Durable Subgraph Matching on Temporal Graphs

Faming Li, Zhaonian Zou, and Jianzhong Li

Abstract—Durable subgraph matching on a temporal graph finds all subgraphs in the temporal graph that not only match the given query graph but also have duration longer than a user-specified duration threshold. The state-of-the-art algorithm (Semertzidis and Pitoura, 2016, 2019) for solving this problem requires lots of memory when the input temporal graph is large. In this paper, a new algorithm is proposed to solve this problem. Many effective techniques are developed to improve the performance of this algorithm, including the DFS-based query decomposition method, the TD-tree index structure, the sort-based vertex matching order, and the time instance set compaction method. The experimental results show that the proposed algorithm is an order of magnitude faster and requires significantly less memory than the state-of-the-art algorithm.

Index Terms—Temporal graph, subgraph isomorphism, durable subgraph matching, index

1 Introduction

Given a query graph and a data graph, the *subgraph matching problem* finds all subgraphs of the data graph that match the query graph, usually in terms of *subgraph isomorphism* [1]. Subgraph matching is the foundation of many techniques in graph databases [2], graph mining [3], computer-aided design (CAD) [4], computer vision [5], and so on. A large body of subgraph matching algorithms have been proposed in recent decades, including sequential algorithms, e.g., [6], [7], [8] and distributed parallel algorithms, e.g., [9], [10], [11], [12], [13].

In the real world, graphs mostly change over time, that is, vertices/edges are inserted or deleted over time, and the properties of vertices/edges are updated over time. Such graphs have been known as *temporal graphs*, *dynamic graphs* or *evolving graphs* in the literature. Conceptually, a temporal graph is comprised of a sequence of *snapshots* at the time when changes happen [14]. Subgraph matching on a temporal graph helps people unveil in-depth structural and temporal properties hidden in the evolution of this graph. So far, some pioneering work has been done in this research direction, including durable subgraph matching [15], [16], time-constrained continuous subgraph matching [17], temporal motif enumeration [18], time-respecting flow graph pattern matching [19], [20], and flow pattern enumeration [21].

Durable subgraph matching on a temporal graph focuses on finding all subgraphs that not only match the query graph but also have a long duration in the sequence of snapshots of the temporal graph [15]. This problem has many applications in epidemic risk prediction [22], social network analysis [23], and proximity network analysis [24]. For example, Fig. 1a illustrates a social network that changes over time, which is represented as a temporal graph with 4 snapshots. A query graph is given in Fig. 1b, which asks for the users in the social network who have established durable relationships (longer than the minimum duration threshold of 2) satisfying the pattern specified by the query graph. As highlighted in Fig. 1a, a subgraph in the social network that matches the query graph appears in 3 snapshots, longer than the minimum duration threshold. This result indicates that Ann, Bob and Carl have established durable relationships that satisfy the specified pattern.

Related Work. A large number of subgraph matching algorithms have been proposed on static graphs, that is, graphs that do not change over time, including the join-based algorithms [9], [10], [11], [25] and the algorithms based on filtering and verification [1], [6], [7], [8], [26]. However, these algorithms are not suitable to be adapted to handle subgraph matching on temporal graphs. A straightforward approach is applying any existing static subgraph matching algorithm to find all subgraphs in every snapshot that match the query graph, and counting the duration of each subgraph. Unfortunately, the time and space complexity of this approach is very high because it must carry out subgraph matching on all snapshots and keep all matched subgraphs in memory for duration counting. An alternative approach is merging all snapshots into a single graph where each edge is associated with all its temporal information. Then, an existing static subgraph matching algorithm is applied on the merged graph to find all subgraphs that match the query graph, and the duration of each subgraph is counted on the merged graph. However, this approach can usually generate lots of subgraphs that are actually not durable because the merged graph is comprised of even more edges than any individual snapshot.

• The authors are with the School of Computer and Technology, Harbin Institute of Technology, Harbin, Heilongjiang 150001, China.
E-mail: {lifaming2016, znzou, lijzh}@hit.edu.cn.

Manuscript received 27 Apr. 2021; revised 21 Nov. 2021; accepted 29 Jan. 2022.
Date of publication 7 Feb. 2022; date of current version 3 Apr. 2023.
This work was supported in part by the National Natural Science Foundation of China under Grants 62072138 and 61832003 and in part by the Open Research Projects of Zhejiang Lab under Grant 2021KC0AB02.
(Corresponding author: Zhaonian Zou.)
Recommended for acceptance by Z. Wang.
Digital Object Identifier no. 10.1109/TKDE.2022.3148995

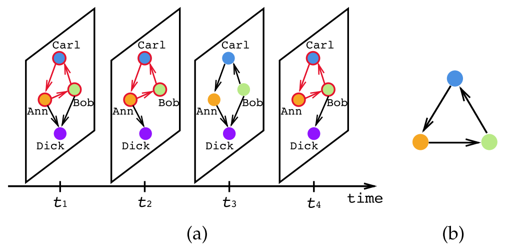


Fig. 1. An example of durable subgraph matching on temporal graphs. (a) A temporal graph with 4 snapshots; (b) A query graph. A matching of this query graph with duration 3 is highlighted in (a).

Recently, Semertzidis and Pitoura [15], [16] first study the durable subgraph matching problem on temporal graphs: given a temporal graph $\mathcal{G}$, a query graph Q, and the minimum duration threshold k, find all subgraphs of $\mathcal{G}$ that not only match Q but also have duration at least k. They use a bitmap to represent the occurrence states of an edge in all snapshots, where the ith bit of the bitmap is set if and only if the edge is contained in the ith snapshot. Therefore, the duration of a subgraph can be fast computed by performing bitwise-and operations on the bitmaps of the edges in this subgraph. This algorithm has two main weaknesses.

First, the algorithm requires at least $n|E| + n|V| \cdot |L|$ bits of memory to store all bitmaps, where n is the number of snapshots, E is the set of distinct edges in all snapshots, V is the set of vertices, and L is the set of labels. For example, for the temporal graph Bitcoin (file size: 3.01 GB) used in our experiments (Section 8), it takes more than 15 GB of memory to store all bitmaps when the time span of a snapshot is one week ($n = 370$). The huge memory footprint often degrades the efficiency and scalability of the algorithm when a large temporal graph with a lot of snapshots or a lot of distinct edges is given as input.

Second, the algorithm maintains the set $C(u)$ of candidate matching vertices of each query vertex u separately, so $C(u)$ often contains many false candidate vertices that cannot appear in any query results. The algorithm tries to eliminate unpromising vertices from $C(u)$ by considering $C(u')$ for all adjacent vertices u' of u in the query graph. However, without considering $C(u')$ for other query vertices u' that are reachable from u via long paths, a lot of unpromising candidate vertices still remain in $C(u)$. Consequently, the algorithm wastes considerable time on enumerating unpromising candidate matchings.

Contributions. In this paper, a new algorithm is proposed to solve the durable subgraph matching problem. Our algorithm overcomes the above weaknesses of [15], [16] and makes the following contributions.

1) A temporal graph is represented and physically stored by merging all snapshots in our algorithm. Unlike [15], [16], the time instances at which an edge exists are represented as a set instead of as a bitmap. We observe that the set-based temporal graph representation is more space-efficient than the bitmap-based representation due to the sparsity of bitmaps for real temporal graphs.
2) Instead of keeping the set $C(u)$ of candidate matching vertices for each query vertex u separately as in [15], [16], we organize all sets $C(u)$ using a data structure called *TD-tree* (Section 5). To construct a TD-tree, we decompose the query graph Q into a query tree T and a set of non-tree edges not in T (Section 4). For each vertex u on a path from the root of T to a leaf of T, unpromising vertices in $C(u)$ are removed by checking $C(u')$ for other vertices on this path. By considering such long-distance dependency, more unpromising candidate matching vertices can be removed from $C(u)$. In addition to all sets $C(u)$, the TD-tree also contains other information to facilitate durable subgraph matching, e.g., the related temporal information is stored in the leaf nodes of the TD-tree. Hence, the TD-tree contains a complete set of information necessary for generating candidate matchings of Q and is generally much smaller than $\mathcal{G}$. Therefore, in our algorithm, candidate matchings of Q are generated directly based on the TD-tree instead of based on $\mathcal{G}$.
3) Based on the TD-tree, we develop an algorithm to generate candidate matchings of Q (Section 6), which follows a path-based matching paradigm. A sort-based vertex matching order is designed to generate fewer unpromising partial matchings. Duration checking is pushed deep down into the matching enumeration procedure to eliminate unpromising partial matchings as early as possible.
4) Extensive experiments have been carried out to evaluate the performance of the proposed durable subgraph matching algorithm. The experimental results show that the proposed algorithm is an order of magnitude faster than Semertzidis and Pitoura's algorithm [15] and 2 orders of magnitude faster than the baseline algorithm adapted from CECI, a state-of-the-art static subgraph matching algorithm [26]. Moreover, the proposed techniques including the DFS-based query decomposition method, the TD-tree index structure, the sort-based vertex matching order, and the time instance set compaction method have positive effects on improving the performance of our algorithm.

2 Problem Statement

In this section, we introduce several fundamental concepts and formally define durable subgraph matching on temporal graphs. Subgraph matching is normally studied on labeled graphs. Let Σ be a set of labels. A *labeled graph* is a triple $G = (V, E, L)$, where V is a set of vertices, E is a set of directed edges, and $L : V \rightarrow \Sigma$ is a function that assigns a label to each vertex. A labeled graph $G' = (V', E', L')$ is a *subgraph* of G if $V' \subseteq V$, $E' \subseteq E$, and L' is $L|_{V'}$, the projection of L onto V', that is, $L'(v) = L(v)$ for all $v \in V'$. For ease of notation, let V_G, E_G and L_G represent the set of vertices, the set of edges and the labeling function of G, respectively. Without otherwise stated, a graph refers to a labeled graph throughout this paper.

Definition 1 (Subgraph Isomorphism). *Given two labeled graphs G and Q, Q is* subgraph isomorphic *to G if there is a bijection $f : V_Q \rightarrow V_G$ such that $(f(u), f(v)) \in E_G$ if and only if $(u, v) \in E_Q$, and for all $v \in V_Q$, $L_G(f(v)) = L_Q(v)$. For a*

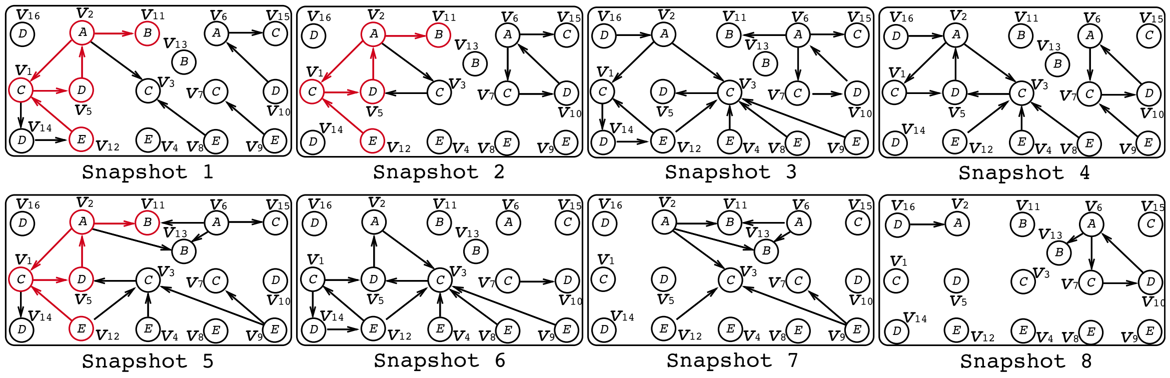


Fig. 2. A temporal graph consisting of 8 snapshots.

subgraph isomorphic mapping f from Q to G, the subgraph $(f(V_Q), f(E_Q), L|_{f(V_Q)})$ of G is called the matching *(or* embedding*) of Q in G under f, where $f(V_Q)$ denotes the image of V_Q under f, and $f(E_Q) = \{(f(u), f(v)) | (u, v) \in E_Q\}$. For ease of notation, we use $f(Q)$ to denote this matching of Q.*

Graphs in the real world mostly change over time, which are often called *temporal graphs*. We give a formal definition of temporal graph similar to Semertzidis and Pitoura's [15], [16]. For simplicity, we also assume that time is discrete and use monotonically increasing integers to represent increasing time instances. For a graph G that changes over time, let V_t, E_t and L_t be the set of vertices, the set of edges and the labeling function of G at time instance t. The graph (V_t, E_t, L_t) is called the *snapshot* of G at time instance t. A *temporal graph* in time interval $[i, j]$, denoted by $\mathcal{G}_{[i,j]}$, is logically a sequence of consecutive snapshots $(G_i, G_{i+1}, \ldots, G_j)$ at time instances in $[i, j]$. Fig. 2 shows a temporal graph consisting of 8 snapshots in time interval $[1, 8]$. A graph H is a subgraph of a temporal graph $\mathcal{G}_{[i,j]}$ if H is a subgraph of a snapshot of $\mathcal{G}_{[i,j]}$. Actually, H may appear as subgraphs in more than one snapshots. Let I be the set of time instances such that H is a subgraph of the snapshot G_i for all $i \in I$. For a positive integer k, H is said to be *k-durable* if $|I| \geq k$.

Based on the concepts presented above, we now define subgraph matching on temporal graphs: Given a temporal graph $\mathcal{G}_{[i,j]}$ in time interval $[i, j]$, a graph Q as a query pattern and a positive integer k as the minimum duration threshold, Q *k-durably matches* $\mathcal{G}_{[i,j]}$ if there exists a k-durable subgraph H of $\mathcal{G}_{[i,j]}$ such that Q is subgraph isomorphic to H. The subgraph H is called a *k-durable matching* of Q in $\mathcal{G}_{[i,j]}$. The problem studied in this paper is defined as follows:

Durable Subgraph Matching on Temporal Graphs

Input: a temporal graph $\mathcal{G}_{[1,n]}$ in time interval $[1, n]$, a query graph Q and the minimum duration threshold $k \in \mathbb{N}$.

Output: all k-durable matchings of Q in $\mathcal{G}_{[1,n]}$.

3 Our Method

In this section, we introduce the representation of temporal graphs and give an overview of our algorithm for finding durable subgraph matchings on temporal graphs.

3.1 Representation of Temporal Graphs

A temporal graph is logically comprised of a sequence of snapshots. However, it is not suitable to physically store a temporal graph as a snapshot sequence because many snapshots have a significant overlap, especially, between consecutive snapshots. Storing redundant overlapping substructures across snapshots wastes a lot of memory. Therefore, we design a compact representation of temporal graph by merging all snapshots into a single graph. The compact representation of a temporal graph $\mathcal{G}_{[i,j]}$ is a labeled graph, where the vertex set is $\bigcup_{t=i}^{j} V_{G_t}$, the edge set is $\bigcup_{t=i}^{j} E_{G_t}$, and the label of a vertex $v \in \bigcup_{t=i}^{j} V_{G_t}$ is the label of v in a snapshot that contains v (This paper assumes that v has the same label across all snapshots that contain v). In the compact representation, each edge is associated with a set of (discrete) time instances at which the edge exists. We use $TS(e)$ to denote the set of time instances of edge e. Note that we do not use a bitmap [15] to represent $TS(e)$ because most edges in a real-world temporal graph exist at very few time instances, thereby encoding $TS(e)$ with a bitmap often wastes a lot space due to the sparsity of the bitmap. For example, Fig. 3 depicts the compact representation of the temporal graph given in Fig. 2.

3.2 An Overview of Algorithm

Given a temporal graph $\mathcal{G}_{[1,n]}$ in time interval $[1, n]$, a query graph Q and the minimum duration threshold k, we develop an algorithm for finding all k-durable matchings of Q in $\mathcal{G}_{[1,n]}$ inspired by the existing work [7], [8], [26] on subgraph matching on static graphs. Our algorithm runs directly on the compact representation of $\mathcal{G}_{[1,n]}$. In the rest of the paper, we use $\mathcal{G}$ to refer to the compact representation of $\mathcal{G}_{[1,n]}$. Our algorithm runs in three phases.

Phase 1 (Query Decomposition). The query graph Q may be cyclic, that is, there is a cycle in Q. Our algorithm first decomposes Q into two nonoverlapping parts: a spanning tree T of Q and the set S of edges (called *non-tree edges*) not contained in T. For example, the query graph given in Fig. 4a can be decomposed into the tree and the set of non-tree edges shown

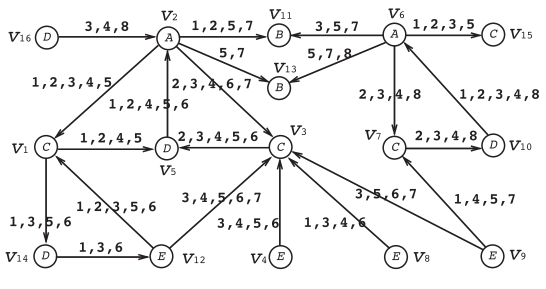


Fig. 3. The compact representation of the temporal graph in Fig. 2.

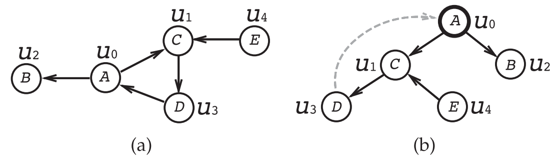


Fig. 4. A query graph and a decomposition of the query graph. (a) A query graph; (b) A decomposition of the query graph in (a). The dashed arrow represents the non-tree edge.

in Fig. 4b. The advantages of query decomposition are twofold. On one hand, the spanning tree T can effectively guide the search for candidate matchings of each vertex in Q as will be presented in Section 5. On the other hand, the non-tree edges in S are useful for filtering out false candidate matchings of some vertices in Q (as will be shown in Phase 2) as well as false k-durable matchings of Q (as will be shown in Phase 3).

In our algorithm, we designate a vertex in Q as the root and extract the (spanning) query tree T by carrying out a depth-first search (DFS) on Q starting from the root. The DFS tree determined by the DFS order is regarded as the query tree T. Actually, the way to decompose Q, that is, the selection of the root and the determination of the DFS order, can affect the efficiency of the succeeding phases of our algorithm and therefore the overall performance of our algorithm. We will discuss these issues related to query decomposition in Section 4.

Phase 2 (Search for Candidate Vertex Matchings). After query decomposition, our algorithm searches for candidate matchings of each vertex in the query graph Q on the compact representation $\mathcal{G}$ of the temporal graph. The search is guided by the query tree T obtained in Phase 1. The vertices in Q are considered in the DFS order for generating T. Instead of finding candidate matchings of each vertex in Q independently, we find candidate matchings of a vertex u based on the detected candidate matchings of the parent of u in T (if u is the root of T, the candidate matchings of u are found unconditionally). During the search, the non-tree edges obtained in Phase 1 are used as constraints for filtering out some false candidate matchings.

Any k-durable matching of Q is composed of one matching vertex for each query vertex in Q. We design an index structure called *TD-tree* to store the candidate matchings of all vertices in Q. Fig. 5 illustrates the TD-tree constructed for the query tree shown in Fig. 4b when $k = 3$. In practice, the TD-tree index is usually much smaller than the temporal graph $\mathcal{G}$. Our experiments in Section 8 show that the TD-trees constructed for the tested queries are at least 2 orders of magnitude smaller than the input data graph. Therefore, in Phase 3, our algorithm can enumerate candidate k-durable matchings of Q on the TD-tree more efficiently than on $\mathcal{G}$. We will present the details of Phase 2 in Section 5.

Phase 3 (Enumeration of Durable Subgraph Matchings). In Phase 3, our algorithm first enumerates candidate k-durable matchings of the query tree T based on the TD-tree. For each enumerated matching of T, we check if it really induces a k-durable matching of the query graph Q by checking the non-tree edges in Q and the duration. The details of Phase 3 will be presented in Section 6.

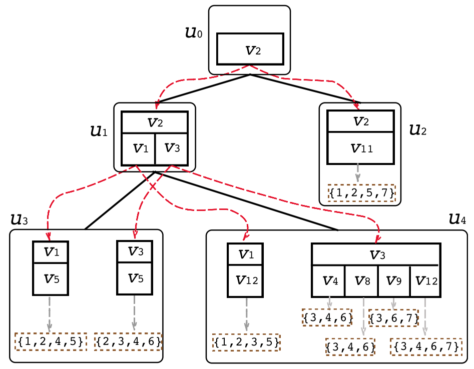


Fig. 5. The TD-tree constructed for the temporal graph $\mathcal{G}$ in Fig. 3, the query graph Q in Fig. 4a, the query tree T in Fig. 4b, and the minimum duration threshold $k = 3$.

4 Query Decomposition

The query graph Q may be cyclic. Our algorithm first decomposes Q into two nonoverlapping parts: a spanning tree T of Q and the set S of non-tree edges not in T. To obtain T, we select a vertex in Q as the root of T and carry out a depth-first search (DFS) on Q starting from the root. The DFS tree determined by the DFS order spans all vertices in Q and is regarded as the query tree T in our algorithm. Note that all the state-of-the-art subgraph matching algorithms on static graphs, e.g. [7], [8], [26], carry out a breadth-first search (BFS) on the query graph in query decomposition. Instead, our algorithm carries out a DFS on Q because a DFS tree usually leads to better performance than a BFS tree in the following phases of our algorithm as will be shown in our experiments in Section 8. The advantage of using a DFS tree is due to the following facts:

- Compared to a BFS tree, a DFS tree has fewer leaves (at most the same number of leaves) and therefore often has longer paths from the root to the leaves. When finding matchings of a longer query path, it tends to be earlier to identify and terminate an unpromising partial matching that violates the minimum duration constraint.
- Since T is a DFS tree, any non-tree edge in Q can only link a vertex to one of its ancestors in T. This property helps us test whether a candidate matching path generated from the TD-tree is qualified to match a query path immediately after generating this candidate matching path. The details will be shown in Section 6.

There are two key issues in generating T: the selection of the root and the determination of the DFS order. We discuss these two issues in the rest of this section.

Root Selection. As with TurboIso [8], we choose the vertex in Q with the lowest *selectivity* as the root of T. The selectivity of a query vertex u is formulated as

$$\frac{\text{the number of vertices in } \mathcal{G} \text{ with the label of } u}{\text{the degree of } u \text{ in } Q}.$$

Intuitively, for a vertex u that has fewer candidate matchings and a higher degree, the query tree T rooted at u tends to lead to a smaller TD-tree, which enables more efficient

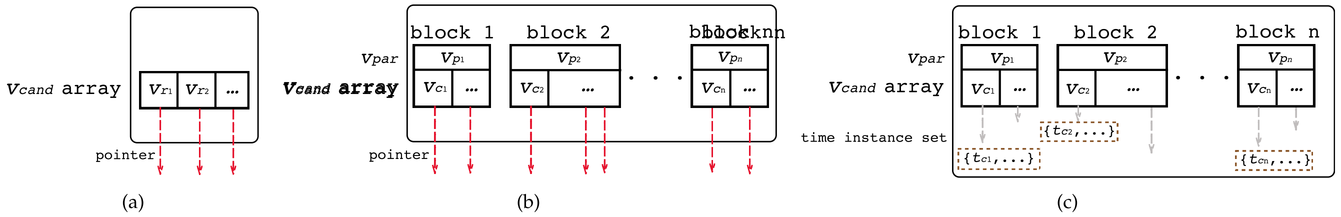


Fig. 6. The internal structures of three types of TD-tree nodes. (a) Root node; (b) Internal node; (c) Leaf node.

query processing in the following phases of our algorithm. For example, for the query graph in Fig. 4a and the temporal graph in Fig. 3, the selectivity of the vertices u_0, u_1, u_2, u_3, and u_4 are 2/3, 4/3, 2, 2, and 4, respectively. Therefore, u_0 is selected as the root of T.

DFS Order. After the root of T has been selected, the DFS order is determined by our algorithm. The DFS starts from the root. Let u be the vertex in Q currently being visited in the DFS. If some neighbors of u have not been visited in the DFS, the unvisited neighbor w of u with the lowest selectivity is going to be visited next in the DFS (ties are broken arbitrarily), and u is designated as the parent of w in the DFS tree. Otherwise, the DFS backtracks to the parent of u. For example, consider the query graph in Fig. 4a. Let u_0 be the root. The vertices in this query graph are visited in the following DFS order: u_0, u_1, u_2, u_3, and u_4. The DFS tree determined by this DFS order is shown in Fig. 4b.

5 Search for Candidate Vertex Matchings

In this section, we present the details of Phase 2 of our algorithm. The TD-tree index structure is introduced in Section 5.1, followed by the TD-tree construction algorithm in Section 5.2.

5.1 TD-Trees

We first introduce the data structure of a TD-tree. Given the query tree T obtained in Phase 1, the TD-tree based on T has the identical tree structure as T. For example, Fig. 5 shows the TD-tree constructed based on the query tree given in Fig. 4b. To show the one-to-one correspondence between all TD-tree nodes and all vertices of T, each TD-tree node is labeled with the ID of its corresponding vertex in T. Due to this one-to-one correspondence, we can interchangeably use a TD-tree node to refer to the vertex in T that the TD-tree node corresponds to for ease of presentation. There are three types of nodes in a TD-tree: *root node*, *internal nodes* and *leaf nodes*.

Root Node. There is exactly one root node in a TD-tree. The root node has no parent. As illustrated in Fig. 6a, the root node stores an array V_{cand} of vertices in $\mathcal{G}$ that match the root r of T. Each vertex v in V_{cand} must satisfy all the following conditions:

1) The label of v is the same as the label of r.
2) The degree of v in $\mathcal{G}$ is greater than or equal to the degree of r in Q.
3) The number of distinct labels of v's neighbors in $\mathcal{G}$ is greater than or equal to that of r's neighbors in Q.
4) The duration of v in $\mathcal{G}$ is at least k.

Internal Nodes. An internal node in a TD-tree has a parent and at least one child. As illustrated in Fig. 6b, an internal node u is composed of several *blocks*. Each block consists of a vertex v_{par} that matches the parent of u and an array V_{cand} of vertices in $\mathcal{G}$ that match u given that the parent of u is mapped to v_{par}. Each vertex v in V_{cand} must satisfy all the following conditions:

1) The label of v in $\mathcal{G}$ is the same as that of u in Q.
2) The degree of v in $\mathcal{G}$ is greater than or equal to the degree of u in Q.
3) The number of distinct labels of v's neighbors in $\mathcal{G}$ is at least that of u's neighbors in Q.
4) The duration of the edge (v_{par}, v) in $\mathcal{G}$ is at least k.

Leaf Nodes. A leaf node in a TD-tree has no children. As illustrated in Fig. 6c, the structure of a leaf node is similar to that of an internal node. The only difference is that, in each block, every vertex v in the array V_{cand} is associated with a set *TS* of time instances. The meaning of these times instances will be explained later.

Auxiliary Pointers. To make a fast search on a TD-tree, we add auxiliary pointers to every non-leaf node in the TD-tree. Let u' be a child of a non-leaf node u. For each vertex v in the array V_{cand} of a block in the node u, there is a pointer to a block b in u' if the vertex v_{par} recorded in the block b is v. The dashed arrows in Fig. 5 represent these auxiliary points. In the rest of this paper, the vertex v and the auxiliary pointers associated with v are called a *cell* in V_{cand}.

Let l be a leaf node. For any vertex v stored in a block of l, we trace back the linked list pointing to v and encounter a sequence of vertices. This vertex sequence is a candidate matching of the query path from the leaf node l to the root node. For example, in Fig. 5, we can obtain a sequence of vertices $v_5 \rightarrow v_1 \rightarrow v_2$ by tracing back the left-most linked list starting from the vertex v_5 in the first block of the node u_3. This vertex sequence is a candidate matching of the query path $u_3 \rightarrow u_1 \rightarrow u_0$.

Time Instance Sets. As mentioned in the introduction to leaf nodes, in each block of a leaf node, every vertex v in the array V_{cand} is associated with a set *TS* of time instances. The sequence of vertices obtained by tracing back the linked list pointing to v is presented at all these time instances. Let us consider the left time instance set $TS = \{1, 2, 4, 5\}$ in the leaf node u_3 in Fig. 5. The query path from the root u_0 to the leaf u_3 is $u_0 \rightarrow u_1 \rightarrow u_3$. One candidate matching of this query path is $v_2 \rightarrow v_1 \rightarrow v_5$. Therefore, *TS* is obtained by intersecting the time instance sets on the edges (v_2, v_1) and (v_1, v_5), i.e., $\{1, 2, 3, 4, 5\} \cap \{1, 2, 4, 5\} = \{1, 2, 4, 5\}$.

5.2 TD-Tree Construction

Next, we introduce the algorithm for constructing a TD-tree. Given the temporal graph $\mathcal{G}$, the query graph Q, the query tree T, and the minimum duration threshold k, the TD-tree is constructed in two stages: *tree growth* and *tree trimming*.

5.2.1 Stage 1: TD-Tree Growth

In this stage, we fully grow the TD-tree by adding candidate matching data vertices to each TD-tree node. This stage basically runs as follows: We start with a TD-tree skeleton that has the same structure as the query tree T. All nodes in the TD-tree are initially empty. We fill all nodes in the DFS order that determines T. For simplicity of presentation, we use Ψ to denote this DFS order in the rest of this paper. First, we fill the root r by adding to the array $r.V_{cand}$ all the vertices in $\mathcal{G}$ that satisfy all the conditions specified in the "Root Node" part of Section 5.1. Then, we fill all the other nodes in the DFS order. Specifically, let c be the current node to be filled and p be the parent of c. For each vertex v in the array V_{cand} of a block in p (or for each vertex v in the array V_{cand} if p is the root), the node c is filled as follows:

1) We add a new block b with $b.v_{par} = v$ and $b.V_{cand} = \emptyset$ to the node c and associate an auxiliary pointer to the block b with the vertex v.
2) We find all the edges (v, v') in $\mathcal{G}$ such that the edge (v, v') and the vertex v' satisfy all the conditions specified in the "Internal Node" part of Section 5.1.
3) For each edge (v, v'), we add v' to the array $b.V_{cand}$. If c is a leaf node, we attach a set of time instances with v'. The collection of this time instance set will be presented later in Algorithm 1.

The above rudimentary TD-tree growth method can usually generate false candidate matchings for some vertices in Q. As mentioned in Section 3, the non-tree edges in Q are useful for filtering out some of these false candidate vertex matchings. Now, we show how to do such filtering during the TD-tree growth. Let c be the current node to be filled, and there is a non-tree edge (u, c) in Q where u is an ancestor of c (as discussed in Section 4, since T is a DFS tree, any non-tree edge in Q must link a vertex to one of its ancestors in T). In Step 3 of the rudimentary TD-tree growth method, before adding the vertex v' to the array $b.V_{cand}$, we can check if v' is a false candidate matching of c by the following *Non-Tree Edge Verification Rule*: *if any vertex v'' in any blocks in the TD-tree node u does not satisfy that (v'', v') is an edge in $\mathcal{G}$, we have that v' is not a candidate matching of c because no vertex in V_G can match u if c is mapped to v'.*

A simple implementation is checking if there is a neighbor of v' with the same label as u that is contained in a block in the node u. The drawback of this simple implementation is that all vertices within all blocks of u are visited in the worst case. To support a fast vertex containment test, we use a Bloom filter in each non-leaf node. A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For all vertices v in all blocks in a TD-tree node, we add v to the Bloom filter in this node. Since a Bloom filter never introduces false negatives, all correct matchings of c must be contained in the blocks in the node c.

The TD-tree growth algorithm is described in Algorithm 1. In line 1, the function `init_tree` initializes the TD-tree $\mathcal{T}$ that has the same structure as the input query tree T. All nodes in $\mathcal{T}$ are initially empty. In line 2, the procedure `fill_root` fills the root node $\mathcal{T}.root$ of $\mathcal{T}$. The detailed steps of `fill_root` have been introduced earlier. To support a fast vertex containment test in $\mathcal{T}.root$, when a data vertex v is added to the array $\mathcal{T}.root.V_{cand}$, v is also added to the Bloom filter in $\mathcal{T}.root$. In lines 3–5, all nodes in $\mathcal{T}$ except the root are filled by the recursive procedure `fill_node` in the order specified by Ψ. Procedure `fill_node` takes as input the node c to be filled, a data vertex v that matches the parent of c and a set TS of time instances. Since `fill_node` is recursively applied to all nodes except the root in the DFS order, when it is applied on the node c, each ancestor of c is currently mapped a specific vertex in $\mathcal{G}$. Particularly, the parent of c is mapped to v. Let $v_1, v_2, \ldots, v_m (v_m = v)$ be the vertices that the ancestors of c (from the root of T to the parent of c) are currently mapped to, respectively. The set TS contains all time instances at which all the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m)$ exist, that is, $TS = \bigcap_{i=1}^{m-1} TS(v_i, v_{i+1})$.

Algorithm 1. GrowTDTree

```
  Input: the temporal graph $\mathcal{G}_{[1,n]}$, the query graph $Q$, the
      query tree $T$, and the minimum duration threshold
      $k$ ($\mathcal{G}[1,n], Q, T, k$ are used as immutable global variables
      in this algorithm)
  Output: a fully grown TD-tree $\mathcal{T}$
 1: $\mathcal{T} \leftarrow$ init_tree($T$)
 2: fill_root($\mathcal{T}.root$)
 3: for all child node $c$ of $\mathcal{T}.root$ do
 4:   for all vertex $v$ in $\mathcal{T}.root.V_{cand}$ do
 5:     fill_node($c, v, \{1, 2, \ldots, n\}$)
 6: return $\mathcal{T}$
    Procedure fill_node($c, v, TS$)
 7: add a new block $b$ with $b.v_{par} = v$ and $b.V_{cand} = \emptyset$ to $c$
 8: associate a pointer to the block $b$ with $v$
 9: for all neighbor $v'$ of $v$ in $\mathcal{G}$ do
10:   if tree_edge_test($v', v, c, c.parent$) = false then
11:     continue
12:   if non_tree_edge_test($v', c$) = false then
13:     continue
14:   if $|TS \cap TS(v, v')| < k$ then
15:     continue
16:   add $v'$ to the array $b.V_{cand}$ and the Bloom filter in $c$
17:   if $c$ is a leaf node then
18:     attach $TS \cap TS(v, v')$ to $v'$
19:   else
20:     for all child node $c'$ of $c$ do
21:       fill_node($c', v', TS \cap TS(v, v')$)
```

Procedure `fill_node` basically follows the steps introduced earlier. First, we add a new block b with $b.v_{par} = v$ and $b.V_{cand} = \emptyset$ to the node c (line 7) and associate a pointer to the block b with the vertex v (line 8). For each vertex v' that is adjacent to v in $\mathcal{G}$, we test whether v' is possibly a candidate matching of c in lines 10–15. Specifically, in line 10, the function `tree_edge_test` tests if the edge (v, v') and the vertex v' satisfy all the conditions specified in the "Internal Node" part of Section 5.1. In line 12, the function `non_tree_edge_test` tests if v' satisfies the condition imposed in the above non-tree edge verification rule. With the Bloom filters in all non-leaf nodes, we implement `non_tree_edge_test` as follows: Let $(u_1, c), (u_2, c), \ldots, (u_l, c)$ be all non-tree edges in Q that link c to its ancestors in T. For all vertices v'' that is adjacent to v' in $\mathcal{G}$, we test if v'' is not contained in the Bloom filters of any nodes $u_1, u_2, \ldots, u_l$. If it is the case, v'' must satisfy the condition described in the above non-tree edge verification rule

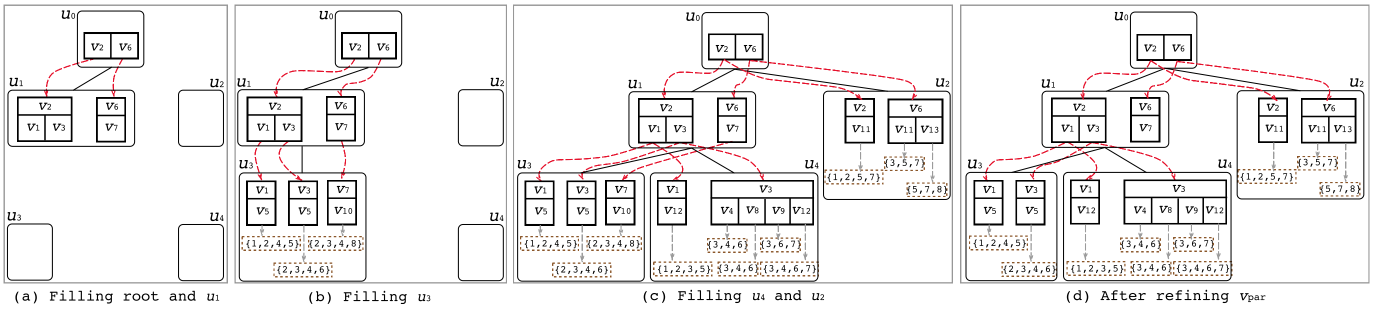

(a) Filling root and u_1
(b) Filling u_3
(c) Filling u_4 and u_2
(d) After refining v_{par}

Fig. 7. The growth of the TD-tree given the temporal graph $\mathcal{G}$ in Fig. 3, the query graph Q in Fig. 4a, the query tree T in Fig. 4b, and the minimum duration threshold $k = 3$.

because a Bloom filter does not introduce false positives. In line 14, we compute $TS \cap TS(v, v')$, the set of time instances at which all the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v, v')$ exist. If $|TS \cap TS(v, v')| < k$, the duration of the subgraph induced by these edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m), (v, v')$ is less than k. Definitely, none of the matchings of Q can contain this subgraph. If v' can pass all these tests in lines 10–15, we add v' to the array $b.V_{cand}$ and the Bloom filter in c (line 16). If c is a leaf node, we associate the time instance set $TS \cap TS(v, v')$ to v' (line 18); Otherwise, we recursively apply Procedure `fill_node` to all children c' of c in the order that they are kept in T (lines 20–21).

In Algorithm 1, Procedure `fill_node` is first called in line 5 to fill each child node c of the root in the order that they are kept in T based on each candidate matching vertex v of the root stored in $\mathcal{T}.root.V_{cand}$. The set of time instances given as input to this call of `fill_node` is $\{1, 2, \ldots, n\}$, the universal set of time instances in the temporal graph $\mathcal{G}$.

For example, given the temporal graph $\mathcal{G}$ in Fig. 3, the query graph Q in Fig. 4a, the query tree T in Fig. 4b, and the minimum duration threshold $k = 3$, the growing process of the TD-tree executed by Algorithm 1 is illustrated in Fig. 7.

5.2.2 Step 2: TD-Tree Trimming

Although many false candidate matching vertices of the vertices in Q have already been filtered out in Step 1, a node in the TD-tree obtained after Step 1 may still contain some false candidate matching vertices. In Step 2, we propose a trimming method to additionally remove as many false candidate matching vertices as possible from the TD-tree. Let us first look at the following lemma which is the foundation of our trimming method.

Lemma 1. *Let u be a non-leaf node and v be a candidate matching vertex of u stored in a block of u. If v does not match u in any matching of Q, all the blocks pointed to by the pointers associated with v can be removed from the TD-tree.*

According to Lemma 1, removing a block from a child node c of u can trigger the removal of some blocks from the children of c. Such block removal operations can cascade downwards in the tree until it reaches the leaf nodes, thereby removing a lot of false candidate matching vertices from the TD-tree.

Next, we introduce how to identify false candidate matching vertices of a query vertex in Q. Removing these false candidate matching vertices from the TD-tree can trigger cascaded block removals. Recall that, the function `non_tree_edge_test` in Algorithm 1 tests if a vertex in $\mathcal{G}$ is not possible to be a candidate matching of a query node u based on the candidate matching vertices of the ancestors of u that are linked to u by non-tree edges in Q. Similar to this method, we can identify false candidate matching vertices of a non-leaf node based on the candidate matching vertices of its children according to the following lemma.

Lemma 2. *Let u be a non-leaf node and v be a candidate matching vertex of u stored in a block of u. For a child node c of u, if there is no pointer associated with v pointing to any blocks in c, then v does not match u in any matchings of Q.*

Based on Lemma 1 and Lemma 2, we propose the TD-tree trimming algorithm in Algorithm 2, which works in two passes: *top-down pass* and *bottom-up pass*.

Algorithm 2. TrimTDTree

Input: a grown TD-tree $\mathcal{T}$
Output: the trimmed TD-tree $\mathcal{T}$

```
1:  for all non-leaf node $u$ in $\mathcal{T}$ in preorder do
2:    for all block $b$ in $u$ do
3:      for all vertex $v$ in $b.V_{cand}$ do
4:        $C \leftarrow$ the set of nodes that contain the blocks pointed
          to by the pointers associated with $v$
5:        if $|C| < |u.children|$ and non_edge_edge_test$(v, u) =$
          false then
6:          for all block $b'$ pointed to by the pointers associated
            with $v$ do
7:            remove_block$(b', u)$
8:          remove $v$ from $b.V_{cand}$
9:      if $b.V_{cand} = \emptyset$ then
10:       remove $b$ from $u$
11: execute lines 1–10 except that all non-leaf nodes in $\mathcal{T}$ are
    visited in postorder
12: return $\mathcal{T}$
    Procedure remove_block$(b, u)$
13: if $b$ is contained in a non-leaf node then
14:   for all vertex $v$ in $b$ do
15:     for all block $b'$ pointed to by the pointers associated
        with $v$ do
16:       remove_block$(b', u)$
17: remove $b$ from $u$
```

Top-Down (Preorder) Pass. In the top-down pass (lines 1–10), the algorithm visits all non-leaf nodes u in the TD-tree in *preorder*. For each candidate matching vertex v in each block b of u, we test if v cannot match u in any matchings of Q by

Lemma 2 (lines 4–5). If it is the case, we remove the useless blocks from the TD-tree according to Lemma 1 (lines 6–7) and remove v from $b.V_{cand}$ (line 8). If $b.V_{cand}$ becomes empty, we remove the block b from the node u (line 10).

Bottom-Up (Postorder) Pass. The bottom-up pass (line 11) is the same as the top-down pass except that the algorithm visits all non-leaf nodes in the TD-tree in *postorder*.

5.2.3 TD-Tree Construction Algorithm

Given the temporal graph $\mathcal{G}_{[1,n]}$, the query graph Q, the query tree T, and the minimum duration threshold k, our TD-tree construction algorithm executes GrowTDTree (Algorithm 1) and TrimTDTree (Algorithm 2) in sequence. Finally, a TD-tree is outputted as the result.

5.2.4 Time and Space Complexity

The root of the query tree T has at most $|V_{\mathcal{G}}|$ candidate matching vertices in $\mathcal{G}$. There are at most $|V_{\mathcal{G}}|\Delta^{|V_Q|-1}$ paths in $\mathcal{G}$ from a candidate matching vertex of T's root to a candidate matching vertex of a leaf node in T, where Δ is the maximum degree of vertices in $\mathcal{G}$, and V_Q is the vertex set of the query graph Q. Moreover, it takes $O(n)$ time to compute the intersection of two time instance sets, where n is the number of snapshots in $\mathcal{G}$. Therefore, in the worst case, the time complexity of the TD-tree construction algorithm is $O(n\Delta^{|V_Q|}|V_{\mathcal{G}}|)$, and the space complexity is $O(n|E_{\mathcal{G}}| + n\Delta^{|V_Q|-1}|V_{\mathcal{G}}|)$.

6 Enumeration of Durable Subgraph Matchings

In Phase 3 of our algorithm, we enumerate all k-durable matchings of the query graph Q based on the TD-tree constructed in Phase 2 instead of on the temporal graph $\mathcal{G}$. The TD-tree stores all candidate matching vertices of each vertex in Q. As will be shown in Section 8, the TD-tree is generally much smaller than $\mathcal{G}$ in practice.

6.1 Path-Based Matching Paradigm

There are a lot of ways to generate candidate k-durable matchings of the query graph Q. For example, the simplest way is choosing a matching vertex for each query vertex in Q. However, this straightforward method overlooks the connections between the vertices in Q and often generates many false matchings of Q in practice. In other ways, non-trivial subgraphs of Q such as paths or twigs can be used as the units of matching. Different generation methods vary in their time efficiency and the amount of generated false matchings of Q.

Han *et al.* [8] has shown that the path-oriented subgraph matching paradigm can usually outperform the vertex-oriented matching paradigm. Inspired by this finding, our algorithm also uses the path-oriented approach to enumerating k-durable matchings of Q. Specifically, we decompose the query tree T into a set of query paths $P_1, P_2, \ldots, P_m$, where each path P_i connects the root of T to a leaf node. We enumerate all k-durable matchings of Q based on these paths. First, we find all k-durable matchings of every query path P_i on the TD-tree. Then, for each k-durable matching p_i of each query path P_i, if $p_1, p_2, \ldots, p_m$ are joinable, we join them together to produce a candidate matching of Q. Here, $p_1, p_2, \ldots, p_m$ are joinable if they agree on the mappings for the common vertices in $P_1, P_2, \ldots, P_m$. If this candidate matching of Q is k-durable and satisfies the constraints imposed by all non-tree edges in Q, we output this matching as a result.

Note that any two query paths P_i and P_j are not independent. They must share a common *prefix* path starting from the root of T (at least they meet at the root). For example, in the query tree shown in Fig. 4b, the paths $u_0 \rightarrow u_1 \rightarrow u_3$ and $u_0 \rightarrow u_1 \rightarrow u_4$ share the common prefix path $u_0 \rightarrow u_1$. Obviously, we have the following lemma.

Lemma 3. *A matching of P_i is joinable with a matching of P_j if and only if every vertex on the longest common prefix path between P_i and P_j is mapped to the same vertex in $\mathcal{G}$ within both matchings.*

According to Lemma 3, we do not need to find the matchings of P_i and P_j independently and then test whether they are joinable. Instead, we can first find a matching of P_i and then find a matching of P_j based on the matching of their longest common prefix path within the matching of P_i. The obtained matchings of P_i and P_j are certainly joinable.

Therefore, in our algorithm, we determine a matching order of the query paths, say $P_1, P_2, \ldots, P_m$, and find the matchings of these paths in this order. A matching of P_i is found as follows. Let $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_t$ be the longest postfix path of P_i whose vertices are not contained in all of $P_1, P_2, \ldots, P_{i-1}$. Let u_0 be the parent of u_1 in T and v_0 be the vertex in $\mathcal{G}$ that has been matched to u_0 within the matchings of $P_1, P_2, \ldots, P_{i-1}$. To find a matching of P_i, we just find a matching of the path $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_t$ given that u_0 is mapped to v_0.

From the matching order of these query paths, we can naturally derive a matching order of all vertices in Q. Let $\overline{P}_i$ denote the longest postfix path of P_i whose vertices are not contained in $P_1, P_2, \ldots, P_{i-1}$. According to the path matching order, all vertices in $\overline{P}_i$ are matched prior to all vertices in $\overline{P}_j$ if $i < j$. In $\overline{P}_i$, the vertices are matched in the order from ancestors to descendants. Hence, matching all vertices in Q in this order is equivalent to matching and joining all query paths $P_1, P_2, \ldots, P_m$ in the path matching order introduced above. This vertex matching order also satisfies the property shown in the following lemma.

Lemma 4. *The root of T must be the first matched vertex in this order. For all vertices u in T except the root of T, the parent of u in T must be matched ahead of u according to the matching order of the vertices in Q.*

According to Lemma 4, we can generate a candidate matching of Q by finding a matching vertex for each query vertex u in Q in the vertex matching order, given that the parent of u has been mapped to a vertex previously in $\mathcal{G}$ in this vertex matching order.

6.2 Enumeration Algorithm

The k-durable subgraph matching enumeration algorithm is presented in Algorithm 3. Assume that all vertices in Q are indexed from 1 to $|V_Q|$. In line 1, the function gen_order generates a matching order of all vertices in Q. This function basically executes the steps described above. The optimization of the order will be introduced later in Section 7.1. Without loss of generality, let $u_1, u_2, \ldots, u_{|V_Q|}$ be the generated matching order (it can be realized by reassigning the indexes of all vertices in Q according to this matching order). In line 2, we create an array M to store a candidate matching of Q. The element $M[i]$ holds the vertex in $\mathcal{G}$ that matches the query vertex u_i in this candidate matching. Note that a vertex may be kept in multiple block cells in a TD-tree node. In line 3, we create an array Pos to record the positions of the matching vertices stored in the TD-tree. The element $Pos[i]$ keeps the block cell that holds the matching vertex $M[i]$ of u_i in the TD-tree node corresponding to u_i. In lines 4–7, the algorithm enumerates all k-durable matchings of Q by matching all vertices in Q in the vertex matching order. By Lemma 4, the root of T must be the first matched vertex in the order. Therefore, for each candidate matching vertex v stored in $\mathcal{T}.root.V_{cand}$, we first set $M[1]$ to v (line 5) and set $Pos[1]$ to the cell holding v in $\mathcal{T}.root.V_{cand}$ (line 6). Then, the algorithm calls Procedure expand to generate a matching of Q, given that the root of T is mapped to v.

Algorithm 3. Match

Input: a temporal graph $\mathcal{G}_{[1,n]}$, a query graph Q, a query tree T, a TD-tree $\mathcal{T}$, the minimum durable threshold k ($\mathcal{G}_{[1,n]}, Q, T, k$ are used as immutable global variables in the algorithm)
Output: all k-durable matchings of Q

```
1:  $O \leftarrow$ gen_order($\mathcal{T}$)
2:  $M \leftarrow$ an array of length $|V_Q|$
3:  $Pos \leftarrow$ an array of length $|V_Q|$
4:  for all vertex $v$ in $\mathcal{T}.root.V_{cand}$ do
5:      $M[1] \leftarrow v$
6:      $Pos[1] \leftarrow$ the block cell storing $v$ in $\mathcal{T}.root.V_{cand}$
7:      expand($M, 1, Pos, O$)
    Procedure expand($M, i, Pos, O$)
8:  if $i = |V_Q|$ then
9:      print $M$
10: else
11:     $u_p \leftarrow$ the parent of $u_{i+1}$ in $\mathcal{T}$
12:     $b \leftarrow$ the block in the node $u_i$ of $\mathcal{T}$ that the block cell in
        $Pos[p]$ points to
13:     for all vertex $v$ in $b.V_{cand}$ do
14:         if backward_edge_test($v, M$) = true then
15:             if $u_{i+1}$ is a leaf node in $\mathcal{T}$ then
16:                 if duration_test($v.TS, Pos$) = false then
17:                     continue
18:             if $v$ is not in $M[1:i]$ then
19:                 $M[i+1] \leftarrow v$
20:                 $Pos[i+1] \leftarrow$ the block cell storing $v$ in $b.V_{cand}$
21:                 expand($M, i+1, Pos, O$)
```

Procedure expand takes as input the arrays M and Pos and an integer i called matching index. The matching index i indicates that the first i vertices $u_1, u_2, \ldots, u_i$ in the vertex matching order have been mapped to the vertices in $M[1], M[2], \ldots, M[i]$, respectively. Thus, $M[1:i]$ records a matching of the subgraph of Q induced by $u_1, u_2, \ldots, u_i$, where $M[x:y]$ represents the slice of M composed of the elements from position x to y. There are two cases to be considered in this procedure.

Case 1 (lines 10–21): If $i < |V_Q|$, we continue to choose a matching vertex of u_{i+1}. Let u_p be the parent of u_{i+1} in the TD-tree. By Lemma 4, we have $1 \leq p \leq i$. In the block b in the TD-tree node u_{i+1} pointed to by the pointer associated with the block cell $Pos[p]$, we choose a vertex v from $b.V_{cand}$ as a candidate matching of u_{i+1}. Then, we test whether v can expand $M[1:i]$ to a k-durable matching of the subgraph of Q induced by $u_1, u_2, \ldots, u_{i+1}$ (lines 14–17). The function backward_edge_test tests whether there is an edge from v to $M[j]$ in $\mathcal{G}$ for every backward edge (u_{i+1}, u_j) adjacent to u_{i+1} in Q where $j \leq i$. If this test fails, the current partial matching $M[1:i]$ cannot be expanded to a valid matching of the subgraph of Q induced by $u_1, u_2, \ldots, u_{i+1}$ by mapping u_{i+1} to v. If u_{i+1} is a leaf node, the function duration_test is called to test whether the current matching $M[1:i]$ can be expanded to a k-durable subgraph of $\mathcal{G}$ by mapping u_{i+1} to v. Because u_{i+1} is a leaf node, there is a set of time instances $v.TS$ associated with v in the TD-tree node corresponding to u_{i+1}. We can compute

$$TS = v.TS \cap \bigcap_{1 \leq j \leq i, u_j \text{ is a leaf in } T} Pos[j].TS. \quad (1)$$

The set TS is composed of all time instances at which all edges in the matching obtained based on $M[1:i]$ by mapping u_{i+1} to v exist. If $|TS| < k$, the current matching $M[1:i]$ cannot be expanded to a k-durable subgraph of $\mathcal{G}$ by mapping u_{i+1} to v. If v is validated by these tests, we set $M[i+1]$ to v (line 19) and $Pos[i+1]$ to the block cell holding v in $b.V_{cand}$ (line 20). To further expand the current array M to be a k-durable matching of Q, we make a recursive call to Procedure expand with the input M, Pos and $i+1$ (line 21).

Case 2 (lines 8-9): If $i = |V_Q|$, all vertices in M form a k-durable matching of Q. Thus, M is outputted as a result.

7 Optimizations

In this section, we present some methods to optimize the performance of the durable subgraph enumeration algorithm described in Algorithm 3.

7.1 Selection of Vertex Matching Order

The vertex matching order used in Algorithm 3 is derived from the matching order of the query paths $P_1, P_2, \ldots, P_m$ that connect the root to the leaves of T. There are $m!$ path matching orders. Different orders can usually lead to various amount of false candidate matchings of Q generated by Algorithm 3, thereby significantly affecting the performance of the algorithm. It is not easy to choose the optimal path matching order that can minimize the number of generated false candidate matchings of Q. Therefore, we design a heuristic method to select a suboptimal path matching order. Let $card(i)$ be the total length of the arrays $b.V_{cand}$ in all blocks b in the leaf node on the path P_i. We call $card(i)$ the *matching cardinality* of P_i because it approximates the number of data paths that match P_i within all matchings of Q. We sort all the query paths P_i in decreasing order of $card(i)$. The vertex matching order used in Algorithm 3 is then derived from this sorted order of the query paths.

7.2 Compaction of Time Instance Sets

In a leaf node of the TD-tree, if there are multiple cells in the $b.V_{cand}$ array of a block b, the time instance sets associated with these cells are usually very similar. For example, in Fig. 5, there are 4 cells in the V_{cand} array of the right block in

Vertex	Time instance set
v_4	$\{3, 4, 6\}$
v_8	$\{3, 4, 6\}$
v_9	$\{3, 6, 7\}$
v_{12}	$\{3, 4, 6, 7\}$

(a) Before compaction.

$\overline{TS} = \{3, 4, 6, 7\}$

Vertex	Time instance set
v_4	$\{7\}$
v_8	$\{7\}$
v_9	$\{4\}$
v_{12}	$\{\}$

(b) After compaction.

Fig. 8. Compaction of time instance sets.

the leaf node u_4. We observe that the time instance sets associated with these four cells are very similar. It is because the data paths obtained by backtracking the pointers that point to the cells in block b share the same longest prefix path. Therefore, these data paths almost appear at the same time instances. For example, in Fig. 5, by backtracking the pointers that point to the 4 cells in the V_{cand} array of the block on the right in the leaf node u_4, we obtain 4 data paths $v_2 \rightarrow v_3 \rightarrow v_4$, $v_2 \rightarrow v_3 \rightarrow v_8$, $v_2 \rightarrow v_3 \rightarrow v_9$, and $v_2 \rightarrow v_3 \rightarrow v_{12}$. These paths share the same common prefix path $v_2 \rightarrow v_3$. Hence, these paths appear at the nearly same time instances.

According to the observation described above, if there are multiple cells in the array $b.V_{cand}$ of a block b in a leaf node of the TD-tree, a lot of space is wasted to separately store all the time instance sets associated with these cells because these time instance sets usually have significant overlaps. Therefore, we re-design the internal structure of the block b to reduce space overhead. First, we record with b the union $\overline{TS}$ of the time instance sets associated with all cells in $b.V_{cand}$. Second, for each cell in $b.V_{cand}$, we substitute its associated time instance set by the difference between $b.\overline{TS}$ and its original time instance set. The original time instance sets can be easily computed based on $\overline{TS}$. For example, Fig. 8 shows the time instance sets of the vertices in the block on the right of the leaf node u_4 in Fig. 5 before and after time instance set compaction.

The new structure of a leaf node's block b has two advantages. First, it can generally reduce the space overhead because the redundant time instances in the time instance sets originally associated with the cells in $b.V_{cand}$ are stored only once in $b.\overline{TS}$. As will be shown in Fig. 13, this new representation can reduce the size of a TD-tree by about 8 percent.

Second, $b.\overline{TS}$ is a superset of all the time instance sets associated with the cells in $b.V_{cand}$. This fact can be taken advantage of to improve the efficiency of Algorithm 3. Recall that if the ith matched vertex u_i in the vertex matching order is a leaf node in T, the function duration_test is called to test the duration of the partial matching $M[1 : i]$ by Eq. (1). After compacting the time instance sets associated with the block cells in the TD-tree node u_i, a variant of this duration test can be executed before line 16 of Algorithm 3 to filter out multiple invalid candidate matching vertices of u_i simultaneously. Specifically, let b be the block pointed to by the pointer associated with the block cell stored in $Pos[i-1]$. Every vertex in the array $b.V_{cand}$ is a candidate matching vertex of u_i. Since u_i is a leaf node in $\mathcal{T}$, we have the union time instance set $b.\overline{TS}$. Before line 16 is executed, we compute a set $\overline{TS}$ of time instances by Eq. (1), where the last term $Pos[i].TS$ is replaced with $b.\overline{TS}$. If $|\overline{TS}| < k$, none of the vertices in $b.V_{cand}$ can match u_i, given that the $i-1$ vertices preceding u_i in the vertex matching order have been mapped to the data vertices in $M[1], M[2], \ldots, M[i-1]$, respectively. Therefore, the entire block b can be skipped in Algorithm 3, thereby reducing the computational cost.

TABLE 1
Statistics of Datasets

Dataset	# Vertices	# Edges	Time span	File size
SuperUser	194K	1.4M	2,773 days	34MB
WikiTalk	1.1M	7.8M	2,320 days	176MB
StackOverflow	2.6M	63M	2,774 days	1.64GB
Bitcoin	48M	113M	2,584 days	3.01GB

8 Experiments

In this section, we experimentally evaluate the performance of the proposed durable subgraph matching algorithm. We call this algorithm DSM in the experiments.

8.1 Experimental Setup

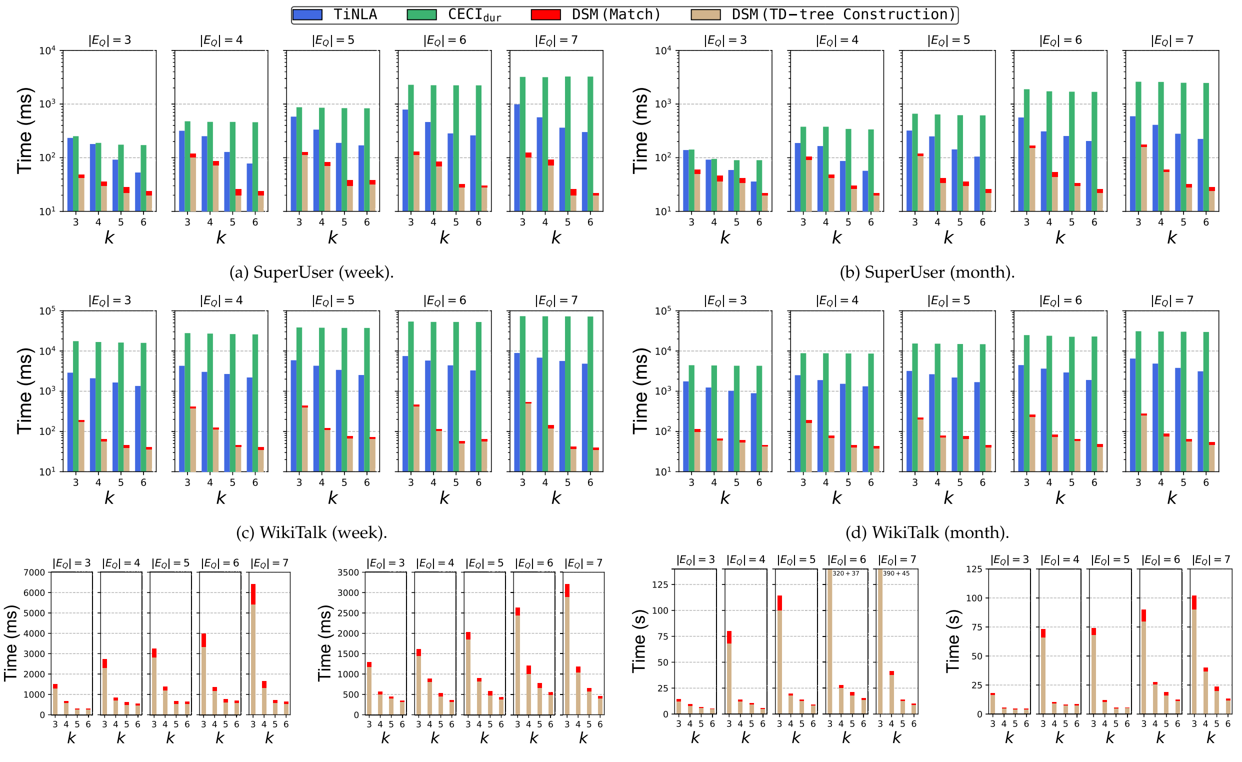
Algorithms. In the experiments, our algorithm DSM was compared with two algorithms. The first one is the state-of-the-art durable subgraph matching algorithm on temporal graphs proposed by Semertzidis and Pitoura [15]. As shown in [15], this algorithm works better on the index called TiNLA than on the other indices in terms of both memory overhead and time efficiency. Therefore, we implemented this algorithm based on the TiNLA index and call it TiNLA in our experiments. The other algorithm is developed based on CECI [26], a state-of-the-art subgraph matching algorithm on static graphs. Given a temporal graph $\mathcal{G}$, a query graph Q and the minimum duration threshold k, this algorithm first finds all matchings of Q on each snapshot of $\mathcal{G}$ by CECI. We count the number of occurrences of each matching in the snapshots of $\mathcal{G}$ using a hash table. If the occurrence number of a matching is not less than k, the matching is outputted as a k-durable matching of Q in $\mathcal{G}$. This algorithm is called CECI$_{dur}$ in our experiments. All these algorithms were implemented in C++ without multithreading.

Datasets. The experiments were carried out on 4 real temporal graphs, namely SuperUser, WikiTalk, StackOverflow, and Bitcoin[1]. In the temporal graphs SuperUser and StackOverflow, a directed temporal edge (u, v, t) represents that user u answered user v's question or u commented on v's answer at time t. In the tempral graph WikiTalk, a directed temporal edge (u, v, t) means that user u edited user v's talk page at time t. In the temporal graph Bitcoin, a directed temporal edge (u, v, t) represents that a block that contains a Bitcoin transaction from account u to account v was created on the blockchain. The statistics of these datasets are shown in Table 1. Since these temporal graphs are unlabeled, we assigned 5 labels to their vertices uniformly at random.

To vary the number of snapshots of a temporal graph, we partition the whole time interval of the temporal graph into a series of consecutive non-overlapping time intervals of equal length and put all edges that occur within a time interval into a snapshot. In the experiments, the length of a time interval is varied from a week to a month.

Queries. In the experiments, the query graphs on a temporal graph $\mathcal{G}$ were generated using random walks on $\mathcal{G}$. Given the desired number m of edges in the generated

1. The temporal graphs SuperUser, WikiTalk, and StackOverflow are obtained from http://snap.stanford.edu, and the temporal graph Bitcoin is obtained from http://www.cs.cornell.edu/~arb/data/index.html



(a) SuperUser (week). (b) SuperUser (month).

(c) WikiTalk (week). (d) WikiTalk (month).

(e) StackOverflow (week). (f) StackOverflow (month). (g) Bitcoin (week). (h) Bitcoin (month).

Fig. 9. Average query processing time of DSM, TiNLA and CECI$_{dur}$.

query graph and the minimum duration threshold k, a query graph is generated using the following algorithm. First, we initialize a set V of vertices and a set E of edges to be empty. Then, we select a vertex u from $\mathcal{G}$ uniformly at random and add it to V. While $|E| < m$, we select a vertex v that is adjacent to u in $\mathcal{G}$ uniformly at random. If there is no such adjacent vertices v, the algorithm returns "Nothing". If all the edges in $E \cup \{(u, v)\}$ occur in at least k time instances, we add v to V, add the edge (u, v) to E, and set u to v; otherwise, the algorithm returns "Nothing". When $|E| = m$, the graph (V, E) is returned as a query graph. This algorithm guarantees that if it does not return "Nothing", the outputted query graph consists of m edges and is connected and k-durable.

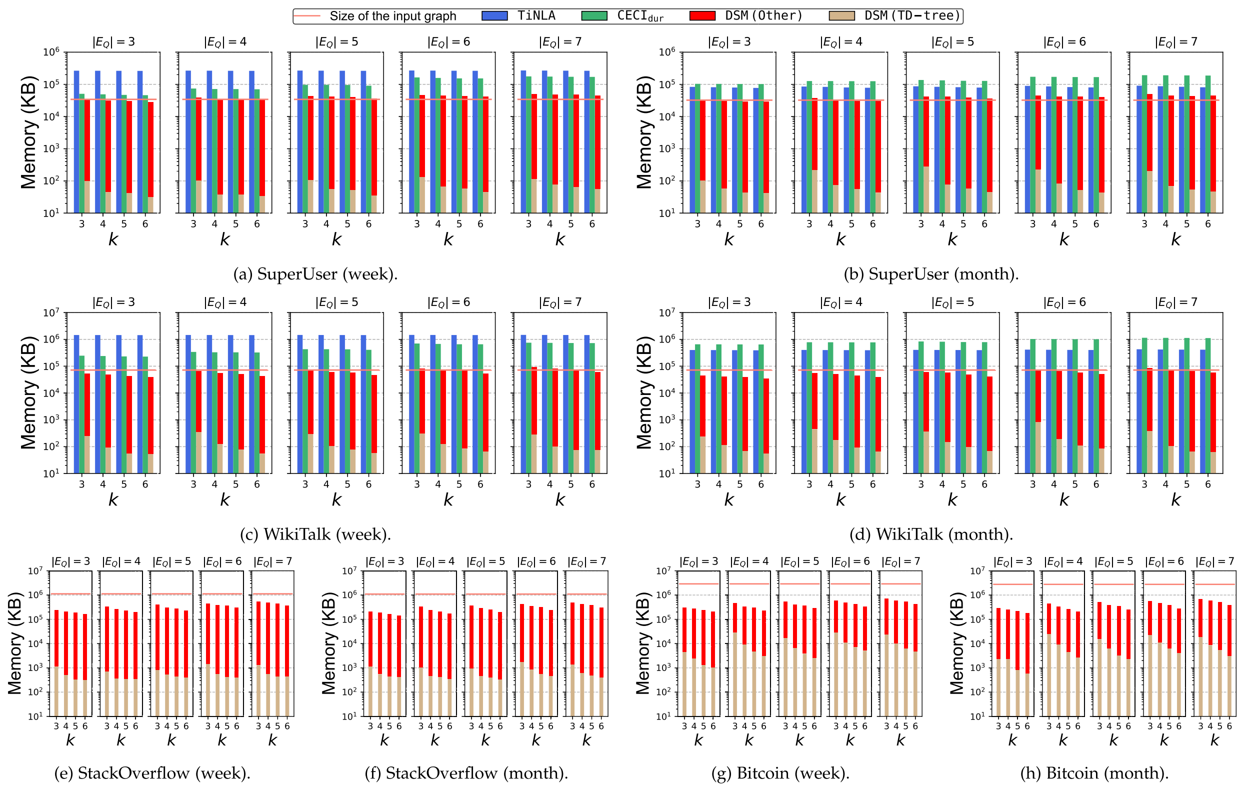
To generate query graphs of various sizes and duration, m is varied from 3 to 7, and k is varied from 3 to 6. Given a temporal graph, we generated 500 query graphs for each specific m and k using the algorithm described above and selected 10 cyclic query graphs Q with the highest density $|E_Q|/|V_Q|$, i.e., the smallest $|V_Q|$ since $|E_Q| = m$.

Hardware. All the experiments were carried out on a computer with Intel Core i5 CPU (2.4GHz and 8 cores), 16GB of DRAM, and 1TB of SSD, running Windows 10. In every experiment, each tested algorithm was executed 5 times in a single thread, and the averages of its performance metrics (execution time, memory usage, etc.) were recorded.

8.2 Query Processing Time

First, we compared our DSM algorithm with the algorithms TiNLA and CECI$_{dur}$ in terms of their running time to process a query. Fig. 9a shows the experimental results obtained on the temporal graph SuperUser, where the time interval of a snapshot is set to a week. When the number m of edges in a query graph is varied from 3 to 7, and the minimum duration threshold k is varied from 3 to 6, the average execution time of these algorithms to process 10 queries generated for m and k is plotted in Fig. 9a. In particular, the execution time of DSM includes the TD-tree construction time and the matching enumeration time (the query decomposition time is insignificant in the execution time). It can be seen that DSM is an order of magnitude faster than TiNLA and 2 orders of magnitude faster than CECI$_{dur}$. Fig. 9b shows the experimental resutls on SuperUser, where the time interval of a snapshot is set to a month. We can see that DSM is also several orders of magnitude faster than TiNLA and CECI$_{dur}$. The experimental results obtained on the temporal graph WikiTalk are shown in Figs. 9c and 9d, where the time intervals of a snapshot are set to a week and a month, respectively. These experimental results are very similar to those obtained on SuperUser shown in Figs. 9a and 9b. The experimental results on StackOverflow and Bitcoin are shown in Figs. 9e, 9f, 9g, and 9h. Note that TiNLA and CECI$_{dur}$ cannot terminate in reasonable time on StackOverflow and Bitcoin due to their high memory footprint, whereas DSM can find all k-durable subgraph matchings of all the queries on StackOverflow and Bitcoin. For various temporal graphs and various queries, the TD-tree construction time accounts for 83%–93% of the execution time of DSM.

When the number m of edges in a query graph is fixed, the average execution time of DSM to process 10 queries with m edges decreases as the minimum duration threshold k increases. This is because, given a query graph, the number of k-durable matchings decreases as k becomes larger, and more false candidate vertex matchings and more false candidate subgraph matchings can be filtered out as k increases. When k is fixed, the average execution time of DSM to process 10 queries with m edges generally increases as m becomes larger. This is because more query edges



(a) SuperUser (week). (b) SuperUser (month). (c) WikiTalk (week). (d) WikiTalk (month). (e) StackOverflow (week). (f) StackOverflow (month). (g) Bitcoin (week). (h) Bitcoin (month).

Fig. 10. Average memory usage of DSM, TiNLA and CECI$_{dur}$.

need to be checked in Phase 2 and matched in Phase 3, and therefore, the recursion of Procedure `expand` (Algorithm 3) generally goes down deeper and takes more time.

8.3 Memory Usage

We also compared DSM with TiNLA and CECI$_{dur}$ in terms of their memory usage. In the experiments reported in Section 8.2, we also kept track of the average size of the memory used by each algorithm to process 10 queries. Fig. 10 shows the memory usage of these algorithms on SuperUser, WikiTalk, StackOverflow, and Bitcoin. The bars in Fig. 10 represent the memory used by these algorithms except the memory for storing the input temporal graph. The size of the input temporal graph is indicated by the red line. In particular, for DSM, we show the memory for storing the TD-tree and the other memory used by DSM itself. It can be seen that the memory for storing the input temporal graph accounts for about 50% of the memory usage of DSM, and the memory for storing the TD-tree accounts for about 1% of the memory usage of DSM. The TD-tree is at least 2 orders of magnitude smaller than the input temporal graph, thereby significantly accelerating the enumeration of candidate subgraph matchings in Phase 3 of DSM.

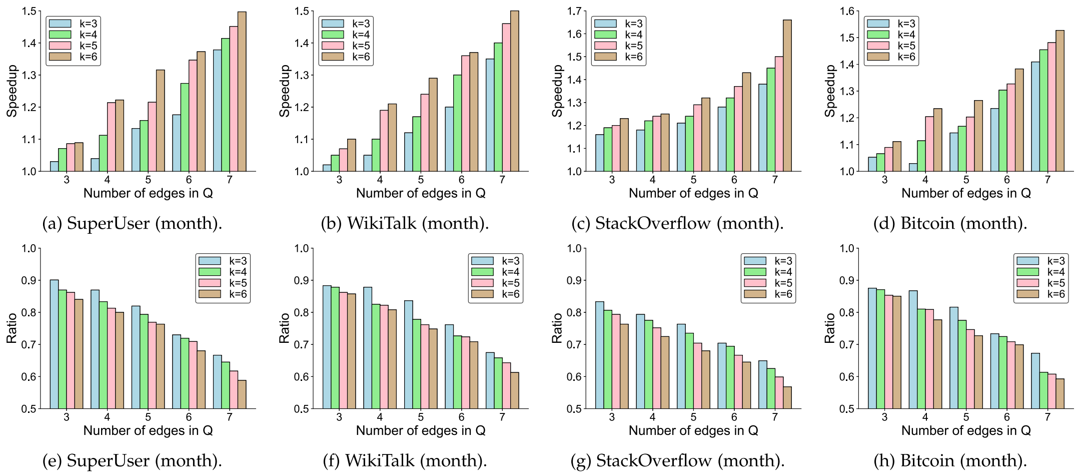
Figs. 10a and 10b show the memory usage of these algorithms on SuperUser. It can be seen that TiNLA uses 3.6$\times$ and 1.5$\times$ more memory than DSM, and CECI$_{dur}$ uses 1.3$\times$ and 2.1$\times$ more memory than DSM when the time interval of a snapshot is set to a week and a month, respectively. Figs. 10c and 10d show the memory usage of these algorithms on WikiTalk. It can be seen that TiNLA uses 10.5$\times$ and 3$\times$ more memory than DSM, and CECI$_{dur}$ uses 2.3$\times$ and 5.4$\times$ more memory than DSM when the time interval of a snapshot is set to a week and a month, respectively.

Figs. 10e, 10f, 10g, and 10h only show the memory usage of DSM on StackOverflow and Bitcoin because TiNLA and CECI$_{dur}$ used up all the available main memory to process any queries on StackOverflow and Bitcoin. However, DSM requires up to 1.9GB and 3.6GB of main memory on StackOverflow and Bitcoin, respectively. In the memory used by DSM, it requires 1.34GB and 2.67GB memory to store StackOverflow and Bitcoin, respectively.

8.4 Effects of DFS-Based Query Tree Generation

In Section 4, a query graph Q is decomposed into a query tree T and a set of non-tree edges. The query tree T is obtained by carrying out a depth-first search (DFS) on Q. To show the effectiveness of our DFS-based query tree generation scheme, we also develop a query tree generation scheme that carries out a breadth-first search (BFS) on Q as in the state-of-the-art subgraph matching algorithms on static graphs [7], [8], [26], where the root is selected in the same way as in our DFS-based scheme. In this experiment, we compared the execution time of our DSM algorithm that uses the DFS-based query tree generation scheme (DSM$_{DFS}$ for short) with the execution time of DSM that uses the BFS-based query tree generation scheme (DSM$_{BFS}$ for short).

Figs. 11a, 11b, 11c, and 11d show the speedup of DSM$_{DFS}$ with respect to DSM$_{BFS}$, that is, the average ratio of the execution time of DSM$_{BFS}$ to the execution time of DSM$_{DFS}$ to process 10 queries. It can be seen that DSM$_{DFS}$ runs faster than DSM$_{BFS}$, and the speedup increases with k and generally increases with the number of edges in the query graph. Figs. 11e, 11f, 11g, and 11h show the average ratio of the size of the TD-tree constructed by DSM$_{DFS}$ to the size of the TD-tree constructed by DSM$_{BFS}$ in this experiment. We can see that the TD-trees constructed by DSM$_{DFS}$ are up to 40%



(a) SuperUser (month). (b) WikiTalk (month). (c) StackOverflow (month). (d) Bitcoin (month).

(e) SuperUser (month). (f) WikiTalk (month). (g) StackOverflow (month). (h) Bitcoin (month).

Fig. 11. Comparison between the DFS-based query tree generation scheme and the BFS-based query tree generation scheme.

smaller than those constructed by $\mathsf{DSM_{BFS}}$, and the decreasing ratio of the TD-tree size increases with k and the number of edges in the query graph. The experimental results in Fig. 11 were obtained when the time interval of a snapshot was set to a month. When the time interval is set to a week, the experimental results are very similar, which are not presented in the paper due to space limit. In summary, this experiment verifies that the DFS-based query tree generation scheme is more suitable for the durable subgraph matching problem on temporal graphs than the classic BFS-based query tree generation scheme.

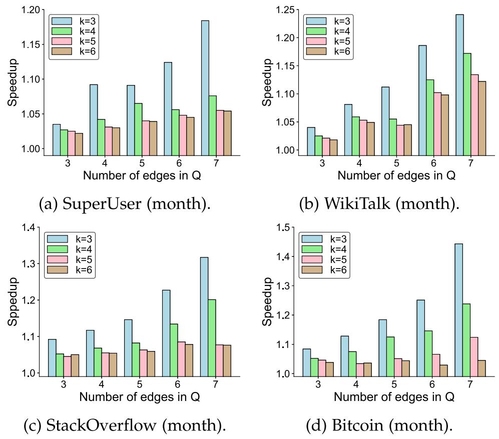
8.5 Effects of Sorting-Based Vertex Matching Order

The matching order of the vertices in the query graph Q has a potential impact on the time for matching enumeration (Phase 3) of the DSM algorithm. In Section 7.1, we propose a heuristic method to approximately select the optimal vertex matching order for Q that can minimize the matching enumeration time of DSM. Our vertex matching order is obtained by sorting the query paths in the query tree T. To test the effectiveness of such order, we compared it with other vertex matching orders derived from random orders of the query paths in T. In this experiment, we compared the execution time of DSM that uses our vertex matching order derived from a sorted order of query paths ($\mathsf{DSM_{sort}}$ for short) with the execution time of DSM that uses a vertex matching order derived from a random order of query paths ($\mathsf{DSM_{rand}}$ for short). Since $\mathsf{DSM_{sort}}$ and $\mathsf{DSM_{rand}}$ use the same query tree T, they spend the same time on TD-tree construction. Therefore, we only compared the matching enumeration time of $\mathsf{DSM_{sort}}$ and $\mathsf{DSM_{rand}}$ in this experiment. Note that the query graphs whose query trees contain at least 3 query paths are used in this experiments.

Fig. 12 shows the speedup of $\mathsf{DSM_{sort}}$ with respect to $\mathsf{DSM_{rand}}$, that is, the average ratio of the matching enumeration time of $\mathsf{DSM_{rand}}$ to the matching enumeration time of $\mathsf{DSM_{sort}}$ for processing 10 queries. We can see that $\mathsf{DSM_{sort}}$ enumerates candidate matchings faster than $\mathsf{DSM_{rand}}$. The speedup normally decreases with k and increases with the number of edges in the query graph. The experimental results show that our sort-based vertex matching order is a better choice than random vertex matching orders.

8.6 Effects of Time Instance Set Compaction

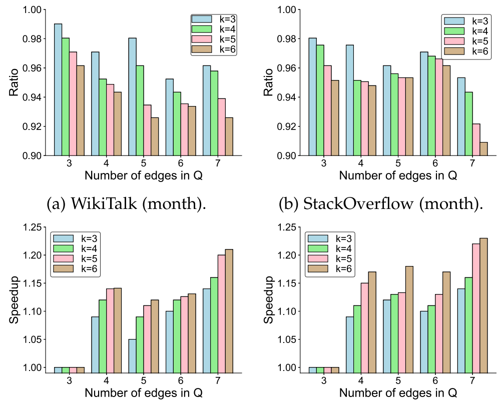
In Section 7.2, we propose an optimization method to compact the time instance sets in the leaf nodes of the TD-tree. To test the effectiveness of this method, we compared the execution time and the memory usage of DSM and its variant that does not use time instance set compaction ($\mathsf{DSM_{nc}}$ for short). Figs. 13a and 13b show the average ratio of the size of the TD-tree constructed by DSM to the size of the TD-tree constructed



(a) SuperUser (month). (b) WikiTalk (month).

(c) StackOverflow (month). (d) Bitcoin (month).

Fig. 12. Comparison between the sort-based vertex matching order and the random-based vertex matching order.



(a) WikiTalk (month). (b) StackOverflow (month).

(c) WikiTalk (month). (d) StackOverflow (month).

Fig. 13. Effects of time instance set compaction.

by $\mathsf{DSM}_{\mathsf{nc}}$ for processing 10 queries. It can be seen that the TD-tree constructed by DSM is slightly smaller than that constructed by $\mathsf{DSM}_{\mathsf{nc}}$. Figs. 13c and 13d show the speedup of DSM with respect to $\mathsf{DSM}_{\mathsf{nc}}$, that is, the average ratio of the matching enumeration time of $\mathsf{DSM}_{\mathsf{nc}}$ to the matching enumeration time of DSM. The TD-tree construction time of DSM and $\mathsf{DSM}_{\mathsf{nc}}$ is comparable. We can see that DSM enumerates candidate matchings faster than $\mathsf{DSM}_{\mathsf{nc}}$. This experiment verifies that the time instance set compaction method can slightly reduce the TD-tree size and improve the matching enumeration efficiency of DSM.

9 Conclusion

In this paper, a new algorithm is proposed to solve the durable subgraph matching problem on temporal graphs. This algorithm adopts several effective techniques in its three phases. In Phase 1, our DFS-based query tree generation scheme outperforms the BFS-based scheme that has been widely applied in the state-of-the-art subgraph matching algorithms on static graphs. In Phase 2, the TD-tree constructed by our algorithm acts as a compact representation of candidate data vertices that match the vertices in the query graph. The TD-tree is smaller than the input temporal graph and facilitates matching enumeration in Phase 3 of the algorithm. In Phase 3, the sort-based vertex matching order is effective in reducing the number of enumerated false candidate matchings of the query graph. Our algorithm is an order of magnitude faster and is more memory-efficient than the state-of-the-art algorithm.

References

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

[2] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proc. Clustering Valid. Very Large Databases*, vol. 11, no. 4, pp. 420–431, 2017.

[3] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," in *Proc. Int. Conf. Des. Mater.*, 2002, pp. 721–724.

[4] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, "SubGemini: Identifying subcircuits using a fast subgraph isomorphism algorithm," in *Proc. 30th ACM/IEEE Des. Automat. Conf.*, 1993, pp. 31–37.

[5] E. K. Wong, "Model matching in robot vision by subgraph isomorphism," *Pattern Recognit.*, vol. 25, no. 3, pp. 287–303, 1992.

[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, Oct. 2004.

[7] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 1199–1214.

[8] W. Han, J. Lee, and J. Lee, "$\text{Turbo}_{\text{iso}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 337–348.

[9] F. N. Afrati, D. Fotakis, and J. D. Ullman, "Enumerating subgraphinstances using map-reduce," in *Proc. Int. Conf. Data Eng.*, 2013, pp. 62–73.

[10] L. Lai, L. Qin, X. Lin, and L. Chang, "Scalable subgraph enumeration in mapreduce," *Proc. Clustering Valid. Very Large Databases*, vol. 8, no. 10, pp. 974–985, 2015.

[11] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. Clustering Valid. Very Large Databases*, vol. 5, no. 9, pp. 788–799, 2012.

[12] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 625–636.

[13] X. Ren, J. Wang, W. Han, and J. X. Yu, "Fast and robust distributed subgraph enumeration," *Proc. Clustering Valid. Very Large Databases*, vol. 12, no. 11, pp. 1344–1356, 2019.

[14] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *Proc. Int. Conf. Data Eng.*, 2013, pp. 997–1008.

[15] K. Semertzidis and E. Pitoura, "Durable graph pattern queries on historical graphs," in *Proc. Int. Conf. Data Eng.*, 2016, pp. 541–552.

[16] K. Semertzidis and E. Pitoura, "Top-k durable graph pattern queries on temporal graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 1, pp. 181–194, Jan. 2019.

[17] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *Proc. Int. Conf. Data Eng.*, 2019, pp. 1082–1093.

[18] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proc. 10th ACM Int. Conf. Web Search Data Mining*, 2017, pp. 601–610.

[19] T. Zhang, Y. Gao, L. Qiu, L. Chen, Q. Linghu, and S. Pu, "Distributed time-respecting flow graph pattern matching on temporal graphs," *World Wide Web*, vol. 23, no. 1, pp. 609–630, 2020.

[20] Y. Gao, T. Zhang, L. Qiu, Q. Linghu, and G. Chen, "Time-respecting flow graph pattern matching on temporal graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 33, no. 10, pp. 3453–3467, Oct. 2021.

[21] C. Kosyfaki, N. Mamoulis, E. Pitoura, and P. Tsaparas, "Flow computation in temporal interaction networks," in *Proc. Int. Conf. Data Eng.*, 2021, pp. 660–671.

[22] E. Valdano, C. Poletto, A. Giovannini, D. Palma, L. Savini, and V. Colizza, "Predicting epidemic risk from past temporal contact data," *PLoS Comput. Biol.*, vol. 11, no. 3, e1004152, 2015. [Online]. Available: https://doi.org/10.1371/journal.pcbi.1004152

[23] R. Wang, W. Ji, and B. Song, "Durable relationship prediction and description using a large dynamic graph," *World Wide Web*, vol. 21, no. 6, pp. 1575–1600, 2018.

[24] A. Clauset and N. Eagle, "Persistence and periodicity in a dynamic proximity network," 2012, *arXiv:1211.7343*.

[25] T. D. Plantenga, "Inexact subgraph isomorphism in mapreduce," *J. Parallel Distrib. Comput.*, vol. 73, no. 2, pp. 164–175, 2013.

[26] B. Bhattarai, H. Liu, and H. H. Huang, "CECI: Compact embedding cluster index for scalable subgraph matching," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1447–1462.

Faming Li received the BE and ME degrees from the Harbin Institute of Technology, China, in 2012 and 2014, respectively, and the PhD degree from the School of Computer Science and Technology, Harbin Institute of Technology, in 2021. His research interests include graph databases and graph mining.

Zhaonian Zou received the BS and ME degrees from Jilin University, China, in 2002 and 2005, respectively, and the PhD degree from the Harbin Institute of Technology (HIT), China, in 2010. He is currently a professor with the School of Computer Science and Technology, Harbin Institute of Technology, China. In 2011, he joined HIT, as a faculty member. He has authored or coauthored more than 50 papers in refereed journals and conferences. His research interests include graph analytics and database systems.

Jianzhong Li is a professor with the School of Computer Science and Technology, Harbin Institute of Technology, China. From 1986 to 1987 and from 1992 to 1993, he was a staff scientist with Lawrence Berkeley National Laboratory. From 1991 to 1992 and from 1998 to 1999, he was also a visiting professor with the University of Minnesota. He has authored or coauthored more than 300 papers in refereed journals and conferences. His research interests include databases, wireless sensor networks, data quality, and complexity theory. He was an associate editor for the *IEEE Transactions on Knowledge and Data Engineering*.