Mach Learn (2017) 106:1133–1169
DOI 10.1007/s10994-016-5583-7

Scalable computational techniques for centrality metrics on temporally detailed social network

Venkata M. V. Gunturi[1] · Shashi Shekhar[2] · Kenneth Joseph[3] · Kathleen M. Carley[3]

Received: 14 March 2015 / Accepted: 5 July 2016 / Published online: 8 September 2016

Abstract Increasing proliferation of mobile and online social networking platforms have given us unprecedented opportunity to observe and study social interactions at a fine temporal scale. A collection of all such social interactions among a group of individuals (or agents) observed over an interval of time is referred to as a temporally-detailed (TD) social network. A TD social network opens up the opportunity to explore TD questions on the underlying social system, e.g., "How is the betweenness centrality of an individual changing with time?" To this end, related work has proposed temporal extensions of centrality metrics (e.g., betweenness and closeness). However, scalable computation of these metrics for long time-intervals is challenging. This is due to the non-stationary ranking of shortest paths (the underlying structure of betweenness and closeness) between a pair of nodes which violates the assumptions of classical dynamic programming based techniques. To this end, we propose a novel computational paradigm called epoch-point based techniques for addressing the non-stationarity challenge of TD social networks. Using the concept of epoch-points, we develop a novel algorithm for computing shortest path based centrality metric such as betweenness on a TD social network. We prove the correctness and completeness of our

Editors: Céline Rouveirol, Rushed Kanawati, and Ruggero G. Pensa.

✉ Venkata M. V. Gunturi
gunturi@iiitd.ac.in

Shashi Shekhar
shekhar@cs.umn.edu

Kenneth Joseph
kjoseph@cs.cmu.edu

Kathleen M. Carley
kathleen.carley@cs.cmu.edu

1 Department of Computer Science and Engineering, IIIT, Delhi, India

2 Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, USA

3 Institute for Software Research, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

algorithm. Our experimental analysis shows that the proposed algorithm out performs the alternatives by a wide margin.

Keywords Dynamic programming · Time-varying social networks · Time varying networks · Graph theory · Centrality

1 Introduction

Given the ever-increasing proliferation of mobile and online social networking platforms, social interactions can increasingly be observed and studied at a fine temporal scale. Data from these platforms typically consists of time-stamped social events. These social events can represent several types of social interactions such as emails, phone calls, post-comment interactions, co-location pairs, etc. A time-stamped social event can be described as a relational tuple $<A_i, A_j, T>$, where A_i and A_j are the two individuals (or agents) who were interacting, and T is the time-stamp when this was observed. Such an event creates a temporary link between A_i and A_j, which can then be used in a time-aware analysis. A time-aware analysis can either consider the social events individually (e.g. change detection in streams) or could first create a set of frames containing interactions among members of a specific social system (e.g., employees of a university). Here, each frame would either contain events that occurred at a specific time point or comprise of all the events spread over a certain time duration (e.g., hour, day, week, etc.). A collection of all such frames (over a time window) recording interactions among the members of the same social system is referred to as a Temporally Detailed (TD) social network in this paper. Figure 1 illustrates a sample TD social network among W, X, A, C, D, U, U and Y. Here, an edge between two individuals denotes a social interaction.

In this paper, given a TD social network, we explore the computational challenges underlying a potential realization of what we refer to as temporally-detailed (TD) social network

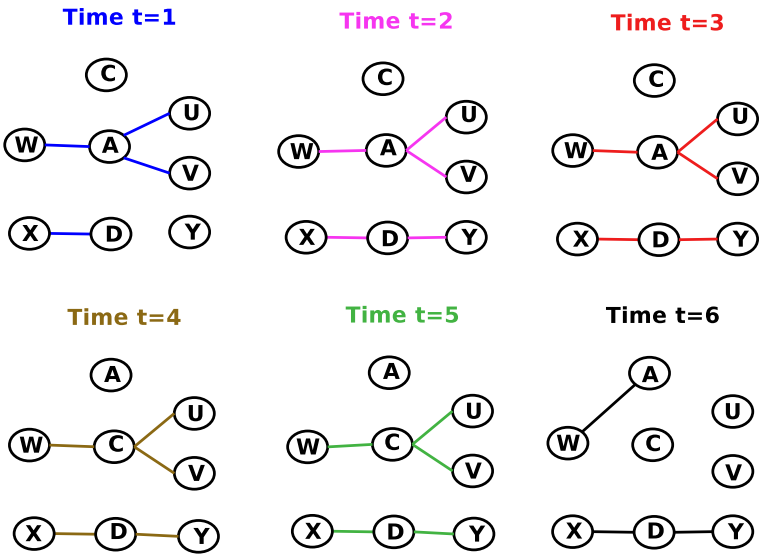


Fig. 1 Sample temporally-detailed social network (best in color)

analytics. TD social network analytics can answer novel time-aware questions on the underlying social system. Examples include: "How did the betweenness centrality of an individual vary over past year?" "How is the closeness centrality of a person changing over time?" Etc. To this end, several pilot studies (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012; Kim et al. 2012) have extended the traditional centrality metrics such as Betweenness and Closeness for TD social networks. These extended metrics capture the dynamics of the underlying social system and thus provide a better opportunity to understand its evolutionary behavior.

However, a TD social network still poses significant challenges for a computationally scalable realization of TD social network analytics. For instance, the shortest path between any two individuals in a TD social network—a key component in the extended versions of Betweenness and Closeness (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012)—is not stationary across time. As an example, in our previous TD social network shown in Fig. 1, the shortest path between W and U passes through A for times $t = 1$, $t = 2$ and $t = 3$. Whereas, the same passes through C for times t = 4 and t = 5. With the intention of keeping the example simple, we did not consider the temporal order of edges in this example, which is important while analyzing a TD social network as noted by other works as well (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012). It is important to note that non-stationarity can be seen when we consider temporal order of edges as well. If we consider the temporal order of the edges in Fig. 1, then the shortest path between W and U passes through A only for times $t = 1$ and $t = 2$. This non-stationarity present in TD social networks violates the stationary ranking assumptions (Russel and Norwig 1995; Bertsekas 1987) of classical dynamic programming (DP) based techniques (e.g. Dijkstras and its variants), and thus, makes it non-trivial to use them for computing the temporal extensions of shortest-path-based centrality metrics such as Betweenness and Closeness.

1.1 Related work

Relevant literature includes the following: (a) work done in the area of shortest path based centrality metrics for dynamic social networks and, (b) work done in the general area of shortest paths for time-varying networks.

Work in dynamic social networks In this area, for shortest path based metrics (e.g., temporal extensions of Betweenness and Closeness), the current works (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012) intrinsically resort to re-computing the shortest paths (using a DP based technique) for all the times (for which the network is desired to be analyzed) while calculating the metric. This approach yields a correct answer as given a particular time point, the ranking among paths is stationary. One could indeed use a DP based technique to re-compute the shortest paths (or the shortest path tree) for the all desired time-points and then process them to determine the centrality metric. While such an approach provides a correct answer, it performs redundant work across times sharing a common solution. For instance, in our previous sample shown in Fig. 1, the shortest path tree rooted at W does not change across $t = 1$, $t = 2$ and $t = 3$. Thus, it would be computationally redundant to re-compute the shortest path tree for $t = 2$ and $t = 3$ as the solution did not change. Such redundant work becomes a major computational bottleneck in case of large TD social networks observed for a long time-interval.

Work in time-varying networks Literature in this area is rich with many algorithms (Demiryurek et al. 2011; Kanoulas et al. 2006; Ding et al. 2008; George et al. 2007; Nannicini et al. 2012, 2008; Delling and Nannicini 2008; Delling and Wagner 2007; Delling 2008; Delling and Wagner 2009) for computing shortest paths in a time-varying network.

This literature can be broadly divided into following categories: (a) Goal oriented algorithms (Kanoulas et al. 2006; Demiryurek et al. 2011; Nannicini et al. 2012, 2008; Delling and Nannicini 2008; Delling and Wagner 2007; Delling 2008; Delling and Wagner 2009), (b) Generic shortest path algorithms (George et al. 2007; George and Shekhar 2007) and, (c) Shortest path for multiple time points (Ding et al. 2008).

Goal oriented algorithms (Kanoulas et al. 2006; Demiryurek et al. 2011; Nannicini et al. 2012, 2008; Delling and Nannicini 2008; Delling and Wagner 2007; Delling 2008; Delling and Wagner 2009) are techniques which have been "fine tuned" to find a shortest path between *two nodes* for particular time-point as efficiently as possible. Several novel search strategies were proposed which could streamline the search to a small candidate space. Examples of some of these strategies include, bi-directional search (Nannicini et al. 2012, 2008; Delling and Nannicini 2008), routing based on landmarks (Delling and Wagner 2007; Delling 2008; Delling and Wagner 2009), hierarchical partitioning of the underlying network (Demiryurek et al. 2011) and other A* based techniques (Kanoulas et al. 2006). However, for the kind of analytical questions we are aiming to support (e.g., how is the betweenness centrality of an individual changing over time?), we need techniques which can compute shortest path trees rooted at a node for a range of time points. One could indeed use these goal oriented techniques to compute shortest path between all-pairs of nodes for all times and then process them to compute the centrality metric. This however would be extremely inefficient as we would not be able to use efficient path counting techniques such as Brandes (2001) which work on shortest path trees. Furthermore, it would be highly counter intuitive to use a "goal-oriented" technique to compute shortest path to every node in the network from a source node.

Generic methods such as George et al. (2007), George and Shekhar (2007) have adapted Dijkstra's algorithm for computing shortest path tree rooted at a node for given time-point of interest. We have used this approach as our baseline approach in the experiments.

The most relevant work for our work would be Ding et al. (2008) (here after referred to as the LTT algorithm) which attempts to efficiently compute shortest path trees for multiple time points by avoiding redundant re-computations across time-points sharing a common solution. For instance, consider our earlier example of shortest path tree of W (Fig. 1) not changing across times $t = 1$, $t = 2$ and $t = 3$. LTT would attempt *not to* recompute the tree for all the three times $t = 1$, $t = 2$ and $t = 3$. The nature LTT algorithm is such that it avoids re-computation by comparing cost of candidate path in a tree (and thus the tree itself in some sense) against the lowest possible cost of its alternatives. While this also gives some reduction, its conservative nature may not always yield maximum possible reduction. Following example illustrates this approach at its inner most core level. Consider a case where we have two candidate paths P_1 and P_2 to a node v. Cost of P_1 was [1 3 4 4 5 7 9], which means that its cost was 1, 3, 4, 4, . . . ,etc for times $t = 0, 1, 2, 3, \ldots$, etc. On the other hand, assume that cost of P_2 was [3 3 5 5 5 5 5] over times $t = 0, 1, 2, 3, \ldots$, etc. Now the nature of the LTT algorithm is such that it can compare the cost of P_1 (the preferred path for $t = 0$) against 3 (lowest possible cost along the alternative path P_2) and would decide that the ranking changes (and thus re-compute to guarantee correctness) at time $t = 2$. However, in reality path P_1 has the lowest cost until time $t = 4$.

1.2 Key idea of the paper

In this paper, we propose to reduce redundant re-computation through a much more aggressive method. At its core-level, for our previous example with paths P_1 and P_2, our algorithm is able to compare the cost of paths across multiple instants of time and determine $t = 5$ as

the time instant when ranking changes. These aggressively determined change time-points are called as *Epoch-points*. Determining these epoch-points at large scale was made possible through our novel temporally detailed (TD) priority queue (detailed in Sect. 3).

In other words, we realize our novel approach of epoch-points through TD priority queue. More formally, Epoch-points are the time points when the shortest path tree rooted a node changes. For example, in our previous sample shown in Fig. 1, $t = 1$ (trivially), $t = 4$ and $t = 6$ form the epoch-points for the shortest path tree rooted at W. Epoch-points can be computed using one of the two strategies, pre-computing or lazy. In a pre-computing based method, all the candidate solutions (spanning trees) are enumerated and compared to determine the best tree for each time. This approach, however, would become a bottleneck in case of a large TD social network. To this end, we propose to use the lazy technique which computes epoch-points on-the-fly (through our temporally-detailed priority queue). The key insight in a lazy approach is that immediately after the computation of the shortest path tree for the earliest (nth) time, one has adequate information to forecast the first (nth) epoch-point. Epoch-point based algorithms for computing the temporal extensions of shortest-path-based centrality metrics such as Betweenness (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012) show better scalability by reducing the work across time-points sharing a common solution.

Contributions This paper makes following contributions:

1. Introduces the concept of epoch-point based approaches as a computational technique for temporally detailed analytics on TD social networks.
2. Propose a novel data structure called temporally-detailed priority queue helps in computing epoch-points on the fly (the lazy approach of computing epoch-points).
3. Propose an *epoch-point based algorithm called EBETS* for computing temporal extensions of Betweenness centrality. Note that EBETS can be trivially modified to compute temporal extensions of closeness centrality (Sabidussi 1966), graph centrality (Hage and Harary 1995) and stress centrality (Shimbel 1953) as well. For purposes of brevity, we chose not to include those extensions in the paper.
4. We also provide some guidelines on adapting our EBETS algorithm for metrics proposed in Habiba et al. (2007), Tang et al. (2010), Tang et al. (2009) and Kim and Anderson (2012).
5. Establish the correctness and completeness of the proposed approach.
6. Provide a detailed space and time complexity analysis of the proposed approach.
7. Evaluate EBETS experimentally via real datasets.
8. Compare EBETS against the related work via both experimental and asymptotic complexity analysis.
9. Provide an empirical comparison of temporal extension of betweenness centrality and traditional betweenness centrality on a real dataset.

Outline of the paper The rest of the paper is organized as follows. Section 2 describes the basic concepts (e.g., representation models for TD social networks) and defines the problem of temporally-detailed social network analytics. In Sect. 3, we propose the concept of epoch-points for the problem of temporally-detailed social network analytics and describe our temporally-detailed priority queue to compute them on-the-fly. We propose an epoch-point based algorithm called EBETS for computing the temporal extension of betweenness centrality in Sect. 4. Section 4.3 provides some guidelines on adapting EBETS algorithm to temporal extensions proposed in Habiba et al. (2007), Tang et al. (2010), Tang et al. (2009), Kim and Anderson (2012). In Sect. 5.1, we prove the correctness and completeness our epoch-point based algorithm. We provide a detailed space and time complexity analysis

our algorithm in Sect. 5.2. In the same section we also included text on comparing EBETS with DZ-AFP and LTT algorithms in terms of asymptotic complexity. The experimental evaluation of our algorithm is given in Sect. 6. Empirical comparison of temporal extension of betweenness centrality and traditional betweenness centrality in given in Sect. 7. We conclude this paper in Sect. 8.

Scope of the paper This paper focuses on developing a computational technique for only shortest path based metrics for dynamic social networks. Metrics based on all temporal paths (Lerman et al. 2010) are beyond the scope of this paper. Similarly, our experimental comparison with related work is limited to only a representative set of the state-of-the-art computational methods (George et al. 2007; Kanoulas et al. 2006; Ding et al. 2008) for temporally detailed social network analytics. Exhaustive experimental comparison with all the minor variants of the algorithms is out of scope of this paper.

2 Basic concepts and problem definition

2.1 Background on representational models

A temporally-detailed (TD) social network can be conceptually represented in multiple. Current representational models used for TD social networks can be broadly classified into following two: (a) Snapshot based or (b) Explicit flow-path based models. It is important to note that choice of model depends on the 'social phenomena' being studied. We now briefly describe both these kind of models and give an intuition on when they are preferred.

Snapshot model This model represents the temporally-detailed social network as a series of snapshots taken at different time instants. Here, each snapshot can either comprise of all the social events that happened at a specific time instant or can be an aggregation of all the events which happened over a certain time window (e.g., hour, day etc.). Figure 1 uses this model to represent a sample temporally-detailed social network. This kind of models are most suitable for studying temporally evolving patterns of activity of individuals (Braha and Bar-Yam 2006), creation of temporally linked structures (Eckmann et al. 2004), addition of social links (Kossinets and Watts 2006) etc.

Explicit flow-path models These models are based on the idea of representing the temporal relationship among events upfront. For example, a sample temporal relationship could be based on the idea of "which temporally ordered events could imply a potential flow of information in the network?" Importance of these kind of models was highlighted by many works (Nia et al. 2010; Howison et al. 2011; Kossinets et al. 2008; Tang et al. 2010; Lerman et al. 2010) which focused on studying information flow in a social network. The basic idea proposed by these works being that "Information flows along the network in time." To this end, representational models (Kim and Anderson 2012; Habiba et al. 2007) were proposed in literature which show the temporal relationship of social event upfront. We refer to them as 'explicit flow-path models.' Figure 2 illustrates one such approach with the social event data shown in Fig. 1. Here, the node corresponding to every agent is replicated across the time instants and directed edges, representing a potential information flow, are added in a "temporally-ordered" sense. For instance, consider a potential flow path $W \rightarrow A \rightarrow U$ which starts at W at $t = 1$ and reaches U at $t = 3$ via A (assuming that it takes one time unit for the information to flow on an edge). This flow path is implied through following two directed edges (shown in bold in Fig. 2) in the graph: (a) an edge between the copy of node W at $t = 1$ and A at $t = 2$, and (b) an edge between the copy of node A at $t = 2$ and U at $t = 3$.

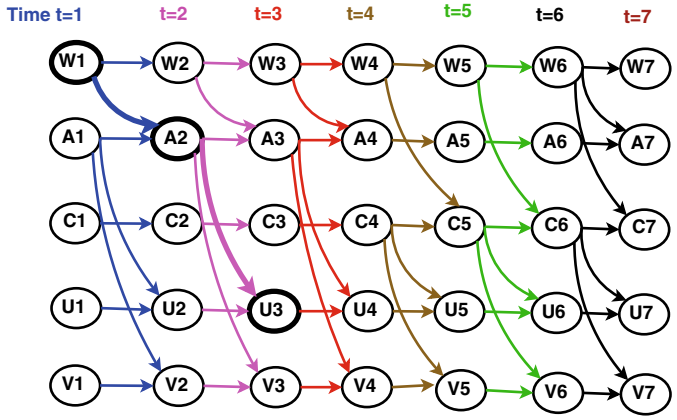


Fig. 2 Explicit flow-path models (best in color)

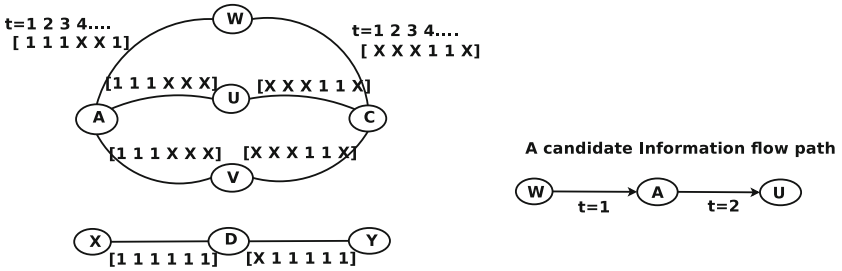


Fig. 3 Time aggregated graph

Snapshot models vs Explicit flow-path models Both snapshot and explicit flow-path models represent the same social event data but in two different ways. Further, one representation can be easily converted into another, i.e, snapshot based models can also be used to study flow based questions (by considering events across snapshots). And similarly, explicit flow-path based models can be used to study temporally evolving patterns, potentially seen upfront in a snapshot based model.

In this paper, we will use a *compressed* version of snapshot based models to describe our work. These are called as *Time aggregated graphs* (George and Shekhar 2007). In this model, we assign a time series, called an *interaction time series*, to each edge. This time series captures the time instants at which interactions have occurred. Figure 3 illustrates this model with the TD social network shown in Fig. 1. Here, edge (W, A) is associated with an interaction time series [111XXXX]. This implies that W interacted with A at times $t = 1, 2, 3$. Using this model, the information flow paths can also be easily retrieved by following the edges in a temporally-ordered fashion. Figure 3 also illustrates our previous information flow path $W \rightarrow A \rightarrow U$ which starts at W at $t = 1$ and reaches U at $t = 3$ via A.

Problem Definition: The problem of temporally-detailed social network analytics (TDSNA) is defined as follows:

Input: (a) Temporally-detailed social network; (b) Shortest path based centrality metrics (e.g., betweenness and closeness)

Output: An algorithm design paradigm for computing these metrics over an discrete interval of time.
Objective: Computational scalability.
Constraints: Correctness and completeness of the solution.

2.2 Basic concepts of temporal betweenness centrality metrics

Equation 1 formally defines the temporal betweeness centrality used in this paper. It is important to note that several works (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012) have defined temporal extensions of betweenness centrality. Section 4.3 puts our definition in perspective of the existing definitions and shows the interrelationship between ours and the ones from related work. Also note that the purpose of definition given in Eq. 1 is to just provide an intuitive base to easily describe our epoch-point based approach. Our approach can easily be adapted (without losing correctness) to other temporal extensions defined in Habiba et al. (2007), Tang et al. (2010), Tang et al. (2009), Kim and Anderson (2012) as well (details in Sect. 4.3).

In Eq. 1, $C_B(v)[t]$ denotes the betweenness centrality of a node v at time t. In the equation, $\sigma_{sd}[t]$ refers to the number of shortest paths between node s and node d at time t, and $\sigma_{sd}(v)[t]$ refers to the number of shortest paths between node s and node d that pass through node v at time t. Here, the concept of *"shortest path between node s and node d at time t"* is interpreted according to Definition 1.

$$C_B(v)[t] = \sum_{\forall s \neq v \neq d \in V} \frac{\sigma_{sd}(v)[t]}{\sigma_{sd}[t]} \tag{1}$$

Definition 1 (*Temporal Shortest Path*) A path $P_{sd} = s, x_1, x_2, x_3, \ldots, x_k, d$ between s and d, where s and x_1 interacted at time t, is considered to be a shortest path for time t if it has following two properties.

- Any individual (node of path) may appear only once in the path.
- All the nodes in P_{sd} should be visited in a *temporally ordered* sense. This means that if (x_i, x_j) and (x_j, x_k) are two consecutive edges in P_{sd} corresponding to the interactions at times $t_{x_i x_j}$ and $t_{x_j x_k}$, then $t_{x_i x_j} < t_{x_j x_k}$.
- If there is another path $Q_{sd} = s, y_1, y_2, y_3, \ldots, d$ between s and d, where s and x_1 interacted at time t, then the interaction time of the edge (y_k, d) is not less than that of (x_k, d), i.e, $t_{y_k d} \geq t_{x_k d}$.

In order to study the betweenness centrality of an individual over a time interval, Eq. 1 would have to be computed repeatedly for all the times in the time-interval of interest. Internally we would use a temporal generalization of path counting technique proposed in Brandes (2001). More specifically, the notion of *pair-dependencies* in Brandes (2001) was generalized to a temporal case. Equation 2 shows this generalization for the metric given in Eq. 1. Here, $\delta_{sd}(v)[t]$ denotes the *temporal pair-dependencies* on node v at time t.

$$\begin{aligned} C_B(v)[t] &= \sum_{\forall s \neq v \neq d \in V} \frac{\sigma_{sd}(v)[t]}{\sigma_{sd}[t]} \\ &= \sum_{s \neq v \neq d \in V} \delta_{sd}(v)[t] \quad \text{where } \delta_{sd}(v)[t] = \frac{\sigma_{sd}(v)[t]}{\sigma_{sd}} \\ &= \sum_{\forall s \in V} \delta_{s\cdot}(v)[t] \quad \text{where } \delta_{s\cdot}(v)[t] = \sum_{\forall d \in V} \delta_{sd}(v)[t] \end{aligned} \tag{2}$$

Here, a single $\delta_{s.}(v)[t]$ is computed using Eq. 3. This is a temporal generalization of Theorem 6 in Brandes (2001). This equation is computed using recursive fashion (using stacks), starting with the last node in a shortest path whose $\delta_{s.}()$ is initialized to zero.

$$\begin{aligned} \delta_{s.}(v)[t] &= \sum_{w:v\in P_s(w)[t]} \frac{\sigma_{sv}(v)[t]}{\sigma_{sw}[t]} \cdot (1+\delta_{s.}(w)[t]) \\ &where\ P_s(w) = \{u \in V : u\ is\ predecessor\ to\ w\ in\ a\ shortest\ path \\ &(According\ to\ Definition 1) from\ s\ for\ time\ t\} \end{aligned} \tag{3}$$

2.3 Importance of the proposed metric

It is valuable to have a temporally fine grain measure such as ours (Eq. 1). This of kind metric can help in studying social phenomena which are transient in nature (e.g., varying importance of an individual in a network). In general, one may observe that social relationships are dynamic and fluid (Palinkas et al. 1998; Sampson 1968; Wei and Carley 2015), more so than nearly any other element of our socio-cultural lives (Vaisey and Lizardo 2010). Consequently, there is a necessity for algorithms that can easily adapt to the level of dynamism which reflects the moment-to-moment shifts (Bucholtz and Hall 2005) in localized social structure. We believe our metric its algorithm (EBETS) would be a step forward towards capturing this dynamism.

Furthermore, a feature of time variant data is that nodes in the network enter and leave, and links become available or unavailable. Now if the temporal dimension is collapsed (by aggregating across times) and all data is kept, the resulting centrality would be quite misleading (more details given in case-study, Sect. 7). For example, when dealing with terror groups or rapidly changing regimes in a noisy data environment, this collapse often leads to individuals who are incarcerated or dead as being listed with the highest betweenness (Tambayong and Carley 2012). And lastly, instances of such dynamism go beyond human networks—in transportation networks, for example, knowing exactly when abrupt shifts in efficient pathways occur may help to improve travel as delays develop, rather than at the end of a "chunk" of aggregated network data (Cats and Jenelius 2014).

3 Computational structure of TD social network analytics

A nave approach for the TDSNA problem could be to repeatedly re-compute the shortest path tree using a DP based algorithm for each time. However, this would incur redundant re-computation across times sharing a common solution, leading to a severe computational bottleneck in case of long time-intervals for large TD social networks. To reduce this computational cost, we propose the notion of epoch-points which divide the given time-interval into a set of disjoint sub-intervals in which the ranking among alternative candidate paths is stationary. Within these intervals, the shortest path tree can now be computed using a single run of a dynamic programming (DP) based algorithm. The advantage of such an approach is that we would not compute the shortest path tree for many times in the given interval. We would re-compute only at the epoch-points without missing any time instant in the given time-interval where the shortest path tree can change. Note that the shortest path tree rooted at any node is not unique. It just represents 'a' shortest path to the internal and leaf nodes of the tree from its root. To this end, we define epoch-points in terms of the maximum duration of 'optimality' of the current shortest path tree. Epoch-points are formally defined in Definition 2.

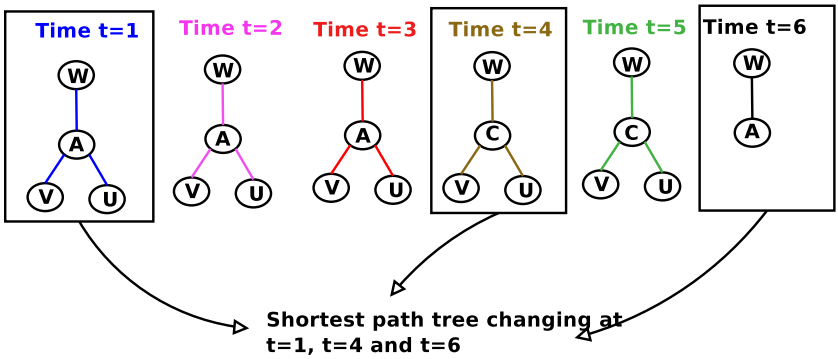


Fig. 4 Shortest path tree rooted at node W for different times

Definition 2 Epoch-Points Given a shortest path tree $STree(v)$ for a time $t = t_{cur}$ which is rooted at node v and contains nodes $U = \{u_1, u_2, \ldots, u_k\}$. The earliest time instant $t_{epoch} > t_{cur}$ is an epoch-point with respect to $STree(v)$ if one of the following holds:

1. There is a path P (at $t = t_{epoch}$) from v to some $u_i \in U$ which is shorter than the current path in $STree(v)$.
2. The current path to some $u_i \in U$ in $STree(v)$ no longer exists at $t = t_{epoch}$.
3. There is a node $w \notin U$ which is reachable from v at time $t = t_{epoch}$.

Figure 4 illustrates the proposed idea of epoch-points for the shortest path tree rooted at W for the TD social network shown in Fig. 1. The figure shows that the shortest path tree rooted at node W changes at $t = 1$ (trivially), $t = 4$, and $t = 6$ making them the epoch-points in this case. Note that the trees represented in Fig. 1 are constructed based on the snapshot model where the temporal order of edges was not considered. Basically, each shortest path tree in Fig. 4 corresponds to an instance of Dijsktra's (from node W) on each of the 6 snapshots in Fig. 1. These trees would be slightly different if we consider temporal order of edges (for modeling flows). We did not consider them in this example for ease of communication. Note, that the concept of Epoch-Points is valid for both snapshot and flow-path based studies. Just "what constitutes a shortest path?" changes across snapshot and flow-path based studies. In a flow-path based study, the edges must be considered in a temporally-ordered fashion.

A key challenge in designing an epoch-point based algorithm for the TDSNA problem is to minimize the amount of time needed to compute the epoch-points while ensuring correctness and completeness. Epoch-points can be computed in two ways, pre-computing strategy or a lazy strategy. In a pre-computing based method, all the candidate solutions (spanning trees) are enumerated and compared to determine the best tree for each time. This approach, however, may become a computational bottleneck in the case of a temporally-detailed social network, as there can be large number of candidate spanning trees. In this paper, we investigate the lazy technique, which computes epoch-points on-the-fly while exploring candidate paths. The key insight in a lazy approach is that immediately after the computation of the shortest path tree for the first (nth) time, one has adequate information to forecast the first (nth) epoch-point. In our previous example of shortest path tree of node W, this would mean that, after computing the shortest path at $t = 1$ we could forecast $t = 4$ as the next epoch-point for the re-computation to start. In other words, a single logical work-unit of the proposed algorithm would be: "compute the shortest path tree for one time and forecast the next epoch-point for re-computation."

Table 1 Priority queue versus temporally-detailed priority queue

Properties/operations	Priority queue	Temporally-detailed priority queue
Keys	Scalar values	Time-series of values
Ordering	Ascending or descending	Ascending or descending on a time index t_{ord}
Insert operation	Inserts scalar value	Inserts a time-series
Update-key operation	Updates a scalar value	Updates an entire time-series
Delete-key operation	Deletes a scalar value	Deletes an entire time-series
Extract-min	Returns a scalar with lowest cost	Returns a time-series with least value of t_{ord}
Forecast-epoch-point	–Not available–	Maximum time $t_{ept} (> t_{ord})$ for which the previous extract-min has least value among all keys in the queue

An epoch-point is forecasted through the use of a modified priority queue, which we call as a ***temporally-detailed (TD) priority queue***. Table 1 illustrates its properties in contrast the regular priority queues. Unlike a regular priority queue, whose elements are scalar values, a TD priority queue is composed of time-series of values. For our shortest path tree problem, a time-series in this queue would contain the cost along a path to a node in the open list for the times under consideration. The queue is ordered on the current time (t_{ord} in Table 1) under consideration. For instance, in our previous example of shortest path tree of node W over the time interval [1 6], we would order the queue based on costs for $t = 1$ for the first logical work-unit and then for the forecasted epoch-point for the second work unit, and so on. Apart from the standard priority queue operations such as insert, extract-min, update-key, we have a new operation called "forecast-epoch-point." This new operation is called at the end of each extract-min, at that stage it compares the time-series corresponding to the extract-min with the other time-series in the queue to determine the maximum time duration (beyond t_{ord}) for which the chosen time-series has the lowest value. Ties are broken arbitrarily. This value is stored in an auxiliary data-structure as a "potential epoch-point." At any stage during the execution, we have the following loop invariant:

Proposition 1 *The choices (i.e., extract-mins from the TD priority queue) made during execution hold their validity for all the times between the time for which the queue is ordered (trivially) and the* min*{potential epoch-points observed so far}.*

Using this loop invariant and exploring the paths in a greedy fashion (similar to Dijkstra's enumeration style), when we expand (extract-min + forecast-epoch-point) a node v, we have a shortest path between the root of the tree to v which is optimal for all the times between t_{ord} (on which the queue was ordered) and the min{potential epoch-points observed so far}. This min{potential epoch-points} then becomes the forecasted epoch-point when the re-computation starts.

4 Epoch based betweenness centrality

This section describes a general epoch-point based algorithm for computing temporal extension of betweenness centrality on a temporally-detailed social network. A key component of

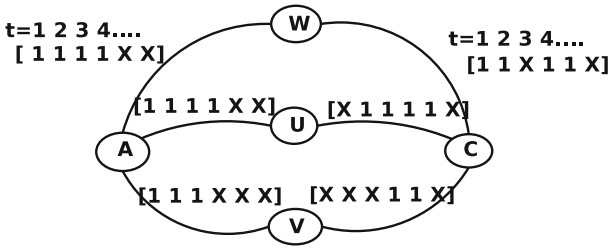


Fig. 5 Sample temporally-detailed social network

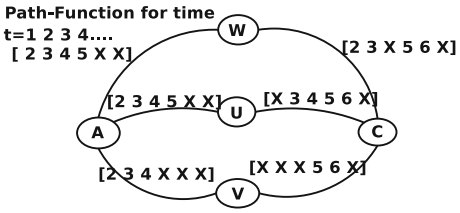
this computation would be to determine shortest paths between all pairs of nodes for either all the times or for a given range of time instants. As mentioned previously, these shortest paths could change with time. As soon as the optimal shortest path to a node in the tree changes, we deem the shortest path tree to have changed (refer Definition 2). The time points where these changes happen are termed as the epoch-points (ref Definition 2). These epoch-points need to be determined in order to efficiently compute a temporal extension of betweenness centrality (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012) for several time points in a TD social network. Note that, even if the shortest path tree changes according to Definition 2, the overall betweenness centrality value may not change. However, in order to ensure correctness, we re-compute the shortest path tree (and the betweenness centrality) for all the epoch-points as defined in Definition 2.

In order to determine the epoch-points using a temporally-detailed (TD) priority queue, we need to model the total cost of a path for different time instants which would be inserted in the TD priority queue as keys. We propose to do this via—what we refer to—as a *path-function*. A formal definition of the path-function is given below.

Definition 3 (*Path-function*) A path function is a time series that represents the total cost of the path from the root of the tree to the end node of path as a function of time. A path function is determined by combining the interaction time series of its component edges in a suitable fashion, e.g., in a time-ordered way for modeling information flow.

For instance, consider the path $<W, A, U>$ in the TD social network shown in Fig. 5. This path contains two edges, (W, A) and (A, U). The interaction time series of edge (W, A) is [1 1 1 1 X X], while that of (A, U) is [1 1 1 1 X X]. Depending on the nature of the 'social phenomena' being studied, these interaction time-series can be combined in multiple ways. If the 'social phenomena' of interest is preferable via a snapshot model, then these can be simply added (interpreting 'X' as $+\infty$) to create the path function of $<W, A, U>$ over time instants [1 6] as [2 2 2 2 X X]. This means that length of the path $<W, A, U>$ for times $t = 1, 2, 3, 4, \ldots$ is two edges, and the path is undefined for times $t = 5, 6$.

On other hand, if the 'social phenomena' being studied requires modeling information flows then we need to consider the temporal order of interaction events on the edges (Nia et al. 2010; Howison et al. 2011; Kossinets et al. 2008; Tang et al. 2010; Lerman et al. 2010). Further, the cost of the path is noted in terms of the arrival time at the end node of the path (Habiba et al. 2007). In other words, cost of a path, e.g. S-X-Y, for time $t = \alpha$ is interpreted as "If I start at S at time $t = \alpha$ and traverse S-X-Y, then when (a time $t > \alpha$) can I take an outgoing edge from Y?" This boils down to considering following two aspects: (a) How much time does it take for the information to flow on an edge? and (b) Is information allowed to wait at a node? Depending on our choices for item (a) and item (b), the resulting path-function may be different. We now illustrate the procedure for constructing path-function for modeling flows under following design decisions: For item (a), we assume it takes 1 time unit



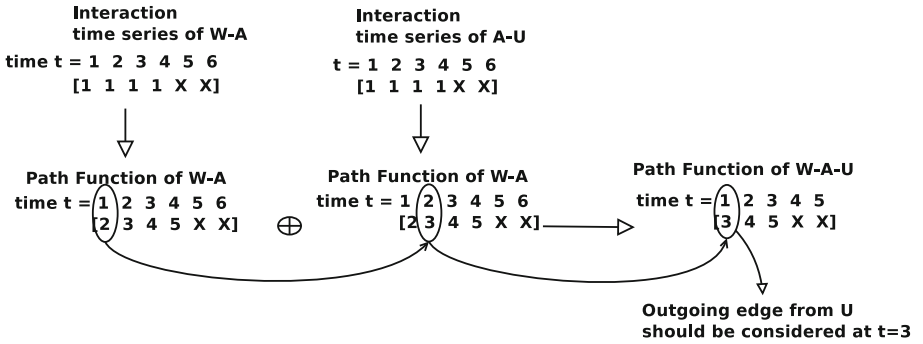


Fig. 6 Path functions for all the singleton edges

Fig. 7 Path function computation for modeling information flows

for the information to travel on an edge. And for item (b), we assume that the information is not allowed to wait. Note that these assumptions are only for illustration purposes, and are not constraints on our epoch-point based approaches. Usually, these design decisions would depend on nature of the 'social phenomena' being studied.

Consider the interaction time series of edge (W, A): [1 1 1 1 X X] which denotes that W and A interacted at times $t = 1, 2, 3$ and 4. Following our previous assumptions for item (a) and item (b), we interpret this interaction time series in following way: If we start at W at time $t = 1, 2, 3, 4$, we can take the next outgoing edge from A at times $t = 2, 3, 4, 5$. Note that 'X' is interpreted as "cannot follow this edge" (implemented as $+\infty$). This gives the path-function [2 3 4 5 X X] for the edge (W, A). Similarly, we would get the path-function [2 3 4 5 X X] for the edge (A, U). Fig. 6 shows the path-functions computed in this way for all the singleton edges [(W, A), (W, C), (A, U), etc.] in the TD social network illustrated in Fig. 5. Now consider the case of determining the path function of $<W, A, U>$. Figure 7 illustrates this process. If we take the edge W-A at time $t = 1$, we would be able to edge (A-U) at time $t = 2$ and then take the next outgoing edge from node U at time $t = 3$. Thus, the value of path-function of $<W, A, U>$ for $t = 1$ would be 3. Following this process, the complete path-function of $<W, A, U>$ for $t = 1, 2, 3, 4, 5$ (under the previous assumptions for item (a) and item (b)) would be [3 4 5 X X]. Note that a path function for $<W, A, U>$ modeling flows cannot be defined for $t = 6$ for the network shown in Fig. 5 because the network was not 'observed' for $t = 7$ which is needed if a flow has to modeled for $t = 6$.

As mentioned previously, different path-functions would be obtained if waiting is allowed or it takes more than 1 time unit for the information to travel on a edge. For instance, consider the case when waiting is allowed. Waiting is a natural consequence when the goal is to model the notion of "lifetime" of an information unit (Weng et al. 2012; Wang et al. 2011; Wu and Huberman 2007) flowing through the network. Waiting can be implemented by incorporating the notion of ***maximum wait allowed*** into the path-function construction. For example, consider the interaction time series of (C, V) [X X X 1 1 X]. Under the no waiting

scheme, this translated to the path-function: [X X X 5 6 X] (see Fig. 6). Now, consider the case when an information unit is allowed to wait for a maximum of 2 time units at any node. This means if the information has to flow through the edge (C, V) it must arrive at node C (or V as this is an undirected edge) by times $t = 2, 3, 4$ or 5. Case of $t = 4$ and $t = 5$ is trivial as C and V interacted at these times. However, $t = 2$ and $t = 3$ imply that the information waits at C (or V) for two and one time units respectively. In this case, we would get the following path-function for (C, V): [X 5 5 5 6 X]. Note the value 5 for times $t = 2$ and $t = 3$ in this path-function, these capture the notion of maximum wait allowed to be 2 time units. It is important to note that our proposed epoch-based approaches are transparent to the way the path-functions are constructed. It retains correctness across both snapshot style and information flow style paths. However, waiting does play a role on the computational performance of the proposed algorithm. To this end, we vary this parameter in our experiments (Sect. 6).

Given path-functions of candidate paths, epoch-points are determined by computing the earliest intersection point between the path functions and 'X' is interpreted as $+\infty$.

4.1 Epoch-point based BETweenness centrality Solver

Our proposed Epoch-point based BETweenness centrality Solver (EBETS) efficiently computes the epoch points *on the fly* without enumerating all the possible candidate paths. We do this by using a search and prune strategy (similar to Dijkstra's algorithm) coupled with our temporally-detailed priority queues. This ensures bounded exploration of the space of possible candidate paths while ensuring correctness at the same time. EBETS employs the previously mentioned temporal generalization of the path counting technique proposed in Brandes (2001) to count the number of paths passing through any particular node. The input and output of EBETS are formally defined next.

Input The input EBETS consists of (a) temporally-detailed social network dataset over V individuals observed for $\mathcal{T}$ time steps and; (b) a time interval $\lambda \subset \mathcal{T}$ represented as discrete time instants

Output Temporal betweenness centrality (as defined in Eq. 1) of all individuals $v \in V$ for all the time instants $t \in \lambda$.

Note that the proposed EBETS algorithm can be easily adapted to other variations of temporal betweenness centrality (apart from Eq. 1) and temporal shortest paths (apart from Definition 1).

As mentioned previously, the spirit of an epoch based approach is to efficiently determine the time points (i.e, the epoch points) when the shortest path changes (according to Defnition 2). Epoch-points would partition the given time interval λ into a set of disjoint sub-intervals, over which the shortest path tree does not change. The *sub-interval SPtree* denotes this shortest path tree.

Definition 4 (*sub-interval SPtree* (SPT_v)) A pair of a tree rooted at v and a set of time instants ω_v such that the path from v to any of the nodes in SPT_v is the temporal shortest path (according to Definition 1) for all the time instants in ω_v. In other words, the shortest path tree SP_v is guaranteed not to change during ω_v.

The key idea of EBETS is compute successive sub-interval SPtree's for each node $v \in V$ until all the time instants in the input time-interval λ are covered. For each sub-interval SPtree, the temporal generalization of pair-dependency (Eq. 3) is computed for the times ω_v. These are also summed up during the execution to obtain the centrality given by Eq. 1. While computing a sub-interval SPtree, EBETS maintains following two tables:

Path Intersection Table Stores the potential epoch points returned from Forecast-Epoch-Point operation of the temporally-detailed priority queue. This table is referred while determining the next epoch-point for re-computation.
Predecessor Table Stores all the possible predecessor(s) of a node across possible temporal shortest paths from the root of tree. This is used to compute Eq. 3.

Algorithm 1 Epoch-point based BETweenness centrality Solver (EBETS) Algorithm

1: **for all** nodes $s \in V$ **do**
2: $cur\text{-}time \leftarrow$ first time instant in λ
3: $upper\text{-}time \leftarrow$ last time instant in λ
4: **while** $cur\text{-}time \leq upper - time$ **do**
5: Initialize a temporally-detailed priority queue $TDPQ$
6: Insert path functions corresponding to neighbors of s into $TDPQ$
7: **while** $TDPQ$ is not empty **do**
8: $pf_{min} \leftarrow$ Extract-Min operation on $TDPQ$
9: $min_{tail} \leftarrow$ Tail node of path corresponding to pf_{min}
10: $t_{min} \leftarrow$ Forecast-Epoch-Point operation on $TDPQ$
11: Save t_{min} in the path intersection table
12: **if** multiple paths in $TDPQ$ end on min_{tail} **then**
13: **for all** time instants t in interval $cur\text{-}time$ through $t_{min} - 1$ **do**
14: Set the second last node of all the paths (including pf_{min}) which have the same cost pf_{min} for time t as predecessor min_{tail}
15: **end for**
16: **else**
17: Set the second last node in pf_{min} as the predecessor of min_{tail} for all times $cur\text{-}time$ through t_{min}
18: **end if**
19: Delete all the paths ending on min_{tail} from $TDPQ$
20: Determine the path functions resulting from expansion of the chosen path
21: Insert the newly determined path functions into $TDPQ$
22: **end while**
23: $cur\text{-}time \leftarrow \min\{t_{min}$ in path intersection table$\}$
24: Clear path intersection table
25: Re-initialize the predecessor table for times greater than $cur\text{-}time$
26: Compute the dependency of s on each vertex for times between the previous and the $cur\text{-}time$
27: **end while**
28: **end for**

We now provide a detailed description of the EBETS algorithm and illustrative execution trace on the temporally-detailed social network shown in Fig. 5. A pseudocode of EBETS is shown in Algorithm 1. The outermost for-loop (line 1–28 in Algorithm 1) ensures that the shortest path tree from all the nodes is determined. Within this for loop, the algorithm computes the shortest path tree of a single node v for all the time instants in λ by determining successive sub-interval SPtrees rooted at v. This is done in the while-loop on line 4. Before this, we initialize two variables, $cur\text{-}time$ and $upper\text{-}time$ to the first and last time in λ respectively. The while-loop on line 4 executes until the value of $cur\text{-}time$ is less than $upper\text{-}time$. The inner most while-loop on line 7 determines a single *sub-interval SPtree* for node v. Here, we first initialize a temporally-detailed priority queue, $TDPQ$, with path functions corresponding to immediate neighbors of source node. The queue is ordered based on the values of the path-functions for time $cur\text{-}time$. The while loop on line 7 runs until $TDPQ$ is not empty. In each iteration of this while loop, the path function having the least cost for $cur\text{-}time$ (pf_{min}) is extracted its corresponding epoch-point is determined using forecast-epoch-point operation on $TDPQ$. The forecast-epoch-point operation does this by

computing the earliest (in terms of time) of intersection point of pf_{min} with the other path functions in the $TDPQ$. This intersection point t_{min}, a potential epoch-point, is stored in the *path intersection table*. Note that in the absence of intersection points, the value of t_{min} is set to one more than the last time in λ.

Following this, the algorithm determines all the possible predecessors of the tail node (min_{tail}) of the path corresponding to pf_{min}. This is accomplished as follows: We first choose all the paths in the $TDPQ$ whose tail node is same the as min_{tail}. Now, the second to last nodes of all these paths are potential predecessors of min_{tail}. Among these candidates the ones who have the same cost as that of pf_{min} are stored as predecessors of min_{tail} (at their corresponding time instants). For example, consider a path $<s, x, y>$ whose corresponding path function [2 3 4 5 X] was chosen as pf_{min}. Further, consider another path function [2 3 4 X X], corresponding to path $<s, z, y>$, present in the queue. For convenience, assume that the $cur\text{-}time$ was $t = 1$ (corresponding to the first value in the path function), t_{min} would be 5. Here, the node z would be included in predecessors of node y for times $t = 1, 2, 3$. Whereas, node x would be included for times $t = 1, 2, 3, 4$. This process is accomplished in lines 12–18 of the Algorithm 1. After the predecessors are determined, other paths ending on min_{tail} are deleted from $TDPQ$ (line 19) and pf_{min} is expanded to include each of its neighbors.

The newly determined path functions (line 21) are inserted into $TDPQ$. This process continues until $TDPQ$ is empty. Note that when pf_{min} is expanded, min_{tail} is closed for $cur\text{-}time$. This means that there cannot be any other path leading to min_{tail} from the root with a shorter length. After exiting from the inner while loop on line 7, we have a sub-interval SPtree rooted at node s (as per Line 1) whose optimality is guaranteed for all times between $cur\text{-}time$ and the minimum of $t_{min}s$ stored in the path intersection table. Following this EBETS performs following three tasks. First, $cur\text{-}time$ (next time for re-computation) is set to the minimum of $t_{min}s$ stored in the path intersection table (line 23 in Algorithm 1). In a worst case scenario, this value could just be the next time instant in λ. In such a case, the sub-interval SPtree determined by the inner most while loop would be valid for only one time in λ. Second, the predecessor table is reinitialized for times greater than the new $cur\text{-}time$. Third, the dependency (given by Eq. 3) of each of the nodes in the sub-interval SPtree is computed for times between the previous and the new $cur\text{-}time$.

4.2 Execution trace

We now provide an execution trace of the EBETS on the temporally-detailed social network shown in Fig. 5. Here, the set $\lambda = 1, 2, 3, 4, 5$. For brevity, we only show the computation of a single sub-interval temporally-ordered SPtree with source W. For this execution trace, we have the following: (a) Our path-functions are modeling information flows, (b) It takes 1 time unit for the information to flow on an edge, (c) Waiting is not allowed.

In the first iteration of the while-loop on line 4 the value of $cur\text{-}time$ would be 1 (first time instant in λ). The inner while-loop on line 7 begins with a temporally-detailed priority queue initialized with the path functions corresponding to immediate neighbors of W (see Fig. 8). Now, the path function having the lowest cost at $t = 1$ would be chosen as pf_{min} on line 8. In this case, the path $<W, A>$ has the lowest cost for $t = 1$. The path-function of $<W, A>$ intersects with that of $<W, C>$ at time $t = 5$ which be the value of t_{min}. In other words, forecast-epoch-point would return $t = 5$, which is stored in the path intersection table. Now, since there are no other paths ending on node A in queue, W would be set as the predecessor of A for times 1, 2, 3, 4 (all times between $cur\text{-}time$ and $t_{min} - 1$). Lines 20-21

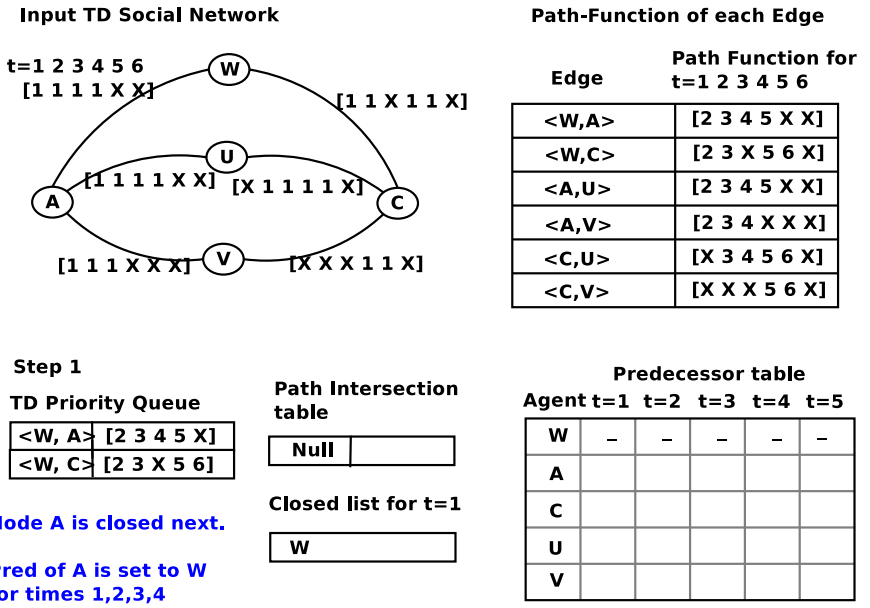


Fig. 8 EBETS Execution trace: start of step 1

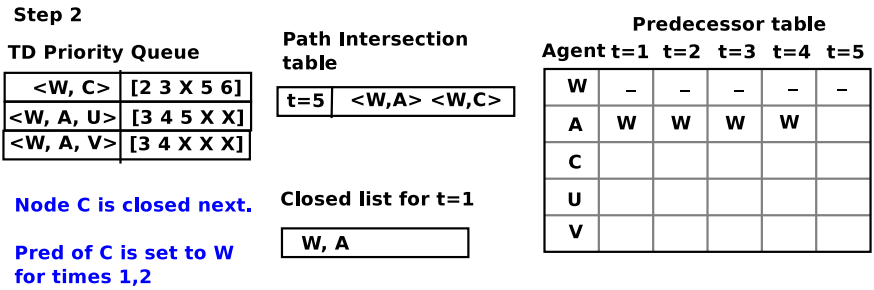


Fig. 9 EBETS execution trace: start of step 2

would expand the path $<W, A>$ to insert the path-function corresponding to $<W, A, U>$ and $<W, A, V>$ (see Fig. 9) into the queue.

For the second iteration of the while loop on line 7, the algorithm would select the path $<W, C>$ as pf_{min} and node C would be closed for time $t = 1$. Forecast-epoch-point would consider the intersection of the path function of $<W, C>$ with $<W, A, U>$ at $t = 3$ (see Fig. 9). This intersection point would be stored in the path intersection table. Now, since there are no paths ending with node C in the queue (see Fig. 9), W would be set as the predecessor of C for times $t = 1, 2$. The algorithm then expands the path $<W, C>$ and the path-functions corresponding to $<W, C, U>$ and $<W, C, V>$ are inserted into the queue.

In the third iteration of the while loop on line 7, the algorithm would select the path $<W, C, U>$ as pf_{min} (ties broken arbitrarily) and node U would be closed for time $t = 1$. Forecast-epoch-point would return $t = 3$ as there is an intersection of the path function of $<W, C, U>$ with $<W, A, U>$ at $t = 3$ (see Fig. 10). This intersection point is stored in the path intersection table. Now, there are two paths, $<W, C, U>$ and $<W, A, U>$, in the queue which end on node U and have the same cost of $t = 1$ and $t = 2$. Thus, both A and U would

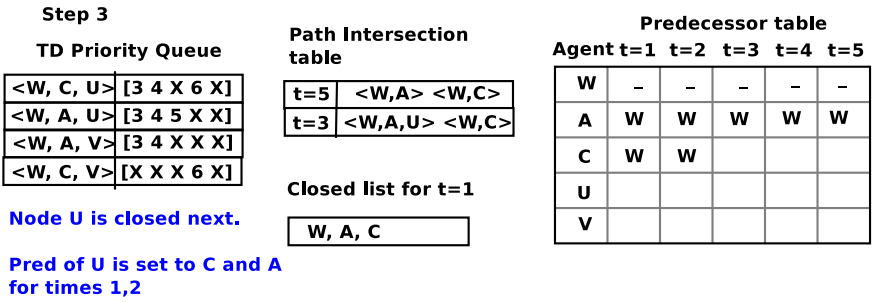


Fig. 10 EBETS execution trace: start of step 3

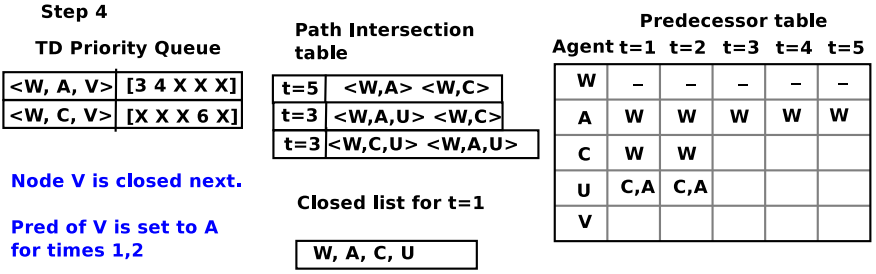


Fig. 11 EBETS execution trace: start of step 4

be set as predecessors of U for times $t = 1$ and $t = 2$. $<W, C, U>$ is not expanded further as it does not have any unclosed neighbors. At line 19, the algorithm deletes both $<W, C, U>$ and $<W, A, U>$ from the queue.

In the fourth iteration of the while loop on line 7, the algorithm would select the path $<W, A, V>$ as pf_{min} and node V would be closed for time $t = 1$. Forecast-epoch-point would return $t = 3$ as $<W, A, V>$ ceased to exist at $t = 3$ (see Fig. 11). This intersection point is stored in the path intersection table. Now, there are no other two paths ending on V for times $t = 1$ and $t = 2$. Thus, only A is set as a predecessor of V for times $t = 1$ and $t = 2$. $<W, A, V>$ is not expanded further as it does not have any unclosed neighbors. At line 19, the algorithm deletes both $<W, C, V>$ and $<W, A, V>$ from the queue.

For the fifth iteration of the while loop on line 7, the priority queue would be empty. This is the termination condition of the while-loop on line 7. After the the termination of this loop, the next $cur\text{-}time$ is determined by selecting the earliest of the $t_{min}s$ stored in the path intersection table. This happens to be $t = 3$ for this example (see Fig. 12). Thus effectively, the algorithm has now determined a sub-interval SPtree with source W valid for times $t = 1$ and $t = 2$. Note that we did not compute the shortest path tree for $t = 2$ separately. The above process of computing a sub-interval SPtree will now repeat for $t = 3$. Before proceeding to the next iteration of the while-loop on line 4, the algorithm re-initializes the predecessor table for times $t = 3$ and greater. After that partial dependencies of all the nodes in this sub-interval SPtree are computed for times $t = 1$ and $t = 2$ using Eq. 3.

4.3 Epoch-point based approaches for other temporal betweenness metrics

As mentioned earlier several works (Habiba et al. 2007; Tang et al. 2010, 2009; Kim and Anderson 2012) have defined temporal extensions of betweenness centrality. Although they slightly vary in their formal definition, they share the common underlying structure given

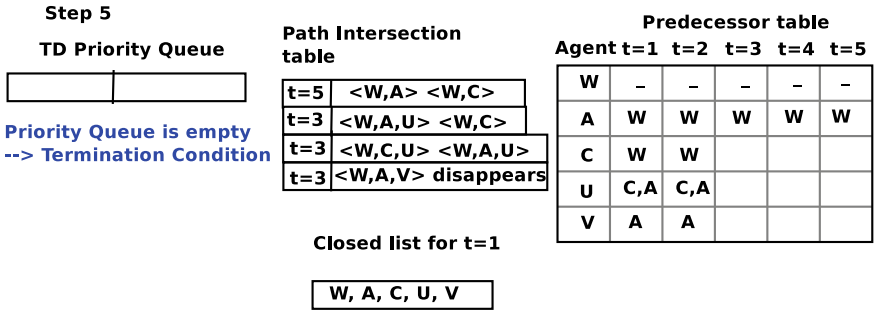


Fig. 12 EBETS execution trace: start of step 5

in Eq. 1. We now briefly describe the interrelationship between our definition of temporally betweeness and others available in the literature. Alongside, we also briefly mention ways in which our epoch-point based approach be used to compute these metrics.

In Kim and Anderson (2012), a temporal extension of betweenness centrality which was based on the notion of "temporal shortest path" [Section III in Kim and Anderson (2012)] was proposed. It should be noted their definition of "temporal shortest path" is same as ours given Definition 1. Based on this definition, the paper proposes the following metric:

$$B_{i,j}(v) = \sum_{i \leq t \leq j} \sum_{\forall s \neq v \neq d \in V} \frac{\sigma_{t,j}(s,d,v)}{\sigma_{t,j}(s,d)} \tag{4}$$

Here the parameters i and j denote the starting and the ending of time-interval over which the temporally detailed social networks was observed. The term $\sigma_{t,j}(s,d)$ defines the number of temporal shortest paths which start from node s at time t and terminate at node d (within the time horizon under consideration). And $\sigma_{t,j}(s,d,v)$ denotes the number of such paths passes through node v. Intuitively, Eq. 4 generates a global metric by summing up the betweenness value of node across all the times of interest. Comparing this with our temporal extension in Eq. 1, we note that the primary difference between them is limited to the fact that Eq. 4 can be computed by simply summing up Eq. 1 over the range of time steps of interest. Thus the fundamental computational unit behind computing Eq. 4 would still be repetitive computation of temporal shortest paths for a range of time instants. To this end, we can use a modified version of the EBETS algorithm for computing Eq. 4. The primary change would be in line number 26 where the dependency is computed. Here, we would simply summing up the dependency values across all times of interest instead of maintaining them separately.

In Habiba et al. (2007), two extensions were proposed; one was based on the notion of what the paper refers to as "shortest simple temporal path" and other was based on the notion of "shortest temporal trails." The definition of "shortest simple temporal path" given in this paper is identical to ours given in Definition 1. On the other hand, "Shortest temporal trails" were defined to be more general in the sense that the nodes are allowed to repeat in a path. In other words these paths are not longer simple. Nevertheless other constraints (second and third bullet of Definition 1) still apply. Given these two notions of temporal paths, the paper defines following two temporal extensions of betweenness centrality.

$$B_T(v) = \sum_{s \neq t \neq v} B_{T(st)}(v) = \sum_{s \neq t \neq v} \frac{g_{st}(v)}{g_{st}} \tag{5}$$

$$B_D(v) = \sum_{s \neq t \neq v} B_{D(st)}(v) = \sum_{s \neq t \neq v} \frac{nst_{st}(v)}{nst_{st}} \tag{6}$$

Here, $B_T(v)$ (Eq. 5) uses the notion of "shortest simple temporal paths". The parameter g_{st} denotes the number of shortest temporal paths between s and t across all the time instants under consideration. And $g_{st}(v)$ denotes the number that pass through the individual v. On the other hand, $B_D(v)$ (Eq. 6) uses the notion of "shortest temporal trails." Here, the parameter nst_{st} denotes the number of shortest temporal trails between s and t across all the time instants under consideration. The parameter $nst_{st}(v)$ is defined in a slightly different manner, it denotes the number of times those trails pass through node v.

Our temporal extension of betweenness given in Eq. 1 is very similar to Eq. 5 proposed in Habiba et al. (2007). The only difference begin our metric computes betweenness values at finer temporal resolutions whereas, Eq. 5 considers all the paths possible over a given time period. Nevertheless, the fundamental computational unit required to compute Eq. 5 still involves repeated computation of temporal shortest paths across a range of time instants *which can benefit from our epoch-point based approach*. Of course the EBETS algorithm given in Algorithm 1 needs to be updated slightly to compute Eq. 5, but the fundamental technique of epoch-points remains the same. The temporal extension given in Eq. 6 is slightly different from our extension because it considers trails instead of simple paths. However, using the Lemma 1 given in Habiba et al. (2007) which shows that shortest trails can be computed as shortest simple paths in a directed acyclic graph representation of the temporally detailed social network. This would again allow us to adapt EBETS algorithm for computing Eq. 6. In this adapted version, we would make following there changes to Algorithm 1. First, we would have add a loop over all nodes over the while loop on line number 7. Second, the termination condition of the while loop on line number 7 needs to be changed to "until the node selected in the newly added loop is not closed". And lastly, the EBETS algorithm should be not maintaining a list of closed nodes as in a trail nodes can be visited more than once. However, the destination node can be visited only once (which would be termination condition of loop on line number 7).

Tang et al. (2009, 2010) also proposed a temporal extension of the betweenness centrality. The paper first puts a new model where a temporally detailed (TD) social network is created by collapsing all the interactions which happened within a window of certain length [parameter w in Tang et al. (2009, 2010)]. Their TD social network is defined as the set $\mathcal{G}_t^w(t_{min}, t_{max}) = \{G_{t_{min}}, G_{t_{min}+w}, t_{min}+2w, \ldots, G_{t_{max}}\}$. Here, each $G_{t_{min}+kw}$ is a window which contains interactions which happened between the times $t_{min}+kw \leq s < t_{min}+(k+1)w$. A temporal path between two nodes i and j on this graph was defined as sequence of nodes across $G's$, where at most h nodes can come from the same $G_{t_{min}+kw}$. Following this, a temporal extension of the betweenness was given which is reproduced in Eq. 7.

$$C_i^B(t) = \frac{1}{(N-1)(N-2)} \sum_{j \in V, j \neq i} \sum_{k \in V, k \neq i, k \neq j} \frac{U(i,t,j,k)}{|S_{jk}^h|} \tag{7}$$

Here, $C_i^B(t)$ denotes the temporal betweenness of node i at time t. The function U returns the number of shortest temporal paths between nodes j and k in which node i either receives a message at time t or, is holding a message from previous until it is forwarded to another node at a time $t' > t$. And $|S_{jk}^h|$ denotes the number of shortest temporal paths between j and k for all times of interest.[1]

It should be noted that even though Eq. 7 defines temporal betweenness for a single time instant, the denominator takes into account shortest paths for multiple time instants. Thus our proposed epoch-based approach can still be used to compute this. If one were to adapt EBETS

[1] Although, we believe it should be sufficient to consider only those paths which start at node j on before time t, others should not effect the centrality of i at time t.

algorithm for computing Eq. 7, following changes need to made in the algorithm. First, the TD priority queue would contain path functions which are partly "snapshot-based" and partly "information-flow based" (refer Sect. 4). Basically, whenever a node is being expanded for a time (*cur-time* variable in Algorithm 1), we would add two kinds of path-functions: (a) paths from the same snapshot $G_{t_{min}+kw}$ at no extra increment in the path-function being expanded and, (b) paths stepping across the current $G_{t_{min}+kw}$ whose path-function whose values would be incremented. This step would basically simulate the reachability within a window that is computed in Tang et al. (2009) [Section 2.2 in Tang et al. (2009)]. Following this change, we would also have to change line 26 in Algorithm 1 to compute Eq. 7. To achieve this we would first move this line out of the loop on line number 4. Now after computing a single sub-interval SPtree in the while loop on line number 7, we would use the predecessor table to compute (and accumulate) $|S^h_{jk}|$ (for a fixed j as set by the loop on line number 1) across all times on interest. Outside this loop, we can compute the inner summation of Eq. 7. Note that even though Eq. 7 focuses only one time instant, we can easily compute this for all the times of interest as we already have the information. And finally, we can compute the outer summation of Eq. 7 through the outer loop on line number 1 in Algorithm 1.

5 Analytical evaluation

5.1 Correctness and completeness

The EBETS algorithm divides a given interval (over which shortest path trees have to be determined) into a set of disjoint sub-intervals. Within these intervals, the tree does not change. The correctness of EBETS algorithm requires that the set of predecessors determined for any particular node (lines 12–18 of Algorithm 1) be comprehensive. On the other hand, the completeness of the algorithm guarantees that none of the epoch-points are missed. Lemma 2 and Corollary 2 are used to show that the set of predecessors determined by EBETS is comprehensive, i.e., the algorithm records all the potential predecessors of a node while building an sub-interval SPtree. On the other hand, Lemma 1 shows that the EBETS algorithm does not miss any epoch-point. These lemmas are used to prove the correctness and completeness of the EBETS algorithm.

Lemma 1 *The EBETS algorithm re-computes the shortest path tree (for any particular node $s \in V$) for all those times $t_i \in \lambda$, where the tree for previous time t_{i-1} can be different from that of t_i.*

Proof The shortest path tree of a node $s \in V$ is considered to have changed if the shortest path to any node t (from s) in the tree is different across times t_i and t_{i-1} or there is node w outside of tree which is now reachable. For the sake of brevity, we only give the proof for case when the shortest path to a single node t in the tree has changed. This can be easily generalized for case of w and the whole tree.

Consider the network shown in Fig. 13a where a shortest path tree of the node s is to be determined for each time instant in $\lambda = [1, 2, \ldots T]$. In this proof we focus on the shortest path to node d instead of the whole tree. First, the source node is expanded for the time instant $t = 1$. As a result, path functions for all the neighbors $<s, u>$, $<$s,$x_1>$, $<$s,$x_2>$, $\ldots$ $<$s,$x_i>$ are added to the priority queue. Without loss of generality, assume that the path $<$s,u$>$ is chosen in the next iteration of the innermost loop. Also assume that the earliest intersection point between $<s, u>$ and the path functions in the priority queue is at $t = \alpha$ (between $<s, u>$ and $<$s,$x_1>$). After expanding $<$s,u$>$ the queue would contain paths $<$s,u,d$>$, $<$s,$x_1>$, $\ldots$,

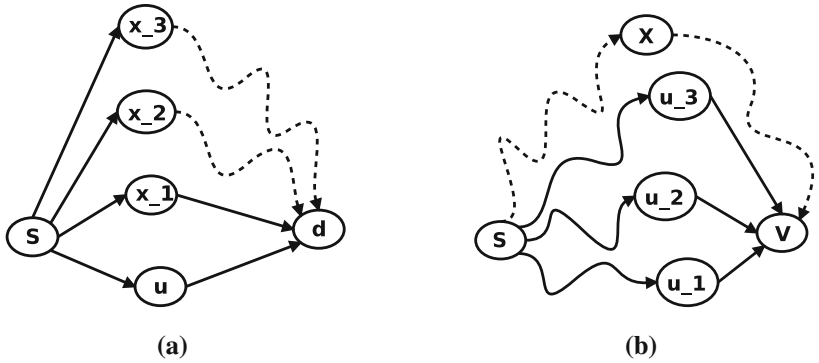


Fig. 13 Sample network for Lemma 1 and Lemma 2

<s,x_i>. Here we can have two cases. First, path <s,u,d> has lower cost for time $t = 1$. Second, path <s,x_1> has lower cost for time $t = 1$.

Consider the first case, again without loss of generality assume that the earliest intersection point between the path functions <s,u,d> and <s,x_1> is at $t = \beta$. Note that both $t = \alpha$ and $t = \beta$ denote the times at the source node. Now, $\beta \leq \alpha$ (note that β cannot be greater than α as all the edges have positive weights). In such a case the shortest path is recomputed for time $t = \beta$ and the path <s,u,d> is closed for all times $1 \leq t < \beta$. Assume for the sake of argument, that there is a shortest path $P_x =< s, x_2, \ldots, d >$ from s to d that is different from path <s,u,d> and is optimal for a time $t_x \in [1, \beta)$. This means that path $< s, x_2 >$ should have had least cost for time $t = t_x$. In other words, the earliest intersection point between functions in the priority queue and $< s, u, d >$ should have been at $t = t_x$ rather than at $t = \beta$ (a contradiction). Moreover, as all the edge costs are positive, if sub path <s,x_2> was not shorter than <s,u,d> for times $t \in [1, \beta)$. Any positive weight addition to the path-function (through other edges) cannot make P_x shorter than <s,u,d> for $t \in [1, \beta)$

Now we consider the second case when <s,x_1> had lower cost for time $t = 1$. Now, path <s,x_1> would be expanded and path function <s,x_1,d> would be added to priority queue. A similar argument as above can be given for this case as well.

Corollary 1 *The EBETS algorithm correctly computes the shortest path tree (for any particular node $s \in V$) for any particular time instant $t \in \lambda$.*

Lemma 2 *The set of predecessors determined for a node v for a time $t \in \lambda$ is complete.*

Proof Let $P(v) = P_{u_1}(v), P_{u_2}(v), P_{u_2}(v), \ldots, P_{u_k}(v)$ denote the set of paths currently in the priority queue whose tail node is v. Here, P_{u_i}, where $i \in [1, k]$, denotes the path whose second-to-last node is u_i (and last node being v) and $cost(P_{u_i}(v))$ denotes the cost of path $P_{u_i}(v)$ for time $t = \alpha$. Without loss of generality assume that the node v is being closed for time $t = \alpha$ (see Fig. 13b) through the path P_{u_1}, i.e., P_{u_1} is found to be a shortest path for node v. Now, the EBETS algorithm assigns a node u_x as a predecessor of node v based on the following cases;

Case 1: u_x is the second-to-last node in the shortest path found, i.e., $u_x = u_1$.

Case 2: $u_x \in U$, where $U = \{u_i | cost(P_{u_i}(v)) = cost(P_{u_1}(v)) \& i \neq 1\}$.

In order to prove the lemma, we show that the above two cases are correct and complete. We first show the correctness followed by completeness. Using Corollary 1 we know that

when a node v is closed by the EBETS algorithm for any particular time t, it has correctly determined the shortest path for v. Therefore, the node preceding v in this path is clearly a predecessor of time t. This proves the correctness of Case 1. Correctness of Case 2 can be also guaranteed in similar fashion as we include the second-to-last nodes of only those paths whose cost is same as that of the shortest path found by EBETS.

In order to show the completeness, we need to prove that there cannot be any other path Q, not currently in the priority queue, with $cost(Q) = cost(P_{u_1})$. For sake of argument assume that there exists a path $Q = <s, \ldots, x, v>$, not currently in $TDPQ$, and x is valid predecessor of v for time $t = \alpha$. By definition the distance of x from s should be less than that of v (as we have only positive weights). This means that node x should be closed before v. Now, whenever a node is closed by the algorithm, path functions corresponding to its neighbors are added to the priority queue. This contradicts our original assumption that path Q was not present in the priority queue.

Corollary 2 *The set of predecessors determined for a node v for all times $t \in \lambda$ is complete.*

Proof Using Lemma 1, we know that EBETS does not miss any epoch point. Now, during the time interval between two consecutive epoch-points, Lemma 2 can be used to conclude that the predecessors are correctly determined for all time time instants within the interval. Using these two arguments we can conclude that EBETS correctly determines the predecessors of a node v for all times $t \in \lambda$.

Theorem 1 *EBETS algorithm is correct and complete.*

Proof For any particular node v, there can be several sub-interval SPtrees, $SPT_v = \{SPT_v^1, SPT_v^2, \ldots, SPT_v^k\}$, over the λ, where each SPT_v^i is associated with a set of time instants ω_v^i and $\bigcup_{\forall i \in |SPT_v|} \omega_v^i = \lambda$. The correctness of the algorithm is shown by Lemma 2 and Corollary 2. The completeness proof is presented in three parts. First, using Lemma 1 we can conclude that the EBETS algorithm does not miss any epoch point. Secondly, the loop on line 4 of the algorithm ensures that all time instants in λ are considered. Finally, the outer most loop on line 1 ensures that all the nodes $v \in V$ are processed. This proves the completeness of the algorithm.

5.2 Space and time complexity analysis

This subsection details the space and time complexity of the EBETS algorithm and also presents a comparison with other computational techniques in the literature. First, we start with the time-complexity of the temporally-detailed (TD) priority queue operations.

TD Priority Queue The time-complexity of operations on TD priority queue would depend on its implementation. Here we describe its complexity for a heap based implementation, i.e., the time-series' in the TD priority queue would be put in a binary heap based on their values for the time index t_{ord}. We assume that there are $size\text{-}TDPQ$ time-series' in the TD priority queue, where each time-series is of length $TS - Len$ (same for all the time-series' in the priority queue).

Cost of Insert Operation This is would be logarithmic in size of the heap, i.e., $O(\log size\text{-}TDPQ)$. This cost follows from a straight forward implementation of binary heaps where a new item is added to the end of the heap followed by a call to the standard heapify operation (Cormen et al. 2001).

Cost of Delete Operation This would also be logarithmic in the size of the heap, i.e., $O(\log size\text{-}TDPQ)$. Similar to insert, the cost of this operation also follows from a straight

forward implementation of a binary heap. Here, the delete operation would exchange the key to be deleted with the bottom-right most key (last item for an array based implementation) in the heap and call the standard heapify operation (Cormen et al. 2001). Variables storing the size of the heap are then updated accordingly. Note that an update operation can also be implemented in similar fashion.

Cost of Extract-Min Operation This would also be logarithmic in the size of the heap, i.e., $O(\log size\text{-}TDPQ)$. Similar to a traditional extract-min implementation on a binary heap, we would exchange the root of the heap with the bottom-right most key (last item for an array based implementation) in the heap and call the standard heapify operation (Cormen et al. 2001).

Forecast-Epoch-Point Operation This operation (typically called before an extract-min operation) compares the time-series on the top of the heap with the rest of the time-series' in the queue ($size\text{-}TDPQ$) to find out the largest time index $t_{epoch} > t_{ord}$ for which the current top of the heap has the lowest cost. If the length of each time-series in the queue is $TS - Len$ then this operation would take $O(TS - Len \times size\text{-}TDPQ)$.

Space Complexity of EBETS The total space requirement of an instance of the EBETS algorithm is the sum of individual requirements of the following: (a) the input temporally detailed (TD) social network, (b) the TD priority queue, (c) the path-intersection table, (d) the predecessor table, and (e) the tables for storing dependency and centrality values. Note that items (b) through (d) are re-initialized for each iteration of the loop on line number 4 in the algorithm (ref Algorithm 1), so at any stage during the execution, we would have only one instance of items (b)–(d). Tables for storing the dependency and centrality values are global in the sense that they are not re-initialized inside the loops. We now describe the space complexity of each of items (a)–(e).

TD social network In this paper, we implemented the input TD social network as a time-aggregated graph (George and Shekhar 2007) described in Sect. 2.1. For purposes of storage efficiency, the interaction time-series is stored as an array of time-instances where an interaction occurred rather than a complete time-series (containing "1s" and "Xs") over the entire time horizon under consideration. Under this representation the worst case space complexity of the TD social network would be $O(m\mathcal{T})$. Here, m is the number of edges in the TD social network and $\mathcal{T}$ is the length of the time horizon over which the interactions were observed. Each edge would have the information on the individuals involved in interaction (a constant number).

TD priority queue At any stage during the execution of the loop on line number 4 in Algorithm 1), the TD priority queue can at most have $degree(v)$ path-functions ending on node v. This is because, after every extract-min operation on line number 8, all paths ending on the node min_{tail} (tail node of the path retrieved during extract-min) are deleted from the queue at line number 19. Thus, the worst case space complexity would be $\sum_{\forall v} Degree(v)$, which is $O(m)$.

Path intersection table This table is cleared and re-initialized after each instance of loop on line number 7 in Algorithm 1. During each iteration of this loop at most one entry can be added to the path intersection table (line number 11). Now given that this loop can run for at most $O(m)$ times (sine the maximum size of the TD priority queue can be $O(m)$), the worst case size of the path intersection table would be $O(m)$.

Predecessor table The predecessor table stores the predecessors to a node v (in worst case $Degree(v)$) along different shortest paths for the times greater than the $cur\text{-}time$ (of loop of line number 4). Also note that the predecessors for times greater than the next $cur\text{-}time$ (the epoch-point discovered in this iteration) are deleted at line number 25 of the EBETS algorithm (Algorithm 1). Thus, during any iteration of the loop on line number 4, this table

can at maximum store the predecessors for all the time instants in λ (given in the input). Thus, worst case space complexity of this table can be $n \times \sum_{\forall v}(degree(v) \times |\lambda|)$, which is $O(nm|\lambda|)$.

Table for storing dependencies ($\delta_s.()[]$): In each iteration of the outermost loop (on line number 1) of the EBETS algorithm, we compute the dependency of a node s on all nodes $v \in V$ and all times in λ. The size of this data structure does not change as the execution proceeds, its size remains at $O(n^2|\lambda|)$, where n is the number of nodes in the TD social network.

Table for Centrality values Similar to the table storing the dependencies, the size of this data structure also does not change during the execution. Its size is $O(n|\lambda|)$, which is basically centrality value for each node $v \in V$ for all time instants given in λ.

Proposition 2 *Given a TD social network containing n nodes and m edges observed over time period of $\mathcal{T}$ time steps, and an instance of EBETS algorithm running over this network for λ time steps; we have the following:*

1. *The total space required in worst case to store the TD social network would be $O(m\mathcal{T})$.*
2. *The total space required in worst case to execute the EBETS algorithm would be $O(m + m + mn|\lambda| + n^2|\lambda| + n|\lambda|)$, which is $O((mn + n^2)|\lambda|)$.*

Time complexity of EBETS algorithm: We assume that the given TD social network containing n nodes and m edges is observed over time period of $\mathcal{T}$ time steps, and an instance of EBETS algorithm running over this network for λ set of time instants. The outermost loop of Algorithm 1 loops over all nodes $v \in V$. For each node $v \in V$, the algorithm computes successive sub-interval SPtrees rooted at v (refer Definition 4) and its corresponding set of time instants ω_v. These sub-interval SPtrees are computed for node v until all the time instants in the set λ are covered. In addition, after computing a sub-interval SPtree we update the dependency and centrality values.

A key aspect which greatly influences the total time taken by the EBETS algorithm is the number of sub-interval SPtrees computed for each node v (and the total over all nodes $v \in V$), which in turn depends on the input dataset. For this reason, we would derive the total time complexity in terms of #epoch-points observed during the execution. Recall that according to our definition of epoch-points (refer Definition 2), they mark the start of new sub-interval SPtree.

Each iteration of the while loop on line number 7 in Algorithm 1 computes a single sub-interval SPtree. Thus, the total number of iterations of the while loop on line number 4 would be $O(num\text{-}epoch\text{-}points(v))$, where $num\text{-}epoch\text{-}points(v)$ denotes the number of epoch-points observed while computing sub-interval SPtrees rooted at node v for times in the set λ. Note that in worst case $num\text{-}epoch\text{-}points(v)$ would be no worse than $|\lambda|$. Now focusing on the operations inside while loop on line number 7. This loop performs the following operations.

1. Extract-min operation (line number 8): We would at most have $|V|$ extract-mins, totaling (across all iterations of loop on line number 7) to a cost of $O(n \log size\text{-}TDPQ)$, which is $O(n \log m)$ (for reference, refer to previous discussion on time complexity of TD priority queue operations).
2. Forecast-epoch-point operation (line number 10): This operation would at most be called $|V|$ times. The total cost of this operation across all iterations of loop on line number 7 would be $O(nm|\lambda|)$. Note that the path-functions in the TD priority queue would be of length $|\lambda|$.

3. Computing predecessors (lines 12–18): The node min_{tail} would at most have $Degree(min_{tail})$ predecessors. This would at most take $O(size\text{-}TDPQ)$ time to find out from the TD priority queue. Furthermore, loop on line number 13 (or 17) can at most have $|\lambda|$ iterations. Thus total cost would be $O(size\text{-}TDPQ + \Sigma_{\forall v \in V} Degree(min_{tail}) \times |\lambda|)$, which is $O(m|\lambda|)$.
4. Deleting path-functions ending on min_{tail} (line number 19): We can at most have $Degree(min_{tail})$ deletes each time. Here min_{tail} is the tail node of the path corresponding to extract-min. This would total to $O(\Sigma_{\forall v \in V} Degree(min_{tail}) \times \log(size\text{-}TDPQ))$, which is $O(m \log m)$.
5. Inserting new path-functions (line number 21): We can at most have $Degree(min_{tail})$ inserts each time. This would total to $O(\Sigma_{\forall v \in V} Degree(min_{tail}) \times \log(size\text{-}TDPQ))$, which is $O(m \log m)$.

Therefore, the total cost of while loop on line number 7 would be $O(n \log m + nm|\lambda| + m|\lambda| + 2m \log m)$, which is $O(nm|\lambda| + (n + m) \log m)$. After this while loop we have two more operations inside the loop of line number 4. On line number 25, we reinitialize the predecessor table from times greater than $cur\text{-}time$. This would at most take $O(size\text{-}of\text{-}predecessor\text{-}table)$ time, which is $O(nm|\lambda|)$. On line number 26, we compute the dependency of the vertex v selected in line number 1 on all the nodes of the sub-interval SPtree computed in while loop on line number 7. This is done using Eq. 3. This equation is computed in recursive fashion (using stacks), starting with the last node in the sub-interval SPtree whose $\delta_s.()$ is initialized to zero. This is done for all times greater the previous $cur\text{-}time$ and new $cur\text{-}time$ determined on line number 23. Thus, the total time required for the task on line 26 would at most be $O(n|\lambda|)$. Combining these costs with the total cost of while loop on line number 7, we would get $O(nm|\lambda| + (n + m) \log m + nm|\lambda| + n|\lambda|)$ $(= O(nm|\lambda| + (n + m) \log m))$ as the total cost of a single iteration of the while loop on line number 4.

As mentioned earlier, the total number of iterations of the while loop on line number 4 would be $num\text{-}epoch\text{-}points(v)$ (which is also the number of sub-interval SPtrees computed for node v). Thus the total time complexity of this while loop would be $num\text{-}epoch\text{-}points(v) \times O(nm|\lambda| + (n + m) \log m)$. If $total\text{-}epochs = \Sigma_{\forall v} num\text{-}epoch\text{-}points(v)$, the total time complexity of the EBETS algorithm would be $O(total\text{-}epochs \times (nm|\lambda| + (n + m) \log m))$. Note that in worst case the $total\text{-}epochs$ would be $O(n|\lambda|)$

Proposition 3 *Given (a) a TD social network containing n nodes and m edges observed over time period of $\mathcal{T}$ time steps; and (b) an instance of EBETS algorithm running over this network for λ time instants. We have the following.*

1. *Total time complexity of the EBETS algorithm is $O(total\text{-}epochs \times (nm|\lambda| + (n + m) \log m))$.*
2. *In worst case, $total\text{-}epochs$ would be $O(n|\lambda|)$.*

Discussion The EBETS algorithm would perform less computation than a naive approach which determines the shortest path tree of a node for each time in the user specified time interval λ. However, this happens only when the ratio of the number of epoch-points to that of time instants in λ is not high. This ratio can be denoted as the change probability shown by Eq. 8.

$$change\_probability = \frac{\#total\text{-}epochs}{n \times |\lambda|} \tag{8}$$

When the change probability is nearly 1, there would be a different shortest path tree (of a node) for each time in the interval λ. In this worst case scenario, the EBETS approach would also have to recompute the shortest path tree for each time in the given set λ.

5.3 Comparing complexity of epoch-point based approaches and related work

As Proposition 3 shows, the time-complexity of the EBETS algorithm heavily depends on the parameter *total-epochs* which is the total number of epoch-points observed across all the nodes (sources) in the loop on line 1 in Algorithm 1. For each of these epoch-points, the algorithm performs $O(nm|\lambda| + (n+m)\log m)$ amount of work. The $((n+m)\log m)$ portion is basically coming due to the "Dijkstra's" flavor of the EBETS algorithm. The $(nm|\lambda|)$ portion is the result of "forecast-epoch-point()" operations.

The time complexity of an adapted Dijkstra's for our problem (our baseline in experiments) would be $n|\lambda| \times O((m+n)\log n)$, assuming a heap based implementation of the priority queue. Clearly, EBETS performs more work per unit re-computation. However, whenever *total-epochs* is less than $n|\lambda|$ EBETS starts to perform better as cost entailed due to extra work balances off against fewer re-computations.

The algorithm given in Ding et al. (2008) (here after referred to as LTT in this paper) can compute shortest path trees rooted at a given node v for a range of time instants. This algorithm can be adapted to compute our temporal extension of the betweenness centrality given in Eq. 1 by maintaining a predecessor table similar to EBETS algorithm. The original time-complexity of the LTT algorithm was given as $O(m+n\log n)\alpha(T)$ in Ding et al. (2008), where $\alpha(T)$ is the cost of manipulating a piece-wise linear function over a time-interval of length T. In our case $T = |\lambda|$ and $\alpha()$ would correspond to cost of things like expanding path-functions to include new neighbors, etc. If LTT were to be adapted for computing Eq. 1 over a time-interval of length $|\lambda|$, we would be running an LTT for each node $v \in V$ and then computing the dependencies. As explained earlier, the cost of computing the dependencies would be $O(n|\lambda|)$. Therefore the total worst case time-complexity of the modified version of LTT would be $O(n \times (m + n\log n)\alpha(|\lambda|) + n|\lambda|)$. If the definition of their $\alpha()$ function is expanded to include operations required while computing the dependencies, then the total complexity would be $O(n \times (m+n+n\log n)\alpha(|\lambda|))$, which is $O(n \times (m+n\log n)\alpha(|\lambda|))$. Space complexity wise, the modified LTT algorithm would need the following (a) $O(m\mathcal{T})$ for storing the TD social network, (b) $O(mn|\lambda|)$, $O(n^2|\lambda|)$ and $O(n|\lambda|)$ for storing predecessor, dependencies and the centrality values, (c) $O(n)$ for storing the priority queue during the execution of the algorithm.

A comparison of EBETS and modified-LTT reveals the following. Space wise, modified-LTT takes the same amount of space for storing the TD social network, predecessors, dependencies and centrality values. However, its priority queue size, $O(n)$, is smaller that $O(m)$ required for EBETS. Time-wise, EBETS takes an extra $O(nm|\lambda|)$ for the forecast-epoch-point operation which is not the case with modified-LTT algorithm. Furthermore, with a smaller priority queue ($O(n)$ in size), the operation of extract-Min could be faster in modified-LTT than in EBETS algorithm which can have $O(m)$ priority queue. However, as explained earlier, LTT is very conservative in determining the epoch-points and thus can end up doing a lot of redundant re-computations which may not be the case with EBETS. Thus, even though the conservative nature of modified-LTT gives it an advantage in terms of lower time-complexity than EBETS, it may end up doing more re-computations than EBETS which is very aggressive in terms of reducing redundant re-computations. This trade-off of low-complexity vs reducing re-computations shows up in the experiments (Sect. 6). Note that the extra "$n\times$" in the complexity of the modified-LTT is not a differentiating factor as EBETS

can also get that from its worst-case value of *total-epochs* which is $O(n|\lambda|)$ (Proposition 3).

6 Experimental evaluation

The overall goal of experimental evaluation was to characterize the performance of epoch-point based betweenness centrality algorithms. To this end, the performance of the EBETS algorithm was compared against the following: (a) LTT algorithm (Ding et al. 2008) and (a) a baseline algorithm (George et al. 2007) which involved computing the shortest path tree for each time in the given time interval of interest (input parameter λ). For comparison, all the algorithms computed the temporal extension of betweenness centrality given in Eq. 1 and followed the temporal shortest path definition given in Definition 1. These algorithms were implemented in C++ language. For our experiments we chose sample email, WikiVote and Foursquare datasets. It is important to note that each of these datasets has a different kind of network and temporal structure (Wei and Carley 2014) as detailed next.

1. **University Email** dataset contains email communications spread over 83 days in early 2004 from a large European university. This dataset recorded about 161227 email communications among approximately 2000 individuals. Events in this dataset (email communications) were recorded in the following format: $<A, B, t>$, where A and B are two individuals in the University and t denotes the time-stamp (in minutes) of the interaction.
2. **WikiVote dataset** contains logs of voting process where by the users and the already administrators of Wikipedia choose a new administrator. This dataset contained voting interactions over 1267 days. Events in this dataset were also recorded in $<A, B, t>$ format. Here, the user A has voted for B at time t. This dataset contains 26773 interactions among 5585 individuals.
3. **Foursquare dataset** contains travel sequences of people. Events in this dataset were also recorded in $<A, B, t>$ format. This means that a certain user has "moved" from location A to location B at time t. This dataset contained about 200000 social events (movements) among 8600 unique locations across New York City and Pittsburgh. A location with high betweenness in this dataset can be interpreted as a location through which many people pass.

Experiments were carried out on a Linux machine with Intel E5-2670 2.50 GHz processor with 64GB RAM. The first step of the experimental evaluation involved creating a time aggregated graph out of the social event data. Since the datasets contained emails communications, voting interactions, spatial movements, these translated into directed edges in a time aggregated graph. Then, a set of different queries (each with a different set of parameters) was run on the EBETS and other candidate algorithms mentioned earlier in the section. The following parameters were varied in the experiments: (a) $|\lambda|$: Length of the time interval over which the centrality metric was computed, and (b) Maximum wait allowed. In each of the experiments the total execution time of all the algorithms were recorded.

Effect of length of time interval ($|\lambda|$) This experiment evaluated the effect of length of time interval (λ) on the candidate algorithms. Figure 14 shows the results of this experiment for the University email dataset. Results on the WikiVote and Foursquare datasets are shown in Figs. 15 and 16. The results show that the execution time of all the algorithms increase almost linearly with increase in $|\lambda|$. However, for any particular value of $|\lambda|$ and maximum

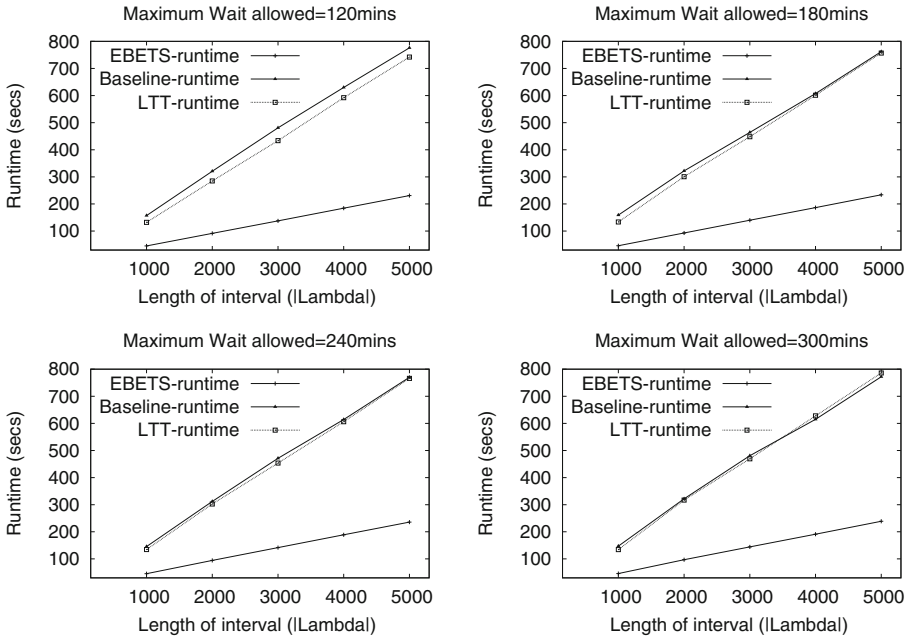


Fig. 14 Effect of length of time interval ($|\lambda|$) on the University email dataset

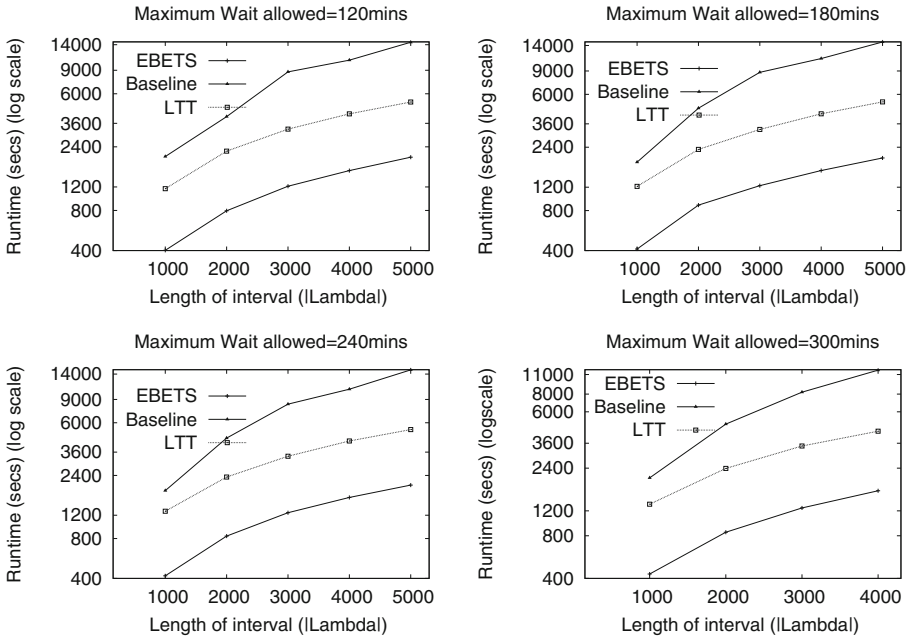


Fig. 15 Effect of length of time interval ($|\lambda|$) on the WikiVote dataset

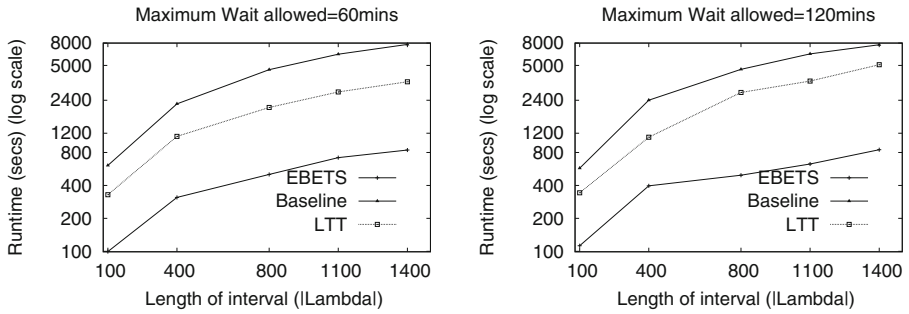


Fig. 16 Effect of length of time interval ($|\lambda|$) on the Foursquare dataset

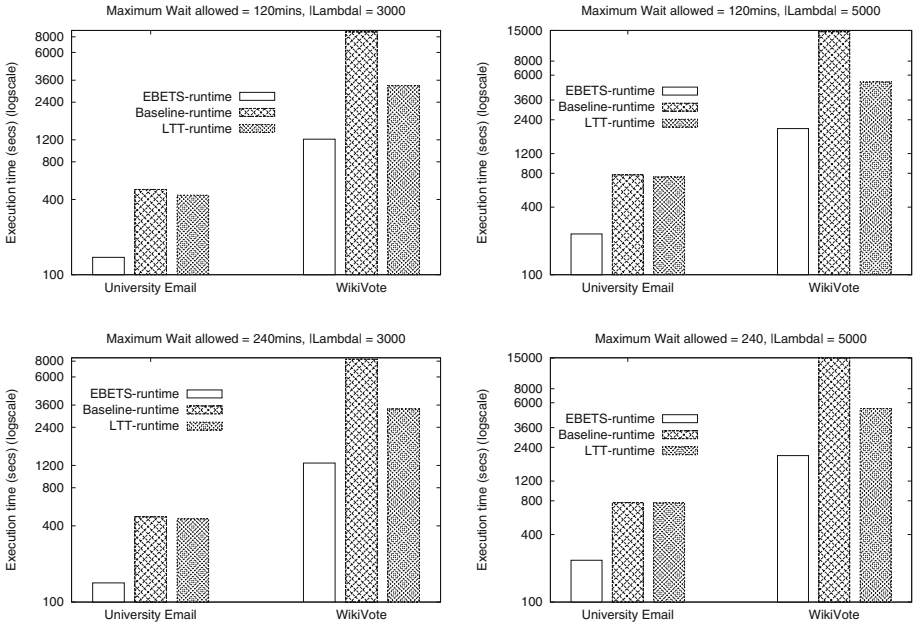


Fig. 17 Effect of network size

allowed wait, EBETS was faster than the alternatives by an order of magnitude. In general, we also observed that runtime of all the algorithms increases with size of the dataset.

Effect of Network size Fig. 17 shows the effect of the Network size on the performance of the algorithms. As can be seen the execution of time all the algorithms increased with the increase in network size, but EBETS still outperforms the alternatives. Recall that WikiVote dataset had 5585 individuals and 26773 interactions, whereas University email dataset had 2000 individuals and 161227 interactions. It is important to note that WikiVote had much lesser number of interactions than University email dataset, however, the number of nodes turned out to have more influence on runtime of the algorithms. In general, we observe that as the execution time increases as the dataset size increases. However, runtime of EBETS increases much slowly than the related work.

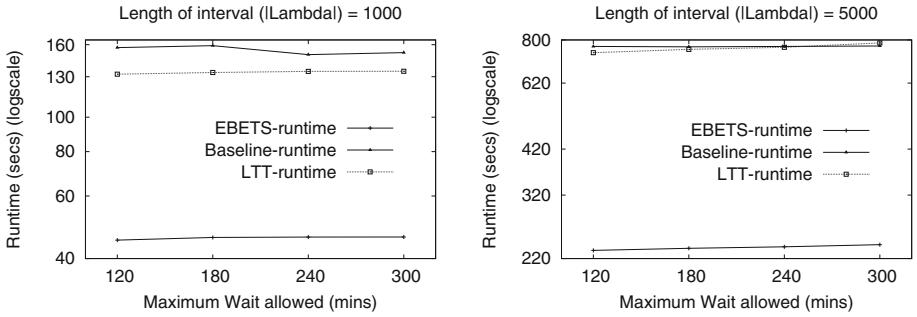


Fig. 18 Effect maximum wait allowed for $|\lambda| = \{1000, 5000\}$ on University email dataset

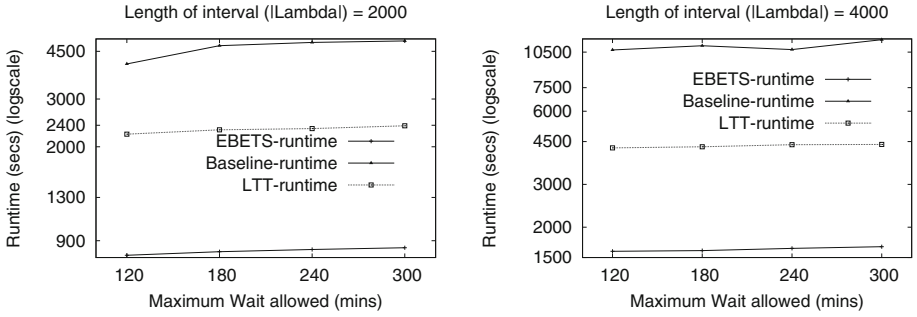


Fig. 19 Effect maximum wait allowed for $|\lambda| = \{2000, 4000\}$ on WikiVote dataset

Effect of Maximum Wait allowed We also tested the effect of increasing the *maximum wait allowed* parameter on the candidate algorithms. As shown in Figs. 18 and 19, run-time of the algorithms was relatively stable across values of maximum allowed wait.

7 Temporal betweenness centrality versus traditional betweennness centrality

In this section, we present an empirical comparison of temporal betweenness centrality (Eq. 1) based on temporal shortest paths (Definition 1) against the traditional betweenness centrality computed on aggregated snapshots. The data we use in this study is a set of email communications spread over eighty three days in early 2004 from a large European university. The dataset has been used in several previous studies (Eckmann et al. 2004; Barabási 2005; Malmgren et al. 2009, 2008), with a focus on both the prevalence of dynamic triads (Eckmann et al. 2004) and on characterizing the inter event dynamics of users (i.e. the time between two consecutive email sends) (Barabási 2005; Malmgren et al. 2009, 2008). Though we will utilize findings from these works extensively, it is important to note that this dataset does contain several nodes which appear to be mail servers as well as several people who do not communicate often enough (as also noted in Malmgren et al. 2009; Barabási 2005; Eckmann et al. 2004). We choose to remove such nodes from the study.

7.1 How does temporal betweenness centrality compare with the traditional betweenness centrality calculated on aggregated panels?

Eckmann et al. (2004) notes, when studying the data utilized in this case study, that "temporal dynamics create coherent space-time structures that...will in general be very different from the fixed ones that appear in the static network". However, Eckmann et al. (2004) is in this quote referring to the network created by aggregating all the events into a single panel, i.e., the network comprised of all eighty three days of interaction data. In contrast, our interest here is in comparing the results of temporal betweenness centrality (based on temporal shortest paths) to panels obtained by aggregating over shorter time intervals.

One disadvantage to our approach is that, while it is straightforward to determine which agent is the most central at any given time instant (a minute, in this case study), it is not immediately obvious how one should determine which agent is the most central over an entire time interval. This is because at each time instant, all agents may be on some number of temporal shortest paths, and thus have some non-zero betweenness centrality score. Therefore, if we wish to determine which agent was the most central on a given day, we must somehow aggregate over all temporal betweenness values, for all agents, for each minute in that day. Though our approach was not geared towards such an cumulative notion of centrality, a quantitative comparison between our method and the aggregated panel is necessary and thus some calculation of the ranks of different agents for betweenness over a given aggregation must be computed.

The manner in which we choose to do this is used for its simplicity and explainability, though others no doubt exist which may be more powerful along one or both of these dimensions. In order to determine which agent is most central over a given period of interval, we first determine the most central agent (i.e. the one with the highest temporal betweenness score) at each time instant (minute), and then aggregate over the desired time period (e.g. a day) to produce a count, for each agent, of the number of time instants he/she was most central for. The rank of a node (e.g. most central, second most central, etc.) for the given time interval is then determined by his or her rank in this list of counts.

Before we can compare to some betweenness metric on aggregated panels of the network, we must also determine what representation of the network each panel will have (Johnson et al. 2012) considers different meanings of an edge in an email network, including removing mass emails to get a more social view of the network. Here, because we consider betweenness as an information flow metric, we choose to keep mass emails in the data. On a different note, Barabási (2005) sees that most users receive more emails than they send, suggesting further that, of course, agents thus receive far more emails than they respond to. This finding suggests email communication may be particularly susceptible to differences in directed versus undirected measures of betweenness, and hence we use a directed approach in order to compare best with our algorithm.

Having operationalized the notion of an agent being "most central" over any given time interval across both methods, we now may compare the results of our metric with the traditional betweenness centrality on aggregated panel. We compare across five different levels of aggregation: one hour, two hours, ten hours, one day and three days. In full, the process for comparing the metric at a single level of aggregation (e.g. a day) can be described in three steps. We first aggregate the interaction data into panels and then compute the traditional *weighted, directed* betweenness centrality (Freeman 1979; Anthonisse 1971) for agents in each using the igraph package in R (Csardi and Nepusz 2006). Note of this step, the traditional betweenness centrality would ignore the temporal order of interactions on the edges. We then use the method described in Sect. 4 to compute the centrality of each

agent within the time-aggregated graph representation of each of the panels for the temporal betweenness centrality metric (we consider the temporal order here). Finally, we utilize one of three comparison metrics, first introduced by Borgatti et al. (2006) and later used by Frantz et al. (2009), to compare agreement between the two approaches for each set of aggregated interactions.

The three comparison metrics we use are "Top 1", "Top 3" and "Top 10 %" [(the original names provided by Borgatti et al. (2006)]. "Top 1" refers to the percentage of panels over which the most central node is the same across both techniques. "Top 3" refers to the percentage of panels that the top node using the traditional approach on aggregated panel is one of the top three nodes utilizing our metric. "Top 10 %" refers, similarly, to the percentage of panels in which the top node in the traditional approach on aggregated panel approach is one of those in the top ten percent using our metric. We choose these metrics over other similar metrics (Kim and Jeong 2007) because they are straightforward to understand while still providing depth for intuition. In collecting results, we only consider as input those panels where at least one node was central using both approaches and where there were no ties as to the most central node in the aggregated panel network. It is also important to note that at lower maximum allowed wait values and aggregations, because often less than 30 nodes have some non-zero centrality value, our results suggest the counter-intuitive finding that in some cases, "Top 3" shows a higher percentage of agreement across the two algorithms than "Top 10 %". Also, because larger aggregation levels naturally have fewer data points, the reader will notice that confidence intervals surrounding the point estimates for these metrics naturally increase with aggregation.

Figure 20 shows the results for each of the three comparison metrics, where points represent the point estimate and error bars give the 95 % confidence intervals using Wilson's formula (Agresti and Coull 1998). The values on the X-axis represent different maximum allowed wait values our algorithm was run with, while the values in the grey bars above each graph represent the level of aggregation, in minutes, over which the interactions were split. In considering the "Top 1" and "Top 3" metrics (the top two plots), we see that although there is reasonably high variance across individual combinations of maximum allowed wait and aggregation level, there are several general patterns. First, agreement between the two methods is slightly higher at lower levels of aggregation. This finding is not particularly surprising; at lower levels of aggregation, there are fewer emails and less time to be aggregated over; resulting in fewer invalid temporal paths and fewer unrealistic durations for which information is held at a given node. However, at these levels, we do see that higher maximum allowed wait values have much lower levels of agreement. This, also, has a fairly obvious explanation in that these values consider data far outside the boundaries of the aggregation, and thus will likely present different pictures of the network. This explanation is further supported by the fact that this difference appears to slow at higher levels of aggregation.

Figure 20 thus suggests following conclusions. First, the agreement between the two metrics on the most central nodes (Top 1 comparison metric) is not more than 60 %. Second, there is very high agreement between the two metrics in case of Top 10 % over long periods of aggregation. This shows that temporal betweenness centrality does measure something which does not entirely agree with what is being measured by the traditional betweenness centrality. In other words, these results do show an inclination towards the fact time agnostic based techniques may consider paths which are not feasible for information flow.

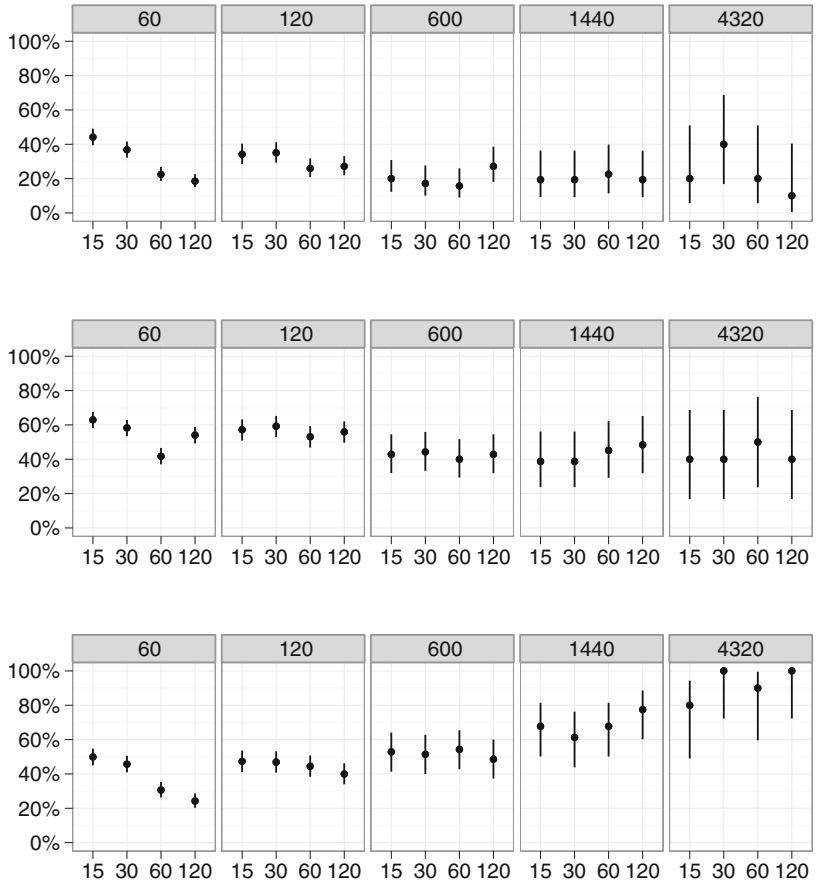


Fig. 20 Percentage of aggregation sets, across all maximum allowed wait (X-axis) and all levels of aggregation (separate plots, level of aggregation in minutes in *grey bar*) where the most central agent in the snapshot method is in the top one (*top set of plots*), top three (*middle set*) or top 10 % (*bottom set*) of most central agents using our approach (best viewed in color)

8 Conclusion

Temporally-detailed Social network Analytics (TDSNA) has value addition potential due to its potential to answer time-aware questions about the underlying social system, e.g., "How is the betweenness centrality of an individual changing over time?" However, designing scalable algorithms for TDSNA is challenging because of the non-stationary ranking of shortest path between any two nodes in a TD social network. This non-stationary ranking violates the assumptions of dynamic programming underlying the common shortest path algorithms such as Dijkstra's used by current techniques. To this end, we proposed the concept of epoch-point based approaches which divide the given time interval-over which we observe non-stationarity-into a set of disjoint time intervals which show stationary ranking. Based on the concept of epoch-points, we developed a novel algorithm, called EBETS, for computing the temporal extension of betweenness centrality. Experimental evaluation showed that EBETS out performs the alternatives. In future, we plan to extend our concept of

epoch-based algorithms to adapt floyd warshall's algorithm for all-pair shortest paths. This would be useful in developing more scalable algorithms for temporal extensions of closeness centrality. We also plan to extend our approach to centrality metrics in multiplex networks and networks where interaction among a group of people are recorded through hyperedges.

Acknowledgements We are particularly grateful to the members of the Spatial Database Research Group at the University of Minnesota and Computational Analysis of Social and Organizational Systems at Carnegie Mellon University for their helpful comments and valuable suggestions. This work was supported by the Center for Artificial Intelligence at the IIIT-Delhi, NSF Grants (Grant No. NSF IIS-1320580, NSF No. 0940818, NSF IIS-1218168) and USDOD Grant (Grant No. HM1582-08-1-0017). The content does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

References

Agresti, A., & Coull, B. A. (1998). Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, *52*(2), 119–126.

Anthonisse, J. M. (1971). The rush in a directed graph. CWI technical report Stichting Mathematisch Centrum. Mathematische Besliskunde-BN 9/71, Stichting Mathematisch Centrum

Barabási, A. L. (2005). The origin of bursts and heavy tails in human dynamics. *Nature*, *435*(7039), 207–211.

Bertsekas, D. P. (1987). *Dynamic programming: Deterministic and stochastic models*. Englewood Cliffs, NJ: Prentice-Hall.

Borgatti, S. P., Carley, K. M., & Krackhardt, D. (2006). On the robustness of centrality measures under conditions of imperfect data. *Social Networks*, *28*(2), 124–136.

Braha, D., & Bar-Yam, Y. (2006). From centrality to temporary fame: Dynamic centrality in complex networks. *Complexity*, *12*(2), 59–63.

Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, *25*(2), 163–177.

Bucholtz, M., & Hall, K. (2005). Identity and interaction: A sociocultural linguistic approach. *Discourse Studies*, *7*(4–5), 585–614.

Cats, O., & Jenelius, E. (2014). Dynamic vulnerability analysis of public transport networks: Mitigation effects of real-time information. *Networks and Spatial Economics*, *14*(3–4), 435–463.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms*. Cambridge: MIT Press.

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal Complex Systems*, *1695*. URL http://igraph.sf.net.

Delling, D. (2008). Time-dependent sharc-routing. In *Proceedings of the 16th annual European symposium on Algorithms* (pp 332–343).

Delling, D., & Nannicini, G. (2008). Bidirectional core-based routing in dynamic time-dependent road networks. In *Proceedings of the 19th international symposium on algorithms and computation, ISAAC'08* (pp. 812–823). Springer.

Delling, D., & Wagner, D. (2007). Landmark-based routing in dynamic graphs. *Experimental algorithms* (pp. 52–65). Berlin: Springer.

Delling, D., & Wagner, D. (2009). Time-dependent route planning. *Robust and online large-scale optimization* (Vol. 5868, pp. 207–230)., Lecture notes in computer science Berlin: Springer.

Demiryurek, U., Banaei-Kashani, F., Shahabi, C., & Ranganathan, A. (2011). Online computation of fastest path in time-dependent spatial networks. In: *Proceedings of the 12th international conference on advances in spatial and temporal databases, SSTD'11* (pp. 92–111) Springer.

Ding, B., Yu, J. X., & Qin, L. (2008). Finding time-dependent shortest paths over large graphs. In *Proceedings of the 11th international conference on extending database technology (EDBT)* (pp 205–216).

Eckmann, J. P., Moses, E., & Sergi, D. (2004). Entropy of dialogues creates coherent structures in e-mail traffic. *Proceedings of National Academy of Sciences*, *101*(40), 14333–14337.

Frantz, T. L., Cataldo, M., & Carley, K. M. (2009). Robustness of centrality measures under uncertainty: Examining the role of network topology. *Computational and Mathematical Organization Theory*, *15*(4), 303–328.

Freeman, L. (1979). Centrality in social networks conceptual clarification. *Social Networks*, *1*(3), 215–239.

George, B., Kim, S., & Shekhar, S. (2007). Spatio-temporal network databases and routing algorithms: A summary of results. In: *Symposium on spatial and temporal databases (SSTD)* (pp. 460–477).

George, B., & Shekhar, S. (2007). Time-aggregated graphs for modelling spatio-temporal networks. *Journal on Data Semantics*, *11*, 191.
Habiba, H., Tantipathananandh, C., & Berger-Wolf, T. Y. (2007). Betweenness centrality measure in dynamic networks. Technical report 2007–19, Center for Discrete Mathematics and Theoretical Computer Science.
Hage, P., & Harary, F. (1995). Eccentricity and centrality in networks. *Social Networks*, *17*(1), 57–63.
Howison, J., Wiggins, A., & Crowston, K. (2011). Validity issues in the use of social network analysis with digital trace data. *Journal of the Association for Information Systems*, *12*(12), 767.
Johnson, R., Kovcs, B., & Vicsek, A. (2012). A comparison of email networks and off-line social networks: A study of a medium-sized bank. *Social Networks, 34*(4), 462–469.
Kanoulas, E., Du, Y., Xia, T., & Zhang, D. (2006). Finding fastest paths on a road network with speed patterns. In *Proceedings of the 22nd international conference on data engineering (ICDE)* (p. 10).
Kim, H., & Anderson, R. (2012). Temporal node centrality in complex networks. *Physcial Review E*, *85*(2), 026107.
Kim, H., Tang, J., Anderson, R., & Mascolo, C. (2012). Centrality prediction in dynamic human contact networks. *Computer Networks*, *56*(3), 983–996.
Kim, P. J., & Jeong, H. (2007). Reliability of rank order in sampled networks. *The European Physical Journal B-Condensed Matter and Complex Systems*, *55*(1), 109–114.
Kossinets, G., Kleinberg, J., & Watts, D. (2008). The structure of information pathways in a social communication network. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '08* (pp. 435–443).
Kossinets, G., & Watts, D. J. (2006). Empirical analysis of an evolving social network. *Science*, *311*(5757), 88–90.
Lerman, K., Ghosh, R., & Kang, J. H. (2010). Centrality metric for dynamic network analysis. In *Proceedings of KDD workshop on mining and learning with graphs (MLG)*
Malmgren, R. D., Hofman, J. M., Amaral, L. A., & Watts, D. J. (2009). Characterizing individual communication patterns. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '09* (pp. 607–616). New York, NY, USA: ACM.
Malmgren, R. D., Stouffer, D. B., Motter, A. E., & Amaral, L. A. N. (2008). A poissonian explanation for heavy tails in e-mail communication. *Proceedings of the National Academy of Sciences*, *105*, 18153–18158.
Nannicini, G., Delling, D., Liberti, L., & Schultes, D. (2008). Bidirectional a* search for time-dependent fast paths. *Experimental algorithms* (pp. 334–346). Berlin: Springer.
Nannicini, G., Delling, D., Schultes, D., & Liberti, L. (2012). Bidirectional a* search on time-dependent road networks. *Networks*, *59*(2), 240–251.
Nia, R., Bird, C., Devanbu, P., & Filkov, V. (2010). Validity of network analyses in open source projects. In *2010 7th IEEE working conference on mining software repositories (MSR)* (pp 201–209).
Palinkas, L. A., Johnson, J. C., Boster, J. S., & Houseal, M. (1998). Longitudinal studies of behavior and performance during a winter at the South Pole. *Aviation, Space, and Environmental Medicine*, *69*(1), 73–77.
Russel, S., & Norwig, P. (1995). *Artificial intelligence: A morden approach*. Upper Saddle River, NJ: Prentice-Hall.
Sabidussi, G. (1966). The centrality index of a graph. *Psychometrika*, *31*(4), 581–603.
Sampson, S. F. (1968). A novitiate in a period of change: An experimental and case study of social relationships. Ph.D. thesis, Cornell University.
Shimbel, A. (1953). Structural parameters of communication networks. *The Bulletin of Mathematical Biophysics*, *15*(4), 501–507.
Tambayong, L., & Carley, K. M. (2012). Network text analysis in computer-intensive rapid ethnography retrieval: An example from political networks of sudan. *Journal of Social Structure*, *13*(2), 1–24.
Tang, J., Musolesi, M., Mascolo, C., & Latora, V. (2009). Temporal distance metrics for social network analysis. In *Proceedings of the 2Nd ACM workshop on online social networks, WOSN '09* (pp. 31–36). New York, NY, USA: ACM.
Tang, J., Musolesi, M., Mascolo, C., Latora, V., & Nicosia, V. (2010). 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).
Vaisey, S., & Lizardo, O. (2010). Can cultural worldviews influence network composition? *Social Forces*, *88*(4), 1595–1618.
Wang, D., Wen, Z., Tong, H., Lin, C. Y., Song, C., & Barabsi, A. L. (2011). Information spreading in context. In *Proceedings of the 20th international conference on world wide web, WWW '11* (p. 735–744) ACM.
Wei, W., & Carley, K. M. (2014). Real time closeness and betweenness centrality calculations on streaming network data. In ASE BIGDATA/SOCIALCOM/CYBERSECURITY conference.

Wei, W., & Carley, K. M. (2015). Measuring temporal patterns in dynamic social networks. *ACM Transactions on Knowledge Discovery from Data*, *10*, 9.

Weng, L., Flammini, A., Vespignani, A., & Menczer, F. (2012). Competition among memes in a world with limited attention. *Scientific Reports*, *2*, 335.

Wu, F., & Huberman, B. A. (2007). Novelty and collective attention. *Proceedings of the National Academy of Science USA*, *104*(45), 17599–17601.