FIDES vs. MEDEN using BJDATA and SYNDATA. We next present our findings.

Exp-1. Verification of the evolving convergence phenomenon. In the first set of tests, we show the rational of the evolving convergence phenomenon, which justifies the way how we identify the top k time intervals.

Given a temporal graph $\mathbb{G}(V, E, F)$, we define a metric $p_{EC} = \frac{\sum_{t=2}^{T} \max(|E^{\geq}(t)|, |E^{\leq}(t)|)}{|E|(T-1)}$ (the proportion of edges that satisfy the evolving convergence phenomenon) to measure to what degree the temporal graph $\mathbb{G}$ obeys the phenomenon, in which $|E^{\geq}(t)|$ and $|E^{\leq}(t)|$ ($t \in [2, T]$) denote the corresponding numbers of edges $e \in E$ such that $F^t(e) \geq F^{t-1}(e)$ and $F^t(e) \leq F^{t-1}(e)$, respectively.

The p_{EC} are 96% on BJDATA and 90% on average on all tested SYNDATA, respectively, which justifies our observation of the evolving convergence phenomenon.

Exp-2. Algorithms computeADS **vs.** topDown. In the second set of tests, we compare the effectiveness and efficiency of computeADS with topDown, both of which compute the dense subgraphs on aggregate graphs for given time intervals, and are called by FIDES and MEDEN, respectively.

Exp-2.1. To evaluate the impacts of the number T_{ti} of snapshots in the time intervals of aggregate graphs, we varied T_{ti} from 50 to 289 for BJDATA and from 200 to $2,000$ for SYNDATA, respectively. We used the entire BJDATA, and fixed SYNDATA with $n = 100,000$, $T = 2,000$ and $ad_r = 0.3$. For fairness, we report the average result of aggregate graphs with 20 distinct time intervals for each T_{ti}, except the largest T_{ti} for BJDATA, shown in Fig. 13.

When varying T_{ti}, the cohesive density scores of the subgraphs found by both algorithms increase with the increment of T_{ti}, as the data has temporal contiguity of positive weight edges. Further, those found by computeADS are consistently better than topDown on both BJDATA ($+0.28\%$ on average) and SYNDATA ($+0.04\%$ on average).

The running time of both algorithms is insensitive to the number T_{ti} of snapshots in the time intervals of aggregate graphs, as the aggregate graphs are basically the same, in terms of both their sizes and structures. But, computeADS is much more efficient than topDown on both datasets, and is around 67 and 22 times faster than topDown on BJDATA and SYNDATA, respectively. This is because computeADS reduces the sizes of aggregate graphs using converted graphs and the strong merging technique, which speeds up the computation.

Exp-2.2. To evaluate the impacts of the graph sizes, we varied n from $50,000$ to $400,000$ on SYNDATA, while fixed $T = 2,000$ and $ad_r = 0.3$. For fairness, we used the average result of 100 aggregate graphs by randomly generating 100 time intervals for each graph size. The results are reported in Figs. 14(a) & 14(b). We did not report topDown on graphs with $n \geq 150,000$, as it ran out of memory.

When varying n, the cohesive density scores of the subgraphs found by both algorithms obviously increase with the increment of n, and computeADS is consistently better ($+0.04\%$ on average) than topDown when $n \leq 100,000$.

When varying n, the running time of both algorithms increases with the increment of n. Algorithm computeADS is consistently much faster than topDown, and is 15 and 24 times faster when $n = 50,000$ and $100,000$, respectively.

Exp-2.3. To evaluate the impacts of the activation density ad_r, we varied ad_r from 0.05 to 0.35 on SYNDATA, while fixed $n = 100,000$ and $T = 2,000$. Due to the way that the synthetic generator works, it is already relatively dense in terms of positive weight edges even when ad_r is 0.35. Note that ad_r was fixed to 0.1 in [7]. For fairness, we also used the same strategy as *Exp-2.2* to use the average result of 100 aggregate graphs for each ad_r, which are reported in Figs. 14(c) & 14(d).

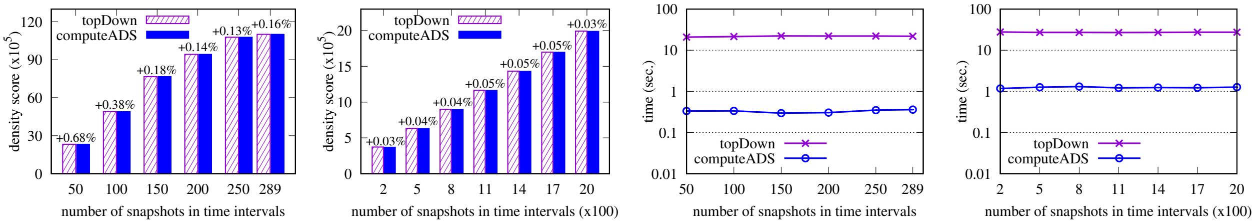
When varying ad_r, the cohesive density scores of the subgraphs found by both algorithms obviously increase with the increment of ad_r. The subgraphs found by computeADS are better ($+4.30\%$ on average) than topDown when $ad_r \leq 0.3$, and are the same in terms of cohesive density when $ad_r = 0.35$. Algorithm computeADS performs significantly better than topDown when $ad_r \leq 0.2$, which is due to our three well-tuning optimization techniques.

When varying ad_r, the running time of computeADS decreases, while the one of topDown increases, especially when varying ad_r from 0.25 to 0.35. This is because (a) there are more edges with positive weights for larger ad_r, and (b) procedure convertAG and the strong merging technique become more effective on reducing the sizes of aggregate graphs when there are more positive weight edges. Indeed, computeADS is 20, 25 and 36 times faster than topDown when ad_r is ≤ 0.25, 0.3 and 0.35, respectively, in our tests.

Exp-3. Algorithms FIDES **vs.** MEDEN. In the third set of tests, we compare the effectiveness and efficiency of our approach FIDES with the state of the art method MEDEN. In addition to the three factors evaluated in *Exp-2*, we further test the impacts of the number k of time intervals used in FIDES. By default, k is set to 10. Due to the superiority of computeADS over topDown as shown in *Exp-2*, FIDES may produce denser subgraphs than MEDEN, despite the limited number of verified time intervals.

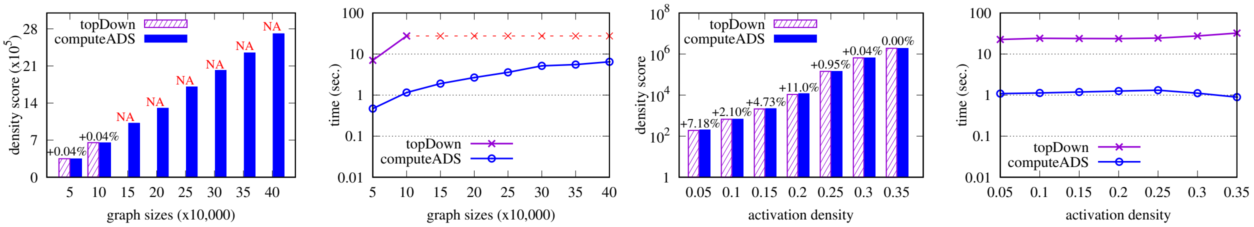
Exp-3.1. To evaluate the impacts of the number T of snapshots of temporal graphs, we varied T from 50 to 289 for BJDATA

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.



(a) BJData (b) SYNData (c) BJData (d) SYNData

Fig. 13. Varying the number of snapshots in time intervals: computeADS vs. topDown



(a) SYNData (b) SYNData (c) SYNData (d) SYNData

Fig. 14. Varying the graph size and activation density: computeADS vs. topDown

and from 200 to 2,000 for SYNData, respectively. We fixed $k = 10$, and used the same setting as *Exp-2.1* for BJData and SYNData. The results are reported in Fig. 15.

When varying T, the cohesive density scores of the subgraphs found by both algorithms increase with the increment of T. Moreover, the dense subgraph found by FIDES are consistently better than MEDEN on both BJData (+0.28% on average) and SYNData (+0.04% on average), in our tests,

When varying T, the running time of both algorithms obviously increases with the increment of T. Moreover, FIDES is consistently faster than MEDEN, and is around 2,980 and 1,079 times faster than MEDEN on BJData and SYNData on average, respectively, in our tests.

Exp-3.2. To evaluate the impacts of the parameter k, we varied k from 2 to 22. We used the same setting as *Exp-2.1* for T, n and ad_r. Algorithm MEDEN uses k time intervals to estimate a lower bound for pruning (line 4 in Fig. 2), and k has impacts on the running time of MEDEN, but not the quality of the dense subgraphs found. The results are reported in Fig. 16. We simply plotted red markers $*$ in Fig. 16(d) when MEDEN could not finish the tests in 2 days.

When varying k, the dense subgraphs found by both algorithms are insensitive to k when k is no less than 10. The dense subgraphs found by FIDES are (+0.15%, +0.04%) better than MEDEN on (BJData, SYNData) when $k \geq 10$.

When varying k, the running time of MEDEN decreases, while the one of FIDES increases, with the increment of k. Algorithm MEDEN could not finish the test in 2 days on SYNData as it used a non-effective lower bound for $k = 2$, and its running time becomes stable when k is around 18 on both datasets, which only differs 1.72% on BJData and 0.06% on SYNData when increasing k from 18 to 22. Moreover, FIDES is consistently faster than MEDEN. Indeed, FIDES is 19,655 and 3,036 times faster than MEDEN, on BJData and SYNData on average, respectively, in our tests.

Exp-3.3. To evaluate the impacts of the graph sizes n, we used the same setting as *Exp-2.2* and fixed $k = 10$. We did not report MEDEN on graphs with size 150,000 or larger, as it ran out of memory, and could not finish the tests. The results are reported in Figs. 17(a) & 17(b).

When varying n, the cohesive density scores of subgraphs found by both algorithms increase with the increment of n. The dense subgraphs found by FIDES are better (+0.05% on average) than MEDEN on graphs with size no more than 100,000, in our tests.

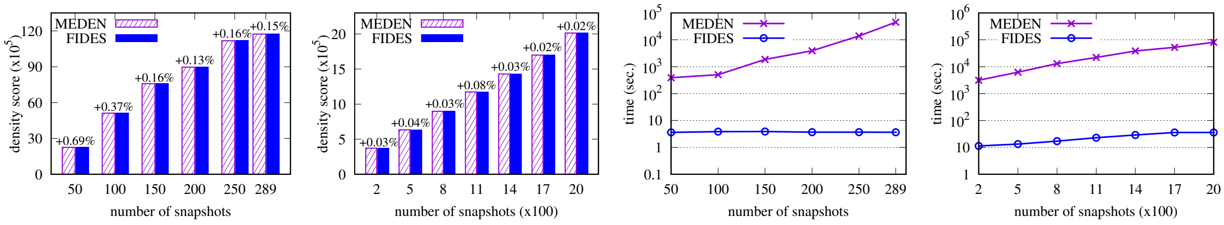
When varying n, the running time of both algorithms obviously increases with the increment of n. Moreover, FIDES is consistently faster than MEDEN, and is 2,519 times faster on graphs with size no more than 100,000 on average, in our tests. In fact, FIDES could finish in 141.2 seconds when the graph size reaches 400,000, while it already took MEDEN 23,180 seconds on small graphs with 50,000 nodes only.

Exp-3.4. To evaluate the impacts of the activation density ad_r, we used the same setting as *Exp-2.3* and fixed $k = 10$. The results are reported in Figs. 17(c) & 17(d). Note that MEDEN ran out of memory when ad_r was 0.05 or 0.25, as in these cases there were too many unpruned time intervals to verify, and too much space to store the corresponding aggregate graphs.

When varying ad_r, the cohesive density scores of subgraphs found by both algorithms increase with the increment of ad_r. The dense subgraphs found by FIDES are slightly worse (−0.16% on average) than MEDEN in our tests. More specifically, those found by FIDES are slightly worse (−0.29% on average) than MEDEN when $ad_r \leq 0.2$, and are no worse than MEDEN when $ad_r \geq 0.3$, in our tests, respectively. This result further shows the effectiveness of the top–k time intervals, especially on graphs with larger ad_r.

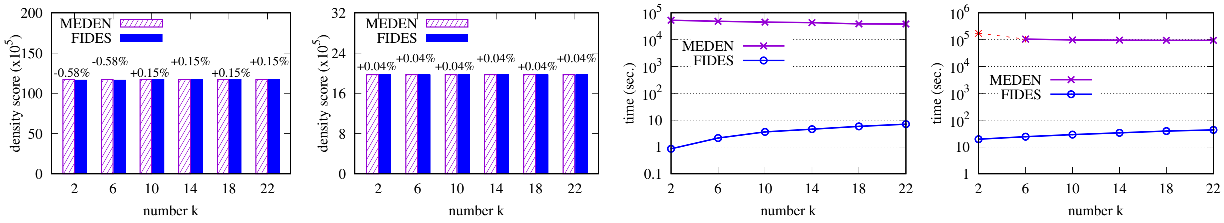
When varying ad_r, the running time of MEDEN first increases and then decreases. This is due to the impacts of the pruning technique of MEDEN. Note that here ad_r = 0.3 is a turning point for MEDEN, as it happens that the estimated bounds of MEDEN are not very effective when ad_r = 0.3. As we can see, FIDES is very robust to ad_r. Further, FIDES is consistently faster than MEDEN, and is about 1,265 times faster on average, in our tests.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.



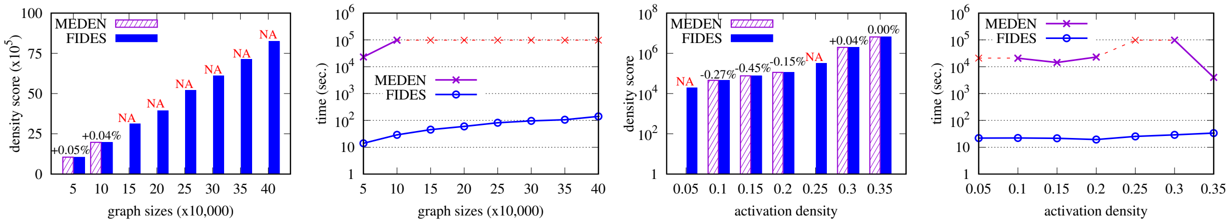
(a) BJDATA (b) SYNDATA (c) BJDATA (d) SYNDATA

Fig. 15. Varying the number of snapshots: FIDES vs. MEDEN



(a) BJDATA (b) SYNDATA (c) BJDATA (d) SYNDATA

Fig. 16. Varying the number k: FIDES vs. MEDEN



(a) SYNDATA (b) SYNDATA (c) SYNDATA (d) SYNDATA

Fig. 17. Varying the graph size and activation density: FIDES vs. MEDEN

Exp-4. Closeness to optimality. (1) One may want to know the closeness to the optimal solutions of computeADS and FIDES. However, as shown by Proposition 1, computing the optimal dense subgraphs is infeasible even for aggregate graphs. Hence, we chose the four groups (K, P, C, D) of 114 benchmark small graphs with known optimal solutions [24]. The cohesive density scores of subgraphs found by computeADS are (92%, 94%, 92%, 90%) of the optima of groups (K, P, C, D), respectively, on average. In contrast, topDown obtains an average performance of (83%, 87%, 87%, 84%) on the four groups, respectively. (2) Roughly speaking, the solutions of FIDES are around 85% of the optima on average, as FIDES is comparable to MEDEN, which achieves a performance of 85% of the optima [7].

Summary. From these tests we find the following.

(1) The evolving convergence phenomenon is quite common. Indeed, there are about 96% and 90% of edges satisfying the phenomenon on BJDATA and SYNDATA, respectively.

(2) The quality of the dense subgraphs found by computeADS is consistently better than those by topDown on both BJDATA (+0.28% on average) and SYNDATA (+0.04% on average). Further, computeADS is much faster than topDown: 67 times and 15 times on average on BJDATA and SYNDATA, respectively. Finally, topDown already ran out of memory for graphs with $150,000$ nodes and $2,000$ snapshots.

(3) The quality of the dense subgraphs found by FIDES is comparable to those by MEDEN: better (+0.28% on average) on BJDATA and slightly worse (−0.16% on average) on SYNDATA. Further, FIDES is $2,980$ and $1,079$ times faster than MEDEN on average on BJDATA and SYNDATA, respectively. Finally, MEDEN already ran out of memory for graphs with $150,000$ nodes and $2,000$ snapshots.

(4) The three characteristics of time intervals (*i.e.,* Proposition 2, Fact 1 and Fact 2 in Section III) together assure a pretty good estimation of the time intervals involved with dense subgraphs. Indeed, a small number of intervals, *e.g.,* 10, already suffice for FIDES to find a good solution.

VI. RELATED WORK

Dense subgraphs in static networks. Dense subgraphs have been widely studied, and are a general concept. The concrete semantics highly depend on the studied problems and applications, such as cohesive subgraphs like maximal cliques, n-clique, k-core and n-clan [31], the prize collecting Steiner tree [13], [22], and densities defined in terms of the numbers or weights of edges and nodes [3], [4], [15], [16], [23].

Our work adopts the strong pruning technique introduced in [22] for finding a better subgraph in aggregate graphs, by building the connection between finding the subgraph of an aggregate graph with the highest cohesive density and finding the maximum net worth subtree [22].

Dense subgraphs in dynamic networks. Dense subgraphs have also been recently investigated in temporal networks under various terms, such as anomalies [6], [8], heavy subgraphs [7], dense subgraphs [5], [10] and network processes [27]. However, they typically refer to connected subgraphs with higher scores, defined in terms of the weights of edges and

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.

nodes in a continuous time interval. Our work adopts the definition of dense subgraphs in [7], and is different from [5], [6], [8], [10], [27]. Further, the study in [5], [10] focuses on dynamic graphs with node and/or edge updates, and, hence, is different from our work.

Close to our work is [7] that proposed and studied the FDS problem. We develop a data-driven solution, totally different from the filter and verification method in [7]. Further, the connection between the FDS and NWM problems has never been employed in the algorithm of [7], not to mention the approximation hardness result of the FDS problem. Indeed, data-driven solutions using hidden statistics of data also shed light on large graph processing.

Other works in dynamic networks. Temporal network analysis has recently attracted more and more attentions [2], [19], [35], such as temporal shortest paths [14], [18], [32], temporal minimum spanning trees [21], incremental graph pattern matching [12], graph stream analysis [33] and continuous aggregate queries [26]. Different from these, we study dense temporal subgraphs in this work.

VII. Conclusions

We have proposed FIDES, a data-driven approach employing hidden data statistics to finding dense subgraphs in large temporal networks. First, we have employed the data characteristics to effectively identify k time intervals from a total of $T * (T + 1)/2$ ones, in which T is the number of snapshots and k is typically much smaller than T. Second, we have developed better algorithm heuristics to solve the problem. Finally, we have experimentally verified that FIDES is much more scalable than the state of the art method MEDEN [7], while the quality of the dense subgraphs found by FIDES is comparable to MEDEN.

Several issues need further study. We are to apply our approach to general temporal graphs with node and edge updates, to develop incremental and distributed algorithms to provide further scalability on large temporal networks, and to extend our techniques to find top–k dense temporal subgraphs.

Acknowledgment

This work is supported in part by NSFC U1636210, 973 Program 2014CB340300, NSFC 61421003 & 61322207, Special Funds of Beijing Municipal Science & Technology Commission, and MSRA Collaborative Research Program.

References

[1] Baidu real-time road traffic status product. *http://map.baidu.com/fwmap/zt/traffic/index.html?city*.

[2] C. C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *ACM CSUR*, 47(1):10:1–10:36, 2014.

[3] R. Andersen. A local algorithm for finding dense subgraphs. *ACM Transactions on Algorithms*, 6(4), 2010.

[4] O. D. Balalau, F. Bonchi, T. H. Chan, F. Gullo, and M. Sozio. Finding subgraphs with maximum total density and limited overlap. In *WSDM*, 2015.

[5] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *STOC*, 2015.

[6] P. Bogdanov, C. Faloutsos, M. Mongiovì, E. E. Papalexakis, R. Ranca, and A. K. Singh. Netspot: Spotting significant anomalous regions on dynamic networks. In *SDM*, 2013.

[7] P. Bogdanov, M. Mongiovì, and A. K. Singh. Mining heavy subgraphs in time-evolving networks. In *ICDM*, 2011.

[8] J. Chan, J. Bailey, C. Leckie, and M. Houle. ciForager: Incrementally discovering regions of correlated change in evolving graphs. *TKDD*, 6(3):11, 2012.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[10] A. Epasto, S. Lattanzi, and M. Sozio. Efficient densest subgraph computation in evolving graphs. In *WWW*, 2015.

[11] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1), 2010.

[12] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *TODS*, 38(3):18, 2013.

[13] J. Feigenbaum, C. H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *J. Comput. Syst. Sci.*, 63(1):21–41, 2001.

[14] L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, 68(4):1075–1097, 2014.

[15] A. Gajewar and A. D. Sarma. Multi-skill collaborative teams based on densest subgraphs. In *SDM*, 2012.

[16] D. Gibson, R. Kumar, and A. Tomkins. Discovering large dense subgraphs in massive graphs. In *VLDB*, 2005.

[17] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *Eurosys*, 2014.

[18] M. S. Hassan, W. G. Aref, and A. M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In *SIGMOD*, 2016.

[19] P. Holme and J. Saramäki. Temporal networks. *Physics Reports*, 519(3):97 – 125, 2012.

[20] H. Huang, J. Song, X. Lin, S. Ma, and J. Huai. TGraph: A temporal graph data management system. In *CIKM*, 2016.

[21] S. Huang, A. W. Fu, and R. Liu. Minimum spanning trees in temporal graphs. In *SIGMOD*, 2015.

[22] D. S. Johnson, M. Minkoff, and S. Phillips. The prize collecting steiner tree problem: theory and practice. In *SODA*, 2000.

[23] S. Khuller and B. Saha. On finding dense subgraphs. In *ICALP*, 2009.

[24] I. Ljubic, R. Weiskircher, U. Pferschy, G. W. Klau, P. Mutzel, and M. Fischetti. An algorithmic framework for the exact solution of the prize-collecting steiner tree problem. *Math. Program.*, 105(2-3):427–449, 2006.

[25] M. Minkoff. The prize collecting steiner tree problem. Master's thesis, MIT, 2000.

[26] J. Mondal and A. Deshpande. EAGr: supporting continuous ego-centric aggregate queries over large dynamic graphs. In *SIGMOD*, 2014.

[27] M. Mongiovì, P. Bogdanov, and A. K. Singh. Mining evolving network processes. In *ICDM*, 2013.

[28] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *ICDE*, 2016.

[29] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–391, 1983.

[30] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[31] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[32] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.

[33] W. Yu, C. C. Aggarwal, S. Ma, and H. Wang. On anomalous hotspot discovery in graph streams. In *ICDM*, 2013.

[34] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: Concepts, methodologies, and applications. *ACM TIST*, 5(3):38:1–38:55, 2014.

[35] B. Zong, X. Xiao, Z. Li, Z. Wu, Z. Qian, X. Yan, A. K. Singh, and G. Jiang. Behavior query discovery in system-generated temporal graphs. *PVLDB*, 9(4):240–251, 2015.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 26,2020 at 04:04:42 UTC from IEEE Xplore. Restrictions apply.