A Contraction Framework for Answering Reachability Queries on Temporal Bipartite Graphs

Lijun Sun[1,2], Junlong Liao[1], Ting Deng[1], Ping Lu[1], Richong Zhang[1,2], Jianxin Li[1,2], Xiangping Huang[3], Zhongyi Liu[3]
[1]Beihang University [2]Zhongguancun Laboratory [3]TravelSky Technology Limited
{sunlijun1,liaojunlong,dengting,luping,zhangrichong,lijx}@buaa.edu.cn,{xphuang,liuzy}@travelsky.com.cn

ABSTRACT

Temporal bipartite graphs model interactions between two distinct entity types over time, and can be used to control disease, manage supply chain, optimize transportation systems, analyze cyber traffic and detect fake news. Existing algorithms rely on index construction to answer such problem, but often struggle with issues like size inflation or inefficient parallel processing. To address these, we propose a contraction-based indexing framework CBSR. This framework first transforms a given temporal bipartite graph into a directed acyclic graph (DAG) preserving the reachability relation between vertices, and then partitions the DAG into multiple fragments to reduce used space and dependency among vertices. Finally, it constructs indices for these fragments using a contraction strategy. Using real-life data, we experimentally show that CBSR outperforms the state-of-the-art algorithms by on average 1245.3× in constructing indices, and 2× in querying.

1 INTRODUCTION

Bipartite graphs (bigraphs) are specialized graphs in which the vertices are divided into two disjoint sets, commonly referred to as top and bottom vertices. Edges are permitted only between vertices from different sets. This structure naturally represents interactions between entities of distinct types, and can be extended with temporal information to capture the occurrence time of these interactions.

Various reachability problems have been extensively studied on temporal graphs, including temporal reachability [10, 12, 45], restless temporal reachability [11, 16, 46], shortest path distance [30, 55, 56] and span-reachability query [12, 53, 57, 64]. In this work, we focus on the span-reachability query (*a.k.a.* temporal reachability query in [64]), which determines whether there exists a temporal path between two vertices within a given time interval. This fundamental problem has critical applications in diverse domains, including disease prevention and contact tracing [12, 17, 22, 29, 65], stock market supervision [50], recommendation systems [38], transport system optimization [24, 26] and fake news detection [37, 44].

For example, (1) in contact tracing, bigraphs can model the movements of individuals among different locations, and span-reachability analysis can be used to identify potential infectors (see Example 1). (2) In stock trading networks, bigraphs can model trades between investors and stocks [50], and reachability analysis can help detecting price ramping [5]. In such cases, accounts controlled by the same trader purchase shares of the same broker within a short time to intentionally inflates its price [4, 34]. Span-reachability can reveal these manipulations by showing that one account can reach another within a short time interval. (3) In recommendation systems, bigraphs can represent ratings between customers and products [38], and reachability analysis can help identify customers who share similar temporal interests or habits [25, 39].

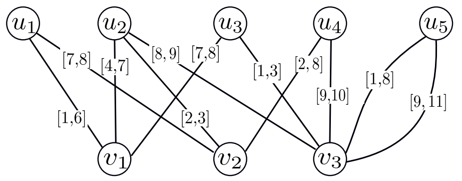

Figure 1: A temporal bipartite graph G

However, it is hard to correctly character the span-reachability in temporal bigraphs. Consider the following example.

Example 1: Figure 1 illustrates a temporal bigraph, which models a disease contact tracing scenario. The top vertices u_1-u_5 correspond to five individuals, and the bottom vertices v_1-v_3 represent three locations. Each edge is labeled with a time interval $[t_1, t_2]$, indicating the period during which an individual is present at a location.

Consider a disease transmission scenario where u_2 becomes infected at timestamp 2. The infection can propagate along the following temporal contact chain: (1) During interval [2,3], both u_2 and u_4 are present at location v_2, allowing potential transmission from u_2 to u_4. (2) Subsequently, during the interval [7,8], u_4 and u_1 are present at location v_2, enabling transmission from u_4 to u_1. This indicates a valid transmission path: $u_2 \rightarrow u_4 \rightarrow u_1$.

However, existing algorithms such as TBP [12] fail to identify this transmission chain due to their coarse-grained definition of "contact". They define contact between two individuals as the union of all their temporal interactions, which results in the contacts between u_2 and u_4, and between u_4 and u_1 both in interval $[2, 8]$. Under the time constraint for a temporal path given in [12] (*i.e.,* the end time of the former contact is no larger than the start time of the latter contact), these two contacts cannot form a valid transmission chain. This coarse-grained temporal modeling produces query results that deviate from real-world transmission scenarios. □

The example above illustrates that existing methods for span-reachability problem suffer from insufficient temporal resolution, which motivates us to propose a new span query based on a refined temporal contact definition. This new definition preserves fine-grained temporal information, enabling more accurate span-reachability analysis in temporal bigraphs.

Contributions. Our contributions are as follows:

(1) Fine-grained definition for span-reachability (Section 3). We propose a fine-grained definition for temporal paths to correctly answer the span-reachability in temporal bigraphs. Unlike conventional temporal paths, we partition the time interval of each

edge into multiple segments, and ensure that the time intervals of these segments increase along the path, where for any pair of two edges in the path leading to the same bottom vertex, the time interval is defined as the intersection of their intervals instead of the union. Based on such paths, we define the span-reachability problem, which allows us to correctly identify temporal path s such as $u_2 \to u_4 \to u_1$ in Example 1 (see Example 3 for details).

(2) Graph transformation (Section 5). We propose a transformation method to convert temporal bigraphs into directed acyclic graphs (DAGs). We introduce the concept of snapshots to compactly capture reachability information, such that the DAG maintains all critical information to answer the span-reachability queries. Moreover, such transformation may also reduce the graph sizes, which in turn reduces the index sizes and accelerates the queries (see Section 8).

(3) Index construction (Section 6). We introduce a contraction-based indexing approach for the constructed DAG. In contrast to existing indices for reachability, we partition the DAG into multiple fragments, and constructs both intra- and inter-fragment indices. In this way, we can reduce the size of the indices, and process large-scale graphs. Moreover, we develop an algorithm to determine the "optimal" number of fragments for the span-reachability queries.

(4) Query algorithm (Section 7). We show that existing algorithms for reachability on DAGs can be adapted with minimum changes to answer the span-reachability on the constructed indices. We develop a query algorithm SRPQ for span-reachability by extending algorithm TopChain of [56] as a case study. We show that SRPQ retains the same complexity of TopChain.

(5) Experimental study (Section 8). Using real-life datasets, we experimentally find the following. (1) Using real-life queries such as contact tracing [17], CBSR correctly answers all these queries due to the fine-grained definition, while TBP achieves only 55.57% accuracy on some dataset, as explained in Example 1. (2) Our graph transformation method can turn bigraphs into DAGs that preserve the reachability relations among vertices, and reduce the graph size to 29.5% of the original graph on average. (3) CBSR is efficient. On average, CBSR is 1245.3× faster than TBP. On a graph with 7M vertices and 129M edges, CBSR constructs the index in less than 600s, while the state-of-the-art algorithm TBP [12] fails to process the graph. (4) On average, SRPQ is 2× faster than the existing algorithm TopChain [56] on reachable queries, and has comparable performance on unreachable queries.

Organization. This paper is organized as follows. We introduce the necessary notations in Section 3 and give an overview of the solution in Section 4. Then we present algorithms for graph transformation, index construction and querying in Sections 5, 6 and 7, respectively. Experimental results are presented in Section 8.

2 RELATED WORK

We categorize the related work as follows.

General graph. The reachability problem in general graphs has been extensively studied. It is to determine whether there is a path between two nodes in a graph. Classical algorithms for this problem include depth-first search (DFS) and breadth-first search (BFS). However, these methods do not scale well to large-scale graphs [33], motivating the development of various indexing methods to accelerate the computation: (1) tree-cover based methods (*e.g.,* path-tree [31], gripp [47] and grail [58]), which construct (a) a tree capturing most reachability information, and (b) a supplementary index table for the remaining information; (2) hop-based methods (*e.g.,* 2-hop [14], 3-hop [32], HHL [3], O'Reach [27], BL [59] and TF-Label [13]), which pick a set of nodes as the landmarks and answer reachability queries using the landmarks as bridges; (3) pruned-based methods (*e.g.,* IP [52], BFL [42], PReaCH [36] and ELF [43]), which compute some conditions on vertices to filter out unreachable queries; and (4) contraction-based methods (*e.g.,* [66]), which convert a graph into a DAG in which each vertex represents a strongly connected component. See [61, 63] for more discussion.

Closer to this work are [28, 62], which support reachability queries on DAGs. MGTag [62] is a graph labeling method that recursively partitions graphs into multiple fragments and computes four-dimensional labels for indexing. TDS [28] further improves MGTag with more reasonable dimension selection.

Discussion. This work differs from prior work as follows. (1) We target the reachability problem on temporal bigraphs, which is more challenging than the reachability problem on general graphs [28, 62, 66]. (2) Using contracted graphs, we can exploit existing algorithms to conduct queries, rather than developing a new one. (3) To construct the inter-fragment indices, we merge vertices connecting to the same set of border nodes and use the trie data structure [23] to reduce border nodes, which is not exploited in [28, 62].

Temporal graphs. The reachability problem has been studied on temporal graphs [41, 51, 53, 54, 56, 64]. ReachGraph transforms temporal graphs into DAGs by materializing connection components at each timestamp [41]. TTL extends hop-based methods to temporal graphs and adopts an index partitioning strategy to handle consecutive time intervals [51]. TopChain also transforms temporal graphs into DAGs, decomposes the DAG into a set of chains, and extends the hop-based methods to construct indices [56]. TVL proposes TVL indices for the reachability queries [64]. TILL extends the 2-hop method with timestamps to answer the span-reachability [53, 54].

Closer to this work are TBP [12], WTB [37] and RQ [45]. TBP focuses on the span-reachability problem (*i.e.,* the single-pair reachability problem) on temporal bigraphs and devises TBP-indices to accelerate query processing [12]. WTB inherits the definition of temporal path from TBP, but its queries are not restricted by time intervals, targeting the global approximate reachability problem [37]. RQ adopts a partition-based strategy to construct indices for reachability queries under budget consraints [45].

Discussion. In contrast to prior work, (1) we adopt a different definition of temporal paths, which can answer the reachability problem more accurate than TBP, WTB and RQ (see Section 3 and **Exp-1** in Section 8). (2) We compute equivalence relations to reduce the size of graphs, and exploit the trie data structure to reduce used memory, which is not studied in TBP, WTB or RQ. (3) We partition the DAG into multiple fragments to process large-scale graphs, and propose an approach to determine the "optimal" number of fragments for span-reachability queries, which is not considered in TBP, WTB or RQ. (4) We develop an algorithm to convert temporal bigraphs into DAGs. Unlike [41, 56], where the resulting DAGs remain similar in size to the temporal graphs,

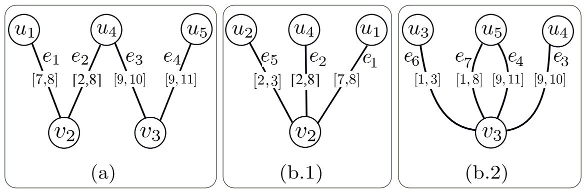

Figure 2: Examples of temporal paths of graph G in Figure 1

our DAGs can be significantly smaller, since a single DAG edge may represent multiple edges in the temporal graphs (*e.g.*, a group of people moving simultaneous from one place to another).

3 PRELIMINARY

Assume a countably infinite set Ω of timestamps with a discrete total order $\leq$. A time interval is a range $[t_1, t_2]$ with $t_1, t_2 \in \Omega$ and $t_1 \leq t_2$. Let $\mathcal{T} \subseteq \Omega\times\Omega$ denote the set of all time intervals.

Temporal bigraph. An undirected temporal bigraph is G=(V, E), where (1) V is a finite set of vertices and partitioned into two disjoint sets U and L, *i.e.,* V=$U \cup L$ and $U \cap L = \emptyset$; vertices in U and L are referred to as top vertices and bottom vertices, respectively; and (2) $E \subseteq U \times L \times \Omega \times \Omega$ is a finite set of temporal edges, where each edge $e = (u, v, t_s, t_e)$ in E connects a top vertex u in U to a bottom vertex v in L during the time interval $[t_s, t_e]$. We also denote the *start time* and *end time* of e by $e.t_s$ and $e.t_e$, respectively.

In this paper, we consider multi-bigraphs, *i.e.,* there may exist multiple edges between a top vertex in U to a bottom vertex in L; for example, a person visit the same location at different time interval.

Overlaps. Given two edges e_1 and e_2, we define the *overlap* of their time intervals, representing, for example, the temporal contact between two individuals. Specifically, two edges e_1=(u_1, v_1, t_s^1, t_e^1) and e_2=(u_2, v_2, t_s^2, t_e^2) in E are said to be overlapped, if $\min(t_e^1, t_e^2) \geq \max(t_s^1, t_s^2)$. The overlap of their time intervals is denoted by $\Theta(e_1, e_2)$ and defined as $\Theta(e_1, e_2) = [\max(t_s^1, t_s^2), \min(t_e^1, t_e^2)]$.

Example 2: Consider the temporal bigraph G in Figure 1, edges e_1=$(u_1, v_2, 7, 8)$ and e_2=$(u_4, v_2, 2, 8)$ are overlapped, and the overlap of their intervals is $[7, 8]$. □

Temporal Path. Given a temporal bigraph $G = (U\cup L, E)$ and two top vertices u and u' in U, a *temporal path* ρ from u to u' of length $2k$ is defined as a sequence of edges $\rho = e_1 e_2 \ldots e_{2k}$ where $e_i = (u_i, v_i, t_s^i, t_e^i)$, which satisfy the following conditions:

(1) $u_1 = u$ and $u_{2k} = u'$;

(2) for each two consecutive edges $e_{2i} = (u_{2i+1}, v_{2i+1}, t_s^{2i+1}, t_e^{2i+1})$ and $e_{2i+2} = (u_{2i+2}, v_{2i+2}, t_s^{2i+2}, t_e^{2i+2})$ of ρ with $i \in [0, k-1]$, they share the same bottom vertex (*i.e.,* $v_{2i+1} = v_{2i+2}$) and their time intervals are overlapped, *i.e.,* $\Theta(e_{2i}, e_{2i+2}) \neq \emptyset$;

(3) for each two consecutive edges e_{2i}=$(u_{2i}, v_{2i}, t_s^{2i}, t_e^{2i})$ and e_{2i+1}= $(u_{2i+1}, v_{2i+1}, t_s^{2i+1}, t_e^{2i+1})$ with $i \in [1, k-1]$, they are incident to the same top vertex (*i.e.,* $u_{2i} = u_{2i+1}$), and the timestamps of e_{2i} and e_{2i+1} increases; more specifically, (a) if $v_{2i} \neq v_{2i+1}$, then $e_{2i}.t_e \leq e_{2i+1}.t_s$ (*i.e.,* top vertex u_{2i} traverses from one bottom vertex v_{2i} to another bottom vertex v_{2i+1}; see e_2 and e_3 in Figure 2(a) for an example); otherwise, (b) either $e_{2i} = e_{2i+1}$ (*i.e.,* top vertex u_{2i} "stays at" the same bottom vertex v_i; see e_2 in Figure 2(b.1)), or $e_{2i}.t_e \leq e_{2i+1}.t_s$ (*i.e.,* top vertex u_{2i} "leaves" and "reenters" the bottom vertex v_{2i}; see e_7 and e_4 in Figure 2(b.2)).

(4) for each i∈$[1, 2k{-}4]$ and any four consecutive edges e_i, $e_{i+1}e_{i+2}$, e_{i+3}, where e_{i+j}=$(u_{i+j}, v_{i+j}, t_s^{i+j}, t_e^{i+j})$ with j∈$[0, 3]$, if they share the same bottom vertex (*i.e.,* v_i=v_{i+1}=v_{i+2}=v_{i+3}), and e_{i+1} and e_{i+2} are the same edge (see Figure 2(b.1)), the end time of the overlap $\Theta(e_i, e_{i+1})$ is no larger than the start time of $\Theta(e_{i+2}, e_{i+3})$.

Intuitively, condition (4) requires that u_i first meets u_{i+1} at "location" v_i during the interval $\Theta(e_i, e_{i+1})$, and then u_{i+1} (*i.e.,* u_{i+2}) meets u_{i+3} at the same "location" v_i during a subsequent interval $\Theta(e_{i+2}, e_{i+3})$. That is, u_{i+1} must meet u_i before meeting u_{i+3}. Without this condition, one could construct a path $e_{i+3}e_{i+2}e_{i+1}e_i$, implying that u_{i+3} meets u_{i+2} (*i.e.,* u_{i+1}) at v_i during $\Theta(e_{i+3}, e_{i+2})$, and only afterwards u_{i+1} (*i.e.,* u_{i+2}) meets u_i at v_i during $\Theta(e_i, e_{i+1})$. This leads to a contradiction because the end time of the overlap $\Theta(e_i, e_{i+1})$ is no larger than the start time of $\Theta(e_{i+2}, e_{i+3})$. That is, condition (4) is not covered by conditions (1)-(3) above.

Example 3: Figure 2 shows three temporal paths ρ_1, ρ_2 and ρ_3 of graph G in Figure 1, each of length 4.

(i) ρ_1=$e_1e_2e_3e_4$ is a temporal path from u_1 to u_5, which satisfies that (a) edges e_1=$(u_1, v_2, 7, 8)$ and e_2=$(u_4, v_2, 2, 8)$ are overlaped, with $\Theta(e_1, e_2)$=$[7, 8]$; (b) edges e_2 and e_3 share the same top vertex u_4, which links to two distinct bottom vertices v_2 and v_3, and $e_2.t_e \leq e_3.t_s$ (*i.e.,* $8 < 9$); and (c) edges e_3=$(u_4, v_3, 9, 10)$ and e_4=$(u_5, v_3, 9, 11)$ are overlaped, with $\Theta(e_3, e_4)$=$[9, 10]$.

(ii) ρ_2=$e_5e_2e_2e_1$ is a temporal path from u_2 to u_1, which satisfies that (a) edges e_5=$(u_2, v_2, 2, 3)$ and e_2=$(u_4, v_2, 2, 8)$ are overlaped, with $\Theta(e_5, e_2)$=$[2, 3]$, and (b) edges e_2 and e_1=$(u_1, v_2, 7, 8)$ are overlaped, with $\Theta(e_2, e_1) = [7, 8]$; and (c) these edges share the same bottom vertex v_2, edge e_2 appears twice in the path, and the end time of $\Theta(e_5, e_2)$ is not larger than the start time of $\Theta(e_2, e_1)$, *i.e.,* $3 \leq 7$.

(iii) ρ_3=$e_6e_7e_4e_3$ is a path from u_3 to u_4, such that (a) edges e_6=$(u_3, v_3, 1, 3)$ and e_7=$(u_5, v_3, 1, 8)$ are overlaped, with $\Theta(e_6, e_7)$=$[1, 3]$; (b) edges e_7 and e_4 share the same top and bottom vertices and $e_7.t_e \leq e_4.t_s$ (*i.e.,* $8 \leq 9$); (c) edges e_4 and e_3 are overlaped, with $\Theta(e_3, e_4) = [9, 10]$; and (d) these edges share the same bottom vertex v_3, u_5 has two different edges to v_3, and the end time of $\Theta(e_6, e_7)$ is not larger than the start time of $\Theta(e_4, e_3)$, *i.e.,* $3 \leq 9$.

Without condition (4), some edge sequences may be mistakenly identified as temporal paths, thus misrepresenting reachability. For example, the edge sequence $e_1e_2e_2e_5$ with edges e_1, e_2 and e_5 given in Figure 2, satisfies conditions (1)-(3), but not condition (4). It implies that u_1 is in contact with u_4 during $[7, 8]$, and then u_4 is in contact with u_2 during $[2, 3]$, which contradicts the required temporal order and cannot be regarded as a valid temporal path. □

The temporal paths are asymmetric, since the timestamps of two consecutive edges in a temporal path increases. Consequently, the existence of a temporal path from u to u' during interval I does not imply the existence of a path from u' to u within the same interval.

Span-Reachability. We say that one top vertex u can reach another top vertex u' in G during time interval I if there exists a temporal path $\rho = e_1e_2 \ldots\ldots e_{2k}$ from u to u' such that the end time of $\Theta(e_1, e_2)$ and the start time of $\Theta(e_{2k-1}, e_{2k})$ fall in the time interval $[t_s, t_e]$. That is, (1) the end time of the overlap $\Theta(e_1, e_2)$ is no less

Table 1: Notations

Notations	Definitions
$G = (U \cup L, E)$	an undirected temporal bigraph
$Q(u, u', G, I)$	the span-reachability query
$\mathcal{G}_G$	the trajectory graph of temporal graph G
$\Theta(e_1, e_2)$	common time interval of e_1 and e_2
$[t_1, t_2]_v$	a snapshot of a bottom vertice v
$\mathcal{S}(G)/\mathcal{S}(u)$	the set of snapshots constructed in G/from vertex u

than the start time of I, and (2) the start time of $\Theta(e_{2k-1}, e_{2k})$ is no larger than the end time of I. We use the end time of the first two edges and the start time of the last two edges to ensure that all edges in ρ are within interval I, which results in more temporal paths between vertices, and enables more precise contact tracing.

Example 4: Consider graph G in Fig. 1. Top vertex u_1 can reach u_5 during interval $I = [8, 9]$ via path $\rho_1 = e_1e_2e_3e_4$ in Example 3, since (1) $\Theta(e_1, e_2) = [7, 8]$ and $\Theta(e_3, e_4) = [9, 10]$, (2) the end time of $\Theta(e_1, e_2)$ is no less than the start time of I (*i.e.,* $8 \geq 8$), and (3) the start time of $\Theta(e_3, e_4)$ is no larger than the end time of I (*i.e.,* $9 \leq 9$). In fact, u_1 can reach u_5 during any interval that contains I. □

Problem Statement. We study *the span-reachability problem*, denoted by SRP, which is stated as follows.

- *Input*: A temporal bigraph G, two top vertices u and u', and a time interval I
- *Question*: Whether u can reach u' in G in time interval I?

Denote by $Q(u, u', G, I)$ the span-reachability query on the temporal bigraph G in interval I, where u and u' refer to the *source vertex* and the *target vertex*, respectively.

Remark. Our definition of temporal paths differs from the one presented in TBP [12] and WTB [37]. More specifically, the time interval of a pair of edges e_1 and e_2 of the same bottom vertex in the path is defined as the overlap of the time intervals of e_1 and e_2, rather than their union as in TBP and WTB. The overlap of e_1 and e_2 is a subinterval of the time intervals of e_1 and e_2, allowing the same edge to appear multiple times in a temporal path. In contrast, under the union-based semantics in TBP and WTB, each edge can appear at most once in a temporal path. Such difference leads to more temporal paths in our SRP setting than in the problem studied in TBP and WTB (see the following example).

Example 5: Consider path $\rho_2 = e_5e_2e_2e_1$ from Example 3.

(1) Path ρ_2 is not a valid path under the definitions of [12, 37]. (a) Subpath e_5e_2, where $e_5 = (u_2, v_2, 2, 3)$ and $e_2 = (u_4, v_2, 2, 8)$, forms a wedge $\mathcal{W}_1 = (e_5, e_2)$ with start time 2 (*i.e.,* $\min(e_5.t_s, e_2.t_s) = 2$) and end time 8 (*i.e.,* $\max(e_5.t_e, e_2.t_e) = 8$). (b) Similarly, path e_2e_1 forms another wedge $\mathcal{W}_2 = (e_2e_1)$ with start time 2 and end time 8, where $e_1 = (u_1, v_2, 7, 8)$. (c) Wedges $\mathcal{W}_1$ and $\mathcal{W}_2$ cannot be concatenated to form a temporal path from u_2 to u_1, since the start time of $\mathcal{W}_2$ (*i.e.,* 2) is earlier than the end time of $\mathcal{W}_1$ (*i.e.,* 8).

(2) However, path ρ_2 is valid in the context of disease tracing. After being infected by u_2 during $[2, 3]$ at location v_2, u_4 remains at v_2, and infects u_1 during $[7, 8]$, though u_2 has already left v_2 at timestamp 3. Under our definition, such indirect propagation is permitted, and thus u_2 can reach u_1 in $[2, 8]$. Moreover, edge e_2 appears twice in path ρ_2, which is not allowed in TBP and WTB. □

Graph Partition. Given a number k and a graph $G = (V, E)$, we can partition G into k fragments $F_1, F_2, \ldots, F_k$ such that (1) $F_i = (V_i, E_i, \mathsf{IN}_i, \mathsf{OUT}_i)$ is a fragment; (2) $V = \cup_{i \in \{1,\ldots,k\}} V_i$; (3) $V_i \cap V_j = \emptyset$ for $i \neq j$; (4) IN_i is the set of vertices $v \in V_i$ with incoming edges from a vertex in $V \setminus V_i$ (5) OUT_i is the set of vertices $v \in V \setminus V_i$ with incoming edges from a vertex in V_i; and (6) E_i consists of all edges between vertices in $V_i \cup \mathsf{IN}_i \cup \mathsf{OUT}_i$. Such partitions are called edge-cut [8], *i.e.,* each vertex in V_i has all its adjacent edges in F_i as in G.

Vertices in $\mathsf{IN}_i \cup \mathsf{OUT}_i$ are called *border vertices.* Vertices in IN_i are contained in V_i, while vertices in OUT_i are not.

A partition is ε-balanced, if $|V_i| \leq (1 + \varepsilon)\frac{|V|}{k}$ for each fragment $F_i = (V_i, E_i, \mathsf{IN}_i, \mathsf{OUT}_i)$, where $\varepsilon \geq 0$ is a threshold for the partitions.

The notations used in the paper are listed in Table 1.

4 SOLUTION OVERVIEW

In this section, we present an overview of a contraction-based framework for the SRP problem, denoted by CBSR (see Figure 3).

Architecture. Given a temporal bigraph G, CBSR first converts it into a DAG, denoted by $\mathcal{G}_G$, It then partitions $\mathcal{G}_G$ into multiple fragments, and constructs both intra-fragment and inter-fragment indices for $\mathcal{G}_G$. These indices are used to efficiently check whether one top vertex can reach another top vertex.

Step 1: Graph transformation. Given a temporal bigraph G, CBSR first constructs a DAG $\mathcal{G}_G$ from G, called the trajectory graph of G, such that an edge in $\mathcal{G}_G$ corresponds to multiple edges in G (Section 5). This transformation reduces the span-reachability on G to the standard reachability problem on $\mathcal{G}_G$. Most existing approaches for the reachability problem convert the graphs into a DAG, where each vertex represents a strongly connected component (*e.g.,* [66]). Different from these approaches, CBSR partitions time intervals on edges to construct the DAG and thereby optimize the reachability analysis.

Step 2: Graph partitioning. To handle large-scale graphs, CBSR partitions $\mathcal{G}_G$ into multiple fragments using a greedy strategy guided by temporal order of time stamps (Section 6.1). The number of fragments is refined via a binary search so that each fragment has a size comparable to the graph built on border nodes constructed in the next step, ensuring that queries after index construction have comparable performance for all vertex pairs no mater whether they are in the same fragment or across distinct fragments.

step 3: Index construction. We construct both intra-fragement and inter-fragment indices to answer span-reachability queries (Section 6.2). For intra-fragment indexing, CBSR leverages existing algorithms (*e.g.,* [27, 31]). For inter-fragment indices, CBSR applies a graph contraction strategy to reduce graphs [20].

Step 3: Answering reachability queries on DAG. We develop a query algorithm SRPQ for SRP. Given two top vertices u and u' in G as the source and target vertices, it first identifies the corresponding vertices $\mathsf{src_u}$ and $\mathsf{tgt_{u'}}$ in $\mathcal{G}_G$, and then checks whether $\mathsf{src_u}$ and $\mathsf{tgt_{u'}}$ belong to the same fragment. If so, intra-fragment indices are used to answer the query; otherwise inter-fragment ones are employed.

Property. (1) After reducing the span-reachability problem on tem-

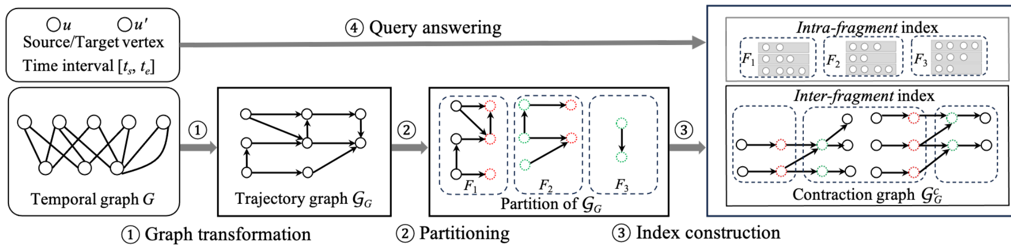

Figure 3: The framework of CBSR. (1) It first constructs a DAG $\mathcal{G}_G$ from temporal bipartite G, to reduce the span-reachability problem on G to the standard reachability problem on $\mathcal{G}_G$; (2) it next partitions $\mathcal{G}_G$ into multiple fragments, to reduce index sizes; (3) it then constructs both intra-fragement indices for each fragment and inter-fragment indices for the graph $\mathcal{G}^c_G$ constructed by border nodes; and finally (4) given a source vertex u and a target vertex u' it checks whether they belong to the same fragment; if so, it uses intra-fragment indices to answer the query; otherwise it uses the inter-fragment.

poral graphs G to the standard reachability problem on DAGs, we can exploit the existing algorithms to conduct the queries. Moreover, the constructed DAGs can be much smaller than the original temporal graph G, which accelerates the computation (see Sections 7).

(2) For large graphs on which existing methods [12, 37] fail to build indices due to memory limits, CBSR can construct both intra-fragment and inter-fragment indices. By partitioning the graph into multiple fragments, CBSR avoids building indices for vertices far from each other, thereby reducing memory usage. Inter-fragment indices are then used to check reachability between such vertices, to strike a balance between query efficiency and space consumption (See Sections 7).

(3) After partitioning the DAGs into multiple fragments, the query algorithm SRPQ uses either intra-fragment or inter-fragment indices, to answer the queries. Since intra-fragment indices are smaller than the ones constructed directly on the whole DAGs, SRPQ get faster when the number of fragments increases (see **Exp-5**). However, such number cannot be too large, otherwise the inter-fragment indices would be larger. Therefore, we develop a strategy to determine the "optimal" number of partitions for SRPQ (Section 6.1).

5 GRAPH TRANSFORMATION

In this section, we transform the temporal bigraph G into its trajectory graph $\mathcal{G}_G$, which is a DAG, preserving the reachability relations between vertices. Here $\mathcal{G}_G$ is constructed based on top vertices, capturing their "movements" across different bottom vertices.

Challenge. To construct a DAG from a temporal bigraph, we face the following challenges: the temporal bigraph is undirected, contain timestamps, and may include cycles (see Figure 1), but how can we derive a DAG from such an undirected temporal graph, while preserving the reachability relationships between the top vertices?

To address these challenges, we proposed the following strategies. (1) We introduce a data structure called snapshots, to capture time intervals during which a top vertex is connected to a bottom vertex. (2) Snapshots are then sequentially ordered by their timestamps, and linked according to the adjacent edges of top vertices, ensuring that the resulting graph is a DAG. Consequently, the reachability between top vertices can be determined by examining paths in this DAG. Intuitively, if a top vertex u can reach another top vertex v in the original temporal graph G, this reachability relation can be confirmed by tracing the u's path through the DAG.

Snapshots. We start with the definition of snapshots.

Definition. Given a temporal bigraph G=$(U \cup L, E)$, let $\mathcal{T}_G$ be the set of timestamps appearing in G. For any two timestamps t_1 and t_2 in $\mathcal{T}_G$, a *snapshot* of a bottom vertex v over the interval $[t_1, t_2]$, denoted by $[t_1, t_2]_v$, indicates that there exist two top vertices in U connected to v throughout $[t_1, t_2]$. Intuitively, $[t_1, t_2]_v$ is a snapshot if there exist two edges e_1 and e_2 both incident to v such that its time interval $\Theta(e_1, e_2)$ contains $[t_1, t_2]$. We call u and u' the top vertices of snapshot $[t_1, t_2]_v$.

Construction. We can construct snapshots from each pair of edges incident to a bottom vertex. However, this method may introduce redundant edges in the resulting DAG, since multiple top vertices may be simultaneously connected to the same bottom vertex.

To address this, we construct snapshots based on the sorted timestamps of edges incident to each bottom vertex v. Let t_1<t_2< $\ldots$ <t_n be the sorted timestamps of v. For each consecutive pair t_i and t_{i+1}, we generate *candidate* snapshots $[t_i, t_i]_v$, $[t_i{+}1, t_{i+1}{-}1]$ and $[t_{i+1}, t_{i+1}]$ when $t_{i+1}{-}t_i{>}1$, and $[t_i, t_i]_v$ and $[t_{i+1}, t_{i+1}]$ when $t_{i+1}{=}t_i{+}1$. We next remove useless snapshots $[t', t'']_v$ during which no pair of top vertices are connected to v, *i.e.,* no pair of edges both incident to v such that their time interval contains $[t', t'']$.

Denote by $\mathcal{S}(G)$ the set of all snapshots constructed above from G. Computing $\mathcal{S}(G)$ takes $O(|E| \log d_{\max})$ time, where $d_{\max}$ is the maximum degree in G, since for each bottom vertex v, it takes $O(d_v \log d_{\max})$ time to sort timestamps where d_v is the degree of v.

Example 6: Consider graph G in Figure 1. the sorted timestamps of v_2 are $2 < 3 < 7 < 8$ from its edges e_5, e_2 and e_1 (see Figure 2(b.1)), yielding candidate snapshots $[2, 2]_{v_2}$, $[3, 3]_{v_2}$, $[4, 6]_{v_2}$, $[7, 7]_{v_2}$ and $[8, 8]_{v_2}$. Among e_5, e_2 and e_1, only e_5 and e_2, and e_2 and e_1 are overlaped, with $\Theta(e_5, e_2)$=$[2, 3]$ and $\Theta(e_2, e_1)$=$[7, 8]$. Then we get snapshots $[2, 2]_{v_2}$ and $[3, 3]_{v_2}$ from edges e_5 and e_2, and $[7, 7]_{v_2}$ and $[8, 8]_{v_2}$ from e_2 and e_1. Similarly, we get snapshots $[4, 4]_{v_1}$, $[5, 5]_{v_1}$, $[6, 6]_{v_1}$ and $[7, 7]_{v_1}$ of bottom vertex v_1, and $[1, 1]_{v_3}$, $[2, 2]_{v_3}$, $[3, 3]_{v_3}$, $[8, 8]_{v_3}$, $[9, 9]_{v_3}$ and $[10, 10]_{v_3}$ of bottom vertex v_3.

A snapshot may be constructed from multiple edge pairs, *e.g.,* $[9, 9]_{v_3}$ can be constructed from (a) e_8 and e_4 with e_8=$(u_2, v_3, 8, 9)$ and e_4=$(u_5, v_3, 9, 11)$, and (b) e_8 and e_3 with e_3=$(u_4, v_3, 9, 10)$. □

Moreover, the overlap of two edges can be split into multiple snapshots, to reduce the size of the DAG $\mathcal{G}_G$.

Example 7: Consider a graph G with k top vertices $u'_1, u'_2, \ldots, u'_k$,

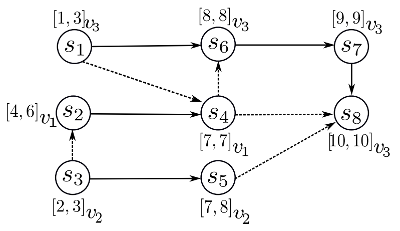

Figure 4: The trajectory graph $\mathcal{G}_G$ of graph G in Figure 1

one bottom vertex v and k edges $e'_j=(u'_j, v, j, k+j)$, $j\in[1,k]$, *i.e.,* each top vertex has an edge connected to the bottom vertex v. We have $\frac{k(k-1)}{2}$ many pairs of edges: e'_i and e'_j with $i < j$ and $\Theta(e'_i, e'_j) = [j, k+i]$. If we construct the DAG using consecutive edges, then when k is large, resulting DAG would be large.

We can construct $2k$ snapshots $[l, l+1]_v$ ($l \in [1, 2k-1]$), and link these snapshots to represent reachability relation among these vertices, leading to smaller DAGs (see below). ☐

Trajectory Graphs. We now define the trajectory graph for a temporal bigraph based on snapshots. For a temporal bigraph $G=(U\cup L, E)$ and snapshot set $\mathcal{S}(G)$, we define the trajectory graph $\mathcal{G}_G=(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}=\mathcal{S}(G)$, *i.e.,* snapshots in $\mathcal{S}(G)$ are vertices in $\mathcal{G}_G$. Each edge in $\mathcal{E}$ connects two snapshots that are constructed from the same edge or two consecutive edges incident to the same top vertex. Intuitively, edges in $\mathcal{G}_G$ capture the movements of top vertices from one bottom vertex to another.

More specifically, for each top vertex u in G, let $\mathcal{S}(u)=\{[t_s^1, t_e^1]_{v_1}, \ldots, [t_s^k, t_e^k]_{v_k}\}$ be the set of snapshots constructed from at least one edge incident to u. We add a directed edge from snapshot $s_i=[t_s^i, t_e^i]_{v_i}$ to $s_j=[t_s^j, t_e^j]_{v_j}$, if one of the following conditions holds: (1) $v_i=v_j$, $t_e^i \leq t_s^j$, and there exists an edge $e=(u, v_i, t_s, t_e)$ in G such that its time interval contains the time intervals of both s_i and s_j (*i.e.,* $t_s \leq t_s^i$ and $t_e^j \leq t_e$); this represent that the top vertex u keeps linking to the bottom vertex v_i, although other vertices leave; or (2) $v_i \neq v_j$, and there exist two edges $e'=(u, v_i, t_s^1, t_e^1)$ and $e'' = (u, v_j, t_s^2, t_e^2)$ both incident to top vertex u such that $t_e^1 = t_e^i$ and $t_s^2 = t_s^j$, that is, the top vertex u moves from bottom vertex v_i to v_j. Intuitively, by the definition of temporal path, such conditions imply that there exists a path of length 4 in G.

<u>*Optimization.*</u> Connecting all snapshots satisfying the above conditions may produce redundant edges in the trajectory graph.

(a) Transitivity. If snapshot s_1 can reach snapshot s_2, which can reach snapshot s_3, the edge from s_1 to s_3 is redundant. To avoid such redundancy, we only add an edge from $s_1=[t_s^i, t_e^i]_{v_i}$ to $s_2=[t_s^j, t_e^j]_{v_j}$, if $s_1, s_2 \in \mathcal{S}(u)$ and there exists no snapshot $s_3 = [t_s^l, t_e^l]_{v_l}$ in $\mathcal{S}(u)$ whose time interval lies between the end time of s_1 and the start time of s_2, *i.e.,* no $s_3 = [t_s^l, t_e^l]_{v_l}$ in $\mathcal{S}(u)$ satisfies $t_e^i \leq t_s^l \leq t_e^l \leq t_s^j$.

(b) Chains. When snapshots $[t_1, t_2]_v, [t_2+1, t_3]_v, \ldots, [t_{n-1}+1, t_n]_v$ form a chain in $\mathcal{G}_G$ such that each intermediate snapshot $[t_i+1, t_{i+1}]_v$ ($i \in [2, n-2]$) has exactly one incoming edge and one outgoing edge, while the first snapshot $[t_1, t_2]_v$ and last snapshot $[t_{n-1}+1, t_n]_v$ have only one outgoing and one incoming edges, respectively, we can merge the chain into a single snapshot $[t_1, t_n]_v$. This merging preserves the reachability relationships in $\mathcal{G}_G$.

Example 8: Figure 4 illustrates the trajectory graph $\mathcal{G}_G$ of the temporal bigraph G in Figure 1, with solid and dashed arrows indicating edges constructed under conditions (1) and (2), respectively. For instance, $\mathcal{S}(u_2)$=$\{[4,6]_{v_1}, [7,7]_{v_1}, [2,3]_{v_2}, [8,8]_{v_3}, [9,9]_{v_3}\}$. A solid edge exists from $s_2 = [4,6]_{v_1}$ to $s_4 = [7,7]_{v_1}$ due to the edge $(u_2, v_1, 4, 7)$, whose interval (*i.e.,* $[4,7]$) contains $[4,6]$ and $[7,7]$; and a dashed edge connects $s_3 = [2,3]_{v_2}$ to $s_2 = [4,6]_{v_1}$ due to the edges $(u_2, v_2, 2, 3)$ and $(u_2, v_1, 4, 7)$ in G.

Redundant edges of types (a) and (b) have been removed from $\mathcal{G}_G$ in Figure 4. For instance, although an edge from s_3 to s_6 could be added based on edges $(u_2, v_2, 2, 3)$ and $(u_2, v_1, 8, 9)$, we omit this edge since s_3 can already reach s_6 via path $s_3s_2s_4s_6$. Furthermore, s_1=$[1,3]_{v_3}$ results from merging $[1,1]_{v_3}$, $[2,2]_{v_3}$ and $[3,3]_{v_3}$, s_2=$[4,6]_{v_1}$ results from merging $[4,4]_{v_1}$, $[5,5]_{v_1}$ and $[6,6]_{v_1}$, and s_3=$[2,3]_{v_2}$ results from merging $[2,2]_{v_2}$ and $[3,3]_{v_2}$. ☐

Properties. We next present properties of trajectory graphs.

Theorem 5.1. *Given a bigraph $G=(U\cup L, E)$, the trajectory graph $\mathcal{G}_G=(\mathcal{V}, \mathcal{E})$ satisfies $|\mathcal{V}|\leq 4|G|$ and $|\mathcal{E}|\leq 4d_{\max}|G|$, where $d_{\max}$ is the maximum degree of vertices in G.*

Proof: (1) We first show that $|\mathcal{V}|\leq 4|G|$. Note that for each bottom vertex v, there are at most $4d_v$ many snapshots where d_v is the degree of v. This is because each edge incident to v contributes at most two distinct timestanps, resulting in at most $2d_v$ timestamps, and for each timestamp, there can be at most two snapshots. Therefore, the total number of vertices in $\mathcal{V}$ is bounded by $\Sigma_{v\in L}4d_v \leq 4|G|$.

(2) We next show that $|\mathcal{E}|\leq 4d_{\max}|G|$. Observe that (a) there exist at most $O(2|G|)$ many snapshots; and (b) each snapshot $[t_1, t_2]_v$ has at most $O(|d_v|)$ many edges, one for each edge of the bottom vertex v, since these edges link to snapshots constructed from (a) the same edges in G as s denoting that a top vertex remains linking to v, or (b) the edges whose end time is t_2 denoting that a top vertex leaving v. Moreover, we assume that each top vertex is connected to at most one bottom vertex at timestamp t, as usually found in disease prevention [35, 49, 60]. Then a top vertex link to only one bottom vertex after a timestamp, and the number of edges in case (b) is bound by $O(d_v)$. Therefore, the number of edges for each snapshot $[t_1, t_2]_v$ is bounded by $O(|d_v|)$, and the number of edges in $\mathcal{G}_G$ is bounded by $\Sigma_{v\in L}4d_v^2 \leq 4d_{\max}|G|$. ☐

<u>*Span-reachability to reachability.*</u> We show that the span-reachability problem on a temporal bigraph G can be reduced to the reachability problem on its trajectory graph $\mathcal{G}_G$.

We start with some notations. Assume that snapshots in $\mathcal{S}(G)$ are ordered by their start times. Given time interval $[t_s, t_e]$ and a top vertex u, let src_u be the *first snapshot* in $\mathcal{S}(G)$ *after* timestamp t_s, and tgt_u be the *last snapshot before* timestamp t_e, which are constructed using the timestamps of edges of a top vertex u. Such snapshots are well defined since we assume that each top vertex is connected to at most one bottom vertex at any timestamp (see above).

Theorem 5.2. *Given a temporal bigraph $G=(U\cup L, E)$, top vertices u and v, and interval $[t_s, t_e]$, u can reach v in G in $[t_s, t_e]$ if and only if src_u can reach tgt_v in $\mathcal{G}_G$.*

Proof: We provide a proof sketch; see [2] for details.

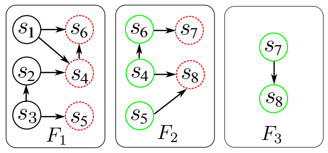

Figure 5: A partition $\{F_1, F_2, F_3\}$ of $\mathcal{G}_G$ given in Figure 4.

($\Rightarrow$) Assume that u can reach u' in G in time interval $[t_s, t_e]$. Then there is a temporal path $\rho = e_1 e_2 \ldots\ldots e_{2k}$, where $e_i{=}(u_i, v_i, t_s^i, t_e^i)$ for each $i \in [1, 2k-1]$, and $u = u_1$ and $u' = u_{2k}$, such that the end time of $\Theta(e_1, e_2)$ and the start time of $\Theta(e_{2k-1}, e_{2k})$ fall in the time interval $[t_s, t_e]$. We can construct a path in $\mathcal{G}_G$ from src_u to $\text{tgt}_{u'}$ by linking snapshots representing each pair of consecutive edges in ρ.

($\Leftarrow$) Assume that src_u reach $\text{tgt}_{u'}$ in $\mathcal{G}_G$. Let $P_{\mathcal{G}} = (s_1{=}[t_s^1, t_e^1]_{v_1}$, $s_2 = [t_s^2, t_e^2]_{v_2}, \ldots, s_n = [t_s^n, t_e^n]_{v_n})$ be a path witnessing the reachability. Starting from the first edge of $P_{\mathcal{G}}$, we construct a path for each edge from s_i to s_{i+1} ($i \in [1, n-1]$), such that they together form a temporal path in G. However, the path may not start with u or end with u'. To fix this, we extend it by adding four consecutive edges that connect u to the begining and u' to the end. □

Complexity. We can verify that $\mathcal{G}_G$ is a DAG. Moreover, $\mathcal{G}_G$ can be constructed in $O(d_{\max}|G| \log |G|)$ time, where $d_{\max}$ is the maximum degree in G, since (a) $\mathcal{G}_G{=}(\mathcal{V}, \mathcal{E})$ has at most $d_{\max}|G|$ vertices and $d_{\max}|G|$ edges; and (b) it takes $O(\log |G|)$ time to sort the edges.

6 INDEX CONSTRUCTION

We present a contraction-based index schema for SRP. We first partition the trajectory graph into fragments (Section 6.1), then construct both intra-fragment and inter-fragment indices (Section 6.2).

6.1 Graph Partitioning

In this section, we partition the trajectory graph into k fragments $F_i{=}(\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i)$ ($i \in [1, k]$). Multiple algorithms have been developed for edge-cut partitions [6, 8, 48]. Leveraging the temporal information in the trajectory graph $\mathcal{G}$ and the order of timestamps, we adopt a greedy strategy to group vertices [18]. More specifically, we compute a topological ordering of the vertices in $\mathcal{G}$. Following this order, each vertex v is assigned to the currenet fragment F_i, if $|\mathcal{V}_i| \leq (1+\varepsilon)\frac{|\mathcal{V}|}{k}$; otherwise, a new fragment F_{i+1} is initialized with v. The procedure proceeds until all vertices are assigned.

Example 9: Figure 5 shows a partition including fragments F_1, F_2 and F_3 of graph $\mathcal{G}_G$ given in Figure 4, with vertex sets $\{s_1, s_2, s_3\}$, $\{s_4, s_5, s_6\}$ and $\{s_7, s_8\}$, respectively. Vertices in IN_i (resp. OUT_i) are marked by green solid (resp. red dashed) circles. □

Remark. (1) Due to the greedy strategy, each fragment has similar number of vertices, resulting in similar index sizes.

(2) Once a path exits a fragment, it cannot reach the fragment again, since the trajectory graph $\mathcal{G}$ is a DAG, and is partitioned following the topological ordering of the vertices in $\mathcal{G}$. Therefore, the graph constructed using border nodes is also a DAG.

The number k of fragments. We partition the trajectory graph into multiple fragments such that intra-fragment indices and inter-fragment indices have the similar size. This ensures that each query has comparable query time, no matter whether its source and target vertices are in the same fragment. More specifically, we partition the trajectory graph such that each fragment $F_i = (V_i, E_i, \mathsf{IN}_i, \mathsf{OUT}_i)$ has similar size as the graph $\mathcal{G}_G^c = (\mathcal{V}^c, \mathcal{E}^c)$ constructed by border nodes (see Section 6.2).

To this end, we adopt a binary search strategy to determine k. We start from $k = 256$, since $\mathcal{G}_G^c$ is usually much larger than F_i when k is large. After partitioning the trajectory graph into k fragments, constructing $\mathcal{G}_G^c$ is a time-consuming process, thus we use the number of border nodes to estimate the size of $\mathcal{G}_G^c$, with a multi factor $\alpha = 8$, and check whether $\mathcal{G}_G^c$ and F_i have similar size, *i.e.,* $||\mathcal{V}^c| - |V_i|| \leq \tau$ for some balance factor τ. If $|\mathcal{V}^c| > |V_i| + \tau$, decrease k and repeat the above step; if $|\mathcal{V}^c| < |V_i| - \tau$, increase k. The process terminates when $||\mathcal{V}^c| - |V_i|| \leq \tau$. We can verify that the process terminates after at most 8 iterations. Since constructing $\mathcal{G}_G^c$ is efficient, on average such binary search strategy takes only 16% of the overall time for indexing on average (see Section 8).

Queries on partitioned trajectory graphs. Given a partitioned graph, the span-reachability queries are divided into two categories:

- the *intra-fragment* reachability queries, where the source vertex and target vertex are in the same fragment; and
- the *inter-fragment* reachability queries, where the source vertex and target vertex are in two different fragments.

Intra-fragment indices. To answer intra-fragment reachability queries, we construct indices for each fragment using existing strategies, such as TopChain [56], pathtree [31], grail [58] and ferrari [40].

In the following, we focus on inter-fragment indices.

6.2 Inter-fragment indexing

We derive inter-fragment indices by constructing a contraction graph from trajectory graph $\mathcal{G}_G$. Intuitively, the contraction graph merges vertices connecting to the same set of border nodes [20].

Equivalent Sets. We first define two equivalence relations on vertices in trajectory graph $\mathcal{G}_G$, which categorize vertices based on their reachability to or from the border vertices.

More specifically, given a fragment $F_i{=}(\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i)$, we define two equivalentce relations EOut_i and EIn_i on $\mathcal{V}_i$, called *out-equivalence relation* and *in-equivalence relation*, respectively. For each $v \in \mathcal{V}_i$, (1) its *out-equivalence class* $[v]_{\mathsf{EOut}_i}$ in EOut_i consists of vertices in $\mathcal{V}_i$ that reach exactly the same set of border vertices in OUT_i as v does; and (2) its *in-equivalence class* $[v]_{\mathsf{EIn}_i}$ in EIn_i is the set of all vertices in $\mathcal{V}_i$ that are reachable from exactly the same set of vertices in IN_i as v is. Here, $[v]_{\mathsf{EIn}_i}$ is defined on vertices in IN_i, *i.e.,* vertices in the same fragment as v, which is used to connect different fragments for reachability (see Section 7).

For each $v \in \mathcal{V}_i$, we define $\mathsf{BIn}_i[v]$ (resp. $\mathsf{BOut}_i[v]$) as the set of border vertices that can reach v (resp. can be reached from v). Let BIn_i (resp. BOut_i) be the collections of all $\mathsf{BIn}_i[v]$ (resp. $\mathsf{BOut}_i[v]$).

Example 10: Consider the partition $\{F_1, F_2, F_3\}$ in Fig. 5. (1) In F_1, $\mathsf{OUT}_1{=}\{s_4, s_5, s_6\}$, s_1 and s_2 both reach s_4 and s_6, while s_3 can reach all vertices in OUT_1. Thus $[s_1]_{\mathsf{EOut}_1}{=}[s_2]_{\mathsf{EOut}_1}{=}\{s_1, s_2\}$, and $[s_3]_{\mathsf{EOut}_1}{=}\{s_3\}$. Since $\mathsf{IN}_1{=}\emptyset$, $[s_1]_{\mathsf{EIn}_1}{=}[s_2]_{\mathsf{EIn}_1}{=}[s_3]_{\mathsf{EIn}_1}{=}\{s_1, s_2, s_3\}$. (2) In F_2, $\mathsf{IN}_2{=}\{s_4, s_5, s_6\}$, s_4 and s_5 are only reached by themselves,

Algorithm 1: OutEquivalentSet

```
Input: Fragment $F_i = (\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i)$.
Output: Equivalent sets $[v]_{\mathsf{EOut}_i}$ for all $v \in V_i$.
1  build $B_i$ of border vertices in $\mathsf{OUT}_i$ without outgoing edges;
2  set $\mathsf{Bout}_i[v] = \{v\}$ for all $v \in B_i$;
3  while $B_i \neq \emptyset$ do
4      $v = B_i$.pop();
5      for each $u$ having an edge $e$ leading to $v$ do
6          $\mathsf{Bout}_i[u] := \mathsf{Bout}_i[u] \cup \mathsf{Bout}_i[v]$;
7          remove edge $e$ from $F_i$;
8          if $u$ has no outgoing edge then
9              push $u$ to $B_i$;
10 $\mathsf{EOut}_i := \mathsf{GroupEquiv}(\mathsf{Bout}_i)$;
11 return $\mathsf{EOut}_i$;
```

while s_6 is reached by s_4 and itself. Then $[s_4]_{\mathsf{EIn}_2} = \{s_4\}$, $[s_5]_{\mathsf{EIn}_2} = \{s_5\}$ and $[s_6]_{\mathsf{EIn}_2} = \{s_6\}$. Likewise, $[s_4]_{\mathsf{EOut}_2} = \{s_4\}$, $[s_5]_{\mathsf{EOut}_2}=\{s_5\}$ and $[s_6]_{\mathsf{EOut}_2} = \{s_6\}$. (3) Similarly, $[s_7]_{\mathsf{EIn}_3} = \{s_7\}$, $[s_8]_{\mathsf{EIn}_3}=\{s_8\}$, and $[s_7]_{\mathsf{EOut}_3} = [s_8]_{\mathsf{EOut}_3} = \{s_7, s_8\}$ since OUT_3 is $\emptyset$. □

Construction. We can construct the equivalence relations EOut_i and EIn_i as follows: (1) for each vertex v in $\mathcal{V}_i$, determine both the set of border vertices reachable from v and the subset of vertices in IN_i that can reach v; (2) cluster vertices in $\mathcal{V}_i$ based on these sets, *i.e.,* two vertices are in the same cluster if they can reach or be reached from the same sets of border vertices (see below). However, such algorithm takes $O(|\mathcal{V}_i||\mathcal{E}_i|)$ time and is costly when F_i is large.

To solve this, we develop Algorithm 1, to compute the out-equivalence relation EOut_i by iteratively removing vertices without outgoing edges, which is extended from the topological sorting algorithm [15]. Given a fragment F_i, it first identifies the set B_i of border vertices in OUT_i with no outgoing edges (line 1). Since F_i is a DAG, such vertices always exist. It then initializes $\mathsf{Bout}_i[v]=\{v\}$ for each border vertex $v \in B_i$ (line 2). The iterative process for computing EOut_i works as follows (lines 3-9). For each border vertex $v \in B_i$ (line 3) and each incoming neighbor u of v, the algorithm merges $\mathsf{Bout}_i[u]$ and $\mathsf{Bout}_i[v]$ (line 5-6), removes the edge e from u to v from the fragment F_i (line 7), and checks whether all outgoing edges of u have been processed (line 8). If so, u is added to B_i for further processing (line 9). The process continues until B_i becomes empty. Finally, it groups vertices in $\mathcal{V}_i$ based on the sets Bout_i (line 10); that is, vertices v_j and v_k are in the same out-equivalence class if $\mathsf{Bout}_i[v_j] = \mathsf{Bout}_i[v_k]$.

The equivalence-class EIn_i can be constructed similarly.

Analysis. One can readily verify the correctness of Algorithm 1. It runs in $O(|\mathcal{E}_i||\mathsf{OUT}_i|)$ time, since (1) each edge is accessed only once (lines 5 and 7), which takes $O(|\mathcal{E}_i|)$ time; (2) it computes the union of two sets in $O(|\mathsf{OUT}_i|)$ time (line 6); and (3) it uses a hashmap to group vertices (line 10), giving $O(|\mathcal{V}_i||\mathsf{OUT}_i|)$ time. In practices, $|\mathsf{OUT}_i|$ is much smaller than $|\mathcal{E}_i|$, making Algorithm 1 much faster than direct vertex comparisons.

Contraction graphs. For inter-fragment queries, we construct a contraction graph to encode reachability between fragments.

For a trajectory graph $\mathcal{G}_G$=$(\mathcal{V}, \mathcal{E})$ and its partition $\{F_1, F_2, \ldots,$

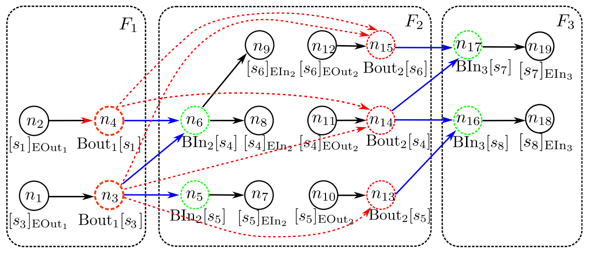

Figure 6: Contraction Graph $\mathcal{G}^c_G$

$F_k\}$, we define its *contraction graph* as $\mathcal{G}^c_G$=$(\mathcal{V}^c, \mathcal{E}^c)$, where the vertex set $\mathcal{V}^c$= $\bigcup_{i\in[1,k]}(\mathsf{EIn}_i \cup \mathsf{EOut}_i \cup \mathsf{BIn}_i \cup \mathsf{BOut}_i)$, *i.e.,* $\mathcal{V}^c$ includes equivalence classes and sets of border vertices, and the edge set $\mathcal{E}^c$ consists of the following three types of edges: for each vertex $v \in \mathcal{V}_i$,

(a) add edges from its out-equivalence class $[v]_{\mathsf{EOut}_i}$ to $\mathsf{Bout}_i[v]$ (*i.e.,* border vertices reachable from v), and from $\mathsf{Bin}_i[v]$ (*i.e.,* border vertices that can reach v in F_i) to the in-equivalence class $[v]_{\mathsf{EIn}_i}$; intuitively, these edges link each equivalence class with border vertices it can reach or be reached from, preserving local reachability information within a fragment;

(b) add an edge from $\mathsf{Bout}_i[v]$ to $\mathsf{Bin}_j[v']$ for $i{\neq}j$, if $\mathsf{Bout}_i[v] \cap \mathsf{Bin}_j[v']{\neq}\emptyset$, *i.e.,* there exists a vertex u in F_j that is also a border vertex in F_i, such that v can reach u in F_i, and u can in turn reach v' in F_j; these edges connect fragments through shared border vertices, enabling cross-fragment reachability; and

(c) add an edge from $\mathsf{Bout}_i[v]$ to $\mathsf{Bout}_j[v']$ for $i{\neq}j$, if v' is in $\mathsf{Bout}_i[v]$; in this case, v can reach v', and subsequently, all vertices in $\mathsf{Bout}_j[v']$; such edges propagate reachability across fragments by chaining border vertices, preserving cross-fragment reachability.

Example 11: We construct contraction graph $\mathcal{G}^c_G$ (see Figure 6) based on the partition in Figure 5, where edges of type (a)-(c) defined above are shown as black, blue, and red dashed edges, respectively. (1) $[s_3]_{\mathsf{EOut}_1}$ can reach $\mathsf{Bout}_1[s_3]$, since s_3 can reach vertices in $\mathsf{Bout}_1[s_3]$, *i.e.,* snapshots s_4, s_5, s_6; (2) $\mathsf{Bin}_2[s_5]$ can reach $[s_5]_{\mathsf{EIn}_2}$, as $\mathsf{Bin}_2[s_5]$ consists of border vertices that can reach s_5; (3) $\mathsf{Bout}_1[s_3]$ can reach $\mathsf{Bin}_2[s_5]$, since $\mathsf{Bout}_1[s_3] \cap \mathsf{Bin}_2[s_5] = \{s_5\}$; (4) $\mathsf{Bout}_1[s_3]$ can reach $\mathsf{Bout}_2[s_5]$, as $s_5 \in \mathsf{Bout}_1[s_3] = \{s_4, s_5, s_6\}$ and $\mathsf{Bout}_2[s_5]$ consists of all vertices that s_5 can reach in F_2. □

Calibration. The contraction graph may contain redundant edges, which degrades query performance. Consider an edge (u, v), where u and v are in fragments F_2 and F_1, respectively, and vertex u appears in m sets $\mathsf{Bout}_1[u_1], \ldots, \mathsf{Bout}_1[u_m]$ within fragment F_1, *i.e.,* u is a border vertex reachable from each u_i. It results in m edges in the contraction graph, from $\mathsf{Bout}_1[u_i]$ ($i{\in}[1, m]$) to $\mathsf{Bout}_2[u]$.

To address this, we use the trie data structure [23] to reduce redundancy. We first assign a unique number to each vertex in $\mathcal{G}^c_G$, sort them accordingly, and then represent the sets $\mathsf{Bout}_1[u_1]$, . . . , $\mathsf{Bout}_1[u_m]$ using a prefix tree built based on the sorted order.

Example 12: Assume that the contraction graph $\mathcal{G}^c_G$ contains sets $\mathsf{BOut}_1[u_1] = \{v_1, v_2, v_3\}$, $\mathsf{BOut}_2[u_2] = \{v_2, v_3, v_4\}$ and $\mathsf{BOut}_1[u_3] = \{v_1, v_2, v_3, v_4\}$. Moreover, $v_1, \ldots, v_4$ have an edge linking to vertex $v_5, \ldots, v_8$, respectively, which exist in another fragment. Observe that v_1, v_2, v_3 and v_4 appear for 2, 3, 3 and 2 times in

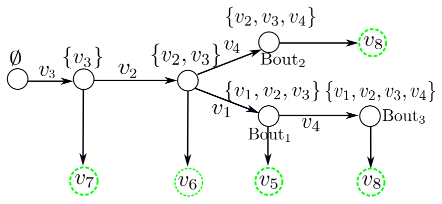

Figure 7: The tries data structure

these equivalence classes, resulting in 10 edges in $\mathcal{G}_G^c$. However, when we store these equivalence classes using a trie as shown in Figure 7, $\mathcal{G}_G^c$ contains only 5 edges to link equivalence classes in other fragments (marked by green dashed circles), which reduces the size of $\mathcal{G}_G^c$ and accelerate the query (see Section 8). □

Analysis. Due to the constructions of equivalence classes, we can verify that given two snapshots s_1 and s_2, s_1 can reach s_2 in $\mathcal{G}_G$ if and only if s_1 can reach s_2 in the contraction graph $\mathcal{G}_G^c$. Indeed, $\text{Bout}_i[v]$ and $\text{Bin}_i[v]$ are the sets of border vertices that can be reach and reach v in $\mathcal{G}_G$, respectively. We can construct $\mathcal{G}_G^c$ in $O(\Sigma_{i\in[1,k]}|\mathcal{V}_i||\text{Out}_i|)$, since each $v \in V_i$ leads to at most four edges.

Remark. To accelerate queries across different fragments, we adopt TopChain [56] to construct the chain cover on $\mathcal{G}_G^c$ [56]. More specifically, (a) we first decompose $\mathcal{G}_G^c$ into chains such that each vertex belongs to exactly one chain. Here, a chain is a sequence of vertices $u_1, \ldots, u_n$ such that u_i has an outgoing edge to u_{i+1} for each $(i \in [1, n-1])$; and (b) we next compute the reachability labels for each vertex in $\mathcal{G}_G^c$, which record the chains that the vertex can reach and can be reached from. Unlike TopChain [56], which applies chain covers directly on temporal graphs, we construct the chain covers on both fragments and contraction graphs, which are smaller than the original graph, leading to better performance (see Section 8).

7 QUERY ON CONTRACTION GRAPHS

We develop algorithm SRPQ for the span-reachability problem using indices presented in Section 6.

Overview. To answer the reachability between two top vertices in partitioned graphs, we must tackle the following challenges: (1) how to convert the reachability problem from a bigraph to the one in contraction graphs preserving the reachability property. (2) How to conduct reachability queries on the contraction graphs, especially when the given two vertices are not in the same fragment.

We solve these problems as follows.

(1) Given two top vertices u and v, and time interval $[t_s, t_e]$, we first identify two snapshots src_u and tgt_v in the contraction graph such that the reachability is preserved, *i.e.,* u can reach v in $[t_s, t_e]$ if and only if src_u can reach dtgt_v (see Section 5). To accelerate the computation, we first group snapshots based on bottom vertices and then sort them based on their time intervals. Then src_u and tgt_v can be identified using a binary search on these sorted snapshots.

(2) We first check whether src_u and tgt_v are in the same fragment. If so, we use intra-fragment indices to check the reachability; recall that each fragment is preprocessed by state-of-the-art indexing strategies, *e.g.,* TopChain [56], pathtree [31], grail [58] and ferrari [40]. Here, we adopt TopChain [56] as a case study. (2) When they are in different fragments, we answer the queries using inter-fragment indices, which is also computed using TopChain [56].

Algorithm 2: SRPQ

```
Input: Trajectory Graph $\mathcal{G}_G = (\mathcal{V}, \mathcal{E})$ and vertices $v_1$ and $v_2$.
Output: Whether $v_1$ can reach $v_2$?
1 if $v_1$ and $v_2$ are in the same fragment $F_i$ then
2     return Query($v_1$, $v_2$, $F_i$);
3 else
4     ($F_1$, $F_2$) := Locate($v_1$, $v_2$, $\mathcal{G}_G$);
5     $v_1'$:=Border$_{\text{out}}$($v_1$, $F_1$, $\mathcal{G}_G$);
6     $v_2'$:=Border$_{\text{in}}$($v_2$, $F_2$, $\mathcal{G}_G$);
7     bReach := CheckReach($v_1'$, $v_2'$);
8     return bReach;
```

Algorithm. We present SRPQ in Algorithm 2. Given vertices src_u and tgt_v, and trajectory graph $\mathcal{G}_G$, it first checks whether src_u and tgt_v are in the same fragment; if so, it exploits the intra-partition indices to check whether src_u can reach tgt_v (line 1); otherwise, it checks whether src_u can reach tgt_v using the chain covers [56]. (1) At first, it identifies fragments F_1 and F_2, in which src_u and tgt_v are located, respectively (line 4). Recall that graph G is partitioned via edge-cut, and each vertex is located in only one fragment. (2) After that it fetches the set $\text{Bout}_1[\text{src}_u]$, *i.e.,* the set of vertices that src_u can reach in F_1 (line 5). This can be done by accessing equivalence class $[\text{src}_u]_{\text{EOut}_1}$. Similarly, it collects $\text{Bin}_2[\text{tgt}_v]$, *i.e.,* the set of vertices that can reach tgt_v in F_2 (line 6). (3) Finally, it checks whether $\text{Bout}_1[\text{src}_u]$ can reach $\text{Bin}_2[\text{tgt}_v]$ via CheckReach using chain covers defined on $\mathcal{G}_G$, *i.e.,* the TopChain algorithm (line 7).

Different from TopChain, SRPQ exploits binary search to identify snapshots src_u and tgt_v in the contraction graph.

Example 13: Given bigraph G in Fig. 1 and the contraction graph in Fig. 5, consider the following two queries.

(1) For query $Q(u_1, u_4, G, [3,7])$, observe that $S_{u_1}^f$=$\{[4,7]_{v_1}\}$=$\{s_3\}$ and $S_{u_4}^l = \{[6,7]_{v_2}\} = \{s_6\}$, which are located in the same fragments, *i.e.,* F_1. The query can be answered by intra-fragment indices.

(2) For query $Q(u_1, u_4, G, [3,9])$, $S_{u_1}^f$=$\{[4,7]_{v_1}\}$=$\{s_3\}$ and $S_{u_4}^l$= $\{[8,8]_{v_3}\}$=$\{s_{11}\}$, which are in different fragments. Note that (a) s_3 belongs to $[s_3]_{\text{EOut}_1}$ and s_{11} belongs to $[s_{11}]_{\text{EIn}_3}$; and (b) $[s_3]_{\text{EOut}_1}$ can reach $[s_{11}]_{\text{EIn}_3}$ in $\mathcal{G}_G^c$. Thus, u_1 can reach u_4 in $[3,9]$. □

Analysis. We next analyze the complexity of SRPQ. Assume that TopChain [56] takes $O(f(|G|))$ time to answer a query. Here, $f(|G|)$ is a polynomial to denote the complexity of TopChain, and is a monotonic function, *i.e.,* if $x < y$, then $f(x) < f(y)$. Indeed, when the graph gets larger, TopChain takes more time to answer a query. Then SRPQ takes $O(\max(f(\max_i |F_i|), f(|\mathcal{G}_G^c|)))$ to answer a query, and is faster than TopChain, since F_i and $\mathcal{G}_G^c$ are much smaller than G, when the number of fragments is large (see Section 8).

Remark. The contraction graphs can also be used to answer both single-source reachability and earliest-arrival reachability.

(1) Single-Source Reachability (SSR) identifies all vertices reachable from a source vertex u within a given time interval on a bigraph. SSR can be answered by performing a Breadth-First Search starting from vertex u using the in-equivalence and out-equivalence classes

Table 2: Dataset Summary

Name	Dataset	$\|E\|$	$\|U\|$	$\|L\|$	Δ
WP	wikiquote-pl	378,979	4,312	49,500	4,750
SO	stack-overflow	1,301,943	545,195	96,678	1,155
LK	linux-kernel	1,565,684	42,045	337,509	12,543
CU	citeulike	2,411,820	153,277	731,769	1,204
BS	bibsonomy	2,555,081	5,794	204,673	7,667
TW	twitter	4,664,606	175,214	530,418	27,742
AM	amazon	5,838,042	2,146,057	1,230,915	3,651
EP	epinion	13,668,321	120,492	755,760	505
LF	lastfm	19,150,869	992	1,084,620	3,149
EJ	edit-jawiki	41,998,341	444,563	2,963,672	5,574
ED	edit-dewiki	129,885,940	1,025,084	5,812,980	5,953

in each fragment and the edges between fragments.

(2) The Earliest-arrival Reachability (EAR) problem is to identify the earliest time at which a source vertex u can reach a target vertex v within a given time interval $[t_s, t_e]$. Since the interval contains $t_e - t_s$ distinct timestamps, EAR can be solved using a binary search strategy, requiring at most $\log_2(t_e - t_s)$ invocation of SPRQ.

8 EXPERIMENTS

Using real-life, we experimentally evaluated CBSR for its (1) correctness, (2) efficiency (3) effectiveness, (4) parameter sensitive and (5) scalability. The source code is publicly available [2].

Experimental setting. We start with the setting.

Datasets. We used eleven real-life graphs, summarized in Table 2. Here, Name denotes dataset abbreviation, $|E|$ is the number of edges, and $|U|$ and $|L|$ are the numbers of top and bottom vertices in the graph, respectively. Δ is the time span(in days) of the graph, defined as the difference between the maximum and minimum timestamps in the graph. All graphs are from the Konect dataset [1]. Since each edge caries only a single timestamp, we generated time intervals by sampling an interval length Δt from a power-law distribution $p(\Delta t) = C\tau^{-\alpha}$ [7], with C ranging from 10 minutes to 8 hours and $\alpha = -2.5$ following [12].

Algorithms. We evaluated the following algorithms.

Index construction algorithms. We implemented (1) CBSR in C++, which comprises graph transformation (Section 5), index construction (Section 6) and query answering (*i.e.,*SPRQ in Section 7). In our comparison with existing indexing algorithms, we focus on the transformation and index construction phases, where the intra-fragment indices are constructed by TopChain [56] as a case study.

We compared with (2) TBP [12], using an implementation from [37]. We also compared CBSR with algorithms for DAGs: (3) TopChain (TC) [56], chain-based indices for temporal DAGs; and (4) PathTree (PT) [31], a path-tree based index for temporal bigraphs.

Note that TopChain was designed for temporal DAGs and can not be applied on bigraphs. Therefore, we first transform bigraphs into DAGs using the approach in Section 5 before applying TopChain.

Query algorithms. We compared query algorithm SRPQ (Section 7) with two kinds of algorithms: (a) query algorithms on bipartite temporal graphs, and (b) query algorithms on DAGs.

(a) For queries on temporal bigraphs, we implemented in C++: (5) SRPQ, our query algorithm from Section 7. We compared with (6) query algorithm of TBP from [12]. Recall that the problem defined in TBP [12] differs from ours (see Example 5). To ensure a fair comparison, we preprocess each edge $e = (u, v, t_s, t_e)$ by splitting it into multiple edges with smaller intervals, such that SRPQ and TBP produce the same answers on the same input. This preprocess aims to applying the intersection definition in TBP and therefore the union semantics will be equal to our interaction semantics.

(b) For queries on DAGs, we compared SRPQ with TopChain [56] and PathTree [31]. Since their original implementations do not support bigraphs, we evaluate them on DAGs obtained via the transformation strategy in Section 5.

Environment. Experiments were conducted on a Linux server with an Intel Xeon Platinum 8358 CPU (2.6GHz, 32 cores, 64 threads) and 1TB memory; each was repeated 5 times and averages are reported.

Experimental results. We next report our findings.

Exp-1: Correctness. To verify the correctness of CBSR, we compared it with TBP in a real-life setting such as contact tracing [17]. The ground truth is obtained by performing a BFS search on real-life datasets with the following constraints: (1) adjacent edges of the same bottom vertex have a common timestamp; and (2) the timestamps increase along the path. Denoted by Init-TBP the variant of TBP that takes the original temporal bigraph as input. We did not report the performance of TopChain and PathTree, since they cannot process temporal graphs.

We evaluated both CBSR and Init-TBP on 100k positive queries. Over all real-life graphs, (1) CBSR correctly answers all these

Table 3: Average Positive and Negative Query Time (μs)

Dataset	Positive queries				Negative queries			
	TC	PT	TBP	SRPQ	TC	PT	TBP	SRPQ
WP(5)	22.94	18.56	**0.21**	17.74	3.59	3.38	**0.52**	3.53
SO(2)	67.14	18.37	**2.03**	30.03	1.07	3.27	3.37	**0.82**
LK(7)	40.34	32.38	**7.97**	12.82	1.27	2.22	19.26	**1.16**
CU(3)	63.40	112.93	OT	**35.18**	**0.20**	7.33	OT	0.32
BS(11)	194.13	82.01	**5.81**	75.97	18.59	22.62	**12.95**	19.81
TW(3)	1,396.67	313.72	OT	**205.17**	3.33	13.21	OT	**0.58**
AM(3)	52.28	8.57	**1.82**	30.75	0.99	1.57	1.86	**0.93**
EP(6)	272.06	OM	OT	**186.46**	2.98	OM	OT	**2.50**
LF(5)	105.36	64.63	**6.14**	64.08	11.00	10.97	11.99	**10.24**
EJ(7)	90.46	204.47	**13.70**	46.16	5.19	13.78	41.70	**4.80**
ED(13)	678.53	OM	OT	**242.34**	14.67	OM	OT	**10.96**

Table 4: Graph Transformation

Dataset	$\|E\|$	$\|U\|$	$\|L\|$	$\|\mathcal{E}\|$	$\|\mathcal{V}\|$
WP	378,979	4,312	49,500	12,746	10,399
SO	1,301,943	545,195	96,678	556,044	520,871
LK	1,565,684	42,045	337,509	506,037	398,311
CU	2,411,820	153,277	731,769	2,262,206	1,381,283
BS	2,555,081	5,794	204,673	111,783	98,884
TW	4,664,606	175,214	530,418	3,445,596	2,684,630
AM	5,838,042	2,146,057	1,230,915	476,363	529,769
EP	13,668,321	120,492	755,760	5,815,905	4,163,505
LF	19,150,869	992	1,084,620	112,935	85,873
EJ	41,998,341	444,563	2,963,672	1,830,929	1,474,038
ED	129,885,940	1,025,084	5,812,980	24,225,743	18,946,499

Table 5: Indexing Time and Memory Usage

Dataset	**Indexing Time (ms)**				**Memory Usage (GB)**			
	TC	PT	TBP	CBSR	TC	PT	TBP	CBSR
WP(5)	**143**	184	78,169	160	0.80	0.81	**0.08**	0.80
SO(2)	**3,528**	40,427	3,673,724	4,899	1.88	9.07	1.32	**0.81**
LK(7)	**1,612**	118,218	530,398	2,789	0.81	19.83	**0.40**	0.81
CU(3)	**6,740**	1,885,985	OT	16,765	**4.10**	212.32	OT	4.57
BS(11)	**979**	2,456	3,094,355	1,081	8.99	0.79	**0.50**	1.44
TW(3)	**19,683**	4,100,630	OT	40,027	**6.67**	597.55	OT	6.77
AM(3)	**3,467**	12,626	7,186,620	5,713	**2.59**	4.90	2.74	2.60
EP(6)	**24,512**	OM	OT	76,409	**9.18**	OM	OT	11.20
LF(5)	10,118	10,964	11,349,943	**6,225**	8.81	4.32	**3.38**	4.32
EJ(7)	**18,687**	2,382,505	35,224,180	26,191	9.68	299.57	**7.12**	10.49
ED(13)	**134,457**	OM	OT	546,569	**50.85**	OM	OT	**50.85**

queries due to the fine-grained definition of the span-reachability (see Section 3). (2) Init-TBP failed to build indices on datasets CU, TW, EP and EJ within 24 hours, and its accuracy ranges from only 55.57% (on SO) to 94.55% (on LF) on the remaining datasets, due to its coarse definition of reachability, as shown in Example 1.

We did not use the BFS algorithm as baseline, as it is much slower than CBSR. Given 100k positive queries on WP, the BFS algorithm takes 90.68ms to finish on average, while CBSR takes only 17.74μs.

Exp-2: Query processing. We then evaluated the performance of the query algorithms. Queries are generated in the form of (u, u', t_s, t_e), where u and u' are top vertices. Starting time t_s is randomly selected from the time intervals associated with edges of the graph, and the interval length $t_e - t_s$ follows the power-law distribution as mentioned above. Specifically, we used SRPQ to generate 100k positive queries and 100k negative queries from the graphs, and report the average query time in Table 3. Here WP(6) indicates that dataset WP was partitioned into 6 fragments, and similarly for other datasets; OT indicates that the algorithm did not finish within 24 hours; and OM denotes an out of memory (OOM) error.

(1) On most datasets, SRPQ consistently outperforms TopChain (TC) on positive queries, with speedups ranging from 1.2× (on WP) to over 4.7× (on TW), and an average of 2.0×. This demonstrates the efficiency of our approach, since it prunes the search space by partitioning. For negative queries, however, SRPQ is slightly worse than TopChain. This is because negative queries often stop early using the rules proposed in [56], whereas SRPQ may require exhaustive checks on a fragment or the contraction graph.

(2) On datasets EP and ED, PathTree (PT) failed to build indices within the memory limit. On CU, TW and EJ, PathTree (PT) is on average 3.1× slower than SRPQ on positive queries. This is because these datasets contain a large number of vertices and edges in their trajectory graphs (see **Exp-3** below), which lead to search space inflation [31]. For negative queries, SRPQ is on average 6.6× faster than PT for the same reason.

(3) Algorithm TBP is faster than SRPQ, since it constructs the minimal TBP-Index to accelerate query processing. However, constructing such indices are time-consuming then TBP cannot process large-scale graphs. For example, on large-scale graphs such as CU, TW, EP and ED, TBP fails to construct the TBP-Index within 24 hours. But SRPQ is the fastest on all these datasets for positive queries.

Exp-3: Index construction. We next evaluated the indexing time and memory usage of CBSR.

Table 6: The Running Time of CBSR (s)

Dataset	Trans	BiSearch	Part	EqSet	Inner	Inter
WP(5)	**0.124**	0.013	0.005	0.003	0.012	0.000
SO(2)	0.773	**1.592**	0.596	1.000	0.792	0.143
LK(7)	0.666	**0.700**	0.325	0.536	0.494	0.066
CU(3)	2.066	3.735	1.697	**6.213**	2.520	0.532
BS(11)	**0.787**	0.102	0.048	0.038	0.098	0.005
TW(3)	3.944	8.000	3.815	**17.923**	5.561	0.781
AM(3)	**2.037**	1.566	0.643	0.756	0.616	0.093
EP(6)	8.414	12.958	6.396	**37.901**	8.205	2.532
LF(5)	**5.851**	0.114	0.062	0.068	0.102	0.026
EJ(7)	**13.856**	3.896	1.713	4.194	2.217	0.312
ED(13)	61.331	56.726	31.281	**358.859**	35.510	2.859

Size of the trajectory graph. We first evaluated the sizes of trajectory graphs transformed from the original temporal bigraphs. The results are shown in Table 4, where $|E|$ is the number of edges in the temporal bigraphs, and $|\mathcal{E}|$ and $|\mathcal{V}|$ denote numbers of edges and vertices in the trajectory graph, respectively. We found that $|\mathcal{E}|$ is consistently smaller than $|E|$ across all datasets. The sizes $|\mathcal{E}|$ of edges in trajectory graphs range from $0.59\%|E|$ (*e.g.,* LF) to $93.8\%|E|$ (*e.g.,* CU), demonstrating the effectiveness of snapshots and trajectory graphs in reducing graph sizes.

Indexing time. We next evaluated the indexing algorithms. As shown in Table 5, (1) CBSR outperforms TBP in index construction, achieving a speedup ranging from 190.2× (*e.g.,* LK) to 2,862.5× faster (*e.g.,* BS), and on average 1245.3× faster, showing the advantage of the contraction-based method. Moreover, TBP cannot handle fine-grained reachability queries as CBSR. Note that to ensure that TBP returns correct answers, we split edges, which degrades its performance on the index construction. (2) CBSR is slower than TopChain in index construction. This is because CBSR incurs extra cost to build inter-fragment indices, which accelerate the reachability analysis (see **Exp-2**).

Memory usage. As shown in Table 5, (1) compared with PathTree, CBSR uses much less memory on most datasets, as PathTree computes additional reachability labels to compensate lost edges, which can grow dramatically on large-scale graphs. (2) CBSR incurs slightly higher memory overhead than TBP, since TBP constructs the minimal TBP-Index. However, due to this reason TBP cannot process large-scale graphs like CU, TW, EP or ED. That is, CBSR strikes a balance between query efficiency and space consumption. (3) CBSR consumes more memory than TopChain, as CBSR takes extra space to store inter-fragment indices.

Cost breakdown. We conducted a cost breakdown analysis on the runtime of CBSR, which is divided into six phases: graph transformation (Trans), binary search for partition number (BiSearch), graph partitioning (Part), equivalent sets construction (EqSet), inner-index construction (Inner), and intra-index construction (Inter). As shown in Table 6, (1) computing equivalent classes dominates the total cost. On CU, TW, EP and ED, step EqSet consumes 37%, 45%, 50% and 66% of the running time of CBSR, respectively. But (2) on WP, BS, AM LF and EJ, Trans consumes 79%, 73%, 36%, 94% and 53% of the running time of CBSR, respectively. This is because these bigraphs are sparse and the number of equivalence classes are small, which accelerates the computation of EqSet.

Exp-4: Parameter sensitivity. We evaluated the impacts of frag-

Table 7: The query time of SRPQ using different fragment numbers (μs)

Dataset	Number of Fragments															
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
CU(3)	33.41	**13.86**	16.28	15.67	15.06	17.41	25.17	31.10	38.19	17.53	27.07	20.73	25.08	32.73	29.42	24.84
TW(3)	561.26	197.82	171.17	147.54	131.82	117.57	128.19	**107.93**	111.41	142.47	129.96	134.53	149.71	129.33	138.75	144.19
EP(6)	167.95	102.32	78.06	73.65	91.30	119.87	110.57	88.20	79.08	**68.28**	235.72	153.74	148.57	194.36	161.99	134.24
EJ(7)	55.52	36.16	37.58	37.45	37.75	**34.81**	35.95	44.03	45.67	38.63	39.34	41.86	38.67	50.25	42.39	46.31
ED(13)	368.02	179.56	172.01	162.27	**155.02**	162.12	169.04	167.48	179.15	179.40	164.23	169.57	170.22	169.41	161.21	168.48

Table 8: The impact of δ on temporal metrics

Dataset	Metric	δ(day)					
		1	**2**	**4**	**8**	**16**	**32**
EJ	IT(ms)	21,269	21,836	22,349	22,984	23,339	24,206
	PQT(μs)	27.33	31.04	31.57	31.19	31.52	33.73
	NQT(μs)	5.03	5.10	5.01	5.03	4.93	5.01
CU	IT(ms)	8,701	9,234	10,107	10,894	11,655	12,726
	PQT(μs)	2.26	3.63	4.97	6.46	9.78	13.81
	NQT(μs)	0.16	0.17	0.17	0.18	0.20	0.18
TW	IT(ms)	26,522	28,784	32,147	37,635	39,270	44,248
	PQT(μs)	50.37	59.45	94.11	117.24	167.94	168.34
	NQT(μs)	2.94	3.02	3.01	3.06	3.18	3.11

Table 9: Scalability

Scale factor	0.20	0.40	0.60	0.80	1.00
IT (s)	45.18	199.75	616.64	1,695.30	4,788.91
MU (GB)	25.60	55.71	92.57	138.38	183.10
QT (μs)	39.00	86.96	191.75	112.81	1068.80

ment numbers and connection latency on the performance of CBSR.

Fragment numbers. We varied the number N_f of fragments from 1 to 16, to evaluate the effectiveness of our binary search strategy that determines the fragment number k. We generated 20k intra-fragment queries and 20k inter-fragment queries on EJ, CU and TW, each with over 1M vertices in its trajectory graph.

The average query times are reported in Table 7. (1) The average query time first decreases and then increases, as expected, since increasing N_f reduces fragment size and enlarge the contraction graph. The optimal performance is achieved when fragment sizes and the size of contraction graph are comparable. (2) On Cu, EJ and ED, the performance of SRPQ using the fragment numbers estimated by our binary search method (shown next to dataset names) are close to the optimal values (marked bold in Table 7), confirming the effectiveness of the proposed method. However, on the other datasets, the predictions of binary search are not close to the optimal values. This is because the size of graphs cannot directly determine the query time, which also depends on many other factors, like topological structure of graphs and the given queries.

Connection latency. We next studied the impact of a parameter, namely latency, on the performance of CBSR. Recall that given two snapshots $s_1 = [t_s^1, t_e^1]_{v_1}$ and $s_2 = [t_s^2, t_e^2]_{v_2}$, we add an edge from s_1 to s_2, if the ending time of s_1 is not larger than the starting time of s_2 (*i.e.,* $t_e^1 \leq t_s^2$) and there exists a top vertex u linked to v_1 and v_2 at time t_e^1 and t_s^2, respectively. However, if t_s^2 is sufficiently larger than t_e^1 (*e.g.,* one year apart), such edge is unlikely to be useful for reachability queries, especially in applications like disease tracing, where the latency of infection is much shorter.

Therefore, we define the latency δ as the maximum time span within which two snapshots can be linked. Formally, s_1 and s_2 cannot have an edge if $|t_e^1 - t_s^2| \geq \delta$ or $|t_e^2 - t_s^1| \geq \delta$.

Table 8 reports the results on indexing time (IT), query time on positive queries (PQT) and negative queries (NQT). (1) Increasing δ slows down CBSR on positive queries, as expected, since increasing δ enlarges the number of edges in trajectory graphs, which leads to a larger search space. In contrast, the negative query time is insensitive to δ, because the rules in TopChain can identify negative queries in the early stage. (2) The impact of δ on CBSR varies across datasets. When δ varies from 32 to 1, CBSR is 1.7× faster on TW, since TW is a user-tag relations dataset, users often use tags for a long time, and increasing δ makes trajectory graphs larger; while on EJ, a wiki-editing dataset, CBSR is only 1.1× faster, as each user (*i.e.,* top vertex) may edit a page within a short time interval.

Exp-5: Scalability. Varying the scale factor from 0.2 to 1, we use the edit-enwiki [1] dataset (with 8.1M top vertices, 42.5M bottom vertices and 572M edges) to evaluate the scalability of CBSR on the indexing time (IT), memory usage (MU) and query time (QT) (see Table 9). (1) When $|G|$ gets larger, CBSR takes longer to construct indices and conduct queries. And CBSR also uses more memory, as expected. (2) CBSR scales well. It takes 4, 788.91s and 183.10GB to construct indices on a graph with 572M edges. And it takes 1068.80μs to process a query on a graph with 50.6M vertices and 572M edges.

Summary. We found the following: (1) Our graph transformation algorithm is effective. On average, the generated DAG is only 29.5% the size of the original graphs. (2) CBSR is efficient. On large-scale graph ED with 7M vertices and 129M edges, it can construct the index in 546.6s while TBP fails to construct the index in 24 hours. And over all datasets, it is on average 1245.3× faster than TBP. (3) CBSR is effective. Using the indices constructed by CBSR, on average SRPQ is 2× faster than TopChain over all datasets.

9 CONCLUSION

This paper proposes algorithms to answer the span-reachability problem on temporal bigraphs. Our contributions are listed as follows: (1) a fine-grained definition for the span-reachability; (2) a graph transformation method that converts the span-reachability problem on temporal bigraphs to the reachability problem on DAGs; (3) a contraction-based indexing schema to process large-scale graphs; and (4) a two-stage query algorithm for the span-reachability queries on temporal bigraphs. Experiments show that the methods are promising in practice.

One topic for future work is to develop effective partition algorithms for trajectory graphs, to reduce border vertices, and strategies to determine the optimal number of fragments.

REFERENCES

[1] The konect project, 2013. *http://konect.cc/*.

[2] Full version, data and code, 2025. *https://github.com/SunboTax/CBSR.git*.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.

[4] C. Alexander and D. Cumming. *Corruption and Fraud in financial markets: Malpractice, Misconduct and Manipulation.* John Wiley & Sons, 2022.

[5] S. Ali. Controlling serious financial crime: A jamaican perspective. *Economic Affairs*, 27(1):14–17, 2007.

[6] K. Andreev and H. Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

[7] A.-L. Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435(7039):207–211, 2005.

[8] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *KDD*, pages 1456–1465, 2014.

[9] J. D. Brunner and N. Chia. Confidence in the dynamic spread of epidemics under biased sampling conditions. *PeerJ*, 8, 2020.

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

[11] A. Casteigts, A. Himmel, H. Molter, and P. Zschoche. Finding temporal paths under waiting time constraints. In *ISAAC*, volume 181, pages 30:1–30:18, 2020.

[12] X. Chen, K. Wang, X. Lin, W. Zhang, L. Qin, and Y. Zhang. Efficiently answering reachability and path queries on temporal bipartite graphs. *Proc. VLDB Endow.*, 14(10):1845–1858, 2021.

[13] J. Cheng, S. Huang, H. Wu, and A. W. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pages 193–204, 2013.

[14] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2022.

[16] A. Deligkas and I. Potapov. Optimizing reachability sets in temporal graphs by delaying. *Inf. Comput.*, 285:104890, 2022.

[17] 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*, 429(6988):180–184, 2004.

[18] W. Fan, R. Jin, P. Lu, C. Tian, and R. Xu. Towards event prediction in temporal graphs. *Proc. VLDB Endow.*, 15(9):1861–1874, 2022.

[19] W. Fan, Y. Li, M. Liu, and C. Lu. A hierarchical contraction scheme for querying big graphs. In *SIGMOD*, pages 1726–1740, 2022.

[20] W. Fan, Y. Li, M. Liu, and C. Lu. Making graphs compact by lossless contraction. *VLDB J.*, 32(1):49–73, 2023.

[21] W. Fan and C. Tian. Incremental graph computations: Doable and undoable. *TODS*, 47(2):6:1–6:44, 2022.

[22] L. Ferreri, E. Venturino, and M. Giacobini. Do diseases spreading on bipartite networks have some evolutionary advantage? In *EvoBio*, volume 6623, pages 141–146, 2011.

[23] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[24] E. Frejinger and M. Hewitt. Perspectives on optimizing transport systems with supply-dependent demand. *INFOR*, 2025. Publisher Copyright: © 2025 Canadian Operational Research Society (CORS).

[25] Z. Guan, L. Wu, H. Zhao, M. He, and J. Fan. Enhancing collaborative semantics of language model-driven recommendations via graph-aware learning. *TKDE*, 37(9):5188–5200, 2025.

[26] V. Hajipour, M. Tavana, D. Di Caprio, M. Akhgar, and Y. Jabbari. An optimization model for traceable closed-loop supply chain networks. *Applied Mathematical Modelling*, 71:673–699, 2019.

[27] K. Hanauer, C. Schulz, and J. Trummer. O'reach: Even faster reachability in large graphs. *JEA*, 27:1–27, 2022.

[28] D. He and P. Yuan. TDS: fast answering reachability queries with hierarchical traversal trees. *Clust. Comput.*, 28(3):178, 2025.

[29] B. H. Hong, J. Labadin, W. K. Tiong, T. Lim, M. H. L. Chung, et al. Modelling covid-19 hotspot using bipartite network approach. *Acta Informatica Pragensia*, 10(2):123–137, 2021.

[30] W. Huo and V. J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, pages 38:1–38:4, 2014.

[31] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS*, 36(1), 2011.

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

[33] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*, pages 595–608, 2008.

[34] S. Khodabandehlou and A. H. Golpayegani. Fifraud: Unsupervised financial fraud detection in dynamic graph streams. *ACM Trans. Knowl. Discov. Data*, 18(5):111:1–111:29, 2024.

[35] H. Lu and S. Uddin. A weighted patient network-based framework for predicting chronic diseases using graph neural networks. *Scientific reports*, 11(1):22607, 2021.

[36] F. Merz and P. Sanders. Preach: A fast lightweight reachability index using pruning and contraction hierarchies. In *ESA*, volume 8737, pages 701–712. Springer, 2014.

[37] L. Meunier and Y. Zhao. Reachability queries on dynamic temporal bipartite graphs. In *SIGSPATIAL/GIS*, 2023.

[38] B. J. Mirza, B. J. Keller, and N. Ramakrishnan. Studying recommendation algorithms by graph analysis. *J. Intell. Inf. Syst.*, 20(2):131–160, 2003.

[39] S. M. Rahimi, B. H. Far, and X. Wang. Behavior-based location recommendation on location-based social networks. *GeoInformatica*, 24(3):477–504, 2020.

[40] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.

[41] H. Shirani-Mehr, F. B. Kashani, and C. Shahabi. Efficient reachability query evaluation in large spatiotemporal contact datasets. *Proc. VLDB Endow.*, 5(9):848–859, 2012.

[42] J. Su, Q. Zhu, H. Wei, and J. X. Yu. Reachability querying: Can it be even faster? *TKDE*, 29(3):683–697, 2016.

[43] Z. Su, D. Wang, X. Zhang, L. Cui, and C. Miao. Efficient reachability query with extreme labeling filter. In *WSDM*, pages 966–975, 2022.

[44] E. Tacchini, G. Ballarin, M. L. D. Vedova, S. Moret, and L. de Alfaro. Some like it hoax: Automated fake news detection in social networks. *CoRR*, abs/1704.07506, 2017.

[45] B. Tesfaye, N. Augsten, M. Pawlik, M. H. Böhlen, and C. S. Jensen. Speeding up reachability queries in public transport networks using graph partitioning. *Inf. Syst. Frontiers*, 24(1):11–29, 2022.

[46] S. Thejaswi, J. Lauri, and A. Gionis. Restless reachability problems in temporal graphs. *arXiv preprint arXiv:2010.08423*, 2020.

[47] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, page 845–856, 2007.

[48] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *WSDM*, pages 333–342, 2014.

[49] D. L. Venkatraman, D. Pulimamidi, H. G. Shukla, and S. R. Hegde. Tumor relevant protein functional interactions identified using bipartite graph analyses. *Scientific Reports*, 11(1):21530, 2021.

[50] K. Wang, M. Cai, X. Chen, X. Lin, W. Zhang, L. Qin, and Y. Zhang. Efficient algorithms for reachability and path queries on temporal bipartite graphs. *VLDB J.*, 33(5):1399–1426, 2024.

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

[52] H. Wei, J. X. Yu, C. Lu, and R. Jin. Reachability querying: An independent permutation labeling approach. *Proc. VLDB Endow.*, 7(12):1191–1202, 2014.

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

[54] D. Wen, B. Yang, Y. Zhang, L. Qin, D. Cheng, and W. Zhang. Span-reachability querying in large temporal graphs. *VLDB J.*, 31(4):629–647, 2022.

[55] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proc. VLDB Endow.*, 7(9):721–732, 2014.

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

[57] H. Xie, Y. Fang, Y. Xia, W. Luo, and C. Ma. On querying connected components in large temporal graphs. *Proc. ACM Manag. Data*, 1(2):170:1–170:27, 2023.

[58] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *Proc. VLDB Endow.*, 3(1-2):276–284, 2010.

[59] C. Yu, T. Ren, W. Li, H. Liu, H. Ma, and Y. Zhao. BL: an efficient index for reachability queries on large graphs. *IEEE Trans. Big Data*, 10(2):108–121, 2024.

[60] J. Yu, X. Zhang, H. Wang, X. Wang, W. Zhang, and Y. Zhang. FPGN: follower prediction framework for infectious disease prevention. *WWW*, 26(6):3795–3814, 2023.

[61] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 181–215. 2010.

[62] P. Yuan, Y. You, S. Zhou, H. Jin, and L. Liu. Providing fast reachability query services with mgtag: A multi-dimensional graph labeling method. *IEEE Trans. Serv. Comput.*, 15(2):1000–1011, 2022.

[63] C. Zhang, A. Bonifati, and M. T. Özsu. An overview of reachability indexes on graphs. In *SIGMOD conference Companion*, SIGMOD '23, page 61–68, 2023.

[64] 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.*, 28(6):871–896, 2019.

[65] R. Zhao and Q. Liu. Dynamical behavior and optimal control of a vector-borne diseases model on bipartite networks. *Applied Mathematical Modelling*, 102:540–563, 2022.

[66] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang. DAG reduction: Fast answering reachability queries. In *SIGMOD*, pages 375–390, 2017.

APPENDIX

A. Proof of Theorem 5.2

($\Rightarrow$) Assume that vertex u can reach v in G in time interval $[t_s, t_e]$. There is a temporal path $\rho = e_1 e_2 \ldots e_{2k-1} e_{2k}$ with $u = u_1$ and $v = u_{k+1}$ such that (1) e_1 and e_{2k} are adjacent edges of u and v, respectively; and (2) the end time of $\Theta(e_1, e_2)$ and the start time of $\Theta(e_{2k-1}, e_{2k})$ fall in the time interval $[t_s, t_e]$.

Given the temporal path ρ, we construct a path from src_u to tgt_v in the traject graph $\mathcal{G}_G$ as follows: for each four consecutive edges $e_1^i e_2^i e_1^{i+1} e_2^{i+1}$, there exist four snapshots constructed from the starting time and ending time of these edges. That is, (a) $s_1 = [t_s^i, t_s^i]_{v_i}$ and $s_2 = [t_e^i, t_e^i]_{v_i}$ from the first two consecutive edges e_1^i and e_2^i, where $\Theta(e_1^i, e_2^i) = [t_s^i, t_e^i]$; (b) $s_3 = [t_s^{i+1}, t_s^{i+1}]_{v_{i+1}}$ and $s_4 = [t_e^{i+1}, t_e^{i+1}]_{v_{i+1}}$ from the last two edges e_1^{i+1} and e_2^{i+1}, where $\Theta(e_1^i, e_2^i) = [t_s^{i+1}, t_e^{i+1}]$; and (c) there is a path from s_2 to s_3 in $\mathcal{G}_G$.

(1) We start with the path from s_1 to s_2. Because both s_1 and s_2 are constructed from the two consecutive edges $e_1^i e_2^i$, the timestamps of s_1 and s_2 are covered by both edges e_1^i and e_2^i. Then there exist a path from s_1 and s_2 in $\mathcal{G}_G$ due to edges e_1^i and e_2^i. Observe that there may exist multiple snapshots constructed from two consecutive edges (see Example 7). Similarly, there exists a path from s_3 to s_4.

(2) We next construct the path from s_2 to s_3. Since snapshots s_2 and s_3 are constructed from two edges of the same top vertex u_{i+1}, we can collect all its adjacent edges $(u_{i+1}, v_1, t_s^{12}, t_e^{12}), (u_{i+1}, v_2, t_s^{22}, t_e^{22}), \ldots, (u_{i+1}, v_k, t_s^{k2}, t_e^{k2})$ during the time interval $[t_e^i, t_s^{i+1}]$. Let $t_s^{12} \leq \ldots \leq t_s^{k2}$. Then there exists a path from snapshot $[t_s^{12}, t_s^{12}]_{v_1}$ to snapshot $[t_s^{k2}, t_s^{k2}]_{v_k}$, to simulate the movements of u_{i+1}. Moreover, there exist an edge from s_2 to $[t_s^{12}, t_s^{12}]_{v_1}$, and an edge from $[t_s^{k2}, t_s^{k2}]_{v_k}$ to s_3, since both of these snapshots are constructed from edges of top vertex u_{i+1} during the time interval $[t_e^i, t_s^{i+1}]$. Therefore, there is a path from s_2 to s_3 in $\mathcal{G}_G$.

Therefore, $([t_s^1, t_e^1]_{v_1}, \ldots, [t_s^n, t_e^n]_{v_n})$ is a path in $\mathcal{G}_G$, with $\mathrm{src}_u = [t_s^1, t_e^1]_{v_1}$ and $\mathrm{tgt}_v = [t_s^n, t_e^n]_{v_n}$.

($\Leftarrow$) Assume that snapshot src_u can reach snapshot tgt_v in $\mathcal{G}_G$. Let $P_{\mathcal{G}} = (s_1, s_2, \ldots, s_{n-1}, s_n)$ be such a path with $s_1 = \mathrm{src}_u$ and $s_n = \mathrm{tgt}_v$ in $\mathcal{G}_G$. For each edge from s_i to s_{i+1} ($i \in [1, n-1]$) in $\mathcal{G}_G$, we construct multiple consecutive edges forming a temporal path in G.

(1) We start with $i = 1$. There exists an edge from $s_1 = [t_s^1, t_e^1]_{v_1}$ to $s_2 = [t_s^2, t_e^2]_{v_2}$. Let (e_s^1, e_e^1) (resp. (e_s^2, e_e^2)) be a pair of edges whose timestamps are used to construct snapshot s_1 (resp. s_2). Assume that $e_s^1 = (u_2, v_1, t_s^{11}, t_e^{11})$, $e_e^1 = (u_3, v_1, t_s^{12}, t_e^{12})$, $e_s^2 = (u_3, v_2, t_s^{21}, t_e^{21})$ and $e_e^2 = (u_4, v_2, t_s^{22}, t_e^{22})$. Observe that (a) the first edge e_1^s may not be an edge of u_1, since u_1 can link to v_1 before timestamp t_s^1 (see the definition of the span-reachability problem); here u_1 is the given source vertex u; (b) the edge from s_1 to s_2 are constructed due to top vertex u_3; (c) the edges e_e^1 and e_s^2 may be the same edge of u_3 (see Cases b.1 and b.2 in Figure 2); and (d) from the definition of $\mathcal{G}_G$, the following holds: $t_e^1 \leq t_s^2$, where $s_1 = [t_s^1, t_e^1]_{v_1}$ and $s_2 = [t_s^2, t_e^2]_{v_2}$.

We construct the temporal path in G for u based on $s_1 = [t_s^1, t_e^1]_{v_1}$ and $s_2 = [t_s^2, t_e^2]_{v_2}$ as follows.

(I) We start with the two consecutive edges from u_2 to u_4. We construct the following path: $u_2 v_1 u_3 v_2 u_4$. This is a valid temporal path, since (a) $t_e^1 \leq t_s^2$, (b) the starting time of the overlap $\Theta(e_s^1, e_e^1)$ is not larger than then starting time of $\Theta(e_s^2, e_e^2)$; and (c) timestamps of consecutive edges e_e^1 and e_s^2 of top vertex u_3 increase.

It remains to construct two consecutive edges between u_1 and u_2. From $s_1 = \mathrm{src}_u$, we know that u_1 and u_2 link to v_1 during interval $[t_s^1, t_e^1]$, and there exists an edge $e_s^0 = (u_1, v_1, t_s^{01}, t_e^{01})$ such that $e_s^0 e_s^1$ are two consecutive edges with $t_s^{01} \leq t_s^{11}$ and $t_s^{11} \leq t_e^{01}$, *i.e.*, the timestamps of e_s^0 and e_s^1 are overlap.

Therefore, we construct the temporal path in G: $u_1 v_1 u_2 v_1 u_3 v_2 u_4$ to replace the first edge in $P_{\mathcal{G}}$.

(II) We can similarly construct other consecutive edges for the edge from s_i to s_{i+1} with $i \in [2, n-1]$. Therefore, we can deduce a temporal path from u to v. we omit the details here. $\square$