Keyword Search on Temporal Graphs

Ziyang Liu, Chong Wang, and Yi Chen, *Member, IEEE*

Abstract—Archiving graph data over history is demanded in many applications, such as social network studies, collaborative projects, scientific graph databases, and bibliographies. Typically people are interested in querying temporal graphs. Existing keyword search approaches for graph-structured data are insufficient for querying temporal graphs. This paper initiates the study of supporting keyword-based queries on temporal graphs. We propose a search syntax that is a moderate extension of keyword search, which allows casual users to easily search temporal graphs with optional predicates and ranking functions related to timestamps. To generate results efficiently, we first propose a best path iterator, which finds the paths between two data nodes in each snapshot that is the "best" with respect to three ranking factors. It prunes invalid or inferior paths and maximizes shared processing among different snapshots. Then, we develop algorithms that efficiently generate top-k query results. Extensive experiments verified the efficiency and effectiveness of our approach.

Index Terms—Temporal graph, versioned graph, keyword search

1 Introduction

Archiving graph data is important in many applications. For example, in social network analysis, scientists are often interested in analyzing the temporal dynamics of social relationships in order to understand how things change and to predict trends. We even want to archive the whole Web (i.e., the pages and their links) [1], so that our future generations are able to see what is happening today, and analyze how things evolve. In a collaborative project, we would like to archive previous versions of data (such as workflows or programs) [2] so that an earlier version can be recovered in case of a mistake. Beyond the basic operations such as retrieving a snapshot from the archive and tracing the history of an element, often users want to query temporal graphs, as illustrated in some examples below.

- *Social networks*.
  Q1: Find k earliest relationships between Mary and John.
  Q2: Find all friends of Mike, ranked by the descending duration of friendship.
  Q3: Find people who have been employed by Microsoft before 2016.
- *Bibliography*.
  Q4: Find the paper written by "Dimitris" from 2004 to 2006
  Q5: Find the earliest relationship of Gray and SIGMOD.
  Q6: Find the paper on "graph search" published after 2015.
- *Workflow Design*.
  Q7: Find the relationship between Tuberin and Hamartin discovered after 2004, ranked by the time of discovery.
  Q8: Find the subworkflows containing tasks "GenBank" and "Process Blast" which no longer existed after July, 2010.
  Q9: Find the workflows containing task "spectral analysis" that are created after 2009.

To address such query needs, studies have been performed on designing a temporal relational model [3] and a temporal XML tree model [4], and developing query languages on temporal data, including TQUEL [5], TSQL2 [3] and SQL3 [6] for temporal relational data, and TXPath [4] for temporal XML trees.

However, performing structured temporal queries on temporal databases is not suitable for some applications for two reasons. First, many applications have graph structured data which can not be efficiently handled by relational databases as generally one join is required to navigate each edge. Second, structured queries are difficult for casual users to learn and are error-prone even for expert users.

To allow casual users who are not computer experts (as found in Web and scientific applications) to express queries on temporal graphs, it is critical to support keyword-based searches on temporal graphs. There is much research on keyword search on graphs that do not encode temporal information [7], [8], [9], [10], [11], [12]. Given a data graph, each query result is a minimal tree that contains all query keywords, and the results are ranked by the reverse of weighted tree sizes. To generate top-k results efficiently, a widely-adopted approach is based on Dijkstra's shortest path algorithm ([7], [9], [10], [12]), which explores the graph from each keyword match using a shortest path iterator. A result is found if a node is visited by iterators from all query keywords. In this way, results are likely generated in the order of their sizes, thus enabling efficient top-k processing.

- Z. Liu is with Facebook, Inc., Menlo Park, CA 94025. E-mail: ziliu@fb.com.
- C. Wang and Y. Chen are with the New Jersey Institute of Technology, Newark, NJ 07102. E-mail: {cw87, yi.chen}@njit.edu.

Manuscript received 21 July 2016; revised 18 Jan. 2017; accepted 6 Mar. 2017. Date of publication 4 Apr. 2017; date of current version 5 July 2017.
Recommended for acceptance by C. Ordonez.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2017.2690637

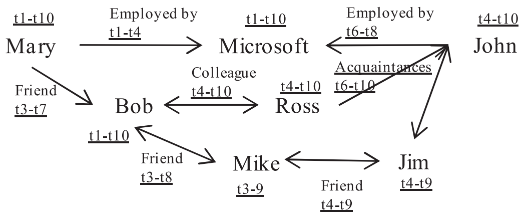


Fig. 1. Social network temporal graph.

We investigate the problem of processing keyword-based queries on temporal graphs. Adapting the existing techniques, one approach is to store every snapshot of the graph, and run existing keyword search methods on each snapshot. However, this will require excessive storage space and a huge number of traversals on the graph (one for each snapshot). Instead, we use a more concise data model that "unions" the snapshot graphs in different time instants to a temporal graph, such as the one in Fig. 1. In this graph, each node and edge can have a label on its *valid times*.

To process a keyword query, one approach is to run one of the existing graph keyword search methods on such a temporal graph to find candidate results oblivious to the timestamps, and then remove the invalid ones (i.e., the results where nodes and edges do not have a common timestamp) at the end. However, such an approach suffers from low efficiency and low result quality: although invalid results are discarded by post-processing, much time may be wasted in generating such results, and valid results may be missing. For example, for query "*Mary, John*" against Fig. 1, a result containing nodes *Mary*, *Microsoft* and *John* will be generated and then discarded due to its invalidity, and this method will fail to generate any result for query "*Mary, John*". In fact, there are valid results for this query: *Mary - Bob - Ross - John* at time instant t6 and t7, and *Mary - Bob - Mike - Jim - John* at t4.

Furthermore, adapting existing techniques as discussed above is also inadequate to support queries involving the temporal aspects of data. For instance, a user may want to rank results based on temporal information, e.g., Q1 and Q2 rank results by time and duration time, respectively. A user may also want to restrict the search using temporal predicates, e.g., "before 2016" in Q3. Existing keyword search methods ignore temporal information and cannot efficiently generate top-k results for queries that involve temporal-based ranking functions and/or temporal predicates.

In this paper we study an open problem of processing keyword search on temporal graphs. We first propose a simple and yet expressive extension to keyword queries to support temporal predicates that are able to express the most common relationships between two time intervals. We support three types of ranking functions: ranking by descending order of relevance/end time/duration, ranking by ascending order of start time, and their combinations.

To generate valid results efficiently, we propose a *best path iterator* algorithm, which finds the valid paths between two data nodes in each time instant that is the "best" with respect to a specified ranking function among all valid paths. This algorithm extends Dijkstra's shortest path algorithm to process temporal graphs, and has the following advantages: First, it avoids generating invalid paths. Second, given two nodes, it guarantees to find the best path between them in each time instant (if exists), and thus avoids the problem of missing results due to version incompatibility. Third, it maximizes shared processing among different snapshots. Besides the extension of Dijkstra's algorithm for shortest paths on temporal graphs, we also make a principled generalization of the algorithm to support other ranking functions, including the commonly used ranking functions for temporal graphs, such as ranking by result time and by duration. Moreover, we leverage advanced data structures that use bitmaps to store the temporal information of the best paths for fast computation.

Given the best path iterator, we integrate it into existing approaches for keyword search on graphs, and address the unique challenges for generating top-k results when searching temporal graphs and for supporting temporal based ranking functions. To handle relevance-based ranking that considers result size, existing keyword search methods always give a higher priority to expand path exploration with the highest rank. However, such a strategy does not work for temporal-based ranking since the relevance-based ranking is strictly decreasing with further exploration versus temporal-based ranking functions are non-increasing. We propose techniques for path exploration and for estimating the score upper bound of unseen results, which is proved to be tight. We discuss the tradeoffs between quality and efficiency for different upper bound estimation.

The contributions of this paper can be summarized as: 1) This paper initiates a study of searching temporal graphs. 2) We have defined a simple query syntax that supports temporal predicates and ranking factors. Adapted from TSQL2, this syntax can express all relationships between the result time and a given time interval (Section 2). 3) We have developed two principled generalization of Dijkstra's algorithm for shortest paths. One is to handle temporal graphs where nodes have timestamps to achieve snapshot reducibility [5]. 4) The other is to characterize the type of ranking functions that it can support, beyond the distance function in the traditional setting (Section 3). 5) We have designed efficient algorithms for top-k result generation that handle keyword queries on temporal graphs (Section 4). 6) Experimental evaluations have verified the efficiency and effectiveness of the proposed algorithms (Section 6).

2 Problem Definition

2.1 Background of Keyword Search on Graphs

There is much existing work on keyword search on non-temporal graphs (see Section 7). Each node in the graph has a label, representing its tag or value. Nodes and edges in the graph may have weights. A keyword in the query can match words in the labels of one or more nodes in the data. An answer to a query is a subtree of the data graph that contains matches to all query keywords. Subtrees with smaller size are considered better results and ranked higher. We adopt a similar model in this paper, except that nodes are also associated with time intervals, and the search syntax is enriched with temporal clauses, as defined in the following sections.

2.2 Data Model

We define the temporal graph model in a similar way as the temporal XML model used in [4]. The data is modeled as a directed graph, where each node n and edge e is annotated

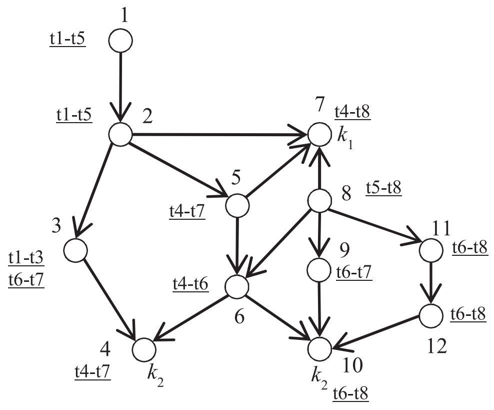


Fig. 2. Sample temporal data graph.

with one or more time intervals in which it is *valid* (i.e., exists), denoted as $val(n)$ and $val(e)$, respectively. A time interval is defined by a start time and an end time. The graph should be valid at any timestamp, thus for every node n, $val(n)$ must be a superset of every $val(e)$, where n is an endpoint of e. A node has a label, recording its name/tag or its value. Each node and edge may optionally have a weight, denoted as $w(n)$ and $w(e)$, respectively. A sample data graph is shown in Fig. 2, where the valid times of an edge is the intersection of the valid times of its endpoints, and are omitted in the figure. Node 7 has a label k_1, and nodes 4 and 10 have a label k_2. Every edge has a unit weight.

In a temporal relational model, there are two types of times: transaction time and valid time [3]. Following [4], we work with a simpler model with one type of time, transaction or valid time. This is sufficient for many applications as end users typically only care about valid time.

2.3 Query Syntax

We support extended keyword queries to temporal graphs, where a users can issue a query with optional temporal predicates and temporal ranking functions as defined below.

Definition 2.1. *The query syntax for searching temporal graphs is defined as:*

- $<Q>$ ::= $<$KEYWORD$>+$ $<$PRED$>^*$ $<$RF$>^*$

where "$<$KEYWORD$>+$" represents one or more keywords, each of which may match the labels of data nodes, "$<$PRED$>^*$" represents zero or more temporal predicates, and "$<$RF$>^*$" represents zero or more ranking factors.

We support the following temporal predicates:

- $<$PRED$>$ ::= RESULT TIME
  - {PRECEDES | FOLLOWS} t_x
  - MEETS t_x
  - OVERLAPS $[t_x, t_y]$
  - {CONTAINS | CONTAINED BY} $[t_x, t_y]$

"result time" refer to the time when the result exists (Definition 2.2). While other predicates are self-explanatory, predicate "meets t_x" means that the result is valid in t_x, and is either invalid in any time instant before t_x, or invalid in any time instant after t_x. Predicates can be combined using AND, OR and NOT.

We support the following ranking factors:

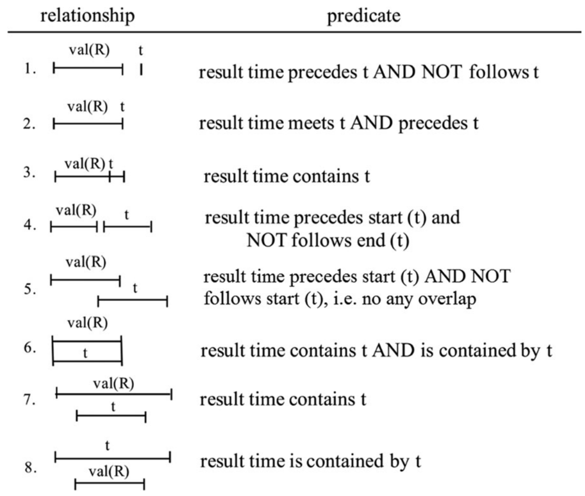

relationship
predicate
1. val(R) t
result time precedes t AND NOT follows t
2. val(R) t
result time meets t AND precedes t
3. val(R) t
result time contains t
4. val(R) t
result time precedes start (t) and NOT follows end (t)
5. val(R) t
result time precedes start (t) AND NOT follows start (t), i.e. no any overlap
6. val(R) t
result time contains t AND is contained by t
7. val(R) t
result time contains t
8. t val(R)
result time is contained by t

Fig. 3. Relationship between result time ($val(R)$) and a single time instant t or a time interval I.

- $<$RF$>$ ::= RANK BY
  - DESCENDING ORDER OF {RELEVANCE | RESULT END TIME | DURATION }
  - ASCENDING ORDER OF RESULT START TIME

The syntax of temporal predicates follows the syntax of TSQL2 [3], the temporal extension to the SQL language standard. It can express any relationships between the result time and a time instant t_x or a time interval $[t_x, t_y]$, as illustrated in Fig. 3. For each pair of symmetric relationships, we only show one of them in the figure. As an example, Table 1 shows Q1-Q3 under this syntax.

We follow state-of-the-art approaches [7], [8], [9], [10], [11], [12] to define relevance as the weighted result tree size, the smaller, the better. For the other ranking factors, "result time" and "duration" are defined in Section 2.4.

To facilitate users to express temporal queries, the system can provide a graphical interface with drop-down menus that list options for user selections.

2.4 Query Result Definition

Following most existing approaches for keyword search on regular graphs, we adopt minimal tree result definition, but extend it with a *validity* constraint: all nodes and edges of a result must co-exist in at least one time instant.

Definition 2.2. *The result of keyword query Q on data graph G is defined by $(T, val(T))$, such that:*

- *T is a rooted, connected subtree that contains all keywords in Q,*
- *removing any node of T will make T no longer a tree containing all keywords in Q,*
- *T satisfies the temporal predicates in Q, if any,*
- result time $val(T) \neq \emptyset$.

where $val(T)$ denotes the valid time of T, i.e., the set of time instants in which every node n and every edge e in T is valid,

TABLE 1
Queries Corresponding to the Questions Q1-Q3 in Section 1

Q1: "**Mary, John**", rank by *ascending* order of *start time*
Q2: "**Mike, friend**", rank by *descending* order of *duration*
Q3: "**Microsoft, employee**", result time *precedes* **2016**

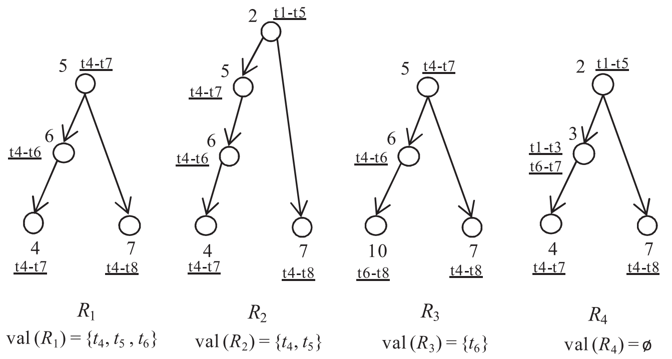


Fig. 4. Valid and invalid results of query "k_1, k_2".

i.e., $val(n) \supseteq val(T)$, $val(e) \supseteq val(T)$, $\forall n, e \in T$. The start, end, and duration (number of time instant included) of result time can be used in ranking.

Example 2.1. Consider query Q = "k_1, k_2" on the data shown in Fig. 2. Fig. 4 shows four of the eight subtrees of the graph that satisfies the first three conditions of Definition 2.2. $(R_1, \{t_4, t_5, t_6\})$, $(R_2, \{t_4, t_5\})$ and $(R_3, \{t_6\})$ are all valid results of this query. Since $val(R_4) = \emptyset$, R_4 does not correspond to a query result.

3 Temporal-Aware Best Path Iterator

To efficiently process the queries defined in Section 2, we adapt a commonly used framework for processing keyword search on graphs [7], [9], [10], [12], which are based on Dijkstra's shortest path algorithm. We first propose temporal-aware best path algorithms in this section that extend Dijkstra's algorithm to efficiently find the "best valid path" between two nodes in every time instant with respect to ranking functions supported. In Section 4, we discuss how to use the best path iterator to generate top-k results.

3.1 Best Paths for Ranking by Relevance

As shown in the example in Section 1, the shortest path between two nodes oblivious to the temporal information may not be a valid path. Furthermore, the shortest path between two nodes in a temporal graph may be different at different time instants. Thus we propose an extension to the Dijkstra's single-source shortest path algorithm in order to generate valid shortest paths for every time instant on a temporal graph where nodes are annotated with timestamps. This algorithm can achieve good search quality by avoiding missing results due to time incompatibility. It also avoids redundant processing on nodes and avoid generating invalid paths as much as possible to achieve efficiency.

Let us start with a brief review on Dijkstra's single-source shortest path algorithm. Each node is assigned a distance. Initially, the distance of the source is 0, and the distances of all other nodes are infinity. In each iteration, we choose the node which has the smallest distance to the source and which has not been chosen before, and then we update the distances of its neighbors. At the time when a node is chosen, its recorded distance is the length of the shortest path to the source. Note that in this case, the exploration unit is a node.

We propose an extension to Dijkstra's algorithm that computes the shortest valid paths between two nodes in a temporal graph in all time instants. Since a node may have different distances to the source at different time instants, we record a set of (node, time interval, distance) triplets for each node, referred to as *NTD triplet* (n, T, d). An NTD triplet (n, T, d) records the current shortest distance d from the source to node n during time interval T (In contrast, Dijkstra's algorithm only needs an (n, d) pair without T). Since we compute the shortest path for every time instant, the union of T in all NTD triplets associated with a node n is equal to the whole span of n's time intervals $val(n)$. Each time instant in $val(n)$ should appear exactly once among all the NTD triplets of a node. For different time instants, if the same path is the shortest one between the source and node n in multiple time instants, then we only need one NTD, where T records the corresponding time instants. Using NTD triplet as the graph exploration unit, we can correctly find shortest paths in every time instant, and meanwhile, maximize shared processing, which is critical for efficiency.

Next we discuss our single-source shortest paths algorithm for temporal graph. Initially, the source node s has a single NTD triplet $(s, val(s), w(s))$; each other node n has a single NTD triplet $(n, val(n), \infty)$. Each time, we choose the NTD triplet (n, T, d) that has the smallest d over all the NTD triplets that have not been chosen before. At this time, d is the shortest distance from source s to n during T.

After selecting an NTD triplet (n, T, d), we update the NTD triplets of all nodes n', where there is an edge from n' to n. Recall that in Dijkstra's algorithm, for each neighbor n' of n, we simply re-calculate its distance, and replace the original distance if the new distance is smaller. On temporal graphs, however, we need to check the temporal relationship between n and n'. This consists of two steps. First we compute the intersection of T and the time instants associated with the edges from n' to n, i.e., $val(n' \rightarrow n)$. Let $T_\cap = T \cap val(n' \rightarrow n)$. $T_\cap$ is the set of "surviving" time instants after expanding the NTD triplet (n, T, d) from n to n'. Note that since $val(n' \rightarrow n) \subseteq val(n')$, $T_\cap$ is among the valid times of node n' as well. If $T_\cap = \emptyset$, we can omit the next step. Second, for each NTD triplet (n', T', d') associated with a node n', if T' does not intersect with $T_\cap$, then this NTD triplet is not affected. Otherwise, let $T'' = T_\cap \cap T'$. If for the time instants in T'' the path from s to n' through n has a smaller distance than the current shortest path from s to n', i.e., $d + w(n, n') + w(n') < d'$, then we update NTD triplet (n', T', d'). Specifically, we remove NTD triplet (n', T', d'), and create two new NTD triplets: $(n', T'', d + w(n, n') + w(n'))$ corresponding to the newly found shortest path valid in T'', and $(n', T' \backslash T'', d')$ (if $T' \backslash T'' \neq \emptyset$) corresponding to the unaffected time instances. Otherwise, if $w(n, n') + w(n') + d \geq d'$, then the shortest distance from source to n' in T' is still d', and we do not need to change this NTD triplet. After an NTD triplet (n, T, d) has been selected, it does not need to be selected again since the shortest distance from the source to n during T has been computed (i.e., d), and its neighbors have been updated.

The algorithm for finding the best paths between two nodes in a temporal graph is shown in Algorithm 1. We store all NTD triplets (instead of nodes, as in Dijkstra's algorithm) in a priority queue, whose key is distance d. When we update a NTD triple (n', T', d') of a neighbor of the selected NTD triplet, we need to remove (n', T', d'). Deleting an item from a priority queue requires $O(\log N)$ time, where

N is the number of nodes in the queue. Notice that when $T'\backslash T'' \neq \emptyset$, such deletion is actually followed by an insertion $(n', T'\backslash T'', d')$. Thus instead of performing the deletion literally, we make an in-place update. Specifically, we keep the original NTD triplet (n', T', d'), but for each $t \in T''$, we mark $visited(n', t)$ = true. Later when (n', T', d') is selected from the queue, we will no longer consider those t such that $visited(n', t)$ = true (lines 5-6 of procedure BESTPATHITERATOR). Since $visited(n, t)$ is small, in-place update is more efficient than a real deletion followed by an addition. On the other hand, when $T'\backslash T'' = \emptyset$, this method results in useless NTD triples in the queue. Although a real removal of (n', T', d') can be done, experiments show that on average only 0.04 percent of the items in the priority queue are useless, thus the overhead is negligible.

Algorithm 1. Best Path Iterator

```
BESTPATHITERATOR (source s, ranking function)
 1: $pq$ = empty priority queue {The sorting function of $pq$
    depends on the ranking function of the query}
 2: Create NTD triplet $(s, val(s), weight(s))$, and push it to $pq$
 3: while $pq$ is not empty do
 4:   $(n, T, d)$ = the NTD triplet on top of $pq$
 5:   if $visited(n, t)$ = true for all $t \in T$ then
 6:     continue
 7:   for each $t \in T$ do
 8:     $visited(n, t)$ = true {This (node, time instant) pair will
        no longer be considered}
 9:   end for
11: UPDATENEIGHBOR $(n, T, d)$
12: end while
UPDATENEIGHBOR $(n, T, d)$
 1: for each node $n'$ such that there is an edge from $n'$ to $n$ do
 2:   $T_\cap = T \cap val(n' \to n)$
 3:   for each NTD triplet $(n', T', d')$ of $n$ do
 4:     $T'' = T_\cap \cap T'$
 5:     if $w(n, n') + d < d'$ then
 6:       create an NTD triplet $(n', T'', d + w(n, n') + w(n'))$, and
          push it into $pq$
```

Example 3.1. Consider the graph in Fig. 2 as an example where every edge has a weight of 1 and every node weighted 0. Suppose node 10 is the source node, and we are looking at the point where the NTD triplet with the smallest d is $(11, t_6 - t_8, 2)$. So we choose this NTD triplet, and update the neighbors of node 11, i.e., node 8. We intersect $t_6 - t_8$ with $val(8) = t_5 - t_8$, and get $T_\cap = t_6 - t_8$. Suppose node 8 currently has an NTD triplet $(8, t_6 - t_7, 2)$, i.e., the current shortest distance from 10 to 8 in time instants t_6 and t_7 is 2. Since 2 < 2 + 1 = 3, we remove t_6 and t_7 from $T_\cap$, and create a new NTD triplet for node 8, which is $(8, t_6 - t_8, 3)$, denoting that the shortest distance from 8 to 10 in time instant $t_6 - t_8$ is 3. The existing NTD triplet for node 8 $(8, t_6 - t_7, 2)$ is also kept.

Proposition 3.1. *Algorithm 1 can correctly compute the shortest path from a source to every other node on a temporal graph in every time instant, if exists.*

Proof. We first prove that every path we output must be the shortest path in the corresponding time instant. In Algorithm 1, at each step, an NTD triplet (n, T, d) with the smallest distance is selected. We claim that $\forall t \in T$ where $visit(n, t) = false$, the shortest distance from source to n at time t must be d. Suppose otherwise, the shortest distance from source to n at t is $d', d' < d$. Since each step we select the NTD triplet with the smallest distance, thus the NTD triplet (n, T', d') where $t \in T'$ must have already been selected. At that time, we set $visit(n, t) = true$, which is a contradiction.

Now we prove that the shortest path from source to a node n in every time instant, if exists, will be found. Note that every time instant in $val(n)$ must appear in at least one of the NTD associated with n, and $visit(n, t)$ can turn from $false$ to $true$ at most once. As proved above, when we select an NTD triplet (n, T, d) where $visit(n, t)$ turns to $true$ for $t \in T$, we find the shortest path from source to n at t. If $visit(n, t)$ never turns to $true$, then we must have d equals to infinity. Since d is not updated after initialization, there is no path from source to n during T. □

In other words, the paths generated by Algorithm 1 are the same as those generated by running Dijkstra's algorithm on the graph snapshot in each timestamp and then merging duplicate paths in different timestamps. This is referred as *snapshot reducibility* [5].

Let $|V|$ be the number of nodes, $|E|$ be the number of edges, and T be the number of time instants in the data graph. For Algorithm 1, the worst-case time complexity of ranking by relevance is $O(T^2(|V|log(T * |V|) + |E|))$, assuming the complexity of computing time intersection is $O(T)$. The worst case happens when the shortest path from a source to every node is different in every time instant, thus we have $T|V|$ NTD triples to process, which is unlikely to happen in practice.

3.2 Best Paths for Ranking by Time

Now we discuss how to generate valid best paths when query results need to be ranked by descending order of result end time. To handle ranking by ascending order of result start time is symmetric. One question is if we can adapt Algorithm 1 that finds the path between two nodes with shortest distance to find the path between two nodes with latest end time. Specifically, in each iteration, instead of choosing the NTD triplet with the smallest distance, we choose the triplet with the latest end time. In other words, using end time instead of distance as the key for the priority queue. We approved the correctness of this approach in Proposition 3.2.

Proposition 3.2. *Algorithm 1 can correctly compute the path with the latest end time, from a source to every other node in every time instance, if exists.*

Proof. When ranking by end time, we claim that for any NTD triplet (n, T, d) that we select at each time, if $t \in T$ and $visitied(n, t)$ = false, then the corresponding path must be the best path among all paths from source to n that is valid at t. The key observation is that the correctness of the Dijkstra's algorithm is based on the *monotonicity* property: when we expand a path, the length of the path will not decrease. Thus each time, it selects an unvisited node with the shortest distance to the source, which must be the true shortest distance to the source. Similarly, when we rank by end time, extending a path cannot increase its end time, In other words, the quality of the path monotonically decreases when the path becomes longer. Due to the monotonicity property, we have a similar argument as the one in the proof of Proposition 3.1. If (n, T, d) is currently the best NTD triplet, it cannot be outperformed by expanding from any other NTD triplet. Therefore, it must correspond to the best path at time instants in t from source to n. Besides, for every time instant t, we will find the path with the latest end time among all paths from source to n that is valid at time t. □

The proof of this Proposition leads to the corollary:

Corollary 3.3. *Algorithm 1 and the original Dijkstra's algorithm for shortest paths can be used to find best paths, when the rank of a path is monotonically non-increasing upon an edge expansion.*

This captures the principles of our generalization of Dijkstra's algorithm for different ranking functions. The traditional Dijkstra's algorithm is used in the literature for ranking by relevance, which is proportional to the reverse of path distance. In this work, we show that Algorithm 1 can be used when rank by descending order of end time, by descending order of duration (Section 3.3), which represent two instances of such a generalization.

Example 3.2. Consider Fig. 2, and suppose the source node is 10, which has an initial NTD triplet $(10, t_6 - t_8, 0)$. We start with node 10, and update the NTDs of its neighbors: $(6, t_6, 1)$, $(9, t_6 - t_7, 1)$ and $(12, t_6 - t_8, 1)$, respectively. In the next iteration, since NTD triplet $(12, t_6 - t_8, 1)$ has the latest end time, it is chosen for further expansion. Using this method, we can quickly find the path from 8 to 10 that has the latest end time, i.e., $8 \rightarrow 11 \rightarrow 12 \rightarrow 10$.

3.3 Best Paths for Ranking by Duration

When ranking by descending order of result duration, the rank is monotonically non-increasing upon an edge expansion, thus we can still use the best path iterator in Algorithm 1 to achieve snapshot reducibility, except that in each iteration, we choose the NTD triplet whose time interval has the longest duration. In other words, the key of the priority queue is duration.

New challenges arise when ranking by duration. When ranking by relevance/result time, we only record each time instant once among all NTDs of a node. However, that would give erroneous results when ranking by duration.

Example 3.3. Consider there are two paths from the source to a node n, p_1 and p_2, whose time intervals and distances are ($t_1 - t_{10}$, d_1) and ($t_6 - t_{15}$, d_2), respectively. Suppose we record NTD triplets: $(n, t_1 - t_{10}, d_1)$ and $(n, t_{11} - t_{15}, d_2)$. There are two paths valid in $t_6 - t_{10}$, and we only record the shorter one.

Now consider a neighbor of n, n', $val(n') = t_4 - t_{15}$. When we expand from n to n' the best path iterator would intersect the two NTD triplets of n with $val(n')$, and obtain two intervals: $t_4 - t_{10}$ and $t_{11} - t_{15}$, which implies that the longest duration from s to n' is $t_4 - t_{10}$. However, this is wrong. The actual longest duration from source to n' is $t_6 - t_{15}$ through path p_2 plus the edge from n to n'. Thus, restricting a time instant to appear in just one NTD triple does not work for duration-based ranking.

As we can see, when ranking by duration, each triplet corresponds to a distinct candidate of path with longest duration, thus a time instant may appear in multiple NTD triplets of a node. While we could record an NTD triplet for each distinct time duration identified during the process, this would result a large number of NTD triplets associated with a node. The question is how to minimize the number of NTD triplets for a node to maximize shared processing, without compromising correctness.

To address this, in each iteration, after selecting an NTD triplet (n, T, d) with the longest duration, we update the NTD triplets of n's neighbors in a different way than the previous two ranking factors. For each node n' that is a neighbor of n, we first compute $T_\cap = T \cap val(n' \rightarrow n)$, which is the "surviving" time interval after expanding from n to n'. Then, we check all NTD triplets of node n' to see if $T_\cap$ subsumes, or is subsumed by, some NTD triplets of n'. There are three cases:

(1) if $T_\cap$ is subsumed by an NTD triplet (n, T, d) of n', then it's impossible for the current path from the source to n to give a longer duration than the path corresponding to (n, T, d), thus we ignore $T_\cap$ and enter the next iteration.

(2) if $T_\cap$ is not subsumed by any NTD triplet of n', we create a new NTD triplet for node n', which is $(n', T_\cap, d + w(n, n'))$ (recall that $T_\cap$ must be contained in $val(n')$ since $val(n' \rightarrow n) \leq val(n')$), and push it into the priority queue.

(3) if $T_\cap$ subsumes some NTD triplets of n', we remove the NTD triplets of n' that are subsumed by $T_\cap$.

As we can see inferior NTD triplets (i.e., the time interval is subsumed by another NTD triplet) are pruned as early as possible to avoid further expanding an inferior path: we prune an NTD triplet (n, T, d) as soon as we find another NTD triplet (n, T', d') such that T' subsumes T. It is not possible to prune (n, T, d) earlier, since as long as T is not subsumed, (n, T, d) could be the winner in the end.

To determine whether $T_\cap$ subsumes or is subsumed by some NTD triplets of n' by scanning all NTD triplets of n' can be inefficient. To improve efficiency, we use a bitmap to represent the NTD triplets of n'. Each row of the bitmap represents the time interval of an NTD triplet of n', and each column of the bitmap represents a single discrete time instant. The bit in the ith row and the jth column is 1 if and only if the time interval of the ith NTD triplet contains time instant j. If n has a large number of NTD triplets, we can compress the columns in this bitmap.

To determine whether a time interval $T_\cap$ is subsumed by some NTD triplets of n', we extract the columns in n''s bitmap that correspond to the time instants in $T_\cap$, and perform an AND operation of all these columns. If the result of the AND operation has at least one '1' bit, it means that there exists at least one NTD triplet of n', whose time intervals subsume $T_\cap$. On the other hand, to determine whether $T_\cap$ subsumes the time interval of any NTD triplet of n', we extract the columns in n''s bitmap that do *not* correspond to the time instants in $T_\cap$, and perform an OR operation of all these columns. If the result of the OR operation has at least one '0' bit, it means there exists NTD triplets of n', whose time intervals are subsumed by $T_\cap$.

Example 3.4. Suppose the bit vector of $T_\cap$ is 11001001, and the bitmap of the NTD triplets of n' is shown in Fig. 5. To see whether $T_\cap$ is subsumed by any of the NTD triplets of n', we extract the columns in the bitmap of n' that correspond to the '1' bits of $T_\cap$, i.e., the 1st, 2nd, 5th, and 8th columns. We perform an AND operation of the four columns, and the result is 0110. Since the result vector contains '1' bits in the 2nd and 3rd positions, it means that the 2nd and 3rd NTD triplets of n' subsumes $T_\cap$.

The pseudo code of updating the neighbors of a node is shown in Algorithm 2, which can be plugged in Algorithm 1 to correctly compute the path with the longest duration from a source to every other node in every time instance, if exists. The time complexity of ranking by duration is $O(2^T|V|(T + Tlog|V|) + T^2|E|)$ in the worst case.

Algorithm 2. Ranking by Duration

```
UPDATENEIGHBOR ($n$, $T$, $d$)
1: for each node $n'$ such that there is an edge from $n'$ to $n$ do
2:     $T_\cap = T \cap val(n' \rightarrow n)$
3:     $vectorSet1 = vectorSet0 = \emptyset$
4:     for $i = 1$ to size of $T_\cap$'s bit vector do
5:         if the $i$th bit of $T_\cap$'s bit vector is 1 then
6:             $vectorSet1 = vectorSet1 \cup$ the $i$th column of $n'$'s bitmap
7:         else
8:             $vectorSet0 = vectorSet0 \cup$ the $i$th column of $n'$'s bitmap
9:     $result1$ = AND of all vectors in $vectorSet1$
10:    $result0$ = OR of all vectors in $vectorSet0$
11:    if $result1$ has at least one '1' bit then
12:        continue
13:    else if $result0$ has at least one '0' bit then
14:        for each '0' bit of $result0$ do
15:            delete the corresponding NTD triplet of $n'$
16:    create a new NTD triplet $(n', T_\cap, d + w(n, n') + w(n'))$, and
       insert it into the priority queue
```

As discussed, the framework in Algorithm 1 can be adapted to handle ranking factors where the rank of a path is monotonically non-increasing upon an edge expansion. For instance, it can handle ranking by descending order of relevance, end time, duration or ascending order of start time, or any combination of them. Other ranking factors, such as ranking by ascending order of relevance, can not be supported using this framework since the non-increasing monotonicity property does not hold. We have discussions on supporting these ranking factors in Section 8.

4 Result Generation

In this section, we discuss how to generate query results based on the best path iterator which efficiently finds the best paths of two nodes in each time instant for the aforementioned ranking factors.

4.1 Generating Results Using Best Path Iterators

The best path iterator discussed in Section 3 can be plugged into any existing backward expansion algorithms (i.e., take the reverse direction of the edges) that involve computing shortest paths [7], [8], [9], [10], [11], [12] to generate results for processing keyword queries on regular data graphs. In our implementation, we use the algorithm in BANKS [9] as the backward search framework. Given a keyword query, BANKS runs multiple copies of Dijkstra's algorithm as path iterators, one for each node matching a query keyword, to find the shortest path from that node to other nodes in the graph. Each path iterator traverses the graph edges in the reverse direction. The idea is to find a common node, where a forward path exists from that node to a keyword match node, for every query keyword. Such paths define a directed connected tree with the common node as the root and the corresponding keyword match nodes as leaves, so that every query keyword has at least one match in the tree. Each tree is considered as a query result, whose relevance ranking score is defined as the inverse of the weighted tree size. In this work, we extend the idea of BANKS to handle temporal data graphs.

	01100010
	11011001
11001001	11001111
	10010111
$T_\cap$	bitmap of n'

Fig. 5. Vector $T_\cap$ and the bitmap of node n'.

Rank by Relevance. The pseudo code of generating query results is shown in Algorithm 3. First, we find the matches to all keywords using an inverted index. Then we run a copy of the best path iterator presented in Algorithm 1, which takes each match as a source node and traverses backward of the graph to find the best path from the source to other nodes in the graph. Recall that when ranking by result relevance, at each time we choose the best path iterator whose next NTD triplet has the smallest distance d. To do so we build a priority queue for all path iterators and pick the NID triplet (n, T, d) at the top. If node n is reached by at least one best path iterator from each keyword in the query, then it is potentially the root of some results.

There are two differences of our algorithm compared to BANKS. First, we need to generate all possible results whose root is n and who contain the shortest path from n to the source node of the NID triplet (n, T, d) considering different time instances. Suppose this source node is a match to a query keyword k_i. We examine every possible set of NTD triplets $NTDset = (NTD_1, \ldots, NTD_{i-1}, (n, T, d), NTD_{i+1}, \ldots, NTD_m)$, where m is the number of keywords in the query, and $NTD_j (1 \le j \le m, j \neq i)$ is an NTD triplet that has reached node n by a best path iterator originating from a match to keyword k_j. Suppose there are r_j path iterators originating from a match to keyword k_j that have reached n at this time, then the possible number of results contributed by this NTD triplet (n, T, d) is $\prod_{k_j \in \text{query}, j \neq i} r_j$. Second, we check result validity before generation. Specifically we intersect the time interval of all NTD triplets in $NTDset$. If the result of the intersection (say T_r) is non-empty, then the source node of each NTD triplet in $NTDset$, node n, and their connecting paths form a subtree rooted at n that is valid in T_r. Similarly as BANKS, we prune the candidate results whose roots only have a single child and do not match a query keyword.

Rank by Result Time or Duration. In practice a user often is interested in top k results only. One unique challenge that we must address is how to generate top ranked results as early as possible when ranking by result time or duration, as illustrated in the following example.

Example 4.1. Consider a query "k_1, k_2" ranking by descending order of end time on the graph shown in Fig. 6.

Keyword k_1 matches node 2 and keyword k_2 matches node 4. Suppose all nodes in the cloud are valid in time t_2. If we select the NTD triplet in each iteration purely based on the latest end time, it will be inefficient to find the query result. This is because we will keep expanding the path iterator starting from node 2 backward (to the cloud) as end time of the nodes visited is t_2. The path iterator starting from node 4 will be stuck at node 3 whose valid time instant is t_1. Thus it will take a long time before we expand from nodes 3 to 1 and to find the result rooted at node 1.

Algorithm 3. Searching Temporal Graphs

```
QueryProcessing (data $G$, inverted index, keywords $k_1, \dots, k_m$,
predicates $P$, ranking function, $m$)
 1: for $i = 1$ to $m$ do
 2:    $M_i$ = the set of nodes matching $k_i$
 3: $\mathcal{M} = \bigcup M_i$
 4: $pq$ = empty priority queue {The sorting function of $pq$
    depends on the ranking function of the query}
 5: for each node $s \in \mathcal{M}$ do
 6:    if qualify$(s, P)$ then
 7:       create a best path iterator (Algorithm 1) with $s$ as
          source, and put it into $pq$
 8: $results = \emptyset$
 9: while $pq$ is not empty do
10:    $itID$ = getNextIterator() {select a best path iterator to
       expand to a new node}
11:    $(n, T, d)$ = the next NTD triplet on top of the priority
       queue of iterator $itID$ {expand the select iterator}
12:    perform an iteration of iterator $itID$
13:    suppose the source of iterator $itID$ matches keyword $k_j$
14:    if $n$ is visited by iterators from all keywords $k_i$, $i \neq j$ then
15:       for each possible set of NTD triplets
          $NTDset = (NTD_1, \dots, NTD_{j-1}, (n, T, d), NTD_{j+1}, \dots,$
          $NTD_m)$, where $NTD_i (1 \leq i \leq m, i \neq j)$ is an NTD trip-
          let originating from keyword $k_i$ and has reached node
          $n$ by some best path iterator do
16:          $CandidateResult$ = a tree rooted at $n$, consisting of
             paths from $n$ to the nodes corresponding to each
             NTD triplet in $NTDset$. In order to find the smallest
             tree, the NTD triplets in $NTDset$ should reach $n$
             from at least two nodes
17:          {Next we check the validity of results, which is neces-
             sary because although each individual path is valid,
             the tree they form may not}
18:          if qualify$(CandidateResult, P)$ then
19:             $result = result \cup CandidateResult$
20:    if $result.size \geq k$ then
21:       $kScore$ = the score of the $k$th best result
22:       $bestRemaining$ = the estimated best score of the
          remaining results
23:       if $kScore$ is better than $bestRemaining$ then
24:          break
```

When ranking by result relevance, at each step we choose which node to expand based on path length, which is inherently a round-robin manner and all nodes will be expanded one by one. However, when ranking by result time or result duration, as can be seen from the example, this may not be the case. For two nodes matching k_1 and k_2, if many nodes on k_1's expansion paths have a later end time or longer

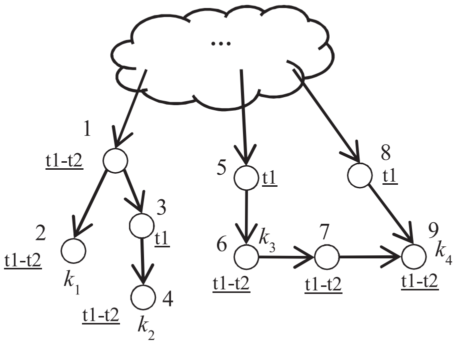


Fig. 6. Data graph for Examples 4.1 and 4.2.

duration than k_2, then we will expand k_1 many times before expanding k_2, which makes it difficult to find the results.

This problem arises because there exist conflicting goals when the results are ranked by time/duration. We have two goals in generating query results: generating results quickly; and generating results in the order of their ranking so we can stop earlier. These two goals can be achieved at the same time when ranking by relevance: each time we pick the best path iterators with the smallest distance to the source, and expand this node. However, these two goals become conflicting when ranking by time/duration. If we still choose the best path iterator with the smallest distance, then the result we found by expanding this best path iterator can be arbitrarily bad in terms of ranking function. On the other hand, if we choose the node whose current path has the latest end time/longest duration, then we may keep expanding one node and ignore the others, which is not helpful for finding results. The situation can be especially bad since unlike shortest path iterator, when ranking by result time or duration, expanding a node may not incur any penalty on the ranking function, thus resulting in a few nodes keep expanding for a long time in vein, as illustrated in the above example.

One solution is to do a round-robin on all best path iterators. However, this approach is problematic in that for each keyword, if we do *not* expand the best path iterator that currently has the highest score, then the score upper bound of unseen results won't change; and even if we find a result, it may not be a highly ranked result and we cannot output it, until the score upper bound decreases after we expand the iterator for this keyword with the best score.

Without prior knowledge, we assume the distance between a result root and each keyword match is the same. We propose that instead of doing a round-robin on all path iterators, we do a round robin on query keywords, so that it has maximal likelihood of finding a result. This idea is also used in BLINKS [7] for backward expansion. Then for each query keyword, we expand the iterator with the best ranking score, so that we can improve the score upper bound of unseen results and likely find highly ranked results. In this way, we balance the goal of finding results quickly and the goal of generating results in ranked order.

To do round-robin on keywords, we use k priority queues, one for all best path iterators for each query keyword. In each iteration, we choose a query keyword in a round-robin manner, then choose the NTD triplet corresponding to this keyword that has the latest end time, i.e., the NTD triplet on top of the corresponding priority queue.

Another question is, when ranking by result time or duration, since the latest end time/duration of a result is obtained by the *intersection of the time intervals* of all edges and nodes in the result, whether we can prune some time instants during the backward expansion for better efficiency. Let us look at an example.

Example 4.2. Considering Fig. 6 and query "k_3, k_4" rank by descending order of end time. We have two best path iterators from nodes 6 and 9. We first expand node 6 with time interval $[t_1, t_2]$ to node 5. Since node 5 has a latest end time t_1, further expansion along this direction can not increase the end time beyond t_1. At this point, can we avoid considering the time instant t_2 further when searching for results involving node 6 for improved efficiency? Unfortunately, we are not able to make such a pruning. In fact, the result composed by $6 \rightarrow 7 \rightarrow 9$ is indeed valid at t_2, which is obtained by the path iterator starting from node 9.

The above example shows as long as there exists one best path iterator whose next node is contained in time instant t, we must continue to consider the nodes in time instant t. Without auxiliary information such as indexes, the only pruning that can be done is to prune time instants that do not contain any match to a query keyword when we consider AND semantics of the query.

The worst-case time complexity of Algorithm 3 is $\mathcal{M} * PI + |R| * T$, where PI is complexity of the corresponding path iterator (Algorithms 1 and 2), and R is the number of candidate results generated.

4.2 Computing Top-k Results

To efficiently generate top-k results, we first generate k results according to the method discussed in Section 4. Then we estimate the score upper bound of the remaining results.

When ranking by relevance, the score of a result is the inverse of the total weights of its nodes and edges. Recall that each result is found by selecting an NTD triplet (n, T, d) from the top of the priority queue. Suppose this NTD triplet originates from a data node m_i that matches a query keyword k_i. If n is also visited by the expansion from all other query keywords, we generate a set of results rooted at n. Since these results use the NTD triplet (n, T, d), they must contain the shortest path from n to m_i in some time instants in T. Thus the total weight of this result tree is at least d. Furthermore, since d is the smallest distance of all unvisited NTD triplets, all unseen results must have a weight that is at least d. Therefore, we can use $1/d$ as the upper bound score of the unseen results. The following proposition shows that this upper bound is tight.

Proposition 4.1. *When ranking by relevance, suppose the next NTD triplet at the top of the priority queue has a distance of d. Then, $1/d$ is a tight score upper bound of results that have not yet been generated.*

Although $1/d$ is tight, in some cases we may need to generate a large number of results before we can stop, even if k is small, thus causing inefficiency. Thus we also propose an empirical score upper bound. If there are m keywords and the next NTD triplet has a distance of d, then we consider $1/md$ as the score upper bound of the unseen results. Intuitively, since there are m keywords, and each best path iterator has already been expanded to at least a distance of d, for *any result whose root node has not been visited* by any best path iterator, its score cannot be higher than $1/md$. That is, $1/md$ is in fact the upper bound score of all such results. Although $1/md$ may be lower than the scores of some unseen results,it cannot be more than m times worse than the score of any unseen results. Using $1/md$ instead of $1/d$ enables the program to stop much faster. As shown in experimental evaluation in Section 6, this empirical upper bound achieves high efficiency with a small compromise on the search quality.

Similarly, when results are ranked by time instant and duration, we have the following propositions:

Proposition 4.2. *When ranking by descending order of result end time, we take the NTD triplet at the top of each priority queue, and suppose their latest end time is t. Then, t is a tight score upper bound of results that have not yet been generated.*

Proposition 4.3. *When ranking by descending order of duration, we take the NTD triplet at the top of each priority queue, and suppose their longest duration is D. Then, D is a tight score upper bound of results that have not yet been generated.*

Similar as using $1/d$ as the score upper bound when ranking by relevance, although these two upper bounds are theoretically tight, they might be arbitrarily loose and cause inefficiency. Consider query "k_1, k_2" issued on the graph in Fig. 6. Suppose we use t as an upper bound when ranking by descending order of end time. After we find the result rooted at node 1, since this result's latest end time is t_1 and the estimated latest end time is t_2, we need to find more results. If all nodes in the cloud has the latest end time of t_2, we will have to finish exploring the entire graph before we can stop and output the result rooted at node 1.

Now we propose an empirical score upper bound. When ranking by end time, we check the NTD triplet at the top of each priority queue, and pick the NTD triplet with the smallest latest end time t' among these NTD triplets as the empirical score upper bound of unseen results. When ranking by duration, we check the NTD triplet at the top of each priority queue, and pick the NTD triplet with the smallest duration D' among these NTD triplets as the empirical score upper bound of unseen results. Although these empirical upper bounds may not be the true upper bound and can be smaller than the scores of some unseen results, they are indeed the score upper bound of those *results whose roots have not been visited by any best path iterator*.

5 Processing Temporal Predicates

Now we discuss how to support temporal predicates: precedes/follows, meets, overlaps, contains/contained by. The challenge is how to correctly prune nodes and edges that will not lead to qualified results during the backward expansion to achieve efficiency.

Precedes/Follows. Since "precedes" and "follows" are symmetric, we only discuss "precedes". A result satisfies predicate "precedes t_x" if and only if all nodes and edges of the result is valid in a time instant that precedes t_x (which may be optionally valid after t_x). Thus, during the backward expansion process, if a node or edge visited is not valid in any time instant that precedes t_x, it will be pruned.

Overlaps. Predicate "overlaps $t_x - t_y$" requires that the result time must overlap $t_x - t_y$. During the backward expansion process, we can check whether the valid time of each node and edge we visit satisfies "overlaps $t_x - t_y$", and prune the one that failed the condition.

Meets. Predicate "meets t_x" requires that a result must be valid in time instant t_x, but is either invalid before t_x, or is invalid after t_x. That is, t_x should be either start time or end time of the result time. Note that different from "precedes" and "overlaps", a result that satisfies "meets t_x" does not necessarily mean that its nodes and edges also satisfy "meets t_x".

Example 5.1. Consider a result consisting of two nodes n and n'. Suppose $val(n) = \{1, 3, 5, 7\}$ and $val(n') = \{2, 4, 5, 7\}$ and $val(n \to n') = \{5, 7\}$. Then the time interval of this result is $\{5, 7\}$, which satisfies predicate "meets t_5". However, neither n nor n' satisfies this predicate.

A necessary condition for a result to satisfy "meets t_x" is that all nodes and edges in the result must be valid in time instant t_x. Thus we can prune all nodes and edges that are not valid in t_x during the backward expansion. However, after we find a result, we need to check whether the result time satisfies "meets t_x".

Contains. Predicate "contains $t_x - t_y$" requires that the result time must contain $t_x - t_y$. During the backward expansion, we can prune the nodes and edges that failed this condition and the results generated this way are guaranteed to qualify the predicate. Predicate "contained by $t_x - t_y$" requires that the result time should be contained by $t_x - t_y$. We are not able to prune nodes and edges during backward expansion using this predicate, but have to check its satisfaction on the generated results.

6 Evaluation

6.1 Setup

Data Set. We use two data sets: DBLP (for publications up to October 2002) and Social Network, SNAP (http://snap.stanford.edu). DBLP data has 53 time instant, one for each year. If an article was published in year x, we consider it is valid from year x till now. It contains 3.8 million nodes and 4.0 million edges. The social network dataset is a graph in which each node represents a user, and each edge represents the interaction of two users. We randomly generate a set of time intervals for each edge from 100 time instants in the following way: we set the default probability that any two edges have at least one common time instant as 70 percent, and also vary this probability in evaluation. The valid time of each node is the union of the valid times of its edges. It contains 265 thousand nodes and 420 thousand edges. Both datasets assume the unit weight on edges and no weight on nodes.

These two datasets have different characteristics. First, they have different graph structures. In DBLP, there is a directed path from node "DBLP" to every other node, and there are citation paths. The structure of the Social Network data is more general in the sense that it does not have particular patterns. Thus given a set of query keyword matches it is more likely to find a rooted tree connecting them in the DBLP data. Second, these two datasets are different in temporal characteristics. DBLP is considered as an "insertion-only" (also known as "append-only") dataset, and thus the valid time of each node/edge is represented as a single interval. Whereas Social Network data has both insertions and deletions, and generally multiple time intervals are associated with a node/edge. Moreover, the datasets differ in graph connectivity with respect to time. We define *edge connectivity* as the probability that two adjacent edges have at least one common time instant. DBLP has 100 percent edge connectivity. Any subtree in the DBLP dataset is valid, as all nodes and edges are valid at the latest time instant ("current"). We evaluated Social Network data with various edge connectivity (Fig. 12).

Queries. On each dataset, we randomly generate 100 queries consisting of 2-4 keywords. On DBLP, each query has at least one keyword that matches node values, and other keywords may match either values or tag names. The Social Network data does not contain information of the keywords associated with each node. For each query keyword, we randomly pick 200-5,000 data node as matches, where every node has the same chance to be selected. All the reported times are the average of processing 100 queries unless specified otherwise.

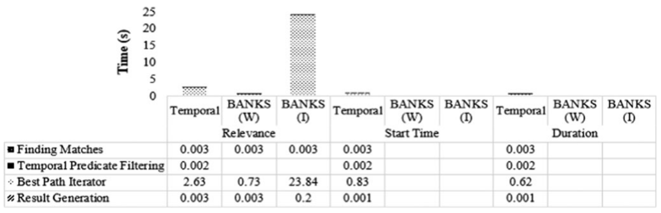
Comparison Systems. Since there is no existing work on supporting keyword search on temporal graphs, we use two baselines that adapt existing work of keyword search on regular graphs, BANKS [9], which exhibits surprisingly high search quality even compared with more recent search engines [13], to search temporal graphs. (1) We run BANKS's algorithm on data graph snapshot at distinct time instant to find keyword search results, referred to as BANKS (I). (2) we run the BANKS algorithm to process a query against the entire temporal graph, and then perform post-processing to filter out invalid results, referred to as BANKS (W). Recall that described in the original paper, BANKS maintains a fixed-size heap of generated results, sorted in the decreasing order of their ranking score, and outputs the result with the highest rank when the heap is full. This approach does not generate true top-k results nor have any quality guarantee. To be fair, we use the proposed empirical upper bounds (Section 4.2) for all three comparison systems by default, and compare the effect of different upper bounds in Section 6.3.

6.2 Efficiency

6.2.1 Different Ranking Functions

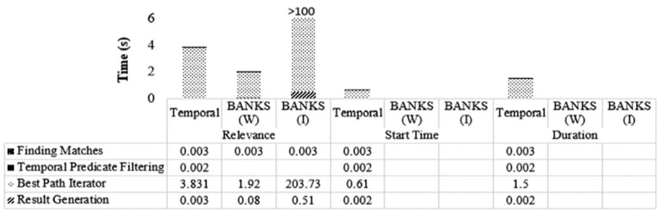
We test system efficiency for three different ranking functions (descending relevance, ascending start time, descending duration) on both data sets. The reported processing time is the average of generating top-20 results for 100 randomly generated test queries using the same empirical upper bound, as shown in Figs. 7 and 8. Besides reporting the total processing time, we breakdown the total time based on four steps in the processing: 1) Finding matches of query keywords using inverted indexes (Line 1-3 in Algorithm 3); 2) Temporal predicate filtering on keyword matches (Line 4-7); 3) Best path iterator from each keyword match for path expansion (Line 9-12); 4) Result generation based on the best paths, with validity checking (line 13-24).

First, let us compare the performance of our approach with BANKS(W) when ranking by relevance. On one hand, our method may take a longer time for best path iterator since the iterator (Algorithm 1) computes and records the shortest paths in all time instants between two nodes, while BANKS(W) only computes and records a single shortest path between two nodes. On the other hand, BANK(W)



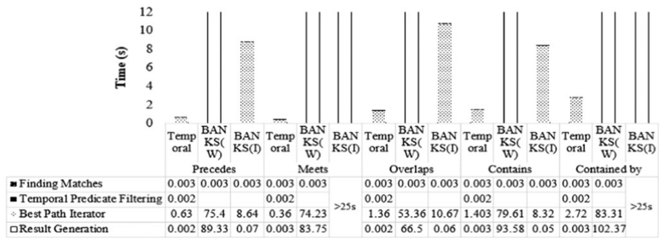
	Relevance			Start Time			Duration		
	Temporal	BANKS (W)	BANKS (I)	Temporal	BANKS (W)	BANKS (I)	Temporal	BANKS (W)	BANKS (I)
■ Finding Matches	0.003	0.003	0.003	0.003			0.003		
■ Temporal Predicate Filtering	0.002			0.002			0.002		
▨ Best Path Iterator	2.63	0.73	23.84	0.83			0.62		
▨ Result Generation	0.003	0.003	0.2	0.001			0.001		

Fig. 7. Efficiency wrt ranking functions (DBLP).



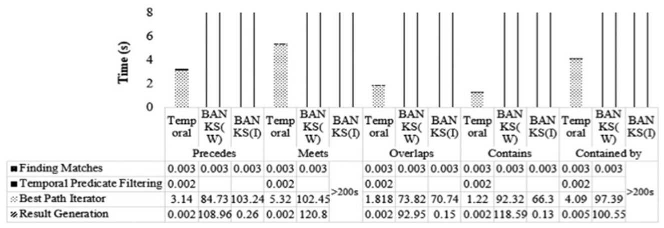
	Relevance			Start Time			Duration		
	Temporal	BANKS (W)	BANKS (I)	Temporal	BANKS (W)	BANKS (I)	Temporal	BANKS (W)	BANKS (I)
■ Finding Matches	0.003	0.003	0.003	0.003			0.003		
■ Temporal Predicate Filtering	0.002			0.002			0.002		
▨ Best Path Iterator	3.831	1.92	203.73	0.61			1.5		
▨ Result Generation	0.003	0.08	0.51	0.002			0.002		

Fig. 8. Efficiency wrt ranking functions (network).



	Precedes			Meets			Overlaps			Contains			Contained by		
	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)
■ Finding Matches	0.003	0.003	0.003	0.003	0.003		0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	
■ Temporal Predicate Filtering	0.002			0.002		>25s	0.002			0.002			0.002		>25s
▨ Best Path Iterator	0.63	75.4	8.64	0.36	74.23		1.36	53.36	10.67	1.403	79.61	8.32	2.72	83.31	
□ Result Generation	0.002	89.33	0.07	0.003	83.75		0.002	66.5	0.06	0.003	93.58	0.05	0.003	102.37	

Fig. 9. Efficiency wrt different predicates (DBLP).



	Precedes			Meets			Overlaps			Contains			Contained by		
	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)
■ Finding Matches	0.003	0.003	0.003	0.003	0.003		0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	
■ Temporal Predicate Filtering	0.002			0.002		>200s	0.002			0.002			0.002		>200s
▨ Best Path Iterator	3.14	84.73	103.24	5.32	102.45		1.818	73.82	70.74	1.22	92.32	66.3	4.09	97.39	
▨ Result Generation	0.002	108.96	0.26	0.002	120.8		0.002	92.95	0.15	0.002	118.59	0.13	0.005	100.55	

Fig. 10. Efficiency wrt different predicates (Network).

may generate many invalid results, which results in the overhead of result validation, and also the need of more path expansions in order to generate more results to obtain k valid ones. For DBLP data, all subtrees are valid since all the nodes and edges are valid at time instant "current". Thus all results generated by BANKS(W) are valid, resulting in negligible time for result generation, and overall fast processing time. On the other hand, for network data, BANKS(W) takes much more time on result generation and path expansion. This is because two adjacent edges only have 70 percent probability to share a common time instant, and thus contributing to a valid result. BANKS(W) expands 10,232 nodes and generates about 1,000 results, whereas our method visits 1,838 and generates less than 50 results. The combined effects on the experiments illustrated in Figs. 7 and 8 demonstrate that BANKS(W) is more efficient, while our processing time is still reasonable.

Note that the more invalid results generated, the more time BANKS(W) needs for path expansion and result generation. When we decrease the edge connectivity further, more top candidate results are likely to be invalid, BANKS (W) has performance degraded (Section 6.2.4). Also, BANKS(W) misses many relevant results on the Social Network data (Section 6.3).

Compared with BANKS(W) and our method, BANKS(I) spends much more time in both best path iterator and result generation because it has to search against the graph snapshots for every time instants (i.e., 53 graph snapshots for DBLP and 100 graph snapshots for SNAP), and generates results for each graph snapshot, resulting extremely long processing times. In fact, there is a lot of opportunities for exploiting shared processing. Using our approach, on average each node has 4.2 NTDs and 11.8 NTDs for DBLP and network data, respectively. This means, on average, there are only 4.2 unique shortest paths for a node in 53 time instants of the DBLP data. By computing and processing the same path only once, our approach is much faster than that of BANKS(I). Also note that the reported time of BANKS(I) is to find top-20 results in each valid time instant. Generating top-20 results among all time instants needs extra time.

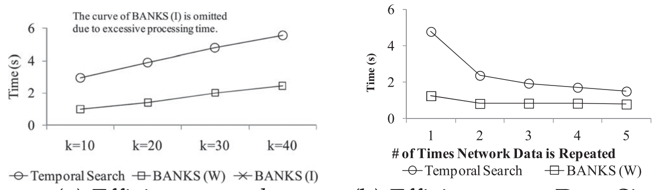
BANKS is not designed for ranking by temporal information. It generates results roughly in the order of their relevance. So it often has to enumerate all the results before identifying highly ranked ones in terms of temporal information, which may take hours and exhaust the memory. Their performance are not reported for ranking by start time and duration.

In contrast, our approach uses less time to generate top-k results when ranking by start time or duration, compared with ranking by relevance. This is because for relevance-based ranking every expansion of a best path iterator increases the distance by 1, and thus all iterators are expanded in a round robin fashion. When there are a large number of keyword matches, it takes a relatively long time to generate results. In contrast, when we rank the results by time instant or duration, different expansions have different impact on the ranking score, and thus often one iterator can be expanded multiple steps before switch to another iterator. Consequently, iterators of different query keyword matches are more likely to meet earlier.

We also test the effect of using the round robin approach discussed in Section 4. On the Social Network data, the round robin approach is 8 times faster than the counterpart (0.6s versus 4.7s per query) for ranking by ascending start time, and on the DBLP data, the no round robin approach exceeds our main memory size. We also check the quality of the results, which are the same for round robin approach and the counterpart.

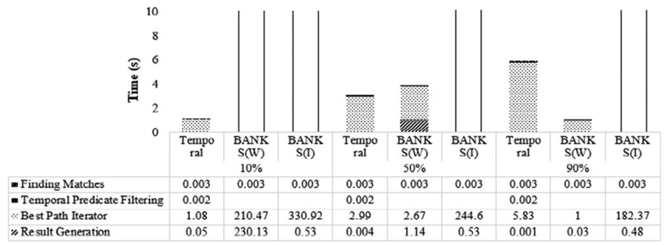
6.2.2 Temporal Predicates

To test the effects of temporal predicates, we generate 100 queries with each of the temporal predicate type, ranked by relevance, and let $k = 20$. The time instant sets associated with the predicates are randomly generated. As we can see from Figs. 9 and 10, when temporal predicates are used, our algorithm has a much better efficiency and is never slower than BANKS. Predicates enable us to prune more nodes during path expansion and thus usually fewer NTDs will be processed. For instance, for Network dataset, the average number of NTDs associated with each node in the priority queue is 3.50, 2.61, 1.83, 1.26, 3.53 for queries with predicates "meet", "precedes", "overlaps", "contains", "contained by", respectively. Consequently, in most cases it spends shorter time in best path iterator, and has better



(a) Efficiency wrt k (b) Efficiency wrt Data Size

Fig. 11. Scalability over k and dataset size.



	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)	Temporal	BANKS(W)	BANKS(I)
		10%			50%			90%	
■ Finding Matches	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003
■ Temporal Predicate Filtering	0.002			0.002			0.002		
⁘ Best Path Iterator	1.08	210.47	330.92	2.99	2.67	244.6	5.83	1	182.37
▨ Result Generation	0.05	230.13	0.53	0.004	1.14	0.53	0.001	0.03	0.48

Fig. 12. Efficiency breakdown wrt different connectivities.

efficiency. However, adding predicate "contained by" may make it harder to find valid results, resulting in a longer time to generate top-k results.

BANKS(W) will generate many more invalid results due to temporal predicates. Sometimes a predicate is so selective that the vast majority of generated candidate results do not satisfy the predicate, thus resulting in excessive long processing times. For example, for "precedes" in the Network data, BANKS(W) visites nearly 200 K nodes and generates 130 K results, including valid and invalid ones, in order to find top-20 results. Whereas our methods visits 1,653 unique nodes (15 K NTDs) and generates 52 results only. Also, compared with queries without temporal predicates, the bottleneck of BANKS(W) is shifted to result generation. This is because a single invalid node or path may result in many invalid results. The more invalid paths, the more time on result generation, as well as the overall processing time.

BANKS (I) becomes faster for queries with some temporal predicates compared with the counterpart without predicates, since it needs to process fewer graph snapshots. For instance, for "precedes" BANKS(I) only traverses the graph snapshots whose time instants are before the time specified in the query. The average number of graph traversals by BANKS(I) is 53.75 for queries with "precedes", and 34.64 for queries with "overlaps" and "contains". Note that, using this approach, BANKS(I) only finds top-20 results for every graph snapshot whose time satisfies the predicates, but does not report the total valid time of a result, nor find the top-20 results across all snapshots. However, to process queries with "contained by", BANKS(I) has to not only traverse snapshots in all time instants, but also merge the same results that appear in multiple time instants in order to find the valid time of result, and then test whether the result time is contained by the query specified time. This results in a longer processing time than processing corresponding queries without predicates. Similar situations exist for processing "meets".

6.2.3 Value of k

We test queries on the Social Network data set with varying values of k, ranging from 10 to 14, for ranking by relevance, as shown in Fig. 11a. As the processing time of BANKS (I) is too long, it is not shown in the figure. The processing times of BANKS(W) and ours increase linearly with k.

6.2.4 Edge Connectivity

Here we test processing efficiency on the Social Network data ranking by relevance, varying the edge connectivity, i.e., the probability that any two edges have at least one common time instant, from 10 to 90 percent, as shown in Fig. 12. As can be seen, our approach significantly outperforms BANKS (W) when the edge connectivity is no more than 50 percent. Note that the processing time of our approach does not necessarily increase or decrease with the probability. This is because with a higher connectivity, it is easier to find results; however, each node may have more NTD triplets for processing, making the total number of NTD triplets larger and the path expansion slower. In contrast, the processing time of BANKS(W) increases with decreasing edge connectivity since more invalid results are generated. As one invalid edge may invalidate many generated candidate results, the result generation is called many more times in order to generate one valid result. The processing times for BANKS(I) are much longer than the other two systems, and increase with decreasing connectivity. The lower connectivity, the longer the path that connects two nodes, and more path expansions need to be made.

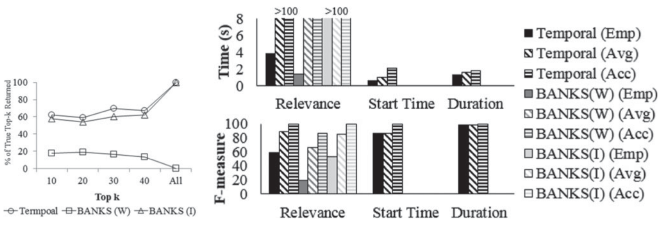
6.2.5 Data Size

We repeat the Network data from 1-5 times, and we randomly add 100 edges among different duplications. The processing time of our approach and BANKS(W) is shown in Fig. 11b. As we can see, when the data becomes bigger, the processing time may not necessarily increase or decrease. When the data is bigger, there are more keyword matches, thus more best path iterators need to be executed. However, when there are more keyword matches, it can be easier to connect matches to generate results and require fewer path expansions to generate top-k results.

6.3 Effectiveness

As anecdotal evidence of the applicability of our system on searching temporal graphs, we asked six students who did not participate in this project to issue three queries on the DBLP dataset. Among the 18 queries, we find 13 are temporal-related. All of them can be expressed in our query language, and 12 of them are correctly expressed by the users. This illustrates the need of temporal search, the expressiveness and user friendliness of our search syntax.

Since we adopt a commonly used result definition for keyword search on graphs and our contribution lies in searching temporal graphs, we use the result defined by BANKS on graph snapshots as ground truth. The result quality for five random queries ranking by relevance, on Social Network data, with k varied from 10 to ALL, using empirical upper bound is presented in Fig. 13a. As we can see, our results usually misses 20 to 30 percent of the results in top-40. The curve of BANKS(I) is similar with our method and is omitted here. BANKS (W) misses more results compared with our method. The larger the k, the higher percentage of results it misses. This is because the longer a path is, the less possible it is a valid path, and BANKS (W) is more likely to miss a larger result. When all results are needed, it returns less than 10 percent of the results in the ground truth.



(a) Quality (b) Performance wrt Upperbounds

Fig. 13. Result quality and the trade-off on efficiency.

Furthermore, we evaluate the result quality and the efficiency of different systems using three upper bounds for generating top-20 results on three ranking functions. 1) *Accurate* upper bounds are stated in Proposition 4.1, 4.2 and 4.3. 2) Discussed in Section 4.2 *Empirical* upper bound of ranking by relevance is $1/md$, where m is the number of keywords in the query, d is the smallest distance at the top of priority queues. The empirical upper bound of ranking by earliest start time is the latest start time at the top of priority queues. For ranking by longest duration, the empirical upper bound is the shortest duration at the top of priority queues. 3) *Average* upper bound is the average between the empirical and accurate upper bound.

As shown in Fig. 13, with the accurate upper bound, both our method and BANKS(I) find true top-k results. Using empirical upper bounds negatively impacts result quality, some true positives are skipped. Even using accurate upper bounds, BANKS(W) cannot achieve 100 percent F-measure as it processes queries on the temporal graph without considering time stamps. The shortest paths it computes may be invalid, and the results that include valid shortest paths between two nodes are missed. For the same reason, using other upper bounds is worse than the other methods.

At the same time, using empirical upper bound is more efficient than the other two upper bounds. The runtime differences of different upper bounds are larger in ranking by relevance. The reason is that ranking by relevance is highly sensitive to the upper bound: slightly increasing the upper bound can make the upper bound much more difficult to be beaten, and the end condition harder to trigger.

In summary, our method efficiently generates results with high quality. Compared with BANKS(W), ours is faster for queries have temporal predicates or ranking functions, or edge connectivity is relatively low. Ours significantly outperforms BANKS(W) in terms of result quality. Our method always has a better efficiency than BANKS(I) due to elimination of redundant computations. Also, our method has stable performance for diverse types of data and queries.

7 Related Work

Keyword Search on Graphs. For the work on processing keyword searches on graphs, the majority of them [8], [9], [10], [11], [12], [14], [15], [16], [17] adopt the minimal tree semantics. There are other result definitions for keyword search on graphs, such as subgraphs [18], individual database tuples [19], etc. However, none of existing work addresses the problem of searching temporal graphs.

Temporal Databases/XML/Graphs and Version Control Systems. Temporal databases extend the relational model by allowing a tuple to have a valid time and/or a transaction time. Several languages have been developed for querying temporal data, including TQUEL [5], TSQL2 [3] and SQL3 [6]. Structured query languages are also available for temporal semi-structured data such as XML [20], [21]. However, these approaches are not suitable for casual users to search temporal graphs due to complex query syntax and inefficiency for handling graphs.

Temporal graphs are also referred as time varying graphs, time evolving graphs and dynamic graphs. There are studies on specific query types on temporal graphs, such as single-node queries that ask for historical information of a node in a graph [22], reachability and subgraph matching queries [23] and centrality. Algorithms for these queries are not applicable for processing keyword queries except that [24] and [25] studied shortest path problem. [24] adopts a different data model, where a collection of snapshots are stored independently and clustered prior. They first run Dijkstra's algorithm on the representative of a cluster, then verify the result for snapshots, and if failed run Dijkstra's algorithm on individual snapshots directly. On the other hand, we extend Dijkstra's algorithm to process a compact data graph for all snapshots with timestamps and to support new ranking functions. [25] uses a similar data model as ours, with start and end times associated with each node and edge. It only supports querying shortest path between two nodes given a specific time range by adapting Dijkstra's algorithm to only process nodes and edges that satisfy the given time range, like the BANKS(I) implementation in our experiments. It does not compute the best path that is valid in at least one time instant, nor does it generate top-k results, or support temporal-based ranking.

Other types temporal graphs include road networks and communication networks, where edge connectivity and edge weight are dynamic during the processing [26], [27], whereas our work is based on existing graph snapshots. Data mining on system-generated temporal graphs representing low-level system entities and interactions has also been studied [28].

For large temporal graphs that don't fit in the memory, there are studies on graph partition [29]. The challenge is the tradeoff between temporal-based partitioning (storing consecutive snapshots together) and structure-based partitioning (storing closely connected nodes together).

There's a large body of work on temporal information retrieval (T-IR), which combine document relevance and document temporal relevance. Since T-IR focuses on unstructured documents rather than graphs, the techniques are fundamentally different than search temporal graphs.

8 Conclusions and Future Work

We initiate the study of the problem of searching temporal graphs. We propose a simple yet expressive keyword based query syntax that allows temporal information to be specified as either predicates or ranking factors. We propose a best path iterator, which finds the "best" paths between two data nodes in each time instant with respect to ranking functions where the rank of a path is monotonically non-increasing upon an edge expansion. Then we propose algorithms to efficiently evaluate this type of queries on a temporal graph to generate top-k results. The efficiency and effectiveness of the proposed approach are verified through extensive empirical studies.

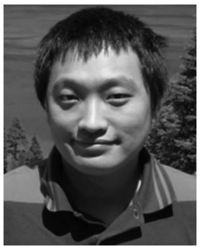
In future, we will study how to support other ranking functions, specifically, the ones that involve descending order of end time, ascending order of start time or duration. For these ranking functions, the ranking of a path may improve after we expand an edge, which is analogous to find shortest paths in a graph with negative edge weights. Therefore Dijkstra's algorithm cannot be adapted. One possibility is to adapt the Bellman-Ford algorithm to compute best paths when ranking by these factors. We leave it to future work. Further, we plan to support temporal queries on graphs in structured languages for experts who need high expressiveness (e.g., variable bindings).

Acknowledgments

This work is partially supported by the US National Science Foundation CAREER Award IIS-1322406, Google Research Award, and an endowment from the Leir Charitable Foundations. Yi Chen is the corresponding author.

References

[1] WebArchive project. [Online]. Available: http://oak.cs.ucla.edu/cho/research/archive.html
[2] VisTrails. [Online]. Available: http://vistrails.org
[3] C. S. Jensen, R. T. Snodgrass, and M. D. Soo, "The TSQL2 data model," in *The TSQL2 Temporal Query Language*. New York, NY, USA: Springer, 1995.
[4] F. Rizzolo and A. A. Vaisman, "Temporal XML: Modeling, indexing, and query processing," *VLDB J.*, vol. 17, no. 5, pp. 1179–1212, 2008.
[5] R. T. Snodgrass, "The temporal query language TQuel," *ACM Trans. Database Syst.*, vol. 12, no. 2, pp. 247–298, 1987.
[6] TSQL2 and SQL3 interactions. [Online]. Available: http://www.cs.arizona.edu/people/rts/sql3.html
[7] H. He, H. Wang, J. Yang, and P. S. Yu, "BLINKS: Ranked keyword searches on graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 305–316.
[8] B. Kimelfeld and Y. Sagiv, "Finding and approximating top-k answers in keyword proximity search," in *Proc. 25th ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2006, pp. 173–182.
[9] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword searching and browsing in databases using BANKS," in *Proc. 18th Int. Conf. Data Eng.*, 2002, pp. 431–440.
[10] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar, "Bidirectional expansion for keyword search on graph databases," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 505–516.
[11] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin, "Finding top-k min-cost connected trees in databases," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 836–845.
[12] K. Golenberg, B. Kimelfeld, and Y. Sagiv, "Keyword proximity search in complex data graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 927–940.
[13] J. Coffman and A. C. Weaver, "A framework for evaluating database keyword search strategies," in *Proc. 19th ACM Int. Conf. Inf. Knowl. Manage.*, 2010, pp. 729–738.
[14] Y. Luo, X. Lin, W. Wang, and X. Zhou, "SPARK: Top-k keyword query in relational databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 115–126.
[15] V. Hristidis, L. Gravano, and Y. Papakonstantinou, "Efficient IR-style keyword search over relational databases," in *Proc. 29th Int. Conf. Very Large Data Bases*, 2003, pp. 850–861.
[16] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano, "Efficient keyword search across heterogeneous relational databases," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 346–355.
[17] L. Zhang, T. Tran, and A. Rettinger, "Probabilistic query rewriting for efficient and effective keyword search on graph data," *Proc. VLDB Endowment*, vol. 6, pp. 1642–1653, 2014.
[18] L. Qin, J. X. Yu, L. Chang, and Y. Tao, "Querying communities in relational databases," in *Proc. IEEE 25th Int. Conf. Data Eng.*, 2009, pp. 724–735.
[19] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "ObjectRank: Authority-based keyword search in databases," in *Proc. 30th Int. Conf. Very Large Data Bases*, 2004, pp. 564–575.
[20] R. Bin-Thalab and N. El-Tazi, "TOIX: Temporal object indexing for XML documents," in *Proc. Int. Conf. Database Expert Syst. Appl.*, 2015, pp. 235–249.
[21] R. Bin-Thalab, N. El-Tazi, and M. E. El-Sharkawi, "TMIX: Temporal model for indexing XML documents," in *Proc. ACS Int. Conf. Comput. Syst. Appl.*, 2013, pp. 1–8.
[22] G. Koloniari, D. Souravlias, and E. Pitoura, "On graph deltas for historical queries," *CoRR*, vol. abs/1302.5549, (2013). [Online]. Available: http://arxiv.org/abs/1302.5549
[23] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller, "Towards efficient query processing on massive time-evolving graphs," in *Proc. 8th Int. Conf. Collaborative Comput.: Netw. Appl. Worksharing*, 2012, pp. 567–574.
[24] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 726–737, 2011.
[25] W. Huo and V. J. Tsotras, "Efficient temporal shortest path queries on evolving social graphs," in *Proc. 26th Int. Conf. Sci. Statist. Database Manage.*, 2014, pp. 38:1–38:4.
[26] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *Proc. 11th Int. Conf. Extending Database Technol.: Advances Database Technol.*, 2008, pp. 205–216.
[27] D. Kempe, J. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," in *Proc. 32nd Annu. ACM Symp. Theory Comput.*, 2000, pp. 504–513. [Online]. Available: http://doi.acm.org/10.1145/335305.335364
[28] B. Zong, et al., "Behavior query discovery in system-generated temporal graphs," *Proc. VLDB Endowment*, vol. 9, no. 4, pp. 240–251, 2015.
[29] M. Steinbauer and G. Anderst-Kotsis, "DynamoGraph: A distributed system for large-scale, temporal graph processing, its implementation and first observations," in *Proc. 25th Int. Conf. Companion World Wide Web*, 2016, pp. 861–866.



Ziyang Liu received the PhD degree in computer science from Arizona State University, in 2011, where his research primarily concerned supporting keyword search on structured data. He is currently with the data infrastructure team at Facebook. Prior to that, he worked at LinkedIn and NEC Labs America.

Chong Wang received the bachelor's degree from the Nanjing University of Posts and Telecommunications, Nanjing, China, in 2010. He is currently working toward the PhD degree in the Department of Information Systems, NJIT. His research interests include machine learning, text mining, and computational advertising.

Yi Chen received the PhD degree from the University of Pennsylvania, in 2005. Currently, she is an associate professor and Leir chair in the Martin Tuchman School of Management, with a joint appointment in the College of Computing Sciences, New Jersey Institute of Technology. Her research interests focus on data management technologies and their applications in healthcare, business, and Web. She is a member of the IEEE.