Top-k Durable Graph Pattern Queries on Temporal Graphs

Konstantinos Semertzidis and Evaggelia Pitoura

Abstract—Graphs offer a natural model for the relationships and interactions among entities, such as those occurring among users in social and cooperation networks, and proteins in biological networks. Since most such networks are dynamic, to capture their evolution over time, we assume a sequence of graph snapshots where each graph snapshot represents the state of the network at a different time instance. Given this sequence, we seek to find the top-k *most durable matches* of an input graph pattern query, that is, the matches that exist for the longest period of time. The straightforward way to address this problem is to apply a state-of-the-art graph pattern matching algorithm at each snapshot and then aggregate the results. However, for large networks and long sequences, this approach is computationally expensive, since all matches have to be generated at each snapshot, including those appearing only once. We propose a new approach that uses a compact representation of the sequence of graph snapshots, appropriate time indexes to prune the search space and strategies to determine the duration of the seeking matches. Finally, we present experiments with real datasets that illustrate the efficiency and effectiveness of our approach.

1 Introduction

Recently, increasing amounts of graph structured data are being generated from a variety of sources, such as social, citation, computer and biological networks. Almost all such real-world networks evolve over time. The analysis of their evolution is important for our understanding of the networks and may reveal interesting information. It also finds a wide spectrum of applications ranging from social network marketing to virus propagation and digital forensics.

In this paper, we look into finding the most persistent matches of an input pattern in the evolution of such networks. In particular, we assume that we are given the history of a node-labeled graph in the form of graph snapshots corresponding to the state of the graph at different time instances. Given a query graph pattern P, we address the problem of efficiently finding those matches of P in the graph history that persist over time, that is, those matches that exist for the longest time, either *contiguously* (i.e., in consecutive graph snapshots) or *collectively* (i.e., in the largest number of graph snapshots). We call the queries that return these matches *durable graph pattern queries*.

Motivation. Locating durable matches in the evolution of large graphs finds many applications. Take for example collaboration and social networks, such as DBLP, Facebook or LinkedIn, where nodes correspond to people and edges indicate relationships such as cooperations, or friendships. Node labels may denote demographics, or other characteristics of the users, such as related venues (for example, schools that the users has attended, or scientific conferences where the user has published). Finding durable matches that follow an input pattern helps us locate the most persistent research collaborations or durable social communities and social positions. It can also assist us in the identification of the essential elements (in the form of node labels) that lead to durable and stable cooperations among teams.

Other types of graphs where durable matches may find applications are complex biological systems such as protein-protein, metabolic interaction and hormone signaling networks where nodes are molecular components and edges relationships between them [1]. Understanding such systems requires a molecular level analysis looking at specific topological subgraphs. For instance, locating durable protein complexes may give insight into repeated motifs that remain stable through the evolution of various protein mechanisms. Durable patterns may also be relevant in viral analysis, where scientist could, for example, be interested in finding durable chains of nucleotides of virus RNA for predicting which genes are prone to mutations.

Durable graph patterns are also useful in the case of graphs modeling network and transportation networks. For example, take a network traffic dataset where nodes represent IP addresses and edges are typed by classes of network traffic [2]. Querying such graphs and locating durable patterns in specific time frames may indicate periodic infiltrations (path queries), denial of service (parallel paths) and malicious spreads (tree queries).

Finally, a problem with graph pattern matching algorithms is that they often return an excessive number of matches [3]. Persistence through time offers a means of discarding transient matches and identifying the ones that are meaningful. It offers a way of ranking the results and presenting to users only the k most durable among them.

Contribution. Although, there has been considerable interest in processing graph pattern queries in static graphs (e.g., [4], [5], [6], [1], [7], [8], [9], [10]), we are not aware of any study on searching for durable graph matches in the history of a graph. There has also been some recent work on historical graph processing but the focus has been on how to efficiently store and reconstruct the snapshots relevant to a query by exploiting among others clustering, operational deltas, and efficient data versioning [11], [12], [13], [14].

• *K. Semertzidis and E. Pitoura are with the Department of Computer Science and Engineering, University of Ioannina, Greece.*
E-mail: {ksemer, pitoura}@cs.uoi.gr

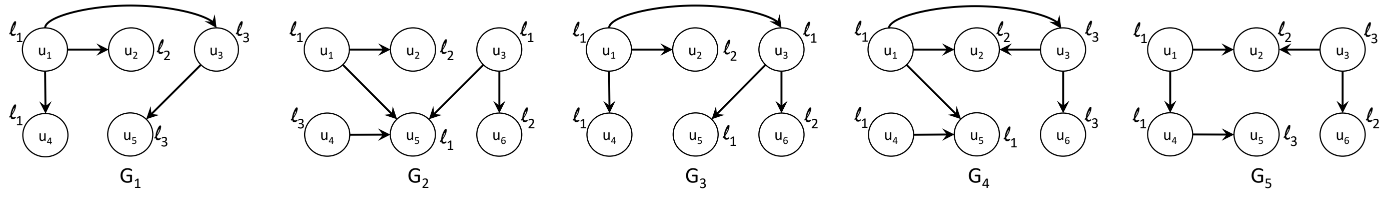


Fig. 1: Example of a temporal labeled graph

Finally, indexes have been proposed for reachability [15] and shortest path [16] queries on non-labeled historical graphs. Instead, in this paper, we propose efficient algorithms and indexes targeting graph pattern queries.

The straightforward approach to processing durable graph pattern queries is to find the matches at each snapshot by applying a state-of-the-art graph pattern algorithm and then aggregate the results. However, even an efficient implementation of this approach incurs large computational costs, since all matching patterns in each snapshot must be identified, even patterns that appear only once. To avoid the computational cost of applying the algorithm per snapshot, we propose an efficient DURABLEPATTERN algorithm.

Our DURABLEPATTERN algorithm identifies the durable matches by traversing a compact representation of the graph snapshots, termed *labeled version graph*. In a labeled version graph (LVG), each node, edge and label is annotated with its *lifespan*, that is, with the set of the time intervals during which the corresponding node, edge and label existed in the graph. An efficient in-memory layout of the LVG allows fast retrieval of neighboring nodes at each snapshot. To prune the number of candidate matches, we introduce neighborhood and path time indexes based on Bloom filters [17], [18]. Finally, our *DurablePattern* algorithm is driven by a ϑ-threshold on the duration of the matches. We exploit various strategies that uses the time-based indexes to efficiently determine an appropriate value for the duration threshold.

We have experimentally evaluated our approach on different datasets and graph pattern queries. Our performance results show the effectiveness of the various aspects of the DURABLEPATTERN algorithm and indicate that it can efficiently process durable queries.

In summary, in this paper, we make the following contributions:

- We formulate the problems of most and top-k durable graph pattern queries.
- We propose a new DURABLEPATTERN algorithm that exploits an LVG-based representation, ϑ-threshold graph exploration search and appropriate Bloom-filter based time indexes to process durable graph pattern queries efficiently.
- We perform extensive experiments on various datasets that show both the efficiency of our DURABLEPATTERN algorithm and the effectiveness of durable graph pattern queries in locating interesting matches.

Roadmap. The rest of this paper is structured as follows. In Section 2, we formally define the durable graph pattern matching problem. In Section 3, we provide the general outline of our DURABLEPATTERN algorithm and in Sections 4-7, we present in detail its various components. In Section 8, we present an experimental evaluation of our approach. Finally, Section 9 provides a comparison with related work, while Section 10 concludes the paper.

2 PROBLEM DEFINITION

Let Σ be a set of labels. We consider directed (node) labeled graphs $G = (V, E, L)$ where V is the set of nodes, E the set of edges and $L : V \rightarrow \Sigma^*$ a labeling function that maps a node to a set of labels. A graph $G^{'}$ whose nodes and edges are subsets of the nodes and edges of G is called a *subgraph* of G. Given a graph G and a user-specified graph pattern $\mathcal{P}$, a *graph pattern query* asks for all occurrences of $\mathcal{P}$ in G.

Definition 1 (Graph Pattern Matching). Given a graph $G = (V, E, L)$ and a graph pattern $\mathcal{P} = (V_{\mathcal{P}}, E_{\mathcal{P}}, L_{\mathcal{P}})$, a graph pattern query returns all subgraphs $m = (V_m, E_m)$ of G for which there exists a bijective function $f : V_p \rightarrow V_m$ such that for each $v \in V_{\mathcal{P}}$, $L_{\mathcal{P}}(v) \subseteq L(f(v))$ and for each edge $(u, v) \in E_p$, $(f(u), f(v)) \in E_m$. Graph m is called a match of $\mathcal{P}$ in G.

Note, that we use subgraph isomorphism semantics for matching. Further, additional edges may exist between the nodes of the subgraph that matches the pattern, besides the edges appearing in the pattern. Also, since, we allow multiple labels per node, we ask that the labels of the matching node are a superset of the labels of the corresponding pattern node (i.e., $L_{\mathcal{P}}(v) \subseteq L(f(v))$, for each $v \in V_{\mathcal{P}}$).

Most previous research in graph pattern queries looks for matches in a single static graph (e.g., [8], [9]). However, most real world graphs change over time. New nodes and edges are added, and existing nodes and edges are deleted. In addition, new labels may be associated with nodes, and existing labels may be deleted.

In this paper, for simplicity, we assume that time is discrete and use successive integers to denote successive time instants. Let $G_t = (V_t, E_t, L_t)$ denote the *graph snapshot* at time instant t, that is, the sets of nodes, edges and the labeling function that exist at time instant t. A *temporal graph* captures the evolution of the graph over time.

Definition 2 (Temporal Graph). An temporal graph $\mathcal{G}_{[t_i,t_j]}$ in time interval $[t_i, t_j]$ is a sequence $\{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$ of graph snapshots.

An example is shown in Fig. 1 which depicts a temporal graph $\mathcal{G}_{[1,5]}$ consisting of five graph snapshots $\{G_1, G_2, G_3, G_4, G_5\}$.

We say that a subgraph m is a match of a pattern $\mathcal{P}$ in a temporal graph $\mathcal{G}_{[t_i,t_j]}$, if m is a match of $\mathcal{P}$ in at least one graph snapshot G_{t_k} in $\mathcal{G}_{[t_i,t_j]}$. Since, a match may appear in more than one graph snapshot of the temporal graph, we would like to find the most durable among the matches. Let us first introduce some terminology.

We use the term *lifespan* for the validity time of a graph element (i.e., node, edge or label), that is, for the set of time intervals during which the corresponding element exists. For example, the lifespan of edge (u_1, u_3) in Fig. 1 is $\{[1,1],[3,4]\}$. Lifespans are set of time intervals (also known as *temporal elements* [19]) to allow the deletion and re-insertion of a graph element. Given two sets of time intervals $\mathcal{I}_1$ and $\mathcal{I}_2$, their time join $\mathcal{I}_1 \otimes \mathcal{I}_2$ is the set of time intervals that include the time instants that belong to both $\mathcal{I}_1$ and $\mathcal{I}_2$. For example, the join of $\{[1,3],\ [5,10],[12,13]\}$ and $\{[2,7],[11,15]\}$ is $\{[2,3],\ [5,7],[12,13]\}$. Finally, we distinguish between two different notions of duration.

Definition 3 (Duration). Let $\mathcal{I}$ be a set of time intervals. We define the *collective duration* of $\mathcal{I}$, $ldur$, as the number of time instants in $\mathcal{I}$ and the *contiguous duration* of $\mathcal{I}$, $ndur$, as the number of instants in the largest time interval in $\mathcal{I}$. We use dur to refer to both.

For example the collective duration of $\mathcal{I} = \{[1,3],$ $[5,10],[12,13]\}$ is 11, while its contiguous duration is 6. Let us now formally define the lifespan of a match.

Definition 4 (Pattern Match Lifespan). Given a temporal graph $\mathcal{G}_{[t_i,t_j]}$ and a pattern query $\mathcal{P}$, the lifespan, $lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m)$, of a match m of $\mathcal{P}$ in $\mathcal{G}_{[t_i,t_j]}$ is the set $\mathcal{I}$ of time intervals that include all time instants, t_k, $t_i \leq t_k \leq t_j$, such that, m is a match of $\mathcal{P}$ in graph snapshot G_{t_k}.

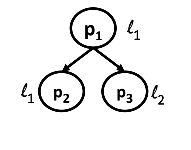
We are now ready to define durable graph pattern queries for temporal graphs. In this case, besides the graph pattern $\mathcal{P}$, the query also includes a set of time intervals, $\mathcal{I}_\mathcal{P}$, that specifies the time periods for which we look for matches. Having $\mathcal{I}_\mathcal{P}$ as part of the query allows us to look for durable matches at specific periods of time within the temporal graph. For example, we may want to locate matches that appear only in snapshots corresponding to weekends, or, to specific seasons of interest.

Definition 5 (Durable Graph Pattern Match). Given a temporal graph $\mathcal{G}_{[t_i,t_j]}$, a graph pattern $\mathcal{P}$ and a set of time intervals $\mathcal{I}_\mathcal{P}$:

- a *most durable graph pattern query* returns the matches m and their lifespans such that $m = \underset{m'\,match\,of\,\mathcal{P}}{\text{argmax}}\ (dur(lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m') \otimes \mathcal{I}_\mathcal{P})$.
- a *top-k durable graph pattern query*, given an integer $k > 0$, returns a set S of k matches and corresponding lifespans such that for all matches m in S, $dur(lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m) \otimes \mathcal{I}_\mathcal{P}) \geq dur(lspan(\mathcal{G}_{[t_i,t_j]}, \mathcal{P}, m') \otimes \mathcal{I}_\mathcal{P})$ for all matches m' not in S.

Based on the definition of duration, we may have contiguous most durable (or, top-k) graph matches and collective most durable (resp. top-k) graph matches.

An example of a graph pattern query is shown in Fig. 2(a) which asks for matches that depict a connection between a node with label l_1 and two other nodes with labels l_1 and l_2. Some matches of this query for $\mathcal{I}_\mathcal{P} = \{[1,5]\}$ in the temporal graph of Fig. 1 are shown in Fig. 2(b). If this query is interpreted as a collective most durable query, it will return only match 1 (and its lifespan), whereas in the contiguous case it will return match 3. A top-2 durable query will return match 1 and either of match 2 or 3, if interpreted as collective, and match 3 followed by either match 1 and 2, if interpreted as contiguous.



(a)

Pattern Nodes	Match 1	Match 2	Match 3
p_1	u_1	u_1	u_3
p_2	u_4	u_5	u_5
p_3	u_2	u_2	u_6
Lifespan	[1,1][3,3][5,5]	[2,2][4,4]	[2,3]
Duration	*ldur*:3 *ndur*:1	*ldur*:2 *ndur*:1	*ldur*:2 *ndur*:2

(b)

Fig. 2: Example of (a) a graph pattern query, (b) the corresponding matches in the temporal graph of Fig. 1.

Algorithm 1 Baseline Algorithm($\mathcal{G}_I$, $\mathcal{P}$, $\mathcal{I}_\mathcal{P}$)

Input: Temporal graph $\mathcal{G}_I$, pattern $\mathcal{P}$, set of intervals $\mathcal{I}_\mathcal{P}$
Output: Most (top-k) contiguous durable matches m

```
1: Hash tables $H$, $H'$
2: $M_0 \leftarrow \emptyset$, $i \leftarrow 1$, $t_p \leftarrow 0$
3: for all $t \in \mathcal{I}_\mathcal{P} \otimes \{I\}$ do
4:     $M_i \leftarrow$ get matches of $\mathcal{P}$ in $G_t$
5:     for each $m \in M_i$ do
6:         if $m \in M_{i-1}$ and $t = t_p + 1$ then
7:             $H[m]$++
8:         else if $H[m]$ not exists then
9:             $H[m] \leftarrow 1$
10:            $H'[m] \leftarrow 1$
11:        else if $H[m] > H'[m]$ then
12:            $H'[m] \leftarrow H[m]$
13:            $H[m] \leftarrow 1$
14:    $t_p \leftarrow t$, $i$++
15: return (all || top-$k$) matches $m$ with the largest $H'[m]$ and
    their lifespan
```

3 The Durable Graph Pattern Algorithm

In this section, we start by describing a baseline approach to processing durable graph pattern queries and then present our *DurablePattern* algorithm.

3.1 Baseline Approach

A straightforward way to process a durable graph pattern query is to first execute the graph pattern query $\mathcal{P}$ at each graph snapshot G_{t_m}, $t_m \in \mathcal{I}_\mathcal{P}$, of the temporal graph using a state-of-the-art graph pattern matching algorithm and then aggregate the results by counting for each match the number of times it appears in the result.

The steps of the baseline approach for finding *contiguous* durable matches are shown in Algorithm 1. We represent each match m as a string $u_1 u_2 ... u_{|V_\mathcal{P}|}$, where u_i, $1 \leq i \leq |V_\mathcal{P}|$, are the nodes of the matched subgraph m ordered following the order of the nodes in $\mathcal{P}$ that each one of them matches. Thus, we reduce graph matching to string matching. Furthermore, to match the resulting strings we use hashing. We maintain two hash tables H and H'. $H[m]$ indicates for each match m the duration of the current largest time interval for which m was found to be a match, while H' the duration of the previous largest interval. We compute the subgraphs that match the input graph pattern $\mathcal{P}$ for each graph snapshot G_t of the temporal graph, for $t \in$

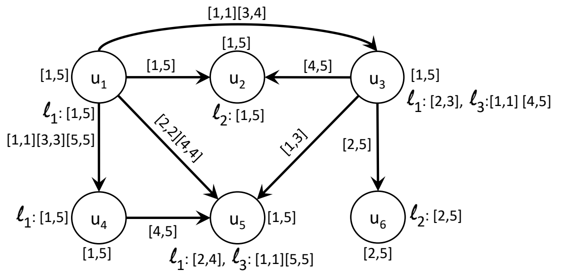


Fig. 3: The LVG of the temporal graph of Fig. 1.

$\mathcal{I}_{\mathcal{P}}$ (line 4). For each match m, the algorithm checks whether it was found in the exact previous time instant and if this is the case it increases $H[m]$ (lines 6–7). Otherwise, if match m is found for the first time, the algorithm initializes both hash tables (lines 9–10), or if match m was previously found, it updates $H[m]$ and $H'[m]$ appropriately (lines 12–13).

To process a *collective* durable graph pattern query, we use just one hash table H and for each time instant that a match m is found, we increase $H[m]$.

Even with these optimizations, the baseline approach is expensive, since we have to retrieve all matches at each and every graph snapshot, even those matches that appear only in just one snapshot. For frequent patterns and long intervals, the number of retrieved matches grows very fast.

3.2 Durable Graph Pattern Matching

We consider a more efficient approach that uses a concise representation of the temporal graph, that we call a *labeled version graph*. The labeled version graph is the union of the graph snapshots where each node, edge and label is annotated by its lifespan.

Definition 6 (Labeled Version Graph). Given a temporal graph $\mathcal{G}_I = \{G_{t_i}, G_{t_i+1}, \ldots, G_{t_j}\}$, its labeled version graph (LVG) is a lifespan annotated directed graph $VG_I = (V_I, E_I, L_I, \mathcal{L}_u, \mathcal{L}_e, \mathcal{L}_l)$ where: $V_I = \bigcup_{t_m \in I} V_{t_m}$, $E_I = \bigcup_{t_m \in I} E_{t_m}$, $L_I = \bigcup_{t_m \in I} L_{t_m}$, $\mathcal{L}_u : V_I \to \mathcal{I}$ assigns to each node $u \in V_I$ its lifespan $\mathcal{L}_u(u)$, $\mathcal{L}_e : E_I \to \mathcal{I}$ assigns to each edge $e \in E_I$ its lifespan $\mathcal{L}_e(e)$ and $\mathcal{L}_l : L_I \to \mathcal{I}$ assigns to each node label $l \in L_I(u)$ its lifespan $\mathcal{L}_l(l)$.

Fig. 3 depicts the labeled version graph of the temporal graph of Fig. 1.

In addition to the LVG, we also maintain a *time-label* index, ViLa, which allows constant time retrieval of all nodes having a specific label at a given time instant. We will refer to LVG augmented with this time index as ViLaG. ViLaG is our basic data structure.

The main steps of our durable graph pattern algorithm are outlined in Algorithm 2. The algorithm runs on the labeled version graph and is driven by a duration threshold ϑ. It consists of two phases. The first phase (lines 2–5) computes the candidate matching nodes in V_I for each node $p \in V_{\mathcal{P}}$ in the given set of time intervals $\mathcal{I}_{\mathcal{P}}$ and stores them in a set $C(p)$. We call the procedure of generating the candidate nodes FilterCandidates. The resulting candidate set $C(p_1) \times \ldots \times C(p_{|V_{\mathcal{P}}|})$ determines the overall search space of the algorithm. To avoid a sequential scan of all nodes of a large graph that would result in a total search space of $\prod_{n=1}^{|V_{\mathcal{P}}|} |C(|V_I|)|$, we use ViLa. ViLa returns for each pattern node p the graph nodes that have the same label as p in at least one time instant in $\mathcal{I}_{\mathcal{P}}$.

In the second phase (lines 6–9), we search for a match. The algorithm exploits the fact that a feasible match of a pattern node must have the appropriate descendants and ascendants nodes. Candidates nodes that do not meet these criteria are pruned and not examined by the algorithm. The check of the appropriate descendants and ascendants is conducted by the RefineCandidates procedure. Then, Algorithm 2 traverses the remaining candidate nodes by calling the recursive DurableGraphSearch procedure. The search procedure uses the candidate sets and searches in a depth-first manner for matches with duration at least ϑ. If no solution is found, the algorithm reduces ϑ by calling RecomputeDuration and searches for matches with a smaller duration until a solution is found.

Algorithm 2 DurablePattern Algorithm(VG_I, $\mathcal{P}$, $\mathcal{I}_{\mathcal{P}}$, P_{type})

Input: Version graph VG_I, pattern $\mathcal{P}$, set of intervals $\mathcal{I}_{\mathcal{P}}$, query type P_{type} (i.e., most/top-k, collective/contiguous)
Output: Durable matches m of type P_{type}

```
1: $\vartheta \leftarrow$ InitializeDuration($P_{type}$), $M \leftarrow \emptyset$
2: for each $p \in V_{\mathcal{P}}$ do
3:     $C(p) \leftarrow$ FilterCandidates($VG_I$, $\mathcal{P}$, $p$, $\mathcal{I}_{\mathcal{P}}$)
4:     if $C(p) = \emptyset$ then
5:         return $\emptyset$
6: while not ($M.found()$ or $\vartheta = 0$) do
7:     $C \leftarrow$ RefineCandidates($VG_I$, $\mathcal{P}$, $C$, $\vartheta$, $\mathcal{I}_{\mathcal{P}}$, $P_{type}$)
8:     DurableGraphSearch($VG_I$, $\mathcal{P}$, $C$, 1, $\vartheta$, $\mathcal{I}_{\mathcal{P}}$, $M$, $P_{type}$)
9:     $\vartheta \leftarrow$ RecomputeDuration($P_{type}$, $\vartheta$)
10: return $M$
```

In-memory storage of the temporal graph. Our basic data structure for the in-memory storage of the temporal graph is LVG. For storing lifespans, we use bit arrays. Assume without loss of generality, that the maximum number of graph snapshots is T. Then, a lifespan, i.e., set of intervals, $\mathcal{I}$ is represented by a bit array B of size T, such that $B[i]$ = 1 if time instant i belongs to $\mathcal{I}$ and 0, otherwise. For example, for T = 16, the bit representation of $\mathcal{I} = \{[2, 4], [9, 10], [13, 15]\}$ is 0111000011001110. This representation supports an efficient implementation of join. In particular, let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two set of intervals and B_1 and B_2 be their bit arrays. Then, $\mathcal{I}_1 \otimes \mathcal{I}_2$ can be computed as B_1 logical-AND B_2.

For the in-memory storage of the LVG, we maintain an array of nodes, where each node is associated with a key-value structure that maps each node u to its neighboring nodes along with a bit array of size T. The bit array keeps the lifespan of each edge during T. The required storage for these adjacency lists is $|E_I|T$, since we have to store for all edges E_I in VG_I their lifespan of size T. We also maintain for each node u its labels during T. A bit array of size T is associated with each label l of u to represent the lifespan of this label during T. The required storage for label lifespans is $|\Sigma_I|T$, where Σ_I is the set of all labels of V_I. Fig. 4(a) depicts the in-memory layout of LVG.

ViLa, our basic time index, consists of two levels. The first level is an array of size T where each position i refers to a time instant t_i and links to a set of labels L. Each label l in this set links to the set of nodes that are labeled with l

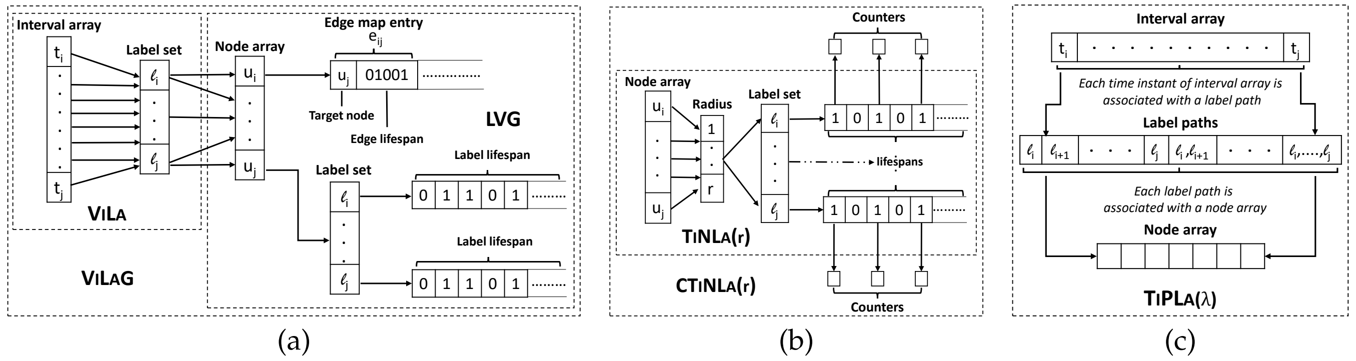


Fig. 4: In-memory layouts of (a) ViLaG, (b) time-neighborhood-label indexes, and (c) time-path-label index

at t_i. Thus, the index has at most $|V_I||\Sigma_I|T$ nodes. Fig. 4(a) depicts the in-memory layout of ViLa.

The total time for constructing ViLaG from scratch is $O(|V_I| + |E_I| + |\Sigma_I|T)$, that is the time needed to create both LVG and ViLa. We incrementally update ViLaG for each newly inserted edge in a time instant t, by updating the edge map entry and label set of the interval array. The bit array structure for lifespans and the map structure for the adjacency lists allow us to perform each update operation in constant time.

Refining the Algorithm

In the following sections, we refine the basic steps of Algorithm 2 to address the following issues:

1) reduce the size of the candidate set $C(p)$ for each node p and efficiently retrieve this set using appropriate time indexes,
2) determine appropriate values for the duration threshold,
3) efficiently search in the labeled version graph, and
4) refine the overall search space.

4 Time Indexes

Besides our basic index, ViLa, we introduce additional time indexes to speed up matching. We explore two types of indexes, namely neighborhood and path indexes. We also present compressed representations of both.

4.1 Neighborhood and Path Time indexes

The *time-neighborhood-label*, TiNLa(r), index maintains for each node $u \in V_I$ information about the labels of its neighbors at distance at most r at each time instant, that is, the neighbors that are at most r hops away from u. For example, TiNLa(1) maintains information for neighbors at distance 1, that is, for the immediate neighbors of each node. Specifically, TiNLa(r) maintains for each $u \in V_I$ a set of labels. Each label l is associated with r bit arrays of size T, where T is the number of graph snapshots. The i-th position of the j-th array, $1 \leq j \leq r$, is set to one, if at least one neighbor of u at distance j has label l at the corresponding time instant t_i. TiNLa(r) is depicted in Fig. 4(b).

We also consider replacing the bit arrays associated with each label l with counter arrays where the i-th position of the j-th counter array is equal to the number of neighbors of u at distance j that have label l at time instant t_i. We call this variation, *counter-time-neighborhood-label* or CTiNLa(r) index. CTiNLa(r) is shown in Fig. 4(b).

Furthermore, we explore a compact representation of TiNLa and CTiNLa using Bloom filters. Bloom filters are probabilistic data structures often used to represent a set A of n elements to support membership queries [17], [18]. The idea is to allocate an array of F bits, initially all set to 0, and then choose l independent hash functions, h_i, $1 \leq i \leq l$, each with range $\{1, \ldots, F\}$. The hash functions are applied to each element a of the set A and the bits at positions $h_1(a)$, $\ldots$, $h_l(a)$ are set to 1. To check whether an element b belongs to the set, the hash functions are applied to b and the bits at positions $h_1(b)$, $\ldots$, $h_l(b)$ are checked. If at least one of the bits is 0, then we are certain that b does not belong to A. Otherwise, we conjecture that b belongs to A, but there is a certain probability that this is not the case. This is called a *false positive*. Parameters F and l are chosen such as the false probability rate is acceptable (usually $\leq 1\%$).

For the probabilistic representation of TiNLa(r), denoted TiNLaB(r), we maintain a Bloom filter with information for the labels of neighbors at distance r of node u. Specifically, we insert in each Bloom filter a set that consists of pairs (t, l) where l is a label of a neighbor of node u at distance r at time instant t.

A more compact representation of CTiNLa(r), denoted CTiNLaB(r), is achieved by using counting Bloom filters [18]. In this case, each entry of the filter array is not a single bit but a small counter. When an element (t, l) is inserted in the filter, the corresponding counters are incremented by one. When we want to find the number of neighbors of node u that have a specific label l at time instant t, again, we apply the hash functions. We retain the smaller of the filter counters as an estimate of the number of neighbors.

TiNLa(r) requires storage at most $r\,|V_I|\,|\Sigma_I|\,T$, since for all nodes in the worst case we have to store for each label a bit array of size T. Using Bloom filter, TiNLaB(r) requires storage at most $r\,|V_I|\,F$, where F is the average size of the Bloom filer. We do not use the same size Bloom filters for all nodes. Instead, we estimate the size of each Bloom so as to achieve a specified false positive rate. In the case of CTiNLa and CTiNLaB, we store an integer value for each label instead of a bit array.

Finally, we consider a *time-path-label* or TiPLa(λ) index, in which we maintain for each time instant t in T and each node $u \in V_I$, the label paths of length up to λ starting from u at t. TIPLA(λ) enumerates all paths up to a maximum length λ using BFS. The number P_l of possible label combinations is very large: $P_l = (\sum_{r=1}^{|\lambda|} \frac{|L|!}{(|L|-r)!})$, but experimentally λ = 3 proved a good choice. Paths are stored as strings. For example, for $\lambda = 2$ the label path $l_1 \rightarrow l_2 \rightarrow l_3$ is stored as key $[l_1, l_2, l_3]$. Each key is associated with the set of nodes that are the sources of the corresponding path. For instance, key $k = [l_1, l_2, l_3]$ is associated with the nodes that are labeled with l_1, connected to a node labeled with l_2, that is in turn connected to a node labeled with l_3. TIPLA is shown in Fig. 4(c). The required storage is $P_l\ |V_I|\ T$.

For the compact representation of TIPLA, TIPLAB, we maintain a Bloom filter for each node u in which we insert pairs (t, l_{path}), where each l_{path} denotes the label path of length up to λ starting from u at time instant t. The required storage is $|V_I|\ F$, where F is the average size of the Bloom filters.

Construction Time. The total time for constructing TINLA(r) is $O(r(|E_I| + |\Sigma_I|T))$. To achieve this, we construct TINLA(1) by checking for each node the labels of its 1-hop neighborhood; this can be constructed in $O(|E_I| + |\Sigma_I|T)$. Then, for each node u and for each r_i-hop, $1 < r_i \leq r$ we retrieve the labels of the (r_{i-1})-hop neighborhood of its adjacency nodes which constitutes the TINLA(r_i) of u. Constructing CTINLA(r) requires the same time as TINLA(r). For TIPLA(λ), for each node we compute all paths of length λ, which requires a total time of $O((|V_I| + |E_I^{\lambda}|)|\Sigma_I|T)$, where $|E_I^{\lambda}|$ is the number of edges that need to be traversed until depth λ. Creating the compressed indexes requires the same time as needed for constructing the uncompressed ones plus the time for applying the hash functions. We evaluate the compression rate and the performance of the compressed filters in Section 8.

4.2 Computing and Filtering Candidate Nodes

The indexes (either, the uncompressed or the compressed versions) are used to compute and filter the candidate nodes. VILA is first used to get the initial set of candidate matches of a pattern node. In the case of neighborhood-based indexes, the indexes are used to retain a node u as a candidate match of a pattern node p, only if the neighborhood subgraph of u is sub-isomorphic to that of p in at least one time instant in $\mathcal{I}$. To enforce this requirement, we use TINLA(r) to remove a node u from the candidate set $C(p)$, if u does not have a matching distance r neighbor whose label lifespan intersects in at least one time instant in $\mathcal{I}$ with the label of a corresponding distance r neighbor of p. CTINLA(r) performs a refined test in which we also take into account the multitude of the labeled nodes in the r-neighborhood, thus the candidate sets produced by CTINLA(r) are subsets of the corresponding candidate sets produced by TINLA(r), i.e., $C(p)^{CTiNLa(r)} \subseteq C(p)^{TiNLa}$.

When TIPLA is used, we first compute for each pattern node p all label paths starting from p up to length λ. Then, for all label paths $L_{path}(p)$ of p and for each time instant of $\mathcal{I}$, we use TIPLA to retrieve the set of nodes that are the source nodes of each $l_{path} \in L_{path}(p)$. Since, a feasible match of p must be a node that is the source node of all paths in $L_{path}(p)$, we intersect the retrieved sets in each time instant.

Generally, for each candidate set $C(p)$ of $p \in V_{\mathcal{P}}$, it holds:

$$|C(p)^{TiPLa(\lambda)}| \leq |C(p)^{TiNLa(r)}| \leq |C(p)^{ViLa}|,\ \lambda = r$$

However, there is no direct relationship between the candidate sets of CTINLA(r) and TIPLA(λ), with $\lambda = r$. Instead, the sizes of the corresponding candidate sets depend on the pattern query. For example, for a pattern query with a node p connecting to two other nodes that have the same label l, TIPLA will return as candidates for p, even nodes that have just a single path l, whereas CTINLA will prune such nodes and return only nodes that have at least two neighbors with label l. On the other hand, for a pattern query where p is connected with a node with label l_1 which in turn is connected with a node with label l_2, CTINLA(2) will return as candidate a node that has a neighbor with label l_1 at distance 1 and a neighbor with label l_2 at distance 2, even if these two nodes are not connected with each other, while TIPLA will prune such nodes.

5 DURATION THRESHOLD

Our durable graph pattern matching algorithm (Algorithm 2) is driven by a threshold duration ϑ, in the sense that the algorithm searches for matches whose lifespan has duration at least ϑ, thus ϑ determines the order of searching for possible matches. The value of ϑ is set to an appropriate initial value (line 1) and in refining of candidates (REFINECANDIDATES) and searching for subgraphs (DURABLEGRAPHSEARCH), we look for subgraphs with duration at least ϑ.

The first strategy for determining ϑ, called MIN, initializes ϑ with 1, that is the minimum possible value, looking for matches that appear in at least one time instant. While we search for matches (DURABLEGRAPHSEARCH), ϑ is updated accordingly. For a *most* durable query, ϑ is updated such as to be equal to the duration of the most durable match found so far. For a *top-k* durable query, ϑ is updated so as to be equal to the duration of the k-th match found so far. With the MIN strategy, in the first calls of the recursive durable graph search procedure, the algorithm explores edges that have a short duration compared to the actual duration of a potential match. Thus, the algorithm pays a cost for exploring many matches of small duration.

The next two strategies, called MAXRANK and MAXBINARY, follow a different approach and initialize ϑ to a value that is close to the actual duration of the seeking match(es). This approach reduces the number of candidate matches, since fewer subgraphs qualify as such. Since the actual duration of the durable matches is not known, we use the time indexes to determine the maximum possible duration of a match and use this value to initialize ϑ. If no matches are found with this estimated duration, we recompute another smaller value for ϑ.

To this end, we introduce the ranking structure $Rank$, which maintains a ranking of candidates for each pattern node p based on their duration. In particular, $Rank^{\theta}(p)$ includes the nodes that are candidate matches of p with duration at least θ ranked by duration. To construct

$Rank^{\theta}(p)$, we use the time indexes ViLa, TiNLa (CTiNLa) and TiPLa during the FilterCandidates procedure. $Rank^{\theta}(p)$ using ViLa refers to a set of nodes that are feasible matches of p and have the same label as p for a duration at least θ. Similarly, the $Rank^{\theta}(p)$ using TiNLa (CTiNLa) refers to a set of nodes that have the correct adjacency and label as p for a duration at least θ. Finally, the $Rank^{\theta}(p)$ using TiPLa refers to a set of nodes that have the required paths as p for a duration at least θ.

The maximum duration of a match cannot be larger than the minimum value among the maximum durations of the candidates for each nodes $p \in \mathcal{P}$. Formally, for each node p, let $\theta_{max}(p)$ be the maximum value of θ for which $Rank^{\theta}(p)$ is not empty. MaxRank and MaxBinary initialize ϑ as:

$$\vartheta = \min_{p \in V_P} \theta_{max}(p) \tag{1}$$

By doing so, the candidate sets that have to be examined are smaller, since we use only candidate nodes with duration greater or equal to ϑ. If no solution is found with duration at least ϑ, a new smaller threshold is determined. The MaxBinary strategy uses binary search for determining the next smaller ϑ value. The MaxRank strategy gets for each node p the maximum θ smaller than the current ϑ for which $Rank^{\theta}(p)$ is not empty and selects as the new ϑ the minimum among these values. In recomputing ϑ, both strategies take also into account the duration of matches found during the previous execution of DurableGraphSearch. This is explained in detail in Section 6.

For a top-k query, for both strategies, we also check whether the combination of candidates nodes with duration at least ϑ produces at least k matches. If this is not the case, we use the largest ϑ value that fulfills this requirement.

Note that at each step we select larger candidate sets including nodes that have candidate duration smaller than the previous threshold. Thus, searches get more expensive as ϑ decreases. In terms of the number of calls to DurableGraphSearch, the algorithm is called at most $|\Theta|$ times, where Θ is the set of distinct values of ϑ that the algorithm uses to find durable matches. For the MaxBinary strategy, Θ is at most logarithmic to the initial value of ϑ.

6 Graph Search

The DurableGraphSearch algorithm (shown in Algorithm 3) searches in a depth-first manner for durable matches with duration at least ϑ.

DurableGraphSearch first checks if the given candidate sets contain isomorphic matches to the given pattern. First, it creates a copy C' of C (line 19), isolates a node u in $C(p_i)$ and treats it as if it were the only node to match pattern node p_i (line 20). Then, a refinement is performed on C', which removes all nodes in $C(p_1), \ldots, C(p_{|V_{\mathcal{P}}|})$ that are not contained in an isomorphic match with u. If the pruning of candidates eliminates all nodes in C', no isomorphic match exists with the current mapping, and the algorithm backtracks. Otherwise, the search procedure is called recursively, passing the subsequent pattern node p_{i+1} until all pattern nodes are examined or refining eliminates all remaining possible matches. The above procedure is performed for each pattern node in $C(p_i)$.

Algorithm 3 DurableGraphSearch(VG_I, $\mathcal{P}$, C, i, ϑ, $\mathcal{I}_{\mathcal{P}}$, M, P_{type})

Input: Version graph VG_I, pattern $\mathcal{P}$, candidates set C, pattern node to be matched i, duration threshold ϑ, set of intervals $\mathcal{I}_{\mathcal{P}}$, matches structure M, query type P_{type}
Output: Solution M of durable graph pattern $\mathcal{P}$ of type P_{type}

```
1: if $i = |V_P|$ then
2:     for each $(p_i, p_j) \in E_{\mathcal{P}}$ do
3:         $\mathcal{I} \leftarrow \mathcal{I}_P \otimes \mathcal{L}_e((C(p_i), C(p_j))$
4:         $\mathcal{I} \leftarrow \mathcal{I} \otimes \mathcal{L}_{C(p_i).label(p_i)} \otimes \mathcal{L}_{C(p_j).label(p_j)}$
5:     if $P_{type} = topk$ then
6:         UpdateTopkState($C, M, \mathcal{I}, \vartheta$)
7:         if $|M| = k$ and $M.durationMin \geq \vartheta$ then
8:             Finish()
9:     else if $P_{type} = most$ then
10:        if $|\mathcal{I}| = \vartheta$ then
11:            UpdateState($C$, $M$)
12:        else if $|\mathcal{I}| > \vartheta$ then
13:            $\vartheta \leftarrow |\mathcal{I}|$
14:            RestoreState($C$, $M$, $\vartheta$)
15:        else
16:            KeepTrack($M$, $|\mathcal{I}|$)
17: else
18:     for each $u \in C(p_i)$ and $u \notin C(p_j)$, j < i do
19:         $C' \leftarrow$ copy of $C$
20:         $C'(p_i) \leftarrow \{u\}$
21:         $C' \leftarrow$ RefineCandidates($VG_I$, $\mathcal{P}$, $C'$, $\vartheta$, $\mathcal{I}_P$)
22:         if $C' \neq \emptyset$ then
23:             DurableGraphSearch($VG_I$, $\mathcal{P}$, $C'$, $i$+1, $\vartheta$, $\mathcal{I}_{\mathcal{P}}$, $M$, $P_{type}$)
```

When a candidate match is found (line 1), an additional check is made (lines 2–4) to ensure that all nodes and edges of the candidate matching subgraph appear in the same time period during $\mathcal{I}_{\mathcal{P}}$. This is achieved by joining both the lifespans of all edges of the matching subgraphs and the lifespans of the labels of their incident nodes.

Next, we present details regarding storing matches across recursive calls. We also discuss how to maintain information for enhancing the selection of the ϑ threshold.

Most durable graph pattern queries. Regardless of the strategy used for selecting ϑ, the algorithm maintains the duration of the best match found so far, let us denote this value as θ_{cur}. Since, our algorithm is using recursion, all recursion calls must be notified when a match is found with a duration larger than θ_{cur}, so as to prune subgraphs with duration less than the new value. In addition, we need to store the new durable matches and delete the ones with duration less than θ_{cur}. UpdateState keeps the current durable matches, while RestoreState removes old matches and keeps the new ones (lines 10 – 14).

We also use the duration of the best match to improve the selection of the new smaller ϑ by the MaxRank or MaxBinary strategies, when DurableGraphSearch finds no matches for a given ϑ. Let us denote with ϑ_{old} the old threshold, with ϑ_{new} the new smaller threshold computed by MaxRank or MaxBinary and with θ_{best}, $\theta_{best} < \vartheta_{old}$, the duration of the best match found by DurableGraphSearch. The new call to DurableGraphSearch is with ϑ = $max\{\theta_{best}, \vartheta_{new}\}$. The reason is that, since we have found at least one match with duration θ_{best}, we should search for matches with a larger or equal duration. The equal duration is needed for locating *all* most durable matches when θ_{best} happens to be the largest possible duration.

Top-k durable graph pattern queries. We maintain a min heap structure M with the top-k matches found so far ordered by their duration. UPDATETOPKSTATE handles this heap. Let θ_{heap} be the minimum duration of any match in the heap and m_{min} be the (top) match in the heap with duration equal to θ_{heap}. The algorithm stores any match in the heap, until the heap becomes full. When the heap is full, m_{min} is replaced by a new match m, if the duration of m is larger than θ_{heap}.

As with most durable graph pattern queries, we use the duration of the matches found so far to improve the selection of the new smaller ϑ, when DURABLEGRAPHSEARCH fails to find k matches with a duration at least ϑ_{old}. Again let ϑ_{new} be the new threshold computed by MAXRANK or MAXBINARY. If the heap is full, the new call to DURABLEGRAPHSEARCH is with $\vartheta = max\{\theta_{heap} + 1, \vartheta_{new}\}$. The reason is that, since we have already found k matches with duration at least θ_{heap}, we should search for matches with a larger duration.

7 REFINE CANDIDATES

Let us now describe the refine procedure outlined in Algorithm 4. Our refine procedure is based on the dual graph simulation technique [20] that was shown in [21] to outperform the commonly used VF2 algorithm [6]. The refine procedure checks for each node p and its candidate node u whether the neighborhood of $p \in V_{\mathcal{P}}$ is sub-isomorphic to that of u in the graph. Specifically, given a set of candidates nodes $C(p)$ of $p \in V_{\mathcal{P}}$, the refine procedure retrieves all its neighbors p' (1–2). Then, for each $u \in C(p)$, it examines if there are any neighbors of u contained in $C(p')$ using TIME_JOIN described next (lines 4–9). If this is not the case, then u is removed from $C(p)$, otherwise its neighbors in $C(p')$ are stored in a temporary set $C'_{p'}$ (lines 6–9). Now, every node in $C(p')$ must be a neighbor of at least one node in $C(p)$. Thus, the candidate set of pattern node p' is updated to contain only the nodes that are neighbors of nodes in $C(p)$ (line 12).

Since we seek durable matches, TIME_JOIN that implements the refinement checks if a candidate node has the required neighbors during $\mathcal{I}_{\mathcal{P}}$. In particular, given a pattern node p' and a graph node u, TIME_JOIN returns the intersection of the neighbors of u with $C(p')$. It starts by checking if the neighbor v of u belongs to $C(p')$ (lines 17–18). Next, the algorithm joins the label lifespan of both u, v with $\mathcal{I}_{\mathcal{P}}$ and then with their edge lifespan (line 19). The reason is that, it has to identify in which time instances u and v are connected with the correct labels as defined by the pattern nodes p and p' respectively. TIME_JOIN ignores all neighboring nodes v of node u for which the resulting duration $\mathcal{I}$ is less than the current duration ϑ. Note that, although REFINECANDIDATES checks for the duration of the lifespans of the labels and edges of the candidate nodes, it does not ensure that all edges of a found match are active at the same time instants. This is the reason why when a pattern match is found, Algorithm 3 checks for its duration in $\mathcal{I}_{\mathcal{P}}$ (lines 1–4 in Algorithm 3).

Algorithm 4 REFINECANDIDATES(VG_I, $\mathcal{P}$, C, ϑ, $\mathcal{I}_{\mathcal{P}}$)

Input: Version graph VG_I, pattern $\mathcal{P}$, candidate sets C, duration threshold ϑ, set of intervals $\mathcal{I}_{\mathcal{P}}$
Output: Candidate sets C after reduction

```
1:  for each $p \in V_{\mathcal{P}}$ do
2:      for each $(p, p') \in E_{\mathcal{P}}$ do
3:          $C'_{p'} \leftarrow \emptyset$
4:          for each $u \in C(p)$ do
5:              $C_u(p') \leftarrow$ TIME_JOIN($p$, $u$, $p'$)
6:              if $C_u(p') = \emptyset$ then
7:                  $C(p).remove(u)$
8:              else
9:                  $C'_{p'} \leftarrow C'_{p'} \cup C_u(p')$
10:         if $C'_{p'} = \emptyset$ then
11:             return $\emptyset$
12:         $C(p') \leftarrow C'_{p'}$
13: return $C$
14:
15: procedure TIME_JOIN($p$, $u$, $p'$)
16:     $C' \leftarrow \emptyset$
17:     for each $(u, v) \in E_I$ do
18:         if $v \in C(p')$ then
19:             $\mathcal{I} \leftarrow \mathcal{I}_{\mathcal{P}} \otimes \mathcal{L}_{u.label(p)} \otimes \mathcal{L}_{v.label(p')} \otimes \mathcal{L}_e((u, v))$
20:             if $|\mathcal{I}| \geq \vartheta$ then
21:                 $C'.add(v)$
22:     return $C'$
```

In the end of the procedure, the new set C' is returned with all nodes that are appropriate neighbors of u, otherwise an empty set is returned and node u is removed (lines 6–7).

8 EXPERIMENTAL EVALUATION

In this section, we evaluate: (i) the efficiency of our durable graph pattern matching algorithm and (ii) the effectiveness of our approach in discovering interesting durable patterns.

TABLE 1: Dataset characteristics

Dataset	# Nodes	# Edges	# Labels	# Instances
DBLP	1,167,796	4,919,780	4	58
DBLP$_C$	42,060	141,899	19	58
YT	1,138,499	4,452,646	10	37
WIKI	2,987,535	9,379,561	10	1,000
AIDS	245	11,792	62	40,000
PCMS	883	52,608	21	200
SYNR (default)	100,000	2,723,856	5	100
SYNP (default)	100,000	3,265,747	5	100

8.1 Datasets and Setting

We use a number of real datasets. The DBLP[1] datasets include publications in time interval [1959, 2016], where each graph snapshot corresponds to one year. A node denotes an author and there is an edge between two authors if they wrote a paper together in the corresponding year. We use two datasets: *DBLP* and *DBLP$_C$*. In *DBLP*, we include all publications in the DBLP dataset and assign labels to authors based on the number of their publications, *pub_no*, at the corresponding year. Specifically, a label takes 4 different values: BEGINNER, if $1 \leq pub\_no \leq 2$; JUNIOR, if $2 < pub\_no \leq 5$; SENIOR, if $5 < pub\_no \leq 10$; and PROF, if

1. http://dblp.uni-trier.de/

TABLE 2: Size and construction time

Dataset	Size in memory (MB)					Construction time (sec)							
	LVG	ViLa	TiNLaB (cprsn)	CTiNLaB (cprsn)	TiPLaB (cprsn)	LVG	ViLa	TiNLaB	TiNLa	CTiNLaB	CTiNLa	TiPLaB	TiPLa
DBLP	1,512	149	467 (16.46%)	1,037 (59.43%)	335 (90.49%)	21.32	3.18	16.35	6.12	38.46	10.43	553	72.39
$DBLP_C$	73	14	17 (56.41%)	39 (71.53%)	19 (81%)	0.65	0.69	0.03	0.32	1.91	1.07	22.33	2.18
YT	1,667	3,104	694 (10.22%)	734 (85.47%)	781 (97.13%)	16.67	29.33	15.32	8	36.65	26.2	2,552	6,265
WIKI	5,123	10,299	770 (29.68%)	1.828 (92.04%)	1.299 (95.13%)	49.61	25.32	71.63	17.61	5,419	155.45	3,539	1,263
AIDS	129	149	32 (41.82%)	246 (89.55%)	11 (98.35%)	1.80	0.08	3.94	0.41	36.39	23.69	84.19	15.97
PCMS	20	5	2.73 (9%)	15 (88.64%)	41 (96.38%)	0.99	0.01	0.70	0.1	2.38	1.3	6.69	11.44
SYNR	294	597	82 (3.53%)	326 (87.86%)	98 (97.03%)	8	4	7.52	3.85	28.22	6.24	23.46	41.54
SYNP	553	596	110 (22.72%)	393 (95.93%)	163 (97.78%)	19	4	8.42	3.88	48.23	9.03	58.16	104.65

$pub\_no > 10$. In $DBLP_C$, we include publications in 19 major database, data mining, computer systems, theory, network, and graphics conferences. Authors are labeled by the venues they published at the corresponding year.

We also use a YouTube (YT) [22] and a Wiki-talk[2] (WIKI) datasets in time intervals [1, 37], and [1, 1000] respectively. For both YT and WIKI, each snapshot corresponds to one day. Since, these datasets do not contain any other information besides the graph structure, we generate 10 different labels and assign them to nodes using a Zipf distribution. For example, labels in WIKI can refer to the expertise, language, nationality, region, and number of edits of a user. Using a larger number of labels would only make the problem easier due to smaller candidate sizes. In addition, we use two biological networks [23] namely AIDS and PCMS. The AIDS dataset consists of 40,000 instances where each instance denotes a topological structure of a molecule. The PCMS dataset consists of 200 instances where each instance represents relationships among amino acids. The AIDS and PCMS datasets have 62 and 21 unique label values, respectively. Finally, we use synthetic datasets with varying number of nodes and snapshots, one random (SYNR) and one (SYNP) generated using preferential attachment [24]. All synthetic datasets have 5 labels assigned using Zipf distribution.

The DBLP, biological and synthetic networks are undirected graphs, while YT and WIKI are directed graphs. The dataset characteristics of the real datasets and the default synthetic datasets are summarized in Table 1. The number of nodes and edges are those of the LVG.

We ran our experiments on a system with an Intel Core i7-3820 3.6 GHz processor using 64 GB memory. We use all 8 threads for index construction and one thread for query processing.

8.2 Time Indexes Storage and Construction

In this set of experiments, we report the size and time needed to construct the various time indexes. We use as default the compressed version of the indexes and compare their performance with their uncompressed counterparts, since the compressed indexes are space efficient and achieve similar query performance. The size of the Bloom filters is set so as to achieve a false positive rate of 1%. We use for TiNLaB and CTiNLaB, $r = 1$ and for TiPLaB, $\lambda = 3$ and present experiments for different values.

Size. In Table 2, we report the size of LVG and the size of the various indexes. LVG is our in-memory representation of the temporal graph. Comparing the size of the various indexes, CTiNLaB is overall the most expensive one due to the use of counters. Although TiPLaB maintains all paths (up to length λ = 3) per time instant, the use of Bloom filters make it space efficient. Comparing the different datasets, note that the size of TiPLaB for the YT dataset is larger than *DBLP* since YT nodes and edges are active during all time instances, whereas *DBLP* is more active in the last 20 years in the interval. Thus, for each time instant of YT, all nodes are assigned to label paths resulting in a larger structure. Although, WIKI has the largest number of instances (almost 30 times more than YT), the corresponding indexes are only 2-3 times larger. For the biological networks, the neighborhood time indexes for AIDS are larger that those for PCMS and this is due to the large number of instances of AIDS. TiPLaB in PCMS is larger than AIDS because AIDS contains smaller graphs with much fewer paths compared to PCMS.

In Table 2, we also report the compression rate achieved by using the compressed indexes over using the uncompressed indexes. We observe that the reduction in size is significant especially for costly indexes such as CTiNLa and TiPLa.

Construction time. As shown in Table 2, the construction of ViLa is the fastest one, because it links only each node with the corresponding label for each time instant. TiNLaB requires time for checking the labels of the neighbors of each node for each time instant. Since WIKI has a large number of instants CTiNLaB require almost 2 hours to be created, since for each time instant we have to check a very large number of neighbors. The TiPLaB construction is also expensive in all datasets, because it has to perform a traversal from each node and compute label paths for each time instant. Also notice that constructing TiPLaB for AIDS requires more time than for PCMS even if it leads to a smaller structure, and this is due to the large number of instances in AIDS dataset that need to be examined for label paths.

In Table 2, we also report the construction time for the uncompressed indexes. Compression introduce overhead, which is however justified by the reduction in storage and the fact that the indexes are constructed once.

Scalability. We also test the scalability of the indexes in terms of both the size of the graphs and the number of snapshots using the synthetic datasets. For testing scalability with size, we create an initial graph snapshot G_1 with N nodes (for N = 100,000 up to 500,000) either using a random (SYNR) or a preferential attachment (SYNP) model. Then for each graph, we create 100 snapshots as follows. Given G_t, we create G_{t+1} be deleting 10% random edges in G_t, and adding 10% of new edges. The addition of edges is done using the corresponding model. The results of the indexes of the created temporal graphs are shown in Fig. 5(a) and 5(c). All indexes scale linearly with the number of nodes, while the increase for TiPLaB and TiNLaB is very small.

2. https://doi.org/10.5281/zenodo.49561

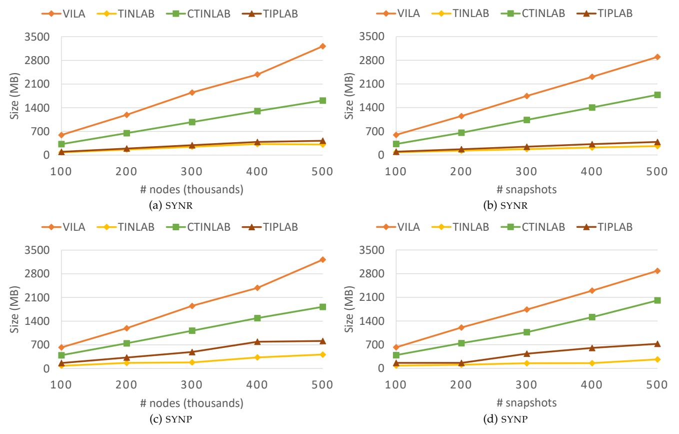


Fig. 5: Index size for varying (a)(c) size of nodes, and (b)(d) number of snapshots

For testing scalability with the number of graph snapshots, we create an initial graph G_1 with 100,000 nodes and then create T = 100 up to 500 snapshots as described previously. We report the results in Fig. 12(b) and 12(b). The results are similar as with the number of nodes. Scalability with time is also linear with the number of nodes and the number of snapshots.

8.3 Graph Pattern Query Processing

Let us now focus on processing durable graph pattern queries. As our default pattern queries, we use two type of queries: (a) random graph pattern queries, and (b) clique queries where all nodes have the same label.

Random graph pattern queries are generated as follows. For a random query of size n, we select a node randomly from the graph and keep among its label the one having the largest lifespan duration. Then, starting from this node, we perform a DFS traversal keeping for each visited node the label with the largest lifespan duration until the required number n of nodes is visited. We use as our pattern, the graph created by the union of visited labeled nodes and traveled edges. We report the average performance of 100 random queries for each size n.

In terms of clique queries, in the *DBLP* dataset, we have four label cliques. This gives us pattern queries with varying selectivities among them the BEGINNER cliques have the largest number of matches and the PROF cliques the smallest. Similarly, for the YT and WIKI datasets, we get 10 different cliques. Let us call MOST and LEAST the cliques with nodes having the most and the least frequent label, respectively. We get similar cliques for the other datasets.

As query interval, we use the whole duration of the temporal graph. We limit our algorithm to get the first 1,000 durable matchings for frequent patterns. In case of durable top-k durable queries we use 10 as the default k value. We only report the response time of collective-time queries, since, in all cases, contiguous-time queries are processed much faster by our algorithm because of the more effective pruning of candidate sets due to the constraint of the consecutive time instances. We use as default the MAXRANK duration strategy.

TABLE 3: Comparison with the baseline algorithm

		Most durable (sec)		**Top-k durable (sec)**	
Dataset	Q. Size	Baseline	VILA	Baseline	VILA
DBLP	2	>5,400	3.01	>5,400	3.18
DBLP	4	>5,400	14.08	>5,400	10.23
DBLP	6	>5,400	161.07	>5,400	111.15
DBLP$_C$	2	3.08	0.006	3.24	0.008
DBLP$_C$	4	3.84	0.11	4.23	0.031
DBLP$_C$	6	2.97	0.157	3.74	0.404
YT	2	>5,400	4.08	>5,400	4.08
YT	4	>5,400	6.79	>5,400	6.58
YT	6	>5,400	12.73	>5,400	12.90
WIKI	2	>5,400	3.28	>5,400	2.86
WIKI	4	>5,400	5.26	>5,400	4.58
WIKI	6	>5,400	120.14	>5,400	108.80
AIDS	2	38.53	0.98	41.56	0.94
AIDS	4	31.77	1.15	32.91	1.06
AIDS	6	27.34	1.38	29.32	1.36

Comparison with the baseline algorithm. Let us first compare the performance of our algorithm with the baseline. In this experiment, we use just VILA, the most basic time index. Table 3 reports the results for random queries for most and top-k durable queries. Since the baseline algorithm needs to generate all matching patterns, it is prohibitively slow. In many cases, we had to stop the baseline after 1.5h. As shown, the baseline algorithm takes less than 1.5h only for small datasets or datasets with selective query patterns, i.e., for query patterns with few matches per snapshot. Still our algorithm with the basic time index is considerably faster in such cases as well. For instance, in *DBLP*$_C$ for small query sizes, it is up to ∼513x faster than the baseline for both most and top-k durable queries. Also even for the AIDS dataset, where the graphs are small, the large number of instances makes baseline ∼30x slower.

We also run the baseline algorithm for finding durable cliques and the results are similar. In general, the baseline algorithm tends to generate many redundant matches even for selective queries. (e.g., for the PROF 2-clique query, the baseline approach generates a total of 62,302 matches, whereas there is only one durable match).

Varying r and λ. Fig. 6, shows the impact of parameters r (TINLAB, CTINLAB) and λ (TIPLAB) for *DBLP*. We observe that increasing radius r for TINLAB and CTINLAB does not improve performance. We examined this behavior and found that the additional checks in each neighborhood do not reduce the search space satisfactorily and thus the overhead induced by these checks leads to larger response times. TIPLAB seems to perform better as we increase λ, since there is a huge decrease in search space. We did not examine larger values for λ because it was prohibitively expensive for our graphs due the very large number of different paths. Similar observations have been made for the other datasets and thus we use $r = 1$ for TINLAB (CTINLAB) and $\lambda = 3$ for TIPLAB as default values.

Duration threshold. In this set of experiments, we compare the different strategies for setting the duration threshold. The results for *DBLP* and YT using the MAXRANK and MAXBINARY strategies are depicted in Fig. 7 for most and in Fig. 8 for top-k durable random queries. We also report the results for WIKI in Fig. 10. Results with the MIN strategy are not shown, since this strategy requires more than 1.5h in many cases. This is due to the large size of the candidate sets of MIN, since setting threshold ϑ equal to one in

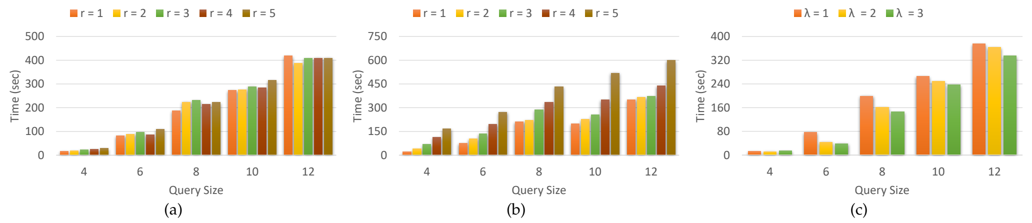


Fig. 6: Query time for random most durable queries for varying r for (a) TINLAB, (b) CTINLAB, and varying λ for (c) TIPLAB in *DBLP*

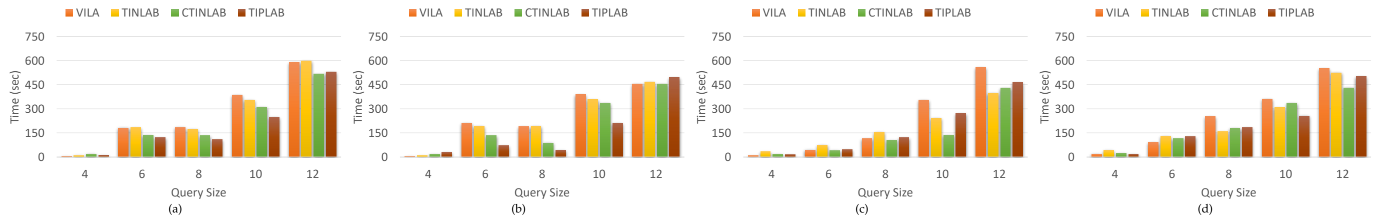


Fig. 7: Query time for random most durable queries using MAXRANK in (a) *DBLP* and (c) YT, and MAXBINARY in (b) *DBLP*, and (d) YT

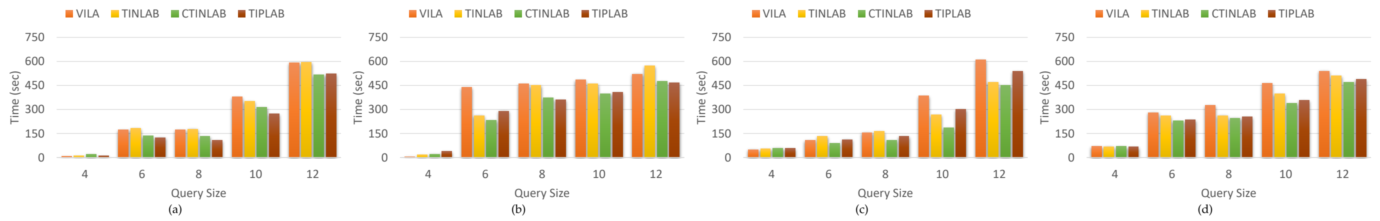


Fig. 8: Query time for random top-k queries using MAXRANK in (a) *DBLP* and (c) YT, and MAXBINARY in (b) *DBLP*, and (d) YT

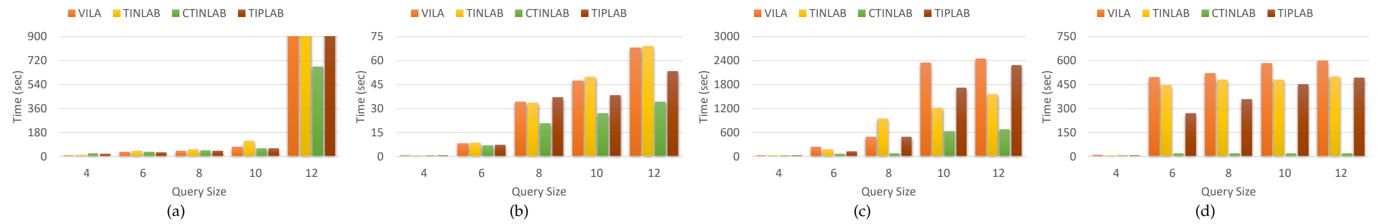


Fig. 9: Query time for most durable clique queries: (a) BEGINNER in *DBLP*, (b) PROF in *DBLP*, (c) MOST in YT and (d) LEAST in YT, note that for cliques of size 12 in *DBLP*, the plot is limited to 900 secs, the actual time for VILAG, TINLAB, is 1555, 1548 respectively and for TIPLAB 1331 secs

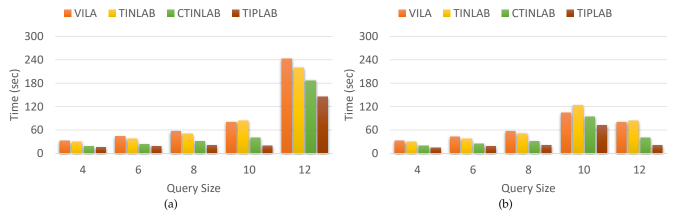


Fig. 10: Query time for random most durable queries using (a) MAXRANK, and (b) MAXBINARY in WIKI

the first steps of the algorithm results in searching in all graph snapshots for durable matches. Similar results hold for cliques queries and the other datasets.

Overall, the MAXRANK strategy outperforms the MAXBINARY strategy for all datasets and all but the largest query sizes. This is because MAXBINARY reduces the ϑ threshold at each step in half often producing values far below the actual duration thus creating large candidate sets and more recursive calls in each step. MAXBINARY performs better only for the largest query sizes since for such queries the actual duration of the matches is small and thus by reducing ϑ at each step in half, MAXBINARY is able to reach the correct threshold faster.

TABLE 4: Number of selected ϑ values and number of recursive calls for TIPLAB in *DBLP*

	MAXRANK		MAXBINARY	
Q. Size	# ϑ	# recursions	# ϑ	# recursions
2	15	3	2	10
4	16	166	4	348
6	19	932	3	2,026
8	19	1,853	3	759
10	19	2,263	3	1,169

In Table 4, we present the number of selected ϑ values and the recursive calls required for returning the most durable cliques in *DBLP* using MAXRANK and MAXBINARY, where the number of recursive calls accounts for the actual cost of the algorithm. The number of calls is not proportional to the number of ϑ values, since for larger ϑ values we have smaller candidate sizes. MAXRANK selects more ϑ values but these values are large, whereas ϑ selects fewer but smaller ones.

Overall, MAXRANK seems to strike a good balance giving few recursive calls with high enough ϑ values and we use this strategy as the default one.

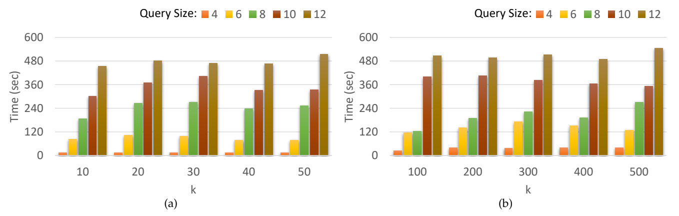


Fig. 11: Query time using TIPLAB for top-k durable queries for various k and query sizes in *DBLP*

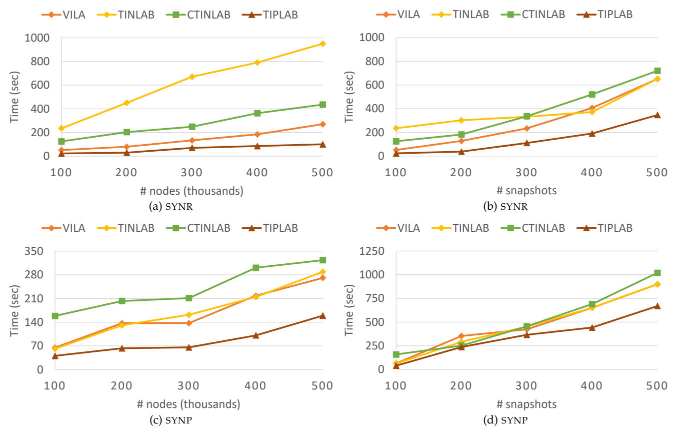


Fig. 12: Query time for random most durable queries for varying (a)(c) size of nodes, and (b)(d) number of snapshots

Time indexes. Let us now compare the performance of the different time indexes using the default MAXRANK strategy. Results are shown for most durable random queries in Fig. 7(a), 7(c) and Fig. 10(a) and for top-k durable random queries in Fig. 8(a) and 8(c). In addition, in Fig. 9, we depict results for the most and least selective cliques queries. A first observation is that the relative performance of the indexes is the same for the most and the top-k random queries. The same observation was found to hold for top-k clique queries (not shown).

Overall, the indexes that lead to smaller candidate sets and thus achieve more effective refinement work better. Which index works best depends on the type of the query. For random queries, TIPLAB and CTINLAB work the best, with TIPLAB working better for large networks such as WIKI. On the other hand, for clique queries, CTINLAB outperforms TIPLAB. The reason is that, since we use cliques with the same labels, the matches need to have a specific number of neighbors with this label, and the pruning achieved by CTINLAB is substantial. Note also, that the most selecting queries PROF and LEAST are considerable faster than the corresponding less selective ones, BEGINNER and MOST respectively. Between the two datasets, YT consists of edges with large lifespan which is an important factor that leads large queries to have matches with high duration. Thus, the algorithm answers faster the corresponding queries in YT than in *DBLP* since more steps are required for locating the durable matches. Finally, in few cases for queries of small size, the reduction in the search space achieved by the indexes is small and the overhead caused by the extra checks surpasses the gain from this reduction. For example, in *DBLP* and for the smallest BEGINNER cliques VILA outperforms all other indexes. Although, the total

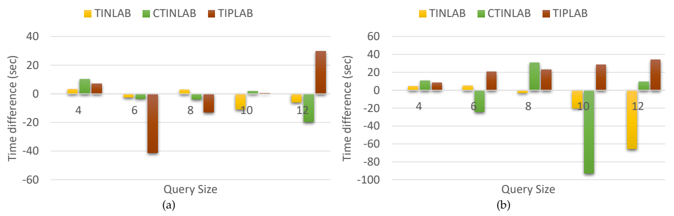


Fig. 13: Comparison with the non-compressed indexes for random most durable queries: in (a) *DBLP*, and (b) YT

recursions using time-neighborhood indexes are less than using VILA, the extra cost of processing the indexes leads to this small difference in query time.

Varying k. We also run the top-k algorithm using TIPLAB for various k values. In Fig. 11(a), we depict the results for small k values. Overall processing does not increase with k as long as there are enough matches in the first runs of the algorithm. Fig. 11(b) depicts the processing time for larger values of k. Overall TIPLAB seems to be stable in term of query processing time, with a small increase with k. Results for the other indexes are similar.

Scalability. In this set of experiments, we use synthetic datasets to study the performance of random most durable queries as we increase the number of nodes and the number of snapshots in Fig. 12(a)(c) and Fig. 12(b)(d) respectively. We observe that for all time indexes the response time increases linearly with both the number of nodes and snapshots. In particular, TIPLAB shows excellent scalability.

Comparison with the non-compressed indexes. We also compare the performance of TINLA (CTINLA) and TIPLA versus their compressed versions using random most durable queries in Fig. 13. Overall, the compressed indexes are clear winners given the size and their comparable query performance.

8.4 Case Studies

Finding durable graph patterns can reveal interesting information about the datasets. In this section, we present example results of durable cooperations among authors using *DBLP* and biological datasets.

Conferences with durable cliques. In our first study, we use the $DBLP_C$ dataset and study the appearance of author cliques in conferences. To this end, we use clique patterns labeled with the name of the conference. The results using cliques of various sizes are summarized in Table 5. Various observation can be made, for example, ICDE has the most durable cliques among the database conferences followed by SIGMOD, while in data mining, the most durable cliques appear in KDD. As expected in theory, cliques are smaller, with SODA having both the largest and the most durable cliques.

Some of the authors forming the most durable matches are shown in Table 6, while the top-5 most durable authors in ICDE are shown in Table 7(a) .

Durable pattern in biological datasets. In our last study, we present in Table 7(b) results regarding amino acids using the PCMS dataset. We study the appearance of different hydrophobic amino acids which are buried inside the protein molecules hydrophobic cores. We observe that

TABLE 5: Author cliques in major cs conferences. Symbol "**" depicts a very large number (≥ 1000) of matches

Cliques	Size 2		Size 3		Size 4		Size 5		Size 6	
Conference	Duration	Matches	Duration	Matches	Duration	Matches	Duration	Matches	Duration	Matches
SIGMOD	11	1	5	2	4	1	3	1	2	**
ICDE	14	1	7	1	4	1	3	1	2	**
VLDB	8	1	4	4	3	6	3	1	2	**
EDBT	10	1	3	5	2	**	2	**	2	**
KDD	14	1	6	2	4	2	3	7	3	1
WWW	7	1	5	1	3	3	2	8	1	1
CIKM	11	1	5	4	4	1	3	1	2	**
SIGIR	11	1	6	1	4	1	3	1	2	**
FOCS	7	1	3	2	2	4	---	---	---	---
STOC	8	1	4	1	3	1	2	3	---	---
SODA	12	1	6	1	3	1	2	6	2	1
ICALP	6	1	4	1	3	1	2	1	---	---
OSDI	4	4	3	2	2	13	2	1	---	---
SOSP	7	1	3	5	2	20	2	2	---	---
USENIX	3	1	3	2	2	34	2	**	2	**
SIGCOMM	10	1	4	2	3	6	3	1	2	**
SIGMETRICS	8	1	4	4	4	1	3	1	2	**
SIGOPS	3	3	2	7	2	1	---	---	---	---
SIGGRAPH	8	1	5	1	4	1	3	1	---	---

TABLE 6: Example authors with durable cooperation

Conferences	Duration	Authors
KDD	14	Charu C. Aggarwal, Philip S. Yu
ICDE	14	Divyakant Agrawal, Amr El Abbadi
SODA	12	Micha Sharir, Pankaj K. Agarwal
SIGMOD	11	Vivek R. Narasayya, Surajit Chaudhuri
CIKM	11	Clement T. Yu, Weiyi Meng
SIGIR	11	Craig Macdonald, Iadh Ounis
SIGCOMM	10	Ion Stoica, Scott Shenker
WWW	7	Andrew Tomkins, Ravi Kumar
SIGMOD – SIGCOMM	7	Joseph M. Hellerstein, Ion Stoica
VLDB – EDBT – SIGMOD	3	Laks V. S. Lakshmanan, H. V. Jagadish, Divesh Srivastava

the most connections to Phenylalanine (F), produce high number of matches. This can be explained by the fact that phenylalanine is an essential amino acid meaning the body needs this ingredient and is unable to produce it naturally. In addition, it is used to biochemically form proteins, coded for by DNA. We also sought durable cliques of various sizes for all amino acids and we observed that they do not participate in cliques of size greater than two.

TABLE 7: (a) Top-5 pairs of authors, (b) results from PCMS

#	Duration	Authors (ICDE)
1	14	Divyakant Agrawal – Amr El Abbadi
2	12	Jeffrey Xu Yu – Xuemin Lin
3	12	Beng Chin Ooi – Kian-Lee Tan
4	10	Vivek R. Narasayya – Surajit Chaudhuri
5	10	Charu C. Aggarwal – Philip S. Yu

(a)

Amino acids	Duration	Matches	Amino acids	Duration	Matches
R – G	40	1	F – V	24	51
L – I	32	2	F – A	24	40
L – V	32	1	L – M	24	33
A – V	28	2	F – M	24	29
L – F	28	1	S – P	24	22
F – I	24	55	G – A – R	24	9

(b)

9 Related Work

Graph pattern matching in static graphs has been widely studied. To the best of our knowledge, we are the first to introduce and study the problem of finding durable graph pattern matches in a graph history. Next, we survey related work on graph pattern queries in static graphs and on queries in temporal graphs.

Graph Pattern Queries. Finding graph matches is an important problem in many applications involving data modeled as graphs. The problem has been studied first in the theoretical literature as the subgraph isomorphism problem [4] where it was shown to be NP-complete [25], and given the size of the graphs, this often proves to be too computationally expensive. In recent years, many algorithms have been proposed to solve it in a reasonable time using different indexing and pruning techniques. Based on the approaches followed to process graph pattern queries, we can categorize representative graph pattern matching approaches into two categories [26].

The first category includes indexing algorithms [1], [4], [6], [23], [27], [28], [29] that find all embeddings for a given query graph and a data graph. In particular, these algorithms first captures auxiliary neighborhood information to retrieve for each query node all the candidate data nodes that may be part of a match. Then, they prune candidate nodes that do not meet the required neighborhood properties defined by the query graph and return the nodes that form a graph pattern match. The second category includes algorithms [5], [7], [8], [9] which are processing pattern queries by decomposing the query graph into paths and looking for candidate graph data paths whose join produces query matches. According to this classification, our approach is closer to the first category.

In both categories, algorithms use various indexing techniques to accelerate subgraph pattern matching. In particular, neighborhood indexes [1], [30] are proposed for the nodes in the graph where each index contains properties of nodes in the neighborhood. Combining these indexes with a distance measure, any pair of query nodes is compared to the data graph nodes in order to locate the query matches. A different type of index is proposed in [7] where the authors use for each node the shortest paths from each node in the graph within its k-neighborhood to capture the local structural information around the node. Then a query graph decomposition is performed into a set of indexed shortest paths in order to locate candidate paths from data graph that cover the original query graph. The study in [29] tries to access label frequency information and the frequencies of a triple $(fromLabel, edgeLabel, toLabel)$. For each pattern query, they weight query graph edges accordingly and uses these weights to order the search by creating a minimum spanning tree. The recent work in [31] is the first one to use caching techniques for higher pruning power during graph pattern query processing. Finally, the authors in [32] identify a set of key factors that influence the performance of subgraph isomorphism algorithms and report the construction, indexing and query processing time of six methods.

In this paper, we are looking for top-k durable matches. Previous work has considered top-k graph matches in different contexts. For example, when there is some weight associated with nodes or edges, matches are ranked based on their weight [33]. Alternatively, to minimize overlap among matches, the authors in [3] introduce diversity constraints and look for the top-k diverse matches of a given query.

Temporal Queries. Although, graph data management has been the focus of much current research, work in processing temporal queries is rather limited. The main focus of research on temporal graphs has been on efficiently storing and retrieving graph snapshots. Various optimizations for reducing the storage and snapshot reconstruction overheads have been proposed. Optimizations include the reduction of the number of snapshots that need to be reconstructed by minimizing the number of deltas applied [12], using a hierarchical index of deltas and a memory pool [11], avoiding the reconstruction of all snapshots [13], and improving performance by parallel query execution and proper snapshot placement and distribution [34].

Concerning temporal graph processing, recent works address indexing for temporal reachability queries using an index that contains information about membership in strongly connected components at various time points [15], indexing in native graph databases [35], and indexing for temporal shortest path distance queries [16]. The work in this paper extends previous work in [36] by considering top-k durable graph pattern queries and introducing compressed time indexes and various optimizations in selecting appropriate values for the threshold duration.

10 Summary

In this paper, given the history of a node-labeled graph in the form of graph snapshots corresponding to the state of the graph at different time instances, we introduce the problem of finding the durable matches of an input pattern, that is, those matches that persist over time, either contiguously or collectively. We have presented an approach termed DurablePattern that efficiently identifies durable matches by traversing a compact representation of the graph snapshots and using a compressed time neighborhood and path indexes for pruning the number of candidate matches. Finally, we have proposed strategies for estimating the actual duration of the durable matches to further reduce the search space. Our extensive experimental evaluation with real datasets demonstrated the efficiency of our algorithm in finding durable matches.

References

[1] S. Zhang, S. Li, and J. Yang, "GADDI: distance index based subgraph matching in biological networks," in *EDBT*, 2009, pp. 192–203.

[2] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *EDBT*, 2015, pp. 157–168.

[3] W. Fan, X. Wang, and Y. Wu, "Diversified top-k graph pattern matching," *PVLDB*, vol. 6, no. 13, pp. 1510–1521, 2013.

[4] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[5] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD*, 2008, pp. 405–418.

[6] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 10, pp. 1367–1372, 2004.

[7] P. Zhao and J. Han, "On graph query optimization in large networks," *PVLDB*, vol. 3, no. 1, pp. 340–351, 2010.

[8] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *PVLDB*, vol. 5, no. 9, pp. 788–799, 2012.

[9] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," in *ICDE*, 2008, pp. 913–922.

[10] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad, "Fast best-effort pattern matching in large attributed graphs," in *ACM SIGKDD*, 2007, pp. 737–746.

[11] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *ICDE*, 2013, pp. 997–1008.

[12] G. Koloniari, D. Souravlias, and E. Pitoura, "On graph deltas for historical queries," *CoRR*, vol. abs/1302.5549, 2013.

[13] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng, "On querying historical evolving graph sequences," *PVLDB*, vol. 4, no. 11, pp. 726–737, 2011.

[14] A. G. Labouseur, J. Birnbaum, P. W. O. Jr, S. R. Spillane, J. Vijayan, J.-H. Hwang, and W.-S. Han, "The g* graph database: efficiently managing large distributed dynamic graphs," *Distributed and Parallel Databases*, 2014.

[15] K. Semertzidis, E. Pitoura, and K. Lillis, "Timereach: Historical reachability queries on evolving graphs," in *EDBT*, 2015, pp. 121–132.

[16] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *WWW*, 2014, pp. 237–248.

[17] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[18] A. Z. Broder and M. Mitzenmacher, "Survey: Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[19] C. S. Jensen and R. T. Snodgrass, "Temporal element," in *Encyclopedia of Database Systems*, 2009, p. 2966.

[20] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *PVLDB*, vol. 5, no. 4, pp. 310–321, 2011.

[21] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy, "Dualiso: An algorithm for subgraph pattern matching on very large labeled graphs," in *IEEE International Congress on Big Data, Anchorage*, 2014, pp. 498–505.

[22] A. Mislove, "Online social networks: Measurement, analysis, and applications to distributed information systems." Rice University, Department of Computer Science, 2009.

[23] R. Giugno, V. Bonnici, N. Bombieri, A. Pulvirenti, A. Ferro, and D. Shasha, "Grapes: A software for parallel searching on biological graphs targeting multi-core architectures," *PloS one*, vol. 8, no. 10, p. e76911, 2013.

[24] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[26] J. Lee, W. Han, R. Kasperovics, and J. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *PVLDB*, vol. 6, no. 2, pp. 133–144, 2012.

[27] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," in *EDBT*, 2008, pp. 181–192.

[28] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: Tree + delta >= graph," in *Very Large DataBases*, 2007, pp. 938–949.

[29] H. Shang, Y. Zhang, X. Lin, and J. X. Yu, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *PVLDB*, vol. 1, no. 1, pp. 364–375, 2008.

[30] S. Zhang, S. Li, and J. Yang, "SUMMA: subgraph matching in massive graphs," in *CIKM*, 2010, pp. 1285–1288.

[31] J. Wang, N. Ntarmos, and P. Triantafillou, "Indexing query graphs to speedup graph query processing," in *EDBT*, 2016.

[32] F. Katsarou, N. Ntarmos, and P. Triantafillou, "Performance and scalability of indexed subgraph query processing methods," *PVLDB*, vol. 8, no. 12, pp. 1566–1577, 2015.

[33] J. Cheng, X. Zeng, and J. X. Yu, "Top-k graph pattern matching over large graphs," in *ICDE*, 2013, pp. 1033–1044.

[34] A. G. Labouseur, P. W. Olsen, and J. Hwang, "Scalable and robust management of dynamic graph data," in *International Workshop on Big Dynamic Distributed Data*, 2013, pp. 43–48.

[35] K. Semertzidis and E. Pitoura, "Historical traversals in native graph databases," in *ADBIS*, 2017, pp. 167–181.

[36] ——, "Durable graph pattern queries on historical graphs," in *ICDE*, 2016, pp. 541–552.

Konstantinos Semertzidis is a PhD candidate at the Dept. of Computer Science and Engineering of the University of Ioannina in Greece. He received his BSc from the same institution in 2012 and his MSc from York University in 2013. He has worked as an Intern for IBM Research and Nokia Bell Labs in Ireland. His research interests include storage, indexing and processing of historical graph data.

Evaggelia Pitoura received the BSc degree in Computer Engineering from the University of Patras Greece, in 1990, the MSc and the PhD degrees in Computer Science from Purdue University, in 1993 and 1995, respectively. She is a Professor at the Dept. of Computer Science and Engineering of the University of Ioannina in Greece, where she leads the Distributed Data Management laboratory. She is a member of the IEEE Computer Society.