Efficient Temporal Shortest Path Queries on Evolving Social Graphs

Wenyu Huo
Department of CSE
University of California, Riverside
Riverside, CA, 92521, USA
whuo@cs.ucr.edu

Vassilis J. Tsotras
Department of CSE
University of California, Riverside
Riverside, CA, 92521, USA
tsotras@cs.ucr.edu

ABSTRACT
Graph-like data appears in many applications, such as social networks, internet hyperlinks, roadmaps, etc. and in most cases, graphs are dynamic, evolving through time. In this work, we study the problem of efficient shortest-path query evaluation on evolving social graphs. Our shortest-path queries are "temporal": they can refer to any time-point or time-interval in the graph's evolution, and corresponding valid answers should be returned. To efficiently support this type of temporal query, we extend the traditional Dijkstra's algorithm to compute shortest-path distance(s) for a time-point or a time-interval. To speed up query processing, we explore preprocessing index techniques such as Contraction Hierarchies (CH). Moreover, we examine how to maintain the evolving graph along with the index by utilizing temporal partition strategies. Experimental evaluations on real world datasets and large synthetic datasets demonstrate the feasibility and scalability of our proposed efficient techniques and optimizations.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *query processing*

General Terms
Algorithms, Measurement, Performance

Keywords
Evolving Graph, Temporal Query, Shortest Path

1. INTRODUCTION
Graphs have been used as a general data structure to model numerous modern applications, such as social networks, internet hyperlinks, roadmaps, bioinformatics, etc. For example, in a social network like Facebook, registered users can be considered as vertices with edges representing friendships between them. In a dynamic world, users and friendships are continuously evolving with time. In September 2005, LinkedIn had about only 1 million users; by April 18, 2014, the number of LinkedIn members had increased to over 300 million. This dynamically changing environment holds very useful information; but at the same time it also creates many challenges, including: how to store the evolution of large-scale graphs and how to utilize the evolving graph data during query evaluation.

The shortest-path query is among the fundamental operations on graph data: typically the shortest-path distance is important in measuring "closeness" between nodes. In social networks, users may be comfortable with adding close users as their friends, and users may be interested in finding contents from users that are close to them in the social graph. Computing shortest-path distances efficiently is thus crucial for a variety of applications.

Different from traditional studies of shortest-path queries on a single graph, our main objective is to efficiently answer *temporal shortest-path queries* within the social graph's evolving history. Such temporal queries can be viewed as being issued on certain historical graph snapshot(s). This type of temporal query is not only essential for searching and retrieving histories, but also useful for trend analysis. Temporal shortest-path queries in a social network can discover how close two given users were in the past and how their closeness evolved over time.

Graph data management and query evaluation are thus important foundations for these novel applications. However, typically, even a single snapshot graph is already very large; maintaining the evolving graph history has much greater volume in data storage and brings more challenges in query processing.

Plenty of research work has studied efficient shortest-path querying of large road network graph data. To improve query times, several preprocessing indexes have been proposed; a survey of route planning is provided by [3]. Nearly all of these techniques rely on some variant of the Dijkstra's algorithm [4]. Hierarchical methods such as Contraction Hierarchies (CH) [5], seek to order the nodes and/or edges within the graph into hierarchically nested levels. However, most previous works focus only on a single (i.e. non-temporal) graph snapshot.

Recently [10] addressed the problem of evaluating historical queries on social graphs. Its storage model maintains the current graph and deltas to previous time snapshots; as a result, the first step of evaluating a historical shortest-path query is to reconstruct the corresponding snapshot or snapshots that relate to the query's temporal predicate. However such a reconstruction phase can be costly; this is an issue also in other related works [8, 9].

Another storage approach was proposed in [11], namely, the historical evolving graph sequence (EGS). Various snapshots and deltas are explicitly stored, but in addition, temporally close snapshots are clustered together. Graph-based queries, like

SSDBM'14, June 30 – July 02, 2014, Aalborg, Denmark.

http://dx.doi.org/10.1145/2618243.2618282

shortest-path and closeness centrality, are answered for the whole graph history; this is done efficiently with the help of a Find-Verify-Fix (FVF) framework. Nevertheless, if the query asks for a single time point or a time interval (i.e., not the whole history), this approach will be slow.

Our work is distinguished from previous studies in various ways: (i) In order to reduce storage overhead and support time-interval querying efficiently, we store the historical evolution in one simple, "integrated" temporal graph instead of a sequence of snapshots or clusters and their deltas. (ii) We explore preprocessing index techniques for the temporal evolving graph query processing, which are very effective and efficient. (iii) We explore further enhancements like temporal partitioning.

The rest of this paper is organized as follows. In section 2, the temporal evolving graph model is described along with temporal shortest-path querying definitions. In section 3, the fundamental solutions are explored as extensions of Dijkstra's algorithms, while section 4 describes speedup techniques using preprocessing indexes. Section 5 discusses further optimizations, in particular how temporal partitioning affects the processing of temporal queries. Section 6 presents our experimental analysis and section 7 concludes the paper with future work.

2. TEMPORALLY EVOLVING GRAPH

A single static graph, either directed or undirected, can be modeled as $G = (V, E)$, where V is the set of nodes and E is the set of edges. If G is a weighted graph, an edge is represented as a triplet $<u, v, w>$: i.e., this edge is from node u to node v with weight w. If the graph evolves with time, a different graph snapshot exists logically at each time. For example, as shown in Figure 1(a), the graph G_1 at time t_1 had six nodes and six directed edges. From then, until the latest time t_5, there are five graph snapshots with four updates. Each update may contain multiple operations including: node insertion, node deletion, edge insertion, edge deletion, and edge weight adjustment. To maintain the graph evolution in a space-efficient way, we use the Temporally Evolving Graph (TEG), which is also a kind of "graph", allowing us to explore/adapt efficient indexing techniques.

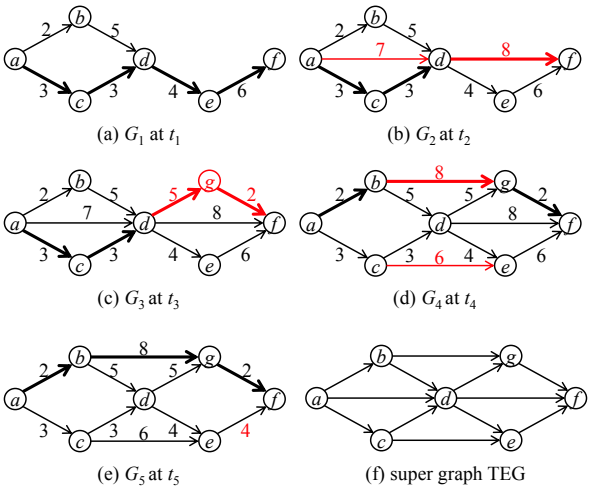

Figure 1. Example of a temporally evolving graph and its TEG (shortest path from *a* to *f* are marked in thick lines)

In a TEG, we add two temporal attributes: start time t_s and end time t_e, to restrict the nodes and edges. Each node is represented in a triplet as $<v, t_s(v), t_e(v)>$ which implies that node v appears in the graph snapshots during the time interval $[t_s(v), t_e(v))$. When a node is first created, its t_e is initialized with the special symbol "***now***", marking a currently existing ('alive') node ([2]). Each edge e in a TEG is represented as $<u, v, w, t_s(e), t_e(e)>$ noting that this edge runs from node u to node v with weight w during the time interval $[t_s(e), t_e(e))$. Figure 1(f) shows the TEG of the graph evolution.

Notice that in a TEG there may be parallel edges connecting two nodes (such edges, however, have non-intersecting time intervals). For example, in Figure 1(f), between nodes *e* and nodes *f*, there are two separate parallel edges $<e, f, 6, t_1, t_5>$ and $<e, f, 4, t_5, now>$. However, between the same pair of nodes, there is only one unique valid edge at any given time point.

For a temporal shortest-path query, in addition to the given source node v_s and target node v_t, a time constraint is required, such as a time-point t_q, or a time-interval $\Gamma_q = [ts_q, te_q)$, which restricts the candidate nodes and edges within a specific part of the whole temporal graph *TEG*. Let $\Gamma(v)$ represent the time interval for node v: $\Gamma(v) = [t_s(v), t_e(v))$; and let $\Gamma(e)$ represent the time interval for edge e: $\Gamma(e) = [t_s(e), t_e(e))$.

The sub-graph of a temporal evolving graph *TEG* for a time-point constraint t_q is defined as ***sub-TEG***$(TEG, t_q) = (sub\text{-}V, sub\text{-}E)$, where $\forall v$ satisfying $(v \in V \wedge t_s(v) \le t_q \square t_e(v) > t_q) : v \in sub\text{-}V$, and $\forall e$ satisfying $(e \in E \square t_s(e) \le t_q \square t_e(e) > t_q) : e \in sub\text{-}E$. It represents the graph snapshot at time point t_q. Similarly, the sub-graph of a temporal evolving graph *TEG* for a time-interval constraint Γ_q is defined as ***sub-TEG***$(TEG, \Gamma_q) = (sub\text{-}V, sub\text{-}E)$, where $\forall v$ $(v \in V \square \Gamma(v) \cap \Gamma \neq \emptyset) : v \in sub\text{-}V$, and $\forall e$ satisfying $(e \in E \square \Gamma(e) \cap \Gamma_q \neq \emptyset) : e \in sub\text{-}E$. It represents the set of graph snapshots during time interval Γ_q.

Definition 1. A **T**ime **P**oint **S**hortest **P**ath query **TPSP**(TEG, t_q, v_s, v_t) returns the distance of a path $p(e_1,\ldots,e_k)$ as a sequence of edges for query time t_q, which is the shortest-path from source node v_s to target node v_t, (both v_s and v_t are temporally valid at query time t_q), while all edges in p are valid at query time t_q. In other words, path p satisfies: $e_1 = (v_s, x) \square e_k = (y, v_t) \square \forall e_i \in p : (t_s(e_i) \le t_q \square t_e(e_i) > t_q)$; and $\forall p' \subseteq$ sub-TEG(TEG, t_q) from v_s to v_t : dist$(p') \ge$ dist(p). Here dist(p) means the distance of the path p as the sum of the all edges' weights in p at time t_q.

For any time point shortest-path query, since the corresponding historical graph snapshot is unique, there is a single distance returned. However, for the time-interval query, the distance from source to target may change within this time interval. We thus define the time interval "all" queries for shortest paths in a TEG as returning all the shortest distances during the time interval.

Definition 2. A **T**ime **I**nterval **S**hortest **P**ath "**all**" query **TISP-all**$(TEG, \Gamma_q, v_s, v_t)$ returns a set of distances for paths $P = \{p_1,\ldots,p_m\}$ which contains all the shortest distance paths from source node v_s to target node v_t during the query time interval Γ_q. Each path $p_i \in$ P is associated with a time interval Γ_{pi} and there is no other path shorter than p_i from v_s to v_t during time interval Γ_{pi}.

3. FUNDAMENTAL SOLUTION

For shortest-path search in graphs, especially social network graphs, a straightforward method is Breadth-First-Search (BFS). The prerequisite of BFS is that the edges of the graph are un-weighted or unit-weighted. However, large social networks (such as Facebook and LinkedIn) are using more complicated models to compute the weight between two connected nodes. Thus here we focus on Dijkstra's algorithm [4] as the classic solution for the point-to-point shortest path query.

For the time point shortest path (TPSP) query on a temporal evolving graph, we use the TPSP-Dijkstra algorithm which is an adaptation of Dijkstra using a priority queue; in addition, we need

to verify if an edge e's time interval $[t_s(e), t_e(e))$ is valid at time t_q (i.e., $t_s(e) \le t_q < t_e(e)$) before relaxing this edge in the search.

For the time interval shortest path "all" (TISP-all) query, the naïve method is to perform the TPSP-Dijkstra for all time points within the query interval Γ_q. Therefore, for a query time interval with k time instants, this approach would run TPSP-Dijkstra k times, which will not be efficient. An improved approach is to run an extended Dijkstra's algorithm once and return all the qualified answers for the TISP-all query.

The TISP-all-Dijkstra's algorithm is different from the naive TPSP in three aspects. First, the algorithm cannot stop until the confirmed shortest path distance covers the whole query interval. As a result, we need to record the parts that are done as well as not done. Second, the distance from the source to a given node v within the query interval is not a single value d, but a set of values D with different time aspects. Last, updating the distance set D and priority queue PQ is more complex.

More details about the description and analysis of both TPSP-Dijkstra and TISP-all-Dijkstra algorithms can be found in [7].

4. SPEED-UP TECHNIQUES

We further propose speed-up techniques for the temporal shortest path algorithms based on our basic solutions by analyzing the utilization of preprocessing indexes, such as Contraction Hierarchies (CH) [5]. The effectiveness of the CH search technique comes from the use of newly-added shortcut edges, which allow Dijkstra's search to effectively bypass irrelevant nodes during the search, without invalidating correctness.

Certain absolute ordering of the vertices is established in the graph, according to some notion of relative importance; then the CH is constructed by "contracting" one vertex at a time in increasing order. Once all necessary shortcuts E' are added to the graph G for a given ordering, shortest path queries may then be carried out using a bidirectional Dijkstra search variant which performs a simultaneous forward search and backward search.

Incorporating the contraction hierarchies into our temporal evolving graph is a non-trivial task. Since the CH indexing adds shortcuts on the original graph, in a form of extra edges, we need to extend the contraction hierarchies by adding temporal information on the shortcut edges as well.

The vertex contracting task can be achieved efficiently with the help of our TISP-all-Dijkstra query processing algorithm without a specified target, and all the "witness" shortest-path distances are stored.

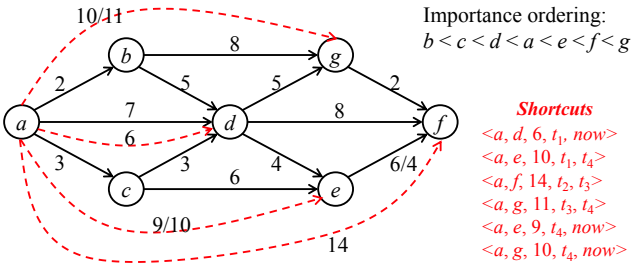

Figure 2. CH on the example temporally evolving graph

The CH on TEG of our running example in Figure 1 is shown in Figure 2, based on the following importance ordering: $b < c < d < a < e < f < g$. Since the shortcuts are also certain types of "edges" in the pre-processed graph, the shortcuts of CH on TEG have two new features inherited from the properties of TEG's edges: i) each shortcut has a time interval validity $[t_s, t_e)$, and ii) parallel shortcuts are supported as well. There are two pairs of parallel shortcuts in our TEG-CH example: $<a, e, 10, t_1, t_4>$ / $<a, e, 9, t_4, now>$ and $<a, g, 11, t_3, t_4>$ / $<a, g, 10, t_4, now>$.

Once the construction phase of CH on TEG is finished, the shortest path queries can be carried out. The algorithm employed on the corresponding CH for TEG is similar to the bidirectional Dijkstra's algorithm on CH for traditional shortest path queries. For each upward and downward search, our proposed TPSP-Dijkstra and TISP-Dijkstra algorithms can be utilized. For example, consider the time-interval "all" query of $[t_2, t_5)$ from a to f. The upward search from a extracts the following distances in order: $<e, 9, t_4, t_5>$, $<e, 10, t_2, t_4>$, $<g, 10, t_4, t_5>$, $<g, 11, t_3, t_4>$, and $<f, 14, t_2, t_3>$, while the downward search from f only extracts the distance $<g, 2, t_3, t_5>$. So the "all" shortest path distances from a to f with $[t_2, t_5)$ are: $<14, t_2, t_3>$ $(a \rightarrow f)$, $<13, t_3, t_4>$ $(a \rightarrow g \rightarrow f)$, and $<12, t_4, t_5>$ $(a \rightarrow g \rightarrow f)$.

5. TEMPORAL PARTITIONING

For historical evolving graphs, one could choose the Graph Sequence (GS) to store all the graph snapshots for each time instance. The GS model is optimal for time-point querying, but it has huge storage overhead and is not efficient for time-interval querying. On the other hand, the TEG model is optimal in space (linear) and efficient for time-interval querying (especially large intervals); however, its time-point querying performance is downgraded due to skipping plenty of temporally invalid edges. A trade-off solution is to make temporal partitions for one TEG along the time axis. For example, if the whole TEG has n time instances, and we create a partition for each m time instances, then we will get $\lceil n/m \rceil$ partitions.

Temporal partitioning brings duplicates between the partitions: an edge with a long time interval will be distributed to all the partitions its lifetime overlaps. However, temporal partitioning reduces the size of the individual queried TEG(s). The temporal shortest path queries (either time-point or time-interval) are issued on the corresponding TEG partition(s) instead of the original "super" TEG, speeding up the query process.

Temporal partitioning was first introduced in temporal databases as an efficient way to optimize temporal queries [12]. Various temporal partitioning methods have also been proposed in [2, 6], but in a different context (temporal keyword queries over versioned textual documents). A simple and easy way to implement partitioning is the fixed-time-window (**fix**): each partition has a time-window with a fixed length. The advantage of the fixed strategy is obvious: for a time-interval query, the number of accessed partitions is bounded. Assuming a **fix** partition with time-window length m, for any time-interval query whose interval length is l, we need to access at least $\lceil l/m \rceil$ and at most $\lceil l/m \rceil+1$ partitions. Other partitioning techniques considered appear in [7].

Temporal partitioning can also take advantage of parallelism in a map-reduce environment. If a time-interval query needs to access multiple partitions, we can map them into multiple nodes, one partition for each node, and they can be processed in parallel. The overall query time would then be the max time spent on a node, plus the results merging time, thus improving query performance.

6. EXPERIMENTAL EVALUATIONS

Firstly, we used social network graphs from YouTube, as provided by [1]. The properties of the real datasets are: 165 continuous daily snapshot graphs from 2007, 1,402,949 nodes / 6,783,917 edges in the first snapshot, and 3,218,658 nodes / 18,524,095 edges in the last snapshot. The experiments were

performed on a single Intel® Core™ i5-2400S CPU at 2.50GHz with 8 GB RAM.

Time-Point Shortest-Path Query. For time-point queries, we get the average query performance time by running the shortest-path algorithms on every dataset day. For each tested day, we choose 1000 uniformly random *s-t* pairs. The results are reported in Table 1. Contraction Hierarchy indexing gains a drastic better querying performance than all Breath First Search (BFS), TPSP-Dijkstra's algorithm and its bidirectional search version.

Table 1: Time-point querying on YouTube data

YouTube	BFS	TPSP	Bidir	CH
Preprocessing	0	0	0	3h47m
Extra Space (MB)	0	0	0	54.1
Query Time (ms)	1976	2159	1283	340

Table 2: Performance of time-interval querying

"all" (s)	Multi-TPSP	1-TISP	Bidir	CH
5-day	10.8	6.3	3.9	1.1
15-day	32.4	15.1	9.4	3.0
25-day	54.1	25.9	16.7	5.0

Time-Interval Shortest-Path Query. For time-interval querying, we tested different query interval lengths of 5-day (3% of graph lifetime), 15-day (9% of graph lifetime), and 25-day (15% of graph lifetime). For each length, we randomly chose 100 time-intervals within the dataset lifetime. The querying performance time is also averaged by 500 uniformly random *s-t* pairs for each query time-interval. The results (time measured in seconds) on the YouTube dataset are reported in Table 2. Clearly, the one-time run of TISP-Dijkstra's algorithm is much better than multiple runs of TPSP-Dijkstra's algorithm. Further, the average performance can be substantially improved by using CH indexing.

Temporal Partitioning. To further demonstrate the scalability of our proposed, we generated a large synthetic graph data. Its life time is from 2013.1.1 to 2013.7.20, with over 200 million nodes and over 10 billion edges. We also considered how temporal partitioning could further improve the querying performance through parallelism, by distributing partitions among multiple machines. We stored the large synthetic graph on the UCR-DB-cluster, which has 10 machine nodes; each machine node has a quad core CPU with 16G RAM and 6TB disk.

We use the fixed-time-window split strategy to partition both the graph data and its corresponding CH index structure. Two different partition lengths of the fixed-time-window are used: 15-day (CH-15) and 30-day (CH-30), which result in 14 and 7 partitions respectively.

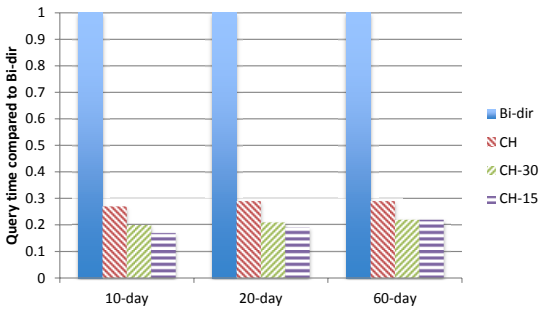

Figure 3. Time-interval querying time compared to Bi-dir

Here, we only show the result for the time interval querying, which can better utilize the cluster system: while accessing multiple partitions for a single query, they could be processed in parallel. As shown in Figure 3, for both small and large time intervals, the improvements for the CH querying performance triggered by temporal partitioning are significant (CH-15 showed between 30% and 60% improvement over CH).

7. CONCLUSION

In this work, we considered how to answer temporal shortest-path distance queries on evolving social graphs. We extended the traditional Dijkstra's algorithm so as to efficiently address both time-point and time-interval shortest-path queries. Moreover, we investigated how to incorporate preprocessing index structures such as CH to speed-up query processing. To analyze trade-offs and explore further optimizations, we considered temporal partitioning technology.

We demonstrated the efficiency and scalability of our algorithms and optimizations, by experimental evaluations on both real-world social-network datasets and large synthetic datasets. More experiments and results can be found in [7].

In future work, we would explore a cost model which based on graph properties and query workload would decide on the best temporal partitioning. Moreover, we would like to investigate more complex graph-based temporal queries.

8. ACKNOWLEDGEMENTS

We would like to thank Michael Rice from ESRI for the many comments and suggestions in improving this paper. This work was partially supported by NSF grants IIS-0910859 and IIS-1161997.

9. REFERENCES

[1] http://socialnetworks.mpi-sws.org

[2] K. Berberich, S. Bedathur, T. Neumann, G. Weikum. A Time Machine for Text Search. In SIGIR, 2007.

[3] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In Algorithmics of Large and Complex Networks, 2009.

[4] E. W. Dijskstra. A Note on Two Problems in Connexion with Graphs. Numerische Mathematik, 1:269-271, 1959.

[5] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchial Routing in Road Networks. In WEA, pages 319-333, 2008.

[6] W. Huo and V. J. Tsotras. A Comparison of Top-k Temporal Keyword Querying over Versioned Text Collections. In DEXA, 2012.

[7] W. Huo. Query Processing on Temporally Evolving Social Data. PhD dissertation, University of California, Riverside, 2013.

[8] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In ICDE, 2013.

[9] G. Koloniari, D. Souravlias, and E. Pitoura. On Graph Deltas for Historical Queries. In Workshop on Online Social Systems (WOSS), 2012.

[10] G. Koloniari and K. Stefanidis. Social Search Queries in Time. In PersDB, 2013.

[11] C. Ren, E. Lo, E. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. In VLDB, 2011.

[12] R.T. Snodgrass, I. Ahn. Partitioned storage for temporal databases. Information Systems, 13(4), pp: 369-391, 1988.