

Adaptive Community Search in Dynamic Networks

Ioanna Tsalouchidou
Pompeu Fabra University
Barcelona Spain
ioannatsalouchidou@gmail.com

Francesco Bonchi
ISI Foundation, Turin, Italy
Eurecat, Barcelona, Spain
francesco.bonchi@isi.it

Ricardo Baeza-Yates
Northeastern University
Silicon Valley campus, CA, USA
rbaeza@acm.org

Abstract—Community search is a well-studied problem which, given a static graph and a query set of vertices, requires to find a cohesive (or dense) subgraph containing the query vertices. In this paper we study the problem of community search in temporal dynamic networks. We adapt to the temporal setting the notion of *network inefficiency* which is based on the pairwise shortest-path distance among all the vertices in a solution. For this purpose we define the notion of *shortest-fastest-path distance*: a linear combination of the temporal and spatial dimensions governed by a user-defined parameter. We thus define the MINIMUM TEMPORAL-INEFFICIENCY SUBGRAPH problem and show that it is NP-hard. We develop an algorithm which exploits a careful transformation of the temporal network to a static directed and weighted graph, and some recent approximation algorithm for finding the minimum Directed Steiner Tree. We finally generalize our framework to the streaming setting in which new snapshots of the temporal graph keep arriving continuously and our goal is to produce a community search solution for the temporal graph corresponding to a sliding time window.

I. INTRODUCTION

Community search, i.e., the problem of extracting cohesive subgraphs around a given set of vertices of interest, is a fundamental graph mining primitive which has received a great deal of attention [4], [7]. If extracted from dynamic graphs, these substructures can help understanding the dynamics of the relationships that exist among these vertices. For instance, in a research collaboration network, these structure can describe the dynamics of collaborations between a given group of researchers along time. As another example, the interactome, which is the set of molecular interactions in a cell, can be modeled as a network, in which the vertices are proteins and through their connections can perform biological functions. However, these connections are not constantly active, and therefore a dynamic analysis is more appropriate for understanding properly this complex network [10]. Other application examples include events detection, friendship recommendation, control of infectious disease, semantic expansion, just to mention a few. However, *surprisingly, little attention has been devoted to the community search problem in temporal networks*. In this paper we propose a framework for adaptive community search in dynamic temporal networks based on a model, defined for static graphs, by Ruchansky et al. [12]. Given a static graph $G = (V, E)$ and a set of query vertices $Q \subseteq V$, the problem defined in [12] requires to find a set of vertices $S \supseteq Q$ such that its induced subgraph minimize *network inefficiency*, a measure (formally defined in the next section) based on the pairwise shortest-path distance among

all the vertices in the subgraph. The characterizing features of the proposal by Ruchansky et al. are that (i) it is totally parameterless and (ii) solutions can be disconnected: this way it allows to detect outliers in the query set, by letting such vertices disconnected from the others in the produced solution subgraph. The latter is an important feature, as community search is an explorative data analysis task, in which we are given some query vertices which are *suspected to be interesting*. However, real-world query-sets are noisy and likely to contain some vertices that are erroneously suspected of being related. This feature is even more important in the temporal setting, in which vertices originally related might become unrelated along time due to concept drift.

In order to generalize the notion of network inefficiency to temporal networks, the first step is to generalize the notion of shortest-path distance. As our goal is to use distances as a measure of cohesiveness, it is important to consider how the vertices connect through paths both in space (network structure) and in time (network evolution). For this purpose we adopt the notion of *shortest-fastest-path distance* [13]: a linear combination of the temporal and spatial dimensions governed by a user-defined parameter $\alpha \in [0, 1]$. This definition generalizes both shortest and fastest path: by setting $\alpha = 1$ we obtain shortest paths, while setting $\alpha = 0$ we obtain fastest paths. In general, depending on the application at hand, one can tune the parameter α to give more importance to the temporal dimension ($\alpha < 0.5$) or the spatial one ($\alpha > 0.5$).

The next step is to define *Temporal Network Inefficiency* for which we have to keep in consideration that, due to the intrinsic directionality of time, distances are no longer symmetric like in static undirected graphs. We finally can introduce the main problem studied in this paper which is the problem of extracting the minimum temporal-inefficiency subgraph problem from a temporal graph. We show that our problem is a genuine generalization of that Ruchansky et al. [12]: in fact, for $\alpha = 1$ and in the case that the temporal graph is made of one single snapshot (it is thus not temporal), then the two problems exactly correspond.

Our proposed algorithm is structured in three phases. In the first phase the temporal graph is transformed in a static directed and weighted graph, by flattening all the temporal snapshots in a unique static graph and by linking the various replicas of the same vertex in different timestamps, and by appropriately weighting the links. We show that the resulting transformed graph is such that the shortest-path distance be-

tween two vertices on this graph, corresponds to the shortest-fastest-path distance between the same vertices in the original temporal network. This property allows us to compute our solutions on this transformed graph. In the second phase we look for a *connector* for the query vertices in Q , i.e., a set of vertices $S \supseteq Q$ such that in the induced subgraph all the vertices in Q are connected, to the maximum possible extent. For this task we extract minimum Directed Steiner Trees from transformed graph. In the third phase, we greedily reduce the solution S , returning as solution the subset which minimizes the temporal inefficiency.

Having an algorithm to extract the minimum temporal-inefficiency subgraph from a temporal graph defined over a fixed temporal window, our next step is to generalize our framework to the streaming setting. This is the setting in which new snapshots of the temporal graph keep arriving continuously and our goal is to produce, at each new timestamp, a community search solution for the temporal graph defined by a sliding temporal window of predefined size. Since the network changes constantly in time, we expect that the communities evolve as well. Therefore, it is natural that the query set Q is updated during the evolution, to keep in consideration the evidence of the key connections and most related vertices emerged during the previous time windows.

We provide a streaming distributed implementation of our framework in Apache Spark and experiment on several real-world networks gathered by a proximity-sensing infrastructure recording face-to-face interactions in schools and co-authorship networks.

II. BACKGROUND AND RELATED WORK

Given a graph $G = (V, E)$ and a set of query vertices $Q \subseteq V$, the community search problem requires to find a connected subgraph H of G , that contains all query vertices Q and that exhibits some nice properties of cohesiveness, compactness or density. While optimizing for different objective functions, the bulk of this literature (see [4], [7] for recent survey) shares a common aspect: the solution must be a *connected* subgraph of the input graph containing the set of query vertices. Three recent approaches allow disconnected solutions in community search: allowing disconnected solutions is equivalent to allow some query vertices not to participate in the solution, thus being recognized as outliers. We call this version of the problem *relaxed community search*. Akoglu et al. [1] study the problem of finding *pathways*, i.e., connection subgraphs for a large query set Q , in terms of the Minimum Description Length (MDL) principle. According to MDL, a pathway is simple when only a few bits are needed to relay which edges should be followed to visit all of Q . Given a graph G and a query set Q , Gionis et al. [5] study the problem of finding a connected subgraph of G that has more vertices that belong to Q than vertices that do not. For a candidate solution S that has p vertices from Q and r not in Q , they define the *discrepancy* of S as a linear combination of p and r , and study the problem of maximizing discrepancy. They show that the problem is NP-hard and develop efficient heuristic algorithms.

The work which we extend to the case of temporal networks, is that of Ruchansky et al. [12] which studies, in the static graph setting, the problem of extracting the *minimum inefficiency subgraph*. Let $d_{G[S]}(u, v)$ denote the shortest-path distance between a pair of vertices $u, v \in S$, computed in the subgraph induced by S , then the problem is defined as follows.

Problem 1 (MIN-INEFFICIENCY-SUBGRAPH [12]): *Given an undirected graph $G = (V, E)$ and a query set $Q \subseteq V$, find the not necessarily connected subgraph H^* minimizing the network inefficiency:*

$$H^* = \arg \min_{G[S]: Q \subseteq S \subseteq V} \sum_{u, v \in S, u \neq v} 1 - \frac{1}{d_{G[S]}(u, v)}.$$

The intuition at the basis of this definition is that all-pairs shortest-path distances provide a good measure of how cohesive is a subgraph [11], [12]. However, one issue with shortest-path distance is that it is enough to have one disconnected vertex to have an infinite measure. A simple yet elegant workaround to this issue, is to use the reciprocal of the shortest-path distance, under the convention that $\infty^{-1} = 0$. Therefore, a disconnected pair of nodes produces $1 - \frac{1}{\infty} = 1$ inefficiency. On the opposite, a pair of adjacent vertices produces $1 - 1 = 0$ inefficiency. A pair of vertices at distance 2 produces $1/2$ inefficiency, a pair at distance 3 produces $2/3$ inefficiency, and so on.

Ruchansky et al. show that Problem 1 is NP-hard and develop an efficient greedy algorithm. They also show that the minimum inefficiency subgraph exhibits some nice properties such as the fact of producing small cohesive solutions, the fact of being able to detect outlier query vertices (by leaving them disconnected in the produced solution), and the fact of being able to recognize when the query vertices belong to different communities. Ruchansky et al. [12] also provide an empirical (qualitative and quantitative) comparison with [1], [5].

Temporal paths and Steiner trees on dynamic networks.

Given a dynamic network, where edges are timestamped, temporal paths are paths in the graph structure, along the temporal dimension. In particular, temporal paths must be *time-respecting*, that is, edges along a path appear in non-decreasing time [8]. Bui-Xuan et al. [2] study interesting paths as either shortest, fastest or foremost journeys. Wu et al. [14] propose more efficient methods to compute these paths in both streaming and transformed graph models. Recently, some parallel and distributed algorithms for computing temporal paths [15], [9] have been proposed.

A related problem that we use in our algorithm is the computation of the minimum Directed Steiner Tree in static graphs for which a first approximation algorithm was proposed by Charikar et al. [3]. Recently Huang et al. [6] have studied the problem of minimum spanning trees in temporal graphs and propose two definitions based on the optimization of time and cost. They also propose an improved approximation algorithm for the minimum Directed Steiner Tree problem for temporal graphs, using a graph transformation from temporal to static graphs, similar to the one we use in Section 4.

III. TEMPORAL NETWORK INEFFICIENCY

We are given a continuous stream of timestamped edges (u, v, t) , where $u, v \in V$ are vertices and t is a timestamp from a potentially infinite domain T . We can represent a dynamic graph as a sequence of graphs, one for each timestamp, defined over the same set of vertices i.e., $\mathcal{G} = \langle G_0, G_1, \dots, G_t, \dots \rangle$ with $G_t = (V, E_t)$ and $E_t = \{(u, v, t)\}$. Given a temporal interval $[i, j]$ we denote $G_{[i, j]}$ the projection of $\mathcal{G}$ over $[i, j]$. A temporal path between a pair of vertices $v, u \in V$ is a time-respecting sequence of edges, i.e.,

$$p(u, v) = \{(u = v_0, v_1, t_0), (v_1, v_2, t_1), \dots, (v_n, v_{n+1} = v, t_n)\}$$

such that $\forall i \in [1, n]$ it holds that $t_{i-1} \leq t_i$.

Example 1: A time-respecting path between vertex 6 and vertex 1 in Figure 1 is $p(6, 1) = \{(6, 4, 0), (4, 2, 1), (2, 1, 2)\}$. However, there exists no time-respecting path from 1 to 6. It is worth noticing that in a dynamic graph, even if undirected, due to the notion of temporal path, distance is not symmetric.

When dealing with temporal dynamic graphs, one can use different characteristics to define the interestingness of a path between two vertices. In fact, besides the usual spatial definition of shortest-path distance based on the number of intermediate vertices, one can also consider the temporal duration of the path itself. For instance, Wu et al. [14] study four different types of interesting paths over temporal graphs within a time window: (1) earliest-arrival path, (2) latest-departure path, (3) fastest path, and (4) shortest path. In this paper we adopt a linear combination of the temporal and spatial distance¹, governed by a parameter $\alpha \in [0, 1]$.

Definition 1 (Shortest Fastest Path [13]): Given a user-defined parameter $\alpha \in [0, 1]$ we define as shortest fastest path (SFP) between a pair of vertices $u, v \in V$ in a dynamic graph, a valid temporal path $p(u, v) = \{(u, v_1, t_0), \dots, (v_n, v, t_n)\}$ minimizing the cost:

$$\mathcal{L}(p) = \alpha |p(u, v)| + (1 - \alpha)(t_n - t_0) \quad (1)$$

This definition generalizes both shortest and fastest path notions. In fact by setting $\alpha = 1$ we obtain shortest paths, while setting $\alpha = 0$ we obtain fastest paths. In general, depending on the application at hand, one can tune the parameter α to give more importance to the temporal dimension ($\alpha < 0.5$) or the spatial one ($\alpha > 0.5$). The parameter α can also be tuned in such a way to favor one dimension, but using the other dimension for tie-breaking among equivalent paths in the first dimension. For instance, by setting α to a small positive quantity ϵ , the temporal paths that we obtain by Eq. (1) correspond to *fastest paths* with the minimum number of intermediate hops. Similarly, if we set $\alpha = 1 - \epsilon$, Eq. (1) will return the shortest paths, and among all shortest paths, the one employing the minimum number of timestamps.

Definition 2 (Shortest-Fastest-Path Distance): The shortest-fastest-path distance between two vertices u, v is the cost of the

¹Throughout this paper by “spatial” distance we mean the usual shortest-path distance defined as the minimum number of hops between two vertices.

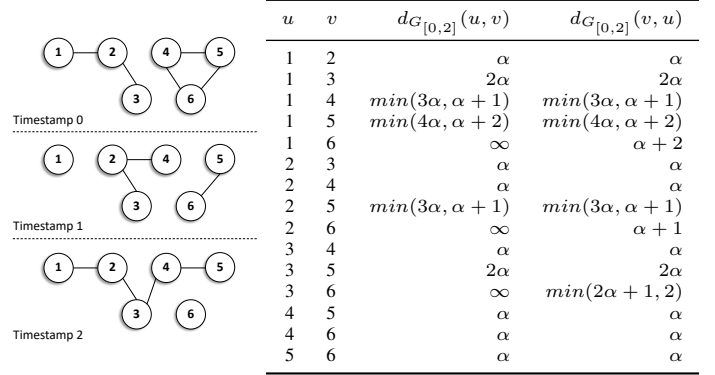


Fig. 1: An example dynamic graph on three timestamps, and the shortest-fastest-path distance between any pair of vertices.

shortest fastest path between the two vertices, i.e., $d_G(u, v) = \mathcal{L}(p^*(u, v))$, where $p^*(u, v) = \arg \min \mathcal{L}(p(u, v))$.

Example 2: Consider again the example in Figure 1 and suppose we want to go from vertex 1 to 4. We have a short path through vertex 2 over two timestamps, i.e., $\{(1, 2, 0), (2, 4, 1)\}$ or alternatively we have a bit longer path which is faster, as it spans only one timestamp: i.e., $\{(1, 2, 2), (2, 3, 2), (3, 4, 2)\}$. The cost of the former path is $\alpha + 1$, while the cost of the latter is 3α . An $\alpha > 0.5$ would favor the shortest path, while an $\alpha < 0.5$ will favor the fastest path.

Two simple observations are in place. First, in a dynamic graph, even if undirected, due to the notion of time-respecting path, the distance is no longer symmetric (nor it is reachability). Second, while distance in static graphs is usually in $[1, \infty]$, with our definition of shortest-fastest-path distance the minimum value is α . With these two observations in mind, we are now ready to extend the notion of network inefficiency by Ruchansky et al. [12], to the case of temporal dynamic graphs.

Definition 3 (Temporal Network Inefficiency): Given a temporal graph $G_{[0, t]}$ defined over a set of vertices V and $t + 1$ timestamps, and given a user-defined parameter $\alpha \in [0, 1]$, the network inefficiency of $G_{[0, t]}$ is defined as

$$\mathcal{I}(G_{[0, t]}) = \sum_{u, v \in V, u \neq v} \frac{(1 - \frac{\alpha}{d_{G_{[0, t]}}(v, u)}) + (1 - \frac{\alpha}{d_{G_{[0, t]}}(u, v)})}{2}.$$

Definition 3 differs from the notion of static network inefficiency in two main points. The first is that it uses reciprocal of the distance between two vertices, multiplied by α . This is to keep the values of the inefficiency in $[0, 1]$, as in the static case, since as we mentioned before, the minimum length of a temporal path is α . The second change, is due to the asymmetry in the distance of the temporal path between two vertices. Therefore, for each pair of vertices, we calculate the inefficiency introduced in the network by considering the distance in both directions of the path.

Example 3: Note that two vertices that are directly connected in at least a timestamp (as for instance vertices 1 and 2 in Figure 1) have a distance of α , and thus they do not add any temporal inefficiency to the network. Let us now consider

the pair of vertices (1,6). We already saw that there is no time-respecting path from vertex 1 to 6: thus this direction provides maximum inefficiency of 1. In the other direction we have the temporal path $p(6,1) = \{(6,4,0), (4,2,1), (2,1,2)\}$ of length 3 and spanning 3 timestamps (thus, with a distance of $\alpha + 2$). Assuming $\alpha = 0.5$, temporal inefficiency brought by the pair of vertices (1,6) is

$$\frac{(1 - \frac{0.5}{\infty}) + (1 - \frac{0.5}{2.5})}{2} = \frac{1 + 0.8}{2} = 0.9$$

which is close to the maximum inefficiency of 1.

We are now ready to formalize the problem studied in this paper. We do this in steps: first we define the problem of finding the minimum temporal-inefficiency subgraph in a single time window W ; then we extend the framework to deal with a sliding window, on a potentially infinite stream of graphs, in an adaptive way.

Let us consider the temporal graph $G_W = (V, E_W)$ defined over the time interval $W = [t_i - (|W| - 1), t_i]$ with $t_i \in T$. Given a set of vertices $S \subseteq V$, we denote $G_W[S]$ the temporal subgraph induced by S , i.e., $G_W[S] = (S, \{(u,v,t) \in E_W | u \in S \wedge v \in S\})$. Given a set of query vertices Q , the minimum temporal-inefficiency subgraph problem requires to find a set of vertices S containing Q , such that its induced subgraph minimizes the temporal inefficiency (Definition 3).

Problem 2 (MINIMUM TEMPORAL-INEFFICIENCY SUBGRAPH): Given a temporal graph $G_W = (V, E_W)$, a parameter $\alpha \in [0, 1]$ and a set of query vertices $Q \subseteq V$, find the minimum temporal-inefficiency subgraph of G_W :

$$H^* = \arg \min_{G_W[S]: Q \subseteq S \subseteq V} \mathcal{I}(G_W[S]).$$

We observe that when the temporal graph G_W is made of only one snapshot (i.e., $|W| = 1$) and $\alpha = 1$, Problem 2 exactly coincides to the MIN-INEFFICIENCY-SUBGRAPH on standard graphs (Problem 1) of Ruchansky et al. [12]. As Problem 1 is NP-hard and it is a special case of Problem 2, it holds that also the MINIMUM TEMPORAL-INEFFICIENCY SUBGRAPH problem is NP-hard.

Problem 2 above produces one solution subgraph for an input temporal graph defined over a fixed temporal interval. We next consider the *sliding window* setting, where the temporal graph of interest is defined by a temporal interval $W = [t - (|W| - 1), t]$ with $t \in T$ of predefined length $|W|$, which keeps sliding continuously along time. At every new timestamp the window W is updated by adding a new snapshot of the graph and removing the most obsolete one. Therefore, at every new timestamp $t \in T$, we have a new window W which, in turns, defines a new temporal graph G_W , and we want to solve a new instance of the Problem 2 outputting a new community. This enables the analysis of how the community around a query set of vertices Q changes along time.

Adaptive Query Set. One of distinguishing feature that our framework inherits from [12] is the fact of producing solutions which are not necessarily connected: if a vertex in the query

set Q results to be not so related to the others, it can be treated as an *outlier* by leaving it disconnected in the solution. This flexibility is an even more interesting feature in the context of temporal graphs analysis as it allows to deal with *concept drift*: something which is interesting and relevant at the beginning of the temporal domain under analysis, can become less and less relevant as time progresses.

Based on this observation and exploiting the flexibility of our framework, we propose the *adaptive query set* Q mechanism, as a way to deal with concept drift. This is achieved by means of two simple update rules that have the faculty of changing Q amid evidence emerging from the analysis. The two update rules are as follows:

- if a vertex $v \in Q$ remains disconnected in the solution H_t for a number of consecutive timestamps t larger than a user-defined threshold $\lambda_{out} \in \mathbb{N}^+$, then v is removed from Q ;
- if a vertex $v \in V \setminus Q$ appears in the solution H_t for a number of consecutive timestamps t larger than a user-defined threshold $\lambda_{in} \in \mathbb{N}^+$, then v is added to Q .

The rationale for these update rules is that in an explorative data analysis task, real-world query-sets are likely to contain some vertices that are erroneously suspected of being related, or that become unrelated along time due to concept drift. Similarly, a vertex which is added constantly to the solution, as it helps reducing the distance among the vertices in Q , can be considered related to the others in Q and added to the query set for future monitoring.

IV. ALGORITHMS

We next present our method for solving Problem 2, i.e., extracting the minimum temporal-inefficiency subgraph from a temporal graph $G_W = (V, E_W)$. Later we discuss the sliding window setting and its streaming-distributed implementation.

Our method is structured in three phases:

Phase 1: Graph Transformation – the temporal graph G_W is transformed in a static directed and weighted graph G' , by flattening all the snapshots G_t with $t \in W$ in a unique graph linking the various replicas of the same vertex in different timestamps, appropriately weighting these edges and the original edges, and adding some source and sinks dummy vertices. We show that the resulting transformed graph G' is such that the cost $\mathcal{L}(p)$ of the shortest-fastest path between two vertices in G_W can be computed as shortest-path distance between the corresponding source and sink vertices in G' .

Phase 2: Temporal connector – during the second phase we compute a *connector* for the query vertices in Q , i.e., a set of vertices $S \supseteq Q$ such that in the induced subgraph $G_W[S]$ all the vertices in Q are connected, to the maximum possible extent. For this task we extract many Directed Steiner Trees from transformed graph G' .

Phase 3: Minimum Temporal-Inefficiency Subgraph – in the third phase, following [12], we greedily reduce the solution S , returning as solution the subset which minimizes the temporal inefficiency.

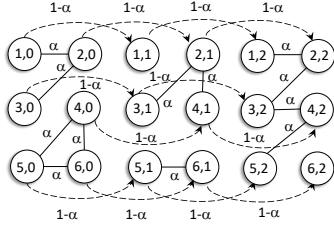


Fig. 2: The transformed graph for the example temporal graph $G_{[0,2]}$ from Figure 1.

We next present each phase in details, then we discuss sliding window setting and the adaptivity of the query set.

A. Phase 1: Graph Transformation

Given a temporal graph $G_W = (V, E_W)$ and the parameter $\alpha \in [0, 1]$ as in Problem 2, we build a static directed weighted graph $G'(V', E', \ell)$, where $\ell : E' \rightarrow [0, 1]$ is the edge weighting function, as follows:

- **Vertices:** for each $t \in W$ and each $v \in V$ we create a new vertex having as id the pair vertex-timestamp (v, t) , that is $V' = \{(v, t) : t \in W, v \in V\}$.
- **Edges:** for each $v \in V$ and each pair of consecutive timestamps $t_i, t_{i+1} \in W$ with we create a directed edge $((v, t_i), (v, t_{i+1}))$ with weight $1 - \alpha$. The edges in E_W , are instead assigned a weight of α .

Figure 2 shows the transformed graph for the example temporal graph $G_{[0,2]}$ from Figure 1.

Let us consider now a pair of vertices $(u, v) \in V \times V$, and let us define $P(u, v) = \{p((u, t_i), (v, t_j)) | t_i, t_j \in W\}$ the set of all paths from any replica of u to any replica of v in the transformed graph G' . Let us also denote $\ell(p)$ the length of one such path p , i.e., the sum of the weights of the edges in the path. Thanks to the edge labeling in G' , the following (straightforward) lemma holds.

Lemma 1: A path p on the transformed graph G' from a replica of u to a replica of v , corresponds to a valid temporal path p' from u to v in the temporal graph $G_W = (V, E_W)$, and the length of p in G' corresponds to the cost of p' in G_W , i.e., $\ell(p) = \mathcal{L}(p')$.

Example 4: On the temporal graph $G_{[0,2]}$ from Figure 1, consider the time-respecting path $p(6, 1) = \{(6, 4, 0), (4, 2, 1), (2, 1, 2)\}$. Figure 2 reports the same path on the transformed graph G' . We can observe that the cost of the path is $\alpha + 2$ as dictated by Eq.(1) in Definition 1.

From Lemma 1 it follows that to compute the shortest fastest path (SFP) between a pair of vertices $u, v \in V$ we can simply find the shortest path between any replica of u to any replica of v on the transformed graph G' . Moreover it holds that the length of such shortest path on G' corresponds to the shortest-fastest-path distance $d_{G_{[0,2]}}(u, v)$. In order to consider all possible shortest paths on G' from any replica of a vertex u to any replica of a vertex v we create two dummy vertices: a dummy source with id u and a dummy sink with id $(v, -1)$. By starting Dijkstra's algorithm from dummy source vertex u we can find the shortest path to sink $(v, -1)$, which corresponds to the SPF from u to v on $G_{[0,2]}$.

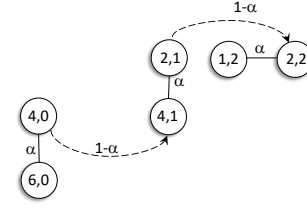


Fig. 3: The temporal path $p(6, 1) = \{(6, 4, 0), (4, 2, 1), (2, 1, 2)\}$ from Example 1 reported on the transformed graph G' . We can see that the path has a cost of $\alpha + 2$.

Example 5: Consider again the running example on $G_{[0,2]}$ (Figure 1), and suppose we want to compute the shortest-fastest-path distance $d_{G_{[0,2]}}(1, 4)$. On G' augmented with dummy source vertex 1 and sink $(4, -1)$, we compute all shortest paths from the source to the sink.

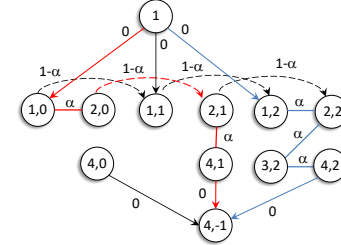


Fig. 4: Possible shortest-fastest paths from 1 to 4: the red one (if $\alpha > 0.5$) or the blue one (if $\alpha < 0.5$).

In Figure 4 we report the two paths, previously discussed in Example 2, $\{(1, 2, 0), (2, 4, 1)\}$ (in red) and $\{(1, 2, 2), (2, 3, 2), (3, 4, 2)\}$ (in blue). We can see how their length computed over G' are $\alpha + 1$ and 3α , respectively. Which of the two is the shortest-fastest path depends on the value of α , as discussed in Example 2. We observe that it holds that $d_{G_{[0,2]}}(1, 4) = \min(3\alpha, \alpha + 1)$ as reported in Figure 1.

The phase 1 of our method, graph transformation, is described in lines 1-8 of Algorithm 1.

B. Phase 2: Temporal Connector

In the second step we aim at building a *connector* for the set of query vertices Q , i.e., a set of vertices $S \supseteq Q$ such that in the induced subgraph $G_W[S]$ all the vertices in Q are connected, to the maximum possible extent. In fact, due to the requirement of time-respecting paths, not all vertices in Q are necessarily reachable from all the other vertices in Q .

We deal with this task by building *Directed Steiner Trees* (DST) [3] on the transformed graph G' augmented with the dummy source and sink vertices for all $v \in Q$.

Definition 4 (Minimum Directed Steiner Tree): Given a directed weighted static graph $G = (V, E, \ell)$ where ℓ is the edge-weighting function. Given a set of terminal vertices $X \subset V$ and a root vertex $r \in V$, the minimum directed Steiner tree problem (DST) ask to find the minimum cost tree which connects the root r to each terminal $v \in X$. In the case in which not all the terminals are reachable from the root, then the minimum cost achievable is $+\infty$. In this case the problem requires to return a tree that covers as much as possible of X , and among the trees that do so, the one of minimum cost.

More precisely, for each $r \in Q$ we search for the minimum DST using the dummy source vertex r as the root and the dummy sink vertices $\{(u, -1) | u \in Q \setminus \{r\}\}$ as terminals. For computing the minimum DST on the transformed graph G' augmented with the dummy source and sink vertices, we exploit the recent approximation algorithm by Huang et al. [6] appropriately modified to take care of the disconnected cases. It is important to note that the method in [6] requires in input the transitive closure of the graph G' which might be a computational bottleneck. In Algorithm 1, lines 9 we report the transitive closure computation as it was a pre-processing step. However, later in Section IV-D we discuss a way we use to reduce the computational overhead.

Let us denote $MDST(G', r, Q \setminus \{r\})$ such minimum DST, or simply $MDST(r)$. We also overload the weighting function ℓ to denote the total cost of the tree. i.e., $\ell(MDST(r))$. Our method selects

$$r^* = \arg \min_{r \in Q} \ell(MDST(r)).$$

If $\ell(MDST(r)) < \infty$ then we are done: we select as connector in the original temporal graph G_W the set of vertices whose replicas are involved in $MDST(r)$ computed over the transformed graph G' , i.e., $S = \{u \in V | (u, t_i) \in MDST(r)\}$. Otherwise, if $\ell(MDST(r)) = \infty$, then there are disconnected terminals. Let us denote the set of disconnected terminals $\bar{Q}$. Until $\bar{Q}$ is not empty, we pick randomly a $v \in \bar{Q}$, retrieve its $MDST(v)$ that we already computed, add its vertices to the connector S , and update $\bar{Q}$ by removing the terminal nodes which are reachable from v .

The details of the second phase of our method are reported in Algorithm 1, lines 9-19.

C. Phase 3: Minimum Temporal-Inefficiency Subgraph

During the second step we have created a connector, i.e., a set of vertices $S \supseteq Q$. Following [12], in the third and last step our method greedily removes from $S \setminus Q$ the vertex v^* which provides the greatest improvement in the objective function.

The process is iteratively continued until $S \setminus Q$ is not empty. In the end all the intermediate subgraphs produced during the process are compared and the one which provides the smallest temporal inefficiency is returned as solution. The third phase of our method is described in Algorithm 1, lines 20-28.

D. Streaming Distributed Implementation

In the previous sections we have presented in details our method, summarised in Algorithm 1, for the case of a temporal graph G_W defined over a fixed temporal window W . In the more general case, we consider an infinite stream of input graphs. At each new timestamp t the incoming graph G_t becomes the latest snapshot of the *sliding window* W , which maintains its size stable by leaving out the most obsolete snapshot. Each time the window W gets updated, we apply Algorithm 1 to the new G_W to produce a new minimum temporal-inefficiency subgraph. After this, the query set Q is potentially updated by using the update rules presented

Algorithm 1: Minimum Temporal-Inefficiency Subgraph

input : Temporal graph $G_W = (V, E_W)$, $\alpha \in [0, 1]$,
 $Q \subseteq V$
output: $G_W[S]$ such that $Q \subseteq S \subseteq V$

- 1 $Q' \leftarrow \{(u, -1) | u \in Q\}$
- 2 $V_W \leftarrow \{(v, t) | t \in W, v \in V\}$
- 3 $V' \leftarrow V_W \cup Q' \cup Q$
- 4 $E' \leftarrow \{((v, t_i), (v, t_{i+1})) | t, t+1 \in W, v \in V\}$
- 5 **for** $e \in E'$ **do** $\ell(e) \leftarrow 1 - \alpha$
- 6 **for** $e \in E_W$ **do** $\ell(e) \leftarrow \alpha$
- 7 $E' \leftarrow E' \cup E_W$
- 8 $G' \leftarrow (V', E', \ell)$
- 9 $\mathbb{G}' \leftarrow$ Transitive closure of G'
- 10 **for** $r \in Q$ **do**
- 11 $MDST(r) \leftarrow$ MinimumDST($\mathbb{G}', r, Q'$)
- 12 $r^* \leftarrow \arg \min_{r \in Q} \ell(MDST(r))$
- 13 $S \leftarrow \{u \in V | (u, t_i) \in MDST(r^*)\}$
- 14 **if** $\ell(MDST(r^*)) = \infty$ **then**
- 15 $\bar{Q} \leftarrow$
 $\{(u, -1) \in Q' | d_{MDST(r^*)}(r^*, (u, -1)) = \infty\}$
- 16 **while** $\bar{Q} \neq \emptyset$ **do**
- 17 pick v from $\bar{Q}$
- 18 $S \leftarrow$
 $S \cup \{u \in V | (d_{MDST(v)}(v, (u, -1)) \neq \infty)\}$
- 19 $\bar{Q} \leftarrow$
 $\bar{Q} \setminus \{(u, -1) \in \bar{Q} | d_{MDST(v)}(v, (u, -1)) \neq \infty\}$
- 20 $mincost \leftarrow \mathcal{I}(G_W[S])$
- 21 $S^* \leftarrow S$
- 22 **while** $S \setminus Q \neq \emptyset$ **do**
- 23 $v^* \leftarrow \arg \max_{v \in S \setminus Q} \mathcal{I}(G_W[S]) - \mathcal{I}(G_W[S \setminus \{v\}])$
- 24 $S \leftarrow S \setminus \{v^*\}$
- 25 **if** $\mathcal{I}(G_W[S \setminus \{v^*\}]) < mincost$ **then**
- 26 $mincost \leftarrow \mathcal{I}(G_W[S \setminus \{v^*\}])$
- 27 $S^* \leftarrow S \setminus \{v^*\}$
- 28 **return** $G_W[S^*]$

in Section 3. We deployed our framework in a streaming distributed implementation on Apache Spark.

The main computational bottleneck of Algorithm 1 is at line 9: in fact, the algorithm from [6] requires in input the transitive closure of the graph. Producing the transitive closure requires some computationally expensive algorithm, such as Floyd-Warshall, which has $\mathcal{O}(V^3)$ complexity. In order to avoid this step, we adopt a lazy evaluation approach computing on the fly the transitive closure between two vertices in the graph, when and only if this is required. Instead of using Floyd-Warshall algorithm, we use Dijkstra's algorithm, and we calculate the shortest paths from a given source vertex to the rest of the reachable vertices of the graph. In this way we avoid computing the shortest paths from the vertices that are unreachable temporally from the query vertices. The computation of all shortest paths from each query vertex is done in parallel. Depending on the depth i that is used as a parameter for the computation of the minimum DST as

described by [6], we continue i times the process by choosing as source vertices the i -th step neighbours of the query vertices in Q . Finally, we distribute the computation of the minimum DST that uses as root each query vertices in Q (lines 10-11 of Algorithm 1).

V. EXPERIMENTS

In our experiments we use three real-world dynamic networks summarized in Table I. Two datasets are face-to-face interaction networks gathered by the SocioPatterns² project using wearable proximity sensors in schools, with a temporal resolution of 20 seconds. PrimarySchool contains the contact events between 242 volunteers (232 children and 10 teachers) in a primary school in Lyon, France, during two days. The data expand across 19 hourly timestamps. HighSchool 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. The data expand across 41 hourly timestamps.

The third dataset is the co-authorship network of 16 conferences and journals from the databases, data mining, and information retrieval areas (VLDB, SIGMOD, ICDE, EDBT, KDD, ICDM, SIGIR, CIKM, WWW, WSDM, ECIR, ECML, TKDE, TODS, IEEE BigData and Data Mining and Knowledge Discovery) collected from DBLP³. Each vertex is an author and each edge represents co-authorship. It contains 18 yearly timestamps from 2000 to 2017.

Experimental environment. We run our experiments in on a 3.1 GHz Intel Core i7 machine with 16 GB of memory. We use local mode Spark execution with four worker nodes. In each worker we allocate one executor. For the computation of the DST we set the recursion depth to 1.

TABLE I: Characteristics of the datasets used.

Network	$ V $	$ E $	$ T $	time granularity
PrimarySchool	242	7 420	19	1 hour
HighSchool	327	40 896	41	1 hour
DBLP	46 160	377 852	18	1 year

A. Effect of Parameter α

One of the key notions of our work is that of shortest fastest path (Definition 1). Instead of choosing between measuring distances in space or time, we propose to use a linear combination of the temporal and spatial distance, governed by a user-defined parameter $\alpha \in [0, 1]$. Depending on the application at hand, one can tune the parameter α to give more importance to the temporal dimension ($\alpha < 0.5$) or the spatial one ($\alpha > 0.5$), with the extreme cases $\alpha = 0$ and $\alpha = 1$ corresponding to fastest-path distance and shortest-path distance, respectively. How to set α might depend on the application at hand, the sensibility of the analyst and the intended semantics of the cohesiveness measure. When there is no reason to favor space

over time, nor the other way around, one can simply adopt the default value of 0.5.

In the first set of experiments we analyse the effect of α . More in details we ask to what extent the communities found are similar, if we use $\alpha = 0.1$ and $\alpha = 0.9$. In these experiments we pick a set of query vertices Q at random, with $|Q| = 4$, and we measure the Jaccard similarity between the community returned at every timestamp for the two values of α . We use $\lambda_{in} = 1$ (i.e., nodes that are added to Q to form the solution S at time t , become part of Q at time $t + 1$), while we never drop vertices from Q . We repeat each experiment for 100 different Q and report the average.

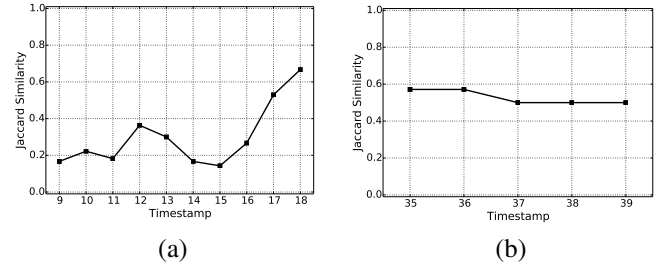


Fig. 5: Jaccard similarity for the communities detected for $\alpha = 0.1$ and $\alpha = 0.9$. (a): PrimarySchool with window length $|W| = 3$; (b): HighSchool dataset with $|W| = 6$.

Figure 5 (a) shows the Jaccard similarity for PrimarySchool when we set the window size to $|W| = 3$, presenting results for 10 consecutive timestamps. Since the window has size 3 at each timestamp we see the results that correspond in 3 hour interval. Therefore, at timestamp 14, we see the results of the temporal connector that corresponds to the window $W = [12, 14]$. The Jaccard similarity varies between 0.18 to 0.7 depending on the timestamp. Confirming that different values of α produce different behaviors. However, as the network evolves, the similarity between the solutions returned with $\alpha = 0.1$ and $\alpha = 0.9$ keeps growing. This is probably due to the adaptivity of the query set: as important vertices are added to the query set Q , the community search becomes more stable, converging on the truly important vertices that relate to the initial set of query vertices.

Figure 5 (b) shows the Jaccard similarity for HighSchool when we set the window size to $|W| = 6$, reporting results for five consecutive timestamps. Since each snapshot of the dataset corresponds to one hour data, the results reported correspond to 6 hour window interval. Therefore, at timestamp 35, we see the results that correspond to $W = [30, 35]$, i.e., starting from the 30th hour until the end of the 35th hour. Here the results are more stable with Jaccard similarity at about 0.5.

B. Solutions Characterization

We next characterize the solutions produced along time. For each experiment we present aggregated results for 100 randomly selected query sets. Following [12] we select the initial query set, which in the rest of this section will be denoted $Q_{t_0} \subseteq V$, by using three different parameters:

²<http://www.sociopatterns.org>

³<http://dblp.uni-trier.de/>

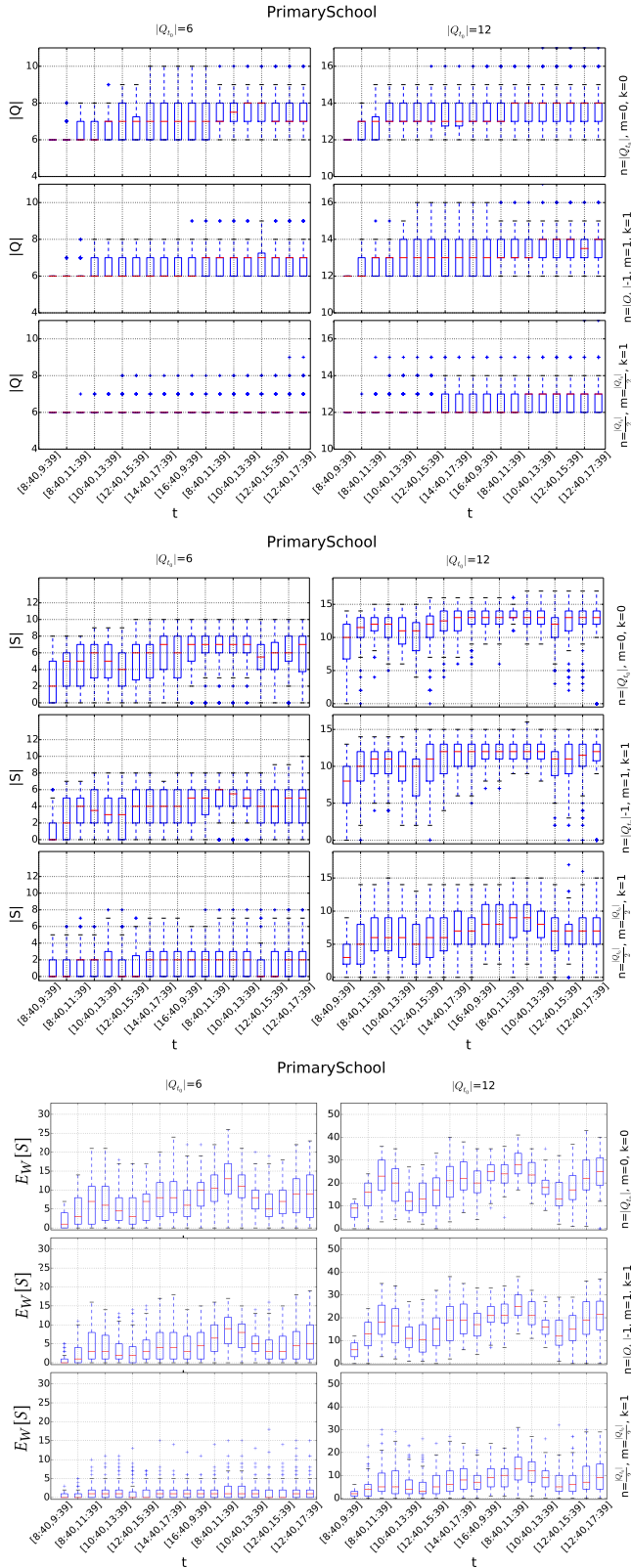


Fig. 6: $|Q|$, $|S|$, and $E_W[S]$ as a function of t for 100 random query sets of size $|Q_{t_0}| = 6$ and 12 for PrimarySchool dataset. The initial query sets Q_{t_0} are selected at random for $(n, m, k) = [(|Q_{t_0}|, 0, 0), (|Q_{t_0}| - 1, 1, 1), (\frac{|Q_{t_0}|}{2}, \frac{|Q_{t_0}|}{2}, 1)]$.

- the size of the initial query set $|Q_{t_0}|$,
- the number of vertices $n < |Q_{t_0}|$ that we select from a randomly chosen “main” community,
- the number of communities k from which we select the $m = |Q_{t_0}| - n$ remaining vertices.

This way we can control the number of “outliers” that we know are contained in Q_{t_0} . More specifically, we select at random a community from which we extract n vertices. The other $m = |Q| - n$ vertices are extracted from k other communities. Notice here that when $m = k$, it is implied that we have n vertices from the same community and m outliers. For this set of experiments we use PrimarySchool and HighSchool datasets, for which we already know the underlying communities, i.e., students belonging to the same class form a community. We produce 100 query sets for each one of the following three combinations of the parameters n, m, k :

- no outliers, i.e., $(|Q_{t_0}|, 0, 0)$;
- one outlier, i.e., $(|Q_{t_0}| - 1, 1, 1)$;
- query set split in two communities, i.e., $(|Q_{t_0}|/2, |Q_{t_0}|/2, 1)$.

We repeat our experiments for values of $|Q_{t_0}| = 6$ and 12. As before, we use $\lambda_{in} = 1$ (i.e., nodes that are added to Q to form the solution S at time t , become part of Q at time $t + 1$), while we never drop vertices from Q . Figure 6 reports the results for PrimarySchool: we focus on the size of the query set, $|Q|$, at the beginning of each timestamp, the size of the solution in terms of number of vertices $|S|$ and number of induced temporal edges $|E_W[S]|$.

We observe that all the different characteristics of the temporal connector ($|Q|$, $|S|$ and $|E_W[S]|$) exhibit the same behavior. In the first setting (first row plots), where all the vertices are concentrated in the same community, the size of the query set increases as the time evolves. However, in the second setting (second row plots), this behavior is smoothed as we notice a smaller increase of the size of Q that starts in a later timestamp. Finally, in the third setting, where the vertices are equally partitioned in the two communities, there is almost no increase in the query set for $|Q_{t_0}| = 6$ and a small increase for $|Q_{t_0}| = 12$ starting at timestamp 7. These results verify our hypothesis that our approach connects vertices that exist in the same community, and adds parsimoniously vertices in the temporal connector. We additionally notice, that thanks to the adaptivity of the query set the increase and the decrease of $|Q|$, $|S|$ and $|E_W[S]|$ are smooth in time.

We see similar trends for HighSchool data (not reported for sake of space).

C. Case Study

Finally we present some anecdotal evidence to give an idea of the practical applicability of our framework in the discovery of the community around a given set of vertices and monitor its evolution along time. Figure 7 reports an example in the DBLP co-authorship temporal network. The initial query

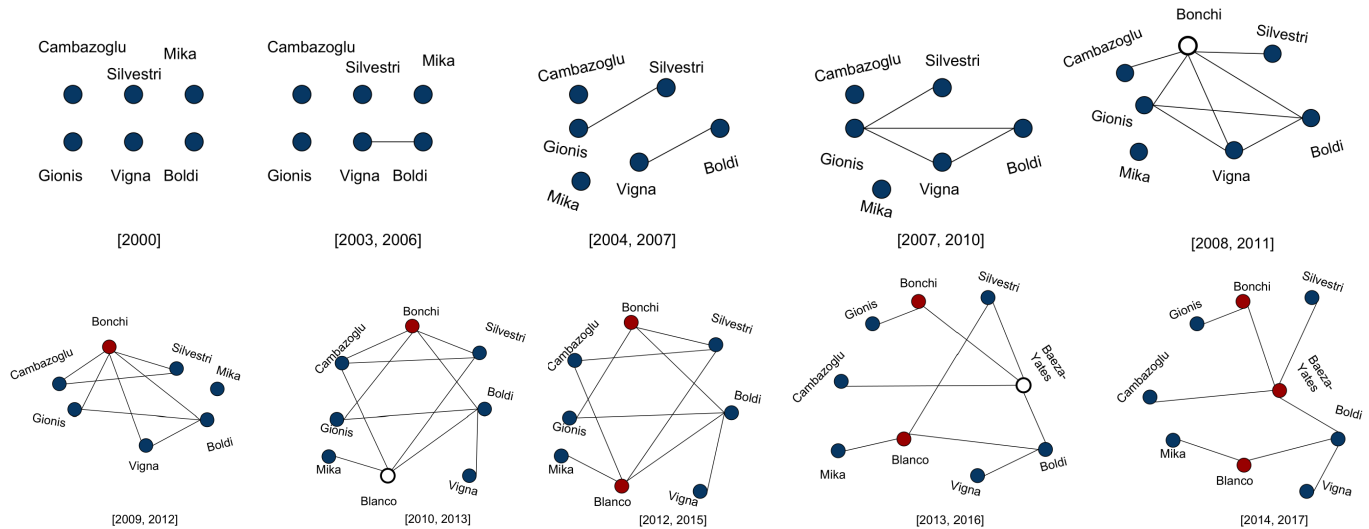


Fig. 7: Case study for DBLP dataset. Initial query set Q_{t_0} marked with blue color. The vertices which form the temporal community with Q are marked in white, while the vertices added to Q are marked in red. In this example $|W| = 4$, $\alpha = 0.1$, and $\lambda_{in} = 1$. The edges are the temporal edges that exist in at least one timestamp within the given time window. The figure reports only some of the

set Q_{t_0} contains six researchers, spanning different research communities, but which were all linked to Yahoo Labs in Barcelona: either employees or frequent academic visitors.

We can observe the following temporal dynamics. Back in time no collaboration was occurring among the six query vertices and the produced solution just leave all of them disconnected. Later on, as the lab in Barcelona is established, collaborations start appearing. However, at the beginning the solution subgraph is still sparse and formed by only the query vertices. As time goes by, more and more collaborators start being involved and they are added to the solution, as they help making the subgraph more cohesive (reducing the distances among the query vertices). The vertices (marked in white) which are used to produce a minimum temporal inefficiency subgraph, are then added to the query set for the future timestamps (and marked in red).

VI. CONCLUSIONS

In this paper we tackled the problem of community search in temporal dynamic networks and proposed the problem of extracting the minimum temporal-inefficiency subgraph. We developed a method based on a careful transformation of the temporal network to a static directed and weighted graph, and some recent approximation algorithm for the extraction of Directed Steiner Tree. We provided a streaming distributed implementation of our framework in Apache Spark and experimented on several real-world networks.

While the community search problem is well studied, both the “relaxed” variant, which allows some query vertices to remain disconnected, and the variant on temporal networks are largely and surprisingly unexplored. These are important problems which, in the coming years, will receive more foundational treatment and will find more applications. This work represents just a first step in this direction.

REFERENCES

- [1] L. Akoglu, D. H. Chau, C. Faloutsos, N. Tatti, H. Tong, and J. Vreeken. Mining connection pathways for marked nodes in large graphs. In *Proc. of SIAM Int. Conf. on Data Mining (SDM)*, pages 37–45, 2013.
- [2] B. Bui-Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *Int. J. Found. Comput. Sci.*, 2003.
- [3] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. *J. Algorithms*, 1999.
- [4] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *The VLDB Journal*, 29(1):353–392, 2020.
- [5] A. Gionis, M. Mathioudakis, and A. Ukkonen. Bump hunting in the dark: Local discrepancy maximization on graphs. In *ICDE*, 2015.
- [6] S. Huang, A. W. Fu, and R. Liu. Minimum spanning trees in temporal graphs. In *SIGMOD 2015*.
- [7] X. Huang, L. V. Lakshmanan, and J. Xu. Community search over big graphs: Models, algorithms, and opportunities. In *Proc. of IEEE Int. Conf. on Data Engineering (ICDE)*, pages 1451–1454, 2017.
- [8] D. Kempe, J. M. Kleinberg, and A. Kumar. Connectivity and inference problems for temporal networks. *J. Comput. Syst. Sci.*, 2002.
- [9] P. Ni, M. Hanai, W. J. Tan, C. Wang, and W. Cai. Parallel algorithm for single-source earliest-arrival problem in temporal graphs. In *Parallel Processing (ICPP)*, 2017.
- [10] T. Przytycka, M. Singh, and D. Slonim. Toward the dynamic interactome: It’s about time. *Briefings in Bioinformatics*, 11(1):15–29, 1 2010.
- [11] N. Ruchansky, F. Bonchi, D. García-Soriano, F. Gullo, and N. Kourtellis. The minimum wiener connector problem. In *Proc. of ACM Int. Conf. on Management of Data (SIGMOD)*, pages 1587–1602, 2015.
- [12] N. Ruchansky, F. Bonchi, D. García-Soriano, F. Gullo, and N. Kourtellis. To be connected, or not to be connected: That is the minimum inefficiency subgraph problem. In *Proc. of Int. Conf. on Information and Knowledge Management (CIKM)*, pages 879–888, 2017.
- [13] I. Tsalouchidou, R. Baeza-Yates, F. Bonchi, K. Liao, and T. Sellis. Temporal betweenness centrality in dynamic graphs. *Int. J. Data Sci. Anal.*, 9(3):257–272, 2020.
- [14] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 2014.
- [15] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering*, 2016.