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

In this paper, we address the reachability problem of temporal bipartite graphs, which model the interactions between two distinct types of entities, and carry the temporal information about the time of the interactions. Such graphs can be used to control disease outbreaks, trace metabolic pathways and detect fake news. Existing algorithms construct indices to answer the reachability problem on such graphs. However, constructing indices suffers from size inflation or cannot be paralleled efficiently. To address these, we propose a contraction framework, named CBTR. It first transforms the temporal bipartite graph into a directed acyclic graph (DAG) without losing reachability information, and constructs indices using a contraction-based index schema, to reduce the used space. Using real-life and synthetic data, we experimentally verify that on average CBTR outperforms the state-of-the-art algorithms by 74 times in constructing indices.

1 INTRODUCTION

A bipartite graph (bigraph) is a special case of graphs, in which the vertices are divided into two disjoint sets such that all edges link vertices from different sets. Such graphs can model the interactions between two distinct types of entities, and carry the temporal information about the time of the interactions. Such graphs are widely used in various fields like disease prevention [34, 48, 61], recommendation systems [6, 22, 32, 54], social networks analysis [3, 7, 67], user intent analysis [17, 33] and machine learning [24, 25, 58].

The reachability problem has been extensively studied in temporal graphs, like temporal reachability problem [9, 12, 44], restless temporal reachability problem [10, 16, 45], shortest path distance problem [27, 55, 56] and span-reachability query [52, 57, 64]. In this paper, we focus on the span-reachability query, which is to decide, given a temporal graph G, a time interval $[t_s, t_e]$ and two vertices u and v in G, whether there exists a temporal path P from u to v such that the timestamps on the path P are within the interval $[t_s, t_e]$. Such problem has also been studied for bipartite graphs [12, 49], and can be used to control disease outbreaks [12, 18, 21, 26, 65], trace metabolic pathways [11, 12, 39] and detect fake news [36, 42].

There exist two approaches to answer the span-reachability query. (1) One method first projects bipartite temporal graphs to temporal unipartite graphs, and then applies the existing techniques [56, 64] to conduct the queries. However, such method is inefficient due to the size inflation and suffices from the information loss [12], since (a) the projection may form some large cliques, *e.g.,* multiple top vertices link to the same lower vertex at the same time, and these vertices can reach each others, *i.e.,* a clique; and (b) either some top vertex or bottom vertex may be removed during the projections, *i.e.,* information loss. (2) The second method conducts the queries on temporal bipartite graphs directly, *e.g.,* [12] extends the 2-hop labeling to temporal bipartite graphs, and conducts queries on temporal bigraphs directly. However, such method also suffices from the efficiency problem. On the temporal graph PubMed with 0.7B edges, the algorithm takes 4.22 hours and uses 32 processors with 512G memory, to construct the indices. This is because the algorithm constructs global indices for *the whole graph*, and needs to address some conflict issue and canonical issue, which degrade the performance of parallel algorithms.

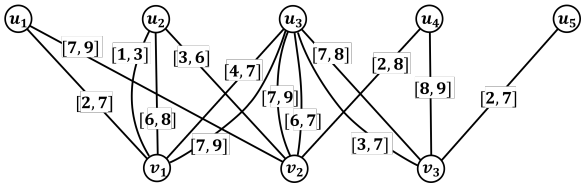


Figure 1: A temporal bipartite graph G

Example 1.1. Figure 1 illustrates a simple temporal bigraph, where vertices in the top layer and vertices in the bottom layer represent two different kinds of entities. The edges represent interactions between these two kinds of entities. Timestamp $[t_s, t_e]$ on an edge indicates the starting time t_s and ending time t_e of the interaction. Consider a fake news propagation scenario [36, 42], where the top layer vertices represent people, and the bottom ones denotes a web page. When fake news is released and affects the economies [28], it is essential to determine the source of the fake news and refute it. Existing approaches have the following problem.

(1) The projection-based method causes size inflation, since the interactions can lead to a clique. Given persons (*i.e.,* vertices in the top layer) u_1, u_2, u_3 and website (*i.e.,* vertex in the bottom layer) v_1, the projected graph consists of only vertices representing persons, and contains four edges $(u_1, u_2, 1, 2)$, $(u_1, u_2, 3, 6)$, $(u_1, u_3, 4, 6)$, $(u_2, u_3, 4, 8)$, representing the time intervals when two persons visit the same website. For example, edge $(u_1, u_2, 1, 2)$ denotes that persons u_1 and u_2 visits the website v_1 from timestamp 1 to 2. On a popular website, there exist many people visiting the website at the same time, which results in a large clique. Moreover, the information about the websites are missing in the constructed graphs. Observe that fake news is removed immediately from such website, and we can only trace the fake news using the time when the users visit the website.

(2) To avoid the size inflation and information loss, [12] develops a hop-based indices to answer queries directly on temporal bipartite graphs. To construct indices, it conducts a DFS-like algorithm on the graph, which cannot be parallelized easily due to a *global* order dependency among vertices, *i.e.,* the degree-based vertex order. Given the graph in Figure 1, the algorithm constructs the indices following the order: $u_3, v_1, v_2, v_3, u_2, u_1, u_4, u_5$, whose degrees are 6, 5, 5, 4, 3, 2, 2, 1, respectively. □

Contributions. In this paper, we propose a novel method to an-

swer the reachability query on temporal bigraphs, which can not only avoid the size inflation and information loss, but also can be paralleled easily. Our contributions are summarized as follows:

(1) Graph transformation (Section 4). We propose a novel temporal bigraphs transformation method to convert the temporal bipartite graph into a directed acyclic graph (DAG). To avoid the size inflation, we design suspended and compose snapshots to represent the reachability information compactly. Moreover, the DAG preserves all the necessary information to answer the span-reachability query.

(2) Index construction (Section 5). We develop a contraction-based indexing schema. Given a transformed DAG, we first partition the graph into multiple fragments, then independently construct indices for each fragment, and finally design a graph contraction strategy to connect these fragments. Using this schema, we can decompose the indices into inter-fragment indices and intra-fragment indices, and develop a parallel algorithm to accelerate the index construction. We can exploit any existing algorithms to construct the intra-fragment indices. And for the inter-fragment indices, we group vertices based on the reachability relationships among vertices in the same fragment, which can reduce the size of constructed indices. Using these strategies, the contraction-based indices can be constructed efficiently on large-scale graphs (see Section 7). Denote by $\mathsf{CBTR}_{\mathsf{In}}$ the proposed algorithm for index construction.

(3) Query algorithm (Section 6). We propose a two-stage query algorithm $\mathsf{TRPChecker}_{\mathsf{In}}$ for the query, to take advantage of the contraction-based indexing schema. When the source vertex and the target vertex are in the same fragment, we apply existing algorithms on intra-fragment indices to answer the query. When they are in different fragments, since the number of fragments are small, we conduct a breadth-first search to answer the query. Due to the contraction-based indices, the complexity of our two-stage query algorithm is comparable to that of the existing algorithms [12].

(4) effectiveness and efficiency (Section 7). We conduct extensive experiments on real-world and synthetic datasets to verify the following. (1) $\mathsf{CBTR}_{\mathsf{In}}$ is efficient. On AM, it is 113× faster than SOTA algorithm TBP. (2) $\mathsf{CBTR}_{\mathsf{In}}$ can handle large graphs, which cannot be processed by TBP. (3) $\mathsf{TRPChecker}_{\mathsf{In}}$ is faster than WTB. (4) $\mathsf{CBTR}_{\mathsf{In}}$ scales well with large graphs. On a graph with $7M$ vertices and $129M$ edges, $\mathsf{CBTR}_{\mathsf{InP}}$ takes less than 5 hours to construct indices.

Organization. This paper is organized as follows. We introduce the preliminaries in Section 2 and give an overview of the solution in Section 3. Then we present the details of our proposed method in Sections 4, 5 and 6. Extensive experiments are conducted and will be introduced in Section 7.

Related work. We categorize the related work as follows.

General graph. The reachability problem in general graphs has been extensively studied. The problem is to determine whether there is a path between two nodes in a graph, directed or undirected. The most well-known algorithm for this problem is the depth-first search (DFS) algorithms and the breadth-first search (BFS) algorithm. However, these methods cannot handle large-scale graphs [31]. Many indexing methods are proposed to accelerate the computation: (1) tree-cover based methods (*e.g.,* path-tree [29], gripp [46] and grail [59]), which construct (a) a tree containing most part of reachability information in the graph, and (b) a supplementary index table storing the remaining information; (2) hop-based methods (*e.g.,* 2-hop [14], 3-hop [30], HHL [2], O'Reach [23], BL [60] and TF-Label [13]), which pick a set of nodes as the landmarks and compute the reachability queries using the landmarks as bridges; and (3) Pruned-based methods (*e.g.,* IP [51], BFL [40], PReaCH [35] and ELF [41]), which compute some conditions for vertices, and filter out unreachable queries. See [62, 63] for more discussion.

This work differs from the prior work in the following. (1) We target the reachability problem on graphs with temporal information, which is more challenges. (2) To handle large-scale graphs, we develop a partition-based strategy and an equivalent relation between vertices in the graphs, to reduce the size of graphs. (3) We design a two-stage algorithm to query on the partitioned graphs.

Temporal graphs. The reachability problem has been studied on temporal graphs [38, 50, 52, 53, 56, 64]. [38] transforms temporal graphs into DAGs by materializing connection components at each timestamp. [50] extends the hop-based methods to temporal graphs and adopts index partitioning strategy to handle consecutive time intervals. [56] transforms temporal graphs into a DAG, decomposes the transformed DAG into a set of chains, and extends the hop-based methods to construct indices. [64] proposes TVL indices for the reachability queries in distributed setting. [52, 53] extends the 2-hop method with timestamps to answer the span-reachability problem in temporal graphs.

This work differs from the prior work as follows. (1) It aims to solve the reachability problem defined on temporal bigraphs, which have special structures. We can exploit these structures to develop efficient algorithms. (2) It defines equivalent sets of vertices to reduce the size of graphs, which can accelerate the computation.

Closer to this work is [12, 44]. [12] studies the reachability problem on temporal bigraphs, and proposed TBP-Indices by extending 2-hop method. [44] proposes a partition-based strategy to construct indices for reachability queries within budget.

In contrast to [12, 44], (1) we adopt different definitions of reachability, which is more complicate than that of [12] (see Section 2), and is different from the problem studied in [44]. (2) We transform the temporal graphs to general graphs without temporal information, and use well-established algorithms to solve the reachability problems, rather than developing a new one as in [12, 44]. (3) We develop a partition-based algorithm to construct the indices, which can be paralleled without solving "conflict" issues as in [12].

2 PRELIMINARY

Assume a countably infinite set Ω of timestamps, which are ordered *w.r.t.* a discrete order $\leq$. A time interval is a range $[t_1, t_2]$ for any two timestamps $t_1, t_2 \in \Omega$ satisfying $t_1 \leq t_2$. Let $\mathcal{T} \subseteq \Omega \times \Omega$ be the set of all time intervals. Given two intervals $I_1 = [t_1, t_2]$ and $I_2 = [t_3, t_4]$, they are *consecutive* if the ending time of interval I_1 is the same as the starting time of interval I_2, *i.e.,* $t_2 = t_3$. They are *overlapped* if $\min(t_2, t_4) \geq \max(t_1, t_3)$. To simplify the presentation, given a time interval $I = [t_1, t_2]$ denote by $I.t_s$ and $I.t_e$ the starting time and the ending time of I, respectively, *i.e.,* $I.t_s = t_1$ and $I.t_e = t_2$.

Temporal bigraph. Consider an undirected temporal bigraph

$G = (V, E)$, where (1) V is a finite set of vertices which are partitioned into two disjoint sets U and L such that $V = U \cup L$ and $U \cap L = \emptyset$; vertices in U and L are called top vertices and bottom vertices, respectively. (2) $E \subseteq U \times L \times \mathcal{T} \times \mathcal{T}$ 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 from the starting time t_s to the ending time t_e, *i.e.,* in the time interval $[t_s, t_e]$. We also use $e.t_s$ and $e.t_e$ to denote the *starting time* and *ending time* of edge e, respectively.

Given a vertex u, denote by $E(u)$ the set of edges incident to u, $d(u)$ the degree of u, *i.e.,* $d(u) = |E(u)|$.

Connection. Given a temporal bigraph $G = (U \cup L, E)$ and two top vertices $u, u' \in U$, there is a connection between u and u' if u and u' connect to some bottom vertex in L at the same timestamp; *e.g.,* two persons visit the same location at the same time. More specifically, a connection between two top vertices u and u' at bottom vertex v, denoted by $u \prec\!\!\{e_1, v, e_2\}\!\!\succ u'$, consists of two different edges $e_1 = (u, v, t_s^1, t_e^1)$ and $e_2 = (u', v, t_s^2, t_e^2)$ in E such that their time intervals are overlapped, *i.e.,* $\min(t_e^1, t_e^2) \geq \max(t_s^1, t_s^2)$. The time interval of the connection, denoted by $\Theta(e_1, e_2)$, is the intersection of time intervals in edges e_1 and e_2, *i.e.,* $\Theta(e_1, e_2) = [\max(t_s^1, t_s^2), \min(t_e^1, t_e^2)]$.

Temporal Path. Given a temporal graph $G = (U \cup L, E)$ and two top vertics u and u' in U, a *temporal path* ρ from u to u' of length k is a sequence of vertices $\rho = u_1 v_1 u_2 v_2 \ldots u_k v_k u_{k+1}$ with $u = u_1$ and $u' = u_{k+1}$ such that for each $i \in [1, k-1]$, there exist two consecutive connections $u_i \prec\!\!\{e_1^i, v_i, e_2^i\}\!\!\succ u_{i+1}$ and $u_{i+1} \prec\!\!\{e_1^{i+1}, v_{i+1}, e_2^{i+1}\}\!\!\succ u_{i+2}$ in G such that: (1) the starting times of two consecutive connections increase, *i.e.,* $\Theta(e_1^i, e_2^i).t_s \leq \Theta(e_1^{i+1}, e_2^{i+1}).t_s$; and (2) the timestamps in two adjacent edges of a top vertex increases; more specifically, (a) if $v_i \neq v_{i+1}$, then $e_2^i.t_e \leq e_1^{i+1}.t_s$; and (b) if $v_i = v_{i+1}$, then $e_2^i = e_1^{i+1}$ (*i.e.,* e_2^i and e_1^{i+1} are the same edge) or $e_2^i.t_e \leq e_1^{i+1}.t_s$. Here, both e_2^i and e_1^{i+1} are edges of top vertex u_{i+1}. Intuitively, when $v_i \neq v_{i+1}$, top vertex u_{i+1} moves from one bottom vertex to another bottom vertex, and the timestamp increases, *i.e.,* $e_2^i.t_e \leq e_1^{i+1}.t_s$; and when $v_i = v_{i+1}$, top vertex u_{i+1} "stays at" the same bottom vertex; that is, either e_2^i and e_1^{i+1} are the same edge, or u_{i+1} "leaves" and "reenters" the bottom vertex.

Reachability. We define the starting time (resp. the ending time) of path ρ as the starting time $\Theta(e_1^1, e_2^1).t_s$ of the first connection (resp. the ending time $\Theta(e_1^k, e_2^k).t_e$ of the last connection). We say that u can reach u' in G in the time interval I if there exists a temporal path ρ from u to u' and its starting time and ending time fall in I.

Example 2.1. Consider the temporal bigraph G in Figure 1. (1) $u_2 \prec\!\!\{e_1, v_2, e_2\}\!\!\succ u_4$ is a connection due to edges $e_1 = (u_2, v_2, 3, 6)$ and $e_2 = (u_4, v_2, 2, 8)$, and $\Theta(e_1, e_2) = [3, 6]$, *i.e.,* the starting time and ending time of the connection are 3 and 6, respectively; and (2) $u_4 \prec\!\!\{e_2, v_2, e_3\}\!\!\succ u_1$ is also a connection with e_2 and e_3=$(u_1, v_2, 7, 9)$, and $\Theta(e_2, e_3)$=[7, 8]. The two connections form a temporal path ρ=$u_2 v_2 u_4 v_2 u_1$ from u_2 to u_1 with starting time 3 and ending time 8. *i.e.,* u_2 can reach u_1 in any time interval containing [3, 8]. Note that edge e_2 appears twice in ρ, *i.e.,* the second edge in the first connection and the first edge in the second connection. □

Problem Statement. We study *the temporal reachability problem*, referred to simply as TRP problem, 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 temporal reachability query with u as the *source vertex* and u' as the *target vertex*.

Remark. Our definition of temporal paths is different from the one in [12, 36]. Here, the starting time and ending time of a connection $u \prec\!\!\{v, e_1, e_2\}\!\!\succ u'$ are those of the common range of time intervals of two edges e_1 and e_2, *i.e.,* $\Theta(e_1, e_2)$. But in [12, 36], the starting time and ending time of two adjacent edges in a path are those of the union of time intervals of e_1 and e_2, *i.e.,* $[\min(t_s^1, t_s^2), \max(t_e^1, t_e^2)]$. Intuitively, (1) the time interval of a connection is a fragment of the time intervals of edges e_1 and e_2. Then a temporal edge can appear multiple times in a temporal path for TRP. (2) However, the time interval of two adjacent edges covers all timestamps in e_1 and e_2, then each edge can appear only once in a temporal path defined in [12, 36]. Due to such difference, there are more temporal paths between any two vertices in TRP than in [12, 36].

Example 2.2. We use the path $u_2 v_2 u_4$ in Example 2.1 to show the difference between our definition and the one in [12, 36].

We first show that $u_2 v_2 u_4$ is not a valid path in under the definition in [12, 36]. (1) the path $u_2 v_2 u_4$ connecting u_2 and u_4 is formed by two edges $e_1 = (u_2, v_2, 3, 6)$ and $e_2 = (u_4, v_2, 2, 8)$, called wedge in [12, 36] and denoted by $\mathcal{W}_1 = ((u_2, v_2, 3, 6), (u_4, v_2, 2, 8))$; the starting time and ending time of the wedge $\mathcal{W}_1$ are 2 (*i.e.,* $\min(e_1.t_s, e_2.t_s)$) and 8 (*i.e.,* $\max(e_1.t_e, e_2.t_e)$), respectively. (2) The path $u_4 v_2 u_1$ from u_4 to u_1 is formed by wedge $\mathcal{W}_2 = ((u_4, v_2, 2, 8), (u_1, v_2, 7, 9))$, whose starting time and ending time are 2 and 9, respectively. (3) Wedges $\mathcal{W}_1$ and $\mathcal{W}_2$ cannot form a temporal path from u_2 to u_1, since the starting time of $\mathcal{W}_2$ is 2, which is smaller than the ending time (*i.e.,* 8) of $\mathcal{W}_1$.

However, the path $u_2 v_2 u_4$ is practical, *i.e.,* u_2 can reach u_1 via u_4 in real-life setting. For example, u_4 keeps linking to bottom vertex v_3 (*e.g.,* website) after "reading" a fake news spread by u_2 at timestamp 3, and passes the fake news to u_1 at timestamp 7 via another bottom vertex (*e.g.,* website) v_2.

Such a connection can be represented by a temporal path $u_2 \prec\!\!\{e_1, v_2, e_2\}\!\!\succ u_4 \prec\!\!\{e_2, v_2, e_3\}\!\!\succ u_1$ from u_2 to u_1 under our definition, and u_2 can reach u_1 in the time interval [2, 9]. Observe that (1) the starting time and ending time of the connection between u_2 and u_4 (resp. u_4 and u_1) are 3 and 6 (resp. 3 and 8), respectively; and (2) edge $(u_4, v_2, 2, 8)$ appears more than once in the path, which is not allowed in the definition of [12, 36]. □

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

3 SOLUTION OVERVIEW

In this section, we overview the contraction-based framework for the TRP problem, denoted by CBTR.

Architecture. Taking a temporal bigraph G as input, CBTR first transforms G into a directed acyclic graph (DAG) $\mathcal{G}_G$, called traject graph, then partitions $\mathcal{G}_G$ into multiple fragments and contracts these fragments to reduce their sizes, and finally checks whether two vertices are connected on the contracted graphs.

Step 1: Graph transformation. Given a temporal bigraph G, it first

Table 1: Notations

Notations	Definitions
$G = (U \cup L, E)$	an undirected temporal bigraph
$Q(u, u', G, I)$	the temporal reachability query, *i.e.,* whether u can reach u' in G in the time interval I
$\mathcal{G}_G$	the traject graph of temporal graph G
$\mathcal{T}_G$	the set of timestamps appearing graph G
$d_{\max}$	the maximum degree of vertices in G
$u \prec\!\!-\{e_1, v, e_2\}\!\!\succ u'$	connection between top vertices u and u' at bottom vertex v
$\Theta(e_1, e_2)$	time interval of the connection consisting of e_1 and e_2
$[t_1, t_2].t_s/[t_1, t_2].t_e$	starting/ending time of time interval $[t_1, t_2]$
$[t_1, t_2]_v$	A snapshot of a bottom vertice v
$\mathcal{S}(G)/\mathcal{S}(u)$	the set of all snapshots constructed in graph G/from top vertex u
$[v]_{\mathsf{EIn}_i}/[v]_{\mathsf{EOut}_i}$	the in-equivalence/out-equivalence class
$\mathsf{Bin}_i[v]/\mathsf{Bout}_i[v]$	the set of border vertices that can reach / be reached from v in fragment F_i

constructs a traject graph $\mathcal{G}_G$ such that the reachability relations between top vertices in G preserve in $\mathcal{G}_G$. Moreover, the traject graph $\mathcal{G}_G$ is a DAG, in which (1) each vertex in $\mathcal{G}_G$ is a snapshot of a bottom vertex v that records the time interval in which there exist some top vertices connected to v, and (2) each edge represents a reachability relation between top vertices. Actually, most existing approaches answer the reachability problem on temporal graphs by constructing a DAG from G, in which each vertex represents a strongly connected components in G (*e.g.,* [66]).

Step 2: partition and indexing. To handle large-scale graphs, CBTR first partitions the traject graph $\mathcal{G}_G$ into multiple fragments, and then construct intra-fragement and inter-fragment indices for the reachability problem. Then CBTR can apply well-established algorithms for the reachability problem [14, 23, 29, 30, 46, 59] to compute intra-fragment indices. For inter-fragment indices, CBTR applies a graph contraction strategy to reduce the size of graphs [20], and defines equivalent sets to construct the indices. We also develop parallel algorithms to compute the indices.

Step 3: Answering the temporal reachability queries. We develop a query algorithm for TRP by using the constructed indices. Given two top vertices, it first identifies the snapshots, which are constructed from the adjacent edges of the two top vertices; then for these snapshots it checks whether they are in the same fragment; if so, it exploits the intra-fragment indices to answer the queries; otherwise it answers the queries by using the inter-fragment indices.

Properties. CBTR has the following properties.

(1) The traject graph $\mathcal{G}_G$ is only larger than the given graph G by a small factor, which avoid the size inflation of the projection-based methods [56, 64] (Section 4).

(2) We can also avoid the global order dependency of [12] by using both intra-fragment and inter-fragment indices, which can be effectively parallelized. There does not exist any dependency among the construction of these indices (Section 5).

(3) Given a large scale graph G, when existing algorithms [12, 36] cannot construct the indices for G due to insufficient memory, CBTR can also construct both intra-fragment indices and inter-fragment indices to answer the reachability query. Intuitively, after partitioning the graph into multiple fragments, CBTR does not construct indices for vertices that are far from each other, and the reachability between these vertices can be checked via inter-fragment indices. This is to strike a balance between the query time and the space used for indices (Sections 6).

4 GRAPH TRANSFORMATION

In this section, we transform a temporal bipartite graph into a DAG where the reachability relations between top vertices preserve.

Challenge. We can use either top vertices or bottom vertices to construct the DAG. In this paper, we use bottom vertices as a case study. Intuitively, such DAG represents the "movements" or "propagation" of top vertices among different bottom vertices.

To construct a DAG using bottom vertices, we need to address the following challenges. (1) The given temporal bigraph is undirected and may contain multiple timestamps and cycles (see Figure 1). And (2) how can we ensure that the DAG preserves the reachability relationships between top vertices in the time interval?

We address these challenges as follows. (1) To preserves the reachability relationships between top vertices, we define a data structure named snapshots to represent bottom vertices and the time intervals during which some top vertices link to these bottom vertices. (2) Then we order these snapshots based on the timestamps and link them based on the adjacent edges of top vertices. In this way, when a top vertex u can reach another top vertex v, we can verify this following the "movements" of top vertex u in the DAG.

Snapshots. We start with the definition of snapshots. Given a temporal bigraph $G = (U \cup L, E)$, let $\mathcal{T}_G$ be the set of timestamps appearing in G. A *snapshot* of a bottom vertex v in L, denoted by $[t_1, t_2]_v$, represents that there exist two top vertices in U linking to v in the time interval $[t_1, t_2]$. More specifically, bottom vertex v and time interval $[t_1, t_2]$ satisfy the following condition: there exist two top vertices u_1 and u_2 in U such that $u_1 \prec\!\!-\{e_1, v, e_2\}\!\!\succ u_2$ is a connection in G, and $t_1 = \Theta(e_1, e_2).t_s$ and $t_2 = \Theta(e_1, e_2).t_e$, *i.e.,* $[t_1, t_2]$ consists of common timestamps of e_1 and e_2. We call u_1 and u_2 the top vertices of snapshot $[t_1, t_2]_v$.

There may exist multiple pairs of edges linking to v such that the overlap of their time intervals is $[t_1, t_2]$. For each snapshot $s = [t_1, t_2]_v$ and each of its top vertex u, we record the maximum and minimum timestamps of the edges of u, from which snapshot s is constructed. More specifically, let $e_1, e_2, \ldots, e_l$ be all adjacent edges of u that are used to construct s. Denote by $s.\mathsf{smax}[u]$ (resp. $s.\mathsf{emin}[u]$) the maximal starting time (resp. minimal ending time) of edges in $\{e_1, e_2, \ldots, e_l\}$. Such timestamps are used to link snapshots (see below). Note that different top vertices may have different $s.\mathsf{smax}[\cdot]$ and $s.\mathsf{emin}[\cdot]$ for the same snapshot s.

Construction. Given a temporal bigraph G, we can construct the set of all snapshots as follows: for each bottom vertex v, (1) collect all its adjacent edges $e_1 = (u_1, v, t_s^1, t_e^1), \ldots, e_n = (u_n, v, t_s^n, t_e^n)$; and (2) for each pair of adjacent edges e_i=(u_i, v, t_s^i, t_e^i) and $e_j = (u_j, v, t_s^j, t_e^j)$, construct a snapshot $[t_s, t_e]_v$ such that $\max(t_s^i, t_s^j) = t_s$ and $\min(t_e^i, t_e^j) = t_e$, if e_i and e_j form a connection, *i.e.,*

Table 2: All snapshots and their top vertices

U	Snapshots
u_1	$[2,3]_{v_1}, [4,7]_{v_1}, [6,7]_{v_1}, [7,7]_{v_1}, [7,7]_{v_2}, [7,8]_{v_2}, [7,9]_{v_2}$
u_2	$[2,3]_{v_1}, [3,6]_{v_2}, [6,6]_{v_2}, [6,7]_{v_1}, [7,8]_{v_1}$
u_3	$[3,7]_{v_3}, [4,7]_{v_1}, [6,6]_{v_2}, [6,7]_{v_1}, [6,7]_{v_2}, [7,7]_{v_1}, [7,7]_{v_2}, [7,7]_{v_3}, [7,8]_{v_1}, [7,8]_{v_2}, [7,9]_{v_2}, [8,8]_{v_3}$
u_4	$[3,6]_{v_2}, [6,7]_{v_2}, [7,8]_{v_2}, [8,8]_{v_3}$
u_5	$[3,7]_{v_3}, [7,7]_{v_3}$

$\min(t_e^i, t_e^j) \geq \max(t_s^i, t_s^j)$. Denote by $\mathcal{S}(G)$ the set of all snapshots constructed for all bottom vertices as above in graph G.

The construction of $\mathcal{S}(G)$ takes $O(|E| \cdot d_{\max})$ time, where $d_{\max}$ is the maximum degree of vertices in G, since for each bottom vertex v, it takes $O(d_v d_{\max})$ time to compare all pairs of adjacent edges.

Example 4.1. Consider the temporal bigraph G in Figure 1 and its bottom vertex v_2. Recall the connections given in Example 2.1, *i.e.,* $u_2 \prec\{e_1, v_2, e_2\}\rightarrow u_4$ with $\Theta(e_1, e_2)=[3,6]$ and $u_4 \prec\{e_2, v_2, e_3\}\rightarrow u_1$ with $\Theta(e_2, e_3) = [7,8]$, where $e_1=(u_2, v_2, 3, 6)$, $e_2=(u_4, v_2, 2, 8)$ and $e_3=(u_1, v_2, 7, 9)$. We construct snapshot $[3,6]_{v_2}$ (resp. $[7,8]_{v_2}$) for top vertices u_2 and u_4 (resp. u_1 and u_4). Similarly, we get snapshot $[6,7]_{v_1}$ from the connection $u_2 \prec\{e_4, v_1, e_5\}\rightarrow u_3$, where $e_4 = (u_2, v_1, 6, 8)$, $e_5 = (u_3, v_1, 4, 7)$ and $\Theta(e_4, e_5) = [6,7]$.

Moreover, for snapshot $s = [3,6]_{v_2}$ and top vertex u_2, e_1 is the only adjacent edge of u_2 that is used to construct s. Then $s.\text{smax}[u_2] = 3$ and $s.\text{emax}[u_2] = 6$, where 3 and 6 are the starting time and ending time of e_1, respectively. Similarly, e_1 (resp. e_4) is the only adjacent edge of u_2 that is used to construct $s' = [6,6]_{v_2}$ (resp. $s'' = [6,7]_{v_1}$), and then $s'.\text{smax}[u_2] = s'.\text{emax}[u_2] = 6$ (resp. $s''.\text{smax}[u_2] = 6$ and $s''.\text{emax}[u_2] = 7$).

Table 2 lists all snapshots of G and their top vertices. Observe that (1) for each snapshot $[t_1, t_2]_v$, it may be constructed from multiple connections, *e.g.,* snapshot $[7,8]_{v_2}$ can be constructed from another connection $u_3 \prec\{e_4, v_2, e_2\}\rightarrow u_4$, where $e_4 = (u_3, v_2, 7, 9)$; thus $[t_1, t_2]_v$ may have more than two top vertices, *e.g.,* u_1, u_3 and u_4 for $[7,8]_{v_2}$; (2) similarly, for each top vertex u, it may be the top vertex of snapshots for more than one bottom vertex, *e.g.,* $u_1 - u_4$. □

Traject Graphs. Using the snapshots in $\mathcal{S}(G)$, we define the traject graph $\mathcal{G}_G$ for temporal bigraph G. Intuitively, the edges between snapshots in $\mathcal{S}(G)$ represent that some top vertex "moves" from one bottom vertex to another bottom vertex.

Given temporal bigraph $G = (U \cup L, E)$ and the set $\mathcal{S}(G)$ of all snapshots, the traject graph of G is $\mathcal{G}_G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{S}(G)$. We construct the edges in $\mathcal{E}$ by connecting snapshots that are constructed from adjacent edges of the same top vertex. More specifically, for each top vertex u of 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 adjacent edges of u. Based on the definition of temporal paths, $s_i=[t_s^i, t_e^i]_{v_i}$ has an edge to $s_j=[t_s^j, t_e^j]_{v_j}$, if one of the following conditions holds: (1) when $v_i \neq v_{i+1}$, $s_i.\text{emin}[u] \leq s_j.\text{smax}[u]$ holds; and (2) when $v_i = v_{i+1}$, either (i) $t_s^i \leq t_s^j$ holds and s_i and s_j are constructed from the same edge, or (ii) $s_i.\text{emin}[u] \leq s_j.\text{smax}[u]$ holds. We can record the edges, from which s_1 and s_2 are constructed during their constructions. Intuitively, when $s_i.\text{emin}[u] \leq s_j.\text{smax}[u]$ holds, there exist two connections $u_1 \prec\{e_4, v_2, e_2\}\rightarrow u$ and $u \prec\{e_3, v_2, 4_2\}\rightarrow u_2$ such that $e_2.t_e = s.\text{emin}$ and $e_3.t_s = s.\text{smax}$. Then $u_1 \prec\{e_4, v_2, e_2\}\rightarrow u \prec\{e_3, v_2, 4_2\}\rightarrow u_2$ forms a temporal path.

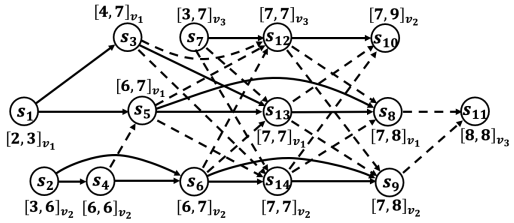


Figure 2: The traject graph $\mathcal{G}_G$ of graph G in Figure 1

Remark. Connecting all snapshots satisfying the above conditions may result in redundant edges, *e.g.,* when 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 remove such edges, we require that there exists an edge from $s_1=[t_s^i, t_e^i]_{v_i}$ to $s_2=[t_s^j, t_e^j]_{v_j}$, only when no $s_3=[t_s^l, t_e^l]_{v_l}$ in $\mathcal{S}(u)$ satisfies that the time interval $[t_s^l, t_e^l]$ of s_3 is between the ending time of s_1 and the staring time of s_2, *i.e.,* $t_e^i \leq t_s^l \leq t_e^l \leq t_s^j$.

Example 4.2. Figure 2 presents the traject graph $\mathcal{G}_G$ of the temporal bigraph G in Figure 1. The edges represented by dashed arrows and solid arrows represent that conditions (1) and (2) hold, respectively. For example, from Table 2 and Example 4.1, $\mathcal{S}(u_2) = \{[2,3]_{v_1}, [3,6]_{v_2}, [6,6]_{v_2}, [6,7]_{v_1}, [7,8]_{v_1}\}$, and (a) there exists an edge from $s_2 = [3,6]_{v_2}$ to $s_4 = [6,6]_{v_2}$ (represented by a solid arrow), since $s_2.\text{emin}[u_2] = 6 \leq s_4.\text{smax}[u_2] = 6$; and (b) there exists an edge from s_4 to $s_5 = [6,7]_{v_1}$ (represented by a dashed arrow), since $s_4.\text{emin}[u_2] = 6 \leq s_5.\text{smax}[u_2] = 6$. □

Optimizations. The traject graphs constructed above may also contain redundant edges. Consider the following example.

Example 4.3. From Table 2 and Figure 1, u_3 links to v_1, v_2 and v_3 at the timestamp 7. Based on the construction of traject graph, these snapshots bring multiple edges in $\mathcal{G}_G$. As shown in Figure 2, it makes snapshot s_3 has edges in $\mathcal{G}_G$, linking to s_{12}, s_{13} and s_{14}; similarly for snapshots s_5, s_6 and s_7, all having edges linking to s_{12}, s_{13} and s_{14}. This results in $4 \times 3 = 12$ edges. For the same reason, s_{12}, s_{13} and s_{14} all have edges linking to s_8, s_9 and s_{10}, which gets us nine edges in $\mathcal{G}_G$. □

Compose snapshots. To reduce the number of edges, we introduce an intermediate snapshot to represent that multiple snapshots are linked to each other. More specifically, given a time interval $[t, t]$ appearing in G and a set $L_v = \{v_1, \ldots, v_k\}$ of bottom vertices in G, we define a *compose snapshot* $[t, t]_{L_v}$ to represent multiple snapshots $[t, t]_{v_1}, \ldots, [t, t]_{v_k}$. For snapshots in Example 4.3, we introduce a compose snapshot $[7,7]_{\{v_1, v_2, v_3\}}$ to represent all three snapshots $[7,7]_{v_1}$, $[7,7]_{v_2}$ and $[7,7]_{v_3}$ (*i.e.,* s_{13}, s_{14} and s_{12}). Then we directly link snapshots s_3, s_5, s_6 and s_7 to the compose snapshot $[7,7]_{\{v_1, v_2, v_3\}}$, which results in only four edges, rather than 12 edges as in Example 4.3.

Compose snapshots are defined on a single timestamp, rather than on an interval as snapshots. This is because the snapshots have different sizes of time intervals representing different connections of top vertices, and compose snapshots with a single timestamp can be used to link snapshots of different sizes (see below).

Maximum compose snapshots. However, there may exist expo-

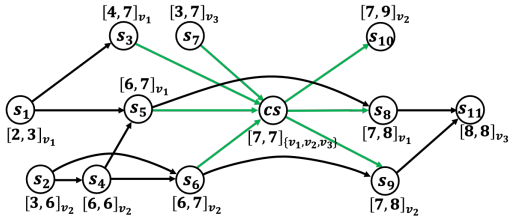


Figure 3: Graph $\mathcal{G}_G$ optimized by composite snapshots

nentially many compose snapshots, one for each subset of $\{v_1, v_2, \ldots, v_k\}$. To solve this, we consider only maximum compose snapshots $[t,t]_{\{v_1,v_2,\ldots,v_k\}}$, which satisfies the following two conditions: (a) a top vertex u links to bottom vertices v_i and v_j at timestamp t for any two bottom vertices v_i and v_j in $\{v_1, v_2, \ldots, v_k\}$, *i.e.,* there exist two edges $e_1 = (u, v_1, t_s^1, t_e^1)$ and $e_2 = (u, v_2, t_s^2, t_e^2)$ satisfying that $t \in [t_s^1, t_e^1]$ and $t \in [t_s^2, t_e^2]$; that is, $[t,t]_{\{v_1,v_2,\ldots,v_k\}}$ is a compose snapshot; and (b) no other bottom vertex $v' \notin \{v_1, v_2, \ldots, v_k\}$ has an edge $e_1 = (u_1, v', t_s', t_e')$ such that $t \in [t_s', t_e']$, *i.e.,* $[t,t]_{\{v_1,v_2,\ldots,v_k\}}$ cannot be further extended.

Construction. We can construct all maximum compose snapshots by collecting all bottom vertices that can reach each other via edges having the same timestamp. More specifically, let $\mathcal{T}_G = \{t_1, \ldots, t_l\}$ be all timestamps appearing in G. We (1) first partition the temporal bipartite G into multiple subgraphs $G_1, \ldots, G_l$ such that G_i ($i \in [1,l]$) consists of only edges whose starting time or ending time is t_i, then (2) compute connected components in G_i, and collect all bottom vertices in the same connected components. Let $L_v = \{v_1, v_2, \ldots, v_k\}$ be a set of all bottom vertices existing in a connected component in G_i, then construct a compose snapshot $[t_i, t_i]_{\{v_1,v_2,\ldots,v_k\}}$. Here, we only use the starting and ending time of the edges to construct subgraphs, since such compose snapshots can connect snapshots with different starting times.

Computing all compose snapshots can be done in $O(|G|)$ time. Observe that (1) the total number of edges in $G_1, \ldots, G_l$ is bounded by $O(|G|)$, since each edge in G appears in two subgraphs, which represent the starting time and the ending time of the edge; and (2) connected components can be identified in $O(|G|)$ time [43].

Example 4.4. Figure 3 shows the traject graph $\mathcal{G}_G$ optimized by using composite snapshots. Observe that (1) the three snapshots $s_{12} = [7,7]_{v_3}, s_{13} = [7,7]_{v_1}$ and $s_{14} = [7,7]_{v_2}$ are replaced by a single compose snapshot $cs = [7,7]_{\{v_1,v_2,v_3\}}$ in Figure 3, and thus (2) 21 edges in Figure 2 that are adjacent to s_{12}, s_{13} or s_{14} are replaced by seven edges (in green color) adjacent to $[7,7]_{\{v_1,v_2,v_3\}}$. □

Suspended snapshots. To further reduce the number of redundant edges, we introduce another intermediate states, namely suspended snapshots, representing that a top vertex u has "left" some bottom vertex, but has not "entered" other bottom vertex at a timestamp. More specifically, a *suspended snapshot* for a top vertex u is in the form of $[t,t]_{\emptyset,u}$, whether t is a timestamp in G. Assume that top vertex u has the following snapshots $[1,2]_{v_1}, [1,3]_{v_2}, [1,4]_{v_3}, [5,10]_{v_3}, [6,10]_{v_2}, [7,10]_{v_1}$. Based on the construction about, there exist 9 edges between these snapshots; for example, $[1,2]_{v_1}$ has edges to $[5,10]_{v_3}, [6,10]_{v_2}, [7,10]_{v_1}$; similar for $[1,3]_{v_2}, [1,4]_{v_3}$. Using the suspended snapshots $[4,4]_{\emptyset,u}$, we can link $[1,2]_{v_1}, [1,3]_{v_2}, [1,4]_{v_3}$ to $[4,4]_{\emptyset,u}$, which has edges to $[6,10]_{v_2}, [7,10]_{v_1}$; similar for $[1,3]_{v_2}, [1,4]_{v_3}$. The resulted graph has only six edges. Given top vertex u, we only consider the timestamps of snapshots constructed with adjacent edges of u (see below).

Calibration of traject graphs. We next show how to calibrate traject graphs using compose and suspended snapshots. Rather than first constructing traject graphs as above and then merging states using compose snapshots, we directly construct traject graphs by connecting compose snapshots and suspended snapshots with snapshots.

More specifically, for each top vertex u we connect its snapshots, compose snapshots and suspended snapshots as follows: (1) for each snapshot $s_1 = [t_s^1, t_e^1]_{v_1}$ and each timestamp t that is the minimum timestamp larger than $s_1.\mathsf{emin}[u]$ among all timestamps in snapshots constructed from u, (a) if there exists a compose snapshot $c = [t_e^1, t_e^1]_{\{v_1', \ldots, v_k'\}}$, add an edge from s_1 to c; (b) if there exist a snapshot $s_2 = [t_s^2, t_e^2]_{v_2}$ such that $s_2.\mathsf{smin} = t$ then add an edge from s_1 to s_2; otherwise, (c) an edge from s_1 to the suspended snapshot $[t,t]_{\emptyset,u}$. (2) We can add the edges for compose snapshots and suspended snapshots similarly. Observe that each snapshot, compose snapshot or suspended snapshot has at most one outgoing edge.

Properties. We next present some properties of the traject graphs.

Theorem 4.5. *Given a temporal bigraph G=$(U \cup L, E, \mathcal{T})$, its traject graph $\mathcal{G}_G$=$(\mathcal{V}, \mathcal{E})$ satisfies $|\mathcal{V}| \leq 3d_{\max}|G|$ and $|\mathcal{E}| \leq 3d_{\max}^3|G|$.*

Here $d_{\max}$, which is the maximum degree of vertices in G.

Proof. (1) We first show that $|\mathcal{V}| \leq 3d_{\max}|G|$. Note that (a) for each top vertex u, there are at most $3d_u d_{\max}$ many snapshots, compose snapshots and suspended snapshots, which are constructed from edges of u. Here, d_u is the degree of u; and (b) the number of vertices in $\mathcal{V}$ is bounded by $\Sigma_{u \in U} 3d_u d_{\max} \leq 3|E|d_{\max} \leq 3d_{\max}|G|$.

(2) We next show that $|\mathcal{E}| \leq 3d_{\max}^3|G|$. This is because each snapshot s links to the following snapshots: (a) the ones that are from the same edges in G as s; and (b) the ones s_1 satisfying that $s.\mathsf{emin} \leq s_1.\mathsf{smax}$. For edges of class (a), there exist at most $O(d_{\max})$ many snapshots, since the snapshots of each edge are constructed by the bottom vertex. For edges of class (b), the number is bounded by the number of timestamps in the snapshots of a top vertex, *i.e.,* $d_u d_{\max}$, due to the usage of compose snapshots. Therefore, the number of edges for each snapshot is bounded by $2d_u d_{\max}$. And the total number of edges in $\mathcal{G}_G$ is bounded by $2d_{\max}^3|G|$. □

Equivalence of reachability. We next show that the traject graph $\mathcal{G}_G$ preserves the reachability relationship among top vertices in G.

We start with some notations. Assume that snapshots in $\mathcal{S}(G)$ are ordered by their starting times. Given time interval $[t_s, t_e]$, let S_u^f (resp. S_u^l) be the set of *first snapshots* (resp. *last snapshots*) in $\mathcal{S}(G)$ *after* timestamp t_s (resp. *before* timestamp t_e), which are constructed from adjacent edges of top vertex u. Note that there exist multiple snapshots after and before a given timestamp.

Theorem 4.6. *Given temporal bigraph G=$(U \cup L, E)$, its traject graph $\mathcal{G}_G$, two top vertices u and u', and a time interval $[t_s, t_e]$, u can reach u' in G in the time interval $[t_s, t_e]$ if and only if S_u^f can reach $S_{u'}^l$ in $\mathcal{G}_G$,* i.e., *a snapshot in S_u^f can reach one snapshot in $S_{u'}^l$.*

Proof. We only provide a proof sketch. See [1] for more details.

($\Rightarrow$) Assume that u can reach u' in G in time interval $[t_s, t_e]$. Then there exists a temporal path $\rho = u_1 v_1 u_2 v_2 \ldots u_k v_k u_{k+1}$ with $u = u_1$ and $u' = u_{k+1}$ such that the starting time and ending time of ρ fall in the time interval $[t_s, t_e]$. We can first pick two snapshots s_1 and s_2 from the sets S_u^f to $S_{u'}^l$, respectively, and then construct a path from s_1 to s_2 in the traject graph $\mathcal{G}_G$ by constructing an edge in $\mathcal{G}_G$ from each connection $u_i \prec\!\!\{v_i, [e_1^i, e_2^i]\}\!\!\succ u_{i+1}$ in the path ρ.

($\Leftarrow$) Assume that there exists a snapshot $[t_s^0, t_e^0]_{v_0}$ in S_u^f can reach one snapshot $[t_s^n, t_e^n]_{v_n}$ in S_v^l. Let $P_{\mathcal{G}} = (s_0 = [t_s^0, t_e^0]_{v_0}, s_1 = [t_s^1, t_e^1]_{v_1}, \ldots, s_n = [t_s^n, t_e^n]_{v_n})$ be a path witnessing the reachability. Starting from the first edge, we construct four consecutive connections for each edge from s_i to s_{i+1} ($i \in [1, n-1]$), such that they form a temporal path in G. However, the constructed path may not start with u and end with u'. To solve these, we extend the path with two more connections to link u and u'. □

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 of vertices in G. Observe that (a) the graph $\mathcal{G}_G = (\mathcal{V}, \mathcal{E})$ has at most $3d_{\max}|G|$ many vertices and $3d_{\max}|G|$ many edges; and (b) it takes $O(\log |G|)$ time to sort the edges.

5 INDEX CONSTRUCTION

In this section, we present a contraction-based index schema. We first present a simple partition strategy (Section 5.1), then show how to construct intra-fragment and inter-fragment indices by contracting graphs, *i.e.,* reducing the sizes of graphs (Section 5.2), and finally present the parallelization of these steps (Section 5.3).

5.1 Graph Partitioning

In this section, we partition the graph into multiple fragments.

Partition. Given a number k, graph $\mathcal{G}$=($\mathcal{V}, \mathcal{E}$) is partitioned into k fragments $F_1, F_2, \ldots, F_k$, where (1) F_i=($\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i$) is a fragment; (2) $\mathcal{V}$=$\cup_{i\in\{1,\ldots,k\}} \mathcal{V}_i$; (3) $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ for $i \neq j$; (4) IN_i (resp. OUT_i) is the set of vertices $v \in \mathcal{V} \setminus \mathcal{V}_i$ that satisfy the following conditions: (a) v is not a vertex in fragment F_i; and (b) v has an edge linking to a vertex in V_i (resp. from a vertex in V_i); and (5) $\mathcal{E}_i$ consists of all edges between vertices in $\mathcal{V}_i \cup \mathsf{IN}_i \cup \mathsf{OUT}_i$. Vertices in $\mathsf{IN}_i \cup \mathsf{OUT}_i$ are called *border vertices.* Such partitions are called edge-cut [8], *i.e.,* each vertex in $\mathcal{V}_i$ has all its adjacent edges in F_i as in $\mathcal{G}$.

A partition is ε-balanced, if $|\mathcal{V}_i| \leq (1+\varepsilon)\frac{|\mathcal{V}|}{k}$ for each fragment $F_i = (\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i)$ [8]. Here, the parameter $\varepsilon \geq 0$ is the tolerance threshold for the imbalance of a partition.

Multiple algorithms have been developed for edge-cut partitions [4, 8, 47]. Given that traject graph $\mathcal{G}$ contains temporal information and there exist orders among edges, we exploit a simple greedy strategy to group vertices pertaining to close timestamps [19]. More specifically, it (1) first sorts vertices based on the timestamps on its adjacent edges, and then (2) iteratively assigns these vertices to fragments following this order, *i.e.,* for each vertex $v \in \mathcal{V}$, if F_i is not full, *i.e.,* $|\mathcal{V}_i| \leq (1+\varepsilon)\frac{|\mathcal{V}|}{k}$, then assign v to F_i; otherwise, it assigns v to another new fragment, *i.e.,* F_{i+1}. We repeat these steps until all vertices have been assigned.

Example 5.1. Figure 4 illustrates a partition of traject graph $\mathcal{G}_G$ in Figure 3. Following the greedy strategy, $\mathcal{G}_G$ is partitioned into three fragments F_1, F_2 and F_3, where the border vertices of each fragment are marked by green and red dash circles, denoting border vertices in IN_i and OUT_i, respectively. □

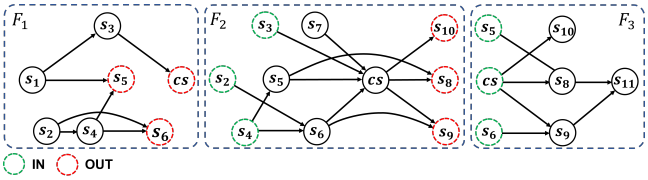


Figure 4: A partition $\{F_1, F_2, F_3\}$ of $\mathcal{G}_G$ given in Figure 3.

Reachability queries. After the graph has been partitioned, the temporal reachability queries can be divided into two categories:

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

Intra-fragment indices. To answer such reachability queries, we first preprocess each fragment using state-of-the-art indexing strategies, *e.g.,* pathtree [29], grail [59] and ferrari [37], we plug-in existing strategies to construct indices (see Section 6). In the following, we focus on the indexing for inter-fragment queries.

5.2 Inter-fragment indexing

We compute a contraction graph from the traject graph for inter-fragment indices. Intuitively, the contraction graph is constructed by merging vertices that can reach or be reached from the same set of border nodes [20]. As a result, the size of the graph is reduced.

Equivalent Sets. We first define equivalent relations among vertices in the traject graph, to group vertices that can reach or be reached from the same set of border nodes.

More specifically, given a fragment $F_i = (\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i)$, we define two equivalent relations EOut_i and EIn_i on $\mathcal{V}_i$, called *out-equivalent relation* and *in-equivalent relation*, respectively, as follows: 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 can reach the same set of border vertices in OUT_i as v can. (2) To define the in-equivalence class of v, let V_i^e be the set of vertices in $\mathcal{V}_i$ that have incoming edges from border vertices in IN_i; the *in-equivalence class* $[v]_{\mathsf{EIn}_i}$ in EIn_i is the set of vertices in $\mathcal{V}_i$ that can be reached from the same set of vertices in V_i^e that can reach v. Here, $[v]_{\mathsf{EIn}_i}$ checks vertices in V_i^e rather than border vertices as $[v]_{\mathsf{EOut}_i}$, since the border vertices of fragment F_i are in the set V_j^e of fragment F_j, which has a common edge with F_i. Such difference is used in the algorithm for reachability queries (see Section 6).

Example 5.2. Consider fragment F_2 in Figure 4. (1) Since all the vertices in F_2 can reach border vertics s_8, s_9 and s_{10}, we have that $[s_i]_{\mathsf{EOut}_2}$=$[cs]_{\mathsf{EOut}_2}$=$\{s_5, s_6, s_7, cs\}$, $i \in \{5, 6, 7\}$. (2) Firstly, we have V_2^e=$\{s_5, s_6, cs\}$, since these three vertices have incoming edges from border vertices in IN_2. Because s_5 and s_6 both can reach cs and themselves, and cs only can reach itself, we get the in-equivalent classes $[s_5]_{\mathsf{EIn}_2}$=$\{s_5\}$, $[s_6]_{\mathsf{EIn}_2}$=$\{s_6\}$, and $[cs]_{\mathsf{EIn}_2}$=$\{cs\}$. So, EOut_2 = $\{[s_5]_{\mathsf{EOut}_2}\}$ and EIn_2 = $\{[s_5]_{\mathsf{EIn}_2}, [s_6]_{\mathsf{EIn}_2}, [cs]_{\mathsf{EIn}_2}\}$. □

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 the set B_i of border vertices in EOut_i without outgoing edges;
2 set $\mathsf{Bout}_i[v] = \{v\}$ for all $v \in B_i$;
3 **while** $\mathsf{B}_i \neq \emptyset$ **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 EOut_i;

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

To solve this, we develop Algrithm 1, to compute the out-equivalence relation EOut_i, by revising the algorithm for topological sorting [15] and iteratively computing EOut_i by removing vertices without outgoing edges. Given a fragment F_i, it first identifies a set B_i of border vertices in OUT_i that have no outgoing edges (line 1); such border vertices always exist since F_i is a DAG. And it constructs sets $\mathsf{Bout}_i[v] = \{v\}$ for each border vertex v in B_i (line 2). After that it iteratively computes EOut_i (lines 3-9). Starting from a border vertex v in set B_i (line 3), it first updates the set $\mathsf{Bout}_i[u]$ for each incoming neighbor u of v (line 5-6). After that, it removes the edge from u to v in fragment F_i (line 7), and checks whether u has no outgoing edge (line 8); if so, it adds u to the set B_i (line 9). The iteration proceeds until there exists no vertex in B_i (line 3). At last, it groups vertices in $\mathcal{V}_i$ based on the sets Bout_i that they can reach (line 10), *i.e.,* two vertices v_j and v_k are in the same group if the sets $\mathsf{Bout}_i[v_j]$ and $\mathsf{Bout}_i[v_k]$ are equal.

Similarly, we can construct equivalent relations EIn_i. Denote by $\mathsf{Bin}_i[v]$ the set of vertices in V_i^e that can reach v.

Analysis. It is readily to verify the correctness of the algorithm. Algorithm OutEquivalentSet runs in $O(|\mathcal{E}_i||\mathsf{OUT}_i|)$ time, since (1) each edge is accessed only once (lines 5 and 7), and there exist $O(|\mathcal{E}_i|)$ many edges; (2) after removing an edge, 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), which takes $O(|\mathcal{V}_i||\mathsf{OUT}_i|)$ time.

Contraction graphs. To facilitate inter-fragment queries, we compute contraction graphs to encode connections between fragments.

Consider a traject graph $\mathcal{G}_G = (\mathcal{V}, \mathcal{E})$ and a partition $\{F_1, F_2, \ldots, F_k\}$ of $\mathcal{G}_G$, where $F_i = (\mathcal{V}_i, \mathcal{E}_i, \mathsf{IN}_i, \mathsf{OUT}_i)$ with equivalent classes EIn_i and EOut_i ($i \in \{1, ..., k\}$). We define a *contraction graph* $\mathcal{G}_G^c = (\mathcal{V}^c, \mathcal{E}^c)$ as follows: (1) $\mathcal{V}^c = \bigcup_{i\in[1,k]}(\mathsf{EIn}_i \cup \mathsf{EOut}_i \cup \mathsf{BIn}_i \cup \mathsf{BOut}_i)$, which consists of all equivalent classes of all fragments; here BIn_i and BOut_i contain sets of borders vertices that can reach and be reached from some vertex in F_i, respectively; (2) $\mathcal{E}^c$ consists of edges connecting border vertices; more specifically, the edges are constructed as follows: for each vertex $v \in \mathcal{V}$, (a) add an edge from its out equivalence class $[v]_{\mathsf{EOut}_i}$ to the set $\mathsf{Bout}_i[v]$ of border vertices that v can reach; (b) add an edge from $\mathsf{Bout}_i[v]$ to $\mathsf{Bin}_j[v']$, if they share a common vertex, *i.e.,* $\mathsf{Bout}_i[v] \cap \mathsf{Bin}_j[v'] \neq \emptyset$; and (c) add an edge from the set $\mathsf{Bin}_i[v]$ to the in-equivalence class $[v]_{\mathsf{EIn}_i}$ in fragment F_i, if $\mathsf{Bin}_i[v]$ is the set of vertices in V_i^e that can reach v.

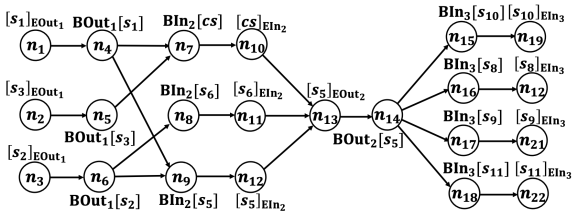


Figure 5: Contraction Graph $\mathcal{G}_G^c$

Example 5.3. We construct the contraction graph $\mathcal{G}_G^c$ (see Figure 5) based on the partition of traject graph $\mathcal{G}_G$ in Figure 4. (1) $[s_1]_{\mathsf{EOut}_1}$ can reach $\mathsf{Bout}_1[s_1]$, because s_1 can reach all vertices in $\mathsf{Bout}_1[s_1]$, *i.e.,* snapshots s_5 and cs; (2) $\mathsf{Bout}_1[s_1]$ can reach $\mathsf{Bin}_2[s_{12}]$, because $\mathsf{Bout}_1[s_1] = \{s_5, cs\}$, $\mathsf{Bin}_2[cs] = \{s_5, s_6, s_7, cs\}$, and there exists a common snapshot cs in $\mathsf{Bout}_1[s_1]$ and $\mathsf{Bin}_2[cs]$; and (3) $\mathsf{Bin}_2[cs]$ can reach $[cs]_{\mathsf{EIn}_2}$, because $\mathsf{Bin}_2[cs]$ is the set of vertices in V_i^e that can reach cs. □

Analysis. Due to the constructions of equivalent classes, we can verify that given two snapshots s_1 and s_2, s_1 can reach s_2 in the traject graph $\mathcal{G}_G$ if and only if s_1 can reach s_2 in the contraction graph $\mathcal{G}_G^c$. Indeed, $\mathsf{Bout}_i[v]$ and $\mathsf{Bin}_i[v]$ are the sets of border vertices that can be reach and reach v in $\mathcal{G}_G$, respectively.

The contraction graph can be constructed in $O(\Sigma_{i\in[1,k]}|\mathcal{V}_i|)|\mathsf{Out}_i|$, since each vertex v in V_i has at most two edges: one edge linking to the vertex representing $\mathsf{Bout}_i[v]$, and the other edge from the vertex representing $\mathsf{Bin}_i[v]$.

5.3 Parallelization

When the graph is large, it is costly to construct the indices [49]. To solve this, we develop a parallel algorithm to construct the indices. Observe that the above construction consists of the following four steps: (1) partition the graph based on timestamps; (2) construct the intra-fragment index in each fragment using existing algorithms *independently*; (3) compute the equivalent sets for vertices in each fragment *independently*; and (4) construct the contraction graphs, which can be done by linking each vertex *independently*. Since steps (2)-(4) can be conducted *independently*, they can be parallelized. For step (1), it can be done by a linear scan of all edges after sorting all edges based on their timestamps, and it takes less time than the other three steps (see the cost breakdown in Section 7).

6 QUERY ON CONTRACTION GRAPHS

We develop algorithm TRPchecker for the temporal bigraph reachability problem using indices presented in Section 5.

Overview. To answer the reachability problem in partitioned

Algorithm 2: TRPChecker

Input: a Traject Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and two 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 **return** CBFS(v_1, v_2, $\mathcal{G}$);

graphs, we need to address the following challenges: (1) the inputs of the problem are two vertices in the bipartite graph and a time interval; how can we transform the reachability problem in bipartite graph to the one in contraction graphs? (2) the source node and the target node may be in different fragments; how can we answer the problem if they are in different fragments?

We solve these problems as follows.

(1) Given two vertices u and v, and time interval $[t_s, t_e]$, we first identify the sets S_u^f and S_u^l of snapshots in the contraction graph such that the reachability relation is preserved, *i.e.,* u can reach v in time interval $[t_s, t_e]$ if and only if any snapshot in S_u^f can reach one snapshot in S_u^l in the contraction graph (see Section 4).

Given snapshots $s_1 \in S_u^f$ and $s_2 \in S_u^l$, we first check whether s_1 and s_2 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.,* pathtree [29], grail [59], and ferrari [37], and we can use such indices for the problem.

(2) When s_1 and s_2 are in different fragments, we conduct a breadth first search on the contraction graph to check the reachability. More specifically, it (a) first identifies fragments f_u and f_v, where s_1 and s_2 are located, respectively; (b) then in fragment f_u, it collects the border nodes that s_1 can reach, by using the set $\mathsf{Bout}_i[u]$; (c) after that, it identifies the fragments, where vertices in $\mathsf{Bout}_i[s_1]$ are located; (d) it repeats steps (b) and (c), until it finds fragment f_v; let W_{s_1} be the set of all borders nodes in f_v that u can reach; and finally, (c) it computes whether W_{s_1} and $\mathsf{Bin}_i[s_2]$ have a common vertex, *i.e.,* whether $\mathsf{W}_u \cap \mathsf{Bin}_i[s_2] = \emptyset$; here $\mathsf{Bin}_i[s_2]$ is the set of vertices that can reach s_2 in fragment f_v.

Algorithm. We present TRPChecker in Algorithm 2. Given two vertices u_1 and u_2, and a traject graph $\mathcal{G}$, it first checks whether u_1 and u_2 are in the same fragment; if so, it exploits the intra-partition indices to check whether u_1 can reach u_2 (line 2). If u_1 and u_2 are in different fragments, it checks the reachability via procedure CBFS.

Procedure CBFS. Given two vertices u_1 and u_2, and a traject graph $\mathcal{G}$, it checks whether u_1 can reach u_2 via the BFS-search. At first, it identifies the fragments F_1 and F_2, in which u_1 and u_2 are located (line 1). Recall that the graph G is partitioned via edge-cut, and each vertex is located in only one fragments. After that it fetches the border vertices of F_1, which u_1 can reach, via procedure Border (line 2). This can be done by directly accessing the equivalent set $[u_1]_{\mathsf{EOut}_i}$. Then it computes the fragments reached by following one edge via ReachFrag. This can be done by visiting the edges in the contraction graph, since the border vertices are the vertices existing in other fragments. After that it repeats these steps until it finds fragment F_2, in which u_2 is located (lines 4-11). Given F_2,

Algorithm 3: CBFS

Input: a Traject Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and two vertices v_1 and v_2.
Output: Whether v_1 can reach v_2?

1 (F_1, F_2) := Locate(v_1, v_2, $\mathcal{G}$);
2 V_r := Border(v_1, F_1, $\mathcal{G}$); F_r := ReachFrag(V_r, $\mathcal{G}$);
3 push F_r into FQueue;
4 **while** FQueue $\neq \emptyset$ **do**
5 F_r' := FQueue.pop();
6 **if** $F_r' = F_2$ **then**
7 **break** ;
8 V_r' := Border($F_r \cap V_r$, $\mathcal{G}$);
9 $V_r := V_r \cup V_r'$;
10 F_r' := ReachFrag(V_r', $\mathcal{G}$);
11 push F_r into FQueue;
12 **if** $F_r \neq \emptyset$ **then**
13 bReach := CheckReach(F_i, v_2);
14 **return** bReach;

it checks whether u_2 can be reached by directly checking if some vertices in V_r exists in the equivalent set $[u_2]_{\mathsf{EIn}_i}$, *i.e.,* the set of vertices in V_2 that can reach u_2 (line 13). Observe that V_r consists of all border vertices that v_1 can reach.

Example 6.1. Given the bipartite graph G in Figure 1 and the contraction graph in Figure 4, 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 $Q(u_1, u_4, G, [3, 9])$, observe that $S_{u_1}^f = \{[4, 7]_{v_1}\} = \{s_3\}$ and $S_{u_4}^l = \{[8, 8]_{v_3}\} = \{s_{11}\}$, which are located in different fragments. Observe that (a) s_3 belongs to $[s_3]_{\mathsf{EOut}_1}$ and s_{11} belongs to $[s_{11}]_{\mathsf{EIn}_3}$; and (b) $[s_3]_{\mathsf{EOut}_1}$ can reach $[s_{11}]_{\mathsf{EIn}_3}$ in the contraction graph $\mathcal{G}_G^c$. Therefore, u_1 can reach u_4 in time interval $[3, 9]$. □

Analysis. To analyze TRPChecker, consider the following two cases.

(1) When the given vertices are in same fragments, the complexity of the query depends on the selected algorithm to construct intra-fragment indices. For example, if we pick pathtree [29], then answering the query takes $O^2(\log n)$ time.

(2) Given two vertices u and v in different fragments, it is in $O(\sum_{i=[1,k]}(|\mathsf{BIn}_i| + |\mathsf{BOut}_i|))$ time to determine whether u can reach v. Here, (a) k is the number of fragments, (b) $|\mathsf{BIn}_i|$ is the number of different sets of border vertices that can reach vertices in in-equivalence class; and (c) $|\mathsf{BOut}_i|$ are the numbers of different sets of border vertices that vertices in out-equivalence classes can reach.

There exist three key steps in CBFS: (1) given a vertex v, compute the set of that v can reach (line 8). This step is in $O(|\mathsf{BOut}_i|)$ time, since it can be done by accessing the set BOut_i, the size of which is bounded by $O(|\mathsf{BOut}_i|)$. (2) Given a set V_r' of vertices, compute the set of fragments, where the vertices in V_r' are located (line 10). This can be done by one scan of vertices in V_r', the number of which is also bounded by $O(|\mathsf{BOut}_i|)$. (3) Given a set V_r' of vertices, check whether V_r' can reach v in a fragment (line 13). This can be done by accessing the set BIn_i, which is bounded by $|\mathsf{BIn}_i|$. As shown

Table 3: Dataset Summary

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

in Section 7, $O(\sum_{i=[1,k]}(|\mathsf{BIn}_i| + |\mathsf{BOut}_i|))$ is small compared with the size of the original graph.

7 EXPERIMENTS

Using real-life and synthetic data, we evaluated (1) efficiency (2) effectiveness and (3) scalability of our algorithms.

Experimental setting. We start with the setting.

Graphs. We used eleven real-life graphs and a synthetic graph in our expriments, which are listed in Table 3. Here, Name is the abbreviation of the dataset, $|E|$ is the number of edges, $|U|$ and $|V|$ are the numbers of top and bottom vertices in the graphs, respectively.

All real-life graphs are from the Konect dataset[1]. However, the edges carry only one timestamp. To generate time intervals for edges, we used this timestamp as starting time, and generate the ending time by simulating the interval length, following the power-law distribution $p(\Delta t) = C\tau^{-\alpha}$ [5], where C are set from 10 minutes to 8 hours, and α is -2.5.

We generated synthetic graph by conducting random walks on graphs. More specifically, we first generated a graph representing connections between bottom vertices, then conducted random walks on this graph to simulate "movements" of top vertices. Timestamps are added following the same power-low distribution above.

Queries. For each dataset, we randomly generated one million vertex pairs, and used the average query time to evaluate query algorithms.

Algorithms. We evaluated the following algorithms.

Index construction algorithms. (1) $\mathsf{CBTR}_{\mathsf{In}}$, the proposed algorithm in Section 4 and Section 5, which consists of both graph transformation and index construction, and is implemented in C++; the intra-fragment indices are constructed by the state-of-the-art indexing strategy **pathTree** as a case study. Considering the number of logical cores in the server, we set the partition number to 64 if a graph can be processed using 1TB memory. For EJ, the number is set to 2048; (2) $\mathsf{CBTR}_{\mathsf{InD}}$, a variant of $\mathsf{CBTR}_{\mathsf{In}}$ that does not compute the equivalent classes and directly constructs the set $\mathsf{Bout}_i[u]$ for each vertex. The number of partitions is set to be the same as $\mathsf{CBTR}_{\mathsf{In}}$; (3) $\mathsf{CBTR}_{\mathsf{InP}}$, a parallel version of $\mathsf{CBTR}_{\mathsf{In}}$ using 64 threads; (4) TBP [12], a 2-hop based index for temporal bipartite graphs; and (5) WTB [36], a wedge-based index for dynamic temporal bipartite graphs. We used the implementations of TBP and WTB provided in [36].

Query algorithms. (6) $\mathsf{TRPChecker}_{\mathsf{In}}$, the query algorithm in Section 6, which is implemented in C++; (7) $\mathsf{TRPChecker}_{\mathsf{InD}}$ is the query algorithm in Section 6, except that indices are constructed by $\mathsf{CBTR}_{\mathsf{InD}}$. (8) The query algorithms of TBP and WTB from [12, 36].

Note that the definitions of reachability in [12, 36] are different from ours (see Example 2.2). To make algorithms comparable, we modified their initial codes to make them work under our definition. Moreover, timestamps on edges are cut into multiple small intervals, to make paths between two vertices consistent in all algorithms.

Environment. We run experiments on a Linux server with Intel Xeon Platinum 8358 CPU (2.6GHz) and 1TB memory. It has 2 CPUs, each of which has 64 logical cores. Each experiment was run 5 times, and the average is reported.

[1] https://github.com/kunegis/konect-extr

Table 4: Average indexing time(ms)

Name	TBP	WTB	$\mathsf{CBTR}_{\mathsf{In}}$	$\mathsf{CBTR}_{\mathsf{InD}}$	$\mathsf{CBTR}_{\mathsf{InP}}$
WP	109,031	67	5,444	4,725	2,657
SO	3,749,892	28,833	12,225	75,817	8,792
LK	870,997	1,231	48,106	31,382	13,665
CU	-	32,478	417,415	-	44,319
BS	8,603,007	407,603	173,943	-	40,311
TW	-	-	778,810	-	108,536
AM	6,921,445	93,604	60,772	1,293,789	44,292
EP	-	231,846	4,436,361	-	617,062
LF	2,162,083	17,315	2,304,292	2,572,726	412,315
EJ	67,450,178	30,788	14,826,991	-	1,232,355
ED	-	-	16,678,018	-	2,429,938

Table 5: Indexing size

Name	TBP	WTB	$\mathsf{CBTR}_{\mathsf{In}}$	$\mathsf{CBTR}_{\mathsf{InD}}$
WP	29,694	58,812	24,709,072	24,759,195
SO	58,034,094	63,089,346	6,807,106	6,813,578
LK	7,660,085	2,734,722	274,654,743	278,751,204
CU	-	308,508,570	1,311,651,567	-
BS	741,216	1,570,026	2,874,424,723	-
TW	-	-	2,369,271,501	-
AM	96,636,521	199,767,234	24,791,994	23,045,658
EP	-	767,154,144	57,947,953,182	-
LF	12,746,629	934,350	25,707,997,549	25,748,372,419
EJ	96,686,256	12,027,648	116,428,076,208	-
ED	-	-	51,731,241,669	-

Table 6: Memory(GB)

Name	TBP	WTB	$\mathsf{CBTR}_{\mathsf{In}}$	$\mathsf{CBTR}_{\mathsf{InD}}$	$\mathsf{CBTR}_{\mathsf{InP}}$
WP	0.171	0.076	1.19	1.18	0.077
SO	2.64	1.83	2.66	2.56	1.65
LK	0.52	0.47	5.27	5.17	2.17
CU	-	7.48	20.66	-	22.06
BS	0.61	0.076	22.80	-	17.84
TW	-	-	38.64	-	35.84
AM	5.9	4.59	10.86	10.56	11.29
EP	-	18.6	357.02	-	361.09
LF	3.84	0.07	177.49	177.41	180.11
EJ	8.2	7.9	714.55	-	717.36
ED	-	-	636.36	-	643.07

Table 7: The average query time(μs)

Name	TBP	WTB	CBTR_{In}	CBTR_{InD}
WP	10.00	7.22	40.24	103.38
SO	4.07	18200.50	10.87	759.18
LK	23.74	234.20	19.24	464.20
CU	-	42,878.64	2,062.22	-
BS	32.28	179.18	8,782.87	-
TW	-	-	17,074.96	-
AM	1.90	39,394.52	90.37	5,067.02
EP	-	70,873.79	10,925.43	-
LF	43.61	54.05	1,400.88	25,180.09
EJ	62.65	1,768.75	2,533.76	-
ED	-	-	131,179.62	-

Table 8: The running time of CBTR_{In} (s)

Name	Transformation	Partition	Index construction
WP	1.26 (32.31%)	0.76 (19.49%)	1.88 (48.21%)
SO	4.33 (41.20%)	3.89 (37.01%)	2.29 (21.79%)
LK	6.29 (30.80%)	3.64 (17.83%)	10.49 (51.37%)
CU	9.18 (15.91%)	6.47 (11.22%)	42.04 (72.87%)
BS	11.97 (11.35%)	7.16 (6.79%)	86.31 (81.86%)
TW	21.18 (17.62%)	20.70 (17.22%)	78.35 (65.17%)
AM	20.97 (38.80%)	20.80 (38.48%)	12.28 (22.72%)
EP	91.09 (2.93%)	119.68 (3.85%)	2900.65 (93.23%)
LF	95.68 (6.93%)	67.70 (4.90%)	1217.39 (88.17%)
EJ	234.76 (2.39%)	227.89 (2.32%)	9359.60 (95.29%)
ED	842.06 (13.15%)	1001.96 (15.65%)	4558.58 (71.20%)

Experimental results. We next report our findings.

Exp-1: Index construction. We first evaluated the indexing time, index size and the memory usage of the indexing algorithms.

Indexing time. We first compared the indexing time of the index construction algorithms. The results are shown in Table 4, where '-' denotes that the algorithm cannot finish within 24 hours or run out of memory. We found the following: (1) On large scale graphs ED with $1M$ top vertices, $5M$ bottom vertices and $129M$ edges, only CBTR_{In} can construct the indices. On EJ, CBTR_{In} is 5× faster than TBP. And on the other smaller graphs, CBTR_{In} is far faster than TBP. On ED and EJ, WTB is faster than both CBTR_{In} and TBP. However, the query time of indices generated via WTB is much longer than that of indices computed by CBTR_{In} and TBP. (2) Even CBTR_{InD} does not compute equivalent classes, it is still slower than CBTR_{In} on large scale graphs. This is because on these graphs, the potential connections between border vertices are dense, and computing these connections is time-consuming. (3) The parallel algorithm CBTR_{InP} is efficient. Using 64 threads, CBTR_{InP} is 7× faster than CBTR_{In}. This is because the intra-fragment indices can be constructed in parallel, which is the most time-consuming part on large scale graphs.

Index size. Table 5 reports the index sizes of these algorithms. We found the following: (1) The number of indices generated via CBTR_{In} is larger than that of TBP and WBP, since the lower bound of number of indices that CBTR_{In} constructs on the transformed graphs is limited by the bound of PathTree which is expected to be $O(nk)$ [29], where n is the vertex number of input dag, and k is the number of paths constructed from the DAG. (2) Compared to CBTR_{In}, CBTR_{InD} has a much denser contraction graph without the compression by equivalent classes, and therefore the number of indices generated via CBTR_{InD} is larger than that of CBTR_{In}.

Memory usage. Table 6 reports the memory used by the algorithms. We found the following: (1) On large scale graphs EJ and ED, CBTR_{In} exploits the memory to accelerates the index construction. Although CBTR_{InD} uses rather little memory, it cannot terminate within 24 hours. (2) On some median scale graphs like TW, CBTR_{In} uses much smaller memory, *i.e.,* 10.86GB memory.

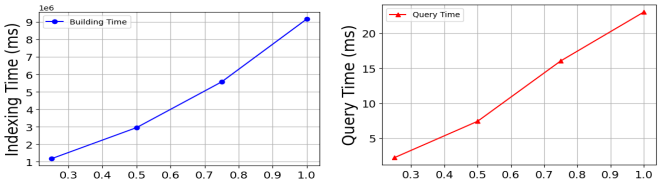
Cost breakdown. We conducted the cost breakdown analysis of the running time of CBTR_{In}, which is broken down into three phases: graph transformation, graph partition and index construction. The results are shown in Table 8. We found the following. (1) When the graphs get larger, the running times of all three steps increase as expected, since transformed graphs get larger. (2) The time for index construction dominates the computation, *e.g.,* on EJ and LF over 90% of the time is spent to construct the indices. (3) The greedy algorithm for graph partition is efficient. It takes less than 20% of the running time. Actually, on large graphs like EP, LF and EJ, it can account for fewer than 5% of the total time.

Varying the number of fragments. We evaluated the performance of CBTR_{In} and $\text{TRPChecker}_{\text{In}}$ on BS and AM, by varying the number of fragments from 1 to 512. The results are shown in Table 9, where IT, IS, MU and QT represent indexing time, index size, memory usage and query time, respectively. We found the following. (1) When the number of fragments increases, CBTR_{In} is 11× faster when the number of fragments varies from 8 to 512 for BS, and 7× faster when the number of fragments varies from 1 to 512 for AM. (2) When the number of fragments increases, CBTR_{In} constructs fewer indices. This is because the number of vertices in each fragment decreases, and the indices constructed by PathTree get smaller. Note that the indices constructed by PathTree is in the size of $O(nk)$ [29], which also dominates the index size. (3) When the number of fragments increases, the memory usage decreases due to the reason given above. However, on graph AM when the number of fragments is too large, the memory usage will increase, since the number of equivalent classes will increase, and the algorithm use more memory to store the inter-fragment information. (4) On graph BS, when the number of fragments increases, the query time first decreases and then increases. This is expected since the number of fragments strikes a balance between the size of intra-fragment indices and the size of inter-fragment indices.

Exp-2: Query processing. We next evaluated the performance of the query algorithms. The results are shown in Table 7. We found the following: (1) On large scale graph ED, only $\text{TRPChecker}_{\text{In}}$ can generate the indices, and conduct the queries. (2) On median scale graphs, for AM, $\text{TRPChecker}_{\text{In}}$ outperforms WBP by over 400× and for TW, WTB cannot even finish the index construction. However, on graphs LF and BS, WTB is on average 37× faster than $\text{TRPChecker}_{\text{In}}$. This is because the temporal bigraph is extremely asymmetric, *i.e.,* the number of top vertices is much smaller than the number of bottom vertices, on which WBP can efficiently run. (3) On small graphs like WP, SO and LK, the performance of $\text{TRPChecker}_{\text{In}}$ is similar to that of WTP, as expected, since the transformed graphs are sparse. (4) On average, $\text{TRPChecker}_{\text{In}}$ outperforms $\text{TRPChecker}_{\text{InD}}$ by 34×, since using vertices in the original graphs lead to dense graphs, which degrades the query processing.

Table 9: The impact of fragment number on four metrics for two datasets

Name	Metric	Fragment Number									
		1	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
BS	IT(s)	-	-	-	889.91	436.92	253.44	176.09	137.84	112.10	81.63
	IS($\times 10^9$)	-	-	-	21.26	11.12	5.60	2.87	1.48	0.76	0.39
	MU(GB)	-	-	-	118.85	66.09	37.83	22.80	15.34	11.68	10.01
	QT(ms)	-	-	-	10.74	9.19	8.91	8.93	9.07	9.22	9.39
AM	IT(s)	381.92	274.19	145.96	88.20	66.63	60.64	58.53	58.188	57.620	58.746
	IS($\times 10^8$)	43.84	10.57	2.11	0.48	0.26	0.24	0.24	0.25	0.27	0.28
	MU(GB)	31.13	14.13	10.99	10.06	10.51	10.67	10.87	11.06	12.05	12.37
	QT(μs)	7.53	4.75	9.40	22.69	34.93	54.39	83.31	131.95	192.36	261.18



(a) Varying $|G|$ (Indexing time) (b) Varying $|G|$ (Query time)

Figure 6: Performance evaluation

Exp-3: Scalability. We tested the scalability of our algorithms.

Varying $|E|$. We varied the scale factor from 0 to 1 with a step length 0.25, to evaluate the scalability of $\mathsf{CBTR}_{\mathsf{In}}$. The initial bigraph has $1.5M$ top vertices, $10K$ bottom vertices and $234M$ edges. The fragment number is set to be 4096.

Indexing time. As shown in Figure 6(a), (1) when G gets larger, $\mathsf{CBTR}_{\mathsf{In}}$ takes longer, as expected. (2) $\mathsf{CBTR}_{\mathsf{In}}$ scales well with $|G|$. On average, it takes 9, 161 seconds to construct the indices.

Querying time. The results are reported in Figure 6(b). We found the following: (1) when $|G|$ gets larger, the query time of $\mathsf{CBTR}_{\mathsf{In}}$ increases, as expected. Note that for different graphs, the query is regenerated following the time interval, and the query time is the average of $1M$ queries. (2) $\mathsf{CBTR}_{\mathsf{In}}$ scales well with $|G|$. It takes 23ms to process a query on G with $1.5M$ vertices and $234M$ edges.

Memory. We evaluated the memory usage of $\mathsf{CBTR}_{\mathsf{In}}$. We found the following (not shown): (1) when $|G|$ gets larger, more memory is used, as expected. (2) $\mathsf{CBTR}_{\mathsf{In}}$ scales well with $|E|$. it takes $728GB$ to process a graph with $1.5M$ vertices and $234M$ edges.

Exp-4: Connection latency. We next studied the impact of a parameter, namely latency. Observe 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 for some top vertex u if $s_1.\mathsf{emin}[u] \leq s_2.\mathsf{smax}[u]$ However, when t_s^2 is sufficiently larger than t_e^1, *e.g.*, t_s^2 is one year after t_e^1, the edge from s_1 to s_2 is useless for the reachability problem, especially for disease tracing, in which the disease has a much shorter latency. Then the latency is defined as the maximum time span that two snapshots can be linked, *i.e.*, two snapshots $s_1 = [t_s^1, t_e^1]_{v_1}$ and $s_2 = [t_s^2, t_e^2]_{v_2}$ cannot have an edge if $|t_e^1 - t_s^2| \geq \delta$ or $|t_e^2 - t_s^1| \geq \delta$.

We next evaluated the latency, denoted by δ, on the performance of $\mathsf{CBTR}_{\mathsf{In}}$. The results are shown in Table 10. We found the following. (1) When δ gets larger, the number of edges in the traject graphs increases, as expected. Therefore, both $\mathsf{CBTR}_{\mathsf{In}}$ and $\mathsf{TRPChecker}_{\mathsf{In}}$ get slower. (2) The latency has different impacts on $\mathsf{CBTR}_{\mathsf{In}}$ for different graphs. On CU, $\mathsf{CBTR}_{\mathsf{In}}$ is 327× faster when δ varies from 32 to 1. This is because graph CU models the relations between users and papers that they read, and is not "updated" frequently. However, on AM, $\mathsf{CBTR}_{\mathsf{In}}$ is only 3× faster, with the same variation. This is because each user (*i.e.*, top vertex) may visit Amazon every day. The latency has a slight impact on the sizes of traject graphs.

Table 10: The impact of δ on four metrics

Name	Metric	δ(day)					
		1	**2**	**4**	**8**	**16**	**32**
SO	IT(s)	9.29	9.53	9.76	10.10	10.58	11.29
	IS($\times 10^6$)	4.05	4.15	4.32	4.64	4.99	5.38
	MU(GB)	1.45	1.38	1.54	1.56	1.52	1.52
	QT(μs)	1.80	1.81	1.84	1.92	2.23	3.20
LK	IT(s)	22.06	26.03	30.26	35.38	40.17	44.19
	IS($\times 10^8$)	1.04	1.75	2.21	2.49	2.64	2.70
	MU(GB)	4.23	4.57	4.82	4.98	5.07	5.17
	QT(μs)	10.78	11.15	11.43	11.88	13.02	15.62
Cu	IT(s)	74.07	108.11	154.39	221.36	281.62	325.49
	IS($\times 10^9$)	0.54	0.75	0.92	1.06	1.15	1.21
	MU(GB)	7.49	8.68	9.83	11.54	13.7	15.73
	QT(μs)	4.41	31.12	107.9	335.5	889.56	1441.03
AM	IT(s)	53.07	53.85	54.68	56.35	57.81	60.22
	IS($\times 10^7$)	1.81	1.81	1.82	1.85	1.89	1.96
	MU(GB)	8.36	8.02	7.84	7.47	9.46	9.35
	QT(μs)	1.96	1.98	1.96	2.01	2.46	5.88

Summary. We found the following: (1) $\mathsf{CBTR}_{\mathsf{In}}$ is efficient. On BS and AM, it is 50× and 113× faster than TBP, respectively. (2) $\mathsf{CBTR}_{\mathsf{In}}$ can handle large graphs like ED, which cannot be processed by the state-of-the-art algorithm. (3) $\mathsf{TRPChecker}_{\mathsf{In}}$ is faster than WTB. On a graph with $876K$ vertices and $13M$ edges, on average $\mathsf{CBTR}_{\mathsf{In}}$ is 6.5× faster than WTB. (4) $\mathsf{CBTR}_{\mathsf{InP}}$ scales well with large graphs. On ED with $7M$ vertices and $129M$ edges, $\mathsf{CBTR}_{\mathsf{InP}}$ takes only less than 5 hours to construct indices.

8 CONCLUSION

We proposed an approach to answer the reachability problem on the temporal bipartite graphs. The novelty of the work consists of the following: (1) a graph transformation method that converts temporal bipartite graphs into DAGs; (2) a hierarchy indexing schema, to handle large-scale graphs; and (3) a two-stage query algorithm for the span-reachability query. Experiments show that the proposed methods are promising in practice.

One topic for future work is to reduce the number of vertices in the traject graphs. Another topic is to develop an effective graph partition algorithm, to reduce the number of border nodes.

REFERENCES

[1] 2024. Full version. *https://github.com/dengt2000/TRPfull/blob/main/TRPfull.pdf*.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2012. Hierarchical hub labelings for shortest paths. In *ESA*. 24–35.

[3] Shorouq Al-Eidi, Yuanzhu Chen, Omar Darwishand, and Ali MS Alfosool. 2020. Time-ordered bipartite graph for spatio-temporal social network analysis. In *ICNC*. 833–838.

[4] Konstantin Andreev and Harald Räcke. 2006. Balanced Graph Partitioning. *Theory Comput. Syst.* 39, 6 (2006), 929–939.

[5] Albert-Laszlo Barabasi. 2005. The origin of bursts and heavy tails in human dynamics. *Nature* 435, 7039 (2005), 207–211.

[6] Idir Benouaret and Sihem Amer-Yahia. 2020. A Comparative Evaluation of Top-N Recommendation Algorithms: Case Study with Total Customers. In *IEEE BigData*. 4499–4508.

[7] Smriti Bhagat, Graham Cormode, Balachander Krishnamurthy, and Divesh Srivastava. 2009. Class-Based Graph Anonymization for Social Network Data. *Proc. VLDB Endow.* 2, 1 (2009), 766–777.

[8] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *KDD*. 1456–1465.

[9] Arnaud Casteigts, Timothée Corsini, and Writika Sarkar. 2024. Simple, strict, proper, happy: A study of reachability in temporal graphs. *Theor. Comput. Sci.* 991 (2024), 114434.

[10] Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. 2020. Finding Temporal Paths Under Waiting Time Constraints. In *ISAAC*, Vol. 181. 30:1–30:18.

[11] Wenbin Chen, Andrea M. Rocha, William Hendrix, Matthew C. Schmidt, and Nagiza F. Samatova. 2010. The Multiple Alignment Algorithm for Metabolic Pathways without Abstraction. In *ICDMW*. 669–678.

[12] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2021. Efficiently answering reachability and path queries on temporal bipartite graphs. *Proc. VLDB Endow.* 14, 10 (2021), 1845–1858.

[13] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: a topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*. 193–204.

[14] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[15] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[16] Argyrios Deligkas and Igor Potapov. 2022. Optimizing reachability sets in temporal graphs by delaying. *Inf. Comput.* 285 (2022), 104890.

[17] Hongbo Deng, Michael R. Lyu, and Irwin King. 2009. A generalized Co-HITS algorithm and its application to bipartite graphs. In *SIGKDD*. 239–248.

[18] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. 2004. Modelling disease outbreaks in realistic urban social networks. *Nature* 429, 6988 (2004), 180–184.

[19] Wenfei Fan, Ruochun Jin, Ping Lu, Chao Tian, and Ruiqi Xu. 2022. Towards Event Prediction in Temporal Graphs. *Proc. VLDB Endow.* 15, 9 (2022), 1861–1874.

[20] Wenfei Fan, Yuanhao Li, Muyang Liu, and Can Lu. 2023. Making graphs compact by lossless contraction. *VLDB J.* 32, 1 (2023), 49–73.

[21] Luca Ferreri, Ezio Venturino, and Mario Giacobini. 2011. Do Diseases Spreading on Bipartite Networks Have Some Evolutionary Advantage?. In *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics - 9th European Conference, EvoBIO 2011, Torino, Italy, April 27-29, 2011. Proceedings (Lecture Notes in Computer Science)*, Clara Pizzuti, Marylyn D. Ritchie, and Mario Giacobini (Eds.), Vol. 6623. Springer, 141–146. https://doi.org/10.1007/978-3-642-20389-3_14

[22] Esra Gündoğan and Buket Kaya. 2017. A link prediction approach for drug recommendation in disease-drug bipartite network. In *IDAP*. 1–4.

[23] Kathrin Hanauer, Christian Schulz, and Jonathan Trummer. 2022. O'reach: Even faster reachability in large graphs. *JEA* 27 (2022), 1–27.

[24] Fang He, Feiping Nie, Rong Wang, Haojie Hu, Weimin Jia, and Xuelong Li. 2021. Fast Semi-Supervised Learning With Optimal Bipartite Graph. *TKDE* 33, 9 (2021), 3245–3257.

[25] Jingrui He and Rick Lawrence. 2011. A graphbased framework for multi-task multi-view learning.. In *ICML*. 25–32.

[26] Boon Hao Hong, Jane Labadin, W King Tiong, Terrin Lim, Melvin Hsien Liang Chung, et al. 2021. Modelling COVID-19 hotspot using bipartite network approach. *Acta Informatica Pragensia* 10, 2 (2021), 123–137.

[27] Wenyu Huo and Vassilis J. Tsotras. 2014. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*. 38:1–38:4.

[28] Zakwan Jaroucheh, Mohamad Alissa, William J. Buchanan, and Xiaodong Liu. 2020. TRUSTD: Combat Fake Content using Blockchain and Collective Signature Technologies. In *COMPSAC*. IEEE, 1235–1240.

[29] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-Tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs. *TODS* 36, 1, Article 7 (2011), 44 pages.

[30] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*. 813–826.

[31] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD*. 595–608.

[32] Bai Li, Allam Maalla, and Mingbiao Liang. 2021. Research on Recommendation Algorithm Based on E-commerce User Behavior Sequence. In *ICIBA*, Vol. 2. 914–918. https://doi.org/10.1109/ICIBA52610.2021.9688086

[33] Xiao Li, Ye-Yi Wang, and Alex Acero. 2008. Learning query intent from regularized click graphs. In *SIGIR*. 339–346.

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

[35] Florian Merz and Peter Sanders. 2014. PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies. In *ESA*, Vol. 8737. Springer, 701–712.

[36] Lohan Meunier and Ying Zhao. 2023. Reachability Queries on Dynamic Temporal Bipartite Graphs. In *SIGSPATIAL/GIS*. Article 97, 11 pages.

[37] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. 2013. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*. 1009–1020.

[38] Houtan Shirani-Mehr, Farnoush Banaei Kashani, and Cyrus Shahabi. 2012. Efficient reachability query evaluation in large spatiotemporal contact datasets. *arXiv preprint arXiv:1205.6696* (2012).

[39] Ashley G Smart, Luis AN Amaral, and Julio M Ottino. 2008. Cascading failure and robustness in metabolic networks. *Proceedings of the National Academy of Sciences* 105, 36 (2008), 13223–13228.

[40] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2016. Reachability querying: Can it be even faster? *TKDE* 29, 3 (2016), 683–697.

[41] Zhixiang Su, Di Wang, Xiaofeng Zhang, Lizhen Cui, and Chunyan Miao. 2022. Efficient Reachability Query with Extreme Labeling Filter. In *WSDM*. 966–975.

[42] Eugenio Tacchini, Gabriele Ballarin, Marco L. Della Vedova, Stefano Moret, and Luca de Alfaro. 2017. Some Like it Hoax: Automated Fake News Detection in Social Networks. *CoRR* abs/1704.07506 (2017). http://arxiv.org/abs/1704.07506

[43] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.

[44] Bezaye Tesfaye, Nikolaus Augsten, Mateusz Pawlik, Michael H. Böhlen, and Christian S. Jensen. 2022. Speeding Up Reachability Queries in Public Transport Networks Using Graph Partitioning. *Inf. Syst. Frontiers* 24, 1 (2022), 11–29.

[45] Suhas Thejaswi, Juho Lauri, and Aristides Gionis. 2020. Restless reachability problems in temporal graphs. *arXiv preprint arXiv:2010.08423* (2020).

[46] Silke Trißl and Ulf Leser. 2007. Fast and Practical Indexing and Querying of Very Large Graphs. In *SIGMOD*. 845–856.

[47] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: streaming graph partitioning for massive scale graphs. In *WSDM*. 333–342.

[48] Divya Lakshmi Venkatraman, Deepshika Pulimamidi, Harsh G Shukla, and Shubhada R Hegde. 2021. Tumor relevant protein functional interactions identified using bipartite graph analyses. *Scientific Reports* 11, 1 (2021), 21530.

[49] Kai Wang, Minghao Cai, Xiaoshuang Chen, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2024. Efficient algorithms for reachability and path queries on temporal bipartite graphs. *The VLDB Journal* (2024), 1–28.

[50] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In *SIGMOD*. 967–982.

[51] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability querying: An independent permutation labeling approach. *Proc. VLDB Endow.* 7, 12 (2014), 1191–1202.

[52] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently answering span-reachability queries in large temporal graphs. In *ICDE*. 1153–1164.

[53] Dong Wen, Bohua Yang, Ying Zhang, Lu Qin, Dawei Cheng, and Wenjie Zhang. 2022. Span-reachability querying in large temporal graphs. *VLDB J.* 31, 4 (2022), 629–647.

[54] Maresha Wijanto, Rachmi Rahmadiany, and Oscar Karnalim. 2020. Thesis Supervisor Recommendation with Representative Content and Information Retrieval. *JISEBI* 6 (10 2020), 143–150. https://doi.org/10.20473/jisebi.6.2.143-150

[55] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path Problems in Temporal Graphs. *Proc. VLDB Endow.* 7, 9 (2014), 721–732.

[56] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *ICDE*. 145–156.

[57] Haoxuan Xie, Yixiang Fang, Yuyang Xia, Wensheng Luo, and Chenhao Ma. 2023. On Querying Connected Components in Large Temporal Graphs. *Proc. ACM Manag. Data* 1, 2 (2023), 170:1–170:27.

[58] Weiqing Yan, Jindong Xu, Jinglei Liu, Guanghui Yue, and Chang Tang. 2022. Bipartite Graph-based Discriminative Feature Learning for Multi-View Clustering. In *MM*. 3403–3411.

[59] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *Proc. VLDB Endow.* 3, 1–2 (2010), 276–284.

[60] Changyong Yu, Tianmei Ren, Wenyu Li, Huimin Liu, Haitao Ma, and Yuhai Zhao. 2024. BL: An Efficient Index for Reachability Queries on Large Graphs. *IEEE*

Trans. Big Data 10, 2 (2024), 108–121.

[61] Jianke Yu, Xianhang Zhang, Hanchen Wang, Xiaoyang Wang, Wenjie Zhang, and Ying Zhang. 2023. FPGN: follower prediction framework for infectious disease prevention. *WWW* 26, 6 (2023), 3795–3814.

[62] Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph Reachability Queries: A Survey. In *Managing and Mining Graph Data*. Advances in Database Systems, Vol. 40. 181–215.

[63] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2023. An Overview of Reachability Indexes on Graphs. In *SIGMOD conference Companion (SIGMOD '23)*. 61–68.

[64] Tianming Zhang, Yunjun Gao, Lu Chen, Wei Guo, Shiliang Pu, Baihua Zheng, and Christian S. Jensen. 2019. Efficient distributed reachability querying of massive temporal graphs. *VLDB J.* 28, 6 (2019), 871–896.

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

[66] Junfeng Zhou, Shijie Zhou, Jeffrey Xu Yu, Hao Wei, Ziyang Chen, and Xian Tang. 2017. DAG Reduction: Fast Answering Reachability Queries. In *SIGMOD*. 375–390.

[67] Zhiguo Zhu, Jingqin Su, and Liping Kong. 2015. Measuring influence in online social network based on the user-content bipartite graph. *Comput. Hum. Behav.* 52 (2015), 184–189.

APPENDIX
PROOF OF THEOREM 4.6

Proof. In the following, we consider the traject graphs without compose snapshots and suspended snapshots, since such snapshots do not affect the reachability relations between snapshots.

($\Rightarrow$) Assume that vertex u can reach u' in G in time interval $[t_s, t_e]$. Then there exists a temporal path $\rho = u_1 v_1 u_2 v_2 \dots u_k v_k u_{k+1}$ with $u = u_1$ and $u' = u_{k+1}$ such that the starting time and ending time of ρ fall in $[t_s, t_e]$, where for each $i \in [1, k-1]$ there exist two consecutive connections $u_i \prec\!\!-\!\!\{e_1^i, v_i, e_2^i\}\!\!-\!\!\succ u_{i+1}$ and $u_{i+1} \prec\!\!-\!\!\{e_1^{i+1}, v_{i+1}, e_2^{i+1}\}\!\!-\!\!\succ u_{i+2}$ in G satisfying that (1) the starting times of two consecutive connections increase, *i.e.,* $\Theta(e_1^i, e_2^i).t_s \le \Theta(e_1^{i+1}, e_2^{i+1}).t_s$; and (2) the timestamps in two adjacent edges of a top vertex increases; *i.e.,* (a) if $v_i \neq v_{i+1}$, then $e_2^i.t_e \le e_1^{i+1}.t_s$; and (b) if $v_i = v_{i+1}$, then $e_2^i.t_s \le e_1^{i+1}.t_s$.

Given the above temporal path, we can construct a path from S_u^f to $S_{u'}^l$ in the traject graph $\mathcal{G}_G$. More specifically, for each connection $u_i \prec\!\!-\!\!\{e_1^i, v_i, e_2^i\}\!\!-\!\!\succ u_{i+1}$ associated with path ρ, by the definition of snapshots, $[\max(e_1^i.t_s, e_2^i.t_s), \min(e_1^i.t_e, e_2^i.t_e)]_{v_i}$ must be a snapshot. Moreover, for any two consecutive connections $u_i \prec\!\!-\!\!\{e_1^i, v_i, e_2^i\}\!\!]\!\!-\!\!\succ u_{i+1}$ and $u_{i+1} \prec\!\!-\!\!\{e_1^{i+1}, v_{i+1}, e_2^{i+1}\}\!\!-\!\!\succ u_{i+2}$, there exist two snapshots $s_1 = [\Theta(e_1^i, e_2^i).t_s, \Theta(e_1^i, e_2^i).t_e]_{v_i}$ and $s_2 = [\Theta(e_1^{i+1}, e_2^{i+1}).t_s, \Theta(e_1^{i+1}, e_2^{i+1}).t_e]_{v_{i+1}}$ in the traject graph. Moreover, there exist an edge from s_1 to s_2, since (a) when $v_i \neq v_{i+1}$, $s_1.\mathsf{emin}[u_{i+1}] \le \Theta(e_1^i, e_2^i).t_e \le \Theta(e_1^{i+1}, e_2^{i+1}).t_s \le s_2.\mathsf{smax}[u_{i+1}]$; and (b) when $v_i = v_{i+1}$, (i) if s_1 and s_2 are the same edge, then $\Theta(e_1^i, e_2^i).t_s \le \Theta(e_1^{i+1}, e_2^{i+1}).t_s$, *i.e.,* $s_1.t_s \le s_2.t_s$; otherwise, (ii) $s_1.\mathsf{emin}[u_{i+1}] \le \Theta(e_1^i, e_2^i).t_e \le \Theta(e_1^{i+1}, e_2^{i+1}).t_s \le s_2.\mathsf{smax}[u_{i+1}]$.

Therefore, $([t_s^1, t_e^1]_{v_1}, \dots, [t_s^n, t_e^n]_{v_n})$ be the constructed path in $\mathcal{G}_G$, with $[t_s^1, t_e^1]_{v_1} \in S_u^f$ and $[t_s^n, t_e^n]_{v_n} \in S_{u'}^l$.

($\Leftarrow$) Assume that a snapshot in S_u^f can reach one snapshot in S_v^l. Let $P_{\mathcal{G}} = ([t_s^0, t_e^0]_{v_0}, S_1 = [t_s^1, t_e^1]_{v_1}, S_2, \dots, S_{n-1}, [t_s^n, t_e^n]_{v_n})$ be such a path with $[t_s^1, t_e^1]_{v_1} \in S_u^f$ and $[t_s^n, t_e^n]_{v_n} \in S_{u'}^l$. Starting from the first edge, we construct four consecutive connections for each edge from s_i to s_{i+1} ($i \in [1, n-1]$), such that they form 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 from which snapshot s_1 (resp. s_2) is constructed. Here $e_s^1 = (u_2, v_1, t_s^{11}, t_e^{11})$, $e_e^1 = (u_3, v_1, t_s^{12}, t_e^{12})$, $e_s^2 = (u_3, v_2, t_s^{21}, t_e^{21})$ and $e_e^2 = (u_4, v_2, t_s^{22}, t_e^{22})$. Observe that (a) the first edge may not be an edge of u_1, since one snapshot can be constructed by multiple pairs of edges; (b) the edge from s_1 to s_2 are constructed due to top vertex u_3; and (c) from the definition of these edges, the following conditions hold: when $v_1 \neq v_2$, we have that $s_1.\mathsf{emin}[u_3] \le s_2.\mathsf{smax}[u_3]$; and when $v_i = v_{i+1}$, either (i) $t_s^i \le t_s^j$ holds and s_1 and s_2 are constructed from the same edge, or (ii) $s_1.\mathsf{emin}[u_3] \le s_2.\mathsf{smax}[u_3]$ holds.

We start with the connections from u_2 to u_4. Due to the conditions in (a), assume that e_{emin} is the edge that has the minimum ending time among all edges used to construct s_1; similarly, e_{smax} is the edge that has the maximum starting time among all edges used to construct s_2. We construct the connection as follows: (a) When $v_1 \neq v_2$, $u_2 \prec\!\!-\!\!\{e_s^1, v_1, e_e^1\}\!\!-\!\!\succ u_2 \prec\!\!-\!\!\{e_e^1, v_1, e_{\mathsf{emin}}\}\!\!-\!\!\succ u_2 \prec\!\!-\!\!\{e_{\mathsf{smax}}, v_2, e_s^2\}\!\!-\!\!\succ u_3 \prec\!\!-\!\!\{e_s^2, v_2, e_e^2\}\!\!-\!\!\succ u_4$; (b) when $v_1 = v_2$ and s_1 and s_2 are constructed from the same edge e_{same}, $u_2 \prec\!\!-\!\!\{e_s^1, v_1, e_e^1\}\!\!-\!\!\succ u_2 \prec\!\!-\!\!\{e_e^1, v_1, e_{\mathsf{same}}\}\!\!-\!\!\succ u_2 \prec\!\!-\!\!\{e_{\mathsf{same}}, v_1, e_s^2\}\!\!-\!\!\succ u_3 \prec\!\!-\!\!\{e_s^2, v_1, e_e^2\}\!\!-\!\!\succ u_4$; and (c) when $v_1 = v_2$ and s_1 and s_2 are not constructed from the same edge, the connections are the same as case (a) above.

It remains to construct a connection between u_1 and u_2. To this end, since u_1 and u_2 link to v_1 during the time interval $[t_s^1, t_e^1]$, there exist two edges $e_s^0 = (u_1, v_1, t_s^{01}, t_e^{01})$ $e_e^0 = (u_2, v_1, t_s^{02}, t_e^{02})$ such that $u_1 \prec\!\!-\!\!\{e_s^0, v_1, e_e^0\}\!\!-\!\!\succ u_2$ is a connection, and t_s^{i2} is the minimum starting time among all such pair of edges, *i.e.,* $t_s^{02} \le t_s^{11}$.

Putting these together, we construct the following connections from u_1 and u_4 by replacing the first edge in the above connections with the edge e_s^0 of u_1: $u_1 \prec\!\!-\!\!\{e_s^0, v_1, e_e^0\}\!\!-\!\!\succ u_2 \prec\!\!-\!\!\{e_e^0, v_1, e_{\mathsf{emid}}\}\!\!-\!\!\succ u_2 \prec\!\!-\!\!\{e_{\mathsf{smid}}, v_2, e_s^2\}\!\!-\!\!\succ u_3 \prec\!\!-\!\!\{e_s^2, v_2, e_e^2\}\!\!-\!\!\succ u_4$. Here, e_{emid} is either e_{emin} or e_{same} (see above). Similar for e_{smid}.

(2) We can similarly construct other connections for the edge from s_i to s_{i+1} with $i \in [2, n-1]$, except that we do not need to connect the first top vertex u. Therefore, we can deduce a temporal path from u_1 to u_n. we omit the details here.

(3) However, the last top vertex u_n may not be the target top vertex u'. We can link u' into the path as we link u_1 (see above).

Putting these together, by extending the above temporal paths, we can deduce a temporal path from u to u'. That is, u can reach u' in G during the time interval $[t_s, t_e]$. □