technologies

MDPI

Article

Temporal Multi-Query Subgraph Matching in Cybersecurity

Min Lu, Qianzhen Zhang * and Xianqiang Zhu *

National Key Laboratory of Information Systems Engineering, National University of Defense Technology, Changsha 410073, China; lumin19@nudt.edu.cn

* Correspondence: zhangqianzhen18@nudt.edu.cn (Q.Z.); zhuxianqiang@nudt.edu.cn (X.Z.); Tel.: +86-150-7315-9353 (Q.Z.); +86-189-7490-0040 (X.Z.)

Abstract

Regarding attack scenarios as query graphs and conducting subgraph matching on the data system is an important approach to identify and detect cyber threats. However, existing subgraph matching methods are not suitable for detecting time-evolving attacks since they either focus on single-query graphs or ignore the temporal constraints between multiple queries. In this paper, we model the time-evolving attack detection as a novel temporal multi-query subgraph matching problem and propose an efficient algorithm to address this problem. We first give a compact representation of the temporal query graph by merging all queries into one. Based on the temporal query graph, we propose a concise auxiliary data structure to maintain partial solutions. In addition, we employ a query matching tree to generate an efficient matching order and enumerate matchings based on the order. Extensive experiments over real-world datasets confirm the effectiveness and efficiency of our approach.

Keywords: cyber security; temporal graph; multiple queries; subgraph matching

1. Introduction

Cyber security is one of the most significant technical challenges in current times. Detecting cyber attacks and preventing the spread of threat is a high priority for commercial companies and government agencies around the world. In a study titled "Data Breach Investigations Report", US communications company Verizon analyzed 100,000 security incidents from the past decade and concluded that 90% of the incidents fell into ten attack patterns [1]. A number of these attacks can be described as subgraph patterns. This motivates us to model cyber attack detection as a subgraph matching problem.

Recently, some graph-matching-based cyber attack detection methods have been proposed [2–7]. These studies extract attack behavior patterns from Cyber Threat Intelligence (shorted as CTI) reports and conduct graph pattern matching to hunt suspicious threats. However, all of these methods only focus on the detection of single static attack patterns and ignore the time-evolving characteristic of attacks, which is very critical for attack detection. With the fast development of cyber attack technologies, in some attack scenarios, attackers choose to change attack patterns over time in order to evade detection. For example, Figure 1 shows an attack scenario where attack patterns change over time. In this case, attack behavior cannot be regarded as a single query graph but a series of query graphs with specified temporal constraints. Such a time-evolving cyber attack detection variant task can be modeled as a *temporal multi-query subgraph matching* (shorted as TMQSM) problem.

check for updates

Academic Editor: Arash Habibi Lashkari

Received: 5 June 2025
Revised: 14 July 2025
Accepted: 30 July 2025
Published: 1 August 2025

Citation: Lu, M.; Zhang, Q.; Zhu, X. Temporal Multi-Query Subgraph Matching in Cybersecurity. *Technologies* **2025**, *13*, 335. https://doi.org/10.3390/technologies13080335

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

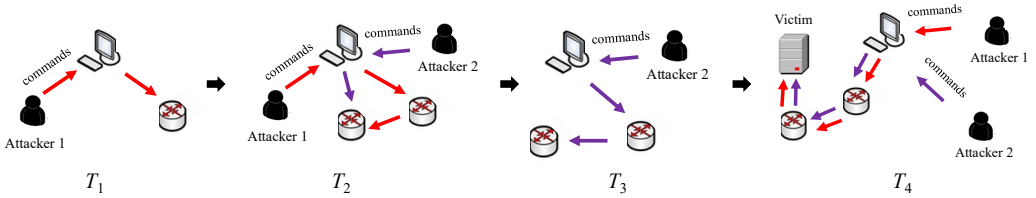


Figure 1. A time-evolving attack.

An intuitive idea to handle TMQSM is to separately search for matchings of each query pattern using existing methods and then check whether the matching result satisfies the specified temporal constraints. However, this method faces obvious drawbacks. First, it is costly to search for matchings of each query separately as it is NP-hard to conduct subgraph isomorphism. Moreover, a lot of candidate matchings cannot satisfy the specified temporal constraints, so searching for these false candidates and checking temporal constraints on these false candidate matchings waste both time and memory. Therefore, a novel and efficient subgraph matching method is critical and in demand.

In this paper, we propose a novel and efficient temporal multi-query subgraph matching approach. First, as stated above, it is not sensible to search for matchings of each query separately. Thus, we want to search for all matchings of multiple queries simultaneously. We introduce a compact representation of multiple queries. We merge all query graphs into a single graph and record temporal constraints on the edges. The merged graph is regarded as a new query graph for subsequent subgraph matching. Then, we design a novel index structure called a *temporal candidate graph* (shorted as TCG) to store the candidate results for enumerating matchings. In general, TCG is a directed temporal graph where vertices are the same as the data graph and a directed edge $u \to v$ indicates v appearing in at least one partial match of u. If two vertices u and v coexist in a match of a query graph, there must be a bidirectional edge between u and v (i.e., $u \leftrightarrow v$). In addition, each bidirectional edge is associated with a time set that denotes the timestamps of the edge existing and satisfying specified temporal constraints. This can be used as an efficient pruning rule for subsequent enumerating process. In addition, we develop an efficient subgraph matching enumerating algorithm based on the TCG. We first generate an effective matching order by constructing a query matching tree. Then, we enumerate all matchings by expanding partial matchings following the matching order.

In summary, the major contributions include the following: (1) We propose a novel temporal multi-query subgraph matching problem by considering the temporal constraints between multiple queries. (2) We design an efficient index structure TCG to maintain candidate results and support subsequent enumeration processes. (3) We build a query matching tree to generate an efficient matching order, which helps to efficiently enumerate subgraph matchings. (4) We conducted comprehensive experiments to evaluate our techniques.

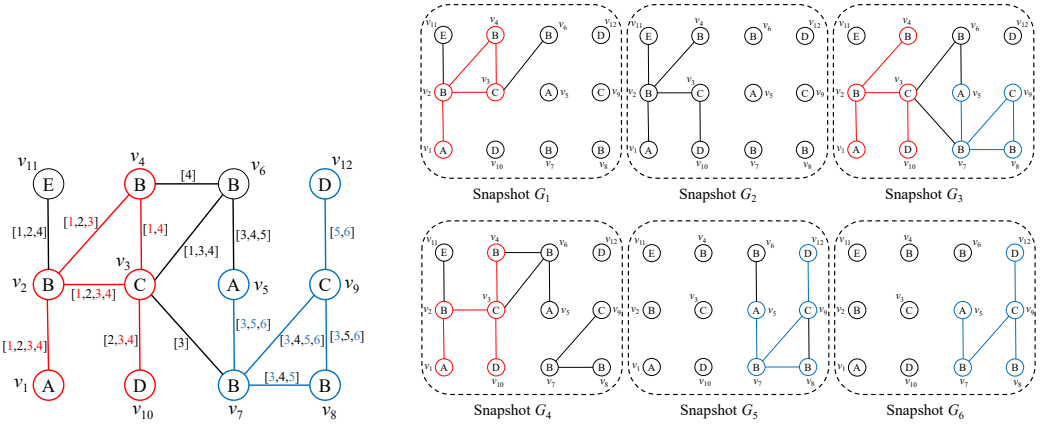
2. Preliminaries

In this section, we define the problem and review related work.

2.1. Problem Definition

In this paper, we focus on undirected and vertex-labeled graphs. Our techniques can be easily extended to edge-labeled and directed graphs. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ be a temporal graph with $|\mathcal{V}|$ vertices and $|\mathcal{E}|$ temporal edges, and $\mathcal{L}$:$\mathcal{V} \to \Sigma$ is a function that assigns a label to each vertex, where Σ is a set of labels. Each temporal edge $e \in \mathcal{E}$ is a triplet (u, v, t), where u and v are vertices in $\mathcal{V}$, and t is the timestamp. We use $t(u, v)$ to represent the existence of the timestamps of (u, v). Let $N(u) = \{v \in V | (u, v) \in \mathcal{E}\}$ be the set of neighbor vertices of u, and $d(u) = |N(u)|$ be the degree of u. A graph $\mathcal{G}_S = (\mathcal{V}_S, \mathcal{E}_S, \mathcal{L}_S)$

is a temporal subgraph of $\mathcal{G}$ if $\mathcal{V}_S \subseteq \mathcal{V}, \mathcal{E}_S \in \mathcal{E}$, and $\mathcal{L}_S = \mathcal{L}$. A temporal graph $\mathcal{G}$ can be extracted into a series of snapshots $\mathcal{G} = \{G_1, G_2 \dots G_{|\mathcal{T}|}\}$, where $\mathcal{T} = \{t|(u, v, t) \in \mathcal{E}\}$ is the set of timestamps. Figure 2a,b show a temporal graph and its snapshots within the time span $[1, 6]$.



(**a**) Temporal graph (**b**) Snapshots

Figure 2. A temporal graph and snapshots.

Definition 1 (Subgraph isomorphism). *Given a query graph* $q = (V_q, E_q, L_q)$ *and a data graph* $g = (V_g, E_g, L_g)$, *q is subgraph isomorphic to g if and only if there exists an injective mapping* $M{:}V_q \rightarrow V_g$, *such that (1)* $\forall u \in V_q, L_q(u) = L_g(M(u))$, *and (2)* $\forall (u, v) \in E_q, (M(u), M(v)) \in E_g$.

We call the subgraph induced by M a matching of q in g.

There is only one query graph in the traditional subgraph matching problem. In this paper, we consider multiple queries with the temporal constraints between queries. Let $\mathcal{Q} = \{Q_1, \dots, Q_n\} (n \geq 2)$ be a set of query graphs. And $F : \mathcal{Q} \rightarrow \mathbb{N}$ is a function defined to specify the temporal constraints of queries. Note that F defines orders and time intervals between queries. We call $(\mathcal{Q} = \{Q_1, \dots, Q_n\}, F)$ a *temporal query set*. Figure 3 shows a temporal query set containing three query graphs. The function F defines the temporal order and time intervals between queries. That is, query Q_1 appears before Q_2 with time interval $F(Q_2) - F(Q_1) = 3 - 1 = 2$, and Q_2 appears before Q_3 with time interval $F(Q_3) - F(Q_2) = 4 - 3 = 1$.

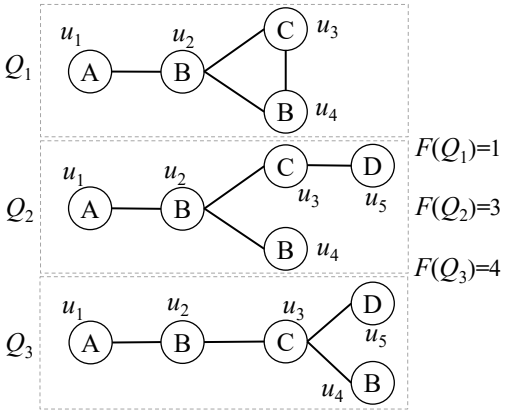


Figure 3. A temporal query set.

Definition 2 (Temporal multi-query subgraph isomorphism). *Given a temporal query set* $(\mathcal{Q} = \{Q_1, \dots, Q_n\}, F)$ *and a temporal graph* $\mathcal{G} = \{G_1, G_2 \dots G_{|\mathcal{T}|}\}$, *we check whether a graph*

set $\mathcal{M} = \{M_{i_1}, M_{i_2} \dots M_{i_n}\}$ *satisfies the following: (1)* $\forall j \in \{1, \dots, n\}$, M_{i_j} *is a subgraph of* G_{i_j} *and* M_{i_j} *is subgraph isomorphic to* Q_j*; (2)* $\forall j \in \{2, \dots, n\}$, $i_j - i_{j-1} = F(Q_j) - F(Q_{j-1})$*; and (3)* $\forall j, k \in \{1, \dots, n\}, v_j \in V_{Q_j}, v_k \in V_{Q_k}$*, and if* $v_j = v_k$*, then* $v_j^{'} = v_k^{'}$*, where* $v_j^{'} \in V_{M_{i_j}}$ *and* $v_k^{'} \in V_{M_{i_k}}$ *are matchings of* v_j *and* v_k*. If so, we call* $\mathcal{M}$ *a temporal multi-query matching of* $(\mathcal{Q} = \{Q_1, \dots, Q_n\}, F)$.

Problem Statement. Given a temporal query set $(\mathcal{Q} = \{Q_1, \dots, Q_n\}, F)$ and a data graph $\mathcal{G}$, the problem of *temporal multi-query subgraph matching* is to find all temporal multi-query matchings of $(\mathcal{Q} = \{Q_1, \dots, Q_n\}, F)$ in $\mathcal{G}$.

Example 1. *Consider the temporal query set in Figure 3 and temporal graph snapshots in Figure 2b. The subgraphs marked in red and the subgraphs marked in blue are two temporal multi-query matchings that not only match the query graphs in the temporal query set but also satisfy the temporal constraints. Thus, we obtain two matching results, i.e.,* $\{< u_1, v_1 >, < u_2, v_2 >, < u_3, v_3 >, < u_4, v_4 >, < u_5, v_{10} >\}$ *and* $\{< u_1, v_5 >, < u_2, v_7 >, < u_3, v_9 >, < u_4, v_8 >, < u_5, v_{12} >\}$.

2.2. Related Work

We summarize the related work as follows.

Graph-Matching-based Attack Detection. Prior works have implemented graph matching methods to detect cyber attacks. ThreatRaptor [8] extracts threat behaviors from open source Cyber Threat Intelligence as the query graph and conducts exact query matching to hunt threats. POIROT [5] formulates threat hunting as a graph pattern matching problem and conducts approximation matching. DeepHunter [7] detects APTs using graph similarity matching between extracted subgraphs and attack query graphs. MEGR-APT [2] discovers APTs by conducting subgraph matching between a provenance graph and query graph. ACMA [4] applies subgraph matching techniques to detect cyber attacks in smart cities. However, those systems only focus on a single-attack query scenario, which cannot meet the needs of complex time-evolving attack detection.

Subgraph Matching on Static Graph. A lot of subgraph matching algorithms have been proposed on static graphs in recent decades. Ullmann [9] first proposed a backtracking algorithm to solve this problem in 1976, which iteratively maps the vertices from a query graph to a data graph by following the given order of query vertices. The VF2 algorithm [10] generates the matching order by selecting a vertex connected to an already matched one, which can prune many false-positive candidates early. QuickSI [11] determines the matching order using the infrequency information of vertices and edges to filter invalid candidates efficiently. GraphQL [12] and SPath [13] exploit a neighborhood-based filtering strategy to prune infeasible candidates of query vertices. TurboISO [14] defines the concept of neighborhood equivalence class and merges similar vertices to avoid useless computations. BoostIso [15] further improves the performance of TurboISO by compressing the data graph. CFL-Match [16] converts the query graph into a spanning tree to find candidates for query vertices and applies path-ordering techniques to postpone Cartesian products. DAF [17] uses a directed acyclic graph instead of a spanning tree to obtain candidates and utilizes the knowledge gained from past computations to prune redundant computations. VEQ [18,19] leverages vertex equivalence to reduce the candidate matching space. RI+ [20] improves the matching time using candidate region-based decomposition and ordering techniques. SUFF [21] generates a structure filtering framework to prune the search space using historical matching results. However, the algorithms mentioned above were adapted to static graphs and cannot be directly used to solve the subgraph matching problem on temporal graphs.

Multi-query Subgraph Matching. Recently, some solutions have been proposed to process multiple queries. GraphflowDB [22] calculates subgraph matching for multiple query graphs using a worst-case optimal join algorithm. The performance heavily relies on the decomposition of the multi-query graph and join order. TRIC+ [23] decomposes queries into a series of minimum covering paths and tries to index the common parts of those paths. It maintains materialized views to store intermediate matching results and conducts join operations to obtain matching results. However, this method is limited by the expensive join cost and maintenance cost of materialized views. IncMQO [24] merges all queries into a large query graph and uses a tree index to store intermediate results. But it still enumerates each query graph separately to obtain matching results. MQ-Match [25] proposes a data-centric index structure to store partial results and uses the computation-sharing method to calculate matchings. However, none of the above methods take temporal constraints between multiple queries into consideration, and the definition of our problem is different from those of the above problems, so none of these existing methods can be applied to solve our problem.

3. Overview of Our Algorithm

To solve TMQSM, we first give a compact representation of the temporal query set and then give an overview of our algorithm for finding matchings.

As stated in Section 1, it is not advisable to search for matchings of each query separately. Therefore, we want to compute matchings of all queries simultaneously using a compact representation of a temporal query set. Inspired by [26], we merge all queries in the temporal query set into a single graph and record the temporal constraints on the edges. We call the merged graph the temporal query pattern, and regard the temporal query pattern as a new query q. Specifically, $(\mathcal{Q} = \{Q_1, \ldots, Q_n\}, F)$ can be represented as a graph $q = (V_q, E_q, L_q)$, where $V_q = V_{\mathcal{Q}}$, $E_q = \bigcup_{i=1,\ldots,n} E_{Q_i}$. Each edge $e \in E_q$ is associated with a time set T_e. T_e is obtained as follows. Initially, T_e is empty. Next, for each $Q_i \in \mathcal{Q}$, $T_e \leftarrow T_e + F(Q_i)$ if $e \in E_{Q_i}$. Figure 4 shows the representation of the temporal query pattern generated by the temporal query set in Figure 3. Note that the time set on the edges of the temporal query pattern only represents the temporal constraints and does not specify the concrete time instances. For example, the set $[1, 3, 4]$ on edge (u_1, u_2) indicates that the edge (u_1, u_2) exists in three queries where the time intervals between these queries is 2 and 1, respectively. In the rest of this paper, we use q to represent the temporal query pattern.

Example 2. *Reconsider the temporal graph in Figure 2a and temporal query pattern in Figure 4, where the temporal multi-query matchings are marked in red and blue. And the corresponding timestamps on the data edges that satisfy the temporal constraints defined on the query edges are also colored red and blue.*

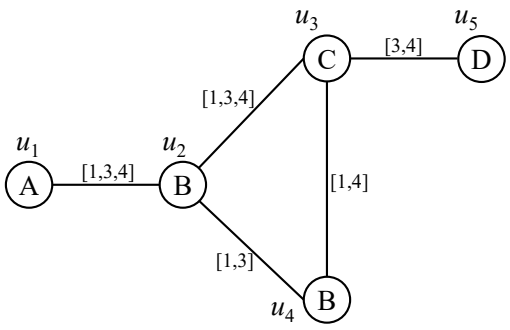


Figure 4. The temporal query pattern merged by a temporal query set.

Given query q and data graph $\mathcal{G}$, Algorithm 1 summarizes our TMQ algorithm for finding matchings, which consists of two phases:

- **Phase 1 (Construction of index structure)**. In this phase, we build a novel index structure called the *temporal candidate graph* based on the query graph and data graph. The TCG stores all necessary information required for enumerating matchings. The details about Phase 1 will be presented in Section 4.
- **Phase 2 (Enumeration of matchings)**. In this phase, we first construct a query matching tree for generating a matching order. Then, we enumerate all matchings on the TCG by following the matching order. Details about Phase 2 will be presented in Section 5.

Algorithm 1: TMQ($(\mathcal{Q} = \{Q_1, \ldots, Q_n\}, F), \mathcal{G}$)

Input : A temporal query set $(\mathcal{Q} = \{Q_1, \ldots, Q_n\}, F)$ and a data graph $\mathcal{G}$.
Output: All matchings of $(\mathcal{Q} = \{Q_1, \ldots, Q_n\}, F)$ in $\mathcal{G}$.

1 $q \leftarrow$ the new query graph merged by $(\mathcal{Q} = \{Q_1, \ldots, Q_n\}, F)$;
2 $TCG \leftarrow BuildTCG(q, \mathcal{G})$;
3 $QT \leftarrow BuildQueryMatchingTree(q, TCG)$;
4 $SearchMatch(QT, TCG)$;

4. Construction of Index Structure

Previous studies [11,12,14–16] have proved that index structure is crucial to improve the performance of searching for candidates. A good index structure should be (1) compact and memory-friendly, (2) powerful in filtering unpromising candidates, and (3) complete enough to support matching enumeration. In this section, we propose the *temporal candidate graph*, a concise and efficient index structure with which to maintain valid candidates for enumeration.

In general, TCG is a directed temporal graph where vertices are the same as in the data graph and a directed edge $u \to v$ indicates v appearing in at least one partial match of u. If two vertices u and v coexist in a match of the query graph, there must be a bidirectional edge between u and v. In addition, each bidirectional edge is associated with a time set, which denotes the existing timestamps of an edge and satisfies specified temporal constraints. Thus, a data vertex is a valid candidate if it has at least one bidirectional edge in the TCG. In this section, we first determine the state of candidate vertices and then construct the TCG.

4.1. Vertex State Determination

We designed a novel auxiliary data structure, called the *temporal neighborhood matching graph* (TNMG), to determine the state of vertices in the data graph.

Definition 3 (Temporal Neighborhood Matching Graph (shorted as TNMG)). *Given a query q and a data graph $\mathcal{G}$, for a vertex $v \in \mathcal{G}$, the TNMG of v, denoted as $TNM(v) = (\mathbb{V}_1, \mathbb{L}, \mathbb{V}_2, \mathbb{T}, \mathbb{E})$, is constructed as follows:*

1. *$\mathbb{V}_1$: A set of query vertices with the same label as v, i.e., $\mathbb{V}_1 = \{u | u \in V_q \wedge L(u) = L(v)\}$.*
2. *$\mathbb{L}$: A label set of neighbors of vertices in $\mathbb{V}_1$, i.e., $\mathbb{L} = \{L(w) | w \in N(u) \wedge u \in \mathbb{V}_1\}$.*
3. *$\mathbb{V}_2$: A set of neighbors of v in $\mathcal{G}$ that have a label in $\mathbb{L}$, i.e., $\mathbb{V}_2 = \{w | w \in N(v) \wedge L(w) \in \mathbb{L}\}$.*
4. *$\mathbb{T}$: A time set recording the timestamps of the edge connecting v and each vertex in $\mathbb{V}_2$, i.e., $\mathbb{T} = \{t(w, v) | w \in \mathbb{V}_2\}$.*
5. *$\mathbb{E}$: First, for each vertex $w \in \mathbb{V}_2$, add an edge $(w, t(w, v))$ between w and its time set in $\mathbb{T}$ if $t(w, v)$ passes through the TE-Filter check; then, add an edge (L, w) between w and its label $L \in \mathbb{L}$ if w is connected with a time set; then, for each vertex $u \in \mathbb{V}_1$ and its neighbors'*

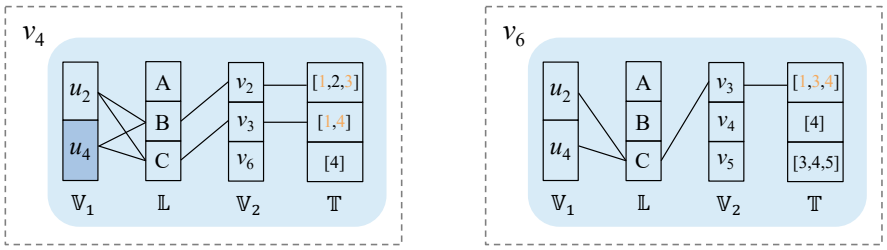
label $L \in \mathbb{L}$, add an edge (u, L) if L has at least τ neighbors in $\mathbb{V}_2$, where τ is the number of neighbors of u with label L in the query graph.

Here, TE-Filter is a pruning technology used to check the temporal constraints on an edge, which is defined as follows.

Definition 4 (Temporal Edge Filter (TE-Filter))**.** *Given an edge (u_1, u_2) in the query q, let $\{t_1^q, \ldots, t_k^q\}$ be the time set on the edge. Let $\{t_1, \ldots, t_m\}$ be the time set on the data edge (v_1, v_2). We have $L(v_1) = L(u_1)$ and $L(u_2) = L(v_2)$. We say (v_1, v_2) passes through the TE-Filter when $\{t_1, \ldots, t_m\}$ satisfies the following conditions: 1) $k \leq m$ and 2) there exists at least one subset $\{t_{i_1}, \ldots, t_{i_k}\} \subseteq \{t_1, \ldots, t_m\}$ such that $t_{i_j} - t_{i_{j-1}} = t_j^q - t_{j-1}^q$ for any $j \in \{2, \ldots, k\}$.*

Note that, it is not clear that a data edge (v_1, v_2) will be matched to a query edge when searching for candidates. So, we use a relaxed TE-Filter to check temporal constraints when constructing the TNMG. Specifically, given an edge (u_1, u_2) in query q, $(L(u_1), L(u_2))$ is the *label edge* of (u_1, u_2). Let $\{t_1^q, \ldots, t_k^q\}$ be the time set on the label edge $(L(u_1), L(u_2))$. A label edge might correspond to multiple query edges. Thus, there will be multiple time sets on a label edge. In this case, we say a data edge passes through the TE-Filter if it satisfies the above conditions with at least one time set on the label edge.

Example 3. *Consider the temporal data graph in Figure 2a and query graph in Figure 4, where the temporal neighborhood matching graphs of vertices v_4 and v_6 are constructed as in Figure 5. For v_4, $\mathbb{V}_1 = \{u_2, u_4\}$ because u_2 and u_4 have the same label B with v_4. Then, because the neighbors of u_2 and u_4 are $\{u_1, u_3, u_4\}$ and $\{u_2, u_3\}$, we have $\mathbb{L} = \{A, B, C\}$ by collecting the labels of these neighbors. Next, we obtain the neighbors of v_4 with label A, B, or C; thus $\mathbb{V}_2 = \{v_2, v_3, v_6\}$. Then, we record the timestamps of (v_2, v_4), (v_3, v_4), and (v_4, v_6) in $\mathbb{T}$, respectively. Lastly, we add edges into $TNM(v_4)$. For (v_2, v_4), the label edge of (v_2, v_4) is (B, B). For the query graph in Figure 4, the time set on the label edge (B, B) is $[1, 3]$. Because the timestamps of edge (v_2, v_4) is $[1, 2, 3]$, which successfully passes through TE-Filter, an edge between v_2 and $[1, 2, 3]$ is added. For edge (v_3, v_4) with label edge (B, C), there exist two time sets $[1, 3, 4]$ and $[1, 4]$ on the query label edge (B, C). The timestamp of (v_3, v_4) is $[1, 4]$, which can pass through TE-Filter with one time set, so we also add an edge between v_3 and $[1, 4]$. Then, we add edges between $\mathbb{L}$ and $\mathbb{V}_2$, that is, $\{(v_2, B), (v_3, C)\}$, and add edges between $\mathbb{V}_1$ and $\mathbb{L}$, including $\{(u_2, B), (u_2, C), (u_4, B), (u_4, C)\}$. The temporal neighborhood matching graphs of v_6 and other vertices are constructed in the same manner.*



(a) TNM(v_4) **(b)** TNM(v_6)

Figure 5. Temporal neighborhood matching graph of v_4 and v_6.

Then, we determine the vertex state based on the TNMG.

Definition 5 (Vertex state)**.** *For a vertex v in $\mathcal{G}$, the state of v, denoted as $s(v)$, is a binary value represented by "True" or "False".*

Let $TNM(v)$ be v's temporal neighborhood matching graph. For a vertex $u \in \mathbb{V}_1(TNM(v))$, we say u is completely matched if, for each of its neighbors w in the query graph, there is an edge $(u, L(w))$ in $TNM(v)$. We set $s(v)$ to "True" if there exists at least one completely matched vertex in $\mathbb{V}_1(TNM(v))$, and "False" otherwise. For a vertex v, the "True" state indicates that there exists at least one query vertex u that can be matched to v, and all neighbors of u in the query graph are completely matched by the neighbors of v in the data graph. For example, u_4 is completely matched in Figure 5a, so the state of v_4, $s(v_4)$, is "True". Both u_2 and u_4 are not completely matched in Figure 5b, so the state of v_6, $s(v_6)$, is "False".

4.2. Index Construction

After setting the state of vertices in $\mathcal{G}$, we construct the index structure TCG.

Definition 6 (Temporal Candidate Graph (TCG)). *Given a query q and a data graph $\mathcal{G}$, the temporal candidate graph $TCG = (V_{TCG}, E_{TCG}, T_{TCG})$ is constructed as follows:*

1. $V_{TCG} = \{v | v \in V_{\mathcal{G}} \wedge L(v) \in L_q\}$;
2. *For each $v \in V_{TCG}$, if $s(v) =$ "True", then add a directed edge $v \rightarrow w$ into E_{TCG} for each vertex $w \in \mathbb{V}_2(TNM(v))$ if w has at least one edge in $TNM(v)$, where $TNM(v)$ is the temporal neighborhood matching graph of v;*
3. *For $v, w \in V_{TCG}$, if there exists a bidirectional edge between v and w, then add $T_e(v, w)$ into T_{TCG}, where $T_e(v, w)$ is a set of time subsets of $t(v, w)$ that satisfy the temporal constraints.*

For a vertex v in the TCG, we say it is a *valid candidate* if it has at least one bidirectional edge.

Example 4. *Figure 6 shows the TCG of the data graph in Figure 2a and query graph in Figure 4. Excluding v_6 and v_{11}, the state of the other vertices are "True". Taking v_4 as an example, the state of v_4 is "True", and $\mathbb{V}_2(TNM(v_4)) = \{v_2, v_3, v_6\}$. Because v_2 and v_3 have connecting edges in $TNM(v_4)$, we add an outgoing edge from v_4 to v_2 and v_3, respectively. In addition, for each bidirectional edge in the TCG, we add a set to record the time subsets that satisfy the temporal constraints.*

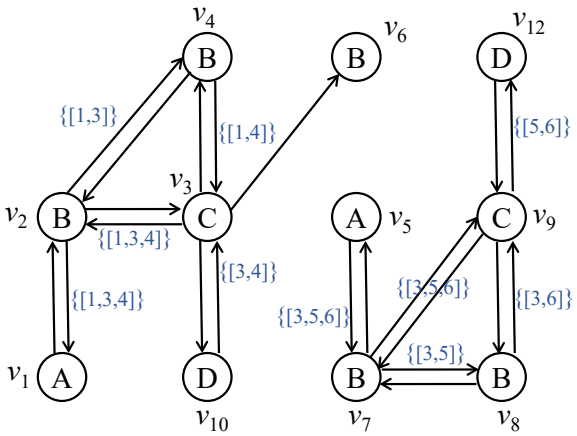


Figure 6. Temporal candidate graph.

Algorithm 2 summarizes the details of constructing the index structure TCG. We first initialize V_{TCG}, E_{TCG}, and T_{TCG} (Line 1). For each vertex v with a label in the query graphs, we first construct its TNMG and set its state (Lines 2–4). If the state is "True", we add an outgoing edge from v to each vertex in $\mathbb{V}_2(TNM(v))$, which is connected with at least one edge (Lines 5–7). If there exists a bidirectional edge between vertex v and w, we add a time

set to T_{TCG} (Lines 8–9). The details for TNMG construction and vertex state determination are illustrated in Lines 11–28 and Lines 29–32, respectively.

Algorithm 2: BuildTCG(q, $\mathcal{G}$)

Input : A query graph q and a data graph $\mathcal{G}$.
Output: The index structure TCG of $\mathcal{G}$.

1 $V_{TCG} \leftarrow V_{\mathcal{G}}$, $E_{TCG} \leftarrow \emptyset$, $T_{TCG} \leftarrow \emptyset$;
2 **foreach** $v \in V_{TCG}$ **do**
3 **if** $L(v) \in L(q)$ **then**
4 $TNM(v) \leftarrow$ *BuildTNMG*(v), $s(v) \leftarrow$ *SetState*(v);
5 **if** $s(v) =$ "True" **then**
6 **foreach** $w \in \mathbb{V}_2(TNM(v)) \wedge d_{TNM(v)}(w) \neq 0$ **do**
7 $E_{TCG} \leftarrow E_{TCG} \cup \{v \rightarrow w\}$;
8 **foreach** $(v, w) \in E_{TCG}$ **do**
9 **if** $v \leftrightarrow w$ **then** $T_{TCG} \leftarrow T_{TCG} \cup \{T_e(v, w)\}$;
10 **return** TCG;

Function *BuildTNMG*(v)

11 $\mathbb{V}_1 \leftarrow \emptyset, \mathbb{L} \leftarrow \emptyset, \mathbb{V}_2 \leftarrow \emptyset, \mathbb{T} \leftarrow \emptyset, \mathbb{E} \leftarrow \emptyset$;
12 $TNM(v) \leftarrow (\mathbb{V}_1, \mathbb{L}, \mathbb{V}_2, \mathbb{T}, \mathbb{E})$;
13 **foreach** $u \in V_q$ **do**
14 **if** $L(u) = L(v)$ **then** $\mathbb{V}_1 \leftarrow \mathbb{V}_1 \cup \{u\}$;
15 **foreach** $u \in \mathbb{V}_1$ **do**
16 **foreach** $w \in N(u)$ **do**
17 $\mathbb{L} \leftarrow \mathbb{L} \cup \{L(w)\}$;
18 **foreach** $w \in N(v)$ **do**
19 **if** $L(w) \in \mathbb{L}$ **then** $\mathbb{V}_2 \leftarrow \mathbb{V}_2 \cup \{w\}$;
20 **foreach** $w \in \mathbb{V}_2$ **do**
21 $\mathbb{T} \leftarrow \mathbb{T} \cup \{t(w, v)\}$;
22 **if** TE-Filter$((w, v), (L(w), L(v))) =$ true **then**
23 $\mathbb{E} \leftarrow \mathbb{E} \cup \{(w, t(w, v)\}, \mathbb{E} \leftarrow \mathbb{E} \cup \{(L(w), w)\}$;
24 **foreach** $u \in \mathbb{V}_1$ **do**
25 **foreach** $l \in L(N(u))$ **do**
26 $\tau \leftarrow$ the number of neighbors of u in q with label l;
27 **if** $|N_{TNM(v)}(l) \cap \mathbb{V}_2| \geq \tau$ **then** $\mathbb{E} \leftarrow \mathbb{E} \cup \{(u, l)\}$;
28 **return** $TNM(v)$;

Function *SetState*(v)

29 **foreach** $u \in \mathbb{V}_1(TNM(v))$ **do**
30 **if** $\exists w \in N(u)$ and $(u, L(w)) \notin \mathbb{E}(TNM(v))$ **then** continue;
31 **else return** "True";
32 **return** "False";

4.3. Algorithm Analysis

Next, we analyze the time complexity and space complexity of our proposed algorithms.

Proposition 1. *The time complexity of constructing the TNMG for all vertices is* $O(|\mathcal{E}_\mathcal{G}| * |T|log|T| + |E_q| * |\mathcal{V}_\mathcal{G}|)$*, where* $|T|$ *is the number of snapshots in data graph* $\mathcal{G}$*.*

Proof. For each vertex v, the time to compute vertices in $\mathbb{V}_1$, $\mathbb{L}$, $\mathbb{V}_2$, and $\mathbb{T}$ is $|V_q|$, $\frac{|V_q|}{|L_q|} * d_q$, $d(v)$, and $d(v)$, where d_q is the average degree of vertices in q. So, the time complexity of computing vertices is $O(|V_q| + \frac{|V_q|}{|L_q|} * d_q + d(v) + d(v)) = O(|V_q| + \frac{|V_q|}{|L_q|} * d_q + d(v))$. Meanwhile, the time to check TE-Filter for each time set in $\mathbb{T}$ is $|T|log|T|$, where $|T|$ is the number of snapshots in data graph $\mathcal{G}$. So the time complexity of adding edges is $O(d(v) * |T|log|T| + d(v) + \frac{|V_q|}{|L_q|} * |L_q| * d_q) = O(d(v) * |T|log|T| + |E_q|)$. Thus, the time complexity of constructing TNMG for v is $O(|V_q| + \frac{|V_q|}{|L_q|} * d_q + d(v) + d(v) * |T|log|T| + |E_q|) = O(|V_q| + \frac{|V_q|}{|L_q|} * d_q + d(v) * |T|log|T| + |E_q|)$, which is bounded by $O(d(v) * |T|log|T| + |E_q|)$. Therefore, the overall time complexity of constructing TNMG for all vertices is $O(\sum_{v \in \mathcal{V}_\mathcal{G}}(d(v) * |T|log|T| + |E_q|)) = O(|\mathcal{E}_\mathcal{G}| * |T|log|T| + |E_q| * |\mathcal{V}_\mathcal{G}|)$. □

Proposition 2. *The space complexity of TNMG for all vertices is* $O(|V_q| * |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}|)$*.*

Proof. For each v, the space cost to store vertices in $\mathbb{V}_1$, $\mathbb{L}$, $\mathbb{V}_2$, and $\mathbb{T}$ is $\frac{|V_q|}{|L_q|}$, $|L_q|$, $d(v)$, and $d(v)$, respectively. So the space complexity of storing vertices is $O(\frac{|V_q|}{|L_q|} + |L_q| + d(v) + d(v)) = O(\frac{|V_q|}{|L_q|} + |L_q| + d(v))$. Then, we need to store edges in TNMG. The number of edges between $\mathbb{V}_2$ and $\mathbb{T}$ and the number of edges between $\mathbb{L}$ and $\mathbb{V}_2$ are both bounded by $d(v)$. And the number of edges between $\mathbb{V}_1$ and $\mathbb{L}$ is bounded by $\frac{|V_q|}{|L_q|} * |L_q| = |V_q|$. So, the space complexity of storing edges is $O(d(v) + d(v) + |V_q|) = O(d(v) + |V_q|)$. Thus, the space cost of constructing TNMG for v is $O(\frac{|V_q|}{|L_q|} + |L_q| + d(v) + |V_q|)$, which is bounded by $O(d(v) + |V_q|)$. Therefore, the overall space cost for all vertices is $O(\sum_{v \in \mathcal{V}_\mathcal{G}}(d(v) + |V_q|) = O(|\mathcal{E}_\mathcal{G}| + |V_q| * |\mathcal{V}_\mathcal{G}|)$. □

Proposition 3. *The time complexity of constructing TCG is* $O(|\mathcal{E}_\mathcal{G}| * |T|log|T| + |E_q| * |\mathcal{V}_\mathcal{G}|)$*, where* $|T|$ *is the number of snapshots in data graph* $\mathcal{G}$*.*

Proof. For each $v \in V_{TCG}$, the time complexity of constructing TNMG is $O(d(v) * |T|log|T| + |E_q|)$ according to Proposition 1. The time complexity to set vertex state is $O(\frac{|V_q|}{|L_q|} * d_q) = O(\frac{|E_q|}{|L_q|})$. If $s(v) =$ "True", the time complexity of adding edges is bounded by $O(d(v))$. For all vertices, the time complexity is $O(\sum_{v \in \mathcal{V}_\mathcal{G}}(d(v) * |T|log|T| + |E_q| + \frac{|E_q|}{|L_q|} + d(v))) = O(|\mathcal{E}_\mathcal{G}| * |T|log|T| + |E_q| * |\mathcal{V}_\mathcal{G}|)$. For all bidirectional edges, the time complexity of adding time sets is bounded by $O(|\mathcal{E}_\mathcal{G}|)$. Therefore, the overall time complexity of *BuildTCG* is $O(|\mathcal{E}_\mathcal{G}| * |T|log|T| + |E_q| * |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}|) = O(|\mathcal{E}_\mathcal{G}| * |T|log|T| + |E_q| * |\mathcal{V}_\mathcal{G}|)$, which is dominated by that of constructing TNMG. □

Proposition 4. *The space cost of TCG is* $O(|V_q| * |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}|)$*.*

Proof. During the process of *BuildTCG*, we need to store TCG and TNMG for all vertices. The total space cost of constructing TNMG is $O(|V_q| * |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}|)$ according to Proposition 2. The space cost to store vertices in V_{TCG} is bounded by $O(|\mathcal{V}_\mathcal{G}|)$. The TCG is induced from $\mathcal{G}$, so the space cost to store edges is $O(|\mathcal{E}_\mathcal{G}|)$. For the bidirectional edges, the space cost to store time set is bounded by $O(|\mathcal{E}_\mathcal{G}|)$. Therefore, the overall space complexity is $O(|V_q| * |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}| + |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}|) = O(|V_q| * |\mathcal{V}_\mathcal{G}| + |\mathcal{E}_\mathcal{G}|)$, which is also dominated by that of constructing TNMG. □

5. Enumeration of Matchings

In this section, we present our algorithm to find all matchings of the query graph based on the TCG. We first construct a query matching tree for generating an efficient matching order and then compute all matchings following the matching order.

5.1. Matching Order Generation

Given a query graph, we can find matchings by performing a widely used depth-first search paradigm [27]. The search space is actually a depth-first search tree. Different matching orders can lead to a large search space and affect the performance of the algorithm. In order to save search space, we construct a query matching tree based on the query graph and TCG, which can be used to generate an efficient matching order.

The query matching tree is a DFS spanning tree QT of the query graph. To generate QT, we need to select a root and then determine the DFS order. We define the *cost* of a query vertex u as $cost(u) = |C(u)|/d(u)$, where $|C(u)|$ is the number of valid candidates with the same label as u in the TCG, and $d(u)$ is the degree of u in the query graph. Intuitively, a vertex with few candidates and a larger degree means it has a smaller search space and higher chance to prune unpromising partial results. So, we select the vertex with the least cost as the root of QT. Then, we determine the DFS order. Let u be the vertex currently being visited; if there exist unvisited neighbors of u, we select neighbor w with the least *cost* as the next vertex to visit and designate u as the parent of w in the QT. Otherwise, we backtrack to the parent of u. In addition, for each node u in QT, we maintain a set of vertices in the ancestor nodes to record the non-tree edges in the query graph, which is denoted as $u.a$. After constructing the query matching tree, we generate the matching order by performing preorder traversal on QT.

Example 5. *For the query graph in Figure 4 and temporal candidate graph in Figure 6, the cost of the vertices u_1, u_2, u_3, u_4, and u_5 is 2, 4/3, 2/3, 2, and 2, respectively. Thus, u_3 is selected as the root. Then, we construct the query matching tree QT by performing DFS backtracking on the query. The QT is shown in Figure 7. The matching order is determined by preorder traversal on QT, that is, u_3, u_2, u_1, u_4, and u_5.*

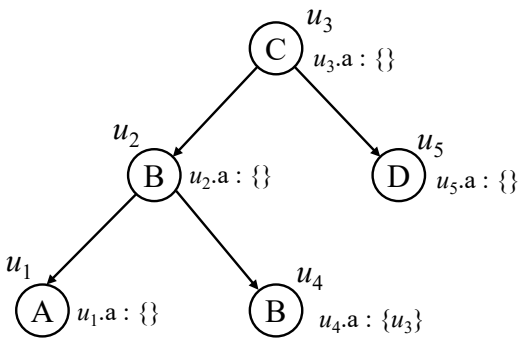


Figure 7. Query matching tree.

5.2. Matching Enumeration

The TCG contains all necessary information to enumerate matchings; thus, we can directly find all matchings in the TCG by following the matching order.

Algorithm 3 illustrates the details of our matching approach. We first use a hashmap M and MT to store the partial matchings and time instance sets (Line 1). Then, we generate a matching order (Line 2). The root r of QT must be the first matched vertex in the order. Thus, for each valid candidate v with the same label as r in the TCG, we match v to r and call Procedure Expand to generate a matching (Lines 4–6).

Algorithm 3: SearchMatch(QT, TCG)

Input : The query matching tree QT and the index structure TCG.
Output: All matchings M.

1 $M \leftarrow \emptyset$, $MT \leftarrow \emptyset$;
2 $\varphi \leftarrow$ the preorder traversal order of nodes in QT;
3 $r \leftarrow \varphi[1]$;
4 **foreach** valid candidate $v \in V_{TCG} : L(v) = L(r)$ **do**
5 $\quad M[r] = v$;
6 $\quad$ *Expand*$(2, M, \varphi, MT)$;

Function *Expand*(i, M, φ, MT)
7 **if** $i = |V_q|$ **then report** M and MT;
8 **else**
9 $\quad n \leftarrow \varphi[i]$;
10 $\quad n_p \leftarrow$ the parent node of n in QT;
11 $\quad v_p \leftarrow M[n_p]$;
12 $\quad$ **foreach** $v \in N_{TCG}(v_p) : L(v) = L(n)$ and $v \notin M$ and $v_p \leftrightarrow v$ **do**
13 $\quad\quad$ **if** TE-Filter$((n_p, n), (v_p, v)) =$true **then**
14 $\quad\quad\quad ts \leftarrow$ the time subsets of $t(v_p, v)$ that satisfy the TE-Filter;
15 $\quad\quad\quad$ **if** $n.a = \emptyset$ or NontreeCheck$(n, v, n.a, M) =$ true **then**
16 $\quad\quad\quad\quad$ **if** $MT \neq \emptyset$ **then**
17 $\quad\quad\quad\quad\quad$ **foreach** $((u_1, u_2), ((v_1, v_2), t)) \in MT$ **do**
18 $\quad\quad\quad\quad\quad\quad$ **if** $\forall$TG-Filter$(((u_1, u_2), (v_1, v_2), t), ((n_p, n), (v_p, v), ts)) =$ true **then**
19 $\quad\quad\quad\quad\quad\quad\quad M[n] = v$;
20 $\quad\quad\quad\quad\quad\quad\quad MT[(n_p, n)] \leftarrow ((v_p, v), ts)$;
21 $\quad\quad\quad\quad\quad\quad\quad$ *Expand*$(i + 1, M, \varphi, MT)$;
22 $\quad\quad\quad\quad$ **else**
23 $\quad\quad\quad\quad\quad M[n] = v$;
24 $\quad\quad\quad\quad\quad MT[(n_p, n)] \leftarrow ((v_p, v), ts)$;
25 $\quad\quad\quad\quad\quad$ *Expand*$(i + 1, M, \varphi, MT)$;

Function *NontreeCheck*(n, v, S, M)
26 $\quad$ **foreach** $s \in S$ **do**
27 $\quad\quad p \leftarrow M[s]$;
28 $\quad\quad$ **if** $v \nleftrightarrow p$ or TE-Filter$((s, n), (p, v)) =$ false **then**
29 $\quad\quad\quad$ **return** false;
30 $\quad$ **return** true;

During the Expand process, once the size of partial matchings equals the number of query vertices, we report M and MT as a result (Line 7). Otherwise, we process the next query vertex n according to the matching order. We determine the matches of n according to the matched result of its parent n_p (Lines 9–12). In this process, we use TE-Filter and NontreeCheck to check the temporal constraints and structure information separately (Lines 13–15). Note that, the TE-Filter only checks the temporal constraints on a single edge. We further use TG-Filter to check the temporal constraints between two edges (Lines 16–21). The temporal graph filter (TG-Filter) is designed as follows. Let $T^*(u_1, u_2)$ and $T^*(u_1', u_2')$ be the time sets on query edges (u_1, u_2) and (u_1', u_2'). Suppose a data edge

(v_1, v_2) has been matched to (u_1, u_2). Let $\{t_1, \ldots, t_k\}$ be the time subset on (v_1, v_2) that passes through the TE-Filter, and $t^{'}(v_1^{'}, v_2^{'})$ be the time subset on the data edge $(v_1^{'}, v_2^{'})$ that passes through the TE-Filter. If $|T^*(u_1, u_2) \cap T^*(u_1^{'}, u_2^{'})| > |\{t_1, \ldots, t_k\} \cap t^{'}(v_1^{'}, v_2^{'})|$, then $(v_1^{'}, v_2^{'})$ cannot be matched to $(u_1^{'}, u_2^{'})$. If a candidate passes through all checkings, it is verified as a match of n. We then update the M and MT, and execute the next Expand process. The details about NontreeCheck are illustrated in Lines 26–30.

Example 6. *Figure 8 shows the search tree for the query graph in Figure 4 and TCG in Figure 6 according to the matching order* $u_3 \rightarrow u_2 \rightarrow u_1 \rightarrow u_4 \rightarrow u_5$. *During the searching process, we use TE-Filter and TG-Filter to prune invalid branches. Finally, we can obtain the matching results for query q, i.e.,* $\{< u_3, v_3 >, < u_2, v_2 >, < u_1, v_1 >, < u_4, v_4 >, < u_5, v_{10} >\}$ *and* $\{< u_3, v_9 >, < u_2, v_7 >, < u_1, v_5 >, < u_4, v_8 >, < u_5, v_{12} >\}$.

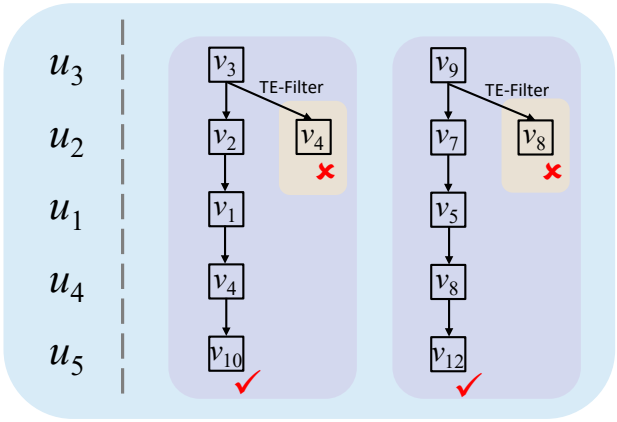


Figure 8. Search tree for enumeration.

6. Experiments

In this section, we evaluate the performance of our proposed TMQ algorithm. All experiments were implemented in C++ and ran on a PC with an Intel(R) Core Ultra 7 1.4 GHz CPU and 16 GB of RAM.

6.1. Experimental Setup

Datasets. The experiments were carried out on five real-world temporal graphs: *CollegeMsg*, *MathOverflow*, *SuperUser*, *WikiTalk*, and *StackOverflow* (all datasets were obtained from https://snap.stanford.edu/data, accessed on 29 July 2025). Since these temporal graphs are unlabeled, we assigned five labels to the vertices at random. We partitioned the time span of a temporal graph into a series of consecutive non-overlapping time intervals of equal length and merged all edges that occur within a time interval into a snapshot. The edges within a snapshot share the same timestamp. In the experiments, we set the time interval of CollegeMsg as an hour (60×60 s) and the time interval of other datasets as a day ($24 \times 60 \times 60$ s). The detailed statistics of the datasets are shown in Table 1.

Table 1. Statistics of datasets.

Dataset	Vertices	Temporal Edges	Static Edges	Time Span
CollegeMsg (college)	1899	59,835	20,296	193 days
MathOverflow (math)	21,688	107,581	90,489	2350 days
SuperUser (super)	194,085	1,443,339	924,886	2773 days
WikiTalk (wiki)	1,140,149	7,833,140	3,309,592	2320 days
StackOverflow (stack)	2,464,606	17,823,525	16,266,395	2774 days

Queries. In the experiments, given the desired number of edges $|E_q|$ in the temporal query pattern q and the desired number of queries n in temporal query set $\mathcal{Q}$, we generated the temporal query patterns as connected subgraphs of the data using random walks on the temporal graph. To generate different temporal query patterns, we varied the number of edges $|E_q|$ from five to eight with a default value of six and varied the number of queries n from two to four a with default value of three. For a data graph, we generated 50 temporal query patterns under each specific $|E_q|$ and n.

Comparative algorithm. In the experiments, we compared our TMQ algorithm with a baseline developed based on VEQ [19], a state-of-the-art subgraph matching algorithm for a single query on static graphs. Given a temporal graph $\mathcal{G}$ and a temporal query set $(\mathcal{G} = \{Q_1, \ldots, Q_n\}, F)$, this algorithm first finds all matchings of each query in $\{Q_1, \ldots, Q_n\}$ on each snapshot of $\mathcal{G}$ using VEQ. Then, for each matching result of $\{Q_1, \ldots, Q_n\}$, it checks whether the timestamp order satisfies the temporal constraint defined by function F. If the temporal order satisfies the constraint, the matching is output as a temporal multi-query subgraph matching of $(\mathcal{G} = \{Q_1, \ldots, Q_n\}, F)$ in $\mathcal{G}$. This algorithm is called VEQ_T in our experiments.

Metrics. We measured and evaluated (1) the average elapsed time to process a query by varying the number of query edges and the number of queries; (2) the average memory usage to process a query by varying the number of query edges and the number of queries; and (3) the scalability of algorithm. In every experiment, we set the time limit to process a query to 1 h (i.e., 3.6×10^6 ms). If an algorithm could not finish within the time limit, then we plottef the elapsed time as INF. Each experimental test was repeated five times, and the average metric is reported.

6.2. Processing Time

In this part, we compare our method TMQ with VEQ_T in terms of the average running time to process a query on five real-world datasets.

Figure 9 shows the experimental results under the default parameter settings over different datasets. We can see that VEQ_T cannot finish within the time limit on the *wiki* and *stack* datasets, while our TMQ can return results within 100 s on all datasets. TMQ significantly outperforms the competitor on all datasets under the default settings. The processing time of TMQ is up to three orders of magnitude faster than that of VEQ_T, even excluding the cases when VEQ_T cannot finish within the time limit. The reason for this is that VEQ_T needs to process the multiple queries sequentially and check the temporal constraints between queries one by one, which costs a lot of time overhead. TMQ utilizes an efficient index structure TCG to maintain the partial matchings and uses an effective matching order to enumerate matchings, both of which can efficiently prune a lot of unpromising intermediate results and significantly speed up the querying process.

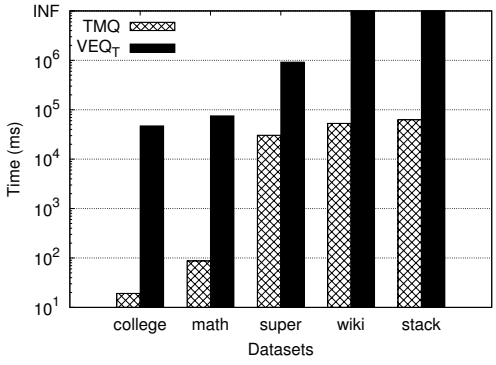


Figure 9. Average processing time on different datasets.

Figure 10 shows the average processing time of TMQ and VEQ_T on the *college* dataset when varying the number of edges $|E_q|$ in the temporal query pattern from five to eight and the number of queries n in the temporal query set from two to four. From Figure 10a, we can see that TMQ significantly outperforms VEQ_T under all parameter settings and it can improve the processing time by one to three orders of magnitude on the *college* dataset. The reason is illustrated above. When setting the number of queries n as a default value and varying the number of edges $|E_q|$, the average processing time of TMQ and VEQ_T is plotted in Figure 10b. When n is fixed, the average execution time of TMQ to process a query increases as the number of edges $|E_q|$ in a temporal query pattern increases, while the average processing time of VEQ_T changes very smoothly with increasing $|E_q|$. This is because, when $|E_q|$ becomes larger, TMQ requires more time to construct the index structure and enumerate matchings for more edges. For VEQ_T, the process of matching result verification takes up the majority of the execution time; thus the subtle change in edges in a single query will not obviously affect the overall time. For fixed $|E_q|$, the average processing time of TMQ and VEQ_T when varying n is demonstrated in Figure 10c. The results show that the running time of VEQ_T increases with increasing n, while the running time of TMQ decreases as n increases. This is because VEQ_T needs to find all subgraph matchings for each query in the temporal query set one by one; thus, the time cost will increase when the number of queries in the temporal query set increases, while TMQ equipped with filtering techniques can significantly reduce the search space. When n becomes larger, the temporal constraints on the edges of temporal query pattern become stricter, so more unpromising candidates can be filtered, and the time to verify candidates and enumerate results decreases.

Figures 11–14 depict the experimental results on the temporal graphs *math*, *super*, *wiki*, and *stack*, respectively. These experimental results are very similar to those obtained on *college*. Specifically, TMQ is up to three orders of magnitude faster than VEQ_T on *math* and two orders of magnitude faster than VEQ_T on *super*. For large temporal graphs *wiki* and *stack*, VEQ_T cannot terminate within the time limit under most parameter settings, whereas TMQ can find all matchings and is also several orders of magnitude faster than VEQ_T.

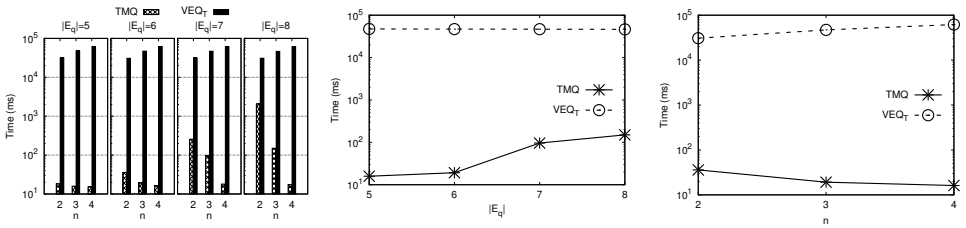


Figure 10. Average processing time on *college*. (**a**) Processing time under different settings. (**b**) Processing time with varying $|E_q|$. (**c**) Processing time with varying n.

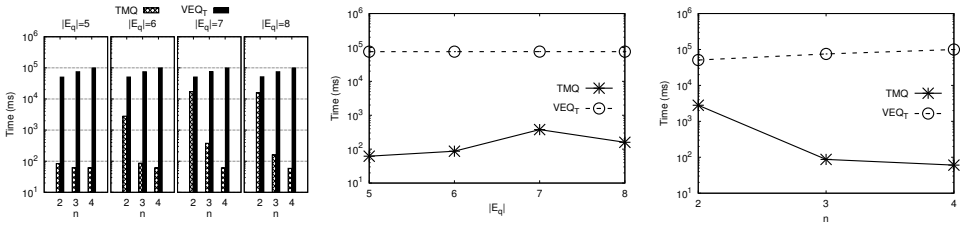


Figure 11. Average processing time on *math*. (**a**) Processing time under different settings. (**b**) Processing time with varying $|E_q|$. (**c**) Processing time with varying n.

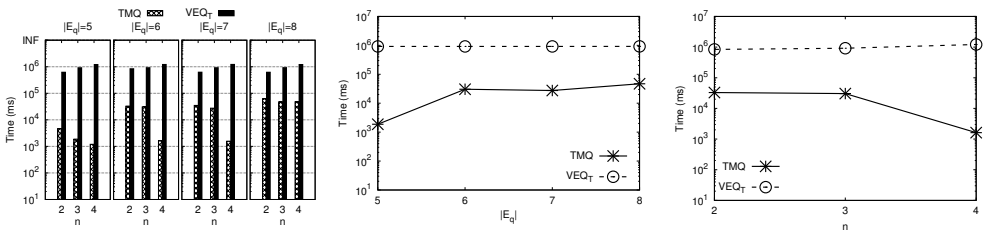


Figure 12. Average processing time on *super*. (**a**) Processing time under different settings. (**b**) Processing time with varying $|E_q|$. (**c**) Processing time with varying n.

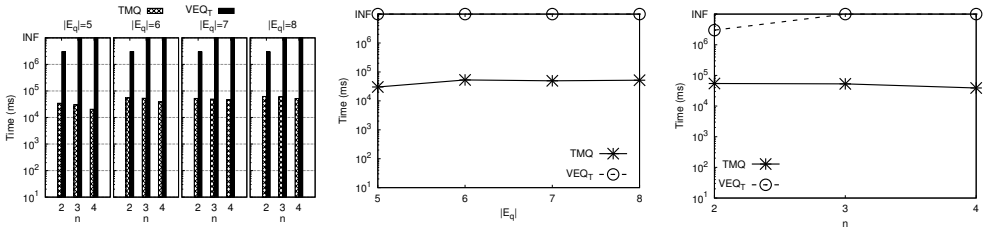


Figure 13. Average processing time on *wiki*. (**a**) Processing time under different settings. (**b**) Processing time with varying $|E_q|$. (**c**) Processing time with varying n.

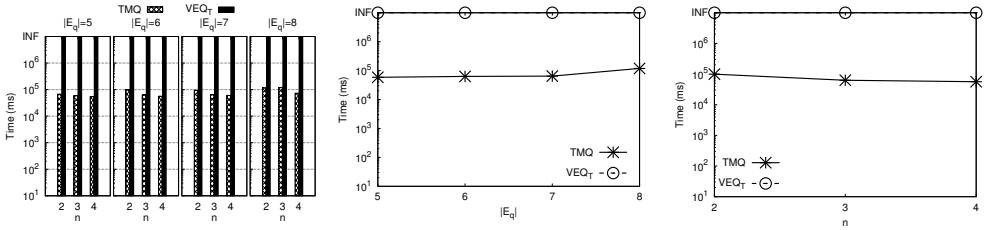


Figure 14. Average processing time on *stack*. (**a**) Processing time under different settings. (**b**) Processing time with varying $|E_q|$. (**c**) Processing time with varying n.

6.3. Memory Usage

In this part, we evaluate the performance of TMQ against $\mathsf{VEQ_T}$ based on the aspect of average memory usage to process a query on five different datasets.

Figure 15a–e show the average memory usage of two algorithms on temporal graphs *college*, *math*, *super*, *wiki*, and *stack*, respectively. The size of the corresponding temporal graph is marked by the line. In the experiments, we did not record or plot the memory usage if the algorithm could not finish within the time limit. The experimental results show that TMQ and $\mathsf{VEQ_T}$ were close in the memory cost to process a query. This is because TMQ needs to verify the state of all vertices when constructing the index structure, which uses a lot of memory. As for $\mathsf{VEQ_T}$, it takes a lot of memory to store all matchings of each query even if most matchings do not satisfy the temporal constraints.

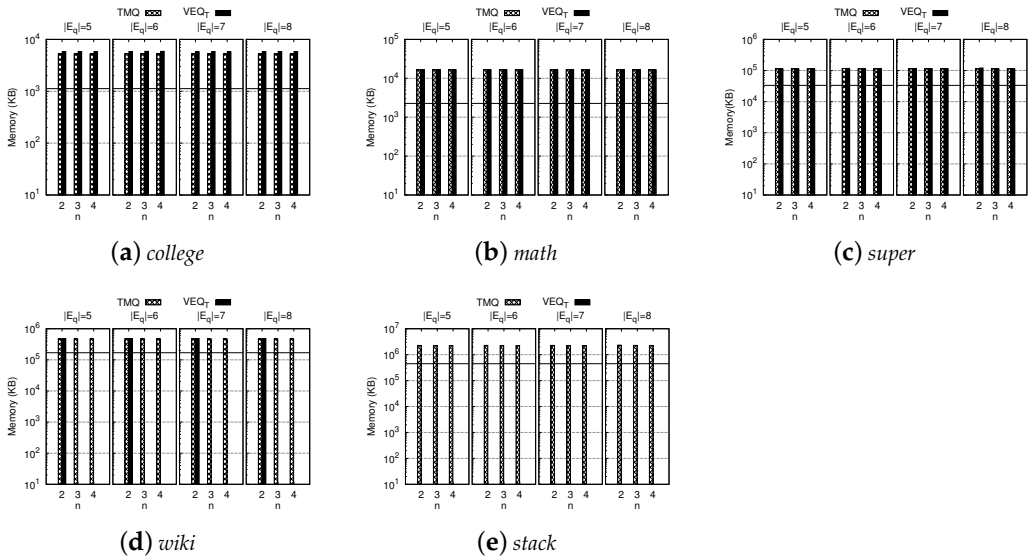


Figure 15. Average memory usage on different datasets.

6.4. Scalability

In this part, we evaluate the scalability of TMQ and $\mathsf{VEQ_T}$ by varying the size of the data graph under default parameter settings.

Figure 16a shows the average running time of the two algorithms when varying the size of the dataset under default parameter settings. It can been seen that $\mathsf{VEQ_T}$ cannot finish on large datasets, whereas TMQ can find all results on all datasets. And the running time of TMQ increases with increasing dataset size. Figure 16b shows the average memory usage when varying the size of the dataset under default parameter settings. We can see that the memory usage increases linearly as dataset size increases. The above results demonstrate the good scalability of TMQ.

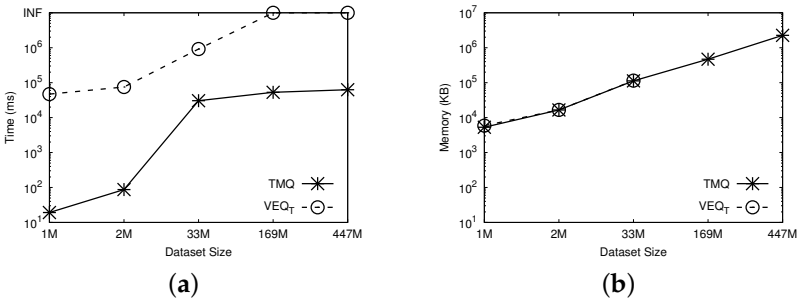


Figure 16. Scalability of algorithms. (**a**) Average running time with varying data graph size. (**b**) Average memory usage with varying data graph size.

7. Conclusions

In this paper, we model the time-evolving cyber attack detection task as a novel temporal multi-query subgraph matching problem and propose an efficient algorithm to tackle this problem. Specifically, we first merge multiple queries into a new query graph with temporal constraints, which is helps improve the searching efficiency. Then, we design a novel index structure called TCG to maintain valid candidates. During this process, we use an efficient pruning technology TE-Filter to filter unpromising intermediate results by considering the temporal constraints. Lastly, we generate an efficient matching order by constructing a query matching tree and enumerate subgraph matching results based on the matching order. Extensive experiments on five real datasets demonstrate that our method

is several orders of magnitude faster than the existing method, while achieving comparable performance in memory consumption.

Author Contributions: Conceptualization, M.L.; methodology, M.L.; validation, M.L.; writing—original draft preparation, M.L.; supervision, Q.Z. and X.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets can be found at https://snap.stanford.edu/data (accessed on 29 July 2025).

Conflicts of Interest: The authors declare that they have no known competing financial interest or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Verizon. Data Breach Investigations Report 2020. *Comput. Fraud. Secur.* **2020**, 4.
2. Aly, A.; Iqbal, S.; Youssef, A.; Mansour, E. MEGR-APT: A Memory-Efficient APT Hunting System Based on Attack Representation Learning. *IEEE Trans. Inf. Forensics Secur.* **2024**, *19*, 5257–5271. [CrossRef]
3. Choudhury, S.; Holder, L.; Chin, G.; Agarwal, K.; Feo, J. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. *arXiv* **2015**, arXiv:1503.00849. [CrossRef]
4. Jia, Y.; Gu, Z.; Du, L.; Long, Y.; Wang, Y.; Li, J.; Zhang, Y. Artificial intelligence enabled cyber security defense for smart cities: A novel attack detection framework based on the MDATA model. *Knowl.-Based Syst.* **2023**, *276*, 110781. [CrossRef]
5. Milajerdi, S.M.; Rigel, G.; Birhanu, E.; Venkatakrishnan, V.N. POIROT: Aligning Attack Behavior with Kernel Audit Records for Cyber Threat Hunting. In Proceedings of the CCS'19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019.
6. Satvat, K.; Gjomemo, R.; Venkatakrishnan, V. Extractor: Extracting Attack Behavior from Threat Reports. In Proceedings of the 2021 IEEE European Symposium on Security and Privacy (EuroS&P), Vienna, Austria, 6–10 September 2021.
7. Wei, R.; Cai, L.; Zhao, L.; Yu, A.; Meng, D. DeepHunter: A Graph Neural Network Based Approach for Robust Cyber Threat Hunting. In Proceedings of the Security and Privacy in Communication Networks, Virtual Event, 6–9 September 2021.
8. Peng, G.; Fei, S.; Xiaoyuan, L.; Xusheng, X.; Zheng, Q.; Fengyuan, X.; Prateek, M.; Kulkarni, S.R.; Dawn, S. Enabling Efficient Cyber Threat Hunting With Cyber Threat Intelligence. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021.
9. Ullmann, J.R. An Algorithm for Subgraph Isomorphism. *J. ACM* **1976**, *13*, 31–42. [CrossRef]
10. Cordella, L.P.; Foggia, P.; Sansone, C.; Vento, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* **2004**, *26*, 1367–1372. [CrossRef] [PubMed]
11. Shang, H.; Zhang, Y.; Lin, X.; Yu, J.X. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* **2008**, *1*, 364–375. [CrossRef]
12. He, H.; Singh, A.K. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In Proceedings of the SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of Data, Vancouver, BC, Canada, 10–12 June 2008.
13. Zhao, P.; Han, J. On graph query optimization in large networks. *Proc. VLDB Endow.* **2010**, *3*, 340–351. [CrossRef]
14. Han, W.S.; Lee, J.; Lee, J.H. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In Proceedings of the SIGMOD '13: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013.
15. Ren, X.; Wang, J. Exploiting Vertex Relationships in Speeding up Subgraph Isomorphism over Large Graphs. *Proc. VLDB Endow.* **2015**, *8*, 617–628. [CrossRef]
16. Bi, F.; Chang, L.; Lin, X.; Qin, L.; Zhang, W. Efficient Subgraph Matching by Postponing Cartesian Products. In Proceedings of the SIGMOD'16: Proceedings of the 2016 International Conference on Management of Data, San Francisco, CA, USA, 26 June–1 July 2016.
17. Han, M.; Kim, H.; Gu, G.; Park, K.; Han, W.S. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In Proceedings of the SIGMOD'19: Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019.

18. Kim, H.; Choi, Y.; Park, K.; Lin, X.; Hong, S.H.; Han, W.S. Versatile Equivalences: Speeding up Subgraph Query Processing and Subgraph Matching. In Proceedings of the SIGMOD/PODS'21: Proceedings of the 2021 International Conference on Management of Data, Xi'an, China, 20–25 June 2021.
19. Kim, H.; Choi, Y.; Park, K.; Lin, X.; Hong, S.H.; Han, W.S. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *VLDB J.* **2023**, *32*, 343–368. [CrossRef]
20. Ansari, Z.A.; Parwez, M.A.; Thoker, I.R.; Jahiruddin. Enhanced Subgraph Matching for Large Graphs using Candidate Region-Based Decomposition and Ordering. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *35*, 101694. [CrossRef]
21. Jian, X.; Li, Z.; Chen, L. SUFF: Accelerating Subgraph Matching with Historical Data. *Proc. VLDB Endow.* **2023**, *16*, 1699–1711. [CrossRef]
22. Amine, M.; Chathura, K.; Semih, S. Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins. *ACM Trans. Database Syst.* **2021**, *46*, 1–45.
23. Zervakis, L.; Setty, V.; Tryfonopoulos, C.; Hose, K. Efficient Continuous Multi-Query Processing over Graph Streams. *arXiv* **2019**, arXiv:1902.05134. [CrossRef]
24. Xi, W.; Qianzhen, Z.; Deke, G.; Xiang, Z. Continuous multi-query optimization for subgraph matching over dynamic graphs. *Semant. Web* **2022**, *13*, 601–622.
25. Ziyi, M.; Jianye, Y.; Xu, Z.; Guoqing, X.; Jianhua, W.; Liang, Y.; Kenli, L.; Xuemin, L. Efficient Multi-Query Oriented Continuous Subgraph Matching. In Proceedings of the 40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, 13–16 May 2024.
26. Li, F.; Zou, Z.; Li, J.; Yang, X.; Wang, B. Evolving subgraph matching on temporal graphs. *Knowl.-Based Syst.* **2022**, *258*, 109961. [CrossRef]
27. Sun, S.; Luo, Q. Subgraph Matching With Effective Matching Order and Indexing. *IEEE Trans. Knowl. Data Eng.* **2022**, *34*, 491–505. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.