HAL
open science

Span-core Decomposition for Temporal Networks: Algorithms and Applications

Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, Francesco Gullo

To cite this version:

Edoardo Galimberti, Martino Ciaperoni, Alain Barrat, Francesco Bonchi, Ciro Cattuto, et al.. Span-core Decomposition for Temporal Networks: Algorithms and Applications. ACM Transactions on Knowledge Discovery from Data (TKDD), 2020, 15 (1), pp.2. 10.1145/3418226 . hal-03047191

HAL Id: hal-03047191

https://hal.science/hal-03047191

Submitted on 16 Dec 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Span-core Decomposition for Temporal Networks: Algorithms and Applications

Edoardo Galimberti, Martino Ciaperoni, Alain Barrat,
Francesco Bonchi, Ciro Cattuto, Francesco Gullo

December 16, 2020

Abstract

When analyzing temporal networks, a fundamental task is the identification of dense structures (i.e., groups of vertices that exhibit a large number of links), together with their temporal span (i.e., the period of time for which the high density holds). In this paper we tackle this task by introducing a notion of temporal core decomposition where each core is associated with two quantities, its coreness, which quantifies how densely it is connected, and its span, which is a temporal interval: we call such cores *span-cores*.

For a temporal network defined on a discrete temporal domain T, the total number of time intervals included in T is quadratic in $|T|$, so that the total number of span-cores is potentially quadratic in $|T|$ as well. Our first main contribution is an algorithm that, by exploiting containment properties among span-cores, computes all the span-cores efficiently. Then, we focus on the problem of finding only the *maximal span-cores*, i.e., span-cores that are not dominated by any other span-core by both their coreness property and their span. We devise a very efficient algorithm that exploits theoretical findings on the maximality condition to directly extract the maximal ones without computing all span-cores.

Finally, as a third contribution, we introduce the problem of *temporal community search*, where a set of query vertices is given as input, and the goal is to find a set of densely-connected subgraphs containing the query vertices and covering the whole underlying temporal domain T. We derive a connection between this problem and the problem of finding (maximal) span-cores. Based on this connection, we show how temporal community search can be solved in polynomial-time via dynamic programming, and how the maximal span-cores can be profitably exploited to significantly speed-up the basic algorithm.

We provide an extensive experimentation on several real-world temporal networks of widely different origins and characteristics. Our results confirm the efficiency and scalability of the proposed methods. Moreover, we showcase our techniques in a number of applications on temporal networks describing human face-to-face interactions in various settings. Our main findings in this regard highlight the relevance of the notions

of (maximal) span-core and temporal community search in analyzing social dynamics, detecting/correcting anomalies in the data, and graph-embedding-based network classification.

1 Introduction

A temporal network[1] is a representation of entities (vertices), their relations (links), and how these relations are established/broken over time. Notice that here we will consider discrete times, i.e., the temporal networks can be represented as a time-ordered series of snapshots (instantaneous graphs). Extracting dense structures (i.e., groups of vertices exhibiting a large number of links with each other), together with their temporal span (i.e., the period of time for which the high density is observed) is a key mining primitive to characterize such temporal networks and extract relevant structures. This type of pattern enables fine-grain analysis of the network dynamics and can be a building block towards more complex tasks and applications, such as finding temporally recurring subgraphs or anomalously dense ones. For instance, they can help in studying contact networks among individuals to quantify the transmission opportunities of respiratory infections in a population and uncover situations where the risk of transmission is higher, with the goal of designing mitigation strategies [38]. Anomalously dense temporal patterns among entities in a co-occurrence graph (e.g., extracted from the Twitter stream) have also been used to identify events and buzzing stories in real time [5, 15]. Another example concerns scientific collaboration and citation networks, where these patterns can help understand the dynamics of collaboration in successful professional teams, study the evolution of scientific topics, and detect emerging technologies [28].

In this paper we adopt as a measure of density of a pattern the *minimum degree* holding among the vertices in the subgraph during the pattern's span. The problem of extracting *all* these patterns is tackled by introducing a notion of *temporal core decomposition* in which each core is associated with its *span*, i.e., an interval of *contiguous timestamps*, for which the coreness property holds. We term such a notion of temporal core *span-core*.

Moreover, in several application scenarios it is typically required to identify only those dense patterns that contain a given set of query vertices. We therefore introduce the problem of *temporal community search*, whose goal is to find a set of cohesive temporal subgraphs containing the input query vertices and covering the whole temporal domain.

To the best of our knowledge, the problems of (efficient) *span-core* computation and temporal community search have never been studied so far.

1.1 Challenges and contributions

As the number of possible time intervals is quadratic in the size of the input temporal domain T, the total number of span-cores is, in the worst case,

[1] We use "network" and "graph" interchangeably throughout the paper.

quadratic in T too. The naive method to find all *span-cores*, which would be to operate a core decomposition for each of these time intervals, would therefore be very time-consuming. This is a major challenge that we tackle by deriving containment properties between span-cores and by exploiting them to devise an algorithm for computing all the *span-cores* that is significantly more efficient than the naïve exhaustive method.

We then shift our attention to the problem of finding only the *maximal span-cores*, defined as the span-cores that are not dominated by any other span-core by both the coreness property and the span. A straightforward way of approaching this problem is to filter out non-maximal span-cores during the execution of an algorithm for computing the whole span-core decomposition. However, as the maximal ones are usually much less numerous than the overall span-cores, it would be desirable to have a method that effectively exploits the maximality property and extracts maximal span-cores directly, without computing the complete decomposition. The design of an algorithm of this kind is an interesting challenge, as it contrasts with the intrinsic conceptual properties of core decomposition, based on which a core of order k can be efficiently computed from the core of order $k-1$, of which it is a subset. For this reason, at first glance, the computation of the core of the highest order would seem as hard as computing the overall core decomposition. Instead, in this work we derive a number of theoretical properties about the relationship among span-cores of different temporal intervals and, based on these findings, we show how such a challenging goal may be achieved.

Finally, we focus on the problem of *community search in temporal networks*. Community search has been extensively studied in static graphs. It requires to find a subgraph containing a given set of query vertices and maximizing a certain density measure [32, 46]. Here, we propose a formulation of the community-search problem in temporal networks as follows: given a set Q of query vertices, and a positive integer h, find a segmentation of the underlying temporal domain in h segments $\{\Delta_i\}_{i=1}^{h}$ and a subgraph S_i for every identified segment Δ_i such that each S_i contains the query vertices Q and the total density of the subgraphs is maximized. Following the bulk of the literature in community search on static networks, in our definition of temporal community search we adopt the minimum degree as a density measure.

We show that, with some manipulations, temporal community search can be reformulated as an instance of the popular *sequence segmentation* problem, which asks for partitioning a sequence of numerical values into h segments so as to minimize the sum of the penalties (according to some penalty function) on the identified segments [10]. Therefore, the classical dynamic-programming algorithm for sequence segmentation by Bellman [10] can be easily adapted to solve temporal community search in polynomial time.

A criticality of this approach is that a naïve adaptation of the Bellman's algorithm takes quadratic time in the size of the input temporal domain T. As a major contribution in this regard, we prove that the set of maximal span-cores provide a sound and complete basis to still have an optimal solution to temporal community search, while at the same time leading to a significant speed-up with

respect to the naïve method. In fact, let $T^* \subseteq T$ be the subset of timestamps that are covered by the span of at least one maximal span-core, together with the timestamps that immediately precede or succeed any of such spans. We show that considering T^* (instead of T) in the (adaptation of the) Bellman's algorithm is sufficient to optimally solve the underlying temporal-community-search problem instance. As, typically, $|T^*| \ll |T|$, this finding guarantees a considerable improvement in efficiency (as confirmed by our experiments).

A further challenge in our temporal-community-search problem is a typical one in community-search formulations based on minimum degree, namely, that the output subgraphs are typically large in size. We tackle this challenge by devising a method to reduce the size of the output subgraphs without affecting optimality. The proposed method is inspired by the one devised by Barbieri *et al.* [7] for the problem of minimum community search (in static graphs).

To summarize, the main contributions of this paper are as follows:

- We introduce the notion of span-core decomposition and maximal span-core in temporal networks, characterizing structure and size of the search space and providing important containment properties (Section 3).
- We devise an algorithm for computing all span-cores that exploits the aforementioned containment properties and is orders of magnitude faster than a naïve method based on traditional core decomposition (Section 4).
- We study the problem of finding only the maximal span-cores. We derive several theoretical findings about the relationship between maximal span-cores and exploit these findings to devise an algorithm that is more efficient than computing all span-cores and discarding the non-maximal ones (Section 5).
- We introduce the problem of temporal community search and show how it can be solved in polynomial time via dynamic programming. We prove an important connection between temporal community search and maximal span-cores, which allows us to devise an algorithm that is considerably more efficient than the naïve dynamic-programming one. We also propose a method to achieve the critical challenge of having too large communities as output (Section 6).
- We provide a comprehensive experimentation on several real-world temporal networks, with millions of vertices, tens of millions of edges, and hundreds of timestamps, which attests efficiency and scalability of our methods (Section 7).
- We present applications on face-to-face interaction networks that illustrate the relevance of the notions of (maximal) span-core and temporal community search in real-life analyses and applications (Section 8).

The next section provides an overview of the related literature, while Section 9 discusses future work and concludes the paper.

An abridged version of this work, covering Sections 4 and 5, together with the corresponding experiments (i.e., parts of Sections 7 and 8), was presented in [33].

Reproducibility. For the sake of reproducibility, all our code and some of the datasets used in this paper are available at github.com/egalimberti/span_cores.

2 Background and related work

2.1 Core decomposition

Given a simple (static) graph $G = (V, E)$, let $d(S, u)$ denote the degree of vertex $u \in V$ in the subgraph induced by vertex set $S \subseteq V$, i.e., $d(S, u) = |\{v \in S \mid (u, v) \in E\}|$. The notions of *k-core* and *core decomposition* are defined as follows:

Definition 1 (k-core and core decomposition [62]) *The* k-core *(or core of order* k*) of* G *is a* maximal *set of vertices* $C_k \subseteq V$ *such that* $\forall u \in C_k : d(C_k, u) \geq k$*. The set of all* k*-cores* $V = C_0 \supseteq C_1 \supseteq \cdots \supseteq C_{k^*}$ *(*$k^* = \arg\max_k C_k \neq \emptyset$*) is the* core decomposition *of* G*.*

Core decomposition can be computed in linear time by iteratively removing the smallest-degree vertex and setting its core number as equal to its degree at the time of removal [8]. Among the many definitions of dense structures, core decomposition is particularly appealing as, among others, it is fast to compute, and can speed-up/approximate dense-subgraph extraction according to various other definitions. For instance, core decomposition allows for finding cliques more efficiently [27], as a k-clique is contained into a $(k-1)$-core, which can be significantly smaller than the original graph. Moreover, core decomposition is at the basis of approximation algorithms for the densest-(at-least-k-)subgraph problem [51, 3], and betweenness centrality [42]. Core decomposition has also been recognized as an important tool to analyze and visualize complex networks [9, 2] in several domains, e.g., bioinformatics [6, 81], software engineering [84], and social networks [49, 36]. It has been studied under various settings, such as distributed [64], streaming/maintenance [72, 55], and disk-based [20], and generalized to various types of static graphs, such as uncertain [16], directed [39], weighted [35, 24], bipartite graphs [56], or including attributes on the nodes [83]. For a comprehensive survey about theory, algorithms, and applications of core decomposition we refer to [59].

Two types of extension of the core-decomposition bear some relation to our work. First, core decomposition in *multilayer networks* has been studied in [34]. As a core is allowed in this setting to extend on any subset of layers, the total number of cores is intrinsically exponential in the number of layers. Although temporal networks can be seen as a special case of multilayer networks (where each timestamp is interpreted as a layer), there is a fundamental difference: in a temporal network, the "layers" are ordered and the sequentiality of timestamps represents an important structural constraint. Here, we are interested in cores

that span a temporal interval, and not simply any subset of (potentially non-contiguous) timestamps. As a consequence, the search space and the number of cores are no longer exponential as in the multilayer case. Second, Wu *et al.* [80] have proposed a core decomposition on temporal networks, which however does not take any kind of temporal constraint into account. They define indeed the (k, h)-core as the largest subgraph in which every vertex has at least k neighbors and there are at least h temporal edges between the vertex and its neighbors, without any restriction on when these h edges occur: the sequentiality of connections is not taken into account and non-contiguous timestamps can support the same core. In fact, the (k, h)-core decomposition can be seen as a kind of weighted static core decomposition on the weighted static network resulting from the aggregation of the temporal network. In contrast, our temporal cores have each a clear temporal collocation and continuous spans, so that our definition includes temporality in an explicit way and cannot be reduced to the one of Wu *et al.*'s. As we will see in Section 8, associating a temporal collocation to each core is important in applications.

2.2 Patterns in temporal networks

A number of works on extracting dense patterns from a temporal network focus on the well-established notion of *densest subgraph*, i.e., a subgraph maximizing the average-degree density. Jethava and Beerenwinkel [48] consider as input a set of graphs sharing the same vertex set, which can thus also be interpreted as a temporal network. On such an input they study the *densest common subgraph* problem, i.e., the problem of finding a subgraph maximizing the minimum average degree over all graphs (timestamps), and devise a linear-programming formulation and a greedy heuristic algorithm for it. Further (mostly theoretical) advancements to the densest-common-subgraph problem have been provided by Reinthal *et al.* [66] and Charikar *et al.* [19]. Semertzidis *et al.* [73] instead introduce two more variants of the problem, where the goal is to maximize the average average degree and the minimum minimum degree, respectively. They show that the average-average variant easily reduces to the traditional densest-subgraph problem, and that the minimum-minimum variant can be exactly solved by a simple adaptation of the classic algorithm for core decomposition.

Complementary works focus on variants of the densest-subgraph-discovery problem. Rozenshtein *et al.* study the problem of discovering dense temporal subgraphs whose edges occur in short time intervals considering the exact timestamp of the occurrences [68], and the problem of partitioning the timeline of a temporal network into non-overlapping intervals, such that the intervals span subgraphs with maximum total density [67]. Epasto *et al.* [25] deal with the problem of maintaining the densest subgraph in a dynamic setting.

Attention in the literature has also been devoted to densities other than the average degree. The notion of Δ-clique, as a set of vertices in which each pair is in contact at least every Δ timestamps, has been proposed in [79, 43]. Bentert *et al.* [11] introduce the Δ-k-plex, a relaxation of Δ-clique in which each vertex has an edge to all but at most $k - 1$ vertices at least once every Δ

consecutive timestamps. Li *et al.* [54] study the problem of finding the maximum (θ,Δ)-persistent k-core in a temporal network, i.e., the largest subgraph that is a connected k-core in all the subintervals of duration θ of a given temporal interval Δ.

A different, but still slightly related body of literature focuses on other definitions of temporal patterns, such as frequent evolution patterns in temporal attributed graphs [12, 47, 23], link-formation rules in temporal networks [17, 53], frequency-estimation algorithms for counting temporal motifs [52, 57], finding a small vertex set whose removal eliminates all temporal paths connecting two designated terminal vertices [85], finding a subgraph that maximizes the sum of edge weights in a network whose topology remains fixed but edge weights evolve over time [14, 58], and the discovery of dynamic relationships and events [22], or of correlated activity patterns [37].

This work differs from all the above ones as our notions of span-core and temporal core decomposition do not correspond (or are straightforwardly reducible) to any of those temporal patterns.

2.3 Community search

Given a static graph and a set of query vertices, the *community search* problem aims at finding a cohesive subgraph containing the query vertices. Community search has attracted a great deal of attention in the last years [32, 46]. Sozio and Gionis [74] are the first to introduce this problem by employing the *minimum degree* as a cohesiveness measure. Their formulation can be solved by a simple (linear-time) greedy algorithm, which resembles the traditional 2-approximation algorithm for densest subgraph proposed in [18]. More recently, Cui *et al.* [21] devise a local-search approach to improve the efficiency of the method defined in [74], but only for the special case of a single query vertex. The case of multiple query vertices has instead been addressed by Barbieri *et al.* [7], who exploit core decomposition as a preprocessing step to improve efficiency. They also tackle the problem of *minimum community search*, i.e., a variant of community search where the size of the output subgraph has to be minimized.

Community search has also been studied under different names and/or settings. Huang *et al.* [44] introduce a community-search model based on the k-truss notion. Andersen and Lang [4] and Kloumann and Kleinberg [50] study *seed set expansion* in social graphs, in order to find communities with small conductance or that are well-resemblant of the characteristics of the query vertices, respectively. Other works define connectivity subgraphs based on electricity analogues [29], random walks [78], the minimum-description-length principle [1], the Wiener index [70] and network efficiency [69]. Recent approaches also introduce the flexibility of having query vertices belonging to different communities [13, 82]. Finally, community search has been formalized for attributed graphs [45, 30] and spatial graphs [31] as well.

In this work, we study for the first time community search in temporal graphs. Specifically, we provide a novel definition of the problem by asking for a set of subgraphs containing the given query vertices, along with their

corresponding temporal intervals, such that the total minimum-degree density of the identified subgraphs is maximized and the union of the temporal intervals spanned by those subgraphs covers the whole underlying temporal domain. None of the above works deal with such a definition of temporal community search, not even the works by Rozenshtein *et al.* [67] and Li *et al.* [54] discussed in the previous subsection. In particular, Rozenshtein *et al.* [67] and Li *et al.* [54] both search for cohesive subgraphs, but neither consider query vertices as input, which makes a crucial difference (another difference is that they consider different notions of density).

3 Temporal core decomposition: problem statement

In this section we provide preliminary definitions and the needed notations, introduce the problem of finding all span-cores and only the maximal ones, and prove containment properties among span-cores that are at the basis of our efficient algorithms.

3.1 Span-cores

We are given a *temporal graph* $G = (V, T, \tau)$, where V is a set of vertices, $T = [0, 1, \ldots, t_{max}] \subseteq \mathbb{N}$ is a discrete time domain, and $\tau : V \times V \times T \to \{0, 1\}$ is a function defining for each pair of vertices $u, v \in V$ and each timestamp $t \in T$ whether edge (u, v) exists in t. We denote $E = \{(u, v, t) \mid \tau(u, v, t) = 1\}$ the set of all temporal edges. Given a timestamp $t \in T$, $E_t = \{(u, v) \mid \tau(u, v, t) = 1\}$ is the set of edges existing at time t. A temporal interval $\Delta = [t_s, t_e]$ is contained into another temporal interval $\Delta' = [t'_s, t'_e]$, denoted $\Delta \sqsubseteq \Delta'$, if $t'_s \leq t_s$ and $t'_e \geq t_e$. Given an interval $\Delta \sqsubseteq T$, we denote $E_\Delta = \bigcap_{t \in \Delta} E_t$ the edges existing in *all timestamps* of Δ. Given a subset $S \subseteq V$ of vertices, let $E_\Delta[S] = \{(u, v) \in E_\Delta \mid u \in S, v \in S\}$ and $G_\Delta[S] = (S, E_\Delta[S])$. Finally, the *temporal degree* of a vertex u within $G_\Delta[S]$ is denoted $d_\Delta(S, u) = |\{v \in S \mid (u, v) \in E_\Delta[S]\}|$.

Definition 2 ((k, Δ)-core) *The* (k, Δ)-core *of a temporal graph* $G = (V, T, \tau)$ *is (when it exists) a maximal and non-empty set of vertices* $\emptyset \neq C_{k,\Delta} \subseteq V$*, such that* $\forall u \in C_{k,\Delta} : d_\Delta(C_{k,\Delta}, u) \geq k$*, where* $\Delta \sqsubseteq T$ *is a temporal interval and* $k \in \mathbb{N}^+$*.*

A (k, Δ)-core is thus a set of vertices implicitly defining a cohesive subgraph (where k represents the cohesiveness constraint), together with its *temporal span*, i.e., the interval Δ for which the subgraph satisfies the cohesiveness constraint. In the remainder of the paper we refer to this type of temporal pattern as *span-core*.

The first problem we tackle in this work is to compute the *span-core decomposition* of a temporal graph G, i.e., all span-cores of G.

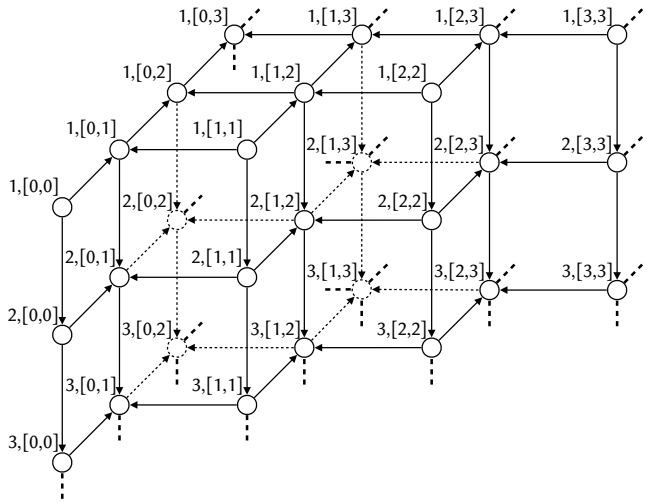

Figure 1: Search space: for a temporal span $\Delta = [t_s, t_e]$, the (k, Δ)-core is depicted as a node labeled "$k, [t_s, t_e]$". An arrow $C_1 \rightarrow C_2$ denotes $C_1 \supseteq C_2$ (the distinction between solid and dotted arrows is for visualization sake only).

Problem 1 (Span-core decomposition) *Given a temporal graph G, find the set of all (k, Δ)-cores of G.*

Unlike standard cores of simple graphs, span-cores are not all nested into each other, due to their spans. However, they still exhibit containment properties. Indeed, it can be observed that a (k, Δ)-core is contained into any other (k', Δ')-core with less restrictive degree and span conditions, i.e., $k' \leq k$, and $\Delta' \sqsubseteq \Delta$. This property is depicted in Figure 1, and formally stated in the next proposition.

Proposition 1 (Span-core containment) *For any two span-cores $C_{k,\Delta}$, $C_{k',\Delta'}$ of a temporal graph G it holds that*

$$k' \leq k \wedge \Delta' \sqsubseteq \Delta \;\Rightarrow\; C_{k,\Delta} \subseteq C_{k',\Delta'}.$$

The result can be proved by separating the two conditions in the hypothesis, i.e., by separately showing that (i) $k' \leq k \Rightarrow C_{k,\Delta} \subseteq C_{k',\Delta}$, and (ii) $\Delta' \sqsubseteq \Delta \Rightarrow C_{k,\Delta} \subseteq C_{k,\Delta'}$. The first point holds as, keeping the span Δ fixed, the maximal set of vertices C for which $d_\Delta(C, u) \geq k$ is clearly contained in the maximal set of vertices C' for which $d_\Delta(C', u) \geq k'$, if $k' \leq k$. To prove (ii), it can be noted that $\Delta' \sqsubseteq \Delta \Rightarrow E_\Delta \subseteq E_{\Delta'}$, which implies that $\forall u \in C_{k,\Delta} : d_\Delta(C_{k,\Delta}, u) \leq d_{\Delta'}(C_{k,\Delta}, u)$. Therefore, all vertices within $C_{k,\Delta}$ satisfy the condition to be part of $C_{k,\Delta'}$ too.

The following observation directly derives from Proposition 1 and states that finding all the span-cores having a fixed span Δ corresponds to computing the core decomposition of a simple graph.

Observation 1 *For a fixed temporal interval $\Delta \sqsubseteq T$, finding all span-cores that have Δ as their span is equivalent to computing the classic core decomposition [8] of the simple graph $G_\Delta = (V, E_\Delta)$.*

3.2 Maximal span-cores

As the total number of temporal intervals that are contained into the whole time domain T is $|T|(|T|+1)/2$, the total number of span-cores is potentially $\mathcal{O}(|T|^2 \times k_{max})$, where k_{max} is the largest value of k for which a (k, Δ)-core exists. It is thus quadratic in $|T|$, which may be too large an output for human direct inspection. In this regard, it may be useful to focus only on the most relevant cores, i.e., the *maximal* ones, as defined next.

Definition 3 (Maximal span-core) *A span-core $C_{k,\Delta}$ of a temporal graph G is said* maximal *if there does not exist any other span-core $C_{k',\Delta'}$ of G such that $k \leq k'$ and $\Delta \sqsubseteq \Delta'$.*

Hence, a span-core is recognized as maximal if it is not dominated by another span-core both on the order k and the span Δ. Differently from the *innermost core* (i.e., the core of the highest order) in the classic core decomposition, which is unique, in our temporal setting the number of maximal span-cores is $\mathcal{O}(|T|^2)$, as, in the worst case, there may be one maximal span-core for every temporal interval. However, as observed in empirical temporal-network data, maximal span-cores are always much less than the overall span-cores: the difference is usually one order of magnitude or more. The second problem we tackle in this work is to compute the maximal span-cores of a temporal graph.

Problem 2 (Maximal Span-core Mining) *Given a temporal graph G, find the set of all maximal (k, Δ)-cores of G.*

Clearly, one could solve Problem 2 by solving Problem 1 and filtering out all the non-maximal span-cores. However, an interesting yet challenging question is whether one can exploit the maximality condition to develop faster algorithms that can directly extract the maximal ones, without computing all the span-cores. We provide a positive answer to this question in Section 5.

4 Algorithms: computing all span-cores

In this section we devise algorithms for computing a complete span-core decomposition of a temporal graph (Problem 1).

A naïve approach. As stated in Observation 1, for a fixed temporal interval $\Delta \sqsubseteq T$, mining all span-cores $C_{k,\Delta}$ is equivalent to computing the classic core decomposition of the graph $G_\Delta = (V, E_\Delta)$. A naïve strategy is thus to run a core-decomposition subroutine [8] on graph G_Δ for each temporal interval $\Delta \sqsubseteq T$. Such a method has time complexity $\mathcal{O}(\sum_{\Delta \sqsubseteq T}(|\Delta| \times |E|))$, i.e., $\mathcal{O}(|T|^2 \times |E|)$.

A more efficient algorithm. Looking at Figure 1 one can observe that the naïve algorithm only exploits one dimension of the containment property: it starts from each point on the top level, i.e., from cores of order 1, and goes down vertically with the classic core decomposition. Based on Proposition 1, it is possible to design a more efficient algorithm that exploits also the "horizontal containment" relationships.

Example 1 *Consider core $C_{1,[0,2]}$ in Figure 1: by Proposition 1 it holds that it is a subset of both $C_{1,[0,1]}$ and $C_{1,[1,2]}$. Therefore, to compute $C_{1,[0,2]}$, instead of starting from the whole V, one can start from $C_{1,[0,1]} \cap C_{1,[1,2]}$. Starting from a much smaller set of vertices can provide a substantial speed-up to the whole computation.*

This observation, although simple, produces a speed-up of orders of magnitude as we will empirically show in Section 7. The next straightforward corollary of Proposition 1 states that, not only $C_{1,[0,2]} \subseteq C_{1,[0,1]} \cap C_{1,[1,2]}$, but this is the best one can get, meaning that intersecting these two span-cores is equivalent to intersecting all span-cores structurally containing $C_{1,[0,2]}$.

Corollary 1 *Given a temporal graph $G = (V, T, \tau)$, and a temporal interval $\Delta = [t_s, t_e] \sqsubseteq T$, let $\Delta_+ = [\min\{t_s + 1, t_e\}, t_e]$ and $\Delta_- = [t_s, \max\{t_e - 1, t_s\}]$. It holds that*

$$C_{1,\Delta} \subseteq (C_{1,\Delta_+} \cap C_{1,\Delta_-}) = \bigcap_{\Delta' \sqsubseteq \Delta} C_{1,\Delta'}.$$

Example 2 *Consider again $C_{1,[0,2]}$ in Figure 1: Proposition 1 states that it is a subset of $C_{1,[0,0]}, C_{1,[0,1]}, C_{1,[1,1]}, C_{1,[1,2]}, C_{1,[2,2]}$. Corollary 1 suggests that there is no need to intersect them all, but only $C_{1,[0,1]}$ and $C_{1,[1,2]}$: in fact, $C_{1,[0,1]} \subseteq C_{1,[0,0]} \cap C_{1,[1,1]}$ and $C_{1,[1,2]} \subseteq C_{1,[1,1]} \cap C_{1,[2,2]}$.*

The main idea behind our efficient Span-cores algorithm (whose pseudocode is given as Algorithm 1) is to generate temporal intervals of increasing size (starting from size one) and, for each Δ of width larger than one, to initiate the core decomposition from $(C_{1,\Delta_+} \cap C_{1,\Delta_-})$, i.e., the smallest intersection of cores containing $C_{1,\Delta}$ (Corollary 1). The intervals to be processed are added to queue Q, which is initialized with the intervals of size one (Lines 2–3): these are the only intervals for which no other interval can be used to reduce the set of vertices from which the core decomposition is started, thus they have to be initialized with the whole vertex set V. The algorithm utilizes a map $\mathcal{A}$ that, given an interval Δ, returns the set of vertices to be used as a starting set of the core decomposition on Δ. The algorithm processes all intervals stored in Q, until Q has become empty (Lines 4–16). For every temporal interval Δ extracted from Q, the starting set of vertices is retrieved from $\mathcal{A}[\Delta]$ and the corresponding set of edges is identified (Line 6). Unless this is empty, the classic core-decomposition algorithm [8] is invoked over $(\mathcal{A}[\Delta], E_\Delta[\mathcal{A}[\Delta]])$ (Line 8) and its output (a set of span-cores of span Δ) is added to the ultimate output set $\mathbf{C}$ (Line 9).

Algorithm 1: Span-cores

Input: A temporal graph $G = (V, T, \tau)$.
Output: The set $\mathbf{C}$ of all span-cores of G.

1 $\mathbf{C} \leftarrow \emptyset$; $Q \leftarrow \emptyset$; $\mathcal{A} \leftarrow \emptyset$
2 **forall** $t \in T$ **do**
3 enqueue $[t, t]$ to Q; $\mathcal{A}[t, t] \leftarrow V$
4 **while** $Q \neq \emptyset$ **do**
5 dequeue $\Delta = [t_s, t_e]$ from Q
6 $E_\Delta[\mathcal{A}[\Delta]] \leftarrow \{(u, v) \in E_\Delta \mid u \in \mathcal{A}[\Delta], v \in \mathcal{A}[\Delta]\}$
7 **if** $|E_\Delta[\mathcal{A}[\Delta]]| > 0$ **then**
8 $\mathbf{C}_\Delta \leftarrow$ core-decomposition$(\mathcal{A}[\Delta], E_\Delta[\mathcal{A}[\Delta]])$
9 $\mathbf{C} \leftarrow \mathbf{C} \cup \mathbf{C}_\Delta$
10 $\Delta_1 = [\max\{t_s - 1, 0\}, t_e]$; $\Delta_2 = [t_s, \min\{t_e + 1, t_{max}\}]$
11 **forall** $\Delta' \in \{\Delta_1, \Delta_2\} \mid \Delta' \neq \Delta$ **do**
12 **if** $\mathcal{A}[\Delta'] \neq$ NULL **then**
13 $\mathcal{A}[\Delta'] \leftarrow \mathcal{A}[\Delta'] \cap C_{1,\Delta}$
14 enqueue Δ' to Q
15 **else**
16 $\mathcal{A}[\Delta'] \leftarrow C_{1,\Delta}$

Afterwards, the two intervals, denoted Δ_1 and Δ_2, for which $C_{1,\Delta}$ can be used to obtain the smallest intersections of cores containing them (Corollary 1) are computed at Line 10. For Δ_1 (and analogously Δ_2), we check whether $\mathcal{A}[\Delta_1]$ has already been initialized (Line 12): this would mean that previously the other "father" (i.e., smallest containing core) of C_{1,Δ_1} has been computed, thus we can intersect $C_{1,\Delta}$ with $\mathcal{A}[\Delta_1]$ and enqueue Δ_1 to be processed (Lines 13–14). Instead, if $\mathcal{A}[\Delta_1]$ was not yet initialized, we initialize it with $C_{1,\Delta}$ (Line 16): in this case Δ_1 is not enqueued because it still lacks one father to be intersected before being ready for core decomposition. This procedural update of Q ensures that both fathers of every interval in Q exist and have been previously computed, thus no a-posteriori verification is needed.

Example 3 *Consider again the search space in Figure 1. Algorithm 1 first processes the intervals* $[0, 0], [1, 1], [2, 2]$, *and* $[3, 3]$. *Then, it intersects* $C_{1,[0,0]}$ *and* $C_{1,[1,1]}$ *to initialize* $C_{1,[0,1]}$, *intersects* $C_{1,[1,1]}$ *and* $C_{1,[2,2]}$ *to initialize* $C_{1,[1,2]}$, *and intersects* $C_{1,[2,2]}$ *and* $C_{1,[3,3]}$ *to initialize* $C_{1,[2,3]}$. *Then, it continues with the intervals of size 3: it intersects* $C_{1,[0,1]}$ *and* $C_{1,[1,2]}$ *to initialize* $C_{1,[0,2]}$ *and so on.*

The next theorem formally shows soundness and completeness of our Span-cores algorithm.

Theorem 1 *Algorithm 1 is sound and complete for Problem 1.*

The algorithm generates and processes a subset of temporal intervals $\mathcal{X} \subseteq \{\Delta \mid \Delta \sqsubseteq T\}$. For every interval $\Delta \subseteq \mathcal{X}$, it computes *all* span-cores $\mathbf{C}_\Delta = \{C_{1,\Delta}, C_{2,\Delta}, \ldots, C_{k_\Delta,\Delta}\}$ defined on Δ by means of the core-decomposition subroutine on the graph $(\mathcal{A}[\Delta], E_\Delta[\mathcal{A}[\Delta]])$. The set of vertices $\mathcal{A}[\Delta]$ is equivalent to $(C_{1,\Delta_+} \cap C_{1,\Delta_-})$ because of Line 13 (Corollary 1) and the fact that Δ is enqueued (Line 14) only when both fathers have been processed and the intersection done. The correctness of doing the classic core decomposition is guaranteed by Observation 1.

As for completeness, it suffices to show that the intervals $\Delta \notin \mathcal{X}$ that have not been processed by the algorithm do not yield any span-core. The algorithm generates all temporal intervals size by size, starting from those of size one and then going to larger sizes. This is done by maintaining the queue Q. As said above, an interval Δ is enqueued as soon as both C_{1,Δ_+} and C_{1,Δ_-} have been processed. Thus, an interval Δ is not in $\mathcal{X}$ only if either C_{1,Δ_+} or C_{1,Δ_-} does not exist. In this case $C_{1,\Delta}$ and all other $C_{k,\Delta}$ do not exist as well.

Discussion. Algorithm 1 exploits the "horizontal containment" relationships only at the first level of the search space. For a given Δ, once the restricted starting set of vertices has been defined for $k = 1$, the traditional core decomposition is started to produce all the span-cores of span Δ. In other words, for $k > 1$ only the "vertical containment" is exploited. Consider the span-core $C_{3,[1,2]}$ in Figure 1: we know that it is a subset of $C_{2,[1,2]}$ ("vertical") and of $C_{3,[1,1]}$ and $C_{3,[2,2]}$ ("horizontal"). One could consider intersecting all these three span-cores before computing $C_{3,[1,2]}$. We tested this alternative approach, but concluded that the overhead of computing intersections and data-structure maintenance was outweighing the benefit of starting from a smaller vertex set.

The worst-case time complexity of Algorithm 1 is equal to the naïve approach, however, in practice, it is orders of magnitude faster, as shown in Section 7.

5 Algorithms: computing maximal span-cores

In this section we focus on Problem 2: computing the *maximal* span-cores of a temporal graph.

A filtering approach. As anticipated above, a straightforward way of solving this problem consists in filtering the span-cores computed during the execution of Algorithm 1, so as to ultimately output only the maximal ones. This can easily be accomplished by equipping Algorithm 1 with a data structure $\mathcal{M}$ that stores the span-core of the highest order for every temporal interval $\Delta \sqsubseteq T$ that has been processed by the algorithm. Moreover, at the storage of a span-core $C_{k,\Delta}$ in $\mathcal{M}$, the span-cores previously stored in $\mathcal{M}$ for subintervals of the temporal interval Δ and with the same order k are removed from $\mathcal{M}$. This removal operation, together with the order in which span-cores are processed, ensures that $\mathcal{M}$ eventually contains only the maximal span-cores.

Efficient maximal-span-core finding. Our next goal is to design a more ef-

ficient algorithm that extracts maximal span-cores directly, without computing complete core decompositions, passing over more peripheral ones, and without generating all temporal cores. This is a quite challenging design principle, as it contrasts the intrinsic structural properties of core decomposition, based on which a core of order k is usually computed from the core of order $k-1$, thus making the computation of the core of the highest order as hard as computing the overall decomposition. Nevertheless, thanks to theoretical properties that relate the maximal span-cores to each other, in the temporal context such a challenge can be achieved. In the following we discuss such properties in detail, by starting from a result that has already been discussed above, but only informally.

Consider the classic core decomposition in a standard (non-temporal) graph G (Definition 1) and let $C_{k^*}[G]$ denote the *innermost* core of G, i.e., the non-empty k-core of G with the largest k.

Lemma 1 *Given a temporal graph $G = (V, T, \tau)$, let $\mathbf{C}_M$ be the set of all maximal span-cores of G, and $\mathbf{C}_{\mathbf{inner}} = \{C_{k^*}[G_\Delta] \mid \Delta \sqsubseteq T\}$ be the set of innermost cores of all graphs G_Δ. It holds that $\mathbf{C}_M \subseteq \mathbf{C}_{\mathbf{inner}}$.*

Every $C_{k,\Delta} \in \mathbf{C}_M$ is the innermost core of the non-temporal graph G_Δ: else, there would exist another core $C_{k',\Delta} \neq \emptyset$ with $k' > k$, implying that $C_{k,\Delta} \notin \mathbf{C}_M$.

Lemma 1 states that each maximal span-core is an innermost core of a G_Δ, for some temporal interval $\Delta \sqsubseteq T$. Hence, there can exist at most one maximal span-core for every $\Delta \sqsubseteq T$ (while an interval Δ may not yield any maximal span-core). The key question to design an efficient maximal-span-core-mining algorithm thus becomes how to extract innermost cores of the graphs G_Δ more efficiently than by computing the full core decompositions of all G_Δ. The answer to this question comes from the result stated in the next two lemmas (with Lemma 2 being auxiliary to Lemma 3).

Lemma 2 *Given a temporal graph $G = (V, T, \tau)$, and three temporal intervals $\Delta = [t_s, t_e] \sqsubseteq T$, $\Delta' = [t_s - 1, t_e] \sqsubseteq T$, and $\Delta'' = [t_s, t_e + 1] \sqsubseteq T$. The innermost core $C_{k^*}[G_\Delta]$ is a maximal span-core of G if and only if $k^* > \max\{k', k''\}$ where k' and k'' are the orders of the innermost cores of $G_{\Delta'}$ and $G_{\Delta''}$, respectively.*

The "$\Rightarrow$" part comes directly from the definition of maximal span-core (Definition 3): if k^* were not larger than $\max\{k', k''\}$, then $C_{k^*}[G_\Delta]$ would be dominated by another span-core both on the order and on the span (as both Δ' and Δ'' are superintervals of Δ). For the "$\Leftarrow$" part, from Lemma 1 and Proposition 1 it follows that $\max\{k', k''\}$ is an upper bound on the maximum order of a span-core of a superinterval of Δ. Therefore, $k^* > \max\{k', k''\}$ implies that there cannot exist any other span-core that dominates $C_{k^*}[G_\Delta]$ both on the order and on the span.

Lemma 3 *Given G, Δ, Δ', Δ'', k', and k'' defined as in Lemma 2, let $\widetilde{V} = \{u \in V \mid d_\Delta(V, u) > \max\{k', k''\}\}$, and let $C_{k^*}[G_\Delta[\widetilde{V}]]$ be the innermost core of*

Algorithm 2: Maximal-span-cores

Input: A temporal graph $G = (V, T, \tau)$.
Output: The set $\mathbf{C}_M$ of all maximal span-cores of G.

1 $\mathbf{C}_M \leftarrow \emptyset$
2 $\mathcal{K}'[t] \leftarrow 0, \forall t \in T$
3 **forall** $t_s \in [0, 1, \ldots, t_{max}]$ **do**
4 $\quad t^* \leftarrow \max\{t_e \in [t_s, t_{max}] \mid E_{[t_s, t_e]} \neq \emptyset\}$
5 $\quad k'' \leftarrow 0$
6 $\quad$ **forall** $t_e \in [t^*, t^*-1, \ldots, t_s]$ **do**
7 $\quad\quad \Delta \leftarrow [t_s, t_e]$
8 $\quad\quad lb \leftarrow \max\{\mathcal{K}'[t_e], k''\}$
9 $\quad\quad V_{lb} \leftarrow \{u \in V \mid d_\Delta(V, u) > lb\}$
10 $\quad\quad E_\Delta[V_{lb}] \leftarrow \{(u, v) \in E_\Delta \mid u \in V_{lb}, v \in V_{lb}\}$
11 $\quad\quad C \leftarrow$ innermost-core$(V_{lb}, E_\Delta[V_{lb}])$
12 $\quad\quad k^* \leftarrow$ order of C
13 $\quad\quad$ **if** $k^* > lb$ **then**
14 $\quad\quad\quad \mathbf{C}_M \leftarrow \mathbf{C}_M \cup \{C\}$
15 $\quad\quad k'' \leftarrow \max\{k'', k^*\}$; $\mathcal{K}'[t_e] \leftarrow \max\{\mathcal{K}'[t_e], k''\}$

$G_\Delta[\widetilde{V}]$. *If* $k^* > \max\{k', k''\}$, *then* $C_{k^*}[G_\Delta[\widetilde{V}]]$ *is a maximal span-core; otherwise, no maximal span-core exists for* Δ.

Lemma 2 states that, to be recognized as a maximal span-core, the innermost core of G_Δ should have order larger than $\max\{k', k''\}$. This means that, if the innermost core of G_Δ is a maximal span-core, all vertices $u \notin \widetilde{V}$ cannot be part of it. Therefore, G_Δ yields a maximal span-core only if the innermost core of subgraph $G_\Delta[\widetilde{V}]$ has order $k^* > \max\{k', k''\}$. Lemma 3 provides the basis of our efficient method for extracting maximal span-cores. Basically, it states that, to verify whether a certain temporal interval $\Delta = [t_s, t_e]$ yields a maximal span-core (and, if so, compute it), there is no need to consider the whole graph G_Δ, rather it suffices to start from a smaller subgraph, which is given by all vertices whose temporal degree is larger than the maximum between the orders of the innermost cores of intervals $\Delta' = [t_s - 1, t_e]$ and $\Delta'' = [t_s, t_e + 1]$. This finding suggests a strategy that is opposite to the one used for computing the overall span-core decomposition: a *top-down* strategy that processes temporal intervals starting from the larger ones. Indeed, in addition to exploiting the result in Lemma 3, this way of exploring the temporal-interval space allows us to skip the computation of complete core decompositions of the whole "singleton-interval" graphs $\{G_{[t,t]}\}_{t \in T}$, which may easily become a critical bottleneck, as they are the largest ones among the graphs induced by temporal intervals.

The Maximal-span-cores algorithm. Algorithm 2 iterates over all timestamps $t_s \in T$ in *increasing order* (Line 3), and for each t_s it first finds all the maximal span-cores that have span starting in t_s. This way of proceeding *ensures that a*

span-core that is recognized as maximal will not be later dominated by another span-core. Indeed, an interval $[t_s, t_e]$ can never be contained in another interval $[t'_s, t'_e]$ with $t_s < t'_s$. For a given t_s, all maximal span-cores are computed as follows. First, the maximum timestamp $\geq t_s$ such that the corresponding edge set $E_{[t_s,t_e]}$ is not empty is identified as t^* (Line 4). Then, all intervals $\Delta = [t_s, t_e]$ are considered one by one in *decreasing order* of t_e (Lines 6–7): this again *guarantees that a span-core that is recognized as maximal will not be later dominated by another span-core, as the intervals are processed from the largest to the smallest.* At each iteration of the internal cycle, the algorithm resorts to Lemma 3 and computes the lower bound lb on the order of the innermost core of G_Δ to be recognized as maximal, by taking the maximum between $\mathcal{K}'[t_e]$ and k'' (Line 8). $\mathcal{K}'$ is a map that maintains, for every timestamp $t \in [t_s, t^*]$, the order of the innermost core of graph $G_{\Delta'}$, where $\Delta' = [t_s - 1, t]$ (i.e., $\mathcal{K}'[t]$ stores what in Lemmas 2–3 is denoted as k'). Whereas k'' stores the order of the innermost core of $G_{\Delta''}$, where $\Delta'' = [t_s, t_e + 1]$. Afterwards, the sets of vertices V_{lb} and of edges $E_\Delta[V_{lb}]$ that comply with this lower-bound constraint are built (Lines 9–10), and the innermost core of the subgraph $(V_{lb}, E_\Delta[V_{lb}])$ is extracted (Lines 11–12). Ultimately, based again on Lemma 3, such a core is added to the output set of maximal span-cores only if its order is actually larger than lb (Lines 13–14), and the values of k'' and $\mathcal{K}'[t_e]$ are updated (Line 15). Specifically, note that the order k^* of core C may in principle be less than k'', as C is extracted from a subgraph of G_Δ. If this happens, it means that the actual order of the innermost core of G_Δ is equal to k''. This motivates the update rules (and their order) reported in Line 15.

Theorem 2 *Algorithm 2 is sound and complete for Problem 2.*

The algorithm processes all temporal intervals $\Delta \sqsubseteq T$ yielding a non-empty edge set E_Δ, in an order such that no interval is processed before one of its superintervals: this guarantees that a span-core recognized as maximal will not be dominated by another span-core found later on. For every Δ it extracts a core C that is used as a proxy of the innermost core of graph G_Δ. C is added to the output set $\mathbf{C}_M$ only if Lemma 3 recognizes it as a maximal span-core, otherwise it is discarded. This proves the soundness of the algorithm. Completeness follows from Lemma 1, which states that to extract all maximal span-cores it suffices to focus on the innermost cores of graphs $\{G_\Delta \mid \Delta \sqsubseteq T\}$, and Lemma 3 again, which states the condition for a proxy core C to be safely discarded because it is a non-maximal span-core.

Discussion. The worst-case time complexity of Algorithm 2 is the same as the algorithm for computing the overall span-core decomposition, i.e., $\mathcal{O}(|T|^2 \times |E|)$. It is worth mentioning that it is not possible to do better than this, as the output itself is potentially quadratic in $|T|$. However, as we will show in Section 7, the proposed algorithm is in practice much more efficient than computing the overall span-core decomposition and filtering out the non-maximal span-cores as, in this case, we avoid the visit of portions of the span-core search space and the computations are run over subgraphs of reduced dimensions.

To conclude, we discuss how the crucial operation of building the subgraph $(V_{lb}, E_\Delta[V_{lb}])$ may be carried out efficiently in terms of both time and space. Consider a fixed timestamp $t_s \in [0, \ldots, t_{max}]$. The following reasoning holds for every t_s. Let $E^-(t_e) = E_{[t_s,t_e]} \setminus E_{[t_s,t_e+1]}$ be the set of edges that are in $E_{[t_s,t_e]}$ but not in $E_{[t_s,t_e+1]}$, for $t_e \in [t_s, \ldots, t^*-1]$. As a first general step, for each t_s, we compute and store *all* edge sets $\{E^-(t_e)\}_{t_e \in [t_s,t^*-1]}$. These operations can be accomplished in $\mathcal{O}(|T| \times |E|)$ overall time, because every $E^-(t_e)$ can be computed incrementally from $E_{[t_s,t_e]}$ as $E^-(t_e) = \{(u,v) \in E_{[t_s,t_e]} \mid \tau(u,v,t_e+1) = 0\}$. Moreover, for any timestamp t_e, we keep a map $\mathcal{D}$ storing all vertices of $G_{[t_s,t_e]}$ organized by degree. Specifically, the set $\mathcal{D}[k]$ contains all vertices having degree $> k$ in $G_{[t_s,t_e]}$. Every vertex in $\mathcal{D}$ is thus replicated a number of times equal to its degree. This way, the overall space taken by $\mathcal{D}$ is $\mathcal{O}(|E|)$, i.e., as much space as G. $\mathcal{D}$ is initialized as empty (when $t_e = t^*$) and repeatedly augmented as t_e decreases, by a linear scan of the various $E^-(t_e)$. The overall filling of $\mathcal{D}$ (for all t_e) therefore takes $\mathcal{O}(|T| \times |E|)$ time. Then, the desired V_{lb} can be computed in constant time simply as $V_{lb} = \mathcal{D}[lb]$.

As for $E_\Delta[V_{lb}]$, for any t_e, we first reconstruct $E_{[t_s,t_e]}$ as $E_{[t_s,t_e+1]} \cup E^-(t_e)$, having previously computed $E_{[t_s,t_e+1]}$. Note that storing all $E^-(t_e)$ takes $\mathcal{O}(|E|)$ space. That is why we store all $E^-(t_e)$ and reconstruct $E_{[t_s,t_e]}$ afterward (instead of storing the latter, which would take $\mathcal{O}(|T| \times |E|)$ space). $E_\Delta[V_{lb}]$ is ultimately derived by a linear scan of $E_{[t_s,t_e]}$, taking all edges in $E_{[t_s,t_e]}$ having both endpoints in V_{lb}. This way, the step of building $E_\Delta[V_{lb}]$ for all t_e takes again $\mathcal{O}(|T| \times |E|)$ overall time.

6 Temporal community search

Community search in static graphs aims at finding a dense subgraph (community) containing a set of input query vertices [32, 46]. In the temporal setting it is very likely that the communities spanning the query vertices change over time. To be more precise, it may happen that a certain subgraph S is a well-representative community for the given query vertices Q, but only for a certain time interval Δ. Instead, for another time interval Δ', a relevant community for Q might correspond to a completely different subgraph S'. For this reason, we formulate community search on temporal networks as the problem of finding h subgraphs (with $h > 0$ being an input parameter) containing the query vertices, together with their temporal span, such that the sum of the density of those subgraphs is maximized and the union of their temporal spans corresponds to the whole input temporal domain. Among the many densities proposed in the literature, here we follow the seminal work by Sozio and Gionis [74] on community search and adopt the minimum degree. Formally:

Problem 3 (Temporal Community Search) *Given a temporal graph $G = (V, T, \tau)$, a set $Q \subseteq V$ of query vertices, and a positive integer $h \in \mathbb{N}^+$, find a set $\{\langle S_i, \Delta_i \rangle\}_{i=1}^h$ of h pairs such that (i) $\forall 1 \leq i \leq h : Q \subseteq S_i \subseteq V$, (ii)*

$\bigcup_{1 \leq i \leq h} \Delta_i = T$, *and (iii) the following is maximized:*

$$\sum_{i=1}^{h} \min_{u \in S_i} d_{\Delta_i}(S_i, u). \tag{1}$$

The input integer h is a user-defined parameter that gives the analyst the flexibility of requiring a specific number of output temporal communities, which might vary from application to application.

6.1 Connection with Sequence Segmentation

Here we provide some theoretical insights into the Temporal Community Search problem. The main result we provide at the end of this subsection is an interesting connection with the well-established Sequence Segmentation problem [10]. As shown in the next subsections, such a result forms the basis for algorithmic design.

Let us first consider a single-interval variant of Problem 3: for a fixed temporal interval Δ, find a subgraph containing the input set Q of query vertices that maximizes the minimum temporal degree within Δ. Formally:

Problem 4 (Single Temporal Community Search) *Given a temporal graph $G = (V, T, \tau)$, a set $Q \subseteq V$ of query vertices, and an interval $\Delta \sqsubseteq T$, find*

$$S^* = \text{argmax}_{Q \subseteq S \subseteq V} \min_{u \in S} d_{\Delta}(S, u).$$

It is easy to see that solving Problem 4 corresponds to solving minimum-degree-based community search on graph G_{Δ}. Therefore, a solution to Problem 4 can straightforwardly be computed by applying a standard result on minimum-degree-based community search, which states that the highest-order core containing all query vertices is a solution to that problem [7]. This finding is formalized next.

Definition 4 ((Q, Δ)-highest-order-span-core) *Given a temporal graph $G = (V, T, \tau)$, a set $Q \subseteq V$ of query vertices, and an interval $\Delta \sqsubseteq T$, the (Q, Δ)-*highest-order-span-core *of G, denoted $C^*_{Q,\Delta}$, is defined as the highest-order span-core among all span-cores of G with temporal span Δ and containing all query vertices in Q. Let also $v^*_{Q,\Delta}$ denote the order of $C^*_{Q,\Delta}$.*

Fact 1 *Given a temporal graph $G = (V, T, \tau)$, a set $Q \subseteq V$ of query vertices, and an interval $\Delta \sqsubseteq T$, the (Q, Δ)-highest-order-span-core of G is a solution to Problem 4 on input $\langle G, Q, \Delta \rangle$.*

Note that Problem 4 may have multiple solutions: $C^*_{Q,\Delta}$ is only one of those possibly many ones. $C^*_{Q,\Delta}$ can be computed by running a core decomposition on (static) graph G_{Δ}, and stopping it when the first core that does not contain all query vertices in Q has been encountered. Therefore, Problem 4 can be solved in $\mathcal{O}(|\Delta| \times |E|)$ time.

In light of the above findings, an alternative yet equivalent way of formulating our Temporal Community Search problem is to ask for a *segmentation* (i.e., a partition) of the time domain T into a set $\{\Delta_i\}_{i=1}^h$ of h intervals so as to maximize the sum $\sum_{i=1}^h v^*_{Q,\Delta_i}$ of the orders of the (Q,Δ)-highest-order-span-cores of those identified intervals. Once such an optimal segmentation of T has been computed, the ultimate $\{\langle S_i,\Delta_i\rangle\}_{i=1}^h$ pairs are derived by simply setting $S_i = C^*_{Q,\Delta_i}$, $\forall 1 \leq i \leq h$. Formally:

Problem 5 (Alternative formulation of Problem 3) *Given a temporal graph $G = (V,T,\tau)$, a set $Q \subseteq V$ of query vertices, and a positive integer $h \in \mathbb{N}^+$, find a set $\{\langle S_i,\Delta_i\rangle\}_{i=1}^h$ of h pairs such that (i) $\forall 1 \leq i \leq h : S_i = C^*_{Q,\Delta_i}$, (ii) $\{\Delta_i\}_{i=1}^h$ is a partition of T, and (iii) the following is maximized:*

$$\sum_{i=1}^{h} v^*_{Q,\Delta_i}. \tag{2}$$

Correspondence between Problem 3 and Problem 5 easily follows from Fact 1 and from the observation that for any feasible solution $\{\langle S_i,\Delta_i\rangle\}_{i=1}^h$ to Problem 3 with overlapping intervals, there exists an overlapping-interval-free feasible solution with not smaller objective-function value. To see the latter, for any two overlapping intervals Δ_i and Δ_j, simply replace one of the two intervals, say Δ_i, with $\Delta'_i = \Delta_i \setminus (\Delta_i \cap \Delta_j)$. As $\Delta'_i \sqsubseteq \Delta_i$, it holds that $v^*_{Q,\Delta'_i} \geq v^*_{Q,\Delta_i}$, therefore the resulting overlapping-interval-free solution will have objective-function value greater than or equal to the objective-function value of the starting solution with overlapping intervals.

Thanks to the reformulation in Problem 5, it is immediate to observe that our Temporal Community Search problem is an instance of the well-established Sequence Segmentation problem, which asks for partitioning a sequence of numerical values into b segments so as to minimize the sum of the penalties (according to some penalty function) on each identified segment [10]:

Problem 6 (Sequence Segmentation [10]) *Given a sequence $X = (x_0, x_1, \ldots, x_{max})$ of numerical values, and a function $p : \{Y\}_{Y \sqsubseteq X} \to \mathbb{R}$ that assigns a penalty score to every subsequence Y of X, partition X into a set $\{X_i\}_{i=1}^b$ of b subsequences such that $\sum_{i=1}^b p(X_i)$ is minimized.*

Fact 2 Temporal Community Search *(Problem 3) on input $\langle G = (V,T,\tau), Q, h\rangle$ is an instance of* Sequence Segmentation *(Problem 6) with $X = T$, $b = h$, and $\forall \Delta \sqsubseteq T : p(\Delta) = -v^*_{Q,\Delta}$.*

In the following two subsections we show how to exploit the result in Fact 2 (and a further important finding about maximal span-cores) to design efficient algorithms for our Temporal Community Search problem.

Algorithm 3: Temporal-community-search

Input: A temporal graph $G = (V, E, T)$, a set $Q \subseteq V$ of query vertices, an integer $h \in \mathbb{N}^+$.

Output: A set $\{\langle S_i, \Delta_i \rangle\}_{i=1}^{h}$, where $Q \subseteq S_i \subseteq V$, $\forall 1 \leq i \leq h$, and $\{\Delta_i\}_{i=1}^{h}$ is a partition of T.

```
/* Initialization */
```

Compute $v^*_{Q,\Delta}$ and $C^*_{Q,\Delta}$, $\forall \Delta \sqsubseteq T$, via Q-constrained span-core decomposition

$\mathbf{P} \leftarrow$ an empty $(|T| \times h)$-dimensional matrix `// Penalty matrix`

$\mathbf{R} \leftarrow$ an empty $(|T| \times h)$-dimensional matrix `// Reconstruction matrix`

forall $t \in T$ **do**

  $\mathbf{P}[t, 0] \leftarrow -v^*_{Q,[0,t]}$

  $\mathbf{R}[t, 0] \leftarrow 0$

```
/* Dynamic-programming step */
```

forall $t \in T$ **do**

  forall $i \in [1, h)$ **do**

    $\mathbf{P}[t, i] \leftarrow \min_{\ell \in [0,t]} \mathbf{P}[\ell, i-1] - v^*_{Q,[\ell+1,t]}$

    $\mathbf{R}[t, i] \leftarrow \operatorname{argmin}_{\ell \in [0,t]} \mathbf{P}[\ell, i-1] - v^*_{Q,[\ell+1,t]}$

```
/* Reconstruction of the solution */
```

$ub \leftarrow t_{max}$

forall $i \in (h, 0]$ **do**

  $lb \leftarrow \mathbf{R}[ub, i]$

  $\Delta_i \leftarrow [lb, ub]$

  $ub \leftarrow lb - 1$

forall $i \in (h, 0]$ **do**

  $S_i \leftarrow C^*_{Q,\Delta_i}$

6.2 A basic algorithm (based on all span-cores)

Sequence Segmentation can be solved in $\mathcal{O}(|X|^2 \times h + \tau_p)$ time via dynamic programming [10], where τ_p is the overall time spent for computing the penalty score of all subsequences of the input sequence X (according to the given penalty function p). Thanks to the connection shown in Fact 2, the dynamic-programming algorithm for Sequence Segmentation can be easily adapted to solve Temporal Community Search as well. The pseudocode of this algorithm – termed Temporal-community-search – is reported as Algorithm 3, and described next.

The Temporal-community-search algorithm makes use of two $(|T| \times h)$-dimensional matrices, i.e., $\mathbf{P}$ and $\mathbf{R}$. Matrix $\mathbf{P}$ represents the *penalty matrix*. It contains, $\forall t \in T$, $\forall i \in [0, h)$, the minimum cost of segmenting the sequence corresponding to the first t timestamps of T into $i + 1$ segments. As a result, $\mathbf{P}[t_{max}, h - 1]$ contains the objective-function value of the ultimate optimal solution to Prob-

lem 5. Matrix $\mathbf{R}$ is the *reconstruction matrix*. It provides information about the optimal segmentation, and is used at the end of the algorithm to reconstruct the output $\{\Delta_i\}_{i=1}^{h}$. Note that the algorithm does not explicitly compute the S_i subgraphs corresponding to the optimal Δ_i intervals. In fact, as discussed above, each S_i can be easily retrieved at the end of the algorithm, by simply setting it equal to the corresponding (Q, Δ_i)-highest-order-span-core C^*_{Q,Δ_i}. According to Fact 2, the penalty score of an interval $\Delta \sqsubseteq T$ corresponds to $-v^*_{Q,\Delta}$, i.e., the negative of the order of the (Q, Δ)-highest-order-span-core $C^*_{Q,\Delta}$. All individual $v^*_{Q,\Delta}$ values, for all $\Delta \sqsubseteq T$, are efficiently computed altogether, at the beginning of the algorithm, via a "Q-constrained" variant of span-core decomposition (an alternative, but much less efficient strategy consists in computing every single $v^*_{Q,\Delta}$ from scratch, on the fly). Specifically, a simple (yet more efficient) variant of the span-core decomposition algorithm (Algorithm 1) is employed for this purpose, which outputs only those span-cores containing all the vertices in Q. This is easily achievable by stopping the core-decomposition subroutine, for every interval $\Delta \sqsubseteq T$, as soon as a core not containing all query vertices in Q has been encountered.

The time complexity of Algorithm 3 is $\mathcal{O}(|T|^2 \times h + \tau_{sc})$, where τ_{sc} is the time spent for computing the Q-constrained span-core decomposition of the input graph G.

6.3 A more efficient algorithm (based on maximal span-cores)

A more efficient algorithm can be designed by noticing that, actually, one does not need to consider all timestamps in T in the dynamic-programming step. Rather, focusing on a subset $T^* \subseteq T$ – which is properly defined based on the maximal span-cores of the input graph, see next – allows for significantly reducing the dimensionality of the penalty matrix P and the reconstruction matrix R, hence the overall time complexity of the algorithm, without affecting optimality of the output solution. The following fact provides the theoretical basis for defining such a reduced temporal domain T^*.

Fact 3 *Given a temporal graph $G = (V, T, \tau)$ and a set $Q \subseteq V$ of query vertices, let $\mathbf{C}_M(Q)$ be the set of all Q-constrained maximal span-cores of G. For a temporal interval $\Delta \sqsubseteq T$, it holds that $v^*_{Q,\Delta} = \max\{0, \max\{k \mid C_{k,\Delta'} \in \mathbf{C}_M(Q), \Delta \sqsubseteq \Delta'\}\}$.*

Fact 3 states that the penalty score $v^*_{Q,\Delta}$ of an interval Δ corresponds to the maximum among the orders of the Q-constrained maximal span-cores whose span includes Δ, if some exist. If an interval Δ is not a subset of any span of a Q-constrained maximal span-core, then $v^*_{Q,\Delta} = 0$. In that case, therefore, Δ can be safely discarded, as it cannot be part of the optimal solution of the given Temporal Community Search problem instance (unless it is needed to fill possible "holes", see below). The ultimate consequence of this finding is that the aforementioned reduced temporal domain T^* is identified by

the timestamps covered by the spans of the maximal span-cores, along with auxiliary timestamps, which are needed to ensure a smooth execution of the dynamic-programming step, as well as a correct handling of some extreme cases. Specifically, let $\mathbf{D} = \{\Delta \sqsubseteq T \mid C_{k,\Delta} \in C_M(Q)\}$ be the set of the spans of the Q-constrained maximal span-cores of the input graph, and $T_{\mathbf{D}} = \bigcup_{\Delta \in \mathbf{D}} \Delta$ be the set of timestamps that are part of a span of a Q-constrained maximal span-core. The first two sets of auxiliary timestamps correspond to the timestamps that immediately precede and succeed the intervals in $\mathbf{D}$, i.e., the sets $T^+_{\mathbf{D}} = \{\min\{t_e+1, t_{max}\} \mid [t_s, t_e] \in \mathbf{D}\}$ and $T^-_{\mathbf{D}} = \{\max\{t_s-1, 0\} \mid [t_s, t_e] \in \mathbf{D}\}$, respectively. The timestamps in $T^+_{\mathbf{D}}$ and $T^-_{\mathbf{D}}$ (along with the last timestamp t_{max} of the input temporal domain T) are needed to allow the dynamic-programming step to identify a solution that actually covers the whole temporal domain T (as per Condition (ii) of Problem 3). In particular, such timestamps may be interpreted as a trick to give the dynamic-programming step the flexibility to select "holes" (i.e., time intervals in-between two consecutive but not necessarily contiguous timestamps in $T_{\mathbf{D}}$). Moreover, we define T_{sup} as the set of the first $h + 1 - |T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}|$ timestamps of T not contained in $T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}$, i.e., $T_{sup} = \{t_i \in T \setminus (T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}) \mid i \in [1, h + 1 - |T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}|]\}$. The timestamps in T_{sup} are further auxiliary timestamps that are needed to return a correct h-sized solution when the timestamps in $T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}$ are less than $h + 1$ (the minimum number of timestamps required in T^* to have a solution of size h). Note that T_{sup} is nonempty only if $|T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}| < h + 1$. Ultimately, T^* is defined as

$$T^* = T_{\mathbf{D}} \ \cup \ T^+_{\mathbf{D}} \ \cup \ T^-_{\mathbf{D}} \ \cup \ \{t_{max}\} \ \cup \ T_{sup}.$$

The proposed more efficient method for Temporal Community Search, termed Efficient-temporal-community-search, is summarized in Algorithm 4 and described next. The first five lines of the algorithm are devoted to the identification of T^*. As said above, matrices $\mathbf{P}$ and $\mathbf{R}$ have here reduced dimensionality with respect to Algorithm 3: they are $(|T^*| \times h)$-dimensional matrices, where $|T^*| \leq |T|$. A mapping function $\mathbf{M}$ is used to assign an index within $[0, |T^*|)$ to every timestamp in $|T^*|$ (Line 6). Such a mapping is needed to have every timestamp in $|T^*|$ logically assigned to a row of matrices $\mathbf{P}$ and $\mathbf{R}$. The rest of the algorithm resembles Algorithm 3, except for the fact that $\mathbf{M}$ is used every time that a row index has to be mapped to its corresponding timestamp (e.g., during the reconstruction of the solution).

An important point to clarify is that, during the execution of the Efficient-temporal-community-search algorithm, we might need the penalty score $v^*_{Q,\Delta}$ of intervals $\Delta \sqsubseteq T$ corresponding to *non-maximal* (Q-constrained) span-cores. Therefore, the algorithm needs the $v^*_{Q,\Delta}$ score of *all* intervals $\Delta \sqsubseteq T$. To compute these $v^*_{Q,\Delta}$ scores (and, related to this, the set $\mathbf{C}_M(Q)$ of Q-constrained maximal span-cores, at Line 1), there are two main options. The first one consists in computing the whole Q-constrained span-core decomposition (as done in Algorithm 3), keep the $v^*_{Q,\Delta}$ scores of all such cores, and eventually compute $\mathbf{C}_M(Q)$ by simply filtering out non-maximal span-cores. The second option

corresponds instead to compute $\mathbf{C}_M(Q)$ directly, without passing through the whole Q-constrained span-core decomposition. This may be carried out by running a simple variant of the algorithm for computing maximal span-cores (Algorithm 2), where containment of query vertices is added as a further constraint. The computation of all the $v^*_{Q,\Delta}$ scores comes for free during the execution of this algorithm for Q-constrained maximal span-cores: these scores can therefore be retained by adding a few straightforward (constant-time) instructions to that algorithm. In our implementation we stick to the latter, as the Maximal-span-cores algorithm has been experimentally recognized as faster than the naïve filtering approach in all tested datasets.

The time complexity of the proposed Efficient-temporal-community-search algorithm is $\mathcal{O}(|T^*|^2 \times h + \tau_{msc})$, with τ_{msc} being the time spent in computing the Q-constrained maximal span-cores and the penalty scores $v^*_{Q,\Delta}$. As in practice (attested by our experiments) $|T^*| \ll |T|$, the proposed Efficient-temporal-community-search algorithm is expected to be much more efficient than its naïve counterpart, i.e., Algorithm 3.

6.4 Minimum community search

An instance of Temporal Community Search may admit several optimal solutions which might differ either in terms of output intervals $\{\Delta_i\}_{i=1}^h$, or in terms of subgraphs assigned to the various identified intervals. More precisely, the latter refers to the fact that two optimal solutions might find the same segmentation $\{\Delta_i\}_{i=1}^h$ of the input temporal domain, but select different subgraphs S_i for any interval Δ_i. Therefore, if the communities S_i are not chosen carefully, they may result to be excessively large, not really cohesive, and containing redundant/outlying vertices. This is a well-recognized issue of minimum-degree-based community search [74]. At the same time, large communities might include more cohesive and denser subgraphs that still exhibit optimality. Motivated by this, in this subsection we devise a method to refine the communities originally found by our algorithms for Temporal Community Search, specifically attempting to minimize their size while preserving optimality. The main idea behind our refinement method is based on the following result:

Proposition 2 (Community containment) *Given a temporal graph $G = (V, T, \tau)$, a set $Q \subseteq V$ of query vertices, and a positive integer $h \in \mathbb{N}^+$, let $\{\langle S_i, \Delta_i\rangle\}_{i=1}^h$ be a solution to Problem 3 on input $\langle G, Q, h\rangle$ with S_i corresponding to the (Q, Δ_i)-highest-order-span-core of G, $\forall i \in [1, h]$. For every other solution $\{\langle S'_i, \Delta_i\rangle\}_{i=1}^h$ (referring to the same segmentation $\{\Delta_i\}_{i=1}^h$) to Problem 3 on input $\langle G, Q, h\rangle$ it holds that $S'_i \subseteq S_i$, $\forall i \in [1, h]$.*

Let k_i be the minimum degree of S_i, i.e., $k_i = v^*_{Q,\Delta_i}$ is the order of the (Q, Δ_i)-highest-order-span-core. Assume that there exists a solution S'_i to Problem 4 that is not contained in S_i. This implies that (i) the minimum degree of a vertex of S'_i in Δ_i is k_i, and (ii) the minimum degree of a vertex of $S_i \cup S'_i$ in Δ_i is k_i

as well. This violates the maximality condition of the definition of span-core, since, by hypothesis, S_i corresponds to the (Q, Δ_i)-highest-order-span-core of G.

The above proposition states that, given a solution $\{\langle S_i, \Delta_i \rangle\}_{i=1}^{h}$ to the Temporal Community Search problem where every S_i corresponds to the (Q, Δ_i)-highest-order-span-core of the input graph, one can focus on the various S_i solely to refine the output communities, as such S_i are guaranteed to contain *all* optimal solutions of the underlying problem instance (while keeping the segmentation $\{\Delta_i\}_{i=1}^{h}$ fixed). Within this view, we formulate the following optimization problem (which is a variant of Problem 4, with the additional constraint of requiring a smallest-sized solution):

Problem 7 *Given a temporal graph $G = (V, T, \tau)$, a set $Q \subseteq V$ of query vertices, and an interval $\Delta \sqsubseteq T$, let $S^* \subseteq V$ be the subset of vertices containing all the solutions to Problem 4 on input $\langle G, Q, \Delta \rangle$ (according to what stated in Proposition 2). Find*

$$S^*_{min} = \text{argmin}_{\{S | Q \subseteq S \subseteq S^*, \min_{u \in S} d_\Delta(S,u) \geq \min_{u \in S^*} d_\Delta(S^*,u)\}} |S|.$$

Theorem 3 *Problem 7 in* **NP***-hard.*

Consider (the optimization version of) the **NP**-hard mCST problem introduced by Cui *et al.* [21]: given a graph $H = (V_H, E_H)$ and a query vertex $q \in V_H$, find a minimum-sized subgraph that contains q, is connected, and maximizes the minimum degree. Given an instance $\langle H, q \rangle$ of the mCST problem, construct an instance $\langle G, Q, \Delta \rangle$ of Problem 7 by defining G as composed by a single temporal snapshot corresponding to graph H, Δ as a singleton interval composed of the single timestamp of G, and setting $Q = \{q\}$. It is straightforward to see that solving Problem 7 on input $\langle G, Q, \Delta \rangle$ is equivalent to solving mCST on input $\langle H, q \rangle$, as the constraint about connectedness is automatically satisfied in Problem 7 for the special case of a single query vertex.

As Problem 7 is **NP**-hard, we devise a heuristic that is inspired to the greedy one proposed for the Minimum Community Search problem in [7]. The proposed heuristic is outlined in Algorithm 5 and described next. In the pseudocode and in the following we denote as k^* and k^*_{min} the minimum degree of S^* and S^*_{min}, respectively, and as $neigh_\Delta(S, u)$ the neighbors of a vertex $u \in V$ in the subgraph induced by $S \subseteq V$ and $\Delta \sqsubseteq T$. Algorithm 5 iteratively adds vertices to the solution S^*_{min} according to a priority queue P. Priorities of vertices in P are defined based on a score that measures how promising a vertex is for making the current solution S^*_{min} reach the optimal minimum degree. Specifically, the score of a vertex $u \in S^*$ is defined as:

$$score(u) = score^+(u) - score^-(u),$$

where

$$score^+(u) = |\{v \in neigh_\Delta(S^*_{min}, u) \mid d_\Delta(S^*_{min}, v) < k^*\}|;$$
$$score^-(u) = \max\{0, k^* - d_\Delta(S^*_{min}, u)\}.$$

$score^+(u)$ is the gain effect of adding u to S^*_{min}, while $score^-(u)$ is the penalty effect. In particular, $score^+(u)$ counts the number of neighbors of u in S^*_{min} that would benefit from the inclusion of u to S^*_{min}, i.e., that have degree less than k^*. On the other hand, $score^-(u)$ represents the number of neighbors of u still required in S^*_{min} so that u has degree at least k^*. The algorithm starts by adding the query vertices to the queue P with priority $+\infty$, in order to ensure that they will be selected at the very beginning. At each iteration of the main cycle of the algorithm (starting at Line 4), the vertex u exhibiting the highest priority is dequeued from P and is added to the solution S^*_{min}. As a consequence, a couple of updates are performed. First, u's neighbors not in the priority queue P are added to it (Lines 8-9). Note that this is the only step of the algorithm where the score of a vertex is computed from scratch and stored in $\mathcal{A}$, a map that keeps the scores of all vertices in P up-to-date during the whole execution of the algorithm. The second update consists in recomputing the score of every v's neighbor w in the queue, if a vertex $v \in S^*_{min}$ has reached the desired minimum degree k^* after the addition of u.

7 Experiments

In this section we present an experimental evaluation to empirically assess the performance of all the proposed methods. Specifically, we focus on whole span-core decomposition (Section 7.1), maximal span-cores (Section 7.2), characterization of the extracted span-cores (Section 7.3), and temporal community search (Section 7.4).

Datasets. We use eleven real-world datasets recording timestamped interactions between entities. For each dataset we select a window size to define a discrete time domain, composed of contiguous timestamps of the same duration, and build the corresponding temporal graph. If multiple interactions occur between two entities during the same discrete timestamp, they are counted as one. The characteristics of the resulting temporal graphs, along with the selected window sizes, are reported in Table 1.

The three smallest datasets were gathered by using wearable proximity sensors in schools, with a temporal resolution of 20 seconds. PrimarySchool[2] contains the contact events between 242 volunteers (232 children and 10 teachers) in a primary school in Lyon, France, during two days [76]. HighSchool[2] describes the close-range proximity interactions between students and teachers (327 individuals overall) of nine classes during five days in a high school in Marseilles, France [61]. HongKong reports the same kind of interactions for a primary school in Hong Kong, whose population consists of 709 children and 65 teachers divided into thirty classes, for eleven consecutive days [71].

ProsperLoans[3] represents the network of loans between the users of Prosper, a marketplace of loans between privates. Last.fm[3] records the co-listening activity of the Last.fm streaming platform: an edge exists between two users if

[2] sociopatterns.org
[3] konect.cc

they listened to songs of the same band within the same discrete timestamp. WikiTalk[3] is the communication network of the English Wikipedia. DBLP[3] is the co-authorship network of the authors of scientific papers from the DBLP computer science bibliography. StackOverflow[4] includes the answer-to-question interactions on the stack exchange of the stackoverflow.com website. Wikipedia[3] connects users of the Italian Wikipedia that co-edited a page during the same discrete timestamp. Finally, for both Amazon[3] and Epinions[3], vertices are users and edges represent the rating of at least one common item within the same discrete timestamp.

Implementation. All methods are implemented in Python (v. 2.7.16) and compiled by Cython. All the experiments were run on a machine equipped with Intel Xeon CPU at 2.1GHz. The experiments reported in Sections 7.1 and 7.2 used 64GB RAM, while the ones in Section 7.4 used 32GB RAM.

7.1 Span-core decomposition

We compare the two methods to compute a complete decomposition described in Section 4, i.e., the baseline Naïve-span-cores and the proposed Span-cores, in terms of execution time, memory, and total number of vertices input to the core-decomposition subroutine. We report these measures, together with the number of span-cores and maximal span-cores of each dataset, in Table 2.

In terms of execution time, Span-cores considerably outperforms Naïve-span-cores in all datasets, achieving a speed-up from 2.1 up to two orders of magnitude. The speed-up is explained by the number of vertices processed by the core-decomposition subroutine, which is the most time-consuming step of the algorithms albeit linear in the size of the input subgraph. The difference of this quantity between Span-cores and Naïve-span-cores reaches over an order of magnitude in the WikiTalk, Wikipedia, and Epinions dataset, confirming the effectiveness of the "horizontal containment" relationships. The memory required by the two procedures is comparable in all cases since the largest structures needed in memory are the temporal graph itself and the set $\mathbf{C}$ of all span-cores.

7.2 Maximal span-cores

We compare our Maximal-span-cores algorithm to the naïve approach, described at the beginning of Section 5, based on running the Span-cores algorithm and filtering out the non-maximal span-cores, which we refer to as Naïve-maximal-span-cores. The results are again reported in Table 2.

Naïve-maximal-span-cores behaves very similarly to Span-cores: they only differ for the filtering mechanism which requires a few additional seconds in most cases. Maximal-span-cores is much faster than Naïve-maximal-span-cores for all datasets, with a speed-up from 1.3 for the Epinions dataset to one order of magnitude for the HongKong dataset. Except for the school datasets and Last.fm, the difference in terms of number of processed vertices is between one and

[4] snap.stanford.edu

three orders of magnitude, attesting the advantages of the top-down strategy of Maximal-span-cores, which avoids the visit of portions of the span-core search space and handles the overhead of reconstructing graphs, i.e., $(V_{lb}, E_{\Delta}[V_{lb}])$, efficiently. Finally, the memory requirements of the two methods are comparable for all datasets.

7.3 Span-cores characterization

We compare and characterize all span-cores against maximal span-cores. At first, Table 2 shows that span-cores are at least one order of magnitude more numerous than maximal span-cores for all datasets, with the maximum difference of three orders of magnitude for the HongKong dataset.

In Figure 2 we show the number (top) and the average size (bottom) of span-cores and maximal span-cores as a function of the order k for the DBLP and Epinions datasets. For both datasets, the number of maximal span-cores is at least one order of magnitude lower than the total number of span-cores up to a quarter of the k domain, where the span-cores are more numerous. Instead, in the rest of the domain, they mostly coincide due to the maximality condition over $|\Delta|$. The average size is also smaller for maximal span-cores, difference that wears thin when the gap between the numbers of span-cores and maximal span-cores starts decreasing since, for high values of k, most (or all) span-cores are maximal.

Figure 3 shows a different picture when numbers and average sizes of span-cores are shown as a function of the size of the span $|\Delta|$. For both datasets, the number of span-cores and maximal span-cores is decreasing – which is expected since the number of intervals decreases when $|\Delta|$ increases – with a constant gap close to one and two orders of magnitude, respectively. On the other hand, the behavior of the average size is quite different between the two datasets. For low values of $|\Delta|$, the average size of span-cores of the DBLP dataset is much higher than the average size of maximal span-cores, then the difference decreases and vanishes at the end of domain where a maximal span-core of $|\Delta| = 37$ dominates all other span-cores with $|\Delta| \geq 20$. Instead, for the Epinions dataset, the average size of all span-cores and of maximal span-cores follow the same behavior, with a difference of less than an order of magnitude, because the maximality condition over k excludes the largest span-cores from the set of maximal span-cores.

7.4 Temporal community search

In this subsection we assess the performance of the proposed algorithms for temporal community search (presented in Sections 6.2–6.3), as well as the greedy procedure for reducing the size of the output communities (presented in Section 6.4). In the remainder of this subsection we refer to our basic algorithm (i.e., Algorithm 3, which precomputes the penalty scores via span-core decomposition) as SC-TCS, and to our more efficient algorithm (i.e., Algorithm 4, which exploits maximal span-cores to reduce the number of timestamps to be considered) as MSC-TCS. We also involve in the comparison a naïve version of

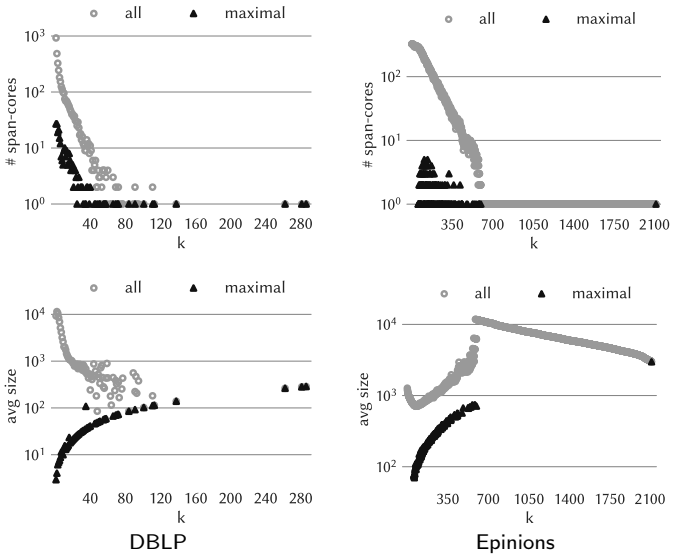

Figure 2: Top plots: number of span-cores and maximal span-cores as a function of the order k. Bottom plots: average size of all span-cores and maximal span-cores as a function of the order k.

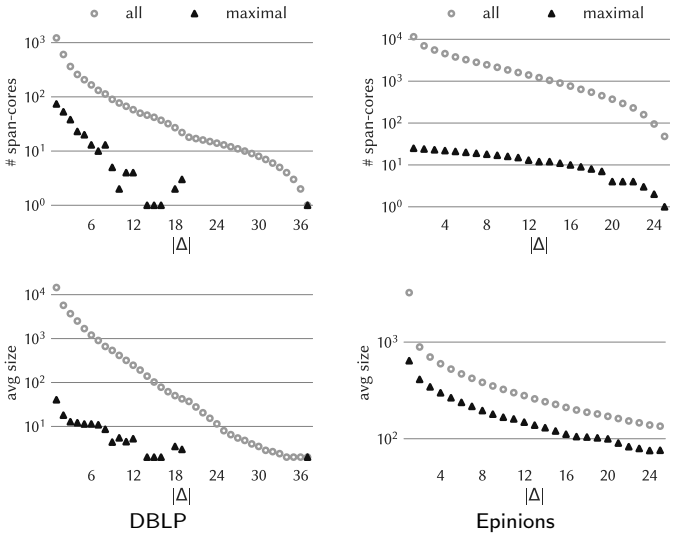

Figure 3: Top plots: number of span-cores and maximal span-cores as a function of the size of the temporal span $|\Delta|$. Bottom plots: average size of all span-cores and maximal span-cores as a function of the size of the temporal span $|\Delta|$.

Algorithm 3, where the penalty scores of the various intervals are computed from scratch during the execution of the algorithm, instead of precomputing them all via span-core decomposition. We refer to such a naïve method as Naïve-TCS.

The experimental setting we consider here is as follows. We vary the number $|Q|$ of query vertices from 1 to 3. In particular, when $|Q| = 1$, we sample the single query vertex uniformly at random from the whole vertex set V. Instead, for $|Q| > 1$, we employ a more sophisticated sampling strategy that aims at finding meaningful query-vertex sets, i.e., vertices interacting with each other during the temporal observations, and, at the same time, independent from the specific form of the resulting span-core decomposition. Specifically, the sampling strategy we use is based on an adaptation of random walk to the temporal settings:

- Select a vertex uniformly at random from the whole V and add such a vertex to the set Q_{visited} of visited vertices
- Starting from the first timestamp of the temporal domain T, iteratively:
  - With probability p, move the random walker to a neighbor of the current vertex and add the neighbor to Q_{visited}. If the current vertex has no neighbors in a given timestamp, the random walker jumps to the first next timestamp in which that vertex has at least one neighbor
  - With probability $1-p$, keep the random walker at the current vertex, but go to the next timestamp
- Restart if the last timestamp of T is reached
- Stop when $|Q_{\text{visited}}|$ reaches a proper (user-defined) size ν
- Sample $|Q|$ query vertices from Q_{visited} with probability proportional to the frequency of the visits during the random walk

In our experiments we set $p = 0.8$ and $\nu = 3|Q|$. As far as the number h of output communities, we consider the range $h \in [10, 20, 30, 40, 50, 60]$ on all datasets, with the exception of StackOverflow, for which we discard $h = 60$, and Epinions, for which we consider $h \in [4, 8, 12, 16, 20, 24]$. For every parameter configuration, we perform five runs of every algorithm (in every run we sample a different query-vertex set). Note that we were not able to run the algorithms for temporal community search on the WikiTalk dataset due to memory constraints.

Running time. In Figure 4 we show the running time of the proposed algorithms as a function of the number h of output communities, for the HighSchool, DBLP, Wikipedia, and Amazon datasets. The first general observation we make is that the running time of all algorithms increases as h gets higher. This in accordance with the time-complexity analysis reported in Section 6. Also, running times are independent of the selected query-vertex set Q. Looking at the individual performance, we notice that, as expected, the Naïve-TCS method has severe limitations in terms of efficiency: it takes hours to run on the HighSchool

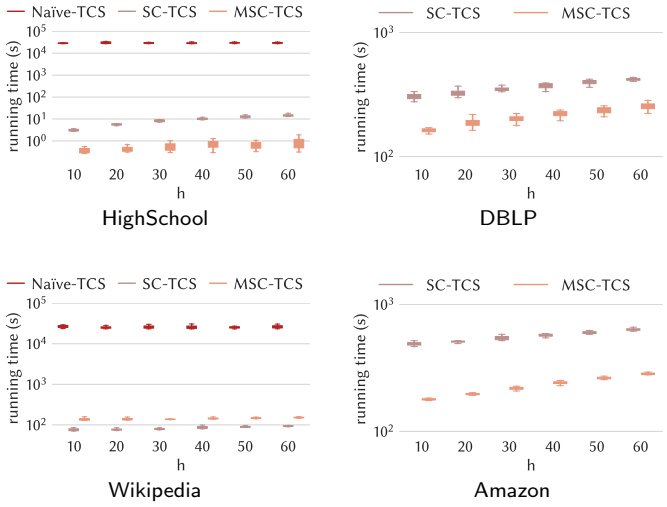

Figure 4: Running time of the algorithms for Temporal Community Search, as a function of the number h of output communities. Each boxplot corresponds to 15 data points.

and Wikipedia datasets, while it is not able to terminate in less than 10 days on the remaining datasets. SC-TCS and MSC-TCS are much faster than Naïve-TCS, achieving a speedup of up to more than four orders of magnitude. MSC-TCS is in most cases faster than SC-TCS, with speedup up to one order of magnitude (on HighSchool, for $h = 60$). This confirms that the exploitation of the maximal span-cores is effective in both shortening the precomputation time and reducing the temporal domain considered in the dynamic-programming step. The only exception is the Wikipedia dataset. To dive deeper into the motivations of this exception, we report in Figure 5 the split of the average running time of SC-TCS and MSC-TCS into the time spent in the dynamic-programming step (DP) (which also includes the identification of the reduced temporal domain T^* for MSC-TCS), and the precomputation time (i.e., the time required for computing all penalty scores via span-core decomposition or maximal span-cores). Interestingly, what affects the most the running time is the precomputation of the scores. Apparently, the Q-constrained version of Span-cores is more efficient than Maximal-span-cores in some datasets, which we believe is due to the structure of the search space. On the other hand, these results confirm that the reduction of the temporal domain considered by the dynamic-programming step is actually effective since the DP running time of MSC-TCS is always less than (or equal to) the DP running time of SC-TCS.

Greedy-minimum-community-search. Here we evaluate the performance of the proposed Greedy-minimum-community-search algorithm (Algorithm 5) for reducing the size of the output communities. We recall that the proposed algorithms for Temporal Community Search (evaluated above) output communities

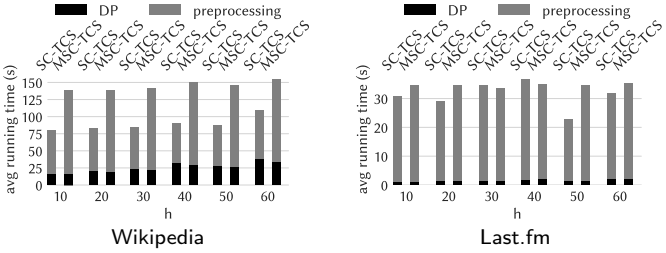

Figure 5: Split of the average running time of the SC-TCS and MSC-TCS algorithms into dynamic programming (DP) and precomputation, for the Wikipedia and Last.fm datasets.

corresponding to the (Q, Δ_i)-highest-order-span-cores for all $\{\Delta_i\}_{i=1}^{h}$ temporal intervals identified. The Greedy-minimum-community-search algorithm takes every (Q, Δ_i)-highest-order-span-core and attempts to reduce its size, while preserving optimality. Thus, the ultimate goal of the evaluation presented next is to show how well Greedy-minimum-community-search is able to reduce the size of the original span-cores, and what is its overhead in terms of running time.

Figure 6 compares the size of the starting (Q, Δ_i)-highest-order-span-cores and the size of the corresponding reduced community yielded by the Greedy-minimum-community-search algorithm, for the PrimarySchool, HongKong, Last.fm, and Epinions datasets. It can be easily observed that, as a general trend, the reduced communities are much smaller than the original ones, in all datasets, up to four orders of magnitude. The results on the Epinions dataset are a bit different than the other three datasets. In fact, on that dataset, the original communities (CS) always include the whole 120k vertices of the graph, while the communities found by Greedy-minimum-community-search (minimum CS) have median size smaller than 10, and, in many cases, they correspond to communities composed of the query vertices only. This means that, on the Epinions dataset, for our tested queries, the algorithms for Temporal Community Search do not extract communities that are really cohesive around the query vertices. This way, the benefits of exploiting an a-posteriori community-size-reduction step are less evident. Also, we do not notice any evident pattern as a function of h, for any dataset.

In Table 3 we report the average running time of an execution of Greedy-minimum-community-search, for all datasets. Note that this is the average time required to process one of the h communities in a solution to Temporal Community Search. Greedy-minimum-community-search runs in 8 seconds or less in all tested datasets. Therefore, the additional running time required by the algorithm is rather negligible.

To summarize, Greedy-minimum-community-search is empirically recognized as a powerful post-processing method for improving the quality of the solutions to Temporal Community Search: it finds much smaller communities at a very small additional computational cost.

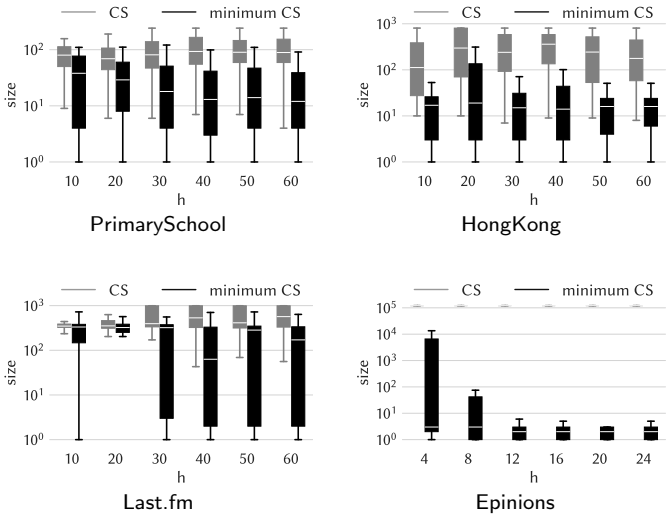

Figure 6: Comparison of the size of the communities in the solutions to Temporal Community Search: original output of the algorithms for Temporal Community Search (CS) and after running the Greedy-minimum-community-search algorithm on top of them (minimum CS). Each boxplot corresponds to 15 data points.

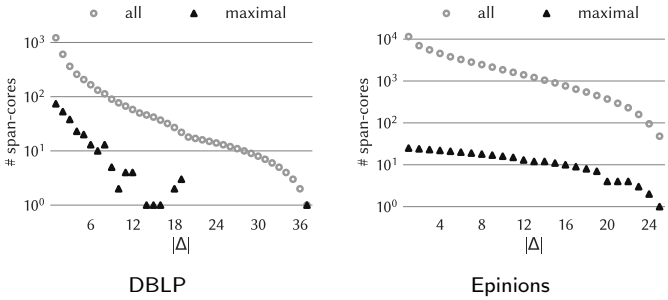

Figure 7: DP domain

8 Applications

In this section we illustrate applications of (maximal) span-cores in the analysis of face-to-face interaction networks, and how the methods for Temporal Community Search can be profitably exploited in a task of graph classification. For these applications we use the three networks gathered in schools, i.e., PrimarySchool, HighSchool, and HongKong, which are described above, at the beginning of Section 7. We use a window size of 5 minutes and, in the analysis, we discard span-cores of $|\Delta| = 1$, i.e., having span of 5 minutes, since they represent short interactions, not significant for our purposes. In the following we show (*i*) three types of interesting temporal patterns (Section 8.1), i.e., social activities of groups of students within a school day, mixing of gender and class, and length of social interactions in groups; (*ii*) a procedure to detect anomalous contacts and intervals that exploits maximal span-cores (Section 8.2); and, (*iii*) an approach to graph classification based on temporal community search (Section 8.3).

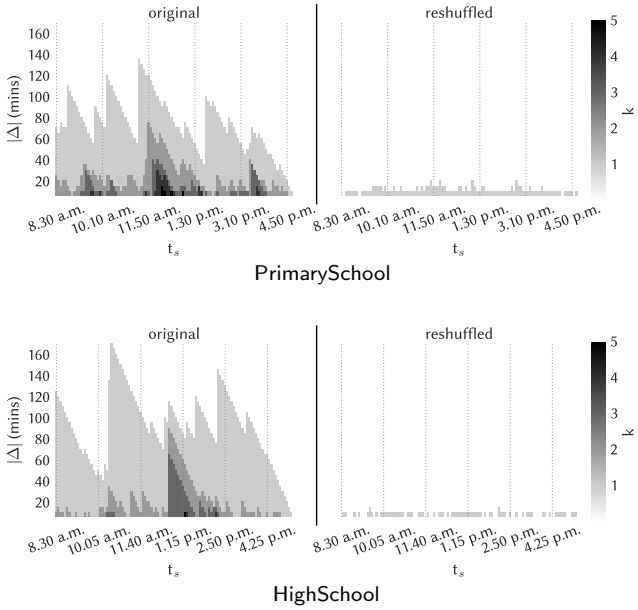

Figure 8: Temporal activity of a school day of the PrimarySchool and HighSchool datasets: the x axis reports the hour of the day at which the span of a span-core starts, the y axis specifies the size of the span (in minutes), and the color scale shows the order k. At a glance, it can be observed that the temporal structure of the span-core decomposition detects time-evolving cohesive structures in the original datasets (left plots) that completely disappear in the reshuffled datasets (right plots).

8.1 Temporal patterns

Temporal activity. We first show how span-cores afford a simple temporal analysis of social activities of groups of people within a school day. The left side of Figure 8 reports colormaps of the order k of the span-cores as a function of their starting time t_s (x axis) and of the size of their temporal span $|\Delta|$ (y axis), for a school day of the PrimarySchool and HighSchool datasets. Darker gray indicates span-cores of high order and slots located in the upper part of the plots refer to span-cores of long span. It is important to notice that the linear decay in span duration is naturally due to the definition of span-core and to the shifting of the starting time t_s; therefore, it is not a distinguishing feature of the activity patterns found in the analyzed data. In both datasets, fluctuations of k and $|\Delta|$ are observed along the day, which can be related to school events. Around 10 a.m., the size of the span $|\Delta|$ reaches a local maximum in correspondence to the morning break, which means that students establish long-lasting interactions that hold beyond the break itself. Moreover, when classes gather for the lunch break, the order k reaches its maximum value since students tend to form larger and more cohesive groups.

In order to verify that these results are not trivially derived from the general temporal activity, as simply given by the number of interactions in each timestamp, we compare our findings to a null model. At each timestamp of the temporal graphs, we reshuffle the edges by the Maslov-Sneppen algorithm [60] which consists in repeating the following operations up to when all edges have been processed: select at random two edges with no common vertices, e.g., (u, v) and (w, z), and transform them into (u, z) and (w, v), if neither (u, z) and (w, v) existed in the original timestamp. This reshuffling preserves the degree of each vertex in each timestamp and the global activity (i.e., the number of contacts per timestamp), but destroys correlations between edges of successive timestamps. In the right side of Figure 8 we show the results of the temporal analysis described above for the reshuffled datasets. In both, the values of $|\Delta|$ and k reached are much smaller than in the original datasets. The size of the span $|\Delta|$ is always shorter than 20 minutes, while in the original datasets it is much longer, up to 170 minutes, and the order k is always equal to 1, compared to the original maximum of 5. The time-evolving cohesive structures detected by the temporal core decomposition in the original datasets are completely lost on reshuffling, since only span-cores of short span and low coreness are observed in the latter case. This shows that the temporal structure exposed by the span-core decomposition is not simply a consequence of temporal patterns of global activity but that span-cores represent a concrete method to detect complex cohesive structures and their temporal evolution.

Mixing patterns. We now show an analysis of mixing patterns of students with respect to gender and class. Such vertex attributes are indeed available for the individuals of the PrimarySchool dataset. We define as *gender purity* of a span-core the fraction of individuals of the most represented gender within the span-core. *Class purity* is analogously defined. The left plot of Figure 9 reports the temporal evolution of the average gender and class purity of the

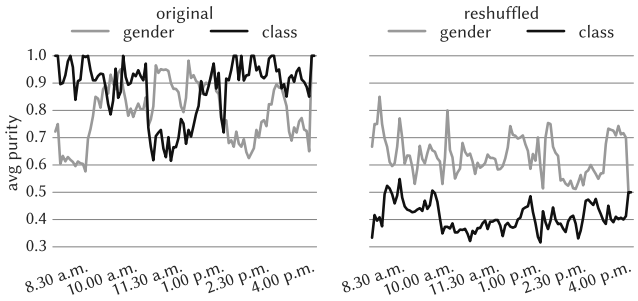

Figure 9: Temporal evolution (time on the x axis) of average gender purity and average class purity (y axis) of the maximal span-cores of the PrimarySchool dataset. Original data on the left, reshuffled data on the right.

maximal span-cores spanning each timestamp, during the first school day of the PrimarySchool dataset. During lessons, when students are in their own classes, class purity has naturally very high values, very close to 1. Gender purity is instead rather low. On the other hand, when students are gathered together, during the morning break at 10 a.m. and the lunch break between 12 a.m. and 2 p.m., the situation is overturned: gender purity reaches large values while class purity drastically decreases. This shows that primary school students group with individuals of the same class, disregarding the gender, only when they are forced by the schedule of the lessons, but prefer on average to form cohesive groups with students of the same gender during breaks. This is in agreement and complements a previous study of the same dataset focusing on single interactions in the static aggregated network [75].

The right plot of Figure 9 shows the temporal evolution of the average gender and class purity for a null model in which gender and class are randomly reshuffled among individuals. The two curves are more flat and the anti-correlation between them completely vanishes. This testifies that the results on the original dataset are not simply due to the relative abundance of individuals of each type interacting at each time, but reflect genuine mixing patterns and their temporal evolution.

Interaction length. Finally, we analyze the duration of interactions of social groups in schools by studying the distribution of the size of the span of the maximal span-cores of the three datasets (Figure 10). All distributions are extremely skewed with broad tails: most maximal span-cores have duration less than 1 hour, but durations much larger than the average can also be observed. Interestingly, the three datasets at hand all exhibit the same functional shape, confirming a robust statistical behavior. We also note that similar robust broad distributions have been observed for simpler characteristics of human interactions such as the statistics of contact durations [76, 61]. Outliers appear also at very large durations, especially for the HongKong dataset that has maximal span-cores lasting up to 83 hours. Group interactions of such long span are

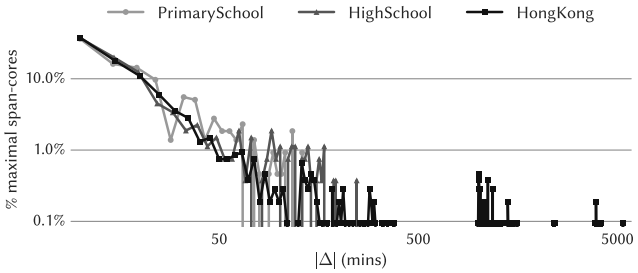

Figure 10: Distribution of the size of the span $|\Delta|$ of the maximal span-cores. The x axis reports the size of the span (in minutes), while the y axis the percentage of maximal span-cores having a given size of the span.

clearly abnormal and represent outliers in the distributions. We will show, in the following of this section, how to exploit such outliers to detect both irregular interactions and anomalous temporal intervals.

8.2 Anomaly detection

The identification of anomalous behaviors in temporal networks has been the focus of several studies in the last few years [63, 71]. Based on the above findings, we devise a simple procedure to detect anomalous edges and intervals of the HongKong dataset that exploits maximal span-cores. The topmost plot of Figure 11 reports the number of edges for each timestamp of the original HongKong dataset. It is easy to notice that there is a lot of constant anomalous activity between school days and during the weekend, i.e., days six and seven: unexpectedly, the number of interactions per timestamp does not drop to zero. This happened in fact because proximity sensors were left in each class and close to each other, at the end of the lessons. In order to automatically detect these steady activity patterns that do not correspond to any genuine social dynamics, we apply the following procedure: (*i*) find a set of anomalously long temporal intervals supporting maximal span-cores, (*ii*) identify anomalous vertices, and, (*iii*) filter out anomalous edges.

The first step of this procedure requires to find the set of temporal intervals $\mathcal{I} = \{\Delta \sqsubseteq T \mid C_{k,\Delta} \in \mathbf{C}_M \wedge |\Delta| > tr\}$ that are the span of a maximal span-core $C_{k,\Delta}$ with size longer than a certain threshold tr. Then, for each timestamp $t \in T$, select as anomalous all those vertices that appear in the span-cores $\{C_{1,\Delta} \mid \Delta \in \mathcal{I} \wedge t \in \Delta\}$, i.e., the span-cores of $k = 1$ whose span is in $\mathcal{I}$ and contains t. Finally, at each timestamp $t \in T$, remove edges that are incident to at least a vertex that has been marked as anomalous at time t. Consistently with the distribution of the span durations of the maximal span-cores, we select the threshold $tr = 22$ (110 minutes). The results of this filtering procedure are shown in the middle plot of Figure 11. The number of edges during school days remains approximately unchanged, while the activity noticeably decreases

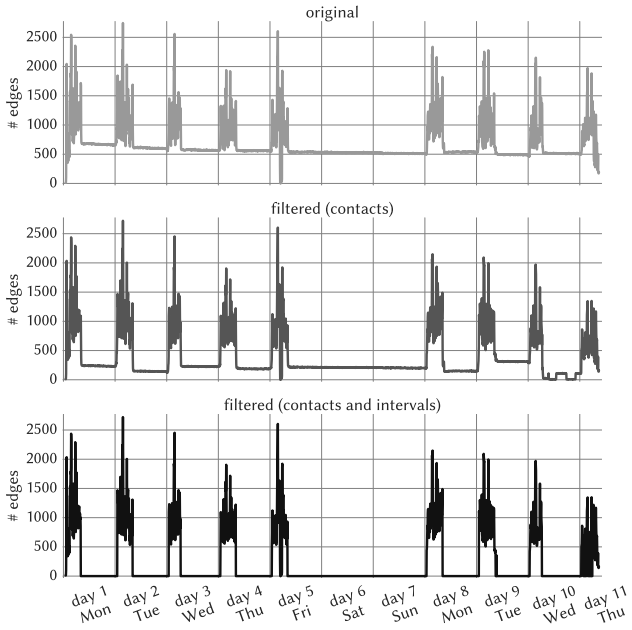

Figure 11: HongKong dataset: number of edges per timestamp in the original data (top), after filtering anomalous edges (middle), and after filtering anomalous edges and intervals (bottom). Days 6 and 7 are weekend.

in-between. Identifying as positives the spurious interactions occurring when the school is closed and as negatives the genuine interactions observed when the school is open, this approach achieves a precision of 0.91 and a recall of 0.64.

We can refine this anomaly detection process by identifying, in addition to anomalous edges, also anomalous temporal intervals. We define a timestamp $t \in T$ as anomalous if the ratio between the number of original edges (top plot of Figure 11) and the number of filtered edges (middle plot of Figure 11) exceeds a given threshold. We apply this further filtering to the HongKong dataset with a threshold of 1.5 and report the results in the bottommost plot of Figure 11. The number of edges when the school is closed drops to zero, while the activity during school days is not modified, except for the last one, which is affected by the proximity to the end of the time domain. The overall procedure yields a slightly higher value of precision, 0.93, and substantially improves the recall to 0.99.

8.3 Graph embedding and (supervised) vertex classification

In this subsection we show how Temporal Community Search can be profitably exploited for classifying the vertices of a temporal graph. Specifically,

the classification framework we set up is based on the paradigm of *graph embedding*, which has attracted a great deal of attention in the last few years, and whose goal is to assign to every vertex of a graph a numerical vector (i.e., an *embedding*) such that structurally similar vertices are represented by similar vectors, and vice versa [41, 26, 40]. Here, our framework simply consists in learning suitable embeddings for the vertices of the input graph, and then give them as input to some (well-established) classifier to ultimately accomplish the desired classification task. Thus, the main goal is to learn embeddings that are well-representative of the relationships among vertices, so as to help the classifier perform accurately. As our main result here, we show how an embedding strategy based on a simple exploitation of the output of Temporal Community Search achieves results comparable to well-established vertex-embedding methods such as DeepWalk [65], LINE [77], and node2vec [41].

Method. For every vertex of the input temporal graph, we build an embedding as an h-dimensional vector conveying the information provided by a solution to the Temporal Community Search problem on the same graph. Specifically, consider a vertex $u \in V$ and a solution $\{\langle S_i, \Delta_i \rangle\}_{i=1}^{h}$ to Temporal Community Search on query-vertex set $Q = \{u\}$. We define u's embedding as

$$\mathbf{X}_u = [v^*_{Q,\Delta_1}, v^*_{Q,\Delta_2}, \ldots, v^*_{Q,\Delta_h}],$$

which corresponds to the temporally-ordered sequence of minimum degrees of the h communities identified by the temporal-community-search solution. Below we show that this simple approach is sufficient to achieve interesting experimental results. Clearly, more sophisticated methods are possible, e.g., by simultaneously exploiting information from the S_i communities. However, our main goal here is to give an idea of how the Temporal Community Search problem can be successfully leveraged in a relevant application scenario, rather than devise the best temporal-community-search-based graph-embedding method.

Evaluation. We assess the performance of our method on the PrimarySchool and HighSchool datasets. In these datasets vertices correspond to students, and vertex labels (to be predicted) are the classes that every student belongs to. We involve in the comparison the following state-of-the-art vertex-embedding methods:

- DeepWalk [65], a method that preserves the proximity between vertices by running a set of random walks and maximizing the sum of the log-likelihood of a set of vertices for each walk.
- LINE [77], which optimizes a suitable objective function preserving both first-order (one-hop) and second-order (two-hop) proximities. Neighborhoods are not explored via random walk, but in a breadth-first fashion.
- node2vec [41], which is based on the same idea underlying DeepWalk, but allowing more flexibility on how random walks explore and leave the neighborhood of the current vertex.

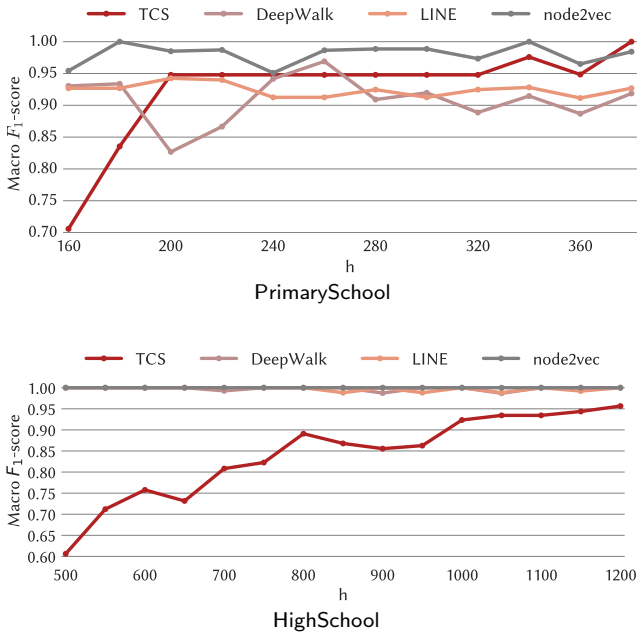

Figure 12: Graph classification: Macro F_1-score of the proposed temporal-community-search-based graph-embedding method TCS and the competing methods, with varying the dimensionality h of the output embeddings, on the PrimarySchool and HighSchool datasets.

These three methods consider non-temporal graphs. Therefore, we feed them with aggregated graphs in which every edge exists if it exists in at least one timestamp. We tune the parameters p and q of node2vec as in the original paper [41], i.e., by performing a grid search with $p, q \in \{0.25, 0.50, 1, 2, 4\}$, while keeping the dimensionality of the embeddings fixed to $h = 200$ and $h = 625$, for the PrimarySchool and HighSchool datasets, respectively. The other competing methods, DeepWalk and LINE, and our method based on temporal community search (which we refer to as TCS in the following) do not have parameters (apart from the dimensionality h of the output embeddings). After filtering out those vertices representing the teachers, we partition the remaining vertices (i.e., the students) into training and test sets with an 80-20 split. A standard scaler is applied to the features extracted by each embedding method and, then, a penalized logistic-regression classifier is trained.

In Figure 12 we report classification results in terms of Macro F_1-score, with varying the dimensionality h of the embeddings. On the PrimarySchool dataset, for $h \geq 200$, our TCS has performance close to 1 in terms Macro F_1-score, similarly to the three baselines. It can be observed that the TCS results are better as h gets higher; in particular, TCS is even better than node2vec for $h = |T|$. This is expected and is motivated as, for higher h, TCS is allowed to rely on more temporal information about the vertices. On the HighSchool dataset, TCS is outperformed by all methods for smaller h. However, again, the performance of TCS becomes competitive for larger h, up to achieving comparable results to the best method(s) for $h = |T|$.

9 Conclusions

Temporal networks are a powerful representation of how relations are established and interrupted along time among a given population of entities. An interesting primitive for analyzing this type of networks is the extraction of relevant patterns, such as dense subgraphs, together with their time interval of existence (or span). Following this idea, we introduced in this paper a notion of temporal core decomposition where each core is associated with its span. Exploiting containment properties among cores we developed efficient algorithms for computing all the span-cores, and also only the maximal ones. We then introduced the problem of temporal community search and showed how it can be solved in polynomial time via dynamic programming. We also proved an interesting connection between temporal community search and maximal span-cores, which made it possible to devise a considerably more efficient algorithm than the naïve dynamic-programming one. Finally, we presented applications on empirical networks of human close-range proximity, that illustrate the relevance of the notions of (maximal) span-core and temporal community search in a variety analyses and applications.

In future work we will study the role of maximal span-cores with large coreness and/or $|\Delta|$ in spreading processes on temporal networks. Furthermore, span-cores represent features that can be used for network fingerprinting and

classification as well as for model validation, and that could provide support for new ways of visualizing large-scale time-varying graphs. Finally, future work will include further detailed investigations of the application of span-cores in the presented case studies.

References

[1] Leman Akoglu, Duen Horng Chau, Christos Faloutsos, Nikolaj Tatti, Hanghang Tong, Jilles Vreeken, and LAJVH Tong. Mining connection pathways for marked nodes in large graphs. In *SDM*, pages 37–45, 2013.

[2] J Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in neural information processing systems*, pages 41–50, 2006.

[3] Reid Andersen and Kumar Chellapilla. Finding dense subgraphs with size bounds. In *WAW*, 2009.

[4] Reid Andersen and Kevin J. Lang. Communities from seed sets. In *WWW*, 2006.

[5] Albert Angel, Nikos Sarkas, Nick Koudas, and Divesh Srivastava. Dense subgraph maintenance under streaming edge weight updates for real-time story identification. *PVLDB*, 5(6), 2012.

[6] Gary D. Bader and Christopher W. V. Hogue. An automated method for finding molecular complexes in large protein interaction networks. *BMC Bioinformatics*, 4:2, 2003.

[7] Nicola Barbieri, Francesco Bonchi, Edoardo Galimberti, and Francesco Gullo. Efficient and effective community search. *DAMI*, 29(5):1406–1433, 2015.

[8] V. Batagelj and M. Zaveršnik. Fast algorithms for determining (generalized) core groups in social networks. *ADAC*, 5(2), 2011.

[9] Vladimir Batagelj, Andrej Mrvar, and Matjaz Zaversnik. Partitioning approach to visualization of large graphs. In *Int. Symp. on Graph Drawing*, pages 90–97, 1999.

[10] R. Bellman. On the approximation of curves by line segments using dynamic programming. *Commun. ACM*, 4(6):284–, 1961.

[11] Matthias Bentert, Anne-Sophie Himmel, Hendrik Molter, Marco Marik, Rolf Niedermeier, and René Saitenmacher. Listing all maximal k-plexes in temporal graphs. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 41–46. IEEE, 2018.

[12] Michele Berlingerio, Francesco Bonchi, Björn Bringmann, and Aristides Gionis. Mining graph evolution rules. In *ECML PKDD 2009*.

[13] Yuchen Bian, Yaowei Yan, Wei Cheng, Wei Wang, Dongsheng Luo, and Xiang Zhang. On multi-query local community detection. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 9–18. IEEE, 2018.

[14] Petko Bogdanov, Misael Mongiovì, and Ambuj K Singh. Mining heavy subgraphs in time-evolving networks. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 81–90. IEEE, 2011.

[15] Francesco Bonchi, Ilaria Bordino, Francesco Gullo, and Giovanni Stilo. Identifying buzzing stories via anomalous temporal subgraph discovery. In *WI 2016*, 2016.

[16] Francesco Bonchi, Francesco Gullo, Andreas Kaltenbrunner, and Yana Volkovich. Core decomposition of uncertain graphs. In *KDD*, pages 1316–1325, 2014.

[17] Björn Bringmann, Michele Berlingerio, Francesco Bonchi, and Aristides Gionis. Learning and predicting the evolution of social networks. *IEEE Intelligent Systems*, 25(4):26–35, 2010.

[18] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In *APPROX*, 2000.

[19] Moses Charikar, Yonatan Naamad, and Jimmy Wu. On finding dense common subgraphs. *arXiv preprint arXiv:1802.06361*, 2018.

[20] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. Efficient core decomposition in massive networks. In *ICDE*, pages 51–62, 2011.

[21] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.

[22] Anish Das Sarma, Alpa Jain, and Cong Yu. Dynamic relationship and event discovery. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 207–216. ACM, 2011.

[23] Elise Desmier, Marc Plantevit, Céline Robardet, and Jean-François Boulicaut. Cohesive co-evolution patterns in dynamic attributed graphs. In *International Conference on Discovery Science*, pages 110–124. Springer, 2012.

[24] Marius Eidsaa and Eivind Almaas. s-core network decomposition: A generalization of k-core analysis to weighted networks. *Phys. Rev. E*, 88:062819, Dec 2013.

[25] Alessandro Epasto, Silvio Lattanzi, and Mauro Sozio. Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*, pages 300–310. International World Wide Web Conferences Steering Committee, 2015.

[26] Alessandro Epasto and Bryan Perozzi. Is a single embedding enough? learning node representations that capture multiple social contexts. In *The World Wide Web Conference*, pages 394–404. ACM, 2019.

[27] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. In *ISAAC*, 2010.

[28] Péter Érdi, Kinga Makovi, Zoltán Somogyvári, Katherine Strandburg, Jan Tobochnik, Péter Volf, and László Zalányi. Prediction of emerging technologies based on analysis of the us patent citation network. *Scientometrics*, 95(1):225–242, 2013.

[29] Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. Fast discovery of connection subgraphs. In *KDD*, 2004.

[30] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. Effective and efficient attributed community search. *The VLDB JournalThe International Journal on Very Large Data Bases*, 26(6):803–828, 2017.

[31] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. Effective community search over large spatial graphs. *Proceedings of the VLDB Endowment*, 10(6):709–720, 2017.

[32] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. A survey of community search over big graphs. *arXiv preprint arXiv:1904.12539*, 2019.

[33] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. Mining (maximal) span-cores from temporal networks. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 107–116. ACM, 2018.

[34] Edoardo Galimberti, Francesco Bonchi, and Francesco Gullo. Core decomposition and densest subgraph in multilayer networks. In *CIKM 2017*, 2017.

[35] Antonios Garas, Frank Schweitzer, and Shlomo Havlin. A k -shell decomposition method for weighted networks. *New Journal of Physics*, 14(8):083030, 2012.

[36] David Garcia, Pavlin Mavrodiev, and Frank Schweitzer. Social resilience in online communities: The autopsy of friendster. *CoRR*, abs/1302.6109, 2013.

[37] Laetitia Gauvin, André Panisson, and Ciro Cattuto. Detecting the community structure and activity patterns of temporal networks: a non-negative tensor factorization approach. *PLOS ONE*, 9(1):e86028, 2014.

[38] Valerio Gemmetto, Alain Barrat, and Ciro Cattuto. Mitigation of infectious disease at school: targeted class closure vs school closure. *BMC infectious diseases*, 14(1):695, December 2014.

[39] Christos Giatsidis, Dimitrios M. Thilikos, and Michalis Vazirgiannis. D-cores: measuring collaboration of directed graphs based on degeneracy. *KAIS*, 35(2):311–343, 2013.

[40] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78 – 94, 2018.

[41] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[42] John Healy, Jeannette Janssen, Evangelos E. Milios, and William Aiello. Characterization of graphs using degree cores. In *WAW*, 2006.

[43] Anne-Sophie Himmel, Hendrik Molter, Rolf Niedermeier, and Manuel Sorge. Enumerating maximal cliques in temporal graphs. In *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, pages 337–344. IEEE, 2016.

[44] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, pages 1311–1322, 2014.

[45] Xin Huang and Laks VS Lakshmanan. Attribute-driven community search. *Proceedings of the VLDB Endowment*, 10(9):949–960, 2017.

[46] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. Community search over big graphs: Models, algorithms, and opportunities. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1451–1454. IEEE, 2017.

[47] Akihiro Inokuchi and Takashi Washio. Mining frequent graph sequence patterns induced by vertices. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 466–477. SIAM, 2010.

[48] Vinay Jethava and Niko Beerenwinkel. Finding dense subgraphs in relational graphs. In *ECML-PKDD*, pages 641–654, 2015.

[49] M. Kitsak, L. K. Gallos, S. Havlin, F. Liljeros, L. Muchnik, H. E. Stanley, and H. A. Makse. Identifying influential spreaders in complex networks. *Nature Physics 6, 888*, 2010.

[50] Isabel M. Kloumann and Jon M. Kleinberg. Community membership identification from small seed sets. In *KDD*, 2014.

[51] Guy Kortsarz and David Peleg. Generating sparse 2-spanners. *J. Algorithms*, 17(2):222–236, 1994.

[52] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics*, page P11005, 2011.

[53] Cane Wing-ki Leung, Ee-Peng Lim, David Lo, and Jianshu Weng. Mining interesting link formation rules in social networks. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 209–218. ACM, 2010.

[54] Rong-Hua Li, Jiao Su, Lu Qin, Jeffrey Xu Yu, and Qiangqiang Dai. Persistent community search in temporal networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 797–808. IEEE, 2018.

[55] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465, 2014.

[56] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. Efficient (a, β)-core computation: an index-based approach. In *The World Wide Web Conference*, pages 1130–1141. ACM, 2019.

[57] Paul Liu, Austin R Benson, and Moses Charikar. Sampling methods for counting temporal motifs. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 294–302. ACM, 2019.

[58] Shuai Ma, Renjun Hu, Luoshu Wang, Xuelian Lin, and Jinpeng Huai. Fast computation of dense temporal subgraphs. In *ICDE*, pages 361–372, 2017.

[59] Fragkiskos Malliaros, Christos Giatsidis, Apostolos Papadopoulos, and Michalis Vazirgiannis. The core decomposition of networks: Theory, algorithms and applications. 2019.

[60] Sergei Maslov and Kim Sneppen. Specificity and stability in topology of protein networks. *Science*, 296(5569):910–913, 2002.

[61] Rossana Mastrandrea, Julie Fournet, and Alain Barrat. Contact patterns in a high school: A comparison between data collected using wearable sensors, contact diaries and friendship surveys. *PLoS ONE*, 10(9):1–26, 09 2015.

[62] David W. Matula and Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 30(3):417–427, 1983.

[63] Misael Mongiovi, Petko Bogdanov, Razvan Ranca, Evangelos E Papalexakis, Christos Faloutsos, and Ambuj K Singh. Netspot: Spotting significant anomalous regions on dynamic networks. In *Proceedings of the 2013 SIAM International Conference on Data Mining*, pages 28–36. SIAM, 2013.

[64] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *TPDS*, 24(2):288–300, 2013.

[65] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[66] Alexander Reinthal, Arvid Andersson, Erik Norlander, Philip Stålhammar, Sebastian Norlin, et al. Finding the densest common subgraph with linear programming. 2016.

[67] Polina Rozenshtein, Francesco Bonchi, Aristides Gionis, Mauro Sozio, and Nikolaj Tatti. Finding events in temporal networks: Segmentation meets densest-subgraph discovery. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 397–406. IEEE, 2018.

[68] Polina Rozenshtein, Nikolaj Tatti, and Aristides Gionis. Finding dynamic dense subgraphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(3):27, 2017.

[69] Natali Ruchansky, Francesco Bonchi, David García-Soriano, Francesco Gullo, and Nicolas Kourtellis. To be connected, or not to be connected: That is the minimum inefficiency subgraph problem. In *CIKM 2017*.

[70] Natali Ruchansky, Francesco Bonchi, David García-Soriano, Francesco Gullo, and Nicolas Kourtellis. The minimum wiener connector problem. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1587–1602. ACM, 2015.

[71] Anna Sapienza, André Panisson, Joseph Wu, Laetitia Gauvin, and Ciro Cattuto. Detecting anomalies in time-varying networks using tensor decomposition. In *Data Mining Workshop (ICDMW), 2015 IEEE International Conference on*, pages 516–523. IEEE, 2015.

[72] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. Streaming algorithms for k-core decomposition. *PVLDB*, 6(6):433–444, 2013.

[73] Konstantinos Semertzidis, Evaggelia Pitoura, Evimaria Terzi, and Panayiotis Tsaparas. Finding lasting dense subgraphs. *Data Mining and Knowledge Discovery*, Nov 2018.

[74] Mauro Sozio and Aristides Gionis. The community-search problem and how to plan a successful cocktail party. In *KDD*, 2010.

[75] Juliette Stehlé, Franois Charbonnier, Tristan Picard, Ciro Cattuto, and Alain Barrat. Gender homophily from spatial behavior in a primary school: A sociometric study. *Social Networks*, 35:604–613, 2013.

[76] Juliette Stehlé, Nicolas Voirin, Alain Barrat, Ciro Cattuto, Lorenzo Isella, Jean-Franois Pinton, Marco Quaggiotto, Wouter Van den Broeck, Corinne Régis, Bruno Lina, and Philippe Vanhems. High-resolution measurements of face-to-face contact patterns in a primary school. *PLoS ONE*, 6(8):e23176, 08 2011.

[77] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[78] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.

[79] Tiphaine Viard, Matthieu Latapy, and Clémence Magnien. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609:245–252, 2016.

[80] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. Core decomposition in large temporal graphs. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 649–658. IEEE, 2015.

[81] Stefan Wuchty and Eivind Almaas. Peeling the yeast protein network. *Proteomics*, 5(2):444–449, 2005.

[82] Yaowei Yan, Yuchen Bian, Dongsheng Luo, Dongwon Lee, and Xiang Zhang. Constrained local graph clustering by colored random walk. In *The World Wide Web Conference*, pages 2137–2146. ACM, 2019.

[83] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. When engagement meets similarity: efficient (k, r)-core computation on social networks. *Proceedings of the VLDB Endowment*, 10(10):998–1009, 2017.

[84] Haohua Zhang, Hai Zhao, Wei Cai, Jie Liu, and Wanlei Zhou. Using the k-core decomposition to analyze the static structure of large-scale software systems. *J. Supercomputing*, 53(2):352–369, 2010.

[85] Philipp Zschoche, Till Fluschnik, Hendrik Molter, and Rolf Niedermeier. The complexity of finding small separators in temporal graphs. In *43rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Algorithm 4: Efficient-temporal-community-search

Input: A temporal graph $G = (V, E, T)$, a set $Q \subseteq V$ of query vertices, an integer $h \in \mathbb{N}^+$.

Output: A set $\{\langle S_i, \Delta_i \rangle\}_{i=1}^{h}$, where $Q \subseteq S_i \subseteq V$, $\forall 1 \leq i \leq h$, and $\{\Delta_i\}_{i=1}^{h}$ is a partition of T.

```
/* Identification of T*                                                    */
```

Compute the set $\mathbf{C}_M(Q)$ of Q-constrained maximal span-cores of G

$\mathbf{D} \leftarrow \{\Delta \sqsubseteq T \mid C_{k,\Delta} \in \mathbf{C}_M(Q)\}$

$T_{\mathbf{D}} \leftarrow \bigcup_{\Delta \in \mathbf{D}} \Delta$; $T^+_{\mathbf{D}} \leftarrow \{\min\{t_e+1, t_{max}\} \mid [t_s, t_e] \in \mathbf{D}\}$; $T^-_{\mathbf{D}} \leftarrow \{\max\{t_s-1, 0\} \mid [t_s, t_e] \in \mathbf{D}\}$

$T_{sup} \leftarrow \{t_i \in T \setminus (T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}) \mid i \in [1, h+1-|T_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup \{t_{max}\}|]\}$

$T^* \leftarrow T_{\mathbf{D}} \cup T^+_{\mathbf{D}} \cup T^-_{\mathbf{D}} \cup \{t_{max}\} \cup T_{sup}$

```
/* Initialization                                                          */
```

Compute $v^*_{Q,\Delta}$, $\forall \Delta \sqsubseteq T$

$\mathbf{M} \leftarrow$ mapping function $[0, |T^*|) \rightarrow T^*$

$\mathbf{P} \leftarrow$ an empty $(|T^*| \times h)$-dimensional matrix `// Penalty matrix`

$\mathbf{R} \leftarrow$ an empty $(|T^*| \times h)$-dimensional matrix `// Reconstruction matrix`

forall $r \in [0, |T^*|)$ **do**
- $\mathbf{P}[r, 0] \leftarrow -v^*_{Q,[0,\mathbf{M}[r]]}$
- $\mathbf{R}[r, 0] \leftarrow 0$

```
/* Dynamic-programming step                                                */
```

forall $r \in [0, |T^*|)$ **do**
- **forall** $i \in [1, h)$ **do**
  - $\mathbf{P}[r, i] \leftarrow \min_{\ell \in [0,r]} \mathbf{P}[\ell, i-1] - v^*_{Q,[\mathbf{M}[\ell+1],\mathbf{M}[r]]}$
  - $\mathbf{R}[r, i] \leftarrow \mathrm{argmin}_{\ell \in [0,r]} \mathbf{P}[\ell, i-1] - v^*_{Q,[\mathbf{M}[\ell+1],\mathbf{M}[r]]}$

```
/* Reconstruction of the solution                                          */
```

$ub \leftarrow |T^*| - 1$

forall $i \in (h, 0]$ **do**
- $lb \leftarrow \mathbf{R}[ub, i]$
- $\Delta_i \leftarrow [\mathbf{M}[lb], \mathbf{M}[ub]]$
- $ub \leftarrow lb - 1$

forall $i \in (h, 0]$ **do**
- $S_i \leftarrow C^*_{Q,\Delta_i}$

Algorithm 5: Greedy-minimum-community-search

Input: A temporal graph $G = (V, E, T)$, a set $Q \subseteq V$ of query vertices, an interval $\Delta \sqsubseteq T$, a subset of vertices $S^* \subseteq V$ containing all the solutions to Problem 4 on input $\langle G, Q, \Delta \rangle$.

Output: A subset S^*_{min} of vertices such that $Q \subseteq S^*_{min} \subseteq S^*$ and $\min_{u \in S^*_{min}} d_\Delta(S^*_{min}, u) \geq \min_{u \in S^*} d_\Delta(S^*, u)$.

```
S*_min ← ∅;   P ← ∅;   A ← ∅
add every q ∈ Q to P with priority +∞
k* ← min_{u∈S*} d_Δ(S*, u);   k*_min ← 0
while k*_min < k* or Q ⊈ S*_min do
    dequeue u from P
    S*_min ← S*_min ∪ {u}
    forall v ∈ neigh_Δ(S*, u) \ S*_min \ P do
        A[v] ← score(v)
        add v to P with priority A[v]
    forall v ∈ neigh_Δ(S*_min, u) do
        if d_Δ(S*_min, v) = k* then
            forall w ∈ neigh_Δ(P, v) do
                A[w] ← A[w] − 1
    k*_min ← min_{v∈S*_min} d_Δ(S*_min, v)
```

Table 1: Temporal graphs used in the experiments.

dataset	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert T \rvert$	window size	domain
HighSchool	327	47k	1212	5 mins	face-to-face
PrimarySchool	242	55k	390	5 mins	face-to-face
HongKong	806	2M	2976	5 mins	face-to-face
ProsperLoans	89k	3M	307	7 days	economic
Last.fm	992	4M	77	21 days	co-listening
WikiTalk	2M	10M	192	28 days	communication
DBLP	1M	11M	80	366 days	co-authorship
StackOverflow	2M	16M	51	56 days	question answering
Wikipedia	343k	18M	101	56 days	co-editing
Amazon	2M	22M	115	28 days	co-rating
Epinions	120k	33M	25	21 days	co-rating

Table 2: Evaluation of the proposed algorithms: number of output span-cores, running time, memory, and number of processed vertices.

dataset	method	# output span-cores	running time (s)	memory (GB)	# processed vertices
HighSchool	Naïve-span-cores	12 320	18	0.1	3M
	Span-cores		1	0.1	581k
	Naïve-maximal-span-cores	450	1	0.1	581k
	Maximal-span-cores		0.3	0.1	181k
PrimarySchool	Naïve-span-cores	4 703	4	0.1	818k
	Span-cores		0.6	0.1	174k
	Naïve-maximal-span-cores	409	0.6	0.1	174k
	Maximal-span-cores		0.1	0.1	63k
HongKong	Naïve-span-cores	2 367 743	85 180	1	819M
	Span-cores		18 389	0.8	216M
	Naïve-maximal-span-cores	1 807	18 641	0.8	216M
	Maximal-span-cores		339	0.5	212M
ProsperLoans	Naïve-span-cores	4 273	101	2	55M
	Span-cores		46	2	27M
	Naïve-maximal-span-cores	293	48	2	27M
	Maximal-span-cores		8	2	980k
Last.fm	Naïve-span-cores	126 819	707	0.5	2M
	Span-cores		199	0.5	531k
	Naïve-maximal-span-cores	1 670	202	0.5	531k
	Maximal-span-cores		57	0.5	271k
WikiTalk	Naïve-span-cores	19 693	322 302	36	25B
	Span-cores		1 084	36	555M
	Naïve-maximal-span-cores	632	1 194	36	555M
	Maximal-span-cores		126	35	2M
DBLP	Naïve-span-cores	6 135	10 506	11	1B
	Span-cores		278	11	150M
	Naïve-maximal-span-cores	268	292	11	150M
	Maximal-span-cores		116	11	620k
StackOverflow	Naïve-span-cores	1 238	5 360	10	1B
	Span-cores		245	10	127M
	Naïve-maximal-span-cores	129	245	10	127M
	Maximal-span-cores		128	10	3M
Wikipedia	Naïve-span-cores	125 191	17 155	4	1B
	Span-cores		522	4	35M
	Naïve-maximal-span-cores	2 147	537	4	35M
	Maximal-span-cores		201	4	320k
Amazon	Naïve-span-cores	29 318	10 415	18	2B
	Span-cores		409	18	247M
	Naïve-maximal-span-cores	303	580	18	247M
	Maximal-span-cores		123	18	688k
Epinions	Naïve-span-cores	63 111	699	4	39M
	Span-cores		186	4	3M
	Naïve-maximal-span-cores	320	201	4	3M
	Maximal-span-cores		154	5	129k

Table 3: Average running time of an execution of the Greedy-minimum-community-search algorithm.

	HighSchool	PrimarySchool	HongKong	ProsperLoans	Last.fm
running time (s)	0.003	0.001	0.02	0.3	0.06

	DBLP	StackOverflow	Wikipedia	Amazon	Epinions
running time (s)	7	8	1	7	6