Efficiently Answering Reachability and Path Queries on Temporal Bipartite Graphs

Xiaoshuang Chen
University of New South Wales
xiaoshuang.chen@unsw.edu.au

Kai Wang*
University of New South Wales
kai.wang@unsw.edu.au

Xuemin Lin
University of New South Wales
lxue@cse.unsw.edu.au

Wenjie Zhang
University of New South Wales
zhangw@cse.unsw.edu.au

Lu Qin
University of Technology Sydney
lu.qin@uts.edu.au

Ying Zhang
University of Technology Sydney
ying.zhang@uts.edu.au

ABSTRACT

Bipartite graphs are naturally used to model relationships between two different types of entities, such as people-location, author-paper, and customer-product. When modeling real-world applications like disease outbreaks, edges are often enriched with temporal information, leading to temporal bipartite graphs. While reachability has been extensively studied on (temporal) unipartite graphs, it remains largely unexplored on temporal bipartite graphs. To fill this research gap, in this paper, we study the reachability problem on temporal bipartite graphs. Specifically, a vertex u reaches a vertex w in a temporal bipartite graph G if u and w are connected through a series of consecutive wedges with time constraints. Towards efficiently answering if a vertex can reach the other vertex, we propose an index-based method by adapting the idea of 2-hop labeling. Effective optimization strategies and parallelization techniques are devised to accelerate the index construction process. To better support real-life scenarios, we further show how the index is leveraged to efficiently answer other types of queries, e.g., single-source reachability query and earliest-arrival path query. Extensive experiments on 16 real-world graphs demonstrate the effectiveness and efficiency of our proposed techniques.

PVLDB Reference Format:
Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. Efficiently Answering Reachability and Path Queries on Temporal Bipartite Graphs. PVLDB, 14(10): 1845 - 1858, 2021.
doi:10.14778/3467861.3467873

1 INTRODUCTION

Bipartite graph serves as a useful data model when modeling relationships between two different types of entities and has been adopted in a large spectrum of applications including disease control on people-location networks [21, 38], fraud detection on user-page networks [36, 55] and recommendation on customer-product networks [26, 53, 56, 57, 66]. In the past decades, bipartite graphs are enriched with node attributes, edge importance and edge timestamps, yielding attributed bipartite graphs [65], weighted bipartite graphs [42, 58] and temporal bipartite graphs [21, 38], which are crucial to capture complex situations and network dynamics. In particular, the temporal bipartite graph further records two timestamps (i.e., the starting and ending times) for each edge, which is an effective model in many real-world applications. For example, as shown in Figure 1, the temporal bipartite graph can be naturally used to model the movements of people between different locations. Such a model is a powerful tool in modeling disease outbreaks since it can capture the physical contact patterns (i.e., people visit a location simultaneously), which play a key role in the spread of many infectious diseases [21]. Motivated by this example, an interesting question raised is: how to identify if an individual is potentially infected by a virus carrier through a series of physical contacts based on the temporal bipartite graph model. *Reachability*, which studies if a vertex is reachable from the other vertex, is a natural fit for answering this kind of question.

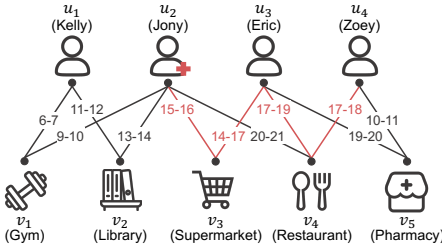


Figure 1: A people-location network for modeling disease outbreaks, which is derived from the paper [21] in *Nature*. Each edge has two timestamps denoting an individual's arriving and leaving times (in 24-hour clock) at the location.

Temporal Bipartite Reachability. In this paper, we study the reachability problem on temporal bipartite graphs. By considering the special characteristics of temporal bipartite graph structure, the reachability on temporal bipartite graphs is actually in a 2-hop manner. Specifically, to reflect the interaction between two same-type entities on temporal bipartite graphs (e.g., the physical contact between two people), we define *time-overlapping wedge*, denoted by $\mathcal{W}=(e, e')$, which consists of two adjacent edges with overlapped time intervals. The starting time and the ending time of $\mathcal{W}$ are then defined as the starting time of e and the ending time of e', respectively. For example, in Figure 1, $\mathcal{W}_1=(e=(u_2, v_3, 15, 16), e'=(u_3, v_3, 14, 17))$ and $\mathcal{W}_2=((u_3, v_4, 17, 19), (u_4, v_4, 17, 18))$ are two time-overlapping wedges. $\mathcal{W}_1$ starts at 15 and ends at 17. $\mathcal{W}_2$ starts at 17 and ends at

*Kai Wang is the corresponding author

Proceedings of the VLDB Endowment, Vol. 14, No. 10 ISSN 2150-8097.
doi:10.14778/3467861.3467873

18. Accordingly, the *temporal bipartite reachability* is defined as: a vertex u reaches a vertex w in a temporal bipartite graph G if they are connected by a series of consecutive time-overlapping wedges (i.e., in a 2-hop manner), and the times of the passing wedges follow a non-decreasing order. Note that if u and w are in different vertex layers, the last wedge should be replaced by an edge. Intuitively, the non-decreasing time constraint reflects the time order dependency, e.g., a person physically contacted with the source of infection can then become a virus carrier, who has the ability to spread the disease later. In Figure 1, u_2 reaches u_4 through the path $< \mathcal{W}_1, \mathcal{W}_2 >$, in which the starting time of W_2 is not smaller than the ending time of $\mathcal{W}_1$ (i.e., non-decreasing). This indicates that Eric (u_3) and Zoey (u_4) are potentially infected by Jony (u_2, the source of infection). The result given by temporal bipartite reachability is reasonable since the contact occurs when Eric and Jony are simultaneously located in the supermarket, and the disease can be spread when Eric then moves to the restaurant and is co-located with Zoey.

Given two vertices u and w in a temporal bipartite graph G, and a time interval $\mathcal{I} = [I_s, I_e]$, in this paper, we study the following temporal bipartite reachability and path queries: (1) *single-pair reachability query*, which answers if u can reach w within $\mathcal{I}$, i.e., the starting time of the first wedge and the ending time of the last wedge (or edge) in the path fall into $\mathcal{I}$; (2) *single-source reachability query*, which returns a vertex set including all the reachable vertices from u within $\mathcal{I}$; and (3) *earliest-arrival path query*, which retrieves a path that connects u and w within $\mathcal{I}$ and has the minimum ending time. Note that the latter two types of queries are based on the single-pair reachability query and are studied to better support real-life applications.

Applications. Answering reachability and path queries on temporal bipartite graphs has extensive real-world applications, and we present two representative scenarios as follows.

• *Supporting control of disease outbreaks.* In everyday life, people move between various locations in the course of carrying out their daily activities, e.g., study, work, and shopping [21, 70]. This can be naturally modeled as a people-location temporal bipartite graph, which captures the physical contact patterns among people and serves as an effective model in disease outbreaks [21]. Note that constructing such a graph only needs the check-in data of people at locations. These data can be easily collected via electronic methods such as QR code, which has been promoted in many countries (e.g., Australia [3], and Singapore [4]). As studied in [21], for many infectious diseases such as influenza, severe acute respiratory syndrome, and recently COVID-19, transmission occurs mainly between people who have physical contacts, and spread is mainly due to people's movements [21]. Consequently, answering reachability and path queries on temporal bipartite graphs can be applied to identify the potentially infected population, uncover high-risk venues, and reveal possible transmission chains, all of which are key elements in preventing an outbreak from becoming an epidemic.

• *Tracing metabolic pathways.* In biochemistry, cellular metabolism, which is a set of biochemical reactions among metabolites, is crucial to maintain the life of organisms [37]. Specifically, the metabolites serve as reactants and regulate the circadian rhythm in a cell at different times [31]. If two metabolites participate in a reaction simultaneously, they will give rise to a product, which may become the substrate for another reaction. A metabolic pathway is a linked series of such reactions, and its end product is crucial for anabolism and catabolism [37]. In the cellular metabolism, metabolites and reactions form a temporal bipartite graph, in which an edge indicates that a metabolite participates in a specific reaction [47]. As a result, the proposed temporal bipartite reachability model can be used to trace the metabolic pathways in a cell, which has significant values in maintaining homeostasis within an organism.

Applying Existing Techniques. To answer the temporal bipartite reachability and path queries, one may consider projecting the temporal bipartite graph into a temporal unipartite graph and extending the existing techniques on temporal unipartite graphs [59, 64, 69] to solve the problems. Despite projection provides a possible solution, size inflation and information loss are two of its main drawbacks as evaluated in [38, 43]. In our experiments, extending the state-of-the-art technique on temporal unipartite graphs [64] is inefficient when answering both single-pair and single-source reachability queries. Furthermore, the existing algorithms [59, 64, 69] are hard to answer the earliest-arrival path queries since the information of one vertex layer is lost after projection. Even though auxiliary information can be recorded for each edge in the projected graph, it is obviously time-prohibitive to retrieve paths by searching from these information. Alternatively, to answer queries directly based on temporal bipartite graphs, a straightforward BFS-based solution can be devised. However, the algorithm has a large search scope, making it impractical to support online queries on large graphs.

Our Approaches. In this paper, we focus on indexing-based approaches to efficiently answer temporal bipartite reachability and path queries. We first propose an index structure, namely `TBP-Index`, which is based on the well-known notion of 2-hop labeling [8, 9, 20] and can efficiently support all possible single-pair reachability queries. To efficiently answer the single-source reachability query and the earliest-arrival path query, we further investigate how to extend the `TBP-Index` with negligible extra costs such that the above two queries can be answered without having to iterate over each vertex or inspecting the original graph.

To efficiently compute the `TBP-Index`, we propose a novel index construction algorithm, namely `TBP-build`*, which incorporates two optimization techniques, i.e., *time-priority-based traversal* and *temporal-based edge partition*. In brief, the time-priority-based traversal technique explores the containment relationship among time intervals. Utilizing this technique, the number of unnecessary vertex visits is substantially reduced. The temporal-based edge partition technique investigates the process of forming time-overlapping wedges and leverages the bipartite structure to accelerate the wedge computation process. In particular, it partitions the edges of the hub vertices (i.e., intermediate vertices of wedges) into several subsets and guarantees that only two edges from the same subset can form time-overlapping wedges. In this way, unnecessary comparisons when computing time-overlapping wedges can be greatly pruned. To leverage the advantages of multi-core computing architectures, we further propose a lock- and atomic-free parallel index construction algorithm `TBP-build`*`-PL`. As evaluated in the experiments, `TBP-build`*`-PL` achieves significant parallel speedup.

Note that even though the concept of 2-hop labeling has been widely adapted in the literature [8, 9, 20, 59, 62], our approach is not a straightforward extension of existing solutions. Instead, it exploits the characteristics of temporal bipartite graphs to achieve

significant improvements on index construction. The proposed techniques leverage not only the temporal information of edges but also the bipartite structure on forming wedges. In addition, we propose a parallel algorithm to further speed up the index construction.

Contributions. Our principal contributions are listed as follows.

- We are the first to study the reachability problem on temporal bipartite graphs by taking the characteristics of temporal bipartite graphs into consideration.
- We devise a novel TBP-Index, by which we can efficiently answer all possible single-pair reachability queries. The TBP-Index is further extended, without changing its theoretical time/space complexity bound, to support fast single-source reachability and earliest-arrival path queries.
- We propose an efficient index construction algorithm TBP-build* with two effective optimization techniques, i.e., time-priority-based traversal and temporal-based edge partition. We further propose a lock- and atomic-free parallel algorithm to take advantage of multi-core computing architectures.
- We conduct comprehensive experiments on 16 real-world temporal bipartite graphs to demonstrate the effectiveness and efficiency of our proposed methods.

Related Work. To the best of our knowledge, this paper is the first to study reachability on (temporal) bipartite graphs. Below we review closely related work on unipartite graphs.

Reachability on general unipartite graphs. Unipartite network is the most popular graph models [16–18, 22–25, 28, 39, 41, 54], and various reachability queries have been studied on unipartite networks [8, 9, 15, 19, 20, 27, 30, 40, 45, 46, 48, 50–52, 60, 61, 67, 68]. To support fast point-to-point queries, researches in the literature mainly focus on indexing-based approaches, which include the following two main groups: (1) *Label-Only*, where the researches aim at computing a complete index to support all possible reachability queries. The algorithms are based on compressed transitive closure [19, 27, 46, 51, 52], 2-hop labeling [8, 9, 20] or 3-hop labeling [30], to just name a few; and (2) *Label+Search*, where the algorithms [15, 45, 48, 60, 61, 67] focus on computing a small partial index to reduce the construction cost. As the index is incomplete, graph traversal will be invoked for the queries that cannot be covered by the index. Interested readers may refer to the survey [68] for more details. Note that these algorithms cannot be directly applied to compute the temporal bipartite reachability as they do not consider the temporal information and the bipartite structure.

Reachability on temporal unipartite graphs. Algorithms in this category take the temporal information of edges into account when computing reachability between two vertices. There are mainly four kinds of reachability models on temporal unipartite graphs, including the temporal reachability model, the restless reachability model, the historical reachability model, and the span-reachability model. Among all the four models, the temporal reachability model [13, 32, 63] is the most widely used one, where a vertex reaches the other if they are connected through a temporal path, i.e., a series of consecutive edges with non-decreasing times. Representative algorithms to compute the temporal reachability include TTL [59] that aims at efficient route planning on transportation networks, TopChain [64] that builds a partial index, and TVL [69] that focuses on distributed environments and computes a partial index as well. The restless reachability model [14] is based on the temporal reachability model, while it requires that in a temporal path, the time spent at each vertex cannot exceed a given threshold and there are no duplicate vertices. The historical reachability model [44] is proposed for evolving graphs, where a vertex reaches the other if there exist paths between them in all or some graph snapshots. The span-reachability model [62] relaxes the time order dependency of the temporal reachability model, and u span-reaches v if they are connected by a path where the times of the edges fall into a given time interval. Among the above works, the algorithms in [59, 64, 69] can be adapted to the projected graph to support the reachability queries on temporal bipartite graphs as discussed before. The experimental result shows that our proposed algorithm significantly outperforms this method.

2 PRELIMINARIES

Our problem is defined over an undirected temporal bipartite graph $G(V{=}(U, L), E)$, where $U(G)$ denotes the set of vertices in the upper layer, $L(G)$ denotes the set of vertices in the lower layer, $U(G) \bigcap L(G) = \emptyset$, $V(G) = U(G) \bigcup L(G)$ denotes the vertex set and $E(G) \subseteq U(G) \times L(G)$ denotes the set of temporal edges. A temporal edge between two vertices u and v in G is denoted as (u, v, t_s, t_e) or (v, u, t_s, t_e), where t_s (resp. t_e) records the starting time (resp. ending time) of the relationship between u and v. $E(S){\subseteq}E(G)$ denotes a set of edges adjacent to the vertices in S, where S is a vertex set. $N_G(x){=}\{(y, t_s, t_e)|(x, y, t_s, t_e) \in E(G)\}$ denotes the neighbor set of a vertex x in G. Accordingly, $d_G(x){=}|N_G(x)|$ denotes the degree of x in G, and $\mathcal{V}(N_G(x))$ denotes the set of all distinct vertices in $N_G(x)$, i.e., $\mathcal{V}(N_G(x)){=}\{y|(y, t_s, t_e) \in N_G(x)\}$. $\mathcal{I}$ denotes a time interval $[I_s, I_e]$, and $ST(\mathcal{I})$ and $ET(\mathcal{I})$ denote the starting time and ending time of $\mathcal{I}$.

Definition 1. *(Wedge) Given a temporal bipartite graph G and three vertices $u, v, w \in V(G)$, a wedge is a path starting from u, going through v and ending at w.*

Definition 2. *(Time-overlapping Wedge) Given a temporal bipartite graph G, a time-overlapping wedge, denoted by $\mathcal{W} = (e_1 = (u, v, t_s, t_e), e_2 = (v, w, t'_s, t'_e))$, is a wedge where the two edges have overlapped time intervals, i.e., $\min(t_e, t'_e) > \max(t_s, t'_s)$. Accordingly, $\mathcal{W}$ is a wedge from u (the starting vertex) to w (the ending vertex), and the starting time and the ending time of $\mathcal{W}$ is defined as t_s (i.e., the starting time of e_1) and t'_e (i.e., the ending time of e_2), respectively.*

Definition 3. *(Time-respecting Path) Given a temporal bipartite graph G, a time-respecting path connecting two same-layer vertices is a sequence of consecutive time-overlapping wedges, denoted by $\mathcal{P} = \langle \mathcal{W}_1, \mathcal{W}_2 \cdots, \mathcal{W}_k \rangle$, such that for any $i \in [1, k-1]$, the ending vertex of $\mathcal{W}_i$ equals the starting vertex of $\mathcal{W}_{i+1}$, and the ending time of $\mathcal{W}_i$ is not larger than the starting time of $\mathcal{W}_{i+1}$. Accordingly, we define the starting time (or the starting vertex) of $\mathcal{P}$ as that of $\mathcal{W}_1$, and the ending time (or the ending vertex) of $\mathcal{P}$ as that of $\mathcal{W}_k$.*

Remark 1. *Definition 3 can be easily adapted to connect two vertices from different layers. To achieve this, the last wedge $\mathcal{W}_k$ should be replaced by an edge (u, v, t_s, t_e), where u equals the ending vertex of $\mathcal{W}_{k-1}$ and t_s is not smaller than the ending time of $\mathcal{W}_{k-1}$. The ending vertex (resp. the ending time) of $\mathcal{P}$ then changes to v (resp. t_e).*

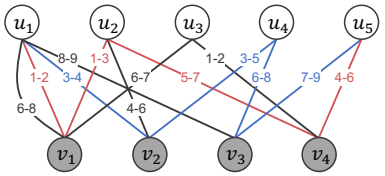


Figure 2: A temporal bipartite graph G

Example 1. *In Figure 2, $\mathcal{W}_1$=$((u_1, v_1, 1, 2), (v_1, u_2, 1, 3))$ and $\mathcal{W}_2$=$((u_2, v_4, 5, 7), (v_4, u_5, 4, 6))$ are two time-overlapping wedges. $\mathcal{P} = \langle \mathcal{W}_1, \mathcal{W}_2 \rangle$ is a time-respecting path from u_1 to u_5 (marked in red), which starts at 1 and ends at 6.*

Definition 4. *(Single-pair Reachability) Given two vertices u and w in a temporal bipartite graph G, and a time interval $\mathcal{I}$, u reaches w within $\mathcal{I}$, denoted by $u \rightsquigarrow_{\mathcal{I}} w$, if there exists a time-respecting path $\mathcal{P}$ from u to w such that the starting time and the ending time of $\mathcal{P}$ fall into $\mathcal{I}$.*

Definition 5. *(Single-source Reachability) Given a vertex u in a temporal bipartite graph G and a time interval $\mathcal{I}$, the single-source reachability aims to identify a vertex set including all the same-layer vertices that u can reach within $\mathcal{I}$.*

Definition 6. *(Earliest-arrival Path) Given two vertices u and w in a temporal bipartite graph G, and a time interval $\mathcal{I}$, a time-respecting path $\mathcal{P}$ is an earliest-arrival path from u to w if it has the minimum ending time among all the paths in $\boldsymbol{P}$, where $\boldsymbol{P}$ is a set including all the time-respecting paths from u to w within $\mathcal{I}$.*

Example 2. *In Figure 2, we have $u_1 \rightsquigarrow_{[1,9]} u_5$. The set of vertices that u_1 can reach within $[1, 9]$ is $\{u_2, u_3, u_4, u_5\}$. $\mathcal{P}_1$ (marked in red) is an earliest-arrival path from u_1 to u_5 within $[1, 9]$.*

Problem Statement. Given a temporal bipartite graph G, two vertices u, w, and a time interval $\mathcal{I}$, we aim to efficiently answer the following queries: (1) single-pair reachability query, which answers if u can reach w within $\mathcal{I}$ following Definition 4; (2) single-source reachability query, which returns a set of vertices that u can reach within $\mathcal{I}$; and (3) earliest-arrival path query, which retrieves an earliest-arrival path from u to w within $\mathcal{I}$.

In the following, we first present solutions for answering single-pair reachability queries. The solutions for single-source reachability and earliest-arrival path queries are introduced in Section 5. Note that we omit the proofs of the lemmas and theorems in this paper due to short of space.

3 SOLUTION OVERVIEW

In this section, we first present two baseline solutions and then introduce a 2-hop labeling index-based solution for answering single-pair reachability queries. Hereafter, we will omit single-pair in single-pair reachability when the context is clear. Note that we mainly focus on answering queries between two vertices in the same vertex layer, e.g., the upper layer, which is the basis of temporal bipartite reachability. Our solutions can be easily adapted to support queries between two vertices in different layers since the time-respecting path between two different-layer vertices only differs in the last edge according to Remark 1.

3.1 Baseline Solutions

Applying Existing Algorithm. `TopChain` [64], the state-of-the-art algorithm on unipartite graphs, can be applied to the projected graph of the temporal bipartite graph to answer single-pair reachability queries. In the following, we briefly introduce the projection process and the key idea of `TopChain`.

Projection: Given a temporal bipartite graph G, its projected graph G^* is a graph defined over a chosen vertex layer of G (e.g., upper layer) such that for $\forall u, w \in V(G^*)$, $(u, w, t_s, t_e) \in E(G^*)$ if and only if there exists a time-overlapping wedge from u to w in G starting at t_s and ending at t_e.

`TopChain`*:* Based on the above projected graph G^*, `TopChain` [64] can compute the single-pair reachability queries. In brief, `TopChain` contains two phases: (1) graph transformation. In this phase, `TopChain` transforms the input graph G^* into a DAG G^+, of which the vertices encode the temporal information of edges in G^*; (2) index construction. `TopChain` then constructs a partial index based on a chain cover of G^+, which records top-k labels for each vertex. The index construction process needs to follow a (reverse) topological order of vertices.

BFS-based Online Approaches. Given a temporal bipartite graph G, two query vertices $u, w \in U(G)$ and a time interval $\mathcal{I}$, Algorithm 1 presents an online approach `OReach` (omit all the blue parts) to answer whether u can reach w within $\mathcal{I}$ regarding the single-pair reachability. `OReach` performs a BFS starting at u and applies a queue Q to store all the reached upper-layer vertices along with the reached timestamps. In Lines 11-19, `OReach` explores the 2-hop neighbors of each vertex $u' \in Q$ and finds those that are connected to u' via valid time-overlapping wedges (Definition 2). `OReach` returns *true* if one of the 2-hop neighbors equals w (Line 17), or otherwise push the found vertex into Q (Line 19).

Pruning rules. We further devise the following pruning rules.

Rule 1: Let an array `minT` record the minimum ending time of each reached upper-layer vertex (Line 2). When computing the time-overlapping wedges, the edges with the ending times larger than or equal to the recorded `minT` values can be skipped (Line 15).

Rule 2: Let `maxMinT` be an array that $\forall v \in L(G)$, `maxMinT[v]` records the maximum `minT` value among all adjacent vertices of v (Line 3). If a new-coming edge's starting time is not smaller than the `maxMinT` value, its related computation can be pruned (Line 13).

Note that `OReach` is based on the naive breadth-first search, and its computation can be further accelerated by using the recently optimized BFS variant, namely direction-optimizing breadth-first search (short as `DO-BFS`) [12]. We then propose `OReach`$^+$ (Algorithm 1 with blue content), which is an optimized online algorithm based on `DO-BFS`. `OReach`$^+$ applies two queues Q and P to store the frontiers from u and those from w, respectively (Lines 1 and 4). Additionally, to reduce unnecessary vertex visits in reverse BFS, `OReach`$^+$ applies two arrays, i.e., `maxT` and `minMaxT`, which record the maximum starting time of each reached upper-layer vertex from w and the related minimum `maxT` value (Lines 5-6). `OReach`$^+$ performs the BFS search if the frontier size from u is smaller than that from w (Line 8). Otherwise, it performs the reverse BFS search (Line 22-23). `OReach`$^+$ returns true if there exists a common vertex that can be reached from both u and w with the time ordering satisfied (Line 18).

Algorithm 1: OReach (OReach$^+$)

Input : a temporal bipartite graph G, two vertices $u \in U(G)$ and $w \in U(G)$, and a time interval $\mathcal{I} = [I_s, I_e]$;
Output : the single-pair reachability from u to w

1 $Q \leftarrow$ an empty queue; $Q.push((u, I_s))$;
2 $minT[u] \leftarrow 0$; $minT[u'] \leftarrow \infty$ for $\forall u' \in U(G) \backslash \{u\}$;
3 $maxMinT[v] \leftarrow \infty$ for $\forall v \in L(G)$;
/* for reverse breadth-first search */
4 $P \leftarrow$ an empty queue; $P.push((w, I_e))$;
5 $maxT[w] \leftarrow \infty$; $maxT[u'] \leftarrow 0$ for $\forall u' \in U(G) \backslash \{w\}$;
6 $minMaxT[v] \leftarrow 0$ for $\forall v \in L(G)$;
7 **while** $Q \neq \emptyset$ **and** $P \neq \emptyset$ **do**
8 **if** $Q.size \leq P.size$ **then**
9 **while** $Q \neq \emptyset$ **do**
10 $(u', t_e) \leftarrow Q.pop()$;
11 **foreach** unvisited $e = (u', v, t_1, t_2)$ of $u' : t_1 \geq t_e$ **do**
12 mark e as visited; $t \leftarrow 0$;
13 **if** $t_1 \geq maxMinT[v]$ **then continue**;
14 **foreach** $w' \in \mathcal{V}(N_G(v))$ **do**
15 **foreach** $e' = (v, w', t_1', t_2') : t_2' < minT[w']$ **do**
16 **if** e, e' are time-overlapping: $t_2' \leq I_e$ **then**
17 **if** $w' = w$ **then return** *true*;
18 (**if** $minT[w'] \leq maxT[w']$ **then return** *true*;)
19 $Q.push((w', t_2'))$; $minT[w'] \leftarrow t_2'$;
20 **if** $minT[w'] > t$ **then** $t \leftarrow minT[w']$;
21 $maxMinT[v] \leftarrow t$;
22 **else**
23 The procedure of performing reverse breadth-first search based on P is similar to Lines 9-21;
24 **return** *false*;

The time complexity of Algorithm 1 is dominated by performing the while loop in Lines 7-21. In the worst case, all the upper-layer vertices will be pushed into Q. For each vertex $u' \in Q$, Lines 12-21 are executed at most $d_G(u')$ times since the adjacent edges of u' can be processed at most once (Line 11). Thus, the overall time complexity of OReach (and OReach$^+$) is $O(\sum_{x \in U(G)} \sum_{v \in N_G(x)} d_G(v))$.

3.2 A 2-hop Labeling Index-based Solution

The Index Structure. Inspired by the 2-hop labeling [8, 9, 20], we design the index structure for the Temporal Bipartite graph, denoted by TBP-Index, as follows.

Definition 7. *(TBP-Index) Given a temporal bipartite graph G, a TBP-Index $\mathcal{L}$ includes two label sets, namely in-label set $\mathcal{L}_{in}$ and out-label set $\mathcal{L}_{out}$. For each vertex $u \in U(G)$, both $\mathcal{L}_{in}(u)$ and $\mathcal{L}_{out}(u)$ records a series of triplets, where each triplet $(w, t_s, t_e) \in \mathcal{L}_{in}(u)$ indicates that w reaches u within $[t_s, t_e]$, and each triplet $(w', t_s', t_e') \in \mathcal{L}_{out}(u)$ indicates that u reaches w' within $[t_s', t_e']$.*

In this paper, we focus on the minimal TBP-Index. We say a TBP-Index $\mathcal{L}$ is a minimal TBP-Index if it is *complete*, and each of its index entries is *necessary*. By complete, it means we can correctly answer all the single-pair reachability queries based on $\mathcal{L}$ by using the following query processing strategy. By necessary, it means $\mathcal{L}$ will become incomplete if we remove any index entry in it. Table 1 presents a minimal TBP-Index for the graph G in Figure 2.

Table 1: A minimal TBP-Index $\mathcal{L}$ of G

Vertex	$\mathcal{L}_{out}$	$\mathcal{L}_{in}$
u_1	-	-
u_2	$(u_1, 1, 2)$, $(u_1, 5, 9)$	$(u_1, 1, 3)$
u_3	$(u_1, 6, 8)$	$(u_1, 6, 7)$
u_4	$(u_1, 3, 4)$, $(u_2, 3, 6)$	$(u_1, 3, 5)$, $(u_2, 5, 8)$, $(u_2, 4, 5)$
u_5	$(u_1, 7, 9)$, $(u_2, 4, 7)$, $(u_4, 7, 8)$	$(u_1, 8, 9)$, $(u_1, 1, 6)$, $(u_2, 5, 6)$, $(u_4, 6, 9)$

Query processing with the index. Given a TBP-Index, two vertices $u \in U(G)$ and $w \in U(G)$, and a time interval $[I_s, I_e]$, u reaches w within $[I_s, I_e]$ if one of the following conditions holds:

(1) $\exists (w, t_s, t_e) \in \mathcal{L}_{out}(u)$: $[t_s, t_e] \subseteq [I_s, I_e]$;
(2) $\exists (u, t_s, t_e) \in \mathcal{L}_{in}(w)$: $[t_s, t_e] \subseteq [I_s, I_e]$;
(3) $\exists (w', t_s, t_e) \in \mathcal{L}_{out}(u), (w', t_s', t_e') \in \mathcal{L}_{in}(w)$: $([t_s, t_e] \subseteq [I_s, I_e]) \wedge ([t_s', t_e'] \subseteq [I_s, I_e]) \wedge (t_e \leq t_s')$.

By sorting the index entries in $\mathcal{L}_{out}(u)$ (and $\mathcal{L}_{in}(w)$) in ascending order of vertex ids and ending times (for the entries with the same id), checking the above conditions can be implemented by linear searches of label sets. Algorithm 2 shows the details of answering a reachability query from u to w based on a sorted TBP-Index.

Algorithm 2: TBP-query

Input : A TBP-Index $\mathcal{L}$ of graph G, two vertices $u \in U(G)$ and $w \in U(G)$, and a time interval $[I_s, I_e]$;
Output : the single-pair reachability from u to w

1 $i, j \leftarrow 1$;
2 **while** $i \leq |\mathcal{L}_{out}(u)| \wedge j \leq |\mathcal{L}_{in}(w)|$ **do**
3 $(x, t_s, t_e) \leftarrow$ the i-th entry in $\mathcal{L}_{out}(u)$;
4 $(x', t_s', t_e') \leftarrow$ the j-th entry in $\mathcal{L}_{in}(w)$;
5 **if** $x = w \wedge ([t_s, t_e] \subseteq [I_s, I_e])$ **then return** *true*;
6 **else if** $x' = u \wedge ([t_s', t_e'] \subseteq [I_s, I_e])$ **then return** *true*;
7 **else if** $x = x' \wedge ([t_s, t_e] \subseteq [I_s, I_e]) \wedge ([t_s', t_e'] \subseteq [t_e, I_e])$ **then**
8 **return** *true*;
9 **else if** $x < x'$ **then** $i \leftarrow i + 1$;
10 **else if** $x > x'$ **then** $j \leftarrow j + 1$;
11 **else** $i \leftarrow i + 1$; $j \leftarrow j + 1$;
12 **return** *false*;

The time complexity of answering a single-pair reachability query from u to w based on the TBP-Index (Algorithm 2) is $O(\delta)$, where δ is the maximum labeling size of the index, i.e., $\delta = \max_{u \in U(G)}(\max(|\mathcal{L}_{out}(u)|, |\mathcal{L}_{in}(u)|))$. While taking the indexing cost into account, the amortized complexity for answering a query is $O(\frac{C_{index}}{M} + \delta)$, where C_{index} is the time cost of building index (discussed in Section 4), and M is the number of queries in total. Clearly, when M is large, the amortized cost will be negligible. One can imagine, in a disease outbreak such as COVID-19, the number of new cases may lead to the same number of single-source queries, and such a number can reach up to hundreds of thousands in a single day [1, 6]. This may cause M to be very large. Note that δ denotes the maximum labeling size, while the labeling sizes of two query vertices in practice can be much smaller than δ. We show in Lemma 1 that under the uniform distribution, the average-case time complexity of answering a query is $O(\bar{\delta})$, where $\bar{\delta} < \frac{2}{3}\delta + \frac{1}{2}$.

Lemma 1. *Suppose that the size of the labeling set for a vertex follows the uniform distribution in the range of* $[0, \delta]$ *and is independent among different vertices. The average-case time complexity of answering a single-pair reachability query based on the* TBP-Index *is* $O(\bar{\delta})$*, where* $\bar{\delta} < \frac{2}{3}\delta + \frac{1}{2}$.

4 INDEX CONSTRUCTION

4.1 Basic Index Construction

Overview of computing 2-hop labeling. We begin by presenting some background knowledge on computing 2-hop labeling (*a.k.a.* 2-hop cover). Cohen et al. [20] propose 2-hop cover to support point-to-point reachability queries on directed graphs. They prove that computing a 2-hop cover with the minimum size is NP-hard. Then, they propose an approximate algorithm that computes a 2-hop cover with the size larger than the minimum size by at most a logarithmic factor. However, the approximate algorithm is inefficient since it invokes a procedure of densest subgraph computation in each iteration. Hierarchical 2-hop labeling [8, 9] (shorted as HHL) is then proposed as an efficient alternative. In particular, a 2-hop cover is called "hierarchical" if it satisfies the *order constraint*, that is, if a vertex w presents in the labeling set of a vertex u (i.e., $\mathcal{L}_{out}(u)$ or $\mathcal{L}_{in}(u)$), then $O(w) < O(u)$, where O gives a total vertex order. With a given O, HHL can find a unique minimal 2-hop cover [59].

Since TBP-Index is based on the idea of 2-hop cover, computing a TBP-Index with the minimum size is also NP-hard. In this paper, we adopt the idea of HHL and aim to find a minimal TBP-Index. We use the degree-based vertex order O [1], which is a widely used ordering scheme in the existing studies [29, 30, 34, 62]. For two vertices u and w, we say u ranks higher than w in the order (i.e., $O(u) < O(w)$) if u has a higher degree than w, and the tie is broken by vertex id. Note that we focus on the reachability between upper-layer vertices in this paper, and thus the vertex order O only needs to be applied to the upper-layer vertices. Below we present some basic concepts for a minimal TBP-Index $\mathcal{L}$ under the vertex order O.

Definition 8. *(Minimal Index Entry) Given a vertex* $u \in U(G)$*, and two index entries* (w, t_s, t_e) *and* (w', t'_s, t'_e) *in* $\mathcal{L}_{out}(u)$ *(or* $\mathcal{L}_{in}(u)$*),* (w', t'_s, t'_e) *dominates* (w, t_s, t_e) *if* $w = w'$ *and* $[t'_s, t'_e] \subseteq [t_s, t_e]$*. Accordingly, an index entry is a minimal index entry if it cannot be dominated by other index entries.*

Definition 9. *(Canonical Index Entry) Given a vertex* $u \in U(G)$*, an index entry* $(w, t_s, t_e) \in \mathcal{L}_{out}(u)$ *(resp.* $\mathcal{L}_{in}(u)$*) is a canonical index entry, if it satisfies: (1) it is a minimal index entry; and (2)* $\nexists w' \in U(G)$ *s.t.* $(w', t_1, t_2) \in \mathcal{L}_{out}(u)$ *(resp.* $\mathcal{L}_{in}(u)$*),* $(w', t'_1, t'_2) \in \mathcal{L}_{in}(w)$ *(resp.* $\mathcal{L}_{out}(w)$*),* $[t_1, t_2] \subseteq [t_s, t_e]$*,* $[t'_1, t'_2] \subseteq [t_s, t_e]$*,* $t'_1 \geq t_2$ *(resp.* $t_1 \geq t'_2$*),* $O(w') < O(u)$*, and* $O(w') < O(w)$.

Following HHL, a minimal TBP-Index $\mathcal{L}$ has the properties below.

- P1. Given a vertex $u \in U(G)$, for each index entry $(w, t_s, t_e) \in \mathcal{L}_{in}(u) \bigcup \mathcal{L}_{out}(u)$, it satisfies $O(w) < O(u)$.
- P2. Each index entry in $\mathcal{L}$ is a canonical index entry.
- P3. Given a vertex $u \in U(G)$, $\forall (w, t_s, t_e) \in \mathcal{L}_{out}(u)$ (or $\mathcal{L}_{in}(u)$), there does not exist another entry $(w, t'_s, t'_e) \in \mathcal{L}_{out}(u)$ (or $\mathcal{L}_{in}(u)$) s.t. $t'_s = t_s$ or $t'_e = t_e$.

[1] There are also some other vertex ordering schemes, e.g., betweenness-based scheme and significant path-based scheme. Please see [35] for more details.

The above property P1 reflects the order constraint of hierarchical 2-hop cover. P2 holds as otherwise there exists an index entry of which the reachability information can be derived from the other index entries, which means this index entry is not necessary and contradicts the definition of minimal TBP-Index. P3 can also be verified by contradiction. If both $\beta_1 = (w, t_s, t_e)$ and $\beta_2 = (w, t_s, t'_e)$ present in $\mathcal{L}_{out}(u)$, then either β_1 dominates β_2 (when $t_e \leq t'_e$) or β_2 dominates β_1 (when $t_e > t'_e$), both of which contradict P2.

Algorithm 3: TBP-build

```
Input  : a temporal bipartite graph G, and a vertex order O;
Output : L_in and L_out
1  L_in(u), L_out(u) ← ∅ for all u ∈ U(G);
2  for k = 1, 2, ··· , |U(G)| do
3      u_k ← the k-th vertex in O; L'_in ← ∅;
4      foreach unique starting time t_s from u_k do
5          MOReach(G, u_k, t_s, L'_in);
6      find the canonical index entries in L'_in, and add them to L_in;
7      the process to update L_out is similar to Lines 3 - 6;
8  return L_in and L_out;
9  Procedure MOReach(G, u, I_s, L'_in)
10 run Algorithm 1 Lines 1-3; mark all e ∈ E(U(G)) as unvisited;
11 mark all e = (u, v, t'_s, t'_e) adjacent to u with t'_s ≠ I_s as visited;
12 while Q ≠ ∅ do
13     (u', t_e) ← Q.pop();
14     if u' ≠ u ∧ no entries in L'_in(u') can dominate (u, I_s, t_e) then
15         remove all entries in L'_in(u') that (u, I_s, t_e) can dominate;
16         insert (u, I_s, t_e) into L'_in(u');
17     run Algorithm 1 Lines 11-19, replace Line 14 by foreach w' ∈
         V(N_G(v)):O(w') > O(u), remove t'_2 ≤ I_2 and Line 17;
```

A Basic Index Construction Algorithm. Algorithm 3 presents a baseline algorithm TBP-build, which computes a minimal TBP-Index by indexing vertices sequentially in the vertex order O (Line 2). Since the process of computing $\mathcal{L}_{out}$ is quite similar to that of computing $\mathcal{L}_{in}$, in the following, we only discuss the process of computing $\mathcal{L}_{in}$ while omitting $\mathcal{L}_{out}$ hereafter. When processing the vertex u_k with the k-th rank, TBP-build first initializes an empty set $\mathcal{L}'_{in}$ to store the minimal index entries (Line 3). Then, it performs the procedure MOReach $O(d_G(u_k))$ times to compute all the minimal index entries related to u_k (Lines 4-5) in $\mathcal{L}_{in}$. MOReach (Lines 9-17) aims to compute the minimal index entries related to the given vertex and starting time. Specifically, for a candidate entry, MOReach checks if it can dominate or be dominated by the entries in $\mathcal{L}'_{in}$ (Lines 14-16). Finally, for each minimal index entry in $\mathcal{L}'_{in}$, TBP-build performs a query procedure (Algorithm 2) to check if it can be covered by a higher-ranked vertex. If not, this minimal index entry is a canonical one and is added into $\mathcal{L}_{in}$ (Line 6). Note that when Algorithm 2 is applied here, $x < x'$ and $x > x'$ in it should be replaced by $O(x) < O(x')$ and $O(x) > O(x')$.

Example 3. *Given the graph G in Figure 2 and a vertex order of* $O(u_1) < O(u_2) < \cdots < O(u_5)$*, Table 1 shows a minimal* TBP-Index *of G computed by Algorithm 3. Figure 3 depicts the procedure when indexing from* u_1*, and* $\mathcal{L}'_{in}$ *is initially set to* $\emptyset$*. When* t_s=1, MOReach *inserts each of the found candidate entries into* $\mathcal{L}'_{in}$ *(Lines 14-16). Note*

that although u_1 can reach u_5 starting at 1 and ending at 9, $(u_5, 9)$ is not a candidate entry. This is because MOReach *finds $(u_5, 6)$ first, and based on the pruning rule of Algorithm 1,* MinT[u_5] *equals 6 and $(u_5, 9)$ will be pruned. While inserting $(u_4, 5)$ into $\mathcal{L}'_{in}$ at $t_s = 3$,* MOReach *removes $(u_1, 1, 5)$ in $\mathcal{L}'_{in}(u_4)$ as this entry is dominated by $(u_1, 3, 5)$ (Line 15). For $(u_5, 9)$, its related index entry $(u_1, 3, 9)$ is inserted into $\mathcal{L}'_{in}(u_5)$. The process under t_s=6 and t_s=8 is similar. Since no vertex ranks higher than u_1, each entry in $\mathcal{L}'_{in}$ is a canonical index entry and is inserted into the final* TBP-Index *(Line 6).*

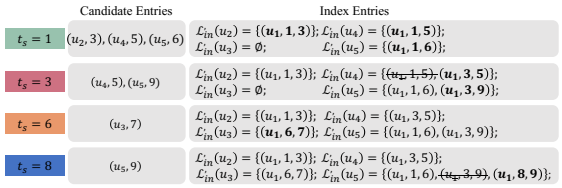


Figure 3: Illustrating the indexing procedure from u_1, and and the starting time t_s is processed in the order of 1, 3, 6 and 8. At time t_s, the index entry in bold indicates that the entry is inserted into the index, while the index entry with a strikethrough means it is removed from the index.

Theorem 1. *(Minimality) The* TBP-Index *$\mathcal{L}$ computed by Algorithm 3 is minimal, which satisfies: (1) completeness, i.e., the reachability of any pair of vertices can be correctly answered by $\mathcal{L}$; (2) necessity, i.e., for $\forall u \in U(G)$ and an arbitrary entry $(w, t_s, t_e) \in \mathcal{L}_{out}(u)$ (or $\mathcal{L}_{in}(u)$), the reachability from u to w (or from w to u) within $[t_s, t_e]$ cannot be correctly answered after removing (w, t_s, t_e).*

Complexity analysis. For Algorithm 3, we analyze the size bound of its computed index and give the time complexity analysis.

Theorem 2. *The size of a minimal* TBP-Index *constructed by Algorithm 3 is bounded by $O(\sum_{u \in U(G)} \sum_{w \in U_{<u}} \min(d_G(u), d_G(w)))$, where $U_{<u} = \{w | w \in U(G) \wedge O(w) < O(u)\}$.*

Theorem 3. *The time complexity of Algorithm 3 is $O(\sum_{u \in U(G)} \sum_{u' \in U_{>u}} (d_G(u)|E(G)| + d_G^2(u) \sum_{v \in N_G(u')} d_G(v)))$, where $U_{>u} = \{u' | u' \in U(G) \wedge O(u') > O(u)\}$.*

4.2 Advanced Index Construction

In this subsection, we propose two novel optimization techniques, i.e., time-priority-based traversal and temporal-based edge partition, to speed up the index construction process.

4.2.1 The Time-priority-based Traversal Strategy.

Issues in computing minimal index entries. In Algorithm 3, for each candidate index entry in Q, TBP-build compares it with all of the entries in $\mathcal{L}'_{in}$ to check if it can dominate or be dominated by the entries in $\mathcal{L}'_{in}$. Because of that, TBP-build incurs high computational cost in maintaining $\mathcal{L}'_{in}$ (or computing minimal index entries). For example, in order to compute the two minimal index entries starting from u_1 and ending at u_5, TBP-build needs to perform ten operations (i.e., comparison, removal and insertion) for $\mathcal{L}'_{in}(u_5)$, as illustrated in Figure 4. Specifically, when the candidate index entry $\delta_1 = (u_1, 3, 9)$ comes, TBP-build will check if it can dominate or be dominated by the existing entry in $\mathcal{L}'_{in}(u_5)$, i.e., $\delta_2 = (u_1, 1, 6)$ (two comparison operations). Since both checkings return false, δ_1 is inserted into $\mathcal{L}'_{in}(u_5)$ (one insertion operation).

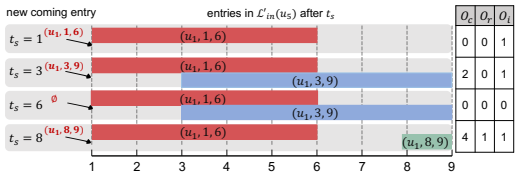


Figure 4: Illustrating the computation of the minimal index entries starting from u_1 and ending at u_5, where O_c, O_r and O_i denote the number of comparison operations, removal operations and insertion operations, respectively.

Time-priority-based traversal strategy. Based on the above observation, to efficiently compute minimal index entries, we present a new time-priority-based traversal strategy: processing the starting times from u_k in decreasing order when indexing from u_k, and for each starting time, exploring the index entries with the minimum ending time first. By doing this, each found candidate index entry cannot be dominated by the entries that are explored afterward (see Lemma 2). Thus, the removal and comparison operations in Algorithm 3 Line 15 can be completely pruned. Based on the new traversal strategy, we can further apply the following pruning rules.

Rule 1: When indexing from u_k (with $O(d_G(u_k))$ starting times), each edge in $E(U(G))$ only needs to be processed at most once.

Rule 2: When indexing from u_k (with $O(d_G(u_k))$ starting times), if w has been reached at t_e before, w only needs to be visited again if it can be reached at t'_e where $t'_e < t_e$.

By applying the new traversal strategy and pruning rules, when computing the minimal index entries starting from u_1 and ending at u_5, we first get the candidate entry $(u_1, 8, 9)$ with starting time 8, which is a minimal index entry. Then, when processing the starting time 3, even though there exists a path $\mathcal{P}$ (marked in blue in Figure 2) from u_1 to u_5 starting at 3, $(u_1, 3, 9)$ is not a candidate entry. This is because the ending time of $\mathcal{P}$ is 9, while u_5 has been reached from u_1 ending at 9 before (Rule 2). Finally, we process the starting time 1 and find the second minimal index entry $(u_1, 1, 6)$. In the experiments, we show that the time-priority-based traversal strategy can significantly speed up the index construction.

4.2.2 The Temporal-based Edge Partition Technique.

Issues in computing time-overlapping wedges. Consider a lower-layer vertex v. When an edge e that is adjacent to v (i.e., $e \in E(v)$) comes, all the others edges in $E(v)$ are the candidates to form time-overlapping wedges with e. However, when v has a high degree, it is more likely that most computations cannot lead to a valid time-overlapping wedge. In Figure 2, given v_1 and $e_1 = (u_2, v_1, 1, 3)$, TBP-build needs to compare e_1 with all the other edges adjacent to v_1 (i.e., $e_2 = (v_1, u_1, 1, 2)$, $e_3 = (v_1, u_1, 6, 8)$, and $e_4 = (v_1, u_3, 6, 7)$). However, both the comparisons with e_3 and e_4 fail to form a time-overlapping wedge. Therefore, it is desirable to design a technique to reduce such unsuccessful comparisons.

Temporal-based edge partition technique. Based on the above observation, we propose the temporal-based edge partition technique (or edge partition for short). We first present the definition of reachability-equivalent partition as follows.

Definition 10. *(Reachability-equivalent Partition) Given a temporal bipartite graph G, and a lower-layer vertex $v \in L(G)$, the reachability-equivalent partition of v, denoted by $\mathcal{S}_v$, is a partition $\{E_1, E_2 \cdots E_T\}$ of $E(v)$ (i.e., the set of edges adjacent to v), such that $\forall 1 \le i \ne j \le T$: (1) $E_i \cap E_j = \emptyset$ and $E(v) = E_1 \bigcup \cdots \bigcup E_T$; (2) $\forall e_1 \in E_i, e_2 \in E_j$, e_1 and e_2 cannot form a time-overlapping wedge; and (3) $|\mathcal{S}_v|$ is maximized.*

Example 4. *In Figure 2, the reachability-equivalent partition of v_1 is $\mathcal{S}_{v_1}$=$\{\{(u_1, v_1, 1, 2), (u_2, v_1, 1, 3)\}, \{(u_1, v_1, 6, 8), (u_3, v_1, 6, 7)\}\}$. Similarly, we have $\mathcal{S}_{v_4}$=$\{\{(u_2, v_4, 5, 7), (u_5, v_4, 4, 6)\}, \{(u_3, v_4, 1, 2)\}\}$, $\mathcal{S}_{v_2} = \{E(v_2)\}$, and $\mathcal{S}_{v_3} = \{E(v_3)\}$.*

By sorting the edges in $E(v)$ according to their timestamps, we can compute $\mathcal{S}_v$ for all lower-layer vertices in $O(\sum_{v \in L(G)} d_G(v) \log d_G(v))$ time. According to Definition 10, only edges in the same subset of a partition can form time-overlapping wedges. Thus, when computing time-overlapping wedges, the computations between edges from different subsets can be skipped, which substantially shrinks the search scope. For example, when e_1=$(u_2, v_1, 1, 3)$ comes during indexing on G, it only needs to be compared with $(u_1, v_1, 1, 2)$, which leads to a valid time-overlapping wedge. In the experiments, we show that the edge partition technique can significantly improve the indexing time.

4.2.3 The Advanced Index Construction Algorithm.

The TBP-build* Algorithm. Putting the above techniques together, details of the advanced index construction algorithm, namely TBP-build*, are shown in Algorithm 4. Given a temporal bipartite graph G, TBP-build* first computes $\mathcal{S}_v$ for each lower-layer vertex v (Line 1) and then constructs the TBP-Index by indexing vertices following the vertex order O (Lines 3-24). When indexing from u_k, TBP-build* processes $O(d_G(u_k))$ starting times in descending order (Line 8) and applies a priority queue which pops the index entry with the minimum ending time first (Line 4). Note that three arrays are initialized when indexing from u_k (Lines 5 - 7), based on which we can mark a number of processed edges/vertices and do not need to visit them repeatedly when a different starting time is applied (Rule 1 and Rule 2). While computing time-overlapping wedges for an unvisited edge $e = (w, v, t_1, t_2)$ (Line 15-24), TBP-build* only compares e with the edges in $\mathcal{S}_v(e)$ (Line 18), where $\mathcal{S}_v(e)$ denotes the subset in $\mathcal{S}_v$ including the edge e. For each entry popped by Q, TBP-build* performs a query procedure (Algorithm 2) to check if it can be covered by the currently constructed index. If the answer is true, TBP-build* directly skips the entry (Line 13).

Lemma 2. *In Algorithm 4, for each candidate index entry popped by Q (Line 11), it cannot be dominated by all the candidate index entries popped by Q afterward.*

Complexity analysis. Note that under the same vertex order, the TBP-Index computed by Algorithm 4 is identical to that computed by Algorithm 3. Thus, the index size is still bounded by Theorem 2. For the time complexity, when indexing from u_k, at most C_{u_k}=$\sum_{w \in U_{>u_k}} \sum_{v \in N_G(w)} d_G(v)$ entries will be inserted into Q (Line 15). For each entry in Q, a query procedure (bounded by $O(|E(G)|)$) will be invoked to check if it can be covered by the currently constructed index. As a whole, the time complexity of Algorithm 4 is $O(\sum_{u \in U(G)} C_u(\log C_u + |E(G)|))$.

Algorithm 4: TBP-build*

```
   Input  : a temporal bipartite graph G and a vertex order O;
   Output : L_in and L_out
1  compute S_v (Definition 10) for ∀v ∈ L(G);
2  L_in(u), L_out(u) ← ∅ for each vertex u ∈ U(G);
3  for k = 1, 2, ··· , |U(G)| do
4      u_k ← the k-th vertex in O; Q ← an empty priority queue;
5      mark all e ∈ E(U(G)) as unvisited;
6      minT[u] ← 0; minT[u'] ← ∞ for ∀u' ∈ U(G)\{u};
7      maxMinT[v] ← ∞ for ∀v ∈ L(G);
8      foreach unique starting time t_s (in desc) from u_k do
9          Q.push((u_k, t_s, t_s));
10         while Q ≠ ∅ do
11             (w, t'_s, t'_e) ← Q.pop();
12             if w ≠ u_k then
13                 if Query(u_k, w, t'_s, t'_e, L) then continue;
14                 else L_in(w) ← L_in(w) ⋃ {(u_k, t'_s, t'_e)};
15             foreach unvisited e = (w, v, t_1, t_2) of w : t_1 ≥ t'_e do
16                 mark e as visited; t ← 0;
17                 if t_1 ≥ maxMinT[v] then continue;
18                 foreach e' = (v, x, t'_1, t'_2) in S_v(e) do
19                     if O(x) ≤ O(u_k) then continue;
20                     if t'_2 ≥ minT[x] then continue;
21                     if e and e' are time-overlapping then
22                         Q.push((x, t'_s, t'_2)); minT[x] ← t'_2;
23                     if minT[x] > t then t ← minT[x];
24                 maxMinT[v] ← t;
25 return L_in and L_out.
```

4.3 Lock- and Atomic-free Parallelization

Reviewing Algorithm 4, the most time-consuming part in it is running Lines 3-24 for each upper-layer vertex. This motivates us to speed up the computation with parallelization. However, a simple parallelization faces the following two issues: (1) *the conflict issue*, write conflicts may occur in Algorithm 4 Line 14. Even though a lock can be applied to provide mutual exclusion, it greatly slows down the performance; and (2) *the canonical issue*, the indexing procedure from a vertex in Algorithm 4 relies on the label sets constructed by its predecessors in the vertex order (to check the canonical property). Such a sequential nature is the key to ensure the minimality of TBP-Index. Simply parallelizing the vertices will break the order dependency and involve redundant index entries.

To tackle the above issues, we propose a two-phase parallel algorithm TBP-build*-PL. In Phase I, it generates local indexes $\mathcal{L}^i_{in}$ and $\mathcal{L}^i_{out}$ for each thread i, and write conflicts can be completely avoided. Thus, *the conflict issue* is solved. In Phase II, TBP-build*-PL cleans all the non-canonical index entries in the sub-indexes, i.e., *the canonical issue* is solved. Note that no conflict appears in both Phase I and Phase II, and TBP-build*-PL achieves lock- and atomic-free parallelization. Additionally, TBP-build*-PL returns an index $\mathcal{L}$ that consists of a series of sub-indexes $\mathcal{L}^i$ ($i \in [1, t]$). Note that TBP-build*-PL constructs a minimal TBP-Index, which has the same index entries as the index constructed by Algorithm 4. In addition, to answer a query based on the above sub-indexes, we

need to scan each sub-index independently, and the answer is true if one of the scans returns true. Thus, the query time complexity remains the same as Algorithm 2.

Remark 2. *Note that the computation of* TopChain *can also be parallelized. The parallelization of the graph transformation phase is straightforward and is omitted due to the short of space. For the index construction phase, the original* TopChain *algorithm applies a (reverse) topological order of vertices, and the indexing procedure of a vertex relies on the labeling of its predecessors in the order. To enable parallelism, we change the topological order to the topological level order. By doing this,* TopChain *can use multiple threads to index the vertices within the same topological level simultaneously.*

5 ANSWER SSRQ AND EAPQ

Next we briefly discuss how to extend the TBP-Index to support fast single-source reachability and earliest-arrival path queries.

5.1 Single-source Reachability Query (SSRQ)

Intuitively, answering an SSRQ can be implemented by processing $|U(G)|$ times of single-pair reachability queries, which is clearly an inefficient approach. To efficiently compute SSRQ, we propose inverted in-label set, denoted by $\widehat{\mathcal{L}}_{in}$, which records each index entry in $\mathcal{L}_{in}$ reversely, i.e., $\forall(w, t_s, t_e) \in \mathcal{L}_{in}(u), (u, t_s, t_e) \in \widehat{\mathcal{L}}_{in}(w)$. With $\widehat{\mathcal{L}}_{in}$ and TBP-Index, SSRQ can be computed by searching label sets rather than performing $|U|$ times of reachability queries.

5.2 Earliest-arrival Path Query (EAPQ)

The TBP-Index cannot be directly applied to support EAPQ as it records no information about passing vertices on paths. To support EAPQ, we present the path-aware TBP-Index, denoted by $\mathcal{L}^*$, which is based on the TBP-Index but additionally records the information of passing vertices. Specifically, for $\forall u \in U(G)$, each index entry in $\mathcal{L}^*{}_{out}(u)$ is a tuple $(w, t_s, t_e, \langle x, y, t_p\rangle)$ meaning that u reaches w within $[t_s, t_e]$, and the first time-overlapping wedge in the path is a wedge from u to x via y starting at t_s and ending at t_p. Similarly, $\forall(w', t'_s, t'_e, \langle x', y', t'_p\rangle) \in \mathcal{L}^*{}_{in}(u)$, it indicates that w' reaches u within $[t'_s, t'_e]$, and the last time-overlapping wedge in the path is from x' to u via y' starting at t'_p and ending at t'_e. Using the path-aware TBP-Index $\mathcal{L}^*$, an earliest-arrival path $\mathcal{P}$ from u to w can be efficiently retrieved in a recursive manner. For instance, in the case that w is the highest-rank vertex in $\mathcal{P}$, we can first search from $\mathcal{L}^*{}_{out}(u)$ to identify the related index entry, and then search its successor's out-label sets until finding w.

6 EXPERIMENTS

This section shows the results of empirical studies. All the algorithms were implemented in C++, and the experiments were conducted on a platform with two Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz (each with 20 cores) and 512GB memory.

Algorithms. We evaluate the following algorithms.

Index Construction Algorithms: (1) TC-build: the TopChain algorithm [64].The related index is called TC-Index. We apply TopChain on the projected graph and use the same settings as [64]; (2) TC-build-PL: the TopChain algorithm in parallel (Remark 2); (3) TBP-build: the baseline algorithm to compute TBP-Index as shown in Algorithm 3; (4) TBP-build*: the advanced algorithm to compute TBP-Index in Algorithm 4; and (5) TBP-build*-PL: the algorithm that constructs a minimal TBP-Index in parallel (Section 4.3).

Query Algorithms: (1) OReach: the online query algorithm in Algorithm 1; (2) OReach$^+$: an optimized online query algorithm (Algorithm 1, with blue content) that bases on the direction-optimizing breadth-first search [12]; (3) TC-query: the query algorithm (algorithm 2 in [64]) using the TC-Index; (4)TBP-query: the TBP-Index-based algorithms that answers the single-pair reachability query (Algorithm 2), single-source reachability query (Section 5.1) and earliest-arrival path query (Section 5.2).

Datasets. We use 16 datasets in our experiments, and Table 2 shows their statistics. Apart from the PM dataset, all the other datasets originally have one timestamp. Thus, we consider the original timestamp as the starting time and generate duration for each edge (ending time is the sum of starting time and duration). According to [11], real-life duration usually follows the power-law distribution. Hereafter, we generate duration from 1 hour to 48 hours using the power-law distribution [7] (α is empirically set to -2.5) by default. We also evaluate the performance of our algorithms under different distribution models in Section 6.4. For the PM dataset, we randomly generate the starting timestamp from 1 Jun 2019 to 31 Dec 2020, and generate the duration for each edge using the same settings. All the networks can be downloaded from KONECT [2].

6.1 Case Studies

We conduct case studies on two real-world datasets, i.e., Brightkite and Gowalla [5], to evaluate the effectiveness of temporal bipartite reachability. Both datasets contain a collection of check-in data shared by users, which forms people-location bipartite graphs. The Brightkite dataset contains 50,686 users (upper layer), 772,788 locations (lower layer) and 4,491,144 check-ins. The Gowalla dataset has 107,092 users, 1,280,969 locations and 6,442,892 check-ins. Note that the above settings are also applied here to generate duration for each check-in data. The difference is that the generated duration for a user's check-in data cannot exceed the time difference between this check-in and the next one. Intuitively, in real-life scenarios, a user can only be at one unique place at the same time.

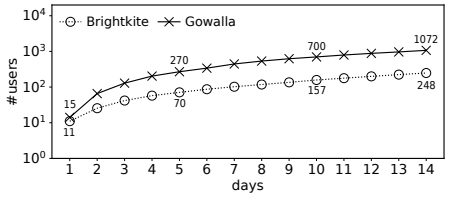


Figure 5: The number of potentially infected users

Firstly, we simulate the process of disease propagation using temporal bipartite reachability. For each dataset, we randomly select a user as the source of infection and consider the other users as potentially infected if they are reachable from the source. Figure 5 shows the number of potentially infected users on average within 14 days. We can see that the number of potentially infected users continues to rise quickly day after day. For example, on Gowalla, 15 users are at risk of infection after 1 day, while this figure surges up to 1072 after 14 days. With such a high transmission rate, it is urgent

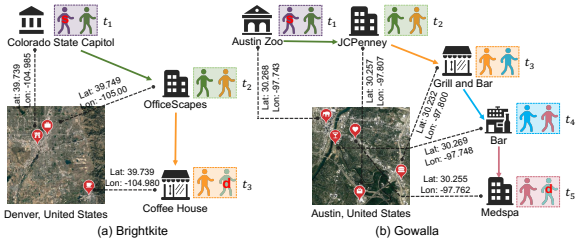
Table 2: Summary of Datasets

Name	Dataset	$\|E(G)\|$	$\|U(G)\|$	$\|L(G)\|$
WQ	Wikiquote	549,210	21,607	92,924
WN	Wikinews	901,416	10,764	159,910
WB	Wikibooks	1,164,576	32,583	133,092
SO	Stackoverflow	1,301,942	545,195	96,678
LK	Linux-kernel	1,565,683	42,045	337,509
CU	Citeulike	2,411,820	153,277	731,769
BS	Bibsonomy	2,555,080	204,673	767,447
TW	Twitter	4,664,605	530,418	175,214
AM	Amazon	5,838,041	2,146,057	1,230,915
WT	Wiktionary	8,998,641	29,348	2,094,520
EP	Epinions	13,668,320	120,492	755,760
LF	Lastfm	19,150,868	992	1,084,620
IW	Itwiki	26,241,217	137,693	2,225,180
EF	Edit-frwiki	46,168,355	288,275	3,992,426
WP	Wikipedia	129,885,939	1,025,084	5,812,980
PM	PubMed	737,869,083	141,043	8,200,000

Table 3: The average query processing time

Name	SPRQ - Answering Positive Queries				SPRQ - Answering Negative Queries				SSRQ		EAPQ
	OReach (ms)	OReach$^+$ (ms)	TC-query (μs)	TBP-query (μs)	OReach (ms)	OReach$^+$ (ms)	TC-query (μs)	TBP-query (μs)	TC-query (ms)	TBP-query (ms)	TBP-query (μs)
WQ	5.56	4.85	0.84	**0.18**	6.02	3.23	1.15	**0.20**	1.88	**0.03**	0.72
WN	9.60	4.05	3.41	**0.41**	9.34	6.52	2.53	**0.72**	1.07	**0.06**	1.08
WB	8.58	4.08	4.40	**0.30**	5.44	2.08	3.11	**0.31**	3.12	**0.02**	0.73
SO	11.68	0.91	9.12	**0.97**	8.57	1.75	1.57	**0.81**	45.32	**0.43**	2.69
LK	6.54	1.67	4.61	**0.85**	4.62	2.40	2.13	**1.91**	3.57	**0.25**	2.36
CU	20.24	2.92	24.62	**2.10**	35.40	21.20	8.77	**4.64**	28.78	**5.38**	5.76
BS	10.65	2.16	52.49	**1.93**	7.71	6.52	6.57	**3.82**	141.30	**4.08**	6.00
TW	79.48	9.32	83.59	**1.79**	53.19	32.34	3.88	**2.98**	118.71	**4.71**	5.76
AM	74.77	4.78	65.83	**2.24**	62.87	11.90	4.09	**2.45**	345.04	**19.52**	6.89
WT	249.00	90.84	8.49	**0.58**	88.63	61.43	3.71	**1.68**	2.44	**0.12**	1.73
EP	35.84	7.50	1269.41	**4.96**	165.42	37.66	90.74	**3.19**	976.89	**7.56**	8.19
LF	312.90	101.86	29.45	**7.87**	91.57	60.37	12.90	**8.68**	0.35	**0.34**	20.83
IW	510.00	119.34	51.00	**2.99**	234.76	152.12	6.26	**3.64**	12.93	**2.04**	10.76
EF	903.24	228.80	36.30	**2.43**	671.80	226.11	5.34	**2.60**	24.24	**4.00**	9.73
WP	25.60s	1.20s	19.32	**4.17**	65.30s	41.40s	5.96	**4.09**	276.62	**26.02**	17.40
PM	58.59s	1.48s	2827.75	**16.96**	172.0s	90.37s	1032.30	**140.25**	12360.0	**180.22**	207.09

for the health department to rapidly identify the infected population to prevent disease propagation. Fortunately, our proposed TBP-Index-based algorithm provides a helpful tool to do the job. For example, on Gowalla, our proposed TBP-Index-based algorithm spends merely 60s identifying the infected population for 10,000 infected sources, while the brute-force online algorithm needs more than 1.4 hours to do the same job. Note that the number of new cases in a single day may reach up to hundreds of thousands [1, 6] when a disease outbreaks, and thus our proposed algorithm is a good choice to identify the potentially infected users.



(a) Brightkite (b) Gowalla

Figure 6: The transmission chains. Users within the same rectangle are co-located at time t_x, and $\forall x < y$, $t_x < t_y$. A colored arrow indicates user movement. The latitude and longitude values are the identifiers for the venues in the datasets.

Secondly, we exhibit the potential of our proposed method on revealing the transmission chains. As discussed in Section 5.2, our algorithms can be extended to compute earliest-arrival paths and thus are able to reveal the transmission chains. Figure 6 presents two examples of the transmission chain between the source of infection (mark with "s") and one of the infected users (mark with "d") on the Brightkite and Gowalla datasets, respectively. In addition, our proposed method also helps uncover the high-risk venues, e.g., the venues as listed in Figure 6.

6.2 Query Processing

In this analysis, we evaluate the performance of OReach, OReach$^+$, TC-query, and TBP-query when answering single-pair reachability queries (SPRQ), single-source reachability queries (SSRQ), and earliest-arrival path queries (EAPQ). In the experiment, we generate 1000 random vertex pairs (or vertices for SSRQ). Note that when generating the time interval $[I_s, I_e]$ for a query, we generate I_s randomly and generate the duration d (i.e., $d = I_e - I_s$) following the power-law distribution. To ensure Lines 11 to 19 in Algorithm 1 can be invoked for a vertex pair (u, w), the generated I_s and d should satisfy $I_s \in [1, t_s]$ and $d \geq t_e - I_s$, where t_s is the maximum starting time from u and t_e is the minimum ending time to w.

Table 3 shows the average query processing time of different algorithms. For SPRQ, we distinguish the results for answering positive queries (i.e., queries that OReach returns true) from those for answering negative queries (i.e., queries that OReach returns false). For EAPQ, we only report the results of effective queries (i.e., queries that TBP-query can find a path), as otherwise, the query time is the same as that TBP-query for answering negative SPRQ. We use the same queries as SPRQ (the positive cases).

According to Table 3, when answering SPRQ, OReach$^+$ performs better than OReach, which indicates that the direction-optimizing BFS can substantially shrink the online searching space. However, OReach$^+$ is still at least an order of magnitude slower than TBP-query and TC-query. The reason is that both TBP-query and TC-query are index-based algorithms, while OReach$^+$ may need to traverse the whole graph for a query. When comparing TBP-query with TC-query, our TBP-query is faster than TC-query with up to two orders (on positive cases) or one order (negative cases) of magnitude. The reason is that our TBP-Index is a complete index that can correctly answer all queries, while TC-Index is a partial index, and TC-query has to perform graph traversal for the uncovered queries. One interesting observation is that TBP-query performs much better than TC-query when answering positive queries. This is reasonable as TBP-query only needs to scan the label sets of the query vertices and returns true immediately if a common vertex exists with time constraints satisfied, while TC-Index may have to traverse the graph for the query. For example, on the EP dataset, OReach and OReach$^+$ spend tens of milliseconds on average for answering both positive and negative queries, and TC-query needs more than 1200 μs for a positive query and 90 μs for a negative query. In comparison, TBP-query takes less than 5 μs in both cases.

When answering SSRQ, TBP-query is faster than TC-query with up to two orders of magnitude. This is because TBP-query is based on a complete index and admits an efficient computation of single-source reachability queries (Section 5.1). In comparison, TC-query

has to perform $|U(G)|$ times of reachability queries, where graph traversals and redundant checkings incur high overheads. Additionally, TC-query cannot be directly applied to answer EAPQ due to the index's incompleteness, while our TBP-query can answer such queries with reasonable time, as shown in Table 3.

6.3 Index Construction

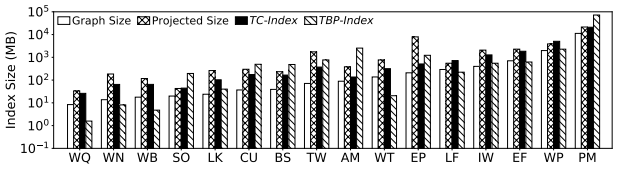


Figure 7: Index Size

Index Size. Figure 7 reports the index size for each dataset. As a comparison, we also report the related graph size and the size of the projected graph (denoted as "Projected Size"), both of which are computed based on the size of the edge list. Note that with the same vertex order, TBP-build, TBP-build*, and TBP-build*-PL output identical indexes, and thus we only report one TBP-Index size. According to Figure 7, the size of the projected graph is substantially larger than the original graph size. Specifically, on the EP dataset, the original graph occupies only 208.6 MB space, while its projected graph takes more than 7.9 GB. In terms of the index size, although TBP-Index is a complete index for answering reachability queries, the index sizes are only 0.1× - 28× of the (temporal bipartite) graph sizes. Moreover, TBP-Index is smaller than TC-Index (a partial index) on 9 out of 16 datasets. The reason is that TBP-Index is a minimal index, and it excludes the index entries that can be covered by the others. However, TC-Index is not minimal and involves redundant index entries. For instance, on the WT dataset, the TC-Index occupies 317.1 MB, while our TBP-Index takes only about 20.5MB. In addition, to make our index support SSRQ, we need an additional inverted in-label set, which is 0.47× - 0.51× the size of the TBP-Index on all the evaluated datasets. To support EAPQ, the path-aware TBP-Index is 1.5× - 2.0× the size of the TBP-Index.

Indexing Time. Table 4 reports the running time of different index construction algorithms, where "–" denotes the cases that cannot finish within 10^5s. We can observe that TBP-build and TBP-build* are slower than TC-build, which is expected as they both compute a complete and minimal index, and need additional time to check if an index entry is redundant. Fortunately, the computation of TBP-build* can be further accelerated by parallelization (Section 4.3). From Table 4, TBP-build*-PL ($t = 32$) is up to 19× faster than TBP-build* and TBP-build. Note that even though the computation of TC-build can also be parallelized, it obtains limited benefits from parallelization. In Table 4, TC-build-PL ($t = 32$) is only 1.1×-4.7× faster than TC-build. In total, both our TBP-build*-PL ($t = 32$) and TC-build-PL can finish the computation on graphs with hundreds of millions of edges within reasonable time. As discussed in Section 6.1 and Section 6.2, a complete and minimal index not only admits an efficient query processing process but also is the key to support path queries. Taking these performance gains into account, the indexing overhead of TBP-build*-PL is moderate and negligible.

Table 4: The indexing time (in sec by default)

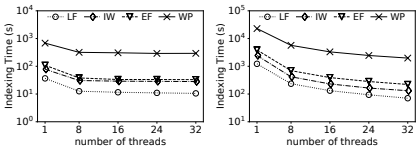
Name	Partial Index		Complete and Minimal Index (TBP-)			
	TC-build	TC-build-PL	build	build+	build*	build*-PL
WQ	1.16	1.11	5.89h	2.61	0.41	0.13
WN	6.26	4.12	26.48h	9.84	4.21	0.38
WB	3.81	2.98	20.15h	7.38	1.36	0.21
SO	2.66	1.55	25.41h	184.58	51.88	6.94
LK	6.94	2.43	-	47.33	29.66	2.39
CU	14.03	4.67	-	1158.59	710.16	70.27
BS	10.09	4.76	-	534.20	447.13	49.02
TW	46.43	15.02	-	1207.85	869.89	112.00
AM	16.75	9.17	-	3160.54	2016.39	320.63
WT	27.65	23.76	-	202.84	88.5	5.00
EP	215.17	45.88	-	7451.13	5007.3	339.92
LF	36.19	10.55	-	5463.48	1226.39	70.59
IW	78.58	28.16	-	7031.39	2453.3	131.62
EF	110.54	38.11	-	4.48h	3875	221.00
WP	568.35	292.78	-	-	6.36h	1256.88
PM	1021.98	523.90	-	-	-	4.22h

We then compare TBP-build with TBP-build* to demonstrate the effectiveness of the proposed techniques, i.e., edge partition and time-priority-based traversal. In Table 4, we additionally report the indexing time of TBP-build+, which only considers the time-priority-based traversal technique. We can see that TBP-build can only complete the index construction for 4 small datasets, while TBP-build+ and TBP-build* can finish the computation on all the datasets. Specifically, TBP-build+ is faster than TBP-build by up to four orders of magnitude in the datasets where TBP-build can finish, which validates the efficacy of the time-priority-based traversal technique. By further applying the edge partition technique, TBP-build* can speed up the computation of TBP-build+ by up to 7 times of magnitude. Finally, TBP-build* is more efficient than the baseline index construction algorithm TBP-build. For instance, TBP-build takes about 20 hours indexing on WB, while TBP-build* spends less than 2 seconds. In addition, to support SSRQ and EAPQ, computing the inverted in-label set consumes less than 100 seconds for each dataset, and the indexing time of the path-aware TBP-Index is 1.02×-1.36× of the TBP-Index construction time, respectively.

Speedup. This experiment studies the parallelization performance of TBP-build*-PL (Section 4.3) and TC-build-PL (Remark 2), and Figure 8 shows the indexing time of them by varying the number of threads t. Note that the patterns are similar across all datasets. Thus, only the results for the four representative datasets are reported.

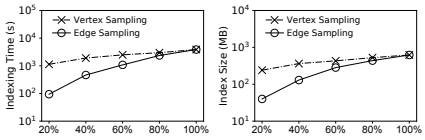
According to Figure 8(a), the curves of TC-build-PL tends to be flat, which indicates that TC-build-PL obtains limited benefits (less than 4.7×) with parallelization. The reasons are mainly two-fold. First, several parts of TC-build-PL program have to be implemented as sequential (e.g., index construction must follow the topological level order), and according to Amdahl's law [10], the parallelization performance is limited. Second, the performance of TC-build-PL relies on the number of vertices that belong to the same topological level, and for the level with few vertices, parallelization cannot be fully effective. For example, on the datasets IW, on average there are fewer than 21 vertices per topological level.

In comparison, our proposed TBP-build*-PL performs well with parallelization. Based on Figure 8(b), we can observe that all the curves demonstrate reasonable decreasing trends as the number of threads increases. The benefits from 1 to 16 threads are substantial.



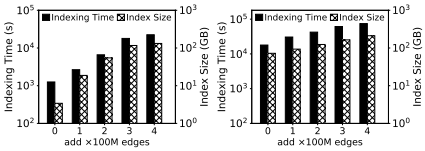
(a) TC-build-PL (b) TBP-build*-PL

Figure 8: Parallelization performance



(a) EF (b) EF

Figure 9: Scalability of TBP-build*



(a) WP (b) PM

Figure 10: Scalability of TBP-build*-PL

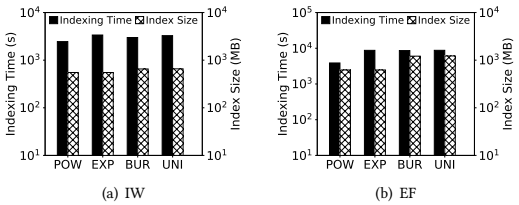
After 16, the reward ratio flattens due to the cost of thread scheduling. At $t = 32$ threads, parallelization can speed up the computation by 11× to 19× on the four representative datasets, which attributes to the nice lock- and atomic-free property.

Scalability. We evaluate the scalability of TBP-build* by varying the graph sizes. For the evaluated dataset, we randomly sample vertices and edges from 20% to 100%. When sampling vertices, we sample from both $U(G)$ and $L(G)$, and derive the induced subgraph based on the sampled vertices. When sampling edges, we construct graphs using the sampled edges. We only report the results of the EF dataset, while the other datasets demonstrate similar trends. Figure 9 reports the results of TBP-build* with respect to both indexing time and index size. We can observe that the increasing trend of all the curves becomes smoother as the graph size or the graph density gets larger. Therefore, TBP-build* is scalable.

To challenge large-scale graphs, we evaluate the performance of TBP-build*-PL ($t = 32$) by adding edges into the two largest datasets, i.e., WP and PM. Specifically, the size of newly added edges ranges from 100M to 400M, by which the sizes of WP and PM will exceed 500M and 1.1B, respectively. Figure 10 reports the results of TBP-build*-PL with respect to indexing time and index size on both datasets. Accordingly, we can observe that TBP-build*-PL can handle graphs at billion scale.

6.4 Varying Duration Distribution

In this analysis, we evaluate the performance of our algorithms (i.e., TBP-build* and TBP-query) by varying the duration distribution models. In addition to the power-law distribution model (denoted by POW) [11], we also consider the models of bursty distribution (BUR) [33], exponential distribution (EXP), and uniform distribution (UNI). The generated duration ranges from 1 hour to 48 hours. We follow the rules in [49] and set $\lambda = \frac{48}{\beta}$ for the EXP model, where $\beta = 1.7$. For the BUR model, we adopt the two-state model as illustrated in [33], and the parameters are set to $\beta_1 = 1.2$ and $\beta_2 = 2.2$, respectively. Note that only results of the datasets IW and EF are reported as the patterns for the others are similar.



(a) IW (b) EF

Figure 11: Performance while varying distribution models

Figure 11 shows the indexing time and index size of TBP-build* under different duration distribution models. We can observe that TBP-build* runs slightly faster when the duration is generated from the POW distribution, while it performs similarly on the other three distributions. This is because, under the POW distribution, most of the generated durations are short, and thus two different edges are less likely to form a time-overlapping wedge. Note that the time difference between the POW distribution and the other distributions is limited. The reason is that with short durations, the ending time of the time-overlapping wedges is relatively small, which increases the possibility to reach the other vertices, and consequently compensate the speedup as mentioned above.

Table 5: Query time (in μs) while varying duration models

Name	Positive Queries				Negative Queries			
	POW	EXP	BUR	UNI	POW	EXP	BUR	UNI
IW	2.99	3.31	3.42	3.47	3.64	4.49	3.97	3.94
EF	2.43	3.46	3.95	3.49	2.60	3.97	4.15	4.25

Table 5 shows the query time of TBP-query when the duration of queries generated by different models. The result for each case is an average of 1000 queries. In general, the performance of TBP-query is negatively correlated to the index size (Figure 11). For example, on the EF dataset, TBP-query under POW and EXP runs faster than that under the BUR and UNI models, since the index sizes under POW and EXP are smaller. An interesting observation is that TBP-query with POW runs slightly faster than TBP-query with EXP even though the used indexes are of similar size (Figure 11). The reason is that the duration for the queries generated by POW is relatively short, and thus the process of searching index may finish earlier.

7 CONCLUSION

In this paper, we study the temporal bipartite reachability problem. We propose an indexing-based solution that can support all possible single-pair reachability queries. Several techniques are devised to compute the index efficiently. Moreover, we show how to extend the computed index to support fast single-source reachability queries and earliest-arrival path queries. Finally, we conduct extensive experiments on 16 real-world graphs to demonstrate the effectiveness and efficiency of our proposed techniques.

ACKNOWLEDGMENTS

Xuemin Lin is supported by ARC DP200101338. Wenjie Zhang is supported by ARC DP210101393 and ARC DP200101116. Lu Qin is supported by ARC FT200100787. Ying Zhang is supported by FT170100128 and ARC DP210101393.

REFERENCES

[1] Coronavirus Research Center: https://coronavirus.jhu.edu/.
[2] KONECT. http://konect.cc/
[3] NSW Government: https://www.nsw.gov.au/covid-19/covid-safe/customer-record-keeping/qr-codes#covid-safe-check-in-with-the-service-nsw-app.
[4] Singapore Government: https://www.gov.sg/article/covid-19-resources.
[5] SNAP: http://snap.stanford.edu/data.
[6] https://covid.cdc.gov/covid-data-tracker/#trends_dailytrendscases.
[7] Wolfram Mathworld: https://mathworld.wolfram.com/RandomNumber.html.
[8] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. 2012. Hierarchical Hub Labelings for Shortest Paths. In *European Symposium on Algorithms*. 24–35.
[9] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.
[10] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.
[11] Albert-Laszlo Barabasi. 2005. The origin of bursts and heavy tails in human dynamics. *Nature* 435, 7039 (2005), 207–211.
[12] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–10.
[13] Binh-Minh Bui-Xuan, Afonso Ferreira, and Aubin Jarry. 2003. Computing Shortest, Fastest, and Foremost Journeys in Dynamic Networks. *International Journal of Foundations of Computer Science* 14, 2 (2003), 267–285.
[14] Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. 2019. The Computational Complexity of Finding Temporal Paths under Waiting Time Constraints. *CoRR* abs/1909.06437 (2019). http://arxiv.org/abs/1909.06437
[15] Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-based Algorithms for Pattern Matching on DAGs. In *Proceedings of the VLDB Endowment*. 493–504.
[16] Xiaoshuang Chen, Longbin Lai, Lu Qin, and Xuemin Lin. 2020. StructSim: Querying Structural Node Similarity at Billion Scale. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. 1950–1953.
[17] Xiaoshuang Chen, Longbin Lai, Lu Qin, and Xuemin Lin. 2021. Efficient structural node similarity computation on billion-scale graphs. *The VLDB Journal* (2021), 1–23.
[18] Xiaoshuang Chen, Longbin Lai, Lu Qin, Xuemin Lin, and Boge Liu. 2020. A Framework to Quantify Approximate Simulation on Graph Data. *arXiv preprint arXiv:2010.08938* (2020).
[19] Yangjun Chen and Yibin Chen. 2011. Decomposing DAGs into spanning trees: A new way to compress transitive closures. In *2011 IEEE 27th International Conference on Data Engineering*. 1007–1018.
[20] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and distance queries via 2-hop labels. In *SIAM Journal on Computing*. 937–946.
[21] 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.
[22] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective community search over large spatial graphs. *PVLDB* 10, 6 (2017), 709–720.
[23] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *PVLDB* 9, 12 (2016), 1233–1244.
[24] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *The VLDB Journal* 29, 1 (2020), 353–392.
[25] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V.S. Lakshmanan, and Xuemin Lin. 2019. Efficient algorithms for densest subgraph discovery. *PVLDB* 12, 11 (2019), 1719–1732.
[26] Yizhang He, Kai Wang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Exploring Cohesive Subgraphs with Vertex Engagement and Tie Strength in Bipartite Graphs. *Information Sciences* (2021). https://doi.org/10.1016/j.ins.2021.04.027
[27] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *ACM Transactions on Database Systems (TODS)* 15, 4 (1990), 558–598.
[28] Glen Jeh and Jennifer Widom. 2002. Simrank: a measure of structural-context similarity. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 538–543.
[29] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop Doubling Label Indexing for Point-to-Point Distance Querying on Scale-Free Networks. *Proceedings of the VLDB Endowment* 7, 12 (2014).
[30] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 813–826.
[31] Takeya Kasukawa, Masahiro Sugimoto, Akiko Hida, Yoichi Minami, Masayo Mori, Sato Honma, Ken-ichi Honma, Kazuo Mishima, Tomoyoshi Soga, and Hiroki R Ueda. 2012. Human blood metabolite timetable indicates internal body time. *Proceedings of the National Academy of Sciences* 109, 37 (2012), 15036–15041.
[32] David Kempe, Jon M. Kleinberg, and Amit Kumar. 2000. Connectivity and inference problems for temporal networks. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*. 504–513.
[33] Jon Kleinberg. 2003. Bursty and hierarchical structure in streams. *Data mining and knowledge discovery* 7, 4 (2003), 373–397.
[34] Wentao Li, Miao Qiao, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Scaling Distance Labeling on Small-World Networks. In *Proceedings of the 2019 International Conference on Management of Data*. 1060–1077.
[35] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *Proceedings of the VLDB Endowment* 11, 4 (2017), 445–457.
[36] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core computation: An index-based approach. In *The World Wide Web Conference*. 1130–1141.
[37] Clare M O'Connor, Jill U Adams, and Jennifer Fairman. 2010. Essentials of cell biology. *Cambridge, MA: NPG Education* 1 (2010), 54.
[38] Georgios A Pavlopoulos, Panagiota I Kontou, Athanasia Pavlopoulou, Costas Bouyioukos, Evripides Markou, and Pantelis G Bagos. 2018. Bipartite graphs in systems biology and medicine: a survey of methods and applications. *GigaScience* 7, 4 (2018), giy014.
[39] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2021. Efficient Hop-constrained s-t Simple Path Enumeration. *The VLDB Journal* (2021), 1–24.
[40] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *Proceedings of the VLDB Endowment* 13, 6 (2020), 812–825.
[41] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained st simple path enumeration. *Proceedings of the VLDB Endowment* 13, 4 (2019), 463–476.
[42] Tytus Pikies and Marek Kubale. 2020. Chromatic cost coloring of weighted bipartite graphs. *Appl. Math. Comput.* 375 (2020), 125073.
[43] Ahmet Erdem Sariyüce and Ali Pinar. 2018. Peeling Bipartite Networks for Dense Subgraph Discovery. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 504–512.
[44] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs.. In *EDBT*, Vol. 15. 121–132.
[45] Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1009–1020.
[46] Klaus Simon. 1988. An Improved Algorithm for Transitive Closure on Acyclic Digraphs. *Theoretical Computer Science* 58 (1988), 325–346.
[47] 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.
[48] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 845–856.
[49] Lucien DJ Valstar, George HL Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 345–358.
[50] Lucien D. J. Valstar, George H. L. Fletcher, and Yuichi Yoshida. 2017. Landmark Indexing for Evaluation of Label-Constrained Reachability Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 345–358.
[51] Sebastiaan J. van Schaik and Oege de Moor. 2011. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 913–924.
[52] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. 2006. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *2006 IEEE 22nd International Conference on Data Engineering (ICDE)*. 75.
[53] Jun Wang, Arjen P. de Vries, and Marcel J. T. Reinders. 2006. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. 501–508.
[54] Kai Wang, Xin Cao, Xuemin Lin, Wenjie Zhang, and Lu Qin. 2018. Efficient computing of radius-bounded k-cores. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 233–244.
[55] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex priority based butterfly counting for large-scale bipartite networks. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1139–1152.
[56] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2020. Efficient bitruss decomposition for large-scale bipartite graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 661–672.
[57] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2021. Towards efficient solutions of bitruss decomposition for large-scale bipartite graphs. *The VLDB Journal* (2021), 1–24.

[58] Kai Wang, Zhang Wenjie, Xuemin Lin, Ying Zhang, Lu Qin, and Yuting Zhang. 2021. Efficient and Effective Community Search on Large-scale Bipartite Graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE.

[59] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 967–982.

[60] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability Querying: An Independent Permutation Labeling Approach. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1191–1202.

[61] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability querying: an independent permutation labeling approach. *The VLDB Journal* 27, 1 (2018), 1–26.

[62] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently Answering Span-Reachability Queries in Large Temporal Graphs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1153–1164.

[63] 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.

[64] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 145–156.

[65] Hongqiang Yan, Yan Jiang, and Guannan Liu. 2018. Telecomm Fraud Detection via Attributed Bipartite Network. In *2018 15th International Conference on Service Systems and Service Management (ICSSSM)*. IEEE, 1–6.

[66] Yixing Yang, Yixiang Fang, Maria E. Orlowska, Wenjie Zhang, and Xuemin Lin. 2021. Efficient Bi-Triangle Counting for Large Bipartite Networks. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 984–996.

[67] Hilmi Yildirim, Vineet Chaoji, and Mohammed Javeed Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *Proceedings of the VLDB Endowment* 3, 1 (2010), 276–284.

[68] Jeffrey Xu Yu and Jiefeng Cheng. 2010. Graph Reachability Queries: A Survey. In *Managing and Mining Graph Data*. 181–215.

[69] 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. *The VLDB Journal* 28, 6 (2019), 871–896.

[70] Matteo Zignani. 2011. Human mobility model based on time-varying bipartite graph. In *2011 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*. 1–4.