On Querying Historical Evolving Graph Sequences

Chenghui Ren[†] Eric Lo[‡] Ben Kao[†] Xinjie Zhu[†] Reynold Cheng[†]
[†]The University of Hong Kong [‡]Hong Kong Polytechnic University
[†]{chren,kao,xjzhu,ckcheng}@cs.hku.hk [‡]ericlo@comp.polyu.edu.hk

ABSTRACT

In many applications, information is best represented as graphs. In a dynamic world, information changes and so the graphs representing the information evolve with time. We propose that historical graph-structured data be maintained for analytical processing. We call a historical evolving graph sequence an EGS. We observe that in many applications, graphs of an EGS are large and numerous, and they often exhibit much redundancy among them. We study the problem of efficient query processing on an EGS and put forward a solution framework called FVF. Through extensive experiments on both real and synthetic datasets, we show that our FVF framework is highly efficient in EGS query processing.

1. INTRODUCTION

Graphs are a pervasive structure that is used to model the state of the world in many real-life applications. For example, users and their relationships in a social network (such as Facebook and Flickr) can be modeled as a graph, with vertices representing users and edges representing friendships among users. In a dynamic world, such relationships are continuously evolving. For example, users join Facebook and friendships are established. A graph that models the world can thus only capture the world's state at a particular instant, or just a "snapshot" of the world. To fully capture the dynamics of the world, we propose that a sizable collection of snapshots should be used. For example, snapshots of the Facebook graph should be taken periodically, forming a sequence of snapshot graphs. We call such a sequence an *Evolving Graph Sequence* or EGS for short. One can interrogate an EGS with many interesting graph-based queries that characterize the world the snapshot graphs depict. For example, given two vertices u and v, "What is the most popular shortest path that connects u to v among all the snapshots in an EGS?" "How does the centrality of a node evolve in an EGS?" Queries of this sort, which are hardly meaningful when applied to a single graph, provide valuable insights into the dynamic world.

Many traditional studies on graph queries and algorithms, such as shortest-path and reachability, focus on answering queries efficiently on a single graph. Different from those previous works, our goal is to efficiently answer queries on a large sequence of *related graphs* given in an EGS. We are particularly interested in those applications in which the snapshot graphs are *large*, *numerous*, and *gradually evolving*. The first two properties call for highly efficient algorithms to deal with the large amount of graph data. The third property implies that successive snapshots in a graph sequence are likely similar to each other. This property allows techniques that exploit redundancies among similar snapshots to be developed to achieve high efficiency.

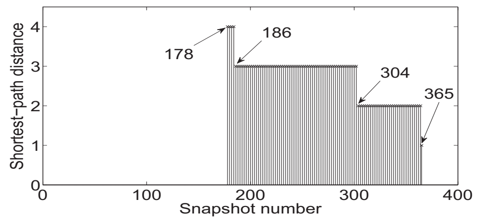


Figure 1: Distance between two users in FACEBOOK friendship graph over a 365-day EGS.

To further illustrate how real-world information can be modeled as an EGS, consider social networks, such as Facebook, Youtube and Flickr. In a social network, people connect to and interact with others to share their interests and experience. Social network analysis (SNA) [14] is a research area whose aims are to capture the various interactions among users and to understand users' behavior. In most studies, the interactions among users are modeled as graphs. For example, an edge between two (user) vertices could model a simple and somewhat static friendship relationship; or an edge can be used to model a more dynamic interaction, such as whether two users have communicated within a certain period of time, or whether they have written about the same topic. Numerous statistical measures have been defined and studied on social network graphs. These include global (graph-based) measures such as the diameters, radii, and degree distributions of the graphs; and local (vertex-based) measures such as centrality. To understand the dynamics of social networks, daily snapshots of various social networks have been collected [11, 13]. For example, an EGS of a few hundred daily snapshots of Facebook's New Orleans regional network is publicly available [13]. The graphs are big (e.g., the last snapshot contains about 60,000 vertices and about 900,000 edges), numerous (hundreds and many more if the collection were continued), and gradually evolving (successive snapshots are very similar, sharing more than 99% of their edges). These snapshot graphs facilitate many interesting studies, particularly *trend analysis*, which discovers pattern of change over time. For example, Figure 1 plots the *shortest-path distances* between two Facebook users over a one-

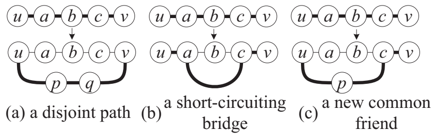


Figure 2: How did u and v get closer?

year period (365 snapshots). This plot gives interesting insights to how "friendships" are established in the social network — we see that the users were disconnected until snapshot #178. Since then, they got closer to each other until they finally became friends at snapshot #365. This plot reveals a few *key moments* in their friendship development (at snapshots #186, #304, #365). By analyzing the changes in their shortest paths at those key moments and the snapshot graphs surrounding those moments, we can answer some interesting questions: Did the users get closer because of a completely new (and shorter) path appeared, which was "disjoint" from the previously shortest path (see Figure 2(a))? Or was it because a "short-circuiting bridge" was established (Figure 2(b))? Or was it because a new user had arrived that acted as a "common friend" of some users along the previously shortest path (Figure 2(c))? How "important" was this common friend in the network and how does its importance evolve over time? (E.g., how does its *centrality* evolve across snapshots?) Furthermore, how does the *diameter* of the friendship network evolve over time? The objective of this paper is to provide efficient solutions for evaluating queries of this kind. Recent works on social network evolution analysis (e.g., [7]) focus on "what" information can be discovered from graph evolution. In this work, we focus on "how" those information can be efficiently obtained. To do so, we have developed a solution framework, called FVF, for efficient EGS query processing. We demonstrate how some important graph measures, including shortest-path distance, closeness centrality, and graph diameter, can be efficiently computed from EGSs using our framework. Since an EGS generally contains numerous large graphs, we also discuss several compact storage models that support our FVF framework.

The remainder of this paper is organized as follows. In Section 2, we present the FVF framework and illustrate how to use the framework to compute different graph measures. In Section 3, we present several EGS storage models. In Section 4, we present the major experimental results. In Section 5, we discuss related work. In Section 6, we conclude the paper. The Appendix contains some pseudocode, supplementary discussion, further experiment details, and a case study that gives answers to the example social network analytical queries mentioned above.

2. SOLUTION FRAMEWORK

Given an evolving graph sequence $EGS = \langle G_1, \ldots, G_n \rangle$, where each G_i is a directed graph, and a query Q (e.g., shortest-path between two vertices), our objective is to efficiently answer Q on each snapshot in an EGS. In this paper, we focus on directed graphs. Applying our solutions to undirected graphs is straightforward and therefore we will skip the discussion on undirected graphs.

To support efficient EGS query processing, we propose a Find-Verify-and-Fix (FVF) solution framework (see Figure 3). The FVF framework consists of two phases, namely, a *preprocessing phase* and a *query-processing phase*. The purpose of the preprocessing phase is to construct a small number of *representative graphs* from the EGS. Each representative, say G_R, captures the structural similarity among some snapshots in the EGS and is thus similar to those snapshots. To achieve that, we group the snapshot graphs into clusters such that graphs in the same cluster satisfy certain similarity requirements. A representative graph is then constructed based on the members of a cluster.

The query-processing phase consists of three steps: The first step is to "**FIND**" a solution S_R for each representative graph G_R. Since G_R is similar to many graphs in the EGS, its solution, S_R, should also be a representative solution for those graphs. The second step is thus to "**VERIFY**" whether S_R is indeed a solution to them. For a graph, say G_i, whose solution cannot be found by successful verification, we proceed to the third step and attempt to modify or "**FIX**" S_R to obtain a solution for G_i. The rationale of FVF is that solution verification and modification can potentially be done much more efficiently than computing the solution from scratch. In this section, we will demonstrate how shortest-path distance and closeness centrality can be efficiently computed from an EGS using our FVF framework. Due to space constraints, we put the discussion on computing graph diameter in Appendix E. Now, we first discuss the preprocessing phase in detail.

2.1 Preprocessing

The preprocessing phase consists of two steps. First, we group similar snapshot graphs together. Then, we extract two representative graphs from each cluster. We discuss the second step first assuming that clustering has already been done. We will come back to the issue of graph clustering later. Given a graph G, we use $V(G)$ and $E(G)$ to denote the vertex set and the edge set of G, respectively.

Suppose a certain number of snapshot graphs have been grouped into a cluster C. W.l.o.g., let $C = \{G_1, \ldots, G_k\}$. We construct two representative graphs for C: (1) $G_\cap$, which is the intersection (the largest common subgraph) of all snapshots in C, and (2) $G_\cup$, which is the union (the smallest common supergraph) of all snapshots in C. Note that,

PROPERTY 1. $G_\cap \subseteq G_i \subseteq G_\cup \quad \forall 1 \leq i \leq k.$[1]

Given two vertices u and v, if we use $\mathcal{P}(u, v, G)$ to denote the set of paths connecting u to v in a certain graph G, then $\mathcal{P}(u, v, G_\cap)$ and $\mathcal{P}(u, v, G_\cup)$ give a subset and a superset of $\mathcal{P}(u, v, G_i)$, respectively. $G_\cap$ and $G_\cup$ are very useful to query-processing. For example, knowing the shortest paths in $G_\cap$ and $G_\cup$ allows us to effectively "*bound*" the shortest path solution in G_i (Section 2.2).

In order to partition the graphs in an EGS into clusters, we need to define a similarity measure. In this paper we use a *normalized graph edit similarity* (ges) measure that is based on the symmetric difference of the graphs' edge sets. Formally,

DEFINITION 1 (GRAPH EDIT SIMILARITY). *Given two graphs G_a and G_b,*

$$ges(G_a, G_b) = \frac{2|E(G_a \cap G_b)|}{|E(G_a)| + |E(G_b)|}. \tag{1}$$

Graphs that are grouped into the same cluster should be sufficiently similar. Since $G_\cup$ and $G_\cap$ essentially "bound" the graphs in the cluster, the similarity requirement of a cluster can be conveniently captured by the similarity between $G_\cup$ and $G_\cap$.

DEFINITION 2 (α-SIMILARITY). *Two graphs G_a and G_b are said to be α-similar if and only if $ges(G_a, G_b) \geq \alpha$.*

DEFINITION 3 (α-BOUNDEDNESS). *A cluster C of snapshots is said to be α-bounded if and only if $G_\cup$ and $G_\cap$ are α-similar.*

[1]Here, "$\subseteq$" denotes sub-graph relation.

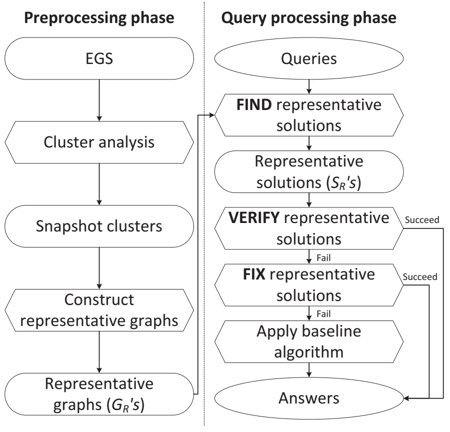

Preprocessing phase
Query processing phase
EGS
Cluster analysis
Snapshot clusters
Construct representative graphs
Representative graphs (G_R's)
Queries
FIND representative solutions
Representative solutions (S_R's)
VERIFY representative solutions
Succeed
Fail
FIX representative solutions
Succeed
Fail
Apply baseline algorithm
Answers

Figure 3: The FVF framework.

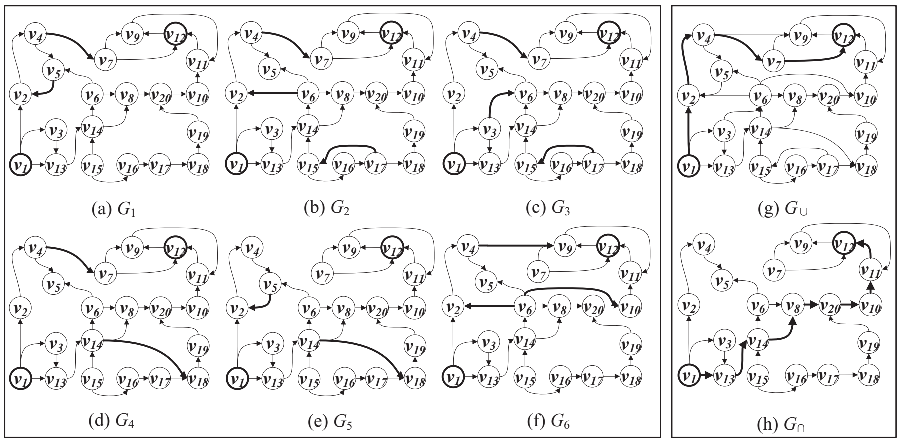


Figure 4: A running example.

Since typically an EGS models the continuous evolution of a graph, we use a simple segmentation strategy to group snapshots into clusters: Let $\langle G_1, \ldots G_n \rangle$ be an EGS and α be a user-specified similarity threshold. We start with an empty cluster C_1 and incrementally insert the snapshots into C_1 beginning with G_1, then G_2, etc., as long as C_1 remains α-bounded. If the bounding requirement would have been violated by adding one more snapshot, we start building the next cluster C_2 and repeat the process. This incremental clustering procedure is well-suited for incremental updates of EGSs — as more snapshots are archived into an EGS, only the last cluster needs to be updated. Also, the procedure essentially processes the graphs in an EGS as a stream — it only needs to maintain one pair of $G_\cup$ and $G_\cap$, and to access one snapshot G_i at a time. This allows the clustering process to be done efficiently without requiring all the snapshots be present in memory. Appendix A shows the clustering algorithm. Finally, we note that graphs of the same cluster are consecutive snapshots of an EGS. Also, a larger α implies that $G_\cap$ and $G_\cup$ of a cluster are more similar, which then implies a tighter bounding requirement. This results in fewer snapshots in a cluster and more clusters segmented from an EGS.

2.2 Shortest-Path Query Processing

We now describe how to apply the concept of Find-Verify-and-Fix to find the shortest path SP_i between a pair of vertices u and v for each snapshot $G_i \in EGS$. Obviously, one baseline algorithm is to execute a shortest-path algorithm (SPA), say, BFS for unweighted graphs or Dijkstra's algorithm for weighed ones, directly on each snapshot in an EGS. This approach, which requires n executions of SPA, is not efficient for EGSs whose graphs are large and numerous because the total execution time is dominated by the SPA executions. Therefore, the objective of FVF is to exploit the precomputed clusters and representative graphs to minimize the number of SPA executions.

First, let us assume the snapshots are unweighted. FVF evaluates a shortest-path query Q on an EGS one cluster at a time. Let $C = \{G_1, \ldots, G_k\}$ be a cluster. Figure 4 shows an example cluster containing 6 snapshots and the derived $G_\cup$ and $G_\cap$. We will use this as a running example to illustrate how we efficiently find a shortest path from v_1 to v_{12} in each snapshot. We use (v_i, v_j) to denote the edge that connects vertex v_i to v_j. For each snapshot G_i, the delta set $\Delta(G_i, G_\cap) = E(G_i) - E(G_\cap)$ is highlighted. For example, the edges (v_5, v_2) and (v_4, v_7) are in G_1 but not in $G_\cap$.

Given vertices u and v, we use $\widetilde{P}_*(u, v)$ to denote a uv-shortest-path in graph G_*, where $* = 1, 2, ..., n, \cup, \cap$. We use $\delta_*(u, v)$ to denote the path length of $\widetilde{P}_*(u, v)$. If v is not reachable from u in G_*, we define $\delta_*(u, v) = \infty$. If a graph G_a is a subgraph of another graph G_b, then obviously any path P in G_a must also be in G_b. Hence, the set of paths connecting u to v in G_a must be a subset of that set of paths in G_b. Therefore, $\delta_b(u, v) \le \delta_a(u, v)$. Since $G_\cap \subseteq G_i \subseteq G_\cup$ for any snapshot G_i, we have

PROPERTY 2. *$\delta_\cup(u, v) \le \delta_i(u, v) \le \delta_\cap(u, v)$ for any vertices u, v and snapshots $G_i \in C$.*

The path length in any snapshot is thus bounded above and below by those of $G_\cap$ and $G_\cup$, respectively. For example, in Figures 4g and 4h, a $\widetilde{P}_\cup(v_1, v_{12})$ and a $\widetilde{P}_\cap(v_1, v_{12})$ are highlighted. Since $\delta_\cup(v_1, v_{12}) = 4$ and $\delta_\cap(v_1, v_{12}) = 7$, the length of any shortest path that connects v_1 to v_{12} in any snapshot is bounded by 4 and 7. These bounds provide interesting methods for solution verification and modification.

Following our framework, our FVF algorithm first runs SPA on $G_\cup$ to FIND a shortest path in $G_\cup$, i.e., $\widetilde{P}_\cup(u, v)$. If it turns out that v is not reachable from u in $G_\cup$, then v is not reachable from u in any snapshot in cluster C. In this case, only one SPA (on $G_\cup$) is executed and there is no need to run any SPAs on the snapshots in C. This observation is captured by the following lemma:

LEMMA 1 (L1). *If $\delta_\cup(u, v) = \infty$ then v is not reachable from u in any snapshot $G_i \in C$.*

If $\widetilde{P}_\cup(u, v)$ exists, we run SPA on the other representative graph $G_\cap$ to obtain $\widetilde{P}_\cap(u, v)$. If $\delta_\cap(u, v) = \delta_\cup(u, v)$, then by Property 2, $\delta_i(u, v) = \delta_\cap(u, v)$. Since $G_\cap \subseteq G_i$ for any snapshot G_i, the path $\widetilde{P}_\cap(u, v)$ must exist in all G_i's. Hence, $\widetilde{P}_\cap(u, v)$ is a uv-shortest-path in all G_i's. Again, there is no need to run any SPA on the snapshots in C. This discussion leads to Lemma L2:

LEMMA 2 (L2). *If $\delta_\cup(u, v) = \delta_\cap(u, v)$, then $\widetilde{P}_\cap(u, v)$ is a solution for any $G_i \in C$.*

If $\delta_\cup(u, v) \ne \delta_\cap(u, v)$, Lemma L2 does not apply. By Property 2, $\delta_\cup(u, v)$ is a lower bound of the length of any uv-shortest-paths in any snapshot G_i. So if $\widetilde{P}_\cup(u, v)$ exists in G_i, it must be a shortest one and hence a solution for G_i. Our algorithm identifies all the snapshots in the cluster in which the path $\widetilde{P}_\cup(u, v)$ exists. This path checking can be easily done and is much less costly than executing SPA on those snapshots.

LEMMA 3 (L3). *If $\widetilde{P}_{\cup}(u,v)$ exists in G_i, then $\widetilde{P}_{\cup}(u,v)$ is a solution for G_i.*

For example, in Figure 4, $\widetilde{P}_{\cup}(v_1,v_{12}) = (v_1,v_2,v_4,v_7,v_{12})$ is present in G_1, G_2, G_3 and G_4. Therefore, $\widetilde{P}_{\cup}(v_1,v_{12})$ is a solution for those four snapshots.

If $\widetilde{P}_{\cup}(u,v)$ does not exist in G_i, we cannot apply Lemma L3. For example, in Figure 4, $\widetilde{P}_{\cup}(v_1,v_{12})$ is not contained in G_5. In this case, we turn to $\widetilde{P}_{\cap}(u,v)$ again and try to verify if $\widetilde{P}_{\cap}(u,v)$ is indeed a solution for G_i. To do this, we consider $\Delta(G_i,G_{\cap}) = E(G_i) - E(G_{\cap})$. First, we know that $\widetilde{P}_{\cap}(u,v)$ exists in every snapshot and so it is a path in G_i in particular. If we can prove that no edges in $\Delta(G_i,G_{\cap})$ can give us a path in G_i that is shorter than $\widetilde{P}_{\cap}(u,v)$, then $\widetilde{P}_{\cap}(u,v)$ must be a solution for G_i.

Consider any edge $e=(p,q) \in \Delta(G_i,G_{\cap})$. Let $\widetilde{P}_i(u,v|e)$ be a shortest path that connects u to v in G_i that goes through the edge e. This path must consist of three segments: (1) a shortest path that goes from u to p, (2) the edge e and (3) a shortest path that goes from q to v. Hence,

$$|\widetilde{P}_i(u,v|e)| = \delta_i(u,p)+1+\delta_i(q,v) \geq \delta_{\cup}(u,p)+1+\delta_{\cup}(q,v)$$

Given $e=(p,q)$, define $\Gamma(e) = \delta_{\cup}(u,p)+1+\delta_{\cup}(q,v)$. Note that $\Gamma(e)$ gives a lower bound on the length of the shortest path from u to v that goes through e in G_i. Consider the following condition:

CONDITION 1. $\Gamma(e) = \delta_{\cup}(u,p)+1+\delta_{\cup}(q,v) \geq \delta_{\cap}(u,v)$

If Condition 1 holds, we know that any uv-shortest-path in G_i that uses e cannot be shorter than $\delta_{\cap}(u,v)$. We can ignore e in quest of a path that is shorter than $\delta_{\cap}(u,v)$. Let Γ_i^* be the smallest $\Gamma(e)$ for all $e \in \Delta(G_i,G_{\cap})$. If Condition 1 holds for all edges in $\Delta(G_i,G_{\cap})$, i.e., $\Gamma_i^* \geq \delta_{\cap}(u,v)$, then $\widetilde{P}_{\cap}(u,v)$ must be a uv-shortest-path for G_i. This gives us the fourth lemma:

LEMMA 4 (L4). *If $\Gamma_i^* = \min_{e\in\Delta(G_i,G_{\cap})}\{\Gamma(e)\} \geq \delta_{\cap}(u,v)$, then $\widetilde{P}_{\cap}(u,v)$ is a solution for G_i.*

For example, in Figure 4, $\Delta(G_5,G_{\cap}) = \{(v_5,v_2),(v_{14},v_{18})\}$ and $\delta_{\cap}(v_1,v_{12}) = 7$. Since $\Gamma((v_5,v_2)) = \delta_{\cup}(v_1,v_5)+1+\delta_{\cup}(v_2,v_{12}) = 3+1+3 = 7$ and $\Gamma((v_{14},v_{18})) = 2+1+5 = 8$, both are larger than or equal to $\delta_{\cap}(v_1,v_{12})$. $\widetilde{P}_{\cap}(v_1,v_{12})$ is thus a v_1v_{12}-shortest-path for G_5.

To apply Lemma L4, we need $\delta_{\cup}(u,p)$ and $\delta_{\cup}(q,v)$ for each $e=(p,q)\in\Delta(G_i,G_{\cap})$. These can be obtained by running SPA twice on $G_{\cup}$ to determine the shortest paths from u to all other vertices and the shortest paths from all other vertices to v. This information will also be used in the next lemma.

If $\Gamma_i^* < \delta_{\cap}(u,v)$ ($\therefore$ Lemma L4 is not applicable), our VERIFICATION based on the four lemmas has not been successful. In this case, we attempt to FIX for a better solution. At this point we know that (1) any uv-shortest-path in G_i that does not use any edges in $\Delta(G_i,G_{\cap})$ cannot be shorter than $\delta_{\cap}(u,v)$; (2) any uv-shortest-path in G_i that uses any edges in $\Delta(G_i,G_{\cap})$ cannot be shorter than Γ_i^*; and (3) $\Gamma_i^* < \delta_{\cap}(u,v)$. Hence, any uv-shortest-path in G_i cannot be shorter than Γ_i^*. Our algorithm attempts to find a path of length Γ_i^* in G_i. If it exists, it must be the shortest one.

Let $\Delta^*(G_i) = \{e \in \Delta(G_i,G_{\cap}) | \Gamma(e) = \Gamma_i^*\}$ be the set of edges with the smallest $\Gamma()$ value in $\Delta(G_i,G_{\cap})$. (In our example, $\Delta^*(G_6) = \{(v_4,v_9),(v_6,v_{10})\}$.) For each edge $e=(p,q)\in\Delta^*(G_i)$, we retrieve the paths $\widetilde{P}_{\cup}(u,p)$ and $\widetilde{P}_{\cup}(q,v)$. Our algorithm checks if these two paths exist in G_i. If so, the path $P = \widetilde{P}_{\cup}(u,p)\|e\|\widetilde{P}_{\cup}(q,v)$ is contained in G_i. Since $|P| = \Gamma_i^*$, path P must be a uv-shortest-path for G_i. For example, $(v_4,v_9)\in\Delta^*(G_6)$ and the paths $\widetilde{P}_{\cup}(v_1,v_4) = (v_1,v_2,v_4)$ and $\widetilde{P}_{\cup}(v_9,v_{12}) = (v_9,v_{11},v_{12})$ are both present in G_6. Therefore, the path $(v_1, v_2, v_4, v_9, v_{11}, v_{12})$ must be a v_1v_{12}-shortest-path for G_6. This leads to the fifth lemma:

LEMMA 5 (L5). *If $\exists e=(p,q)\in\Delta^*(G_i)$ such that both $\widetilde{P}_{\cup}(u,p)$ and $\widetilde{P}_{\cup}(q,v)$ are present in G_i, then $\widetilde{P}_{\cup}(u,p)\|e\|\widetilde{P}_{\cup}(q,v)$ is a solution for G_i.*

Finally, if none of the five lemmas lead to the solution of a snapshot G_i, we execute SPA on G_i to obtain its uv-shortest-path. Algorithm 1 summarizes the FVF algorithm for shortest-path queries. Lines 1 to 5 find representative solutions on $G_{\cup}$ and $G_{\cap}$ (the FIND phase). Lines 2 to 4 and Lines 6 to 29 apply the five lemmas for either verifying a representative solution or fixing a solution (the VERIFY-FIX phase). Line 30 applies SPA on those snapshots whose solutions cannot be determined by the lemmas (the Residual SPA phase).

Algorithm 1: The FVF Shortest Path Algorithm.

```
Input  : Query $(u,v)$, Cluster $C = \langle G_1,G_2,\ldots,G_k\rangle$,
         $G_{\cup}, G_{\cap}$,
         $\langle\Delta(G_1,G_{\cap}),\Delta(G_2,G_{\cap}),\ldots,\Delta(G_k,G_{\cap})\rangle$
Output: $\langle\widetilde{P}_1(u,v),\widetilde{P}_2(u,v),\ldots,\widetilde{P}_k(u,v)\rangle$
1  run SPA on $G_{\cup}$ to obtain $\widetilde{P}_{\cup}(u,v)$
2  if $\delta_{\cup}(u,v) = \infty$ then                                // L1
3      return $\widetilde{P}_i(u,v)$ does not exist (for $i = 1,2,\ldots,k$)
4  end
5  run SPA on $G_{\cap}$ to obtain $\widetilde{P}_{\cap}(u,v)$
6  if $\delta_{\cup}(u,v) = \delta_{\cap}(u,v)$ then                   // L2
7      $\widetilde{P}_i(u,v) \leftarrow \widetilde{P}_{\cap}(u,v)$ (for $i = 1,2,\ldots,k$)
8      return $\langle\widetilde{P}_1(u,v),\widetilde{P}_2(u,v),\ldots,\widetilde{P}_k(u,v)\rangle$
9  end
10 $flag \leftarrow$ false
11 foreach $G_i \in C$ do
12     if $\widetilde{P}_{\cup}(u,v)$ exists in $G_i$ then              // L3
13         $\widetilde{P}_i(u,v) \leftarrow \widetilde{P}_{\cup}(u,v)$
14         continue
15     end
16     if flag = false then
17         run SPA on $G_{\cup}$ to get all shortest paths from $u$
18         run SPA on $G_{\cup}$ to get all shortest paths to $v$
19         $flag \leftarrow$ true
20     end
21     calculate $\Gamma_i^* = \min_{e\in\Delta(G_i,G_{\cap})}(\Gamma(e))$
22     if $\Gamma_i^* \geq \delta_{\cap}(u,v)$ then                    // L4
23         $\widetilde{P}_i(u,v) \leftarrow \widetilde{P}_{\cap}(u,v)$
24         continue
25     end
26     if $\exists e = (p,q) \in \Delta^*(G_i)$ such that both $\widetilde{P}_{\cup}(u,p)$ and
       $\widetilde{P}_{\cup}(q,v)$ are present in $G_i$ then             // L5
27         $\widetilde{P}_i(u,v) \leftarrow \widetilde{P}_{\cup}(u,p)\|e\|\widetilde{P}_{\cup}(q,v)$
28         continue
29     end
               // Solution not found by L1-L5
30     run SPA on $G_i$ to obtain $\widetilde{P}_i(u,v)$
31 end
32 return $\langle\widetilde{P}_1(u,v),\widetilde{P}_2(u,v),\ldots,\widetilde{P}_k(u,v)\rangle$
```

Now, we extend our discussion to EGSs with weighted graphs. In a *weighted EGS*, there are structural changes (vertex and edge inserts/deletes) as well as edge weight adjustments when one snapshot evolves to the next. In real applications, there are relatively few structural changes (e.g., successive graphs in Facebook Wall

Post weighted EGS share more than 99% of the edges; Section 4 has more detail about this EGS) and moderate edge weight adjustments (e.g., the number of Wall Post messages sent daily between two users vary little). If we consider all such edge-weight adjustments, then the shortest paths in the snapshots of a cluster would all have slightly different path lengths. Recall that the idea of FVF is to find a representative solution S_R and verify that S_R is indeed the solution to many snapshots in a cluster. With the slightly changing path lengths across the snapshots, it would be difficult to find an S_R that gives the *exact solution* to many snapshots.

In order to apply the FVF framework on weighted EGSs, we shall trade absolute accuracy for efficiency. Specifically, we consider a user-specified tolerance ε. Given a shortest-path query (u, v), our goal is to find a path P_i in each snapshot G_i such that P_i connects u to v and that its path length is guaranteed to differ from the true shortest path in G_i (i.e., $\widetilde{P}_i(u,v)$) by at most a factor of ε.

To do so, we need to revise some key elements and lemmas that we just presented. Given an edge e, let $w_*(e)$ be the weight of e in a graph G_*. Given a cluster $C = \{G_1, \ldots, G_k\}$, $G_\cup$ and $G_\cap$ are constructed in the same way as in the unweighted case. The edge weights $w_\cup(e)$ and $w_\cap(e)$ are set to be the minimum and the maximum of all $w_i(e)$ ($1 \le i \le k$), respectively. It is easy to show that under these definitions of $w_\cup(e)$ and $w_\cap(e)$, Properties 1 and 2 remain correct. In particular, Property 2 tells us that we can use the lengths of the shortest paths found in $G_\cup$ and $G_\cap$ (i.e., $\delta_\cup(u,v)$ and $\delta_\cap(u,v)$) to bound the solution of the shortest paths in the snapshot graphs. The five lemmas can then be revised accordingly to prune the corresponding SPA executions (e.g., Dijkstra's algorithm). For example, if $\delta_\cap(u,v) \le (1+\varepsilon)\delta_\cup(u,v)$, then by Property 2, we have, $\forall 1 \le i \le k$,

$$\delta_i(u,v) \le \delta_\cap(u,v) \le (1+\varepsilon)\delta_\cup(u,v) \le (1+\varepsilon)\delta_i(u,v).$$

So, $\widetilde{P}_\cap(u,v)$ satisfies the error tolerance requirement and is a solution to all the snapshots in the cluster. Appendix C details the revised definitions and lemmas for processing shortest-path queries on weighted EGSs.

2.3 Closeness Centrality

In this section, we briefly discuss how FVF is applied to compute centrality across snapshots in EGS efficiently. There are a few centrality measures. As an example, let us consider *closeness centrality*, which is defined as:

DEFINITION 4 (CLOSENESS CENTRALITY). *For any vertex u of a graph G_*, the closeness centrality of u in G_*, denoted by $cc_*(u)$, is $cc_*(u) = 1/\sum_{v\in V(G_*)} \delta_*(u,v)$.*

$cc_*(u)$ is the reciprocal of the sum of distances from u to any other vertices in G_*. Obviously, one can obtain $cc_*(u)$ by running a single-source-shortest-path algorithm (SSSP) at vertex u. So, a brute-force approach to determine the closeness centrality of u for all the snapshot graphs in a cluster $C = \{G_1, \ldots, G_k\}$ would be to repeat SSSP k times, one for each snapshot. This takes $O(k(E+V))$ time for unweighted graphs and $O(kE \lg V)$ time for weighted ones, where E and V are the number of edges and the number of vertices of a snapshot graph, respectively.

Similar to finding shortest-paths on weighted-graphs, if a user accepts an approximate answer, the FVF approach can be significantly faster than the brute-force approach. By Property 2, one can easily verify that

$$cc_\cap(u) \le cc_i(u) \le cc_\cup(u) \ \ \forall 1 \le i \le k. \tag{2}$$

We consider the average $\overline{cc(u)} = (cc_\cap(u) + cc_\cup(u))/2$. Given an error tolerance ε, if $cc_\cap(u)(1+2\varepsilon) \ge cc_\cup(u)$, then we can prove that $\overline{cc(u)}$ differs from the true centrality $cc_i(u)$ (for any $1 \le i \le k$) by at most a fraction of ε. We thus accept $\overline{cc(u)}$ as the answer for *all* snapshots in cluster C. In this case, only 2 SSSP are needed (one for each of $G_\cup$ and $G_\cap$).

If the condition $cc_\cap(u)(1+2\varepsilon) \ge cc_\cup(u)$ does not hold, we can tighten the lower bound of $cc_i(u)$ by doing a verification. Notice that when we apply SSSP on $G_\cup$, we obtain the shortest paths from u to all other vertices in $G_\cup$. These paths can be concisely represented by a shortest-paths tree, $SPT_\cup(u)$, rooted at u. We can verify which of those shortest paths are present in G_i and which are not by walking through $SPT_\cup(u)$ and checking which edges in the tree are present in G_i. For example, suppose $(u, v_1, v_2, \ldots, v_d)$ is a path from u to v_d in $SPT_\cup(u)$. If all the d edges in the path are present in G_i, then we know that $\delta_i(u,v_j) = \delta_\cup(u,v_j)$ for all $1 \le j \le d$. If we collect all vertices v whose shortest paths from u are successfully verified into a set A, then

$$\begin{aligned} cc_i(u) &= 1/\sum_{v\in V(G_i)} \delta_i(u,v) = 1/(\sum_{v\in A} \delta_i(u,v) + \sum_{v\in V(G_i)-A} \delta_i(u,v)) \\ &\ge 1/(\sum_{v\in A} \delta_\cup(u,v) + \sum_{v\in V(G_i)-A} \delta_\cap(u,v)) = R_i(u). \end{aligned} \tag{3}$$

Notice that the RHS of Equation 3 (i.e., $R_i(u)$) can be obtained from the executions of SSSP on $G_\cap$ and $G_\cup$ and it is a tighter lower bound than $cc_\cap(u)$. We can revise our answer for snapshot G_i to $\overline{cc(u)'} = (R_i(u) + cc_\cup(u))/2$, which is more likely to satisfy the tolerance requirement. Finally, if this revised answer does not satisfy the tolerance requirement, we run SSSP on G_i to get the exact value of $cc_i(u)$. We remark that the shortest-paths tree verification takes $O(V)$ time because there are only $V-1$ branches in the tree whose existences in G_i are checked. This is much more efficient than the $O(E+V)$ (unweighted) or $O(E \lg V)$ (weighted) time needed to do an SSSP on G_i under the brute-force approach.

3. EGS STORAGE MODELS

An EGS typically consists of a large number of big graphs. The storage requirement of an EGS could be big. In this section we discuss a few storage models for compressing EGS data that (1) are space efficient so that the compressed data is likely small enough to be stored in main memory and (2) can efficiently support the applications of the pruning lemmas of the FVF algorithm.

In the following discussion, let us focus on representing a cluster of snapshots $C = \{G_1, \ldots, G_k\}$. The FVF algorithm requires two representative graphs $G_\cap$, $G_\cup$ and k delta sets, $\Delta(G_i, G_\cap), \forall 1 \le i \le k$, as input. Note that this information is sufficient for FVF to compute the representative solutions and to perform verify-and-fix by applying the lemmas. In case a graph algorithm has to be executed on a snapshot G_i (e.g., line 30 of Algorithm 1), the algorithm can be done by considering the edge sets of $\Delta(G_i, G_\cap)$ and $G_\cap$ together. The snapshots, G_i's, of the cluster need not be explicitly stored. We call this storage model *SM1*:

$$SM1(C) = \{G_\cap, \Delta(G_\cup, G_\cap), \Delta(G_i, G_\cap) | 1 \le i \le k\}.$$

We have collected a number of social network datasets for our experiment (see Section 4 for their descriptions). From them, we observe that the size of a delta set $\Delta(G_i, G_\cap)$ is typically less than 10% of the size of a snapshot G_i[2]. *SM1* is therefore much more efficient than storing each snapshot explicitly. For example, the total size of our Wikipedia graphs (if they are stored independently) is 45GB while it is about 4.5GB under *SM1*.

For very large EGSs, we need to compress the data further for more efficient processing. For an evolving graph sequence, we observe that successive snapshots are typically very similar. Let $\mathcal{D}(G_i, G_{i-1}) = (E_i^+, E_i^-)$, where $E_i^+ = E(G_i) - E(G_{i-1})$ and

[2]The sizes of the delta sets depend on the α threshold value used (see Section 2.1). We will further discuss the effect of α later in Section 4.

Dataset	FBFRIEND	YOUTUBE	WIKIPEDIA	FBWALL
Graph Type	undirected	undirected	directed	directed
Vertex Information	user	user	article	user
Edge Information	friendship	friendship	hyperlink	message
Number of Snapshots	365	203	365	365
Snapshot Frequency	daily	daily	daily	daily
$\|V\|$ of First Snapshot	26,249	1,004,777	1,352,623	18,859
$\|V\|$ of Last Snapshot	61,096	3,223,643	1,870,709	42,859
$\|E\|$ of First Snapshot	251,251	8,782,672	19,956,191	90,694
$\|E\|$ of Last Snapshot	905,552	37,048,190	39,953,145	188,869
Average ges	99.824%	99.644%	99.905%	99.752%

Table 1: Statistics of real datasets.

$E_i^- = E(G_{i-1}) - E(G_i)$, be the changes made to snapshot G_{i-1} to obtain the next snapshot G_i (i.e., $E(G_i) = E(G_{i-1}) \cup E_i^+ - E_i^-$). From our real datasets, we observe that the size of the set of edge changes $\mathcal{D}(G_i, G_{i-1})$ is on average just 1/10 the size of $\Delta(G_i, G_\cap)$. Hence, representing an EGS in terms of the $\mathcal{D}$'s is much more space efficient than in term of the Δ's. We thus consider another storage model *SM2*:

$$SM2(C) = \{G_\cap, \Delta(G_\cup, G_\cap), \Delta(G_1, G_\cap), \mathcal{D}(G_i, G_{i-1}) | 2 \le i \le k\}.$$

It is obvious that we can obtain $G_\cap$, $G_\cup$ and $\Delta(G_1, G_\cap)$ from the first three elements of *SM2*. In order to apply the lemmas for solution verification and fixing, we need to recover the other delta sets ($\Delta(G_i, G_\cap)$ for $2 \le i \le k$) as well. We note that

$$\Delta(G_i, G_\cap) = \Delta(G_{i-1}, G_\cap) \cup E_i^+ - E_i^-. \quad (4)$$

(See Appendix B.) And hence, $\Delta(G_i, G_\cap)$ can be obtained from $\Delta(G_{i-1}, G_\cap)$ and $\mathcal{D}(G_i, G_{i-1})$. By induction, all $\Delta(G_i, G_\cap)$'s can be recovered under *SM2* to apply our lemmas.

Compared with *SM1*, *SM2* results in a much smaller storage requirement. Note that, under *SM2*, decompression is needed (using Equation 4) to recover $\Delta(G_i, G_\cap)$ before some of the lemmas can be applied. For example, in shortest-path query processing on an EGS, this decompression should be done right before line 10 in Algorithm 1. For large EGSs, however, the savings in I/O cost due to a much smaller data size under *SM2* and the possibility of storing the data in main memory far outweighs the penalty of the additional decompression overheads.

Further compression can be achieved by exploiting inter-cluster redundancy. We note that in our real datasets, the $G_\cap$'s of successive clusters are somewhat similar too. Instead of storing the $G_\cap$ of a cluster C, we can store the difference between it and $Gp_\cap$, where $Gp_\cap$ is "the $G_\cap$ of the preceding cluster." In our experiment, we enhance the storage model *SM2* by this extension and we use it as the storage model for our FVF algorithm. We call the resulting storage model *SM-FVF*:

$$SM\text{-}FVF(C) = \{\mathcal{D}(G_\cap, Gp_\cap), \Delta(G_\cup, G_\cap), \Delta(G_1, G_\cap), \mathcal{D}(G_i, G_{i-1}) | 2 \le i \le k\}.$$

For the Wikipedia dataset, the storage needed under *SM-FVF* is 387MB, which is 0.86% of the size of the uncompressed data.

Recall that the baseline algorithm applies a graph algorithm (e.g., BFS) on each individual snapshot graph. It does not require any $G_\cap$'s or $G_\cup$'s. The most compact storage model for the baseline method is to store the differences between successive snapshots. We call this storage model (for Baseline) *SM-BL*:

$$SM\text{-}BL(C) = \{G_1, \mathcal{D}(G_i, G_{i-1}) | 2 \le i \le k\}.$$

4. EXPERIMENTAL EVALUATION

We evaluate the FVF framework on both real datasets (Table 1) and synthetic datasets. In the experiments, we use BFS and Dijkstra's algorithm as SPA/SSSP for unweighted graphs and weighted graphs, respectively. We compare the performance of our FVF framework with the baseline algorithm (Baseline), which executes the corresponding graph algorithm (e.g., BFS/Dijkstra's) on every snapshot. Appendix D.1 details the experiment platform.

4.1 Shortest-Path

Performance Study (Real Data). We first report the performance of the baseline algorithm and the FVF algorithm on three unweighted real datasets, namely FBFRIEND, YOUTUBE, WIKIPEDIA.[3] The properties of the real datasets are given in Table 1. For example, FBFRIEND is an EGS of 365 daily snapshots of the Facebook (New Orleans) friendship graph, all taken in the year 2008. A graph vertex represents a Facebook user and an edge connects two users (vertices) if they are friends in Facebook. We store the snapshot graphs under the storage models proposed in Section 3. The storage requirements of the highly-compressed datasets are listed in Table 2. (Recall that Baseline employs *SM-BL*, while FVF employs *SM-FVF*.) From the table, we see that both storage models result in very compact data (at most several hundred megabytes), which is a reasonable size for main memory processing. The graphs in all datasets are evolving, and in general, they grow with time. For all real datasets, the average values of ges (graph edit similarity) between consecutive snapshots are very high ($> 99\%$), indicating that successive snapshots in the EGSs are indeed very similar to each other. Therefore, we set the default similarity threshold α to a large number, 0.95, in clustering the snapshots in an EGS. (We will study the effect of α later.) The preprocessing times of FBFRIEND, YOUTUBE, and WIKIPEDIA data (which include the time for clustering snapshots and constructing $G_\cap$'s and $G_\cup$'s) are 1 second, 115 seconds, and 26 seconds, respectively.

Table 2 shows the averages of some measurements obtained by executing 500 random queries on the datasets. From the table, we can see that processing queries on an EGS is indeed an expensive operation if the inefficient baseline algorithm is used. For example, processing a query on FBFRIEND and WIKIPEDIA requires running SPA 365 times and the average query processing times are 3.1 and 104 seconds, respectively[4]. The last column of Table 2 (*Speedup*) shows that FVF significantly outperforms the baseline algorithm — it runs 5.4 times (YOUTUBE), 7.7 times (WIKIPEDIA), and 9.1 times (FBFRIEND) faster than the baseline algorithm.

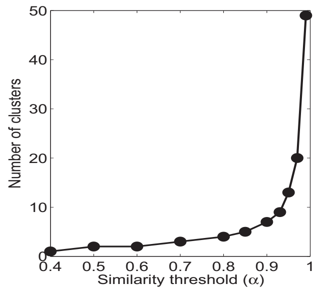
We further break down FVF's execution time into a few key components. These include the times spent on (a) decompressing graph data (*Decompress* Time), (b) finding representative solutions from representative graphs (*Find* Time), (c) verifying whether a representative solution is a solution of a snapshot, and if not, attempting to fix it (Verify-and-Fix, or *VF* Time), and (e) running SPAs on graphs whose solutions cannot be determined by the lemmas (*Residual-SPA* Time). We note that, under FVF, it is very likely that the solution for a snapshot can be deduced by the lemmas. In that case, no SPA needs to be executed on the snapshot, and we say that the SPA is *pruned*. Table 2 column (d) shows the percentage of SPAs pruned by the lemmas: 100% × (1−(total number of SPA executed for 500 random queries / number of graphs in EGS × 500)). From the experimental results, we can see that the "investment" of the FVF algorithm has a very good "return" — times (b) and (c) can be considered as the "time investment" of FVF. In return, more than 95% of the SPAs that would have to be executed on snapshots are avoided. This explains the significant speedup FVF achieves over Baseline.

[3]Data available at `http://socialnetworks.mpi-sws.org`

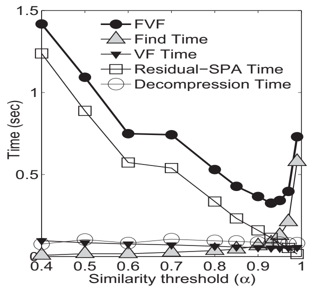
[4]The query-processing time is so much larger for WIKIPEDIA because the Wikipedia snapshot graphs are much larger than those of FBFRIEND (see Table 1).

EGS	Storage		Average Query Time — Baseline			Average Query Time — FVF						Speedup
	SM-BL	SM-FVF	(i) *Decompress*	(ii) *SPA*	**Total**	(a) *Decompress*	(b) *Find*	(c) *VF*	(d) Pruned SPA (%)	(e) *Residual-SPA*	**Total**	
FBFRIEND	12.4MB	25.6MB	0.09s	3.00s	**3.10s**	0.08s	0.13s	0.05s	97.04%	0.08s	**0.34s**	9.1
YOUTUBE	217.8MB	481.6MB	2.33s	50.36s	**52.68s**	1.59s	4.70s	1.48s	95.88%	1.96s	**9.73s**	5.4
WIKIPEDIA	202.5MB	387.1MB	2.48s	101.55s	**104.04s**	3.60s	2.97s	3.61s	96.83%	3.25s	**13.46s**	7.7

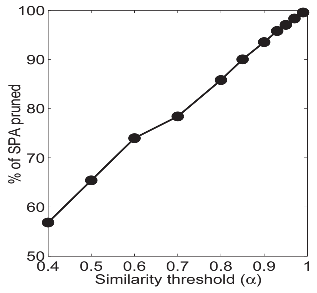
Table 2: Experimental Results (Real Unweighted Datasets); $\alpha = 0.95$.



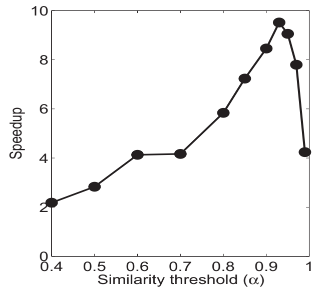
(a) Number of Clusters



(b) Query Time*

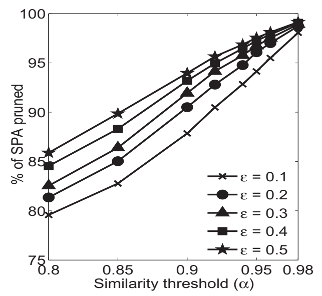


(c) % of SPA Pruned

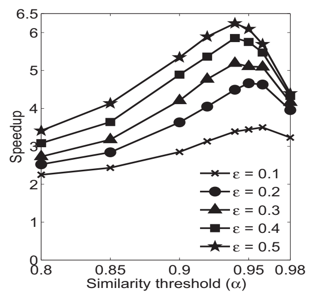


(d) Speed Up

Figure 5: Vary similarity threshold α (FACEBOOK FRIENDSHIP). *Figure 13a shows a zoomed version.



(a) % of SPA Pruned



(b) Speed Up

Figure 6: Vary α (FACEBOOK WALL).

As discussed in Section 2.1, the similarity threshold α controls the clustering result and so it plays an important role in the FVF framework. We have conducted an experiment using the FBFRIEND dataset in which we varied α's value. Figure 5 shows the results[5]. Recall that α is a similarity requirement between $G_\cup$ and $G_\cap$ of a cluster. A larger α implies that snapshots in a cluster have to be more similar. This results in more clusters (Figure 5a) with fewer snapshots in each cluster. Figure 5b shows the query processing time of FVF, with a time breakdown of its key components. As α increases, we see that *Find Time* increases. This is because there are more clusters under a larger α and thus there are more representative graphs ($G_\cap$'s and $G_\cup$'s) on which FVF runs SPA to obtain representative solutions. The *Find Time* curve, therefore, shares a similar shape with the curve in Figure 5a. With a larger α, the representative graphs of a cluster are more similar to the snapshots in the cluster. It is therefore more likely that the solutions on the snapshots can be deduced through verification and fixing by applying the lemmas. Figure 5c shows that more SPAs on snapshots are pruned when α increases. This also explains why the *Residual-SPA Time* decreases when α increases (Figure 5b). The *VF Time* and the *Decompression Time* are relatively small for the FBFRIEND dataset. Verification-and-fixing is performed by applying the lemmas, which involves mostly inexpensive operations except for the two SPAs that are occasionally executed on $G_\cup$ (lines 17-18 of Algorithm 1). This cost is not too expensive because only about 10% of the time do we need to run the two SPAs (for $\alpha = 0.95$), and when they are needed, they are done only once for the whole cluster. The overall speedup is shown in Figure 5d. We observe that FVF is about 2 times faster than Baseline in the worst case ($\alpha = 0.4$) in which all graphs in the EGS are grouped into one single cluster. The optimal case occurs when $\alpha = 0.92$ for which the speedup is 9.8.

[5]α starts at 0.4 because there is only one cluster at $\alpha = 0.4$. Further reducing α, therefore, has no effect.

Similar results are observed from the experiments on the other two real datasets (figures are omitted for space reasons). For example, the optimal values of α for YOUTUBE and WIKIPEDIA are 0.90 and 0.96, respectively. For YOUTUBE, FVF achieves a speedup of 2.7 (worst-case) to 5.8 (optimal α); For WIKIPEDIA, the speedup ranges from 2.7 (worst-case) to 8.1 (optimal α). We remark that although some fine-tuning is needed to determine the optimal α value, it is not critical that the optimal value be used. For example, Figure 5d shows that we can obtain a significant 4-time speedup even a far-from-optimal α (say 0.7) is chosen.

We have also evaluated our FVF framework using an EGS with weighed graphs, FBWALL. The EGS contains 365 directed graphs. Graph vertices represent Facebook users, the weight $w(a, b)$ of an edge (a, b) gives the number of messages user u has posted on user v's wall in the past year. The pre-processing time is 48 seconds. Figure 6 shows the results of the experiments in which we vary the similarity threshold α with different tolerance ε values. The results show that the behavior of FVF is very similar to that for the unweighted case (Figures 5c and d). In general, FVF achieves a speedup of at least two times over the baseline algorithm (which runs Dijkstra's algorithm on each snapshot). When we vary the tolerance ε from 0.1 to 0.5, the average speedup increases. The best α is observed around 0.94, which shows that the optimal α value is quite insensitive to the tolerance.

4.2 Summary of Other Experimental Results

Due to space constraints, we put the experimental results on synthetic data and processing closeness centrality queries in the appendix. For the synthetic data experiments (Appendix D.2), we have implemented a synthetic data generator for creating EGSs with different properties (e.g., with different graph sizes, evolving ratios). The conclusion drawn from the synthetic data experiments of varying the similarity threshold α is consistent with that of the real data experiments above. When we vary (i) the number of snapshots, (ii) the graph size, (iii) the evolving rate, (iv) the average degree, and (v) the ratio of edge insertions and deletions between successive snapshots, FVF consistently outperforms Baseline in all cases. For experiments on closeness centrality queries (Appendix D.3), FVF registered 2 to 10 times speedup over the baseline approach (which runs SSSP on each snapshot).

5. RELATED WORK

In recent years, a plethora of work has focused on efficient methods of managing and querying vast volumes of large graphs. One branch of work on graph query processing focuses on efficient algorithms and data structures for evaluating distance-based queries (e.g., shortest-path queries) and reachability queries [6] on a very large graph. For example, in order to efficiently evaluate shortest-path queries, various shortest-path indices have been developed (e.g., [15, 16, 19]). With those indices, a shortest-path query could be evaluated without accessing vertices that are irrelevant to the results. Other than building indices, some algorithms precompute distance information from the input graph so as to carry out goal-directed search at run-time (e.g., [5, 10]). Those techniques are designed to support efficient query processing on a *single* large graph (e.g., a single snapshot of a social network) but not *a collection of graphs*. Our FVF framework indeed can incorporate those algorithms. For example, if precomputed distance information such as "landmarks" [5] is available, we could choose a goal-directed shortest-path algorithm such as A* algorithm instead of Dijkstra's algorithm as our SPA. Since our experimental results show that the FVF framework is able to prune more than 95% of SPAs, a significant speedup against baseline is still expected, no matter BFS or another algorithm is used as the SPA.

A graph database D is usually used when a collection of graphs is available. However, graph databases usually support *graph queries* (e.g., sub/super-graph query) only. A sub-graph query [12, 17] specifies a query graph G_q and retrieves all graphs in D that contain G_q. A super-graph query [18] can be defined in a similar fashion. The crux of efficient processing of graph queries on a graph database is to determine which graphs in D satisfy the query without accessing the graphs in D that do not belong to the result. In order to achieve that, the filtering-and-verification framework is often used. The principle of the filtering-and-verification framework is to build an index structure to index the features of each graph; and use the index to filter irrelevant graphs during query processing. While our FVF framework has similar favor with the filtering-and-verification framework because both target to minimize the number of expensive graph operations during query processing (e.g., we aim to reduce the number of BFS/Dijkstra's executions while [12, 17, 18] aim to reduce the number of graph isomorphism testings), they are very different. For example, the answer of an EGS query essentially involves all graphs in the EGS, which is not the case in answering a graph query on a graph database.

Processing queries on an EGS is different from processing queries on a time-dependent graph [4]. A time-dependent graph is a graph whose nodeset is fixed and edge weights are defined by a time-dependent function. Thus, it cannot fully capture a dynamic world because its graph structure does not evolve over time. Furthermore, queries on a time-dependent graph only finds answers from a particular snapshot. In contrast, we are interested in queries that find answers on all snapshots. Processing queries on EGSs is also different from mining evolving graph streams [9]. The work in [9] focuses on mining sub-graphs that undergo significant changes over a (small) window of consecutive graphs. In this work, we focus on processing queries on the whole historical collection of snapshot graphs. Our work is also different from the work on query processing on dynamic graphs (e.g., [2]), which focuses on efficient methods that update the shortest path trees when a graph has changed. So, when applied to an EGS, that algorithm is still necessary to be invoked once per snapshot. Finally, we are aware of work related to graph measurements on dynamic graphs (e.g., [8]). These studies focus on "what" kind of graph measures should be used in dynamic networks instead of "how" such measures could be efficiently computed.

6. CONCLUSIONS

In domains like social networks, data evolution could be captured by a sequence of graphs. Graphs of this kind are usually large, numerous, and gradually evolving. We capture these evolving graphs in Evolving Graph Sequences (EGSs). In this paper, we demonstrated that interesting information can be obtained by posing queries on the various EGSs and we discussed how to store and query them. Our experimental results show that interesting information can be unveiled from EGSs and queries could be efficiently evaluated using our techniques.

7. REFERENCES

[1] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[2] E. Chan and Y. Yang. Shortest path tree computation in dynamic graphs. *IEEE Transactions on Computers*, 58(4):541–557, 2009.

[3] D. G. Corneil, M. Habib, and C. Paul. Diameter determination on restricted graph families. In *In WG 98. 24th International Workshop on Graph Theoretic Concept in Computer Science*, pages 192–202. Springer, 1998.

[4] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *EDBT*, pages 205–216, 2008.

[5] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[6] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*, pages 813–826, 2009.

[7] R. Kumar, J. Novak, and A. Tomkins. Structure and evolution of online social networks. In *KDD*, pages 611–617, 2006.

[8] K. Lerman, R. Ghosh, and J. H. Kang. Centrality metric for dynamic networks. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 70–77, 2010.

[9] Z. Liu and J. X. Yu. Discovering burst areas in fast evolving graphs. In *DASFAA (1)*, pages 171–185, 2010.

[10] J. Maue, P. Sanders, and D. Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *ACM Journal of Experimental Algorithmics*, 14:2:3.2–2:3.27, 2009.

[11] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the Flickr social network. In *SIGCOMM Workshop on Social Networks (WOSN'08)*, pages 25–30, 2008.

[12] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *In Symposium on Principles of Database Systems*, pages 39–52, 2002.

[13] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in Facebook. In *SIGCOMM Workshop on Social Networks*, pages 37–42, 2009.

[14] S. Wasserman and K. Faust. *Social Network Analysis: Methods and Applications.* Cambridge Univ. Press, 1994.

[15] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD Conference*, pages 99–110, 2010.

[16] Y. Xiao, W. Wu, J. Pei, W. W. 0009, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, pages 493–504, 2009.

[17] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.

[18] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.

[19] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1):340–351, 2010.

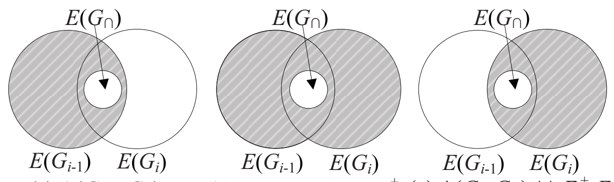
APPENDIX

A. PSEUDO-CODE

Algorithm 2: The Clustering Algorithm.

```
    Input  : EGS = $\{G_1, \cdots, G_n\}$, Similarity threshold $\alpha$
    Output: Clusters $\{C_1, \cdots, C_j\}$, Representative graphs
            $\{(G^1_\cap, G^1_\cup), \cdots, (G^j_\cap, G^j_\cup)\}$
1  $j \leftarrow 1; C_j \leftarrow \{G_1\}; G^j_\cap \leftarrow G_1; G^j_\cup \leftarrow G_1$
2  for $i \leftarrow 2$ to $n$ do
3      $G^{j\,\prime}_\cap \leftarrow G^j_\cap \cap G_i; G^{j\,\prime}_\cup \leftarrow G^j_\cup \cup G_i$
4      if $ges(G^{j\,\prime}_\cap, G^{j\,\prime}_\cup) \geq \alpha$ then
5          $C_j \leftarrow C_j \cup \{G_i\}; G^j_\cap \leftarrow G^{j\,\prime}_\cap; G^j_\cup \leftarrow G^{j\,\prime}_\cup$
6      else  // start building the next cluster
7          $j \leftarrow j+1; C_j \leftarrow \{G_i\}; G^j_\cap \leftarrow G_i; G^j_\cup \leftarrow G_i$
8      end
9  end
10 return $\{C_1, \cdots, C_j\}$ and $\{(G^1_\cap, G^1_\cup), \cdots, (G^j_\cap, G^j_\cup)\}$
```

B. ILLUSTRATION FOR EQUATION 4



(a) $\Delta(G_{i\text{-}1}, G_\cap)$ (b) $\Delta(G_{i\text{-}1}, G_\cap) \cup E_i^+$ (c) $\Delta(G_i, G_\cap) \cup E_i^+\text{-}E_i^-$

C. UPDATED LEMMAS FOR SUPPORTING WEIGHTED GRAPHS

Revised defintions
$w_\cap(e) = \max_{G_i \in C} w_i(e)$ for each edge $e \in G_\cap$ $w_\cup(e) = \min_{G_i \in C} w_i(e)$ for each edge $e \in G_\cup$ $\Delta(G_i, G_\cap) = \{e \in G_i \mid (e \notin G_\cap) \vee (w_i(e) < w_\cap(e)/(1+\varepsilon))\}$ For each edge $e = (p,q) \in \Delta(G_i, G_\cap), \Gamma(e) = \delta_\cup(u,p) + w_i(e) + \delta_\cup(q,v)$

Revised lemmas	
LEMMA 1	*If $\delta_\cup(u,v) = \infty$ then v is not reachable from u in every snapshot $G_i \in C$.*
LEMMA 2	*If $\delta_\cap(u,v)/(1+\varepsilon) \leq \delta_\cup(u,v)$, then $\widetilde{P}_\cap(u,v)$ is a solution for $G_i \in C$.*
LEMMA 3	*If each edge $e \in \widetilde{P}_\cup(u,v)$ exists in G_i and $w_i(e) \leq (1+\varepsilon)w_\cup(e)$, then $\widetilde{P}_\cup(u,v)$ is a solution for G_i.*
LEMMA 4	*If $\Gamma_i^* = \min_{e \in \Delta(G_i, G_\cap)}\{\Gamma(e)\} \geq \delta_\cap(u,v)/(1+\varepsilon)$, then $\widetilde{P}_\cap(u,v)$ is a solution for G_i.*
LEMMA 5	*If $\exists e = (p,q) \in \Delta^*(G_i)$ such that each edge $e' \in \widetilde{P}_\cup(u,p) \cup \widetilde{P}_\cup(q,v)$ exists in G_i and $w_i(e') \leq (1+\varepsilon)w_\cup(e')$, then $\widetilde{P}_\cup(u,p)\|e\|\widetilde{P}_\cup(q,v)$ is a solution for G_i.*

D. FURTHER EXPERIMENTAL DETAILS

D.1 Experiment Setting

All algorithms are implemented in C++. All experiments are run on a Linux machine with 2.83GHz Dual Core Intel(R) with 4GB of memory.

D.2 Experiments on Synthetic Graphs

We run experiments on synthetic datasets to further evaluate our algorithm. We first describe an EGS data generator and then we discuss how the various properties of the data affect our algorithm's performance.

Our synthetic EGS is modeled after typical social networks. The generated snapshot graphs follow some properties of social network graphs, e.g., they are scale-free, evolving, and growing. Our EGS generator takes five parameters:

- d: the average vertex degree.
- V: the number of vertices in the first snapshot.
- n: the number of snapshots in the EGS.
- ir: we add $ir \times V$ vertices to a snapshot to generate the next snapshot.
- k: the ratio $|E^+|/|E^-|$, where E^+ and E^- are the sets of edges added to and removed from a snapshot to generate the next snapshot, respectively.

α	d	V	n	ir	k
0.95	15	100,000	500	0.3%	4

Table 3: Default values used in synthetic data experiments.

To generate an EGS, we first use the BA model [1] to generate a scale-free graph[6] G_1 with V vertices and an average degree d. Then, we repeat the following procedure to generate subsequent snapshot graphs.

1. Add $ir \times V$ new vertices to the current snapshot. For each new vertex v, we generate $m = \frac{d}{2(1-1/k)}$ edges. For each such edge e, one end-point of e is v, and the other end-point is randomly chosen from the vertices of the current snapshot according to the BA model. That is, a vertex with a larger degree has a higher chance of being chosen.
2. Randomly remove $\frac{ir \times V \times m}{k}$ edges from the graph.

We can show that the graphs generated according to the above procedure (1) are scale-free and (2) have the same degree distribution (and hence the same average vertex degree). Due to space limitation, we omit the proofs in this paper.

Next, we present a sensitivity study on the algorithms' performance by varying six variables. These variables and their default values are listed in Table 3. The default values and the ranges over which we will vary the variables are chosen to mimic real datasets. When we vary one variable, the other variables are fixed at their default values. Similar to the experiment on real datasets, 500 random queries are used to obtain the average query-processing times of the algorithms.

D.2.1 Varying Similarity Threshold (α)

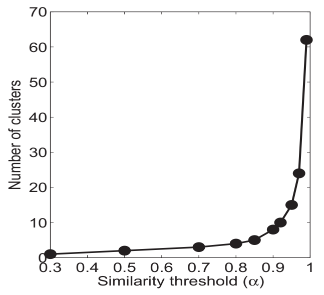
Figure 7 shows the effect of varying the similarity threshold α. The results are consistent with those obtained from the experiments on real datasets. To reiterate, snapshots in a cluster are more similar under a larger α, and so FVF is more effective in pruning SPAs (Figure 7c). This leads to a lower *Residual-SPA Time* (Figure 7b). On the other hand, a larger α results in more clusters (Figure 7a). This leads to a larger *Find Time* (Figure 7b). In the experiment, the optimal α value is 0.95, at which FVF achieves a 6.5 times speedup. Even with a non-optimal α, the speedup is still significant (e.g., the speedup is 3.4 at $\alpha = 0.7$).

D.2.2 Varying Number of Snapshots (n)

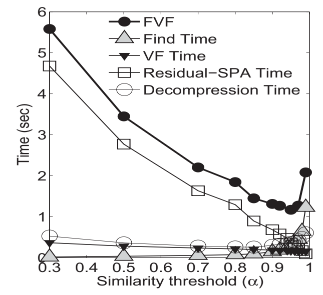
Figure 8 shows the effect of varying the number of snapshots n in an EGS. First, increasing n results in more clusters (Figure 8a) simply because there are more graphs in the EGS. With a fixed α value, clusters remain α-bounded[7] at the same level. The pruning effectiveness of FVF therefore remains more or less the same over different values of n (Figure 8c). Since graphs are growing at a constant rate, increasing n also makes the graphs bigger, particularly for those that appear in the later part of the EGS. Hence, a larger n implies more and bigger graphs. This double impact results in a quadratic growth in the algorithms' execution times (Figure 8b), mostly because the total amount of time spent on SPA grows quadratically w.r.t. n. For large n (say 2,000), the graphs are

[6]A graph is scale-free if the distribution of the vertices' degrees follows a power law: $P(t) \propto 1/t^\gamma$, where $P(t)$ is the probability that a vertex has a degree t, and γ is a constant. Following [1], we set $\gamma = 3$.

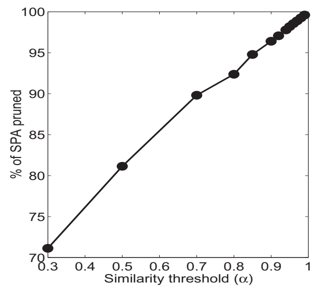
[7]See Definition 3.



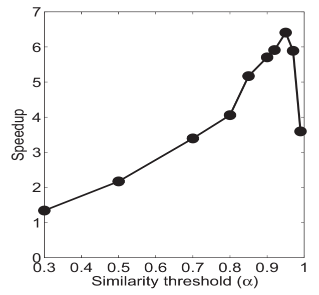
(a) Number of Clusters



(b) Query Time*

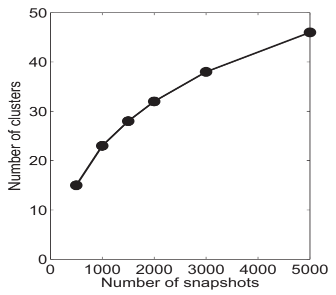


(c) % of SPA Pruned

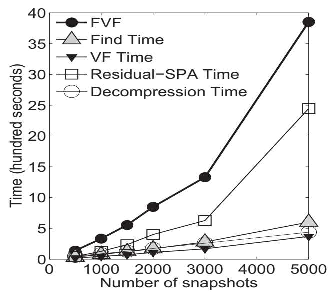


(d) Speed-up

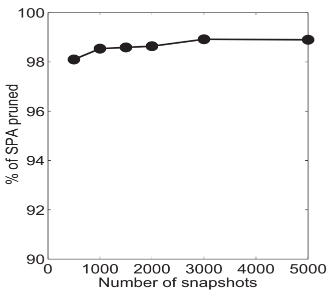
Figure 7: Vary α (Synthetic Data). *Figure 13b shows a zoomed version.



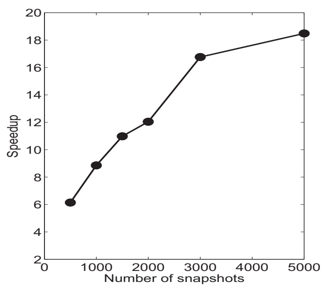
(a) Number of Clusters



(b) Query Time

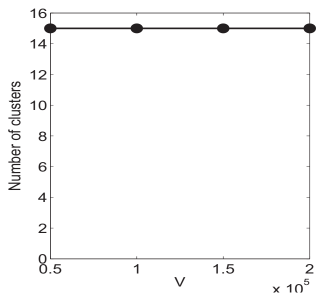


(c) % of SPA Pruned

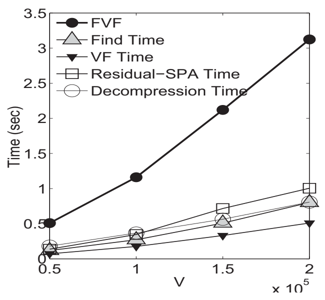


(d) Speed-up

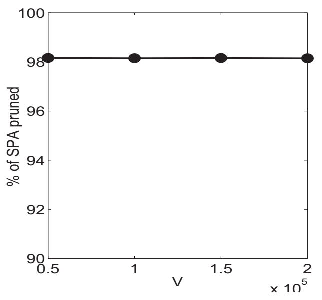
Figure 8: Vary n (Synthetic Data).



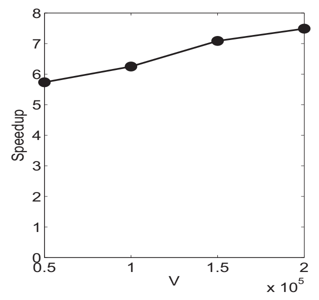
(a) Number of Clusters



(b) Query Time

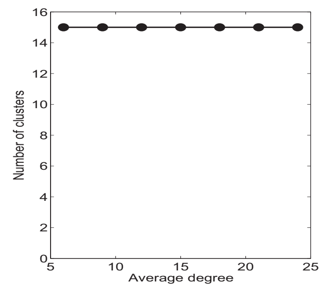


(c) % of SPA Pruned

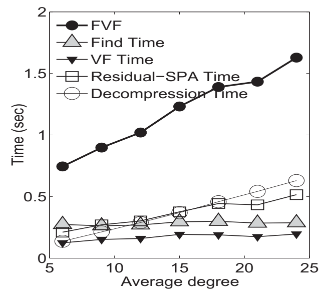


(d) Speed-up

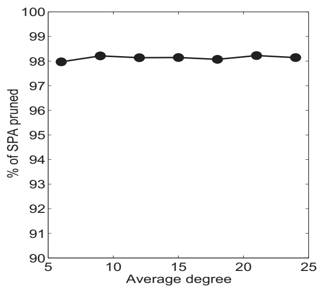
Figure 9: Vary V (Synthetic Data).



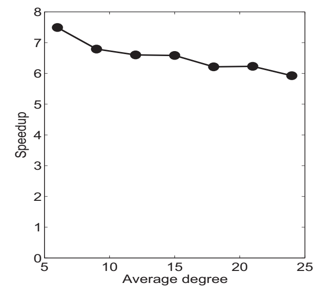
(a) Number of Clusters



(b) Query Time

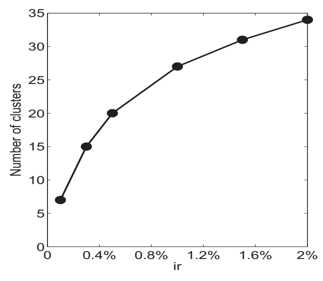


(c) % of SPA Pruned

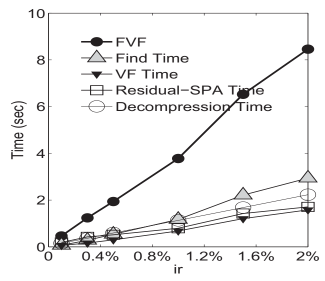


(d) Speed-up

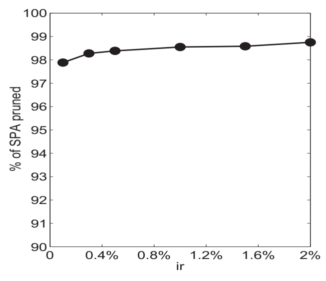
Figure 10: Vary d (Synthetic Data).



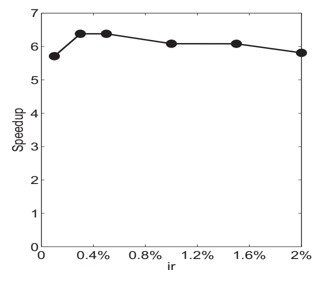
(a) Number of Clusters



(b) Query Time

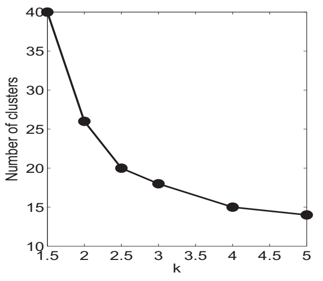


(c) % of SPA Pruned

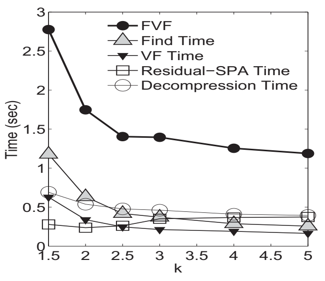


(d) Speed-up

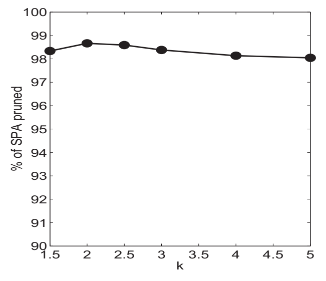
Figure 11: Vary ir (Synthetic Data).



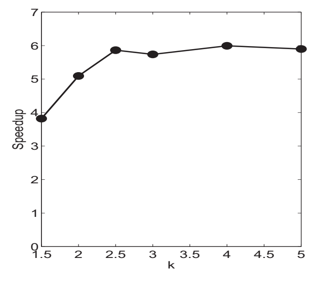
(a) Number of Clusters



(b) Query Time



(c) % of SPA Pruned



(d) Speed-up

Figure 12: Vary k (Synthetic Data).

so numerous and big that the time spent on executing SPA becomes a dominating factor. Since the objective of FVF is to reduce the number of SPA executions, a dominating SPA cost makes the advantage of FVF more pronounced. This results in a larger speedup (Figure 8d). On the other hand, when n is small, the cost of SPA is not so dominating. The advantage of FVF in SPA reduction is watered down by other costs, such as *Decompression Time*. The speedup at small n is thus less significant.

D.2.3 Varying Graph Size (V)

Figure 9 shows the effect of varying V, the number of vertices of the first snapshot graph. With the same number of snapshots and same k and ir values, we get similar clusters and so the number of clusters stays the same as V varies (Figure 9a). Similar to varying n, increasing V makes the graphs larger. Since the runtime of SPA is linear w.r.t. graph size and the number of snapshots remains unchanged, the query-processing time of FVF increases linearly with V. With a fixed α, the pruning effectiveness of FVF is very stable (Figure 9c). Also, we see that the speedup achieved by FVF increases with V because the time spent on executing SPA is becoming more dominating due to bigger graphs. This increase in speedup, however, is not as drastic as that shown in Figure 9d because here we do not get the "double impact" factor we mentioned in Section D.2.2.

D.2.4 Varying Average Degree (d)

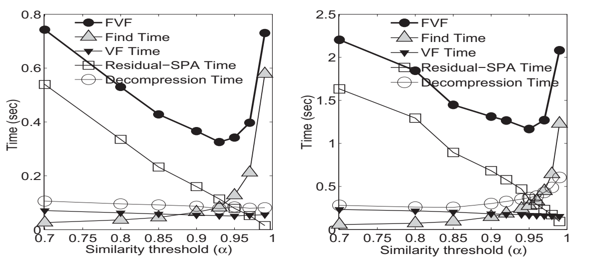
Figure 10 shows the effect of varying the average degree d of the snapshots in the generated EGS. Changing d does not affect the clustering result and so the number of clusters (Figure 10a) and FVF's pruning effectiveness (Figure 10c) do not change much with d. A larger d, however, implies more edges and thus larger graphs. Hence, the query-processing time of FVF increases with d (Figure 10b). The speedup FVF achieves decreases as d increases (Figure 10d). This is because we store the graphs using adjacency lists. A larger degree implies longer lists in the data structure. During decompression, a snapshot graph is reconstructed from the previous one (see e.g., Section 3 Equation 4). In that process, the data structure has to be traversed to locate edges to be deleted. A larger d thus implies a longer *Decompression Time* (Figure 10b). This increase in decompression overhead dwarfs the savings FVF gains.

D.2.5 Varying Insertion Rate (ir)

Figure 11 shows the effect of varying the insertion rate ir of the snapshots in the generated EGS. A higher insertion rate means successive snapshots are less similar. With α fixed, the similarity requirement of the clusters remain the same. As a result, fewer graphs can be grouped into the same cluster before the α-boundedness requirement is violated. Hence, the clusters are smaller and there are more of them (Figure 11a). As k is fixed, a larger ir implies larger snapshot graphs. However, FVF's pruning effectiveness stays relatively stable (Figure 11c) because clusters remain α-bounded at the same level. The speedup FVF achieves drops slightly as ir increases (Figure 11d). This is because with more clusters (Figure 11a), there are more $G_\cup$'s and $G_\cap$'s and the *Find Time* overhead becomes higher (Figure 11b).

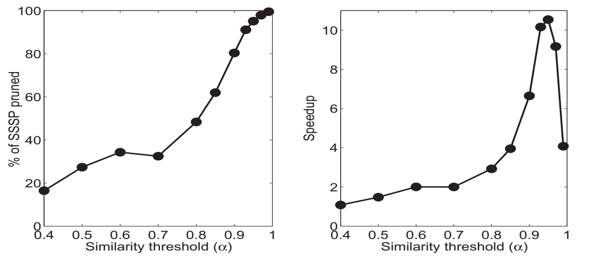
D.2.6 Varying Edge Insertions and Deletions Ratio (k)

Figure 12 shows the effect of varying k, the ratio of edge insertions and deletions, in the generated EGS. Here the insertion rate ir is fixed. So, a larger k means there are fewer edge deletions and successive snapshots are more similar. With the similarity threshold (α) fixed, more graphs can be grouped into the same cluster before the α-roundedness requirement is violated. As a result, the clusters are larger, and there are fewer of them (Figure 12a). The execution time of FVF decreases with k (Figure 12b). The reason is that with fewer clusters, there are fewer $G_\cup$'s and $G_\cap$'s, and the *Find Time* overhead becomes lower (Figure 12b), which also explains why the speedup increases (Figure 12d). FVF's pruning effectiveness stays relatively stable (Figure 12c) because clusters remain α-bounded at the same level.



(a) Zooming Figure 5b (b) Zooming Figure 7b

Figure 13: Zooming in Figures 5b and 7b (α starts from 0.7).



(a) % of SSSP Pruned (b) Speed Up

Figure 14: Vary α (Centrality queries, FBFRIEND).

D.3 Experiments on Computing Centrality

We have conducted experiments on the FBFRIEND EGS. Figure 14 shows the results of the experiments in which we vary the similarity threshold α with $\varepsilon = 1\%$. For example, when $\alpha = 0.95$, our FVF framework pruned many SSSP and registered a 10.2 times speed-up over the baseline approach. We found that $\overline{cc(u)}$ satisfied the error tolerance 71% of the time, and the revised answer $\overline{cc(u)'}$ satisfied the error tolerance 90% of the time. When we vary the error tolerance ε from 0.5% to 5% ($\alpha = 0.95$), the average speedup increases from 6 to 15 (detailed figures are omitted for space reasons).

E. DETERMINING GRAPH DIAMETER

Given a vertex u in a graph G_*, the *eccentricity* of u, denoted by $ecc_*(u)$, is the greatest distance between u and any other vertex. That is, $ecc_*(u) = \max_{v \in G_*} \delta(u, v)$. The *diameter* of G_* ($dm(G_*)$) is the largest eccentricity of any vertex in G_*. That is, $dm(G_*) = \max_{u \in G_*} ecc_*(u)$. A vertex u that gives the largest eccentricity is called a *peripheral vertex*.

To determine $dm(G_i)$ of a snapshot graph G_i, one can run an *all-pairs shortest paths* (APSP) algorithm on G_i to determine the longest shortest-path in G_i. This takes $O(V^3)$ time using the Floyd-Warshall algorithm, which is too expensive for large graphs. Alternatively, one can apply the 2-sweep algorithm (2SA) [3] to obtain an approximate answer. Given a vertex x, let $F_i(x)$ be the set of vertices that are the farthest away from x in G_i (i.e., $\forall y \in F_i(x), \delta_i(x, y) = ecc_i(x)$). Note that $F_i(x)$ can be computed by running an SSSP on G_i. 2SA randomly picks a seed vertex $w \in G_i$, finds a $u \in F_i(w)$ and then a $v \in F_i(u)$. The diameter of the graph is estimated by $ecc_i(u) = \delta_i(u, v)$. The idea is that hopefully u is a peripheral vertex (because it is the farthest away from the seed) and so $dm(G_i) = ecc_i(u)$. One can improve the accuracy of the answer by repeating the procedure with other seeds.

We can use the FVF framework to speed up 2SA when it is applied to a cluster C of k snapshots. The idea is to compute $F_i(x)$ not on each individual snapshot, but on $G_\cup$ and $G_\cap$. We have proved the following theorem:

THEOREM 1. *Given a vertex x, if $ecc_\cup(x) = ecc_\cap(x)$, then $F_\cap(x) \cap F_\cup(x) \neq \emptyset$ and $\forall y \in F_\cap(x) \cap F_\cup(x)$, we have, $y \in F_i(x)$ $\forall 1 \leq i \leq k$.*

Given a seed vertex w, we first run SSSP on both $G_\cup$ and $G_\cap$ to determine $F_\cup(w)$ and $F_\cap(w)$ (and thus $ecc_\cup(w)$ and $ecc_\cap(w)$). Theorem 1 tells us that if $ecc_\cup(w) = ecc_\cap(w)$, then any $u \in F_\cup(w) \cap F_\cap(w)$ (which is guaranteed to exist) is a farthest vertex from w in all the snapshots. Hence, we need only 2 (instead of k) SSSP, one on $G_\cup$ and one on $G_\cap$, to find a peripheral vertex u for all the k snapshots in the cluster. We have conducted experiments on the FBFRIEND EGS and we found that when $\alpha = 0.95$, the condition $ecc_\cup(x) = ecc_\cap(x)$ holds about 50% of the time. Our FVF approach can thus significantly speed up the 2SA algorithm.

F. CASE STUDY

In this section we report a case study that demonstrates how interesting analytical queries can be answered by EGS processing. Figure 1 shows how the shortest-path distance between two users 7058 and 7871 in the Facebook friendship graph changed over a one-year period before they finally became friends. This plot is obtained by running the shortest-path query (7058, 7871) over the FBFRIEND EGS. From the plot, we see that the two users were disconnected before snapshot #178. After that, they got closer gradually until they were linked up at snapshot #365. The result of this EGS query reveals four *key moments* (#178, #186, #304, #365) during the users' friendship development at each of which their distance was shortened by a hop. The query thus helps us pinpoint on the part of the EGS data to drill into in order to better understand what events have occurred at those moments. Consequently, we retrieve the shortest paths found by our FVF algorithm at the four key moments:

Key moment	Shortest-path connecting users 7058 and 7871
#178	7058 → 346 → 9256 → 9264 → 7871
#186	7058 → 3011 → 7098 → 7871
#304	7058 → 51385 → 7871
#365	7058 → 7871

After the key moments are found, we can analyze how friendship evolves by retrieving the preceding moment of each key moment. Take the key moment #186 as an example. We retrieve all shortest-paths between users 7058 and 7871 in snapshot #185 and one of them is:

7058 → 3011 → 7098 → 36931 → 7871

Comparing this shortest path with the one in snapshot #186, we observe that the distance is shortened because of the occurrence of a short-circuiting bridge between users 7098 and 7871 (Figure 2(b)). This observation is found in all the other key moments (#178, #304, #365) of this query. To verify whether short-circuiting is really so predominant, we issued 100 more random shortest-path queries on the EGS and found that 84% of the path-shortening development fall into the short-circuiting bridge case and the rest fall into the *disjoint path* case (Figure 2(a)).

We remark that the case study above would not have been possible if we find shortest paths only on a single snapshot instead of on a complete EGS. This shows that our FVF framework provides a powerful tool for evolving graph analysis.

Acknowledgments

Eric Lo was supported by Grant PolyU 525009E from Hong Kong RGC. Reynold Cheng was supported by the Research Grants Council of Hong Kong (RGC Project HKU 711309E). We would like to thank the reviewers for their insightful comments.