Efficiently Generating Temporal Simple Path Graphs

Zhiyang Tang[†], Yanping Wu[‡], Xiangjun Zai[♮], Chen Chen[§], Xiaoyang Wang[♮], Ying Zhang[‡]

[†]*Zhejiang Gongshang University, China* [‡]*University of Technology Sydney, Australia*

[♮]*The University of New South Wales, Australia* [§]*University of Wollongong, Australia*

zhiyangtang.zjgsu@gmail.com yanping.wu@student.uts.edu.au
xiangjun.zai@student.unsw.edu.au chen_chen@uow.edu.au
xiaoyang.wang1@unsw.edu.au ying.zhang@uts.edu.au

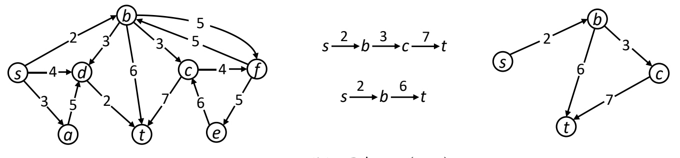
***Abstract*—Interactions between two entities often occur at specific timestamps, which can be modeled as a temporal graph. Exploring the relationships between vertices based on temporal paths is one of the fundamental tasks. In this paper, we conduct the first research to propose and investigate the problem of generating the temporal simple path graph (tspG), which is the subgraph consisting of all temporal simple paths from the source vertex to the target vertex within the given time interval. To solve this problem, we propose an efficient method named Verification in Upper-bound Graph. It first incorporates the temporal path constraint and simple path constraint to exclude unpromising edges from the original graph, which obtains a tight upper-bound graph as a high-quality approximation of tspG in polynomial time. Then, an escape edges verification algorithm is further applied in the upper-bound graph to construct the exact tspG without exhaustively enumerating all temporal simple paths between given vertices. Finally, comprehensive experiments on 8 real-world graphs are conducted to demonstrate the efficiency and effectiveness of the proposed techniques.**

I. Introduction

Path enumeration is a fundamental problem in graph analysis, which aims to enumerate all paths from one vertex to another. In reality, relationships between entities are often associated with the timestamp, which can be modeled as the temporal graph. Recently, many path models are defined in temporal graphs by integrating temporal constraints.

approach to addressing this task is to study the problem of paths connecting one vertex to the other.

Temporal graph is a specialized graph structure where edges are associated with timestamps, representing time-evolving interactions or relationships [1]–[6]. Due to its unique properties and wide applications, the temporal graph has gained significant attention in various domains, such as social networks, transportation systems and biological networks [7]–[9]. Among the key research topics, the temporal path problem, which focuses on finding paths under time constraints, plays a key role in uncovering the structural and temporal characteristics of dynamic systems [10]–[20]. Most existing studies on temporal paths aim to identify either a single path (e.g., reachability [12]–[15]) or all possible paths (e.g., path enumeration [18]–[20]) between two vertices. By enumerating all temporal paths, one can construct a *path graph*, which can further capture the global and structural relationships between vertices. For instance, in communication networks, a path graph can reveal the social group connecting two individuals by representing all intermediate relationships between them,



(a) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ (b) $\mathcal{P}^*_{[2,7]}(s,t)$ (c) $tspG_{[2,7]}(s,t)$

Fig. 1: Motivation example

providing a more comprehensive understanding of the interaction patterns.

In the literature, only a few works consider the path graph generation problem. For instance, Liu et al. [21] define the k-hop s-t subgraph query for all paths from s to t within k-hops. Cai et al. [22] extend this problem and aim to identify the k-hop constraint simple path graph. Wang et al. [23] define the shortest path graph, which exactly contains all the shortest paths between any pair of vertices. Unfortunately, these studies disregard the temporal information, making them applicable only to static networks and unable to capture dynamic features. Moreover, they impose hop constraint or shortest path constraint, limiting the information they provide.

To fill the gap, in this paper, we propose a novel problem, called temporal simple path graph ($tspG$) generation. In our problem, we focus on the strict and simple temporal path, that is, timestamps of edges along the path follow a strict ascending temporal order and no repeated vertices exist in the path [17], [24]. Specifically, given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, two vertices $s, t \in \mathcal{V}$, and a time interval $[\tau_b, \tau_e]$, the temporal simple path graph, denoted by $tspG_{[\tau_b,\tau_e]}(s,t)$, is a subgraph of $\mathcal{G}$ containing exactly all temporal simple paths from s to t within the time interval $[\tau_b, \tau_e]$.

***Example* 1** *Fig. 1(a) shows a directed temporal graph with eight vertices and fourteen directed edges. Given the source vertex s, the target vertex t and the time interval* $[2,7]$*, there exist two temporal simple paths from s to t within* $[2,7]$*, whose details are shown in Fig. 1(b). As we can see, these two paths share the same edge* $e(s,b,2)$*. Fig. 1(c) illustrates the corresponding temporal simple path graph.*

Applications. In the following, we illustrate two representative applications of temporal simple path graph generation.

- **Outbreak Control.** A disease transmission graph can be modeled as a temporal network, where vertices correspond to locations of individual contact, and temporal edges indicate the movement of individuals to specific locations at particular timestamps [25]. The transmission paths in such a graph reflect the spread of pathogens over time. Tracing critical transmission paths and developing isolation strategies for high-risk sites along the paths are widely used to control disease outbreaks. Generating the temporal simple path graph from the outbreak source to the protected area helps visualize the transmission routes, providing a foundational tool to support the development of containment strategies for disease control.

- **Financial Monitor.** Temporal graphs can be used to model financial transaction networks, where each vertex represents an account and each temporal edge represents a transaction from one account to another occurring at a specific timestamp. Typically, the activity of money laundering involves a sequence of monetary transactions adhering to an ascending order of timestamps, forming a simple cycle within a brief temporal window [26]. For a given transaction from account t to s at timestamp τ (i.e., a temporal edge $e(t, s, \tau)$), it will be a part of a monetary transaction simple cycle if there exists a temporal simple path from s to t in the given time interval [22]. Therefore, by employing our problem, we can visualize the flow of money from one account to another, providing a clear representation of the transaction process and aiding in the identification of risk-associated accounts and transactions.

Challenges. To the best of our knowledge, we are the first to propose and investigate the temporal simple path graph generation problem in temporal graphs. Naively, we can enumerate all temporal simple paths from the source vertex to the target vertex in the query time interval, then extract distinct vertices and edges from those paths to form $tspG$. However, this approach is far from efficient since it suffers from an exponential time complexity of $O(d^\theta \cdot \theta \cdot m)$, where d is the largest vertex degree in the original graph $\mathcal{G}$, θ is the length of the query time interval and m is the number of edges in $\mathcal{G}$. The key drawbacks of the naive enumeration strategy can be observed as follows: *i*) The original graph $\mathcal{G}$ includes numerous edges that are definitely not part of any temporal simple paths, which leads to unnecessary large search space for the enumeration. *ii*) Some edges may appear in multiple enumerated temporal simple paths from s to t and will be processed repeatedly during the $tspG$ formation process.

Our solutions. To address the aforementioned drawbacks and efficiently generate the temporal simple path graph, a novel method named *V̲erification in U̲pper−bound G̲raph* (shorted as VUG) is proposed with two main components, i.e., Upper-bound Graph Generation and Escaped Edges Verification.

Specifically, VUG reduces the search space for temporal simple paths by constructing an upper-bound graph as a high-quality approximation of the temporal simple path graph in polynomial time through two phases. In the first phase, it utilizes the QuickUBG method to efficiently filter out edges that are not contained in any temporal paths from s to t within $[\tau_b, \tau_e]$ and obtain a quick upper-bound graph $\mathcal{G}_q$. This can be achieved by comparing the timestamp of an edge with the earliest arrival time $\mathcal{A}(\cdot)$ and latest departure time $\mathcal{D}(\cdot)$ of its end vertices, where $\mathcal{A}(u)$ (resp. $\mathcal{D}(u)$) is when a temporal path from s within $[\tau_b, \tau_e]$ first arrives at u (resp. a temporal path to t within $[\tau_b, \tau_e]$ last departs u). $\mathcal{A}(u)$ and $\mathcal{D}(u)$ for all vertices can be computed using an extended BFS-like procedure in $O(n + m)$ time, where n and m are the number of vertices and edges in $\mathcal{G}$, respectively. In the second phase, VUG applies the TightUBG method on $\mathcal{G}_q$, which further excludes edge that can be efficiently determined as not contained in any temporal simple path from s to t within $[\tau_b, \tau_e]$. This method is powered by the introduction of *time-stream common vertices*, which summarizes the vertices that commonly appear across all temporal simple paths from s to u (resp. from u to t) within a time interval of interest. The computation of time-stream common vertices utilizes a recursive method in $O(n+\theta \cdot m)$ time without explicitly listing the paths, more clearly, time-stream common vertices of paths from s to vertex u can be calculated from time-stream common vertices of paths from s to the in-neighbors of u. After that, an edge $e(u, v, \tau)$ with two disjoint sets of time-stream common vertices, one set for temporal simple paths from s to u arriving before τ and the other set for temporal simple paths from v to t departuring after τ, is considered to have the potential of forming a temporal simple path from s to t and is retained in the tight upper-bound graph $\mathcal{G}_t$.

Finally, we generate the exact temporal simple path graph by verifying each edge in $\mathcal{G}_t$. Instead of employing a brute-force path enumeration procedure, we propose an Escaped Edges Verification (EEV) method that mitigates the repetition in edge verification. This method iteratively selects an unverified edge, identifies a temporal simple path from s to t containing it through bidirectional DFS, then adds a set of vertices and edges to $tspG$. Two carefully designed optimization methods regarding search direction and neighbor exploration order are integrated into the bidirectional DFS to further accelerate the edge verification process.

Contributions. We summarize the contributions in this paper.

- We conduct the first research to propose and investigate the problem of generating temporal simple path graph.
- To address this challenging problem, we propose an efficient method, namely VUG, consisting of two components, including Upper-bound Graph Generation for effectively yielding a high-quality approximate solution and Escaped Edges Verification for efficiently achieving the exact solution.
- We conduct extensive experiments on 8 real-word temporal graphs to compare VUG against baseline methods. The results demonstrate the effectiveness of our tight upper-bound graph and the efficiency of our algorithms.

Note that, due to space limitations, proofs of theorems and lemmas are presented in the Appendix.

II. Preliminaries

In this section, we present important concepts and notations related to temporal simple paths and formally define the

TABLE I: Summary of Notations

Notation	Definition
$\mathcal{G}, \mathcal{S}$	A directed temporal graph and an induced subgraph of $\mathcal{G}$
$N_{out}(u, \mathcal{S}), N_{in}(u, \mathcal{S})$	The out/in-neighbors of u in $\mathcal{S}$
$\mathcal{T}_{out}(u, \mathcal{S}), \mathcal{T}_{in}(u, \mathcal{S})$	The timestamp set of out/in-neighbors of u
θ	The span of time interval $[\tau_b, \tau_e]$
$p_{[\tau_b,\tau_e]}(s, t)$, $p^*_{[\tau_b,\tau_e]}(s, t)$	A temporal path and a temporal simple path from s to t in time interval $[\tau_b, \tau_e]$
$\mathcal{P}_{[\tau_b,\tau_e]}(s, t)$, $\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t)$	The set of all temporal paths $p_{[\tau_b,\tau_e]}(s, t)$ /all temporal simple paths $p^*_{[\tau_b,\tau_e]}(s, t)$
$tspG_{[\tau_b,\tau_e]}(s, t)$	The s-t temporal simple path graph in $[\tau_b, \tau_e]$
$\mathcal{G}_q$, $\mathcal{G}_t$	The quick/tight upper-bound graph of $tspG$
$\mathcal{A}(u)$, $\mathcal{D}(u)$	The earliest arrival/latest departure time of u
$TCV_\tau(s, u)$, $TCV_\tau(u, t)$	The time-stream common vertices of $\mathcal{P}^*_{[\tau_b,\tau]}(s, u)$, $\mathcal{P}^*_{[\tau,\tau_e]}(u, t)$ in $\mathcal{G}_q$

temporal simple path graph generation problem. Table I summarizes the notations frequently used throughout this paper.

A directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of vertices $\mathcal{V} = \mathcal{V}(\mathcal{G})$ and a set of directed temporal edges $\mathcal{E} = \mathcal{E}(\mathcal{G})$. $e(u, v, \tau) \in \mathcal{E}$ denotes a directed temporal edge from vertex u to vertex v, where u, $v \in \mathcal{V}$ and τ is the interaction timestamp from u to v. Without loss of generality, we use the same setting as previous studies for timestamp, which is the integer, since the UNIX timestamps are integers in practice [10], [24], [27]. We use $\mathcal{T} = \{\tau | e(u, v, \tau) \in \mathcal{E}\}$ to represent the set of timestamps. We use $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ to denote the number of vertices and edges, respectively. Given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a subgraph $\mathcal{S} = (\mathcal{V}_\mathcal{S}, \mathcal{E}_\mathcal{S})$ is an induced subgraph of $\mathcal{G}$, if $\mathcal{E}_\mathcal{S} \subseteq \mathcal{E}$, $\mathcal{V}_\mathcal{S} = \{u | e(u, v, \tau) \in \mathcal{E}_\mathcal{S} \vee e(v, u, \tau) \in \mathcal{E}_\mathcal{S}\}$. Given a subgraph $\mathcal{S}$, let $N_{out}(u, \mathcal{S}) = \{(v, \tau) | e(u, v, \tau) \in \mathcal{E}_\mathcal{S}\}$ (resp. $N_{in}(u, \mathcal{S}) = \{(v, \tau) | e(v, u, \tau) \in \mathcal{E}_\mathcal{S}\}$) be the set of out-neighbors (resp. in-neighbors) of u in $\mathcal{S}$. We use $\mathcal{T}_{out}(u, \mathcal{S})$ (resp. $\mathcal{T}_{in}(u, \mathcal{S})$) to represent all distinct timestamps in $N_{out}(u, \mathcal{S})$ (resp. $N_{in}(u, \mathcal{S})$).

Given a time interval $[\tau_b, \tau_e]$, where τ_b, $\tau_e \in \mathcal{T}$ and $\tau_b \leq \tau_e$, we use θ to denote the span of $[\tau_b, \tau_e]$, i.e., $\theta = \tau_e - \tau_b + 1$. The projected graph of $\mathcal{G}$ in $[\tau_b, \tau_e]$ is a subgraph of $\mathcal{G}$, denoted by $\mathcal{G}_{[\tau_b,\tau_e]} = (\mathcal{V}', \mathcal{E}')$, where $\mathcal{V}' = \mathcal{V}$ and $\mathcal{E}' = \{e(u, v, \tau) | e(u, v, \tau) \in \mathcal{E} \wedge \tau \in [\tau_b, \tau_e]\}$.

Given two vertices s, $t \in \mathcal{V}$ and a time interval $[\tau_b, \tau_e]$, a temporal path $p_{[\tau_b,\tau_e]}(s, t)$ from s to t in $[\tau_b, \tau_e]$, simplified as p, is a sequence of edges $\langle e(s = v_0, v_1, \tau_1), \ldots, e(v_{l-1}, v_l = t, \tau_l)\rangle$ such that $\tau_b \leq \tau_i < \tau_{i+1} \leq \tau_e$ for all integers $1 \leq i \leq l - 1$. The length of p is the number of edges on the path, denoted by $l = |p|$. $\mathcal{V}(p)$ and $\mathcal{E}(p)$ are the set of vertices and edges included in p, respectively.

***Remark* 1** *Let l be the length of a temporal path $p_{[\tau_b,\tau_e]}(s, t)$, and θ be the span of $[\tau_b, \tau_e]$, it is easy to obtain that $l \leq \theta$.*

***Definition* 1 (Temporal Simple Path)** *Given two vertices s, $t \in \mathcal{V}$ in a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a time interval $[\tau_b, \tau_e]$, a temporal simple path $p^*_{[\tau_b,\tau_e]}(s, t) = \langle e(s = v_0, v_1, \tau_1), \ldots, e(v_{l-1}, v_l = t, \tau_l)\rangle$ is a temporal path from s to t in $[\tau_b, \tau_e]$, without repeated vertices. That is, for any integers $0 \leq i < j \leq l$, $v_i \neq v_j$.*

We use $\mathcal{P}_{[\tau_b,\tau_e]}(s, t)$ to denote the set of all s-t temporal paths in $[\tau_b, \tau_e]$. The vertex set and edge set of $\mathcal{P}_{[\tau_b,\tau_e]}(s, t)$ are denoted by $\mathcal{V}(\mathcal{P}_{[\tau_b,\tau_e]}(s, t)) = \cup_{p \in \mathcal{P}_{[\tau_b,\tau_e]}(s,t)} \mathcal{V}(p)$ and $\mathcal{E}(\mathcal{P}_{[\tau_b,\tau_e]}(s, t)) = \cup_{p \in \mathcal{P}_{[\tau_b,\tau_e]}(s,t)} \mathcal{E}(p)$, respectively. Similarly, we denote the temporal simple path set as $\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t)$, along with its corresponding vertex set $\mathcal{V}(\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t))$ and edge set $\mathcal{E}(\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t))$.

***Definition* 2 (Temporal Simple Path Graph)** *Given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, two vertices s, $t \in \mathcal{V}$, and a time interval $[\tau_b, \tau_e]$, the temporal simple path graph, denoted by $tspG_{[\tau_b,\tau_e]}(s, t) = (\mathcal{V}^*, \mathcal{E}^*)$, is a subgraph of $\mathcal{G}$ such that $\mathcal{V}^* = \mathcal{V}(\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t))$ and $\mathcal{E}^* = \mathcal{E}(\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t))$.*

***Example* 2** *Consider the directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in Fig. 1(a). There are two temporal simple paths from s to t within time interval $[2, 7]$, i.e., $\langle e(s, b, 2), e(b, c, 3), e(c, t, 7)\rangle$ and $\langle e(s, b, 2), e(b, t, 6)\rangle$. These paths and the corresponding $tspG_{[2,7]}(s, t)$ are shown in Fig. 1(b) and (c), respectively.*

Problem Statement. Given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, two vertices s, $t \in \mathcal{V}$ and a time interval $[\tau_b, \tau_e]$, we aim to efficiently generate the temporal simple path graph $tspG_{[\tau_b,\tau_e]}(s, t)$.

***Remark* 2** *When the context is clear, we can omit the subscript notation $[\tau_b, \tau_e]$ and vertex pair (s, t), that is, $p_{[\tau_b,\tau_e]}(s, t)$, $p^*_{[\tau_b,\tau_e]}(s, t)$, $\mathcal{P}_{[\tau_b,\tau_e]}(s, t)$, $\mathcal{P}^*_{[\tau_b,\tau_e]}(s, t)$ and $tspG_{[\tau_b,\tau_e]}(s, t)$ can be simplified to p, p^*, $\mathcal{P}$, $\mathcal{P}^*$ and $tspG$.*

III. Solution Overview

In this section, we present three reasonable baseline solutions to generate temporal simple path graphs by extending the existing studies for temporal simple path enumeration. Then, we introduce the general framework of our solution.

A. The Baseline Solutions

According to the definition of the temporal simple path graph, a naive method for generating $tspG_{[\tau_b,\tau_e]}(s, t)$ is to first enumerate all temporal simple paths between s and t within $[\tau_b, \tau_e]$ in $\mathcal{G}$, then construct $tspG$ by taking the union of the vertex sets and edge sets of these paths. The running time of this naive strategy is bounded by $O(d^\theta \cdot \theta \cdot m)$, where d is the largest out-degree or in-degree of the vertices in $\mathcal{G}$, i.e., $d = \max_{u \in \mathcal{V}}\{\max(|N_{in}(u, \mathcal{G})|, |N_{out}(u, \mathcal{G})|)\}$, and θ is the span of $[\tau_b, \tau_e]$, i.e., $\theta = \tau_e - \tau_b + 1$. Although this approach can return the result, it may not be efficient due to the large search space, which is the original graph, including numerous vertices and edges that definitely cannot be involved in any temporal simple paths. This leads to unnecessary computational costs and slows down the enumeration process. To reduce the search space of enumeration, our baseline method first identifies a subgraph of $\mathcal{G}$ by filtering out many unpromising edges, which we refer to as the **upper-bound graph** of $\mathcal{G}$. Then, we enumerate temporal simple paths in the upper-bound graph rather than the original graph.

For the temporal simple path enumeration problem, a straightforward upper-bound graph would be the projected

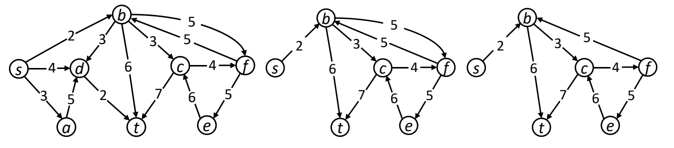

(a) dtTSG (b) esTSG (c) tgTSG

Fig. 2: Upper-bound graphs of baselines within interval $[2, 7]$

graph $\mathcal{G}_{[\tau_b,\tau_e]}$ of $\mathcal{G}$, whose computation method is called dtTSG in this paper. Recently, Jin et al. [18] proposes two graph reduction methods (named esTSG and tgTSG) before the path enumeration, whose construction results also can serve as upper-bound graphs of our problem. Specifically, esTSG excludes edges that do not belong to any path from s to t with non-decreasing timestamps, while tgTSG discards edges that are not contained in any path from s to t with ascending timestamps. It is easy to obtain that these two methods can construct smaller upper-bound graphs than the projected graph. In our experiment, we design three baseline algorithms EPdtTSG, EPesTSG and EPtgTSG based on the above three upper-bound graph generation methods, i.e., we first generate an upper-bound graph using each method, then enumerate temporal simple paths on the generated upper-bound graph to obtain the results by combining these paths.

***Example* 3** *Given the directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in Fig. 1(a). Suppose the source vertex is s, the target vertex is t, and the time interval is $[2, 7]$. In Figs. 2(a)-2(c), we illustrate three upper-bound graphs obtained by three baselines. As observed, esTSG and tgTSG produce tighter upper-bound graphs compared to dtTSG.*

Limitations of baselines. Although the baseline methods can successfully generate temporal simple path graphs, they still have limitations that present opportunities for improvement.

i) The first one is that the upper-bound graph could be further reduced. For example, edge $e(e, c, 6)$ should not be included in the results because it only appears in a temporal path with a cycle. However, since these upper-bound graphs ignore the constraint of the simple path, such edges are not pruned.

ii) The second efficiency limitation is due to the enumeration process. Since some edges may appear in multiple temporal simple paths, this may lead to repeated verifications to check whether an edge is already included in the result. Therefore, the development of efficient techniques for path enumeration can further enhance the performance of the algorithm.

B. The Verification in Upper-bound Graph Framework

To better approximate the $tspG$ by the upper-bound graph and avoid repeatedly checking edges that are already added to the result by previously discovered paths, we propose a novel method named Verification in Upper−bound Graph (i.e., VUG) with two main components, namely Upper-bound Graph Generation and Escaped Edges Verification, whose framework is shown in Algorithm 1.

i) **Upper-bound Graph Generation**. We generate the upper-bound graph for $tspG$ in two phases. In phase one, the quick upper-bound graph $\mathcal{G}_q$ is computed based on temporal constraints. In phase two, $\mathcal{G}_q$ is further reduced to the tight upper-bound graph $\mathcal{G}_t$ by applying the simple path constraint (more details can be found in Section IV).

Algorithm 1: Framework of VUG

```
Input  : a directed temporal graph $\mathcal{G}$, a source vertex $s$, a
         target vertex $t$ and a query time interval $[\tau_b, \tau_e]$
Output: the temporal simple path graph $tspG_{[\tau_b,\tau_e]}(s, t)$
  // Upper-bound Graph Generation
1 $\mathcal{G}_q \leftarrow$ QuickUBG ($\mathcal{G}$, $\mathcal{A}$, $\mathcal{D}$);      /* Algorithm 2 */
2 $\mathcal{G}_t \leftarrow$ TightUBG ($G_q$, $TCV$);       /* Algorithm 5 */
  // Escaped Edges Verification
3 $tspG \leftarrow$ EEV ($s$, $t$, $[\tau_b, \tau_e]$, $\mathcal{G}_t$);    /* Algorithm 6 */
4 return $tspG$;
```

ii) **Escaped Edges Verification**. We obtain exact $tspG$ by extending bidirectional DFS, i.e., select an unverified edge $e(u, v, \tau)$, identify a path $p^*_{[\tau_b,\tau_e]}(s, t)$ through $e(u, v, \tau)$ and add a set of vertices and edges to $tspG$ (more details can be found in Section V).

IV. Upper-bound Graph Generation

In this section, we present two upper-bound graph computation methods, named quick upper-bound graph generation (QuickUBG) and tight upper-bound graph generation (TightUBG). Since the size of the initial graph is very large, we start with a fast approach QuickUBG to quickly obtain an upper-bound graph $\mathcal{G}_q$, allowing us to prune unpromising vertices and edges. Then, we apply our stronger method TightUBG to obtain a more precise upper-bound graph $\mathcal{G}_t$.

A. Quick Upper-bound Graph Generation

Given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a temporal simple path graph query of s to t in the time interval $[\tau_b, \tau_e]$, for an edge to be included in the $tspG$, it must be contained in at least one temporal path in $\mathcal{P}_{[\tau_b,\tau_e]}(s, t)$. Focusing on this condition, we have the observation below.

***Observation* 1** *An edge $e(u, v, \tau)$ is contained in a temporal path $p_{[\tau_b,\tau_e]}(s, t) \in \mathcal{P}_{[\tau_b,\tau_e]}(s, t)$ iff any one of the following conditions holds:*

i) *$u \neq s$, $v \neq t$, there exist two temporal paths $p_{[\tau_b,\tau_i]}(s, u)$ and $p_{[\tau_j,\tau_e]}(v, t)$ s.t. $\tau_i < \tau < \tau_j$.*

ii) *$u = s$, $v \neq t$, there exist a temporal path $p_{[\tau_j,\tau_e]}(v, t)$ s.t. $\tau_b \leq \tau < \tau_j$.*

iii) *$u \neq s$, $v = t$, there exist a temporal path $p_{[\tau_b,\tau_i]}(s, u)$ s.t. $\tau_i < \tau \leq \tau_e$.*

iv) *$u = s$, $v = t$, $\tau_b \leq \tau \leq \tau_e$.*

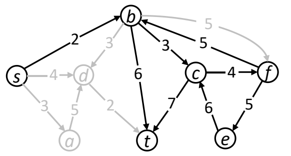
Using the above observation as an edge exclusion condition, we can effectively construct an upper-bound graph for $tspG$, referred to as quick upper-bound graph $\mathcal{G}_q$, that only includes edges that appear in at least one temporal path $p_{[\tau_b,\tau_e]}(s, t)$. Before presenting our detailed strategy, we first formally define the concepts of *arrival time* and *departure time*.

u	$\mathcal{A}$
s	1
a	3
b	2
c	3
d	$4 \rightarrow 3$
e	5
f	$5 \rightarrow 4$
t	$+\infty$

(a) $\mathcal{A}(u)$

u	$\mathcal{D}$
s	$-\infty$
a	$-\infty$
b	6
c	7
d	2
e	6
f	5
t	8

(b) $\mathcal{D}(u)$



(c) $\mathcal{G}_q$

Fig. 3: Quick Upper-bound Graph Generation

***Definition* 3 (Arrival & Departure Time)** *Given a temporal path $p = p_{[\tau_b,\tau_e]}(s,u) = \langle e(s,\cdot,\cdot),\cdots,e(w_a,u,\tau_a)\rangle$, the arrival time of u regarding s in this path, denoted by $a(p,u)$, is the timestamp of the last edge within the path, i.e., $a(p,u) = \tau_a$. Similarly, given a temporal path $p = p_{[\tau_b,\tau_e]}(v,t) = \langle e(v,w_d,\tau_d),\cdots,e(\cdot,t,\cdot)\rangle$, the departure time of v regarding t in this path, denoted by $d(p,v)$, is the timestamp of the first edge within the path, i.e., $d(p,v) = \tau_d$.*

Since each vertex may appear in multiple temporal paths from the source vertex to itself or from it to the target vertex, it can have multiple arrival and departure times. Therefore, we define *polarity time* to store each vertex's earliest arrival time and latest departure time.

***Definition* 4 (Polarity Time)** *For each vertex $u \in \mathcal{V} \setminus \{s\}$, the earliest arrival time of u, denoted by $\mathcal{A}(u)$, is the smallest arrival time of u regarding s among all the paths from s to u within $[\tau_b,\tau_e]$, i.e., $\mathcal{A}(u) = \min\{a(p,u)|p \in \mathcal{P}_{[\tau_b,\tau_e]}(s,u)\}$. Similarly, for each vertex $u \in \mathcal{V} \setminus \{t\}$, the latest departure time of u, denoted by $\mathcal{D}(u)$, is the largest departure time of u regarding t among all the paths from u to t within $[\tau_b,\tau_e]$, i.e., $\mathcal{D}(u) = \max\{d(p,u)|p \in \mathcal{P}_{[\tau_b,\tau_e]}(u,t)\}$.*

***Example* 4** *Given the directed temporal graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$ in Fig. 1(a), a source vertex s, a target vertex t and a time interval $[2,7]$. For the vertex $f \in \mathcal{V} \setminus \{s,t\}$, $\mathcal{P}_{[2,7]}(s,f) = \{\langle e(s,b,2),e(b,f,5)\rangle, \langle e(s,b,2),e(b,c,3),e(c,f,4)\rangle\}$, thus $\mathcal{A}(f) = \min\{4,5\} = 4$. Similarly, $\mathcal{P}_{[2,7]}(f,t) = \{\langle e(f,e,5), e(e,c,6),e(c,t,7)\rangle, \langle e(f,b,5),e(b,t,6)\rangle\}$, thus $\mathcal{D}(f) = 5$.*

With Observation 1 and Definition 4, we can determine whether an edge is contained in any temporal path $p_{[\tau_b,\tau_e]}(s,t)$ based on the polarity time of its end vertices using the following lemma.

***Lemma* 1** *Given a edge $e(u,v,\tau) \in \mathcal{E}$, there exists a temporal path from the source vertex s to the target vertex t containing $e(u,v,\tau)$ within $[\tau_b,\tau_e]$ iff $\mathcal{A}(u) < \tau < \mathcal{D}(v)$.*

Based on Lemma 1, we can efficiently generate $\mathcal{G}_q$, with the detailed process illustrated in Algorithm 2. Keep in mind that, here we suppose the polarity time of every vertex is already given, and its computation will be discussed later. A running example of Algorithm 2 is provided as follows.

***Example* 5** *Given the directed temporal graph $\mathcal{G}$ in Fig. 1(a), the precomputed polarity time in Figs. 3(a)-3(b), the result of edge exclusion is shown in Fig. 3(c). We exclude $e(s,a,3)$ for $3 > -\infty$=$\mathcal{D}(a)$, $e(d,t,2)$ as $2 < 3 = \mathcal{A}(d)$.*

Algorithm 2: Quick Upper-bound Graph Generation

```
Input  : a directed temporal graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$, earliest
         arrival time $\mathcal{A}(u)$ and latest departure time
         $\mathcal{D}(u)$ for each vertex $u \in \mathcal{V}$
Output: quick upper-bound graph $\mathcal{G}_q$=$(\mathcal{V}_q,\mathcal{E}_q)$
1 $\mathcal{V}_q = \emptyset$, $\mathcal{E}_q = \emptyset$;
2 for each $e(u,v,\tau) \in \mathcal{E}$ do
3     if $\mathcal{A}(u) < \tau \wedge \tau < \mathcal{D}(v)$ then
4         $\mathcal{V}_q \leftarrow \mathcal{V}_q \cup \{u,v\}$;
5         $\mathcal{E}_q \leftarrow \mathcal{E}_q \cup \{e(u,v,\tau)\}$;        /* Lemma 1 */
6 return $(\mathcal{V}_q,\mathcal{E}_q)$
```

Algorithm 3: Polarity Time Computation

```
Input  : a directed temporal graph $\mathcal{G} = (\mathcal{V},\mathcal{E})$, a source
         vertex $s \in \mathcal{V}$, a target vertex $t \in \mathcal{V}$ and a query
         time interval $[\tau_b,\tau_e]$
Output: the earliest arrival time $\mathcal{A}(u)$ and latest
         departure time $\mathcal{D}(u)$ for each vertex $u \in \mathcal{V}$
1 $\mathcal{A}(s) \leftarrow \tau_b - 1$, $\mathcal{A}(u) \leftarrow +\infty$ for each $u \in \mathcal{V} \setminus \{s\}$;
2 $\mathcal{D}(t) \leftarrow \tau_e + 1$, $\mathcal{D}(u) \leftarrow -\infty$ for each $u \in \mathcal{V} \setminus \{t\}$;
3 $Q \leftarrow$ a queue containing $s$;
4 while $Q \neq \emptyset$ do
5     $u \leftarrow Q.pop()$;
6     for each $(v,\tau) \in N_{out}(u,\mathcal{G})$ s.t. $v \neq t$ do
7         if $\tau > \tau_e \vee \mathcal{A}(u) \geq \tau \vee \tau \geq \mathcal{A}(v)$ then continue;
8         $\mathcal{A}(v) \leftarrow \tau$;
9         if $\tau \neq \tau_e \wedge v \notin Q$ then $Q.push(v)$;
10 repeat Lines 3-9 with necessary adjustment for updating
   latest departure time;
11 return $\mathcal{A}(u), \mathcal{D}(u)$ for each $u \in \mathcal{V}$;
```

***Theorem* 1** *The time complexity of Algorithm 2 is $O(m)$, and its space complexity is $O(n+m)$.*

Next, we introduce our method for calculating the polarity time for each vertex. With a traversal in $\mathcal{G}$ starting from the source vertex s, we record the current timestamp as the arrival time for each reachable vertex. Since there may be multiple paths from s to a vertex u, we update the arrival time of u if a path allows earlier arrival at u. Similarly, we start a reverse traversal in $\mathcal{G}$ from the target vertex t and record the latest departure time for each vertex. For computational simplicity, we define $\mathcal{A}(s) = \tau_b-1$, $\mathcal{D}(t) = \tau_e+1$. The detailed pseudocode of the polarity time computation is given in Algorithm 3. Note that, here, $\mathcal{A}(u) = +\infty$ (resp. $\mathcal{D}(u) = -\infty$) implies there exists no temporal path from s to u (resp. from u to t) within $[\tau_b,\tau_e]$ without passing through t (resp. s).

Polarity Time Computation. In Algorithm 3, $\mathcal{G}$ is stored such that, within the neighbor list $N_{out}(u,\mathcal{G})$, $N_{in}(u,\mathcal{G})$ of each vertex $u \in \mathcal{V}(\mathcal{G})$, neighbors are sorted based on their timestamps. In Lines 4-9, for each popped vertex u from Q, we iteratively explore its out-neighbors. It is worth mentioning that, a pointer in $N_{out}(u,\mathcal{G})$ is maintained for each vertex

$u \in \mathcal{V}(\mathcal{G})$ to help filter out (v, τ) that has been processed during a previous visit to u, which avoids scanning the entire $N_{out}(u, \mathcal{G})$ each time u is visited. In Line 7, an out-neighbor (v, τ) with $\mathcal{A}(u) \geq \tau$ or $\tau \geq \mathcal{A}(v)$ is ignored since a new temporal path $p_{[\tau_b, \tau]}(s, v)$ cannot be formed or $p_{[\tau_b, \tau]}(s, v)$ has already been explored previously. We update $\mathcal{A}(v)$ in Line 8 as we find a new temporal path $p_{[\tau_b, \tau]}(s, v)$ with an earlier arrival time for v, and in Line 9 we add v to Q for further exploration if it is not yet in Q. We also repeat the process in Lines 3-9 by toggling between s and t, "out" and "in", $\mathcal{A}$ and $\mathcal{D}$, τ_e and τ_b, $>$ and $<$, $\geq$ and $\leq$ to obtain $\mathcal{D}(\cdot)$.

***Example* 6** *Given the directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in Fig. 1(a). For the query of s to t within the time interval $[2, 7]$, the earliest arrival time of each vertex is shown in Fig. 3(a). As we process s, its out-neighbors $\{(b, 2), (a, 3), (d, 4)\}$ are checked. We add b, a, d to Q and update their earliest arrival time to $2, 3, 4$, respectively. Next, $\mathcal{A}(d)$ is updated to 3 when b is being processed. Be aware that, we do not add d to Q since it is currently in Q. Then, a is processed without making changes to $\mathcal{A}(d)$, since $5 > 3 = \mathcal{A}(d)$.*

***Theorem* 2** *The time complexity of Algorithm 3 is $O(n + m)$, and its space complexity is $O(n)$.*

B. Tight Upper-bound Graph Generation

Given the quick upper-bound graph $\mathcal{G}_q$ of a directed temporal graph $\mathcal{G}$ for the query of s to t within $[\tau_b, \tau_e]$, we further remove edges that can be efficiently determined as not contained in any temporal simple path $p^*_{[\tau_b, \tau_e]}(s, t)$ from $\mathcal{G}_q$. Relative to this condition, we give the observation below.

***Observation* 2** *For an edge $e(u, v, \tau)$, where $u \neq s$ and $v \neq t$, it is contained in a temporal simple path $p^*_{[\tau_b, \tau_e]}(s, t)$ iff there exist two temporal simple paths $p^*_{[\tau_b, \tau_i]}(s, u)$ and $p^*_{[\tau_j, \tau_e]}(v, t)$ s.t. $i)$ $\tau_i < \tau < \tau_j$ and $ii)$ $\mathcal{V}(p^*_{[\tau_b, \tau_i]}(s, u)) \cap \mathcal{V}(p^*_{[\tau_j, \tau_e]}(v, t)) = \emptyset$.*

For any edge $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$ with $u \neq s$ and $v \neq t$, following Algorithm 2 and 3, it is straightforward to show that there exist two temporal paths $p_{[\tau_b, \tau_i]}(s, u)$ and $p_{[\tau_j, \tau_e]}(v, t)$ s.t. $t \notin \mathcal{V}(p_{[\tau_b, \tau_i]}(s, u))$, $s \notin \mathcal{V}(p_{[\tau_j, \tau_e]}(v, t))$, $\tau_i < \tau < \tau_j$. Clearly, we can obtain a temporal simple path $p^*_{[\tau_b, \tau_i]}(s, u)$ (resp. $p^*_{[\tau_j, \tau_e]}(v, t)$) from $p_{[\tau_b, \tau_i]}(s, u)$ (resp. $p_{[\tau_j, \tau_e]}(v, t)$) by removing edges that form cycles in the path. Based on this analysis, we can have the following lemma.

***Lemma* 2** *If an edge $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$, then one of the following two conditions must be satisfied:*

- *$i)$ $u \neq s$, $v \neq t$, there exist two temporal simple paths $p^*_{[\tau_b, \tau_i]}(s, u)$ and $p^*_{[\tau_j, \tau_e]}(v, t)$ s.t. $\tau_i < \tau < \tau_j$, $t \notin p^*_{[\tau_b, \tau_i]}(s, u)$, $s \notin p^*_{[\tau_j, \tau_e]}(v, t)$.*
- *$ii)$ $u = s$ or $v = t$, there exists a temporal simple path $p^*_{[\tau_b, \tau_e]}(s, t)$ s.t. $e(u, v, \tau) \in p^*_{[\tau_b, \tau_e]}(s, t)$.*

Disappointingly, even if we can find two simple paths satisfying condition $i)$ of Observation 2, there may still be duplicated vertices across two simple paths, such that condition $ii)$ is unsatisfiable. Exhaustively evaluating condition $ii)$ for every possible combination of $p^*_{[\tau_b, \tau_i]}(s, u)$ and $p^*_{[\tau_j, \tau_e]}(v, t)$ is computationally expensive. Therefore, at this stage, we only aim to efficiently identify edges that clearly do not satisfy condition $ii)$ and construct a tighter upper-bound graph $\mathcal{G}_t$ for $tspG$. The high-level idea of our method is that if there exists a vertex w appearing in all $p^*_{[\tau_b, \tau_i]}(s, u)$ and $p^*_{[\tau_j, \tau_e]}(v, t)$, condition $ii)$ will not hold. Therefore, in the following, we will focus on identifying such common vertex w (note that, $w \neq s$ and $w \neq t$ as discussed previously), and we formally define the *time-stream common vertices* as follows:

***Definition* 5 (Time-stream Common Vertices)** *Given a directed temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, two vertices s, $t \in \mathcal{V}$, a time interval $[\tau_b, \tau_e]$ and a timestamp $\tau \in [\tau_b, \tau_e]$, the time-stream common vertices of $\mathcal{P}^*_{[\tau_b, \tau]}(s, u)$, denoted by $TCV_\tau(s, u)$, is the set of vertices (except s) appearing in all temporal simple paths from s to u not containing t in time interval $[\tau_b, \tau]$, i.e.,*

$$TCV_\tau(s, u) = \bigcap_{p^* \in \mathcal{P}^*_{[\tau_b, \tau]}(s, u) \ s.t. \ t \notin \mathcal{V}(p^*)} \mathcal{V}(p^* \setminus s) \quad (1)$$

Similarly, given $\tau \in [\tau_b, \tau_e]$, $TCV_\tau(u, t)$ is the set of vertices (except t) appearing in all temporal simple paths $p^*_{[\tau, \tau_e]}(u, t)$ not containing s, i.e.,

$$TCV_\tau(u, t) = \bigcap_{p^* \in \mathcal{P}^*_{[\tau, \tau_e]}(u, t) \ s.t. \ s \notin \mathcal{V}(p^*)} \mathcal{V}(p^* \setminus t) \quad (2)$$

***Lemma* 3** *Given an edge $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$ with $u \neq s$, $v \neq t$, if it is contained in a temporal simple path $p^*_{[\tau_b, \tau_e]}(s, t)$, then $\exists \tau_i, \tau_j \in [\tau_b, \tau_e]$ s.t. $i)$ $\tau_i < \tau < \tau_j$, and $ii)$ $TCV_{\tau_i}(s, u) \cap TCV_{\tau_j}(v, t) = \emptyset$; but not necessarily the reverse.*

Lemma 3 reveals that the edge $e(u, v, \tau)$ cannot be contained in any temporal simple path from s to t in $[\tau_b, \tau_e]$ if $\forall$ $\tau_i, \tau_j \in [\tau_b, \tau_e]$ s.t. $\tau_i < \tau < \tau_j$, $TCV_{\tau_i}(s, u) \cap TCV_{\tau_j}(v, t) \neq \emptyset$. This idea can be used to exclude unpromising edges from $\mathcal{G}_q$. However, it is non-trivial to efficiently compute and store time-stream common vertices $TCV_\tau(s, u)$ and $TCV_\tau(u, t)$ for all vertices $u \in \mathcal{V}(\mathcal{G}_q)$ and all timestamps $\tau \in [\tau_b, \tau_e]$, with the main challenges being two-fold.

- $i)$ The span of the interval $[\tau_b, \tau_e]$ can be large. Therefore, storing time-stream common vertices for all $\tau \in [\tau_b, \tau_e]$ requires significant space.
- $ii)$ Explicitly enumerating temporal simple paths to obtain time-stream common vertices is computation-intensive.

To address the first challenge, for each vertex $u \in \mathcal{V}(\mathcal{G}_q) \setminus \{s, t\}$, instead of computing and storing time-stream common vertices $TCV_\tau(s, u)$ (resp. $TCV_\tau(u, t)$) for every possible timestamp $\tau \in [\tau_b, \tau_e]$, we can compute and store those for only a finite subset of timestamps, time-stream common vertices for other timestamps can be inferred from the stored subset. This idea is supported by the following lemmas.

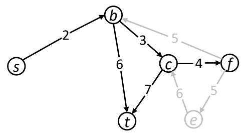
***Lemma* 4** *For each in-coming edge $e(v, u, \tau)$ (resp. out-going edge $e(u, v, \tau)$) of u in $\mathcal{G}_q$, there exists a temporal simple*

u	τ	$\boldsymbol{TCV_\tau(s,u)}$
b	2	$\{b\}$
	5	*null*
c	3	$\{b,c\}$
	6	$\{b,c\}$
f	4	$\{b,c,f\}$
e	5	$\{b,c,f,e\}$

(a) $TCV_\tau(s,u)$

u	τ	$\boldsymbol{TCV_\tau(u,t)}$
b	6	$\{b\}$
	3	*null*
c	7	$\{c\}$
	4	*null*
f	5	$\{c,e,f\} \rightarrow \{f\}$
e	6	$\{c,e\}$

(b) $TCV_\tau(u,t)$



(c) $\mathcal{G}_t$

Fig. 4: Tight Upper-bound Graph Generation

path $p^*_{[\tau_b,\tau]}(s,u)$ *(resp.* $p^*_{[\tau,\tau_e]}(u,t)$*) such that* $e(v,u,\tau) \in \mathcal{E}(p^*_{[\tau_b,\tau]}(s,u))$ *(resp.* $e(u,v,\tau) \in \mathcal{E}(p^*_{[\tau,\tau_e]}(u,t))$*) and* $t \notin \mathcal{V}(p^*_{[\tau_b,\tau]}(s,u))$ *(resp.* $s \notin \mathcal{V}(p^*_{[\tau,\tau_e]}(u,t))$*).*

***Lemma* 5** *Given* τ *and* $\tau_l = \max\{\tau_i | \tau_i \in \mathcal{T}_{in}(u,\mathcal{G}_q), \tau_i \le \tau\}$, *we have* $TCV_\tau(s,u) = TCV_{\tau_l}(s,u)$. *Similarly, given* $\tau_r = \min\{\tau_j | \tau_j \in \mathcal{T}_{out}(u,\mathcal{G}_q), \tau_j \ge \tau\}$, $TCV_\tau(u,t) = TCV_{\tau_r}(u,t)$.

Recall that, $\mathcal{T}_{in}(u,\mathcal{G}_q)$ (resp. $\mathcal{T}_{out}(u,\mathcal{G}_q)$) represents all distinct timestamps in $N_{in}(u,\mathcal{G}_q)$ (resp. $N_{out}(u,\mathcal{G}_q)$). Based on Lemma 4 and 5, we only need to compute and store $TCV_\tau(s,u)$ (resp. $TCV_\tau(u,t)$) for every timestamp in $\mathcal{T}_{in}(u,\mathcal{G}_q)$ (resp. $\mathcal{T}_{out}(u,\mathcal{G}_q)$). Specifically, , we store time-stream common vertices $TCV_\tau(s,u)$ of each timestamp $\tau \in \mathcal{T}_{in}(u,\mathcal{G}_q)$, and organize them in ascending order based on τ. Similarly, we store time-stream common vertices $TCV_\tau(u,t)$ of each timestamp $\tau \in \mathcal{T}_{out}(u,\mathcal{G}_q)$, and organize them in descending order based on τ.

***Example* 7** *Fig. 4(a) and Fig. 4(b) show the data structure of* $TCV_\cdot(s,\cdot)$ *and* $TCV_\cdot(\cdot,t)$ *that correspond to the quick upper-bound graph* $\mathcal{G}_q$ *in Fig. 3(c). Consider* $TCV_\cdot(f,t)$ *as an example,* $N_{out}(f,\mathcal{G}_q) = \{(b,5),(e,5)\}$, *therefore* $\mathcal{T}_{out}(f,\mathcal{G}_q) = \{5\}$. *As depicted in Fig. 4(b), there is only one entry, i.e.,* $TCV_5(f,t)$, *in* $TCV_\cdot(f,t)$.

Regarding the second challenge, according to Lemma 4, a temporal simple path $p^*_{[\tau_b,\tau]}(s,u)$ is composed of two parts, $\langle e(v,u,\tau')\rangle$ and $p^*_{[\tau_b,\tau'-1]}(s,v)$, where $(v,\tau') \in N_{in}(u,\mathcal{G}_q)$, $\tau' \le \tau$ and $u \notin \mathcal{V}(p^*_{[\tau_b,\tau'-1]}(s,v))$, this inspires us to compute time-stream common vertices of a vertex based on those of its neighbors. However, the main obstacle is the need to check for the simple path constraint during the computation. To overcome this obstacle, we propose the following lemma.

***Lemma* 6** *Intersecting the vertex set of each temporal simple path is equivalent to intersecting that of each temporal path, i.e.,* $TCV_\tau(s,u) = \bigcap_{p\in\mathcal{P}_{[\tau_b,\tau]}(s,u)\ s.t.\ t\notin\mathcal{V}(p)} \mathcal{V}(p \setminus s)$; *similarly,* $TCV_\tau(u,t) = \bigcap_{p\in\mathcal{P}_{[\tau,\tau_e]}(u,t)\ s.t.\ s\notin\mathcal{V}(p)} \mathcal{V}(p \setminus t)$.

According to Lemma 6, we can compute $TCV_\tau(s,u)$ (resp. $TCV_\tau(u,t)$) based on the temporal paths $\mathcal{P}_{[\tau_b,\tau]}(s,u)$ (resp. $\mathcal{P}_{[\tau,\tau_e]}(u,t)$) without considering the simple path constraint. Therefore, given a vertex $w \neq s$, $w \in TCV_\tau(s,u)$ iff $\forall$ $(v,\tau') \in N_{in}(u,\mathcal{G}_q)$ s.t. $\tau' \le \tau$ and $w \in (TCV_{\tau'-1}(s,v) \cup \{u\})$. Therefore, intersecting $(TCV_{\tau'-1}(s,v) \cup \{u\})$ for each $(v,\tau') \in N_{in}(u,\mathcal{G}_q)$ s.t. $\tau' \le \tau$ leads to $TCV_\tau(s,u)$. Based on these, we design the computation of $TCV_\cdot(s,\cdot)$ and $TCV_\cdot(\cdot,t)$ in a recursive way with the formulas formally defined below.

Recursive Computation. Given the source vertex and target vertex $s,t \in \mathcal{V}(\mathcal{G}_q)$, the time interval $[\tau_b,\tau_e]$, for each vertex $u \in \mathcal{V}(\mathcal{G}_q) \setminus \{s,t\}$ and timestamp $\tau \in [\tau_b,\tau_e]$, $TCV_\tau(s,u)$ is computed based on the time-stream common vertices of u's in-neighbors. For ease of computation, we define the base case to be: for $\forall\tau \in [\tau_b - 1, \tau_e - 1]$, $TCV_\tau(s,s) = \emptyset$.

$$TCV_\tau(s,u) = \bigcap_{\substack{(v,\tau')\in N_{in}(u,\mathcal{G}_q) \\ \tau' \le \tau}} (TCV_{\tau'-1}(s,v) \cup \{u\}) \quad (3)$$

Similarly, $TCV_\tau(u,t)$ is computed based on the time-stream common vertices of u's out-neighbors. And we define the base case to be: for $\forall\tau \in [\tau_b + 1, \tau_e + 1]$, $TCV_\tau(t,t) = \emptyset$.

$$TCV_\tau(u,t) = \bigcap_{\substack{(v,\tau')\in N_{out}(u,\mathcal{G}_q) \\ \tau' \ge \tau}} (TCV_{\tau'+1}(v,t) \cup \{u\}) \quad (4)$$

The computation of $TCV_\tau(s,u)$ can be interpreted as two steps: *i*) intersect $(TCV_{\tau'-1}(s,v) \cup \{u\})$ for all $(v,\tau') \in N_{in}(u,\mathcal{G}_q)$ s.t. $\tau' \le \tau - 1$; *ii*) intersect $(TCV_{\tau-1}(s,v) \cup \{u\})$ for all $(v,\tau) \in N_{in}(u,\mathcal{G}_q)$. The computation of $TCV_\tau(u,t)$ can be interpreted in a similar manner, and we omit the details here. Clearly, the result of step *i*) is $TCV_{\tau-1}(s,u)$ if $TCV_{\tau-1}(s,u)$ exists, therefore, $TCV_\tau(s,u) \subseteq TCV_{\tau-1}(s,u)$. Based on this insight, $TCV_{\tau-1}(s,u)$ can be an intermediate result of $TCV_\tau(s,u)$, allowing the computation of $TCV_\tau(s,u)$ to begin from $TCV_{\tau-1}(s,u)$, and we also have the following lemma that serves as a pruning rule in the computation of time-stream common vertices.

***Lemma* 7** *If* $TCV_{\tau_m}(s,u) = \{u\}$, *then* $\forall$ $\tau_n > \tau_m$, $TCV_{\tau_n}(s,u) = \{u\}$. *Similarly, if* $TCV_{\tau_m}(u,t) = \{u\}$, *then* $\forall$ $\tau_n < \tau_m$, $TCV_{\tau_n}(u,t) = \{u\}$.

Time-stream Common Vertices Computation. The detailed pseudocode is shown in Algorithm 4. Lines 3-23 compute $TCV_\tau(s,u)$ of each vertex $u \in \mathcal{V}(\mathcal{G}_q) \setminus \{s,t\}$ where $\tau \in \mathcal{T}_{in}(u,\mathcal{G}_q)$. All entries in $TCV_\cdot(s,u)$ are initially set to *null* in Line 4. For each specific vertex u, $TCV_\tau(s,u)$ will be computed sequentially, in ascending order of τ. A pointer in $TCV_\cdot(s,u)$ is maintained for each vertex u to enable constant-time access to the entry currently being processed, which is initialized to 1 Line 6, indicating the $first$ entry (alternatively, the $first$ timestamp) in $TCV_\cdot(s,u)$ is initially in processing.

Edges in $\mathcal{E}(\mathcal{G}_q)$ are arranged in non-descending temporal order when $\mathcal{G}_q$ is generated, and Line 7 scans them forward during the computation of $TCV_\cdot(s,\cdot)$. This ensures the processing of $TCV_\cdot(s,\cdot)$ with a smaller timestamp is finalized before the processing of $TCV_\cdot(s,\cdot)$ with a larger timestamp begins. For each scanned edge $e(v,u,\tau)$, we will process $TCV_\tau(s,u)$ accordingly. Lines 11-13 obtain either the currently processed timestamp or the last processed timestamp in $TCV_\cdot(s,v)$, whichever is closer to but smaller than τ, and refer to this timestamp as τ_j. Then, $TCV_{\tau_j}(s,v)$ is efficiently accessed in Lines 14-15 through the pointer corresponding to τ_j in $TCV_\cdot(s,v)$, and $TCV_{\tau_j}(s,v)$ is equal to $TCV_{\tau-1}(s,v)$ according to Lemma 5.

```
Algorithm 4: Time-stream Common Vertices Computation
Input : quick upper-bound graph $\mathcal{G}_q$ of a temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a source vertex $s \in \mathcal{V}$, a target vertex $t \in \mathcal{V}$ and a time interval $[\tau_b, \tau_e]$
Output: the time-stream common vertices $TCV_{\cdot}(s, \cdot)$ and $TCV_{\cdot}(\cdot, t)$
1  $TCV_{\tau_b-1}(s, s) \leftarrow \emptyset$, $P(s) \leftarrow 1$;
2  $TCV_{\tau_e+1}(t, t) \leftarrow \emptyset$, $P(t) \leftarrow 1$;
3  for $u \in \mathcal{V}(\mathcal{G}_q) \setminus \{s, t\}$ do
4      $TCV_\tau(s, u) \leftarrow null$ for each $\tau \in \mathcal{T}_{in}(u, \mathcal{G}_q)$;
5      mark $u$ as uncompleted;
6      $P(u) \leftarrow 1$;
7  for $e(v, u, \tau) \in \mathcal{E}(\mathcal{G}_q)$ do
8      if $u = t \vee u$ is completed then continue;
9      $i \leftarrow P(u)$, $\tau_i \leftarrow i$-th timestamp in $TCV_{\cdot}(s, u)$;
10     $TCV_{\tau_i}(s, u) \leftarrow i$-th entry in $TCV_{\cdot}(s, u)$;
11     $j \leftarrow P(v)$, $\tau_j \leftarrow j$-th timestamp in $TCV_{\cdot}(s, v)$;
12     if $\tau_j = \tau$ then
13         $j \leftarrow j - 1$, $\tau_j \leftarrow j$-th timestamp in $TCV_{\cdot}(s, v)$;
14     $TCV_{\tau_j}(s, v) \leftarrow j$-th entry in $TCV_{\cdot}(s, v)$;
15     if $TCV_{\tau_j}(s, v) = null$ then $TCV_{\tau_j}(s, v) \leftarrow \{v\}$
16     if $TCV_{\tau_i}(s, u) = null$ then
17         $TCV_\tau(s, u) \leftarrow TCV_{\tau_j}(s, v) \cup \{u\}$;
18     else
19         $TCV_\tau(s, u) \leftarrow TCV_{\tau_i}(s, u) \cap (TCV_{\tau_j}(s, v) \cup \{u\})$;
20         if $\tau > \tau_i$ then $i \leftarrow i + 1$, $P(u) \leftarrow i$;
21     $i$-th entry in $TCV_{\cdot}(s, u) \leftarrow TCV_\tau(s, u)$;
22     if $TCV_\tau(s, u) = \{u\}$ then
23         mark $u$ as completed;                /* Lemma 7 */
24 repeat Lines 3-23 with necessary adjustment for computing $TCV_{\cdot}(\cdot, t)$;
25 return $TCV_{\cdot}(s, \cdot)$, $TCV_{\cdot}(\cdot, t)$
```

In Line 16, we determine whether we are starting to process the *first* entry in $TCV_{\cdot}(s, u)$. If so, we directly obtain $TCV_\tau(s, u)$ in Line 17. If not, we are facing two circumstances: If $\tau > \tau_i$, we can conclude that $TCV_{\tau_i}(s, u)$ is finalized, therefore we start the processing of a new entry in $TCV_{\cdot}(s, u)$, obtain the new entry by intersecting with the last processed entry, and update pointer $P(u)$ accordingly; if $\tau = \tau_i$, we continue the iterative intersection process of $TCV_{\tau_i}(s, u)$. In either $\tau > \tau_i$ or $\tau = \tau_i$, Line 19 fulfills the purpose, and $TCV_\tau(s, u)$ is efficiently stored at the position following the pointer $P(u)$ in Line 21.

Lines 22-23 and Line 8 implement the pruning strategy based on Lemma 7, specifically, if $TCV_\tau(s, u) = \{u\}$, any entry in $TCV_{\cdot}(s, u)$ with a larger timestamp than τ will be $\{u\}$, thus, we can omit the further computation of $TCV_{\cdot}(s, u)$ and mark u as completed. As a result, in Line 15, Since $\tau_j < \tau$, $TCV_{\tau_j}(s, v)$ should already been finalized and $TCV_{\tau_j}(s, v) = null$ if and only if v was completed previously, therefore we can conclude $TCV_{\tau_j}(s, v) = \{v\}$.

We repeat the process in Lines 3-23 by scanning edges backward in Line 7 and toggling between s and t, "out" and "in", $<$ and $>$ to obtain $TCV_{\cdot}(\cdot, t)$.

***Example* 8** *Consider $\mathcal{G}_q$ in Fig. 3(c), the corresponding time-stream common vertices $TCV_{\cdot}(s, \cdot)$ and $TCV_{\cdot}(\cdot, t)$ are shown in Figs. 4(a)-4(b). Take the computation of $TCV_5(f, t)$ as an example. When processing $e(f, e, 5)$, we have τ_i=5, $TCV_{\tau_i}(f, t)$=null, τ_j=6 and $TCV_{\tau_j}(e, t)=\{c, e\}$. Since $TCV_{\tau_i}(f, t)$=null, we are starting to process the first entry in $TCV_{\cdot}(f, t)$ at this time, and we obtain $TCV_5(f, t)=TCV_6(e, t) \cup \{f\}=\{c, e, f\}$ in Line 16. After that, we process $e(f, b, 5)$. This time, since τ_i=τ=5 we continue the iterative intersection process of $TCV_5(f, t)$ and update $TCV_5(f, t)$ to $\{f\}$. Then, the pruning condition in Line 22 is satisfied, and we mark f as completed.*

***Theorem* 3** *The time complexity of Algorithm 4 is $O(n+\theta \cdot m)$ where θ is the span of $[\tau_b, \tau_e]$, i.e., $\theta = \tau_e - \tau_b + 1$. The space complexity of Algorithm 4 is $O(n + \theta \cdot m)$.*

With the precomputed time-stream common vertices, we can exclude unpromising edges from $\mathcal{G}_q$ to generate $\mathcal{G}_t$ following the contrapositive of Lemma 3, i.e., for each $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$, check whether $\forall\ \tau_i, \tau_j \in [\tau_b, \tau_e]$ s.t. $\tau_i < \tau < \tau_j$, $TCV_{\tau_i}(s, u) \cap TCV_{\tau_j}(v, t) \neq \emptyset$. During the process, the following two optimization techniques can be used to accelerate computations. First, based on Lemma 2, all the out-going edges from s and in-coming edges to t can be directly added to $\mathcal{G}_t$. Second, instead of exhausting all possible combinations of τ_i and τ_j and intersecting the corresponding time-stream common vertices, we focus on the maximum timestamp τ_i in $\mathcal{T}_{in}(u, \mathcal{G}_q)$ and minimum timestamp τ_j in $\mathcal{T}_{out}(v, \mathcal{G}_q)$.

***Lemma* 8** *Given an edge $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$, $u \neq s$, $v \neq t$, let $\tau_l = \max\{\tau_i | \tau_i \in \mathcal{T}_{in}(u, \mathcal{G}_q) \wedge \tau_b \leq \tau_i < \tau\}$ and $\tau_r = \min\{\tau_j | \tau_j \in \mathcal{T}_{out}(v, \mathcal{G}_q) \wedge \tau < \tau_j \leq \tau_e\}$, if $TCV_{\tau_l}(s, u) \cap TCV_{\tau_r}(v, t) \neq \emptyset$, then for all $\tau_b \leq \tau_i < \tau_l$ and $\tau_r < \tau_j \leq \tau_e$, $TCV_{\tau_i}(s, u) \cap TCV_{\tau_j}(v, t) \neq \emptyset$.*

With Lemma 8, for each $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$ with $u \neq s$, $v \neq t$, we only need to perform the intersection operation once to determine if it should be included in the tight upper-bound graph $\mathcal{G}_t$. The detailed process of generating $\mathcal{G}_t$ is illustrated in Algorithm 5. Note that, edges are processed in a non-descending temporal order, and we utilize pointers in $TCV_{\cdot}(s, \cdot)$ and $TCV_{\cdot}(\cdot, t)$ to achieve constant-time access to entries in a similar way as Algorithm 4. A running example of the process is provided as follows.

***Example* 9** *Reconsider $\mathcal{G}_q$ in Fig. 3(c), and the precomputed $TCV_{\cdot}(s, \cdot)$ and $TCV_{\cdot}(\cdot, t)$ in Figs. 4(a) and 4(b), respectively. The resulting tight upper-bound graph $\mathcal{G}_t$ is shown in Fig. 4(c). We exclude $e(f, b, 5)$ as $TCV_4(s, f) \cap TCV_6(b, t) = \{b\} \neq \emptyset$.*

***Lemma* 9** *An edge $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_q)$ is in $\mathcal{E}(\mathcal{G}_t)$ iff one of the following two conditions is satisfied:*

i) $u \neq s$, $v \neq t$, $TCV_{\tau_l}(s, u) \cap TCV_{\tau_r}(v, t) = \emptyset$, where $\tau_l = \max\{\tau_i | \tau_i \in \mathcal{T}_{in}(u, \mathcal{G}_q) \wedge \tau_b \leq \tau_i < \tau\}$ and $\tau_r = \min\{\tau_j | \tau_j \in \mathcal{T}_{out}(v, \mathcal{G}_q) \wedge \tau < \tau_j \leq \tau_e\}$.

Algorithm 5: Tight Upper-bound Graph Generation

```
Input  : quick upper-bound graph $\mathcal{G}_q$=($\mathcal{V}_q, \mathcal{E}_q$), the
         time-stream common vertices $TCV_.(s,\cdot)$ and
         $TCV_.(\cdot,t)$
Output: tight upper-bound graph $\mathcal{G}_t$=($\mathcal{V}_t, \mathcal{E}_t$)
1  $\mathcal{V}_t = \emptyset, \mathcal{E}_t = \emptyset$;
2  $P_s(u) \leftarrow 1$, $P_t(u) \leftarrow |\mathcal{T}_{out}(u, \mathcal{G}_q)|$ for each $u \in \mathcal{V}_q$;
3  for $e(u,v,\tau) \in \mathcal{E}(\mathcal{G}_q)$ do
4      if $u = s$ or $v = t$ then
5          $\mathcal{V}_t \leftarrow \mathcal{V}_t \cup \{u,v\}$, $\mathcal{E}_t \leftarrow \mathcal{E}_t \cup \{e(u,v,\tau)\}$;
6          continue;                          /* Lemma 2 */
7      $i \leftarrow P_s(u)$, $j \leftarrow P_t(v)$;
8      while $i$-th timestamp in $TCV_.(s,u) < \tau$ do $i \leftarrow i+1$;
9      $i \leftarrow i-1$, $\tau_i \leftarrow i$-th timestamp in $TCV_.(s,u)$;
10     while $j$-th timestamp in $TCV_.(v,t) \leq \tau$ do $j \leftarrow j-1$;
11     $\tau_j \leftarrow j$-th timestamp in $TCV_.(v,t)$;
12     $P_s(u) \leftarrow i$, $P_t(v) \leftarrow j$;
13     $TCV_{\tau_i}(s,u) \leftarrow i$-th entry in $TCV_.(s,u)$;
14     if $TCV_{\tau_i}(s,u) = null$ then $TCV_{\tau_i}(s,u) = \{u\}$;
15     $TCV_{\tau_j}(v,t) \leftarrow j$-th entry in $TCV_.(v,t)$;
16     if $TCV_{\tau_j}(v,t) = null$ then $TCV_{\tau_j}(v,t) = \{v\}$;
17     if $TCV_{\tau_i}(s,u) \cap TCV_{\tau_j}(v,t) = \emptyset$ then
18         $\mathcal{V}_t \leftarrow \mathcal{V}_t \cup \{u,v\}$;
19         $\mathcal{E}_t \leftarrow \mathcal{E}_t \cup \{e(u,v,\tau)\}$;      /* Lemma 3 */
20 return ($\mathcal{V}_t, \mathcal{E}_t$)
```

ii) $u = s$ *or* $v = t$.

***Theorem* 4** *The time complexity of Algorithm 5 is* $O(n+\theta\cdot m)$*, and its space complexity is* $O(n+m)$*.*

V. Escaped Edges Verification

In this section, we propose an Escaped Edges Verification (EEV) method to generate exact $tspG$ from $\mathcal{G}_t$. The main idea of EEV is that we iteratively select an unverified edge $e \in \mathcal{G}_t$ and identify a temporal simple path $p^*_{[\tau_b,\tau_e]}(s,t)$ through e by applying the bidirectional DFS framework. All the edges along this path belong to $tspG$, and we add them to the result. This process stops until all the edges in $\mathcal{G}_t$ have been verified. Before discussing the details of our method, we introduce the following ideas that extend its core concept and are used in the construction of $tspG$.

***Lemma* 10** *Given* $e(u,v,\tau) \in \mathcal{E}(\mathcal{G}_t)$ *with* $u \neq s$, $v \neq t$, *if i)* $\exists\ e(s,u,\tau') \in \mathcal{E}(\mathcal{G}_t)$ *s.t.* $\tau_b \leq \tau' < \tau$, *or ii)* $\exists\ e(v,t,\tau') \in \mathcal{E}(\mathcal{G}_t)$ *s.t.* $\tau < \tau' \leq \tau_e$, *then there is a temporal simple path* $p^*_{[\tau_b,\tau_e]}(s,t)$ *containing the edge* $e(u,v,\tau)$.

By applying Lemma 2 and 10, we can confirm certain edges belong to $tspG$ without searching for the exact paths containing them, which reduces the number of edges to verify. Moreover, we can accelerate the edge verification process by adding edges across a batch of paths simultaneously based on an identified temporal simple path using the following lemma.

***Lemma* 11** *Given a temporal simple path* $p^* = \langle e(u_0,u_1,\tau_1), \ldots, e(u_{l-1},u_l,\tau_l)\rangle$, *for all integers* $1 \leq i \leq l$, $e(u_{i-1},u_i,\tau_i) \in \mathcal{E}(p^*)$ *can be replaced by any* $e(u_{i-1},u_i,\tau)$ *s.t. i) if* $i = 1$, $\tau \in [\tau_b, \tau_{i+1})$, *ii) if* $1 < i < l-1$, $\tau \in (\tau_{i-1}, \tau_{i+1})$, *iii) if* $i = l$, $\tau \in (\tau_i - 1, \tau_b]$ *, and the resulting path remains a temporal simple path from s to t in the time interval* $[\tau_b, \tau_e]$*.*

Algorithm 6: Escaped Edges Verification

```
Input  : tight upper-bound graph $\mathcal{G}_t$ of a temporal graph
         $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a source vertex $s \in \mathcal{V}$, a target
         vertex $t \in \mathcal{V}$ and a time interval $[\tau_b, \tau_e]$
Output: temporal simple path graph $tspG = (\mathcal{V}^*, \mathcal{E}^*)$
1  $\mathcal{V}^* = \emptyset, \mathcal{E}^* = \emptyset$;
2  for each $e \in \mathcal{E}(\mathcal{G}_t)$ mark $e$ as unverified;
3  for each $e(u,v,\tau)$ satisfies Lemma 2 or 10 do
4      mark $e(u,v,\tau)$ as verified;
5      $\mathcal{V}^* \leftarrow \mathcal{V}^* \cup \{u,v\}$; $\mathcal{E}^* \leftarrow \mathcal{E}^* \cup \{e(u,v,\tau)\}$;
6  for each $e(u,v,\tau) \in \mathcal{E}(\mathcal{G}_t)$ do
7      if $e(u,v,\tau)$ is verified then continue;
8      $\mathcal{S}_v \leftarrow$ an empty stack, $\mathcal{S}_e \leftarrow$ an empty stack;
9      if $\neg BiDirSearch(e(u,v,\tau))$ then continue;
10     build path $p^*$ from $\mathcal{S}_e$;
11     for $3 \leq i \leq l(p^*) - 2$ do
12         $e(u_{i-2}, u_{i-1}, \tau_{i-1}) \leftarrow$ the ($i$-1)-th edge in $p^*$;
13         $e(u_{i-1}, u_i, \tau_i) \leftarrow$ the $i$-th edge in $p^*$;
14         $e(u_i, u_{i+1}, \tau_{i+1}) \leftarrow$ the ($i$+1)-th edge in $p^*$;
15         $\mathcal{V}^* \leftarrow \mathcal{V}^* \cup \{u_{i-1}, u_i\}$;
16         for each $(w_j, \tau_j) \in N_{out}(u_{i-1}, \mathcal{G}_t)$ do
17             if $w_j = u_i$ and $\tau_j \in (\tau_{i-1}, \tau_{i+1})$ then
18                 mark $e(u_{i-1}, u_i, \tau_j)$ as verified;
19                 $\mathcal{E}^* \leftarrow \mathcal{E}^* \cup \{e(u_{i-1}, u_i, \tau_j)\}$;
                   /* Lemma 11 */
20 return $(\mathcal{V}^*, \mathcal{E}^*)$;
```

Following from Lemma 11, when a path $p^*_{[\tau_b,\tau_e]}(s,t)$ is identified during the bidirectional DFS, in addition to all the edges on this path itself, edges that are able to replace each of them to form another temporal simple path from s to t within $[\tau_b, \tau_e]$ can be confirmed at the same time. We give the pseudocode of our escaped edges verification method for producing exact $tspG$ in Algorithm 6.

Escaped Edges Verification. Algorithm 6 illustrates the detailed pseudocode. Lines 2-5 initialize the verification status for each edge in $\mathcal{G}_t$. For each out-neighbor (u, τ') of s (resp. in-neighbor of t), we mark the edge $e(s,u,\tau')$ (resp. $e(u,t,\tau')$) as verified and add it to $tspG$ based on Lemma 2; we also check the out-neighbors (resp. in-neighbors) of u, if a neighbor (v, τ) with a timestamp $\tau > \tau'$ (resp. $\tau < \tau'$) is found, we mark $e(u,v,\tau)$ (resp. $e(v,u,\tau)$) as verified and add it to $tspG$ according to Lemma 10. In $\mathcal{G}_t$, edges are stored in a non-descending temporal order, Lines 6-7 iterate through those edges and select an unverified one to start a bidirectional DFS for a temporal simple path. From Line 8 to Line 10, global stacks $\mathcal{S}_v, \mathcal{S}_e$ are maintained throughout the bidirectional DFS. When a temporal simple path is found, we can build it from $\mathcal{S}_e$. The edge confirmation process is illustrated in Lines 11-

19. For each edge in this identified path, we confirm it together with all edges that can replace it to form temporal simple paths, add those edges to $tspG$, and mark them as verified. Note that, we omit the confirmation for the i-th edge in the path when $i = 1, 2, l(p^*) - 1, l(p^*)$ since it clearly satisfies Lemma 2 or 10 and has been verified in Line 4.

Optimized Bidirectional DFS. To further speed up the path identification process, we propose two optimization methods utilized in our implementation of bidirectional DFS. With the above optimization methods, our implementation of the bidirectional DFS is detailed in Algorithm 7.

i) **Prioritization of Search Direction**. When an unverified edge $e(u, v, \tau)$ is selected, the bidirectional DFS for identifying a path through $e(u, v, \tau)$ includes a forward search for a path $p^*_{[\tau,\tau_e]}(u, t)$ and a backward search for a path $p^*_{[\tau_b,\tau]}(s, v)$, which we refer to as the forward path and the backward path, respectively. Assume $\tau - \tau_b > \tau_e - \tau$, based on Remark 1, the backward path is potentially longer than the forward path. If we perform the backward search prior to the forward search, we may not find a valid forward path since the vertices have been possessed by the longer backward path, and the DFS will need to backtrack extensively due to the failure to form a temporal simple path. Following this idea, if $\tau - \tau_b > \tau_e - \tau$, we perform the forward search followed by the backward search; otherwise, the reverse.

ii) **Neighbor Exploration Order**. Intuitively, a shorter temporal path has a lower probability of revisiting a vertex than a longer one. To guide the DFS in favor of shorter paths, at each vertex, we traverse its neighbors following a temporal order. Specifically, in the forward search, an out-neighbor with a larger timestamp is explored ahead of an out-neighbor with a smaller timestamp, that is, in a non-ascending temporal order; while in the backward search, we explore in-neighbors in a non-descending temporal order.

BiDirSearch. In Algorithm 7, we determine whether to perform a forward or a backward search first in Line 2, then start the search in Line 4 with two flags F (resp. B), indicating whether a forward path (resp. backward path) has been found in the current search, initially set to $false$. The *search* procedure recursively searches for a temporal simple path in its given direction. Once a forward (resp. backward) path is completed, it sets the corresponding flag F (resp. B) and toggles the search direction in Lines 10-11. The *search* terminates upon finding paths in both directions in Line 12 and continues in the other direction if the path in one direction is completed. In Lines 16 and 21, the neighbor exploration order is determined based on the current search direction.

***Theorem* 5** *The time complexity of Algorithm 6 is $O(m \cdot d_t^{\theta-1})$ where d_t is the largest degree of the vertices in $\mathcal{G}_t$, i.e., $d_t = \max_{u \in \mathcal{V}(\mathcal{G}_t)}\{\max(|N_{in}(u, \mathcal{G}_t)|, |N_{out}(u, \mathcal{G}_t)|)\}$ and θ is the span of $[\tau_b, \tau_e]$, i.e., $\theta = \tau_e - \tau_b + 1$. The space complexity of Algorithm 6 is $O(n + m)$.*

Algorithm 7: BiDirSearch

```
Input : an edge $e(u, v, \tau) \in \mathcal{E}(\mathcal{G}_t)$
Output: the existence of $p^*_{[\tau_b,\tau_e]}(s, t)$ through $e(u, v, \tau)$
1  $\mathcal{S}_v.push(u)$, $\mathcal{S}_v.push(v)$;
2  if $\tau - \tau_b > \tau_e - \tau$ then $dir = f$ else $dir = b$;
3  $F, B \leftarrow false$;
4  if Search $(u, v, \tau, dir, F, B)$ then return true;
5  else return false;
6  Procedure Search $(u, v, \tau, dir, F, B)$ :
7      if $dir = f$ then $\mathcal{S}_v.push(v)$ else $\mathcal{S}_v.push(u)$;
8      $\mathcal{S}_e.push(e(u, v, \tau))$;
9      if $v = t \vee u = s$ then
10         if $v = t$ then $dir = b$, $F \leftarrow true$;
11         if $u = s$ then $dir = f$, $B \leftarrow true$;
12         if $F \wedge B$ then return true;
13         $e(u_0, v_0, \tau_0) \leftarrow$ the first edge in $\mathcal{S}_e$;
14         if Search $(u_0, v_0, \tau_0, dir, F, B)$ then return true;
15     if $dir = f$ then
16         for each $(w_i, \tau_i) \in N_{out}(v, \mathcal{G}_t)$ do
17             if $\tau_i \leq \tau$ then break;
18             if $w_i \in \mathcal{S}_v$ then continue;
19             if Search $(v, w_i, \tau_i, dir, F, B)$ then return
               true;
20     else
21         for each $(w_i, \tau_i) \in N_{in}(u, \mathcal{G}_t)$ do
22             if $\tau_i \geq \tau$ then break;
23             if $w_i \in \mathcal{S}_v$ then continue;
24             if Search $(w_i, u, \tau_i, dir, F, B)$ then return
               true;
25     $\mathcal{S}_v.pop()$, $\mathcal{S}_e.pop()$;
26     return false;
27 end
```

VI. Experiments

A. Experiment Setup

Algorithms. To the best of our knowledge, we are first to investigate the temporal simple path graph generation problem. Therefore, we implement and evaluate the following algorithms in our experiments.

- `EPdtTSG`: the baseline method proposed in Section III-A, which applies the path enumeration method in the projected graph and generates $tspG$. The projected graph is constructed by the `dtTSG` method based on the query information.
- `EPesTSG`: the baseline method proposed in Section III-A, which applies the path enumeration method in the upper-bound graph constructed by the `esTSG` method introduced in [18] and generates $tspG$.
- `EPtgTSG`: the baseline method proposed in Section III-A, which applies the path enumeration method in the upper-bound graph constructed by the `tgTSG` method introduced in [18] and obtains $tspG$.

TABLE II: Statistics of datasets

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{T}\rvert$	d	θ
D1 (email-Eu-core)	1,005	332,334	803	9,782	10
D2 (sx-mathoverflow)	88,581	506,550	2,350	5,931	20
D3 (sx-askubuntu)	159,316	964,437	2,613	8,729	20
D4 (sx-superuser)	194,085	1,443,339	2,773	26,996	20
D5 (wiki-ru)	457,018	2,282,055	4,715	188,103	25
D6 (wiki-de)	519,404	6,729,794	5,599	395,780	25
D7 (wiki-talk)	1,140,149	7,833,140	2,320	264,905	20
D8 (flickr)	2,302,926	33,140,017	196	34,174	10

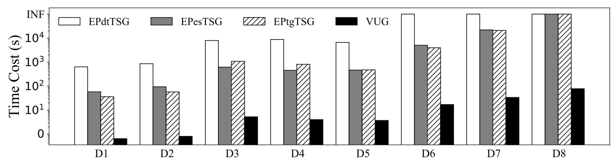


Fig. 5: Response time on all the datasets

- VUG: Algorithm 1 proposed in Section III-B with all optimized strategies.

Datasets. We employ 8 real-world temporal graphs in our experiments. Details of these datasets are summarized in Table II, where d is the maximum degree in a dataset. Among these datasets, D5, D6 and D8 are obtained from KONECT[1], the other five datasets are obtained from SNAP[2].

Parameters and workloads. For each dataset, we specify a default θ for queries on it, as shown in the last column of Table II, and generate 1000 random queries on each dataset with the default θ. That is, a query $(s, t, [\tau_b, \tau_e])$ on a dataset will have $\tau_e - \tau_b + 1 = \theta$. We will not report the result when an algorithm does not terminate within 12 hours.

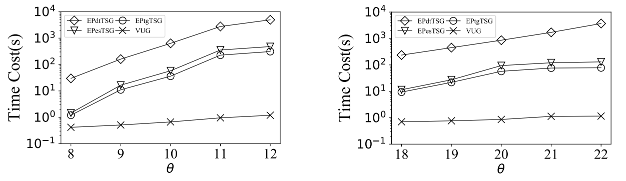
All the programs are implemented in standard C++. All the experiments are performed on a server with Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz and 256G RAM.

B. Performance Comparison with Baselines

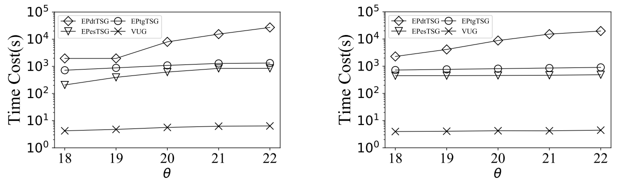
Exp-1: Response time on all the datasets. We report the total response times of VUG for 1000 queries on each dataset under the default settings in Fig. 5, together with those of the three baseline algorithms for comparison. EPesTSG and EPtgTSG are faster than EPdtTSG due to their optimization for tighter upper-bound graphs, i.e., integrating the temporal path constraint. VUG outperforms the baseline algorithms on all the datasets by a significant margin as a result of *i*) it further excludes many unpromising edges by incorporating the simple path constraint when generating the tight upper-bound graph; *ii*) it avoids lots of verification for the existence of edges in the final result. Note that, on the largest dataset D8 with more than 30 million edges, the total response times of three baseline algorithms are represented as *INF*, i.e., they cannot terminate in 12 hours due to the large search space, while VUG only takes 78 seconds to return the result.

[1]http://konect.cc/networks/

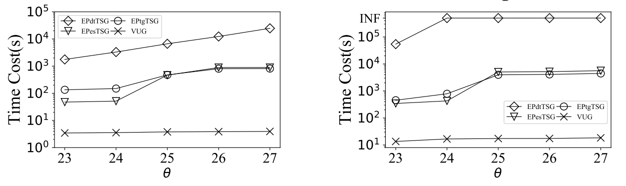
[2]http://snap.stanford.edu



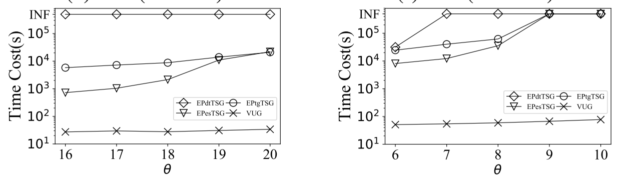
(a) D1 (email-Eu-core) (b) D2 (sx-mathoverflow)



(c) D3 (sx-askubuntu) (d) D4 (sx-superuser)



(e) D5 (wiki-ru) (f) D6 (wiki-de)



(g) D7 (wiki-talk) (h) D8 (flickr)

Fig. 6: Response time by varying parameter θ

Exp-2: Response time by varying parameter θ. In Fig. 6, we report the total response times of VUG for 1000 queries by varying parameter θ on each dataset, along with the three baseline algorithms as the comparison. *INF* indicates the algorithm fails to complete the 1000 queries within 12 hours. We can see that the response times of the three baseline algorithms grow exponentially as θ increases, especially on those dense graphs, while VUG shows response times that grow modestly as θ increases, demonstrating its better scalability. For example, when θ varies from 8 to 12 on D1, the total response time of EPdtTSG, EPesTSG, EPtgTSG increases by a factor of 165, 320, 259, respectively, while that of VUG only increases by a factor of 3.

Exp-3: Space consumption comparison. Fig. 7 reports the space consumption of VUG and the three baseline algorithms. For each algorithm, we report its maximum and minimum space consumption among 1000 queries on each dataset under the default settings. As observed, VUG consistently consumes less space compared to baseline algorithms, attributed to the fact that it avoids enumerating and explicitly storing all temporal simple paths. In addition, for each dataset, VUG maintains a stable space consumption among different queries, regardless of the changing number of temporal simple paths across queries, as opposed to the significant discrepancy between the maximum and minimum space consumption in the baseline algorithms. This supports our theoretical analysis

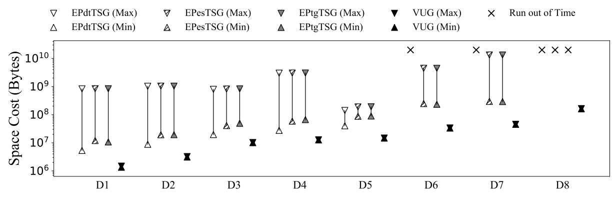


Fig. 7: Space consumption comparison

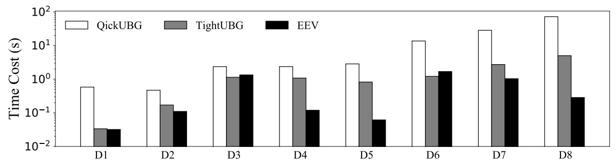


Fig. 8: Response time of each phase in `VUG`

in theorems 1-5, that is, the space complexity of `VUG` is linear in the number of vertices and edges in the dataset.

C. *Performance Evaluation of Each Phase in* `VUG`

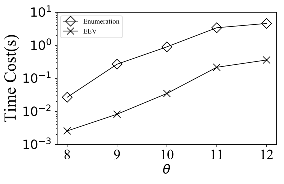
Exp-4: Response time of each phase in `VUG`. In this experiment, we test the total response times of each phase in `VUG` (i.e., `QuickUBG`, `TightUBG` and `EEV`) processing 1000 queries on each dataset. The results are illustrated in Fig. 8. From the results, the response time of `EEV` is the shortest among those three phases in most datasets, confirming that the overhead associated with the bidirectional DFS is limited, despite its exponential time complexity in theory.

Exp-5: Evaluation of upper-bound graph generation. This experiment evaluates the performance of four upper-graph generation methods, i.e., `dtTSG`, `esTSG`, `tgTSG`, and `VUG-UB` (`QuickUBG` and `TightUBG`) on all the datasets. For each algorithm, we calculate the proportion of the resulting $tspG$ within the upper-bound graph produced by the algorithm for 1000 queries on each dataset and report the average upper-bound ratio in Table III. As shown in the results, the upper-bound ratios achieved by `dtTSG` are consistently less than 0.001, demonstrating its impracticality, the upper-bound graphs produced by `esTSG` and `tgTSG` are comparatively tighter. while `VUG` generates the tightest upper-bound graph, achieving an upper-bound ratio exceeding 0.9 on 6 datasets. Besides, `tgTSG` and `QuickUBG` show similar pruning effectiveness since both of them adopt a strict temporal path constraint. However, `QuickUBG` is more efficient as it further optimizes the traversal process. For example on D7, `tgTSG` takes 2.4 hours to complete the queries, while `QuickUBG` only takes 32 seconds.

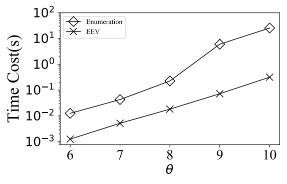
Exp-6: Evaluation of `EEV`. We compare `EEV` with the path enumeration method by applying them separately on the tight upper-bound graph $\mathcal{G}_t$ and generating $tspG$. The total response times for 1000 queries on each dataset are reported in Fig. 9. As observed, `EEV` accelerates the process of generating $tspG$ by at least an order of magnitude compared to the path enumeration method. For example, when θ is 10 on D8, enumerating all paths in $\mathcal{G}_t$ to generate $tspG$ takes 25 seconds, while conducting `EEV` in $\mathcal{G}_t$ only takes 0.3 seconds.

TABLE III: Evaluation of upper-bound graph generation

		D1	D2	D3	D4	D5	D6	D7	D8
`dtTSG`		<0.001	<0.001	<0.001	<0.001	<0.001	-	-	-
`esTSG`		0.104	0.321	<0.001	0.002	0.135	0.098	0.158	-
`tgTSG`		0.594	0.511	0.034	0.046	0.306	0.246	0.284	-
`VUG-UB`	`QuickUBG`	0.594	0.511	0.034	0.046	0.306	0.246	0.284	0.889
	`TightUBG`	0.949	0.984	0.706	0.901	0.972	0.924	0.879	0.988



(a) D1 (email-Eu-core)



(b) D8 (flickr)

Fig. 9: Evaluation of `EEV`

This results from the fact that based on a tight upper-bound graph, `EEV` has the potential to reduce repeated searches as opposed to the path enumeration method, thereby accelerating the generation of $tspG$.

VII. Related Work

Simple Path Enumeration. Simple path enumeration is one of the fundamental tasks in graph analytics. Existing works [28]–[33] address the hop-constrained s–t simple path enumeration problem on directed graphs, which imposes a hop constraint k and aims to find all simple paths from s to t within k hops. Peng et al. [28], [29] propose prune-based algorithms that dynamically maintain hop bounds for each vertex, avoiding the exploration of unpromising branches when the remaining hop budget is smaller than a vertex's hop bound. Sun et al. [30] introduce index-based algorithms to reduce the pruning search space. Yuan et al. [33] focus on batch hop-constrained s–t path enumeration, detecting common computations across queries to boost efficiency. [31] and [32] further extend this problem to hardware devices and distributed environments, respectively.

In addition, Li et al. [34] study labelled hop-constrained s-t simple path enumeration on billion-scale labelled graphs. Rizzi et al. [35] propose length-constrained s-t simple path enumeration on non-negative weighted graphs, where the total weight of each path does not exceed α. Jin et al. [18] explore multi-constrained s-t path enumeration on temporal graphs, with non-decreasing edge timestamps along each path. Mutzel et al. [19] introduce two bicriteria temporal min-cost path problems, focusing on enumerating simple paths that are efficient w.r.t. costs and durations or costs and arrival times.

Still, as discussed previously, simple path enumeration is inefficient for solving simple path graph generation problems.

Path Graph Queries. Path graph captures the structural relationships between vertices by characterizing their connections through reachable paths. Existing works [21]–[23] address path graph queries between two given vertices s and t on unweighted graphs. Liu et al. [21] define the k-hop s-t subgraph query for all paths from s to t within k-hops. Cai et al. [22] extend this problem and further study hop-constraint s-t simple path graphs. Wang et al. [23] define the s-t shortest path graph, which exactly contains all the s-t shortest paths. However, these graphs lack temporal information, and a naive extension of their methods to our problem is inefficient.

Some complex path graph queries involve temporal information [20], [27], [36]. Bhadra et al. [36] compute strongly connected components where any pair of vertices within a component are connected by a temporal path, which might be too rigid and inadequate to identify entity relationships in many real-life situations. Cheng et al. [27] generate associated influence digraph where each edge represents a non-decreasing path. Bast et al. [20] generate a graph that contains a set of optimal connections with corresponding multi-criteria shortest temporal paths for a given query in a large public transportation network. However, the path graphs generated in [27] and [20] are not well-suited for illustrating the underlying connections, as they are not subgraphs of the original graph.

VIII. Conclusion

In this paper, we study the problem of generating temporal simple path graph. Instead of explicitly enumerating all temporal simple paths to obtain the exact result, we propose an efficient method VUG with two main components, i.e., upper-bound graph generation and escaped edges verification. Extensive experiments on 8 real-world temporal graphs show that VUG significantly outperforms all baselines by at least two orders of magnitude.

References

[1] A. Ferreira, "On models and algorithms for dynamic communication networks: The case for evolving graphs," in *In Proc. ALGOTEL*, 2002.

[2] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, "Time-varying graphs and dynamic networks," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 27, no. 5, pp. 387–408, 2012.

[3] P. Holme, "Modern temporal network theory: a colloquium," *The European Physical Journal B*, vol. 88, pp. 1–30, 2015.

[4] P. Holme and J. Saramäki, *Temporal network theory*. Springer, 2019, vol. 2.

[5] M. Latapy, T. Viard, and C. Magnien, "Stream graphs and link streams for the modeling of interactions over time," *Soc. Netw. Anal. Min.*, vol. 8, no. 1, pp. 61:1–61:29, 2018.

[6] O. Michail, "An introduction to temporal graphs: An algorithmic perspective," *Internet Math.*, vol. 12, no. 4, pp. 239–280, 2016.

[7] S. Eubank, H. Guclu, V. Anil Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang, "Modelling disease outbreaks in realistic urban social networks," *Nature*, vol. 429, no. 6988, pp. 180–184, 2004.

[8] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *SIGMOD*, 2015, pp. 967–982.

[9] J. Kasturi, R. Acharya, and M. Ramanathan, "An information theoretic approach for analyzing temporal patterns of gene expression," *Bioinform.*, vol. 19, no. 4, pp. 449–458, 2003.

[10] B. Bui-Xuan, A. Ferreira, and A. Jarry, "Computing shortest, fastest, and foremost journeys in dynamic networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 2, pp. 267–285, 2003.

[11] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proc. VLDB Endow.*, vol. 7, no. 9, pp. 721–732, 2014.

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

[13] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *ICDE*, 2016, pp. 145–156.

[14] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen, "Efficient distributed reachability querying of massive temporal graphs," *VLDB J.*, vol. 28, no. 6, pp. 871–896, 2019.

[15] D. Wen, Y. Huang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "Efficiently answering span-reachability queries in large temporal graphs," in *ICDE*, 2020, pp. 1153–1164.

[16] A. Casteigts, A. Himmel, H. Molter, and P. Zschoche, "Finding temporal paths under waiting time constraints," *Algorithmica*, vol. 83, no. 9, pp. 2754–2802, 2021.

[17] A. Casteigts, T. Corsini, and W. Sarkar, "Simple, strict, proper, happy: A study of reachability in temporal graphs," *Theor. Comput. Sci.*, vol. 991, p. 114434, 2024.

[18] Y. Jin, Z. Chen, and W. Liu, "Enumerating all multi-constrained s-t paths on temporal graph," *Knowl. Inf. Syst.*, vol. 66, no. 2, pp. 1135–1165, 2024.

[19] P. Mutzel and L. Oettershagen, "On the enumeration of bicriteria temporal paths," in *TAMC*, 2019, pp. 518–535.

[20] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger, "Fast routing in very large public transportation networks using transfer patterns," in *ESA*, 2010, pp. 290–301.

[21] Y. Liu, Q. Ge, Y. Pang, and L. Zou, "Hop-constrained subgraph query and summarization on large graphs," in *DASFAA*, 2021, pp. 123–139.

[22] Y. Cai, S. Liu, W. Zheng, and X. Lin, "Towards generating hop-constrained s-t simple path graphs," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 61:1–61:26, 2023.

[23] Y. Wang, Q. Wang, H. Koehler, and Y. Lin, "Query-by-sketch: Scaling shortest path graph queries on very large networks," in *SIGMOD*, 2021, pp. 1946–1958.

[24] D. Kempe, J. M. Kleinberg, and A. Kumar, "Connectivity and inference problems for temporal networks," *J. Comput. Syst. Sci.*, vol. 64, no. 4, pp. 820–842, 2002.

[25] T. M. Yasaka, B. M. Lehrich, and R. Sahyouni, "Peer-to-peer contact tracing: development of a privacy-preserving smartphone app," *JMIR mHealth and uHealth*, vol. 8, no. 4, p. e18936, 2020.

[26] R. Kumar and T. Calders, "Finding simple temporal cycles in an interaction network," in *ECML-PKDD*, 2017, pp. 3–6.

[27] E. Cheng, J. W. Grossman, and M. J. Lipman, "Time-stamped graphs and their associated influence digraphs," *Discret. Appl. Math.*, vol. 128, no. 2-3, pp. 317–335, 2003.

[28] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou, "Hop-constrained s-t simple path enumeration: Towards bridging theory and practice," *Proc. VLDB Endow.*, vol. 13, no. 4, pp. 463–476, 2019.

[29] Y. Peng, X. Lin, Y. Zhang, W. Zhang, L. Qin, and J. Zhou, "Efficient hop-constrained s-t simple path enumeration," *VLDB J.*, vol. 30, no. 5, pp. 799–823, 2021.

[30] S. Sun, Y. Chen, B. He, and B. Hooi, "Pathenum: Towards real-time hop-constrained s-t path enumeration," in *SIGMOD*, 2021, pp. 1758–1770.

[31] Z. Lai, Y. Peng, S. Yang, X. Lin, and W. Zhang, "PEFP: efficient k-hop constrained s-t simple path enumeration on FPGA," in *ICDE*, 2021, pp. 1320–1331.

[32] K. Hao, L. Yuan, and W. Zhang, "Distributed hop-constrained s-t simple path enumeration at billion scale," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 169–182, 2021.

[33] L. Yuan, K. Hao, X. Lin, and W. Zhang, "Batch hop-constrained s-t simple path query processing in large graphs," in *ICDE*, 2024, pp. 2557–2569.

[34] X. Li, K. Hao, Z. Yang, X. Cao, W. Zhang, L. Yuan, and X. Lin, "Hop-constrained s-t simple path enumeration in billion-scale labelled graphs," in *WISE*, 2022, pp. 49–64.

[35] R. Rizzi, G. Sacomoto, and M. Sagot, "Efficiently listing bounded length st-paths," in *IWOCA*, 2014, pp. 318–329.

[36] S. Bhadra and A. Ferreira, "Complexity of connected components in evolving graphs and the computation of multicast trees in dynamic networks," in *ADHOC-NOW*, 2003, pp. 259–270.

Appendix

Proof. **(Proof of Theorem 1)** *Line 2 iterates through each edge in $\mathcal{E}(\mathcal{G})$ once, therefore, the total time complexity is $O(m)$. $\mathcal{G}_q$ takes $O(n+m)$ space, therefore, the total space complexity is $O(n+m)$.*

Proof. **(Proof of Theorem 2)** *Lines 1-2 initialize the $\mathcal{A}(u)$ and $\mathcal{D}(u)$ for each vertex $u \in \mathcal{V}(\mathcal{G})$, which takes $O(n)$ time. Line 6 scans each edge in $\mathcal{E}(\mathcal{G})$ once, which takes $O(m)$ time. Therefore, the total time complexity is $O(n+m)$.*

Throughout the algorithm, we maintain $\mathcal{A}(u)$, $\mathcal{D}(u)$ and pointer in $N_{out}(u,\mathcal{G})$ (resp. $N_{in}(u,\mathcal{G})$) for each vertex $u \in \mathcal{V}(\mathcal{G})$; during the traversal, in Line 9, at most one copy of each vertex is in Q, resulting in a space complexity of $O(n)$.

Proof. **(Proof of Theorem 3)** *Lines 4 initialize $TCV_\tau(s,u)$ for each $\tau \in \mathcal{T}_{in}(u,\mathcal{G}_q)$ of each vertex $u \in \mathcal{V}(\mathcal{G}_q) \setminus \{s,t\}$, which takes $O(m)$ time. Lines 5-6 take $O(n)$ time to initialize the* completed *indicator and the pointer of each vertex $u \in \mathcal{V}(\mathcal{G}_q) \setminus \{s,t\}$. For each edge in $\mathcal{E}(\mathcal{G}_q)$, Lines 8-15 take $O(1)$ time, Lines 17 and 19 take $O(\theta)$ time as the number of vertices in each entry of $TCV_\cdot(s,\cdot)$ is bounded by $\theta-1$. Therefore, the total time complexity is $O(n+\theta\cdot m)$.*

There are $O(m)$ entries in $TCV_\cdot(s,\cdot)$ and $TCV_\cdot(\cdot,s)$, and each entry has a length bounded by $\theta-1$. Throughout the algorithm, we maintain the completed *indicator and the pointer for each vertex $u \in \mathcal{V}(\mathcal{G}_q)$. Therefore, the total space complexity is $O(n+\theta\cdot m)$.*

Proof. **(Proof of Theorem 4)** *The pointer initialization in Line 1 takes $O(n)$ time and the pointer operations in Lines 8-10 take $O(m)$ time in total. Line 3 iterates through each edge in $\mathcal{E}(\mathcal{G}_q)$ once and the intersection operation in Line 17 is performed at most m times. Each intersection operation takes $O(\theta)$ time as the length of each entry in $TCV_\cdot(s,\cdot)$ and $TCV_\cdot(\cdot,s)$ is bounded by $\theta-1$. Therefore, the total time complexity is $O(n+\theta\cdot m)$. $\mathcal{G}_t$ takes $O(n+m)$ space, therefore, the total space complexity is $O(n+m)$.*

Proof. **(Proof of Theorem 5)** *Line 2 initializes a* verified *indicator for each edge, therefore takes $O(m)$ time. Lines 3-5 can be implemented as a traversal in $\mathcal{G}_t$ from s (resp. t), which takes $O(m)$ time. For each unverified edge $e(u,v,\tau)$, the Bidirectional DFS in Line 9 has a depth of $\theta-1$ at most with a time complexity bounded by $O(d'^{\theta-1})$, Lines 11-18 takes $O(d'\cdot\theta)$ time to verify edges in the batch of paths. Therefore the total time complexity is $O(m\cdot d'^{\theta-1})$.*

tspG takes $O(n+m)$ space and the verified indicator for all edges takes $O(m)$ space. The space for stacks $\mathcal{S}_v$ and $\mathcal{S}_e$ is in $O(n)$. Thus, the total space complexity is $O(n+m)$.

Proof. **(Proof of Lemma 3)** ***Sufficiency***. *If there exists a temporal simple path $p^*_{[\tau_b,\tau_e]}(s,t)$ through $e(u,v,\tau)$, there exist two temporal simple paths $p^*_{[\tau_b,\tau_i]}(s,u)$,$p^*_{[\tau_j,\tau_e]}(v,t)$ s.t. $\tau_i < \tau < \tau_j$, and $\mathcal{V}(p^*_{[\tau_b,\tau_i]}(s,u)) \cap \mathcal{V}(p^*_{[\tau_j,\tau_e]}(v,t)) = \emptyset$. Based on the definition of time-stream common vertices, we have $TCV_{\tau_i}(s,u) \subseteq \mathcal{V}(p^*_{[\tau_b,\tau_i]}(s,u))$ and $TCV_{\tau_j}(v,t) \subseteq \mathcal{V}(p^*_{[\tau_j,\tau_e]}(v,t))$, thus, $TCV_{\tau_i}(s,u) \cap TCV_{\tau_j}(v,t) = \emptyset$.*

Necessity. *Consider $e(c,f,4) \in \mathcal{E}(\mathcal{G}_q)$ in Fig. 3(c) as an counterexample. There only exist $\tau_i = 3$ and $\tau_j = 5$ satisfying $\tau_i < 4 < \tau_j$, and we have $TCV_3(s,c) \cap TCV_5(f,t) = \emptyset$ as $TCV_3(s,c) = \{b,c\}$ and $TCV_5(f,t) = \{f\}$. However, there does not exist a temporal simple path $p^*_{[2,7]}(s,t)$ through $e(c,f,4)$. Therefore, the necessity is not established.*

Proof. **(Proof of Lemma 5)** *To proof Lemma 5, we first proof $\mathcal{P}^*_{[\tau_b,\tau_l]}(s,u) = \mathcal{P}^*_{[\tau_b,\tau]}(s,u)$ by contradiction. Since $\tau_l \le \tau$, $\mathcal{P}^*_{[\tau_b,\tau_l]}(s,u) \subseteq \mathcal{P}^*_{[\tau_b,\tau]}(s,u)$. Suppose $\mathcal{P}^*_{[\tau_b,\tau_l]}(s,u) \ne \mathcal{P}^*_{[\tau_b,\tau]}(s,u)$, this implies that there exists $p^*_{[\tau_b,\tau]}(s,u) \in \mathcal{P}^*_{[\tau_b,\tau]}(s,u)$ such that $p^*_{[\tau_b,\tau]}(s,u) \notin \mathcal{P}^*_{[\tau_b,\tau_l]}(s,u)$. It is evident that such $p^*_{[\tau_b,\tau]}(s,u)$ contains an in-coming edge $e(v,u,\tau')$ of u where $\tau_l < \tau' \le \tau$, which contradicts $\tau_l = \max\{\tau_i | \tau_i \in \mathcal{T}_{in}(u,\mathcal{G}_q), \tau_i \le \tau\}$. In conclusion, $\mathcal{P}^*_{[\tau_b,\tau_l]}(s,u) = \mathcal{P}^*_{[\tau_b,\tau]}(s,u)$, and we have $TCV_\tau(s,u) = TCV_{\tau_l}(s,u)$ following the definition of the time-stream common vertices. We omit the proof for $TCV_\tau(u,t) = TCV_{\tau_r}(u,t)$ as it follows a similar approach.*

Proof. **(Proof of Lemma 6)** *For each temporal path $p \in \mathcal{P}_{[\tau_b,\tau]}(s,u)$, there exists a corresponding temporal simple path $p^* \in \mathcal{P}^*_{[\tau_b,\tau]}(s,u)$ such that the path p^* contains all edges in the path p except those forming cycles. Then, we have $\mathcal{V}(p^*) \subseteq \mathcal{V}(p)$ to derive that $\mathcal{V}(p^*) \cap \mathcal{V}(p) = \mathcal{V}(p^*)$. Thus, $TCV_\tau(s,u) = \bigcap_{p^* \in \mathcal{P}^*_{[\tau_b,\tau]}(s,u)\ s.t.\ t \notin \mathcal{V}(p^*)} \mathcal{V}(p^* \setminus s) = \bigcap_{p \in \mathcal{P}_{[\tau_b,\tau]}(s,u)\ s.t.\ t \notin \mathcal{V}(p)} \mathcal{V}(p \setminus s)$. The proof for $TCV_\tau(u,t)$ follows the same approach.*

Proof. **(Proof of Lemma 8)** *If $TCV_{\tau_l}(s,u) \cap TCV_{\tau_r}(v,t) \ne \emptyset$, i.e., there exits w such that $w \in TCV_{\tau_l}(s,u)$ and $w \in TCV_{\tau_r}(v,t)$, then based on the definition of time-stream common vertices, $\forall\ p^* \in \mathcal{P}^*_{[\tau_b,\tau_l]}(s,u)$ s.t. $t \notin \mathcal{V}(p^*)$, $w \in p^*$. Since $\forall\ \tau_b \le \tau_i < \tau_l$, we have $\mathcal{P}^*_{[\tau_b,\tau_i]}(s,u) \subseteq \mathcal{P}^*_{[\tau_b,\tau_l]}(s,u)$, thus, $\forall\ p^*_i \in \mathcal{P}^*_{[\tau_b,\tau_i]}(s,u)$ s.t. $t \notin \mathcal{V}(p^*_i)$, $w \in p^*_i$, that is, $w \in TCV_{\tau_i}(s,u)$. Similarly, $\forall\ \tau_r < \tau_j \le \tau_e$, $w \in TCV_{\tau_j}(v,t)$. Therefore, $TCV_{\tau_i}(s,u) \cap TCV_{\tau_j}(v,t) \ne \emptyset$.*

Proof. **(Proof of Lemma 9)** ($\Rightarrow$) *If an edge $e(u,v,\tau) \in \mathcal{E}(\mathcal{G}_q)$, where $u \ne s$ and $v \ne t$, satisfies condition i), based on Lemma 3 and Lemma 8, such an edge will not be excluded as an unpromising edge. Therefore, it should be included in $\mathcal{G}_t$. If an edge $e(u,v,\tau) \in \mathcal{E}(\mathcal{G}_q)$, where $u = s$ or $v = t$, based on the condition ii) of Lemma 2, we can conclude that $e(u,v,\tau)$ belongs to tspG. Since $\mathcal{G}_t$ is the upper-bound graph of tspG, $e(u,v,\tau)$ must belong to $\mathcal{G}_t$.*

($\Leftarrow$) *If an edge $e(u,v,\tau)$ belongs to $\mathcal{G}_t$, there does not exist a vertex w appearing in all $p^*_{[\tau_b,\tau_i]}(s,u)$ and $p^*_{[\tau_j,\tau_e]}(v,t)$ where $\tau_i < \tau < \tau_j$. Based on the Definiton 5, it is easy to derive that $e(u,v,\tau)$ satisfies condition i) s.t. $u \ne s$ and $v \ne t$. Since $e(u,v,\tau) \in \mathcal{G}_t$ must belong to $\mathcal{G}_q$, and we have discussed $w \ne s$ and $w \ne t$ for each edge in $\mathcal{G}_q$, therefore u can be the source vertex s or v can be the target vertex t.*

Proof. **(Proof of Lemma 10)** *Let $\tau_l = \max\{\tau'' | \tau'' \in \mathcal{T}_{in}(u, \mathcal{G}_q) \wedge \tau_b \le \tau'' < \tau\}$, $\tau_r = \min\{\tau'' | \tau'' \in \mathcal{T}_{out}(v,\mathcal{G}_q) \wedge \tau < \tau'' \le \tau_e\}$. For condition i), if there exists an edge $e(s,u,\tau') \in \mathcal{E}(\mathcal{G}_t)$ such that $\tau_b \le \tau' \le \tau_l < \tau$, then we have a temporal simple* *path $p^*_{[\tau_b,\tau_l]}(s,u) = \langle e(s,u,\tau')\rangle$, therefore $TCV_{\tau_l}(s,u) = \{u\}$. Since $e(u,v,\tau) \in \mathcal{G}_t$, according to Lemma 9, $TCV_{\tau_l}(s,u) \cap TCV_{\tau_r}(v,t) = \emptyset$, that is, $u \notin TCV_{\tau_r}(v,t)$. Then, based on the definition of $TCV_{\tau_r}(v,t)$, there exists a temporal simple path $p^*_{[\tau_r,\tau_e]}(v,t)$ that does not pass through s and u. Therefore, a temporal simple path $p^*_{[\tau_b,\tau_e]}(s,t)$ that includes $e(u,v,\tau)$ can be formed. We omit the proof for condition ii) as it is similar to that of condition i).*