International Journal of Data Science and Analytics (2020) 9:257–272
https://doi.org/10.1007/s41060-019-00189-x

REGULAR PAPER

Temporal betweenness centrality in dynamic graphs

Ioanna Tsalouchidou[1] · Ricardo Baeza-Yates[1,2] · Francesco Bonchi[3,4] · Kewen Liao[5] · Timos Sellis[6]

Received: 20 November 2018 / Accepted: 22 May 2019 / Published online: 5 June 2019

Abstract
Measures of centrality of vertices in a network are usually defined solely on the basis of the network structure. In highly dynamic networks, where vertices appear and disappear and their connectivity constantly changes, we need to redefine our measures of centrality to properly capture the temporal dimension of the network structure evolution, as well as the dynamic processes over the network. Betweenness centrality (BC), one of the most studied measures, defines the importance of a vertex as a mediator between available communication paths. BC value of a node is expressed as the fraction of the shortest paths passing through this node. In other words, given the context that information flow follows the shortest paths, a node with higher BC potentially has greater influence on the information flow in the network. In temporal dynamic graphs, a communication path should be seen as a path both in space (i.e., the network structure) and in time (i.e., the network evolution). Toward this goal, in this paper we propose the bi-objective notion of *shortest–fastest path* (SFP) in temporal graphs, which considers both space and time as a linear combination governed by a parameter. Based on this notion, we then define a novel temporal betweenness centrality (TBC) metric, which is highly sensitive to the observation interval and the importance of space and time distances of the vertices, that can provide better understanding of the communication mediators in temporal networks. We devise efficient algorithms for exactly computing all-pairs SFPs and the corresponding BC values both in a single graph window and sliding graph windows. We also present a distributed implementation of our approach on Apache Spark which shows great solution effectiveness and efficiency. We provide a thorough experimentation on a large variety of datasets. An application to the analysis of information propagation proves that our notion of TBC outperforms static BC in the task of identifying the best vertices for propagating information.

Keywords Temporal networks · Dynamic graphs · Betweenness centrality

1 Introduction

Measuring the importance of a vertex in terms of its position in a static network structure, i.e., its *centrality*, is a fundamental task in network analysis. An actor in a network can be deemed important thanks to its ability to influence other actors, as well as to spread or to block information propagation. As *shortest paths* are often used to model the flow of information in a network, one of the most studied measures of the importance of a vertex is *betweenness centrality* (BC), i.e., the fraction of shortest paths that pass through it [3,10]. BC has been used to analyze a variety of different networks such as online social networks (OSN) [29], social [27], protein [19], wireless ad hoc [28], mobile phone call [9], and multiplayer online gaming [2] networks, just to mention a few. It is also at the basis of one of the first and most well-known algorithms for community detection [11].

✉ Francesco Bonchi
francesco.bonchi@isi.it

Ioanna Tsalouchidou
ioanna.tsalouchidou@upf.edu

Ricardo Baeza-Yates
rbaeza@acm.org

Kewen Liao
kewen.liao@cdu.edu.au

Timos Sellis
tsellis@swin.edu.au

1 Pompeu Fabra University, Barcelona, Spain

2 NTENT Inc., San Diego, CA, USA

3 ISI Foundation, Turin, Italy

4 Eurecat, Barcelona, Spain

5 Charles Darwin University, Sydney, Australia

6 Swinburne University, Melbourne, Australia

However, real-world networks are rarely static: new vertices arrive and old vertices disappear, as well as new connections are created or removed continuously. Therefore, the analysis of dynamic networks is receiving increasing attention. In this regard, substantial research effort has been devoted to the problem of dynamically maintaining BC values up-to-date on streaming graphs: this is to say that at each temporal instant, the BC value of each vertex should match the current status of the network structure, avoiding to recompute everything from scratch each time. Contrarily, the problem of defining and computing notions of BC on a sequence of contiguous temporal snapshots (i.e., a temporal window) has received little attention (brief survey in Sect. 2).

When we drop the strong assumption of measuring centrality instant by instant, we can obtain interesting temporal characterization of a network. For instance a path from vertex a to vertex b might materialize in two different timestamps, e.g., by means of an edge (a, c) at time 1 and an edge (c, b) at time 4, even if the two edges never coexist at the same time. Similarly, a path from a to b might materialize through edges (a, d) and (d, f) at time 2 and edge (f, b) at time 3. Although the first path is *shorter* in terms of network structure, the second one is *faster* in terms of temporal duration. The second path also starts later and ends earlier.

These examples highlight the need of reconsidering the notion of shortest path when reasoning on an extended temporal window in a dynamic network. Wu et al. [41] define four different types of interesting paths over temporal graphs: (1) earliest-arrival path, (2) latest-departure path, (3) fastest path, and (4) shortest path.

In this paper, we generalize the last two of these notions, by means of a linear combination, governed by a parameter, which allows us to give more importance to the length of path in terms of hops or to its temporal duration. We call these paths *shortest–fastest paths* (SFPs). Based on this novel definition of paths, we introduce a new measure of *temporal betweenness centrality* (TBC) and study how to efficiently compute it. Our analysis starts in a static time window which includes snapshots of a graph at different timestamps and continues with the sliding window case where new snapshots of the graph appear in a streaming fashion, while old snapshots are discarded as they fall out of the current window.

1.1 Challenges, contributions, and roadmap

In our endeavor of developing methods for TBC, the main challenge is given by the fact that measuring BC is computationally intensive even in simple static graphs. Indeed, the best known algorithm for BC, proposed by Brandes [6], runs in $\mathcal{O}(nm)$ time. Dealing with TBC in dynamic networks does not make things easier.

The approach developed in this paper starts with a *graph transformation* that converts a temporal graph in a unique directed and weighted graph. We show that, thanks to a careful weighting of the links in this transformed graph, we can obtain all the SFPs by computing all-pairs shortest paths in the transformed graph, and by filtering out some of them. We then extend Brandes' algorithm [6] to deal with the novel notion of TBC: the resulting algorithm computes, on the basis of their participation in SPFs, the TBC of all the vertices for a given temporal window of a dynamic temporal graph. Then we extend our method to deal with a sliding temporal window. Finally, we devise a distributed implementation in Apache Spark which achieves efficiency and scalability for this computationally intensive task, as confirmed by our extensive experimentation.

The contributions of this paper can be summarized as follows:

- We introduce the notion of shortest–fastest paths (SFPs) in dynamic graphs combining spatial length and temporal duration. We then define a notion of TBC based on SFPs (Sect. 3).
- We extend Brandes' algorithm to compute this new notion of TBC over a static temporal window (Sect. 4), and we prove theoretically the correctness of the algorithm. Then we extend our algorithm to the sliding window case (Sect. 5).
- We devise a distributed implementation of the method in Apache Spark for higher efficiency and scalability (Sect. 5.1).
- Extensive experimentation on several real-world dynamic network provides insights on how our notion of TBC is sensitive to the observation interval and to the parameter governing importance of distance and duration of the paths (Sect. 6).
- An application to information propagation proves that our notion of TBC outperforms static BC in the task of identifying the best nodes at propagating information (Sect. 7).

Section 2 provides a brief survey of related literature, while in Sect. 8, we conclude the paper and discuss future work.

2 Related work

2.1 Static and incremental betweenness centrality

For the problem of computing BC of all vertices in a static graph, Brandes' algorithm [6] achieves the current best asymptotic time with linear space. This significantly improves the approach of naively computing and accumulating all-pairs shortest paths (APSP). More recent studies [5,13,18,20,24,25,33] have been devoted to incrementaly maintaining/updating static BC on dynamic networks. These

networks are treated as streaming graphs where edges are inserted or removed.

Kas et al. [20] and Green et al. [13] are the first proposals of update algorithms for evolving graphs to avoid full BC re-computation. Kourtellis et al. [24] extended [13] to fully dynamic and improved in both time and space together with a scalable distributed implementation. Jamour et al. [18] propose an incremental distributed computation that uses linear space and outperforms the previous works. Our work differs from the incremental approaches, since at every time instance our algorithm receives as input the latest view of the graph and removes the most obsolete one out of the observation window, which implies multiple additions and deletions of nodes and edges. Furthermore, we aim at computing the BC values of the vertices in a time frame (TBC), that differs to the above approaches that incrementally calculate/update the static BCs of the vertices in a streaming fashion.

As none of the above exact methods are asymptotically better than Brandes [6], for the sake of scalability, researchers [4,16,35,36] have recently drawn their attentions to approximated BC computation via sampling-based methods. Most noticeably, Riondato and Kornaropoulos [35] sample pairs of vertices for an unbiased estimate of BC in static graphs. Based on that, Bergamini et al. [4] propose the fully dynamic counterpart. However, determining the sample size is the bottleneck in these approaches due to the VC dimension theory. Later, Riondato and Upfal [36] got around this bottleneck with progressive sampling and the analysis of Rademacher Averages. Their sampling step combined with the dynamic maintenance technique of Hayashi et al. [16] is highly accurate and efficient. Just recently, Alghamdi et al. [1] survey and benchmark most of the approximated BC algorithms.

2.2 Temporal paths

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 either strictly increasing or non-decreasing time. Kempe et al. [22] study connectivity problems in temporal networks based on time-respecting paths. Additionally, they focus on reconstructing a partially specified time labeling while respecting the observed history of the information flow in the network. Bui-Xuan et al. [8] analyze three types of least cost journeys: hop count, arrival date and time span. Wu et al. [41] add to the interesting paths studied in [8] the *latest-departure* paths and propose more efficient methods to compute these paths in both streaming and transformed graph models. Recently, some parallel algorithms for computing the four different types of temporal paths [30,42] have been proposed. Our work differs from this literature, as we propose the general notion of *shortest–fastest paths* which considers both space and time as a linear combination.

2.3 Temporal betweenness centrality

Extending BC definition to consider temporal aspects has been investigated by [14,15,23,32,34,38,40] adopting slightly different definitions of shortest paths, but mostly based on the temporal duration of the paths.

Habiba et al. [15] define as *geodesic distance* the time difference between the first and last link of the path. Their approach does not allow simultaneous interactions of the nodes in one time instance. They define three different paths that have the same geodesic distance. *Shortest temporal path* is the shortest time respecting path between two individuals, whereas *shortest link path* is the shortest temporal path with minimum number of individuals on the path. Finally, *shortest temporal trails* is defined with respect to the ratio of the time spent on an intermediate node over the total length of the path. Based on two of these definitions, they define temporal BC based on *shortest temporal paths* and delay-betweenness centrality based on *shortest trails*. Kim and Anderson [23] restrict their model to one edge per timestamp and define as shortest path within a fixed interval of time, the path with the shortest distance in terms of hops. Habiba et al. [15] and Kim and Anderson [23] both only consider strict time-respecting/time-increasing paths without taking into account nodes interacting almost instantly at a given time or in a very small time interval. This is actually the case of online social networks and is also what we care about.

As a remedy, the definition from Tang et al. [38], focusing more on online social networks, allows non-decreasing time paths and within the same time window messages can be passed up to a certain number of hops. More in details, Tang et al. [38] define non-decreasing time paths in non-overlapping windows of time, restricting the number of simultaneous interactions in one timespan to a fixed value. The length of the path is defined as the difference in time between the first and the last interaction of the path. Tang et al. [38] do not actually count these instantaneous paths at any given time when computing their temporal BC, whereas we do consider them here. Additionally, they assume that the snapshots of the temporal graph have the same set of vertices, whereas we allow additions and deletions of vertices in the new snapshots. Moreover, all these notable earlier works focused on studying concepts of temporal BC while ignoring how best it can be computed. On the other hand, we also focus on efficient approaches to carry out our exact BC computation.

The recent and probably the closest works to ours are [32,34]. Pereira et al. [32] study both closeness and betweenness centralities in a streaming model tailored for Twitter data. Nevertheless, in their work only fastest path based on time were considered and their algorithm has high space

and time (at least cubic) costs in maintaining all-pairs fastest paths. Furthermore, how centrality values were accumulated and then computed is not properly described in [32]. Afrasiabi et al. [34] define *foremost BC*, based on *foremost journeys*, i.e., the paths that have the earliest arrival time and propose algorithms with worst-case exponential time. Gunturi et al. [14] define shortest path in an observation period similarly to [32] and propose an epoch-point based approach to avoid redundant computation of shortest paths when the network does not change. The algorithm depends on the number of epoch points, which can be significantly high when we consider fast evolving temporal networks.

Our approach differs from this literature as it considers non-decreasing time paths but also allows simultaneous interactions of the vertices in one time instance similar to [41]. It allows multiple additions and deletions of vertices and edges in time and considers both spatial and time distance of the paths as a linear combination, without imposing further constraints on the starting and finishing times of the paths.

There are also recent studies on spatio-temporal networks [37,40], where our temporal path definition can also be viewed as spatio-temporal if our considered instantaneous path lengths are regarded as a spatial dimension.

3 Problem formulation

A temporal graph is 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 temporal domain T. We can represent the temporal graph as the sequence of sets of edges that arrive in each timestamp, i.e., $\mathcal{G} = \langle E_0, E_1, \ldots, E_t, \ldots \rangle$ where $E_i = \{(u, v, i)\}$. A window graph is a projection of $\mathcal{G}$ over a temporal interval (or window): i.e., given the window $W = [t - (|W| - 1), t]$ of length $|W|$, we denote $\mathbf{G}_W = (V, \mathbf{E}_W)$ the window graph defined over W, where $\mathbf{E}_W = \bigcup_{i \in W} E_i$. We also denote as V_t the subset of vertices V that appear in timestamp t.

Definition 1 (*Temporal path*) A temporal path between a pair of vertices $u, v \in V$ in a window graph $\mathbf{G}_W = (V, \mathbf{E}_W)$ is a sequence of edges $p(u, v) = \{(u = v_0, v_1, t_0), (v_1, v_2, t_1), \ldots, (v_n, v_{n+1} = v, t_n)\}$ such that $\forall i \in [1, n]$ it holds that $t_{i-1} \leq t_i$.

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 based on the number of intermediate vertices, one can also consider the temporal duration of the path itself. For instance, Wu et al. [41] 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 (3) shortest path.

We next introduce our notion of interesting path which combines and generalizes the last two definitions by Wu et al. [41].

Definition 2 (*Shortest–fastest path*) 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 window graph $\mathbf{G}_W$, a valid temporal path $p(u, v) = \{(u, v_1, t_0), \ldots, (v_n, v, t_n)\}$ minimizing the cost:

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

As said above, our definition generalizes both shortest and fastest path notions, since it combines the spatial (number of edges) and the temporal distance (steps in time) as number of hops in two dimensions. 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. More in detail, 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 that expand in fewer number of timestamps.

We next define our notion of TBC based on shortest–fastest paths. Before, we recall the standard definition of BC on a static graph $G = (V, E)$. Let $\sigma(s, d)$ denote the *total number of shortest paths* from s to d in G; moreover, for any $v \in V$, let $\sigma(s, d|v)$ be the number of shortest paths from s to d that pass through v. Note here that $\sigma(s, s) = 1$ and $\sigma(s, d|v) = 0$ if $v \in s, d$ [6]. For every vertex $v \in V$ its betweenness centrality (BC) is defined as:

$$BC(v) = \sum_{s,d \in V, s \neq d} \frac{\sigma(s, d|v)}{\sigma(s, d)}. \tag{2}$$

Let us now consider a temporal graph as defined at the beginning of this section. Given a window W let $\mathbf{G}_W = (V, \mathbf{E}_W)$ denote the corresponding window graph. Let $\sigma_{SFP}(s, d)$ be the number of SFPs from vertex s to vertex d in $\mathbf{G}_W$ according to Definition 2.

Definition 3 (*Temporal betweenness centrality*) *TBC* of a vertex v in a window graph $\mathbf{G}_W$ is defined as:

$$TBC(v) = \sum_{s,d \in V, s \neq d} \frac{\sigma_{SFP}(s, d|v)}{\sigma_{SFP}(s, d)}$$

The first problem studied in this paper is as follows.

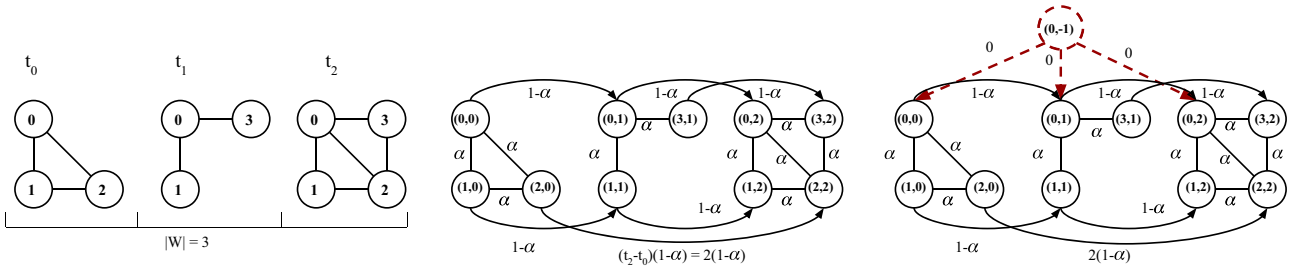

Fig. 1 An input window graph $\mathbf{G}_W$ (left), its transformation G' (center), its augmentation with the dummy vertex (right)

Problem 1 (Static window case) Given a window graph $\mathbf{G}_W = (V, \mathbf{E}_W)$ and a parameter $\alpha \in [0, 1]$, compute the $TBC(v)\ \forall v \in V$.

After having proposed our algorithm to solve Problem 1 (in Sect. 4), we move to the sliding window case (Sect. 5), in which at every new timestamp the window W slides, one position, to include the latest set of edges while excluding the set that falls outside of the limits of the window, as defined in the next problem statement.

Problem 2 (Sliding window case) Given a window length $|W|$, a parameter $\alpha \in [0, 1]$ and a timestamp $t \in T$ that increases continuously in time, compute the $TBC(v)\ \forall v \in V$ in the window graph $\mathbf{G}_W = (V, \mathbf{E}_W)$ defined by the window $W = [t - (|W| - 1), t]$.

4 Static window case

In this section we introduce our method for computing the TBC of all vertices, given a parameter $\alpha \in [0, 1]$ and a window graph $\mathbf{G}_W$.

4.1 Computing shortest–fastest paths

Our approach to compute all-pairs SFPs consists of three phases. In the first phase (inspired by [41]) we transform the input window graph to a static graph by linking, through directed edges, the various replicas of the same vertex in different snapshot and by appropriately weighting these auxiliary edges and the original edges. In the second phase, for each vertex u of the original graph, we create a dummy vertex connected to all the temporal replicas of u and we run Dijkstra's algorithm from the dummy vertex to compute the shortest paths to all the other vertices in the transformed graph. Finally, in the third phase, we aggregate the results so that vertices with the same id across different timestamps are considered as the same vertex. We next describe more formally all three phases.

Phase 1: graph transformation. Given a graph window $\mathbf{G}_W = (V, \mathbf{E}_W)$, we transform it to a static, directed and weighted graph $G'(V', E', r)$, where r is the weighting function, as follows:

1. **Vertices**: for each $t \in W$, $v \in V_t$ we create a vertex id as a pair vertex-timestamp (v, t), i.e., $V' = \{(v, t) : t \in W, v \in V_t\}$.
2. **Edges**: for each $v \in V$ and each pair of timespans $t_i, t_j \in W$ with $t_j = \min\{t : (v, t) \in V', t > t_i\}$, we create a directed edge $((v, t_i), (v, t_j))$ with weight $(t_j - t_i)(1-\alpha)$. The edges in $\mathbf{E}_W$ are instead assigned a weight of α.

An example of this transformation is shown in Fig. 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 window graph $\mathbf{G}_W$, and the length of p in G' corresponds to the cost of p' in $\mathbf{G}_W$, i.e., $\ell(p) = \mathcal{L}(p')$.*

Following this observation, our method computes, for each pair of vertices (u, v), all the shortest paths from any replica of u to any replica of v (and viceversa), on the transformed graph G'.

Let $\ell^*(u, v) = \arg\min_{p \in P(u,v)} \ell(p)$. We want to compute the set of path $SP(u, v) = \{p \in P(u, v)|\ell(p) = \ell^*(u, v)\}, \forall (u, v) \in V \times V$.

This is achieved in the next two phases. Phase 2 produces, for any vertex $(v, t) \in V'$, the shortest paths among all the paths from any replica of u. Phase 3 instead aggregates all the shortest paths from any replica of u to any replica of v to finally produce $SP(u, v)$.

Phase 2: shortest paths in the transformed graph. In order to compute the shortest paths from any replica of u to any other vertex in G', phase 2 creates a dummy vertex $(u, -1)$ and connect it to all the replicas of u in G' by means of

directed arcs with weight of 0. An example is given in Fig. 1 (right), where we want to calculate all shortest paths that start from the vertex with id 0. In this case, we need to concurrently calculate all shortest paths from vertices $(0, 0)$, $(0, 1)$ and $(0, 2)$. Therefore we create the dummy vertex $(0, -1)$ with directed edges to the vertices $(0, 0)$, $(0, 1)$ and $(0, 2)$ highlighted in red. Finally, we run Dijkstra's algorithm with the dummy vertex as source, returning three lists:

- S, which is the list of vertices $(v, t) \in V'$ in non-decreasing distance from the source $(u, -1)$,
- D, which contains the distance, i.e., the length of the shortest path, of each vertex $(v, t) \in V'$ from $(u, -1)$,
- P, which is the list of predecessors for each vertex $(v, t) \in V'$ in all the shortest paths from $(u, -1)$.

Theorem 1 *Running Dijkstra's algorithm from dummy source vertex* $(u, -1)$ *correctly finds the set of all the shortest paths from any replica of* u *to a vertex* $(v, t) \in V'$.

Proof First we observe that in a shortest path $p((u, -1), (v, t))$, there is at most one intermediate vertex from the set $\{(u, t') : t' \in W\}$, that can by proved by absurd. Therefore, a shortest path from a dummy vertex $(u, -1)$ to a vertex $(v, t) \in V'$ will be $p((u, -1), (v, t)) = \langle (u, -1), (u, t_i), x_i, \ldots, (v, t)\rangle$ where $x_i \neq (u, t_j)$ for any $t_j \in W$. Using Bellman Criterion on shortest paths of static graphs [6], we have that the shortest path from (u, t_i) to (v, t) can be obtained by removing $(u, -1)$ from the path $p((u, -1), (v, t))$. □

Phase 3: aggregation. Consider the example of Fig. 2 with a transformed graph defined over only two timestamps. Consider the pair of vertices with id (in the original graph) 0 and 5. Dijkstra's algorithm over the transformed graph detects two shortest paths (marked in red) from some replica of 0 to some replica of 5. At the end of phase 2, vertices with the same id but different timestamps are still treated as different and thus we have two different results (shortest path in timestamp 0 and in timestamp 1). However, $(5, 0)$ and $(5, 1)$ are the same vertex viewed in two different timestamps. Therefore, the SFP from vertex with id 0 to the vertex with id 5 is the one with source $(0, 1)$, destination $(5, 1)$ and length α. On the other hand, the SFP from vertex with id 0 to vertex with id 4 can come from both timestamp zero and one, when $\alpha < 0.5$ (paths highlighted with blue color). In this case, there are four paths from vertex 0 to vertex 4 all of which have length 3α.

Therefore in phase 3, we need to aggregate all the shortest paths that regard the same vertex id. This is done by removing from S vertices for which a replica with the same vertex id has appeared earlier in the list with smaller distance. Following [6], we use an "augmented" Dijkstra that also maintains an additional structure σ that contains the number of shortest paths from the source vertex $(u, -1)$ to each of the other vertices: this will result useful later in Sect. 4.2 to compute TBC. When we update S also σ and D must be updated accordingly: in the pseudocode in Algorithm 1, we use S', D' and σ' to denote the updated S, D and σ, respectively.

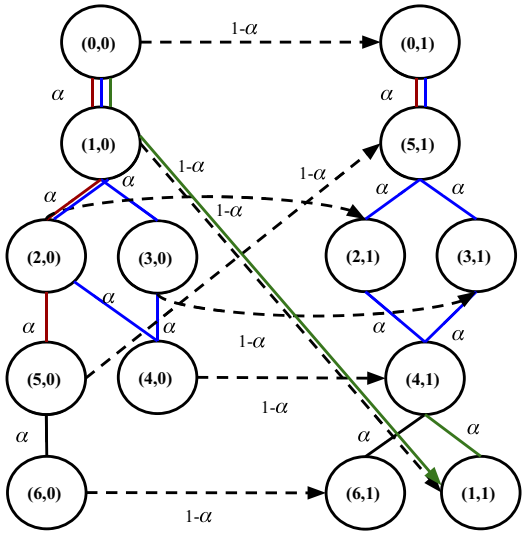

Fig. 2 Shortest paths on a transformed graph defined over two timestamps. Paths from vertex 0 to vertex 5 are marked with red. Paths from vertex 0 to vertex 4 are marked with blue and green. For $\alpha < 0.5$, SFPs are only the blue paths. For $\alpha > 0.5$, the only SFP is the green path, whereas for $\alpha = 0.5$ both blue and green are SFPs (color figure online)

Algorithm 1: All-pair Shortest–Fastest Paths (APSFP)

input : $\mathbf{G}_W = (V, \mathbf{E}_W) = \{(V_t, E_t) : t \in W\}, |W|, \alpha$
output : $S, S', P, D, \sigma, \sigma'$

$V' \leftarrow \emptyset$; $E' \leftarrow \emptyset$
$V' \leftarrow \bigcup\{(v, t) : v \in V_t, t \in W\}$
$E' \leftarrow \bigcup\{((u, t), (v, t), \alpha) : (u, v) \in E_t, t \in W\}$
$E' \leftarrow E' \cup \{((v, t), (v, t'), (1-\alpha)(t'-t)) : (v, t) \in V'\}$, where $t' = min\{t_i : (v, t_i) \in V', t_i > t\}$
for $u \in V$ **do**
  $V'_u \leftarrow V' \cup \{(u, -1)\}$
  for $t \in W$ **do**
    if $(u, t) \in V'$ **then**
      $E'_u \leftarrow E' \cup \{((u, -1), (u, t), 0)\}$
  $G'_u \leftarrow (V'_u, E'_u)$
  $S[u], P[u], D[u], \sigma[u] \leftarrow$ Dijkstra$(G'_u, (u, -1))$
  $S'[u] \leftarrow []$; $D'[u] \leftarrow \{\}$; $\sigma'[u] \leftarrow \{\}$;
  for $(i = 0;\ i < |S|;\ i = i + 1)$ **do**
    $(x, t) \leftarrow S[u][i]$
    if $((x, t) \neq u)$ **and** $(x \notin D'[u]$ **or** $D[u][(x, t)] = D'[u][x])$ **then**
      $S'[u]$.append$((x, t))$
      $D'[u][x] \leftarrow D[u][(x, t)]$
      $\sigma'[u][(x, t)] \leftarrow \sigma[u][(x, t)]$
    else
      $\sigma'[u][(x, t)] = 0$

All-Pair Shortest–Fastest Paths (APSFP). Algorithm 1 summarizes the method. Given $\mathbf{G}_W$ and α the algorithm

starts with the graph transformation (lines 1–4) as described in Phase 1. Lines 5–11 computes the shortest paths in the transformed graph as described in phase 2 and outputs the structures S, P, D and σ. Finally, the aggregation process described in phase 3 is given in lines 12–20. In this phase, we use the output of phase 2 to compute σ', which is the dictionary that contains the number of shortest paths (after the merging) of each vertex $v \in V'$ from the source vertex $(u, -1)$ and is initially empty (line 12). At the end of phase 3, it holds that the $\sigma'_{uv} = \sum_{(v,t):t\in W} \sigma'[u][(v,t)]$, where σ'_{uv} is the number of SFPs between u and v for the given α in $\mathbf{G}_W$.

To produce σ', we need to employ the auxiliary structure D' (initially empty), which contains the distance of each $v \in V$ from the source vertex. We start by traversing all vertices (v,t) contained in S and add their distance $D[(v,t)]$ in D', if these vertices are endpoints of some shortest path after merging (condition in line 15). If (v,t) is endpoint of some shortest path, we update the value of $\sigma'[u][(v,t)]$ with the value of $\sigma[u][(v,t)]$, otherwise, we set it to 0.

Theorem 2 *Algorithm 1 computes all SFPs from each vertex $u \in V$ to the rest of the vertices on the graph window $\boldsymbol{G}_W$.*

Proof Algorithm 1 constructs structure $S'[u]$ from $S[u]$, that contains all vertices in non-decreasing distance from the source $(u, -1)$. The distance of each vertex from the source is the sum of the weight α, when an edge corresponds to a hop during one timestamp, and $(t_j - t_i)(1-\alpha)$ when the edge corresponds to a hop between vertices of the same id that appear in timestamps t_i and t_j. Thus, as already highlighted in Lemma 1, the length of one of these paths in the transformed graph corresponds to the cost $\mathcal{L}(\cdot)$ in Definition 2. By selecting from $S[u]$, among the vertices of the same id, the ones with the smallest distance from the source, we construct the structure $S'[u]$ that contains the destination vertices of all and only the SFPs from u. □

4.2 Temporal betweenness centrality

Brandes' algorithm. In a static graph $G(V,E)$ Brandes' algorithm [6] uses the notion of *dependency* of a vertex $s \in V$ to another, intermediate, vertex $v \in V$, defined as:

$$\delta_{s\bullet}(v) = \sum_{w\in V} \delta_{sw}(v).$$

The *pair-dependency* $\delta_{sw}(v) = \frac{\sigma_{sw}(v)}{\sigma_{sw}}$ of the vertices s, w on the vertex v is the number of shortest paths from s to w that v lies on divided by the total number of shortest paths from s to w. Note here that σ_{sw} is the number of shortest paths from s to w, $\sigma_{sw}(v)$ is the number of shortest paths from s to w that go through v and finally that $\sigma_{ss} = 1$ and $\sigma_{sw}(v) = 0$ if $v \in \{s, w\}$. Brandes proves that the above partial sums obey a recursive relation which is the core of its algorithm:

$$\delta_{s\bullet}(v) = \sum_{w:v\in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_{s\bullet}(w)),$$

where $P_s(w)$ is the list of the predecessors of node w on shortest paths from s. In other words, the dependency of s on the vertex v can be calculated using the dependencies s on the successors of v. Each successor w of v contributes to $\delta_{s\bullet}(v)$ their dependency score $\delta_{s\bullet(w)}$ plus 1 which is the shortest path that starts from s to w. This value is multiplied by the ratio $\frac{\sigma_{sv}}{\sigma_{sw}}$ which is the proportion of shortest paths from s to w passing by the vertex v and the edge $\{v, w\}$. Therefore, by traversing the vertices in non-increasing distance from s we can accumulate the dependency scores for all vertices. Finally, the betweenness centrality of a vertex v can be calculated as:

$$BC(v) = \sum_{s\neq v\neq w\in V} \delta_{sw}(v).$$

Temporal betweenness centrality. Consider Fig. 2: the SFP from vertex with id 0 to the vertex with id 5, is only the path that includes vertices $\langle(0,1),(5,1)\rangle$ as vertex $(5,0)$ is not a destination of any SFPs starting from source with id 0. However, vertex $(5,0)$ lies on the shortest path from $(0,0)$ to $(6,0)$, and therefore, there is pair dependency of vertices $(0,0)$ and $(6,0)$ to the vertex $(5,0)$ which should be calculated. Finally, in order to compute the dependency of vertex $(0,0)$ to the vertex $(2,0)$, which is the endpoint of the SFP, we should also consider the dependency of vertex $(5,0)$ even if it is not the endpoint of any SFP.

Definition 4 Given shortest paths and shortest–fastest paths counts (σ and σ'), we can define the pair-dependency $\delta_{st}(v)$ of a pair of vertices s, t to the vertex v in a graph window, where $s, t, v \in V'$:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \frac{\sigma'_{st}}{\sigma_{st}}$$

According to Definition 4, the pair dependency of vertices s, t on v will be either 0, if vertex t is not endpoint of any SFP, or $\frac{\sigma_{st}(v)}{\sigma_{st}}$, since $\frac{\sigma'_{st}}{\sigma_{st}}$ is either 0 or 1.

Theorem 3 *The dependency of $s \in V'$ on any vertex $v \in V'$ obeys:*

$$\delta_{s\bullet}(v) = \sum_{w:v\in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot \left(\frac{\sigma'_{sw}}{\sigma_{sw}} + \delta_{s\bullet}(w)\right)$$

Proof According to proof of correctness of Brandes' algorithm [6, Theorem 6], we have that

$$\delta_{s\bullet}(v) = \sum_{t\in V'} \delta_{st}(v) = \sum_{t\in V'} \sum_{w:v\in P_s(w)} \delta_{st}(v,\{v,w\}) = \sum_{w:v\in P_s(w)} \sum_{t\in V'} \delta_{st}(v,\{v,w\}), \quad (3)$$

where $\delta_{st}(v,\{v,w\})$ is the pair-dependency that includes the edge $\{v,w\}$. More formally we have that $\delta_{st}(v,\{v,w\}) = \frac{\sigma_{st}(v,\{v,w\})}{\sigma_{st}}$, where $\sigma_{st}(v,\{v,w\})$ is the number of shortest paths from s to t that include both the vertex v and the edge $\{v,w\}$.

Let w be any vertex with $v \in P_s(w)$. If vertex $t = w$ then from σ_{sw} paths that go from s to w only σ_{sv} pass from vertex v first. In case that vertex w is not an endpoint of some SFP, $\delta_{st}(v,\{v,w\}) = 0$. When $t = w$, these two cases can be expressed as $\delta_{st}(v,\{v,w\}) = \frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma'_{sw}}{\sigma_{sw}}$. Recall that $\frac{\sigma'_{sw}}{\sigma_{sw}}$ is either 0 or 1.

In case that $t \neq w$, if t is the endpoint of some SFP ($\sigma'_{st} = \sigma_{st}$), we have $\delta_{st}(v,\{v,w\}) = \frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma_{st}(w)}{\sigma_{st}}$ (see [6]) and 0 otherwise. Therefore, when $t \neq w$ these two cases can be expressed as $\delta_{st}(v,\{v,w\}) = \frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma_{st}(w)}{\sigma_{st}}\frac{\sigma'_{st}}{\sigma_{st}}$. The above are summed up by:

$$\delta_{st}(v,\{v,w\}) = \begin{cases} \frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma'_{sw}}{\sigma_{sw}}, & t = w \\ \frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma_{st}(w)}{\sigma_{st}}\frac{\sigma'_{st}}{\sigma_{st}}, & t \neq w \end{cases}$$

and from Eq. 3:

$$\delta_{s\bullet}(v) = \sum_{w:v\in P_s(w)} \left(\frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma'_{sw}}{\sigma_{sw}} + \sum_{t\in V'\setminus\{w\}} \frac{\sigma_{sv}}{\sigma_{sw}}\frac{\sigma_{st}(w)}{\sigma_{st}}\frac{\sigma'_{st}}{\sigma_{st}} \right) = \sum_{w:v\in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \left(\frac{\sigma'_{sw}}{\sigma_{sw}} + \delta_{s\bullet}(w) \right)$$

□

Merging TBC results. Until here, our algorithm computes the temporal betweenness centralities of all vertices in the transformed graph, i.e., for all vertices (v,t) in V'. The final step of our approach is to merge the BC results of the vertices with the same id in different timestamps to compute TBC for all vertices $v \in V$.

Theorem 4 *The TBC of a vertex v in a graph window $\boldsymbol{G}_W$ is the sum of the TBCs of this vertex in all timestamps of the window W.*

$$TBC(v) = \sum_{t\in W} TBC((v,t))$$

Proof The dependency of a vertex $s \in V$ to a vertex $v \in V$ is:

$$\delta_{s\bullet}(v) = \sum_{w\in V'} \delta_{sw}(v) = \sum_{w\in V'} \frac{\sigma_{sw}(v)}{\sigma_{sw}}\frac{\sigma'_{sw}}{\sigma_{sw}} = \sum_{w\in V'} \frac{\sum_{t\in W}\sigma_{sw}((v,t))}{\sigma_{sw}}\frac{\sigma'_{sw}}{\sigma_{sw}} = \sum_{t\in W} \delta_{s\bullet}((v,t))$$

Therefore, we have:

$$TBC(v) = \sum_{s\in V} \delta_{s\bullet}(v) = \sum_{s\in V}\sum_{t\in W} \delta_{s\bullet}((v,t)) = \sum_{t\in W}\sum_{s\in V} \delta_{s\bullet}((v,t)) = \sum_{t\in W} TBC((v,t))$$

□

5 Sliding window case

In this section, we extend the static window TBC to the sliding window setting. We consider an infinite stream of input graphs that update the window graph at every timestamp with the newest snapshot of the temporal graph, and at the same time, we remove the most obsolete snapshot. This process implies changes on the values of the TBCs of the vertices, not only due to the changes on the shortest paths between the existing vertices, but also due to the appearance and removal of the vertices and edges in the window graph at every timestamp.

Figure 3 shows the case of a sliding window of length 3 in three consecutive timestamps. Timestamp 0, which does not appear in the figure, contains only the first snapshot of the graph in the rightmost position of the window. Timestamp 1 (left side of Fig. 3) shows the window after the appearance of the second snapshot of the graph. Therefore, the first snapshot moves one position to the left of the window and gives its position to the new timestamp. Finally, we create the links between the vertices with same ids in the different timestamps. In the next timestamp, when a new snapshot arrives, occupies the rightmost position of the window, whereas the older snapshots move one position to the left. If all the positions of the window are occupied, the oldest snapshot is removed from the window graph, as shown in the right part of Fig. 3 marked with red color (Timestamp 3).

The calculation of the TBCs of the vertices in a window is done in the following steps that are shown in Algorithm 2. The newest timestamp, upon arrival (line 3), takes the rightmost position in the window, its vertices are renamed, the edges get weighted, while the leftmost snapshot is removed (lines 4–7) so as to produce the updated transformed graph (line 8). To compute the TBC of the vertices (line 9) Algorithm 2 calls the distributed process described by Algorithm 3. In line 2 of Algorithm 3 all ids $= \{v \in \bigcup_{t\in W} V_t\}$ are distributed across the computation entities. Then, the algo-

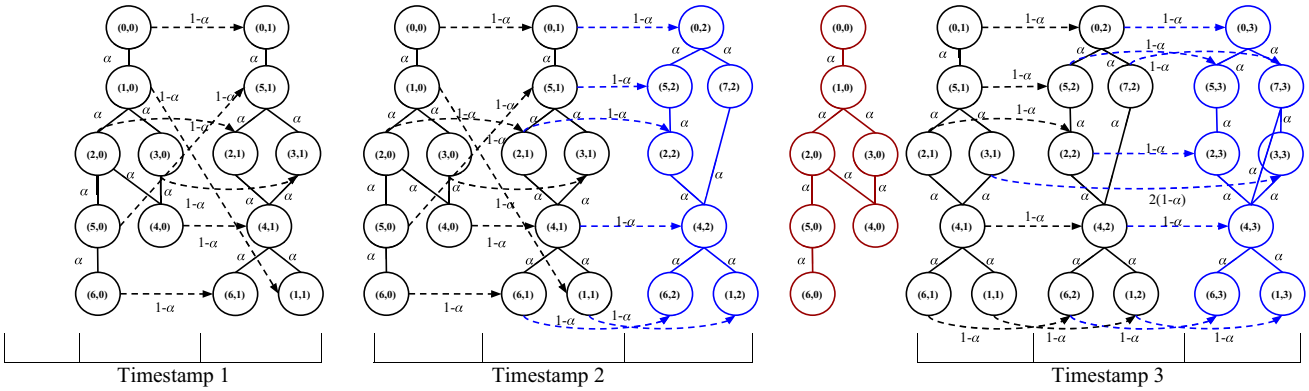

Fig. 3 Transformed graph for 3 consecutive timestamps for $|W| = 3$. The figure shows the second timestamp (Timestamp 1) where W contains two snapshots of $\mathcal{G}$. In the last instance of the figure, we the most obsolete snapshot is removed from W for the newest one to enter

Algorithm 2: Sliding Window TBC

input : $|W|$, α, *rounds*
output : TBC
1 $\mathbf{G}_W \leftarrow [], V' = \emptyset, E' = \emptyset, t = 0$
2 **while** $t <$ *rounds* **do**
3 **read** $G_t(V_t, E_t)$ //Read new timestamp
4 $V' \leftarrow V' \cup \{(v, t) : v \in V_t\}$ **//vertex renaming**
5 $V' \leftarrow V' \setminus \{(v, t_i) : t_i = t - (|W| - 2)\}$
6 $E' \leftarrow E' \cup \{((v, t), (u, t), \alpha) : (v, t) \in V', (u, t) \in V'$ and $(v, u) \in E_t\}$ **//static edges**
7 $E' \leftarrow E' \cup \{((v, t'), (v, t), (1 - \alpha)(t - t'))\}$, where $t' = max\{t_i : (v, t_i) \in V', t_i < t\}$ **//temporal edges**
8 $G' = (V', E')$
9 $TBC \leftarrow$ Algorithm 3
10 **report** *TBC*
11 $t++$

Algorithm 3: Distributed TBC

1 ids $\leftarrow \{v \in \bigcup_{t \in W} V_t\}$;
2 distrIds $\leftarrow$ sc. **parallelize**(ids);
3 dependencies_RDD $\leftarrow$ distrIds.**map**(lambda s: dependencies(s, $|W|$, G', t)) //Algorithm 4.
4 $TBC \leftarrow$ dependencies_RDD.**reduce**(lambda $\delta_{x,\bullet}$,$\delta_{y,\bullet}$: sum($\delta_{x,\bullet}$,$\delta_{y,\bullet}$)) //Sum values of $\delta_{x,\bullet}$ and $\delta_{y,\bullet}$ by key.

Algorithm 4: Dependencies

input : s, W, $G'(V', E')$, t
output : $\delta_{s\bullet}$
1 Add dummy vertex (s,-1): Algorithm 1 lines 6–10;
2 S, P, D, $\sigma \leftarrow$ Dijkstra(G'_s, $(s, -1)$);
3 SFPs: Algorithm 1 lines 12–20;
4 $\delta_{s\bullet} \leftarrow$ Brandes(S, S', P, σ, σ', s, W);
5 **report** $\delta_{s\bullet}$

rithm computes the dependencies of each vertex $s \in \mathbf{G}_W$ to every other vertex (line 3), with the function described by Algorithm 4. Finally, all dependency results are summed up in line 4. The result of this summation is the value of the TBC of all vertices in $\mathbf{G}_W$, according to Theorem 4.

Algorithm 4 calculates the dependencies of a vertex to the rest of the vertices in a window and is called for each one of the vertex ids of the graph. It first creates the dummy vertex (line 1) and then calculates the shortest paths from the dummy vertex to the rest of the vertices using Dijkstra's algorithm (line 2). The calculation of SFPs, described in Algorithm 1, remains the same. The final step is the calculation of the dependencies (line 4) using the extended Brandes' algorithm for window graphs as described in Sect. 4.2.

5.1 Distributed implementation

Due to its computational complexity, BC can be prohibitive to compute for large-scale graphs. This is mostly due to the calculation of APSPs that cannot be avoided. However, by exploiting the properties of the algorithms, i.e., the independence of the calculation of Dijkstra's algorithm for each source vertex and the summation of the dependencies to compute the TBCs of the vertices, we can distribute the computation on a cluster of machine cores (CM). Therefore, while the complexity of the calculation remains the same, we can have a theoretical improvement of the execution time up to a factor of $\frac{1}{|CM|}$, where $|CM|$ is the number of cores.

For the implementation of our algorithms, we used the Apache Spark framework. Figure 4 shows an overview of the distributed process that is described by Algorithm 3. The first step is the distribution of the data, i.e., the vertex ids of the window graph, across the CM. Spark framework uses the notion of *Resilient Distributed Dataset (RDD)*, which is the basic Spark abstraction. Each computation machine has assigned one partition of the data, for which is exclusively responsible. The computation of the shortest paths and the dependencies from all source vertices of the partition to the rest of the vertices of the graph is done in parallel for all CM. After the computation of the dependencies in the various

Table 1 Dataset name, number of vertices n, number of edges m, time span T and temporal granularity

Network	n	m	T	Time granularity
Infectious	279	3928	10	1 h
LastFM	1372	596,496	314	1 day
MathOverflow	12,491	147,968	27	1 month
DBLP	35,851	316,570	18	1 year
FBwall	44,609	310,089	13	1 month
WikiConflict	116,230	4,524,510	60	1 month
WikiTalk	1,094,018	8,020,640	2185	1 day

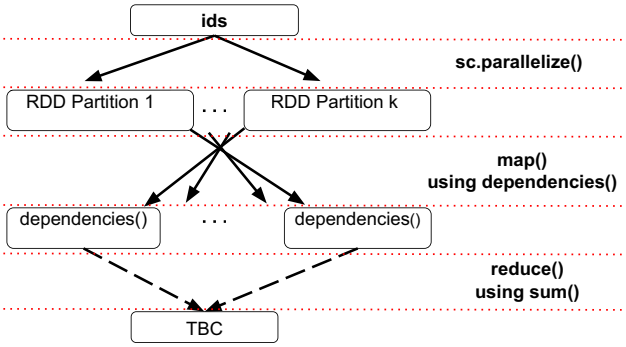

Fig. 4 High level overview of the distributed implementation described by Algorithm 3. Vertex ids are distributed to the CM, whereas the graph is replicated. Dependencies calculated in CM are summed to produce the TBC results

cores, we sum the values in order to calculate the TBC of the vertices [reduce() function in Fig. 4].

Complexity. Given a source $s \in V$ the length and the number of SFPs can be determined in $\mathcal{O}(m + n \log n)$, where m is the total number of edges in the transformed graph. Therefore, the computational complexity for the serial algorithm is $\mathcal{O}(n(m + n \log n))$. The computational cost of the distributed algorithm is reduced by a factor of $|CM|$ to $\mathcal{O}(\frac{n}{|CM|}(m + n \log n))$. The space complexity of the serial algorithm is $\mathcal{O}(m + n)$, whereas the distributed algorithm, which maintains each structure for each source node s, requires $\mathcal{O}(\frac{n}{|CM|}(m + n))$ space.

6 Experimental evaluation

The objectives of our experimentations are: (1) to characterize SFPs in terms of temporal duration and spatial distance; (2) to characterize TBC w.r.t. static BC applied to a dynamic network, w.r.t. time, and w.r.t. the parameter α, (3) to study scalability of the proposed distributed implementation with varying number of cores.

We use seven real-world dynamic networks summarized in Table 1.

Infectious: This is human contact data available from SocioPatterns (http://www.sociopatterns.org/) and described in [17]. Vertices represent visitors of an exhibition and edges represent face-to-face contacts. The data expand in 10 hourly timestamps.

LastFm: This dataset contains the graph of friendship between the users of Last.fm together with a timestamped activity log, i.e., users listening to songs. We define a temporal edge when two users, which are friends in the social network, listen to the same song. It expands in 314 daily timestamps from 1/1/2012 to 11/9/2012.

MathOverflow: The vertices of this graph are users of the Math Overflow website. An edge represents answers or comments to questions or comments between two users. The dataset expands in 35 monthly timestamps between 2014 and early 2016. This network was created in [31] and is available at https://snap.stanford.edu.

DBLP: This is the co-authorship network of nine conferences (VLDB, SIGMOD, ICDE, EDBT, KDD, ICDM, SIGIR, CIKM and WWW) collected from the DBLP database (http://dblp.uni-trier.de/). Each vertex is an author and each edge represents co-authorship. It contains 18 yearly timestamps that expand from the 2000 to 2017.

FBwall: Each vertex represents a user of the Facebook social network. Each edge is a wall post from one user to some other user's wall. For our experiments, we created 13 monthly timestamps from January 2008 to January 2009. This communication network is described in [39] and is available at the http://konect.uni-koblenz.de/.

WikiConflict: Each vertex represents a user of English Wikipedia, and each edge represents a conflict between two users. This is a subset of the dataset described by [7] and is available at the Konect database. It contains 60 monthly timestamps from 2004 to 2009.

WikiTalk: Each vertex represents a user of Wikipedia, and an edge between two users represents an edit from one user to the Talk page of the other user. It contains 2185 daily

timestamps, is described by [26,31] and is available at https://snap.stanford.edu.

Experimental Environment: We created a Spark cluster of 80 cores that expand in 5 machines. Each machine has 16 cores Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz. The driver program has a limited memory of 6GB and runs in one core of the cluster. We created 5 worker nodes, one per each machine. In each worker, we raise 3 executors with 5 cores per executor. For each executor, we allocate 7GB of memory. In total we use up to 70 cores out of the 75 available on the Spark cluster (5workers × 3executors × 5cores).

Reproducibility: Our code (serial and distributed version) is available at https://goo.gl/PAAJvp.

6.1 Shortest–fastest paths characterization

Figure 5 reports a characterization of SFPs in terms of temporal duration and spatial length (as number of hops on the network structure), between pairs of randomly selected vertices, for various $|W|$ and α. For each dataset, we present four plots that show the cumulative function of pairs of vertices: i.e., on the y-axis we have the number of pairs of vertices that have (temporal or spatial) distance smaller than the value on the x-axis. On the left part, we see the temporal distance for two different window sizes $|W|$ and on the right column we see the spatial distance. In each plot, we present the cumulative function for three different values of α. For $\alpha = 0.001$ the SFPs approximate the fastest paths, which are paths with smaller duration. Therefore, we expect more paths with temporal distance 0 than in the case of SFPs with $\alpha = 0.999$, which approximate the shortest paths. Equivalently, when $\alpha = 0.999$ we expect more SFPs with higher spatial distance than in the case of $\alpha = 0.001$.

Starting from the top left to bottom right in Fig. 5, we present the results for $|W| = 6$ and $|W| = 10$ for the Infectious dataset and for $|W| = 6$ and $|W| = 12$ for the LastFM, MathOverflow, DBLP, FBwall and WikiConflict datasets. For Infectious, we have used up to $|W| = 10$ window length, since the dataset contains only 10 timestamps. On the other hand, for the rest of the datasets, we could use even larger size of window. For the Infectious dataset, we selected 130 random pairs of vertices, for LastFM we select 600 random

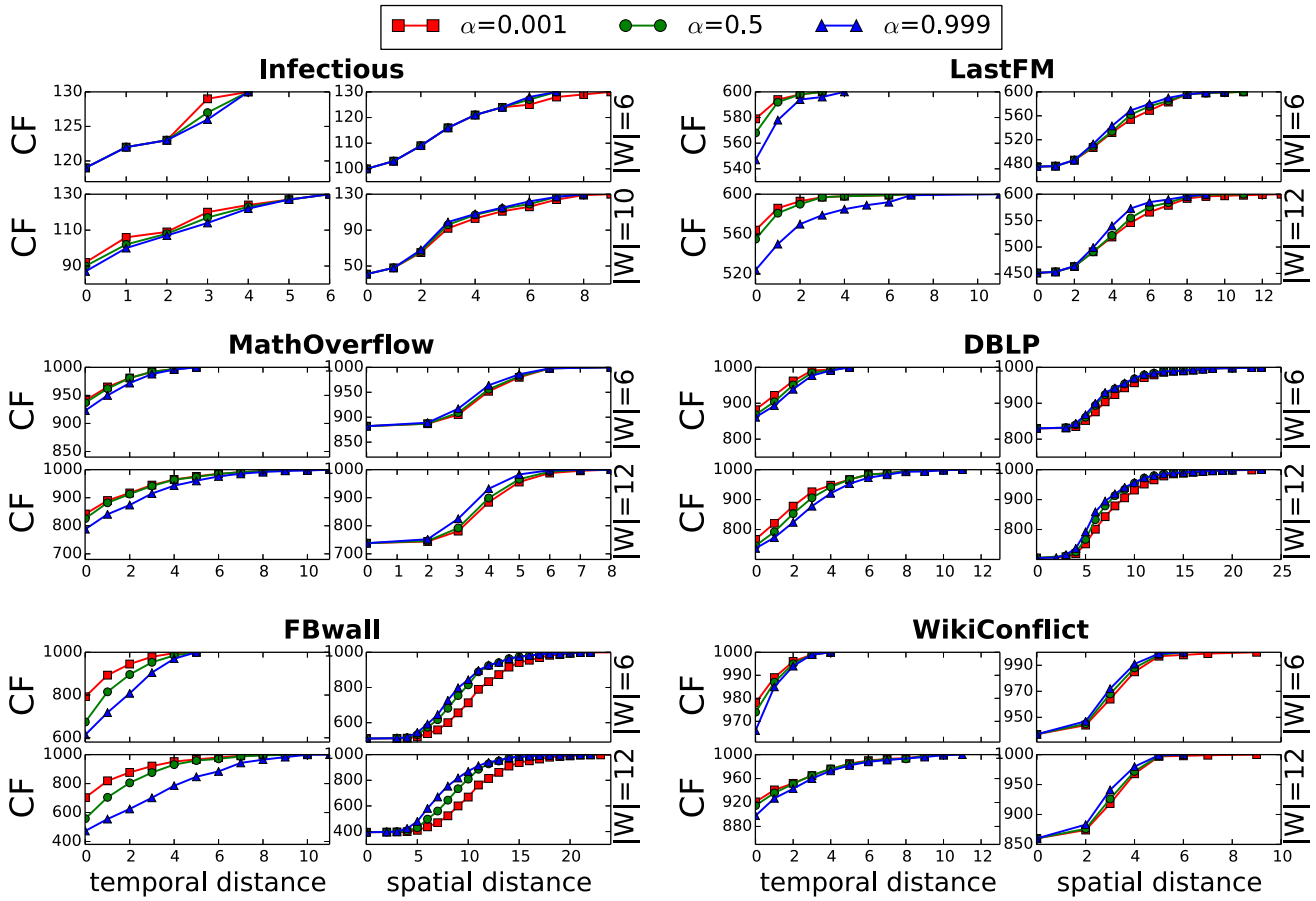

Fig. 5 Cumulative function for spatial and time distance of SFPs of pairs of vertices randomly selected. We present results for two values of $|W|$ for Infectious, LastFM, MathOverflow, DBLP, FBwall and WikiConflict datasets

pairs, whereas for the last two datasets we selected 1000 random pairs.

In Fig. 5, we observe that when α takes small values ($\alpha = 0.001$ marked with a red line), the number of pair of vertices with smaller temporal distance is much higher than in the case of higher α ($\alpha = 0.999$ marked with a blue line). Depending on the dataset, the difference between the number of pairs can vary up to 200 (FBwall dataset). For all different datasets, the red line converges faster to the total number of pairs and follows the green line ($\alpha = 0.5$) and the blue line ($\alpha = 0.999$). On the other hand, the blue line converges always faster in the case of spatial distance and is followed by the green and the red lines. These results are as expected and match the desired behavior of the parameter α: smaller values of α means less importance on the spatial distance, while higher values of α reduces the weight of temporal dimension, and thus, the SFPs are more likely to expand over several timestamps. SFPs affect directly TBC, and therefore, we expect that varying α will impact the value of TBC, as we show in the following subsection.

6.2 Temporal betweenness centrality

We next characterize the behavior of TBC against static BC, against time, and against α.

TBC versus Static BC. We compare the ranking of importance of the vertices by means of TBC, with the rankings that we can obtain by applying static BC to the dynamic network. In particular, we consider three ways of applying static BC to a dynamic network: i.e., *maximum*, *average* and *union graph*. For the maximum and average characterizations, we fix a window length and we run the static BC algorithm at each one of the snapshots of the window. The value of BC of each vertex is the maximum and the average of the values in these snapshots. The union graph characterization is obtained by merging the snapshots of the graph to create a static graph with the union of vertices and edges. The BC of each vertex is given by running the static BC algorithm on the union graph.

Figure 6 shows the results of the Jaccard and Kendall Tau similarity measures comparing the rankings of TBC versus maximum (max with red color), TBC versus average (avg

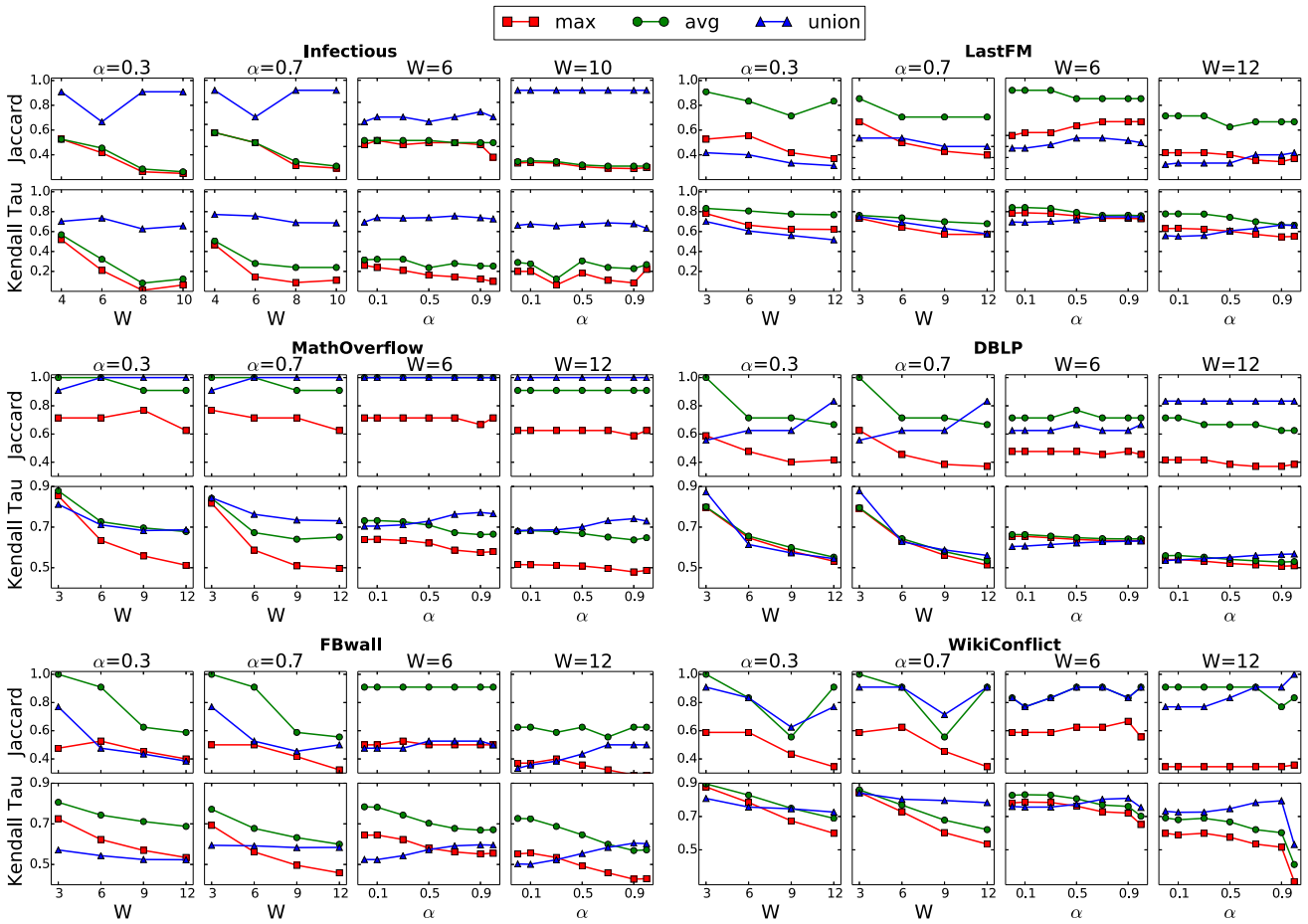

Fig. 6 Jaccard and Kendall Tau values for max (TBC vs. maximum), avg (TBC vs. average) and union (TBC vs. union graph). We present results for different values of $|W|$ and α for Infectious, LastFM, MathOverflow, DBLP, FBwall and WikiConflict datasets

with green color) and TBC versus union graph (union with blue color) for different values of $|W|$ and α. Jaccard is measured among the top-10 vertices by centrality value. It represents the similarity of TBC and maximum (max), TBC and average (avg) and TBC and union graph (union) top-10 sets. This similarity measure shows how the top-10 central nodes chosen by TBC are similar to the three characterizations. Kendall Tau is measured in the first 15% of the total rank of V by centrality, and discarding the vertices which is not common in the two compared rankings. This similarity measures shows the similarity of the rankings of the common vertices among the TBC and maximum (max), TBC and average (avg) and TBC and union graph (union) rankings.

As expected, both Jaccard and Kendall Tau similarities are higher for small windows, whereas for larger windows the importance of using a temporal notion of centrality increases, resulting in lower similarity values. In most of the cases, for the different values of α, max and avg get smaller values as we increase α. This situation is reversed for the union graph which, by construction, includes all temporal paths including not valid ones, i.e., paths that go back in time.

TBC versus Time. In this experiment, we present how the value of TBC changes in time for four vertices taken from the Infectious dataset and show significant changes depending on the choice of parameters. Figure 7 shows the ranking of the vertices in a range of 6 timestamps (timestamps 3 to 8) for three different window lengths $|W| = [4, 6, 8]$. On the y-axis, we report the ranking of the vertices in the network, where a value of 1 indicates the most central vertex in the graph. It is important to note here that depending on the window length the ranking of the vertices can change significantly. For example, we observe that vertex 3 (blue line) has a better ranking value comparing to vertex 1 (red line) for window $|W| = 4$ at timestamp 6. This situation changes when the window increases to length 6 and 8. This confirms our hypothesis that the length of the observation period can change dramatically the vertices' centrality.

Figure 8 shows the ranking of four vertices of MathOverflow dataset for timestamps 14–26. In this set of plots, we observe how the ranking of the vertices change while changing the length of W ($|W| = [9, 12, 15]$) and also the value of α ($\alpha = [0.001, 0.5, 0.999]$). For $\alpha = 0.5$ (second line plots) different values of $|W|$ result in different rankings. For example, vertex 2 (green line) has better rank than vertex 1 (red line) in window 9, which changes dramatically for windows 12 and 15. Finally, for a fixed $|W|$ and for different values of α, the rankings also vary significantly. For example, for $|W| = 12$ the rank of vertex 2 has improved w.r.t. vertex 4 in timestamps 17–21 as α gets bigger values.

TBC versus α. Next, we present the difference between the rankings of the vertices for various $|W|$ and for different α values for all datasets. Figure 9 shows the Jaccard and

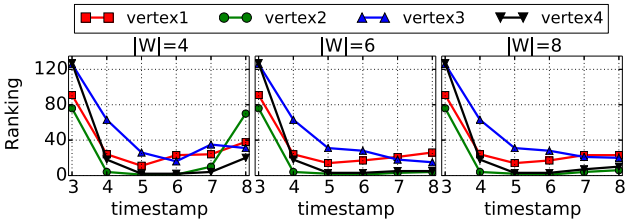

Fig. 7 In the Infectious network, the figure shows how the TBC values change along time for four different vertices. We use $|W| = [4, 6, 8]$ and $\alpha = 0.5$ for six consecutive timestamps (3–8)

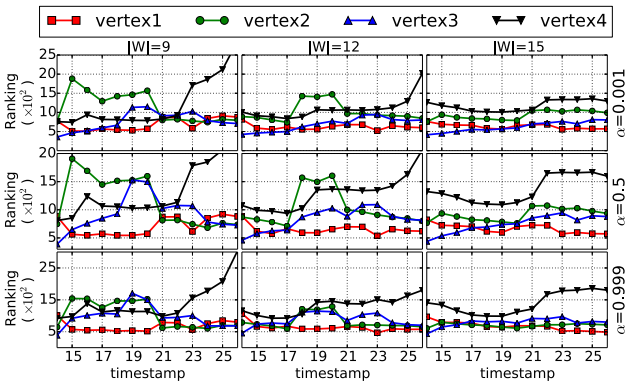

Fig. 8 In the MathOverflow network, we present TBC versus time for four different vertices that show significant changes depending on the choice of parameters. We use $|W| = [9, 12, 15]$ and $\alpha = [0.001, 0.5, 0.999]$ for 13 consecutive timestamps (14–26)

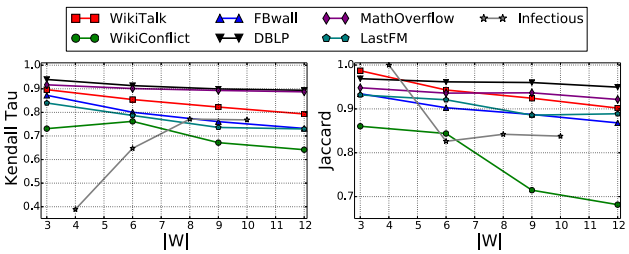

Fig. 9 Kendall Tau and Jaccard similarity for the rankings of TBC method for $\alpha = [0.001, 0.999]$ and different W for all datasets

Kendall Tau similarity measures for the top 15% of the vertices for with $\alpha = 0.001$ and $\alpha = 0.999$. For Infectious, we use $|W| = [2, 4, 6, 8]$ whereas for the rest of the datasets we use $|W| = [3, 6, 9, 12]$. For Infectious, we see that for $|W| = 4$ Kendall Tau is less than 0.4 whereas Jaccard similarity is 1. This means that the top-15% of the two rankings contain the same vertices but the ranking of the vertices is very different. We also see greater dissimilarity of the rankings as the window grows for the majority of the datasets. These results support our hypothesis that by giving different weights to spatial and time links (different values of α) vertices can have very different BCs.

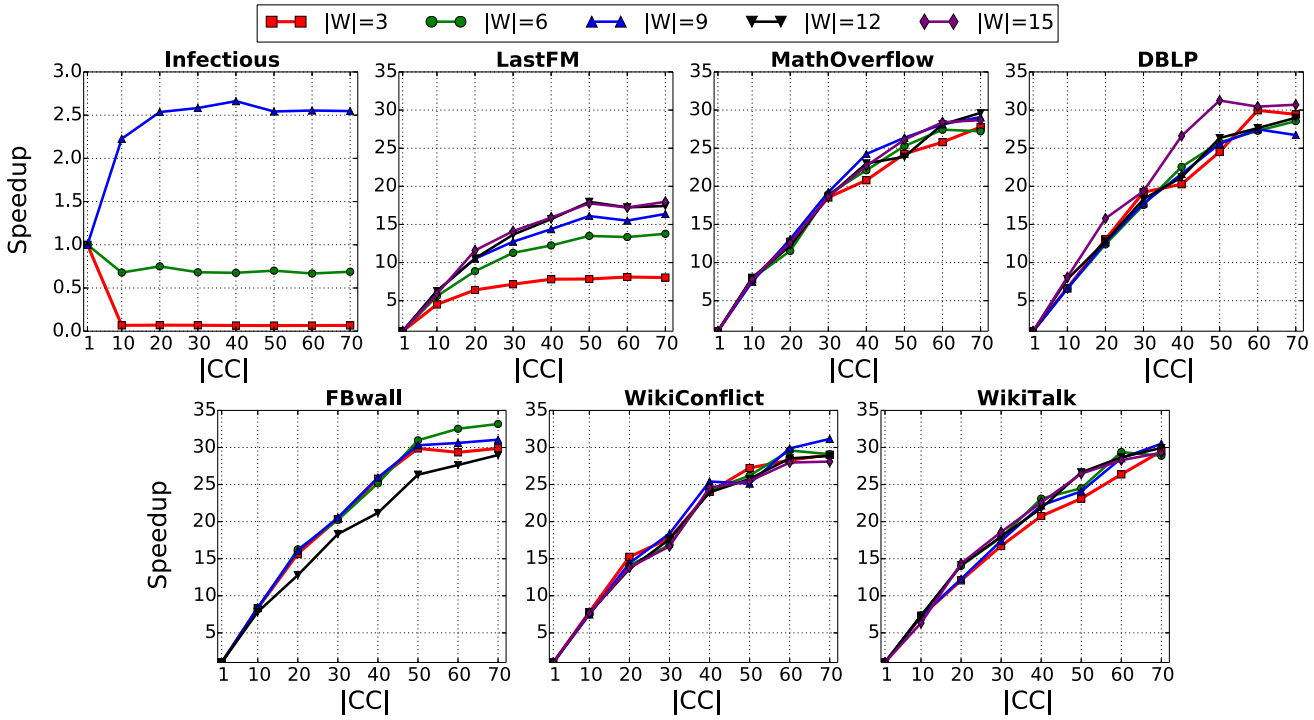

Fig. 10 Execution time versus $|CC|$ for the serial and distributed versions for all datasets and for various $|W|$

6.3 Scalability

Figure 10 shows the performance gain when we increase the number of compute cores (CC) for different $|W|$. We show results for all datasets and for $|W| = [3, 6, 9, 12, 15]$. We performed experiments for the serial version of our implementation ($|CC| = 1$) and for the distributed version using 10–70 compute cores. We define as speedup the fraction of the serial execution time over the distributed execution time, i.e., $\frac{t_{\text{serial}}}{t_{\text{distributed}}}$. For Infectious dataset, first plot from the left, we see that for small windows the distributed implementation does not scale, since the execution time for the serial version is very small. Therefore, the overhead added by the spark processed increases the latency. However, for larger windows ($|W| = 9$), we can observe a speedup up to 2.5. For the rest of the datasets, we see a speedup up to 18 for LastFM and as the size of the datasets increase we get speedup up to 30 for WikiTalk and for $|CC| = 70$.

7 Information propagation

Betweenness centrality can be used to quantify the importance of a vertex in a network in terms of its capacity of spreading information by bridging other vertices. In the temporal setting, this role is extended to a bridge between two other vertices that do not have contact at the same time. Therefore, betweenness centrality can be used to identify a good spreader of information during a period of time [21].

In this section, we compare the information propagation capability of vertices with high TBC measure against the three versions of static BC described before (maximum, average, and union graph) applied to temporal interaction networks, plus a pure static case, i.e., static BC applied to the static social network. For our experiment, we use the LastFM and DBLP datasets.

For assessing the capability of nodes in spreading information, we need to define a propagation model: in this experiment, we adopt the popular *independent cascade model* (IC) [21]. IC propagation model requires a directed and probabilistic graph where each directed arc (u, v) indicates that v is a follower of u, thus information that arrives in u can propagate over the arc and reach v. The probability $p(u, v)$ associated with each arc (u, v) indicates the influence of u over v, or in other terms, the probability of information propagating from u to v.

In the LastFM dataset, we have both the social graph and the additional information of which user listens to which song at which time. Consider two vertices (users), v_1 and v_2, which are connected in the friendship graph. If user v_1 listens to a song at timestamp t_1 and the user v_2 listens to the same song at a timestamp $t_2 > t_1$, there is a probability that the user v_1 has influenced the user v_2. Therefore, we construct the probabilistic graph (G_p), where each direct edge (v_1, v_2) is labeled with a probability $p_{(v_1,v_2)} = \frac{\#\text{interactions}(v_1 \to v_2)}{\#\text{actions}(v_1)}$ i.e., the vertex v_1 can influence the vertex v_2, as it is described by [12].

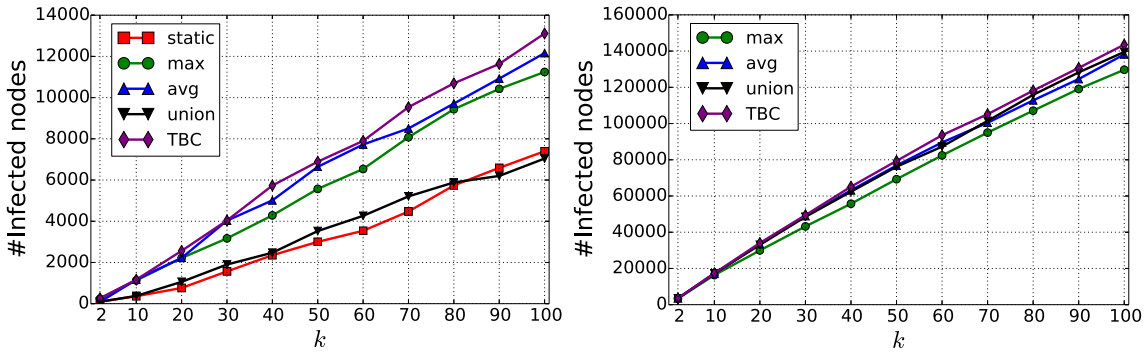

Fig. 11 Number of infected vertices on the graph when we infect the top-k vertices of each ranking for LastFM dataset (left plot) and DBLP dataset (right plot)

For computing TBC, we use the temporal network $\mathbf{G}_W$, using all the 314 timestamps as W, with $\alpha = 0.5$. On the same network, we also compute the three notions of static BC over temporal networks (i.e., maximum, average, and union graph). Finally, we also compute standard BC over the social graph (we call this case "static").

For each of these measures, we select the top-k users as seed set and run IC propagations over the probabilistic graph G_p. For each run, we compute the number of "infected" vertices starting from the seed set and report the average of this measure over 10,000 runs.

For the DBLP dataset, we use a unique temporal window $|W| = 12$ (2000–2011 with yearly granularity) and $\alpha = 0.5$ as before. The probabilistic directed graph (G_p) is constructed by first taking the union graph of the 12 years, by considering each undirected edge as the two directed arcs, and setting all the edge probabilities uniformly to 0.1.

Figure 11 (left plot) shows the results for LastFM dataset for various k values and for the static case, max, avg, union and TBC. If we select the top-k vertices of TBC we get always more vertices infected with up to 1035 more vertices from the second better method (avg) for $k = 70$. Finally, union and static methods give the least number of infected vertices. Figure 11 (right plot) shows the results for DBLP dataset, where TBC outperforms the second better method up to 3982 more vertices for $k = 60$.

It is worth noting how the benefits of using the definition of temporal betweenness centrality are more evident in the dataset (LastFM) in which some temporal information is maintained in the definition of the influence probabilities, than in the dataset in which no temporal information is used in the definition of the influence probabilities (DBLP).

8 Conclusions

We present a general definition of shortest paths in temporal networks that integrates and extends previous definitions. Based on this definition, we introduce a novel metric of temporal betweenness centrality that is highly sensitive to changes on the dynamic graphs, as well as the observation period and the different importance given to the temporal span over the spatial distance covered by a path. We present a fast exact algorithm to compute betweenness centrality for window graphs and we prove its correctness. We present our distributed implementation in Apache Spark that allows us to scale the computation of temporal betweenness centrality in dynamic graphs.

Finally, an application to the analysis of information propagation proves that our notion of temporal betweenness centrality outperforms static betweenness centrality in the task of identifying the best spreaders of information in a network.

For future work, we plan to optimize the computation of betweenness centrality in the sliding window by only updating the values that change in a streaming fashion. Moreover, due to its high computational complexity, it is important to study approximated methods in the dynamic graph settings: in this regards, a good starting point could be to extend the existing approximated methods for betweenness centrality in static graphs.

References

1. AlGhamdi, Z., Jamour, F., Skiadopoulos, S., Kalnis, P.: A benchmark for betweenness centrality approximation algorithms on large graphs. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM), p. 6 (2017)
2. Ang, C.S.: Interaction networks and patterns of guild community in massively multiplayer online games. Soc. Netw. Anal. Min. **1**, 341 (2011)
3. Anthonisse, J.: The rush in a directed graph. Technical Report, Stichting Mathematisch Centrum (1971)
4. Bergamini, E., Meyerhenke, H.: Fully-dynamic approximation of betweenness centrality. In: Algorithms-ESA 2015, pp. 155–166. Springer, Berlin (2015)

5. Bergamini, E., Meyerhenke, H., Ortmann, M., Slobbe, A.: Faster betweenness centrality updates in evolving networks. In: 16th International Symposium on Experimental Algorithms, SEA 2017, June 21–23, 2017, pp. 23:1–23:16, London (2017)
6. Brandes, U.: A faster algorithm for betweenness centrality. J. Math. Sociol. **25**, 163–177 (2001)
7. Brandes, U., Kenis, P., Lerner, J., van Raaij, D.: Network analysis of collaboration structure in Wikipedia. In: Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20–24, pp. 731–740 (2009)
8. Bui-Xuan, B., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. Int. J. Found. Comput. Sci. **14**(2), 267–285 (2003)
9. Catanese, S., Ferrara, E., Fiumara, G.: Forensic analysis of phone call networks. Soc. Netw. Anal. Min. **3**, 15–33 (2012)
10. Freeman, L.: A set of measures of centrality based on betweenness. Sociometry **40**, 35–41 (1977)
11. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Natl. Acad. Sci. USA **99**, 7821–7826 (2002)
12. Goyal, A., Bonchi, F., Lakshmanan, L.V.S.: Learning influence probabilities in social networks. In: WSDM (2010)
13. Green, O., McColl, R., Bader, D.A.: A fast algorithm for streaming betweenness centrality. In: Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Conference on Social Computing (SocialCom), pp. 11–20 (2012)
14. Gunturi, V.M., Shekhar, S., Joseph, K., Carley, K.M.: Scalable computational techniques for centrality metrics on temporally detailed social network. Mach. Learn. **106**(8), 1133–1169 (2017)
15. Habiba, H., Tantipathananandh, C., Berger-Wolf, T.Y.: Betweenness centrality measure in dynamic networks. DIMACS Technical Report 2007-19 (2007)
16. Hayashi, T., Akiba, T., Yoshida, Y.: Fully dynamic betweenness centrality maintenance on massive networks. Proc. VLDB Endow. **9**(2), 48–59 (2015)
17. Isella, L., Stehlé, J., Barrat, A., Cattuto, C., Pinton, J., Van den Broeck, W.: What's in a crowd? Analysis of face-to-face behavioral networks. J. Theor. Biol. **271**(1), 166–180 (2011)
18. Jamour, F., Skiadopoulos, S., Kalnis, P.: Parallel algorithm for incremental betweenness centrality on large graphs. IEEE Trans. Parallel Distrib. Syst. **29**, 659–672 (2018)
19. Jeong, H., Mason, S., Barabási, A., Oltvai, Z.: Lethality and centrality in protein networks. Nature **411**, 41 (2001)
20. Kas, M., Wachs, M., Carley, K.M., Carley, L.R.: Incremental algorithm for updating betweenness centrality in dynamically growing networks. In: 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), pp. 33–40 (2013)
21. Kempe, D., Kleinberg, J., Tardos, E.: Maximizing the spread of influence through a social network. In: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'03 (2003)
22. Kempe, D., Kleinberg, J.M., Kumar, A.: Connectivity and inference problems for temporal networks. J. Comput. Syst. Sci. **64**(4), 820–842 (2002)
23. Kim, H., Anderson, R.: Temporal node centrality in complex networks. Phys. Rev. E **85**(2), 026107 (2012)
24. Kourtellis, N., Morales, G.D.F., Bonchi, F.: Scalable online betweenness centrality in evolving graphs. IEEE Trans. Knowl. Data Eng. **27**(9), 2494–2506 (2015)
25. Lee, M.-J., Choi, S., Chung, C.-W.: Efficient algorithms for updating betweenness centrality in fully dynamic graphs. Inf. Sci. **326**, 278–296 (2016)
26. Leskovec, J., Huttenlocher, D.P., Kleinberg, J.M.: Governance in social media: a case study of the Wikipedia promotion process. In: Proceedings of the 4th International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23–26 (2010)
27. Liljeros, F., Edling, C., Amaral, L., Stanley, H., Aberg, Y.: The web of human sexual contacts. Nature **411**, 907 (2001)
28. Maglaras, L.A., Katsaros, D.: New measures for characterizing the significance of nodes in wireless ad hoc networks via localized path-based neighborhood analysis. Soc. Netw. Anal. Min. **2**, 97–106 (2012)
29. Mislove, A., Viswanath, B., Gummadi, K.P., Druschel, P.: You are who you know: inferring user profiles in online social networks. In: Proceedings of the 3rd ACM International Conference on Web Search and Data Mining, WSDM'10 (2010)
30. Ni, P., Hanai, M., Tan, W.J., Wang, C., Cai, W.: Parallel algorithm for single-source earliest-arrival problem in temporal graphs. In: 2017 46th International Conference on Parallel Processing (ICPP), pp. 493–502 (2017)
31. Paranjape, A., Benson, A.R., Leskovec, J.: Motifs in temporal networks. In: Proceedings of the 10th ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, UK, February 6–10, 2017, pp. 601–610 (2017)
32. Pereira, F.S.F., de Amo, S., Gama, J.: Evolving centralities in temporal graphs: a Twitter network analysis. In: IEEE 17th International Conference on Mobile Data Management, MDM2016, Porto, Portugal, June 13–16, 2016—Workshops, pp. 43–48 (2016)
33. Pontecorvi, M., Ramachandran, V.: Fully dynamic betweenness centrality. In: Algorithms and Computation—26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9–11, 2015, Proceedings, pp. 331–342 (2015)
34. Rad, A.A., Flocchini, P., Gaudet, J.: Computation and analysis of temporal betweenness in a knowledge mobilization network. Comput. Soc. Netw. **4**, 5 (2017)
35. Riondato, M., Kornaropoulos, E.M.: Fast approximation of betweenness centrality through sampling. In: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, WSDM'14, pp. 413–422, New York (2014)
36. Riondato, M., Upfal, E.: Abra: approximating betweenness centrality in static and dynamic graphs with rademacher averages. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1145–1154 (2016)
37. Shekhar, S., Brugere, I., Gunturi, V.M.: Modeling and analysis of spatiotemporal social networks. Encycl. Soc. Netw. Anal. Min. **2014**, 950–960 (2014)
38. Tang, J., Musolesi, M., Mascolo, C., Latora, V., Nicosia, V.: Analysing information flows and key mediators through temporal centrality metrics. In: Proceedings of the 3rd Workshop on Social Network Systems, SNS'10, pp. 3:1–3:6, New York (2010)
39. Viswanath, B., Mislove, A., Cha, M., Gummadi, P.K.: On the evolution of user interaction in Facebook. In: Proceedings of the 2nd ACM Workshop on Online Social Networks, WOSN 2009, Barcelona, Spain, August 17, pp. 37–42 (2009)
40. Williams, M.J., Musolesi, M.: Spatio-temporal networks: reachability, centrality and robustness. Open Sci. **3**(6), 160–196 (2016)
41. Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y.: Path problems in temporal graphs. Proc. VLDB Endow. **7**(9), 721–732 (2014)
42. Wu, H., Cheng, J., Ke, Y., Huang, S., Huang, Y., Wu, H.: Efficient algorithms for temporal path computation. IEEE Trans. Knowl. Data Eng. **28**(11), 2927–2942 (2016)