HR-Index: An Effective Index Method for Historical Reachability Queries over Evolving Graphs

YAJUN YANG, College of Intelligence and Computing, Tianjin University, China and State Key Laboratory of Communication Content Cognition, People's Daily Online, China
HANXIAO LI, College of Intelligence and Computing, Tianjin University, China
XIANGJU ZHU, College of Intelligence and Computing, Tianjin University, China
JUNHU WANG, School of Information and Communication Technology, Griffith University, Australia
XIN WANG*, College of Intelligence and Computing, Tianjin University, China
HONG GAO, School of Computer Science and Technology, Zhejiang Normal University, China

Reachability query is a fundamental problem and has been well studied on static graphs. However, in the real world, the graphs are not static but always evolving over time. In this paper, we study the problem of historical reachability query on evolving graphs. We propose a novel index, named HR-Index, which integrates complete and correct historical reachability information of the evolving graph. A historical reachability query on an evolving graph can be converted into a static reachability query on its HR-Index and thus query efficiency can be improved significantly. We also propose two optimization techniques to reduce the size of HR-Index effectively. We confirm the effectiveness and efficiency of our method through conducting extensive experiments on real-life datasets. Experimental results show both vertex and edge size of HR-Index are far smaller than that of the evolving graphs and our method has at least an order of magnitude improvement in time and space efficiency compared to the state-of-the-art method.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms**.

Additional Key Words and Phrases: evolving graph, reachability query, index

ACM Reference Format:
Yajun Yang, Hanxiao Li, Xiangju Zhu, Junhu Wang, Xin Wang, and Hong Gao. 2023. HR-Index: An Effective Index Method for Historical Reachability Queries over Evolving Graphs. *Proc. ACM Manag. Data* 1, 2, Article 127 (June 2023), 25 pages. https://doi.org/10.1145/3589272

1 INTRODUCTION

Reachability query is a fundamental problem and has been well studied on static graphs, but it has not attracted much attention for evolving graphs. In this paper, we study two kinds of historical reachability queries, disjunctive and conjunctive reachability queries on evolving graphs,

*Corresponding author

Authors' addresses: Yajun Yang, yjyang@tju.edu.cn, College of Intelligence and Computing, Tianjin University, China and State Key Laboratory of Communication Content Cognition, People's Daily Online, China; Hanxiao Li, hanxiaoli@tju.edu.cn, College of Intelligence and Computing, Tianjin University, China; Xiangju Zhu, zhuxiangjv@tju.edu.cn, College of Intelligence and Computing, Tianjin University, China; Junhu Wang, j.wang@griffith.edu.au, School of Information and Communication Technology, Griffith University, Australia; Xin Wang, wangx@tju.edu.cn, College of Intelligence and Computing, Tianjin University, China; Hong Gao, honggao@hit.edu.au, School of Computer Science and Technology, Zhejiang Normal University, China.

2836-6573/2023/6-ART127 $15.00
https://doi.org/10.1145/3589272

which is to answer whether a vertex is reachable from another vertex in at least one snapshot or in all the snapshots during a specified time interval. There are several real applications that can benefit from historical reachability queries. For example, in protein-protein interaction networks, it is important to investigate whether two proteins participate in a common biological process or molecular function [9]. Conjunctive reachability can help to monitor whether these two proteins continuously belong to the same biological organization in a specified period. In money transaction monitoring, a user account can be regarded as a vertex and a money transaction can be regarded as an edge between two user accounts. Disjunctive reachability can help identify whether there exists a transaction path between two suspicious accounts in a specified short period over a long monitoring period.

The main idea of reachability query method on static graphs is to construct various indexes to make reachability query more efficient. However, these methods cannot be used for evolving graphs because an index constructed for a snapshot is not applicable to other snapshots due to the deletions or insertions of the vertices and edges. A naive method is to answer the query by BFS/DFS traversal on every snapshot for a given time interval, which suffers from high query time overhead. An alternative method is to build an index for every snapshot of the evolving graph but it is space inefficient when the evolving graph is with a large number of snapshots.

In recent years, a few works study reachability query on dynamic graphs or temporal graphs. For dynamic graphs, the existing works [14, 24, 27] study how to incrementally maintain reachability index for every deletion/insertion of nodes or edges when graphs evolve over time. These works essentially only consider reachability query on a single static graph, i.e., the current version of evolving graph. They are not suitable for answering historical reachability query because two vertices that are reachable in the current snapshot of evolving graphs may not be reachable at the past time points. Given a historical reachability query on time interval I, these methods will bring quite expensive time cost for updating index on every snapshot in time interval I. For temporal graphs, every edge has a time stamp to indicate when this edge is built/exists and the temporal reachability query is to determine whether there is a time respecting path between two vertices. Time respecting path is defined as a path in which the time points of the edges follow a non-decreasing order. Two vertices that are time-respectingly reachable may not be historically reachable because they may not be reachable in the same snapshot. For example, v_i can time-respectingly reach v_j by the path $v_i \rightarrow v_1 \rightarrow v_2 \rightarrow v_j$, where the edge (v_i, v_1), (v_1, v_2) and (v_2, v_j) in temporal graph are with time point 1, 2 and 3 respectively. Obviously, v_i and v_j are not reachable in the snapshot at time point 1. Therefore the methods for the temporal reachability query cannot be used for answering historical reachability query on evolving graphs. To the best of our knowledge, the TimeReach method proposed in [15] is state-of-the-art for historical reachability queries on evolving graphs. The main idea of TimeReach is to build a compact representation of graph snapshots, called "***version graph***", where each vertex and edge is annotated with a set of time intervals during which the corresponding vertex or edge exists in the evolving graph. The version graph can be considered as a static graph in which every vertex and edge has a time interval and then historical reachability query can be answered by the version graph. It is necessary to utilize BFS or DFS traversal with checking time interval intersection on the version graph, because a vertex v_j may not be reachable from another vertex v_i for all the snapshots in the evolving graph even though there is a path from v_i to v_j in the version graph. Therefore, existing efficient index methods for reachability queries on static graphs cannot be used.

In this paper, we propose a novel index, named HR-Index, for historical reachability query on evolving graphs. An HR-Index essentially is a single condensed graph integrating complete and correct historical reachability information of an evolving graph. Both vertex and edge sizes of HR-Index are far smaller than that of the original evolving graph. A historical reachability query on

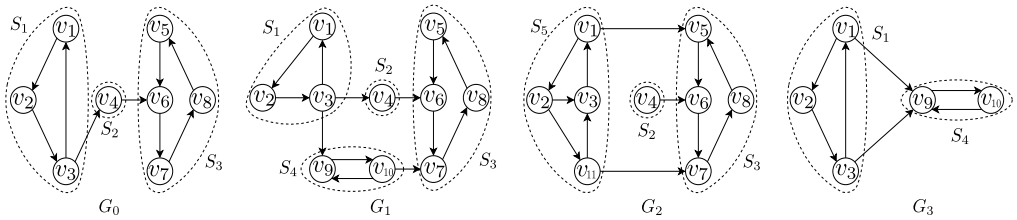


Fig. 1. An example of an evolving graph $\mathcal{G}$

the evolving graph can be converted into a traditional reachability query on HR-Index with no limitation. Therefore, existing efficient methods for reachability query on static graphs can be used on HR-Index straightforwardly. The main contributions are summarized below. First, we design HR-Index such that the existing method for static reachability query can be used for historical reachability query, then the time and space efficiency can be improved significantly. We prove HR-Index is equivalent to the evolving graph for historical reachability query. Experimental results on real-life datasets validate both vertex and edge sizes of HR-Index are far smaller than that of the evolving graphs. For example, SOF dataset has 17,723,799 vertex and 395,547,708 edges but its HR-Index only has 422,782 vertices and 1,017,387 edges. Second, we propose two optimization techniques, redundant nodes deletion and SCC merging, to further reduce the size of HR-Index. Finally, we conduct extensive experiments on real-life datasets to validate the effectiveness and efficiency of our method. The experimental results show our method has at least an order of magnitude improvement in time and space efficiency compared to the state-of-the-art method.

The rest of this paper is organized as follows. Section 2 introduces the problem of historical reachability query. Section 3 proposes HR-Index and Section 4 introduces how to answer queries by utilizing HR-Index. Section 5 proposes two optimization techniques, redundant nodes deletion and SCC merging. The experimental results are presented in Section 6 and the related works are introduced in Section 7. We conclude this paper in Section 8.

2 PROBLEM STATEMENT

An evolving graph, denoted as $\mathcal{G} = (G_0, \cdots, G_{\|\mathcal{G}\|-1})$, is defined as a sequence of directed graphs, where every $G_x = (V_x, E_x)$ is a *snapshot* of $\mathcal{G}$ at time point t_x with a set V_x of vertices and a set E_x of edges. $\|\mathcal{G}\|$ is the number of snapshots in $\mathcal{G}$ and it is called the length of $\mathcal{G}$. Specifically, $\mathcal{G}_I$ is a snapshot sub-sequence of an evolving graph $\mathcal{G}$ at time interval $I = [t_x, t_y]$, i.e., $\mathcal{G}_I = (G_x, \cdots, G_y)$. Fig. 1 illustrates an example of evolving graph $\mathcal{G}$ with four snapshots G_0, G_1, G_2 and G_3, where $||\mathcal{G}|| = 4$. In real scenarios, these snapshots are always pre-given by users in different time granularity, e.g., daily, weekly and monthly. In this paper, we study how to efficiently answer historical reachability queries on the given snapshot sequences.

Reachability query is a fundamental problem on graphs. Given a static directed graph $G = (V, E)$ and two vertices $v_i, v_j \in V$, we say v_j is reachable from v_i, denoted as $v_i \rightsquigarrow v_j$, if there exists a path from v_i to v_j in G. Different from the static graphs, we consider the following two types of reachability on evolving graphs.

Definition 2.1: (**Reachability on Evolving Graphs**). Given an evolving graph $\mathcal{G}$, a time interval $I = [t_x, t_y]$ and two vertices v_i and v_j in $\mathcal{G}$, there are two types of reachability for v_i and v_j on $\mathcal{G}_I$,

- ***disjunctive reachability***: we say v_j is *disjunctive reachable* from v_i at time interval I, denoted as $v_i \overset{I_\vee}{\rightsquigarrow} v_j$, if there exists a path from v_i to v_j in at least one snapshot in $\mathcal{G}_I$, i.e., $\exists G_z \in \mathcal{G}_I, v_i \rightsquigarrow v_j$ in G_z.

- ***conjunctive reachability***: we say v_j is *conjunctive reachable* from v_i at time interval I, denoted as $v_i \overset{I_\wedge}{\rightsquigarrow} v_j$, if there exists a path from v_i to v_j in every snapshot in $\mathcal{G}_I$, i.e., $\forall G_z \in \mathcal{G}_I$, $v_i \rightsquigarrow v_j$ in G_z.

□

Note that in the above definition, $\mathcal{G}_I$ is a sub-sequence of the evolving graph at time interval I. In practice, our method also can work well for an arbitrary $\mathcal{G}_X$, which consists of the snapshots selected from $\mathcal{G}$ arbitrarily. Our method can answer whether two vertices v_i and v_j are disjunctive and conjunctive reachable on $\mathcal{G}_X$, even though two successive snapshots in $\mathcal{G}_X$ may not be successive in $\mathcal{G}$. For simplicity, we only consider the reachability query on $\mathcal{G}_I$ within a time interval I in the rest of this paper. Our method also can be easily extended to handle arbitrary logic combination of conjunctive and disjunctive historical reachability queries, which make our method more applicable for real world problems. For example, the query $v_i \overset{I_{1\vee} \wedge I_{2\wedge}}{\rightsquigarrow} v_j$ can be decomposed to two queries $v_i \overset{I_{1\vee}}{\rightsquigarrow} v_j$ and $v_i \overset{I_{2\wedge}}{\rightsquigarrow} v_j$. The orignal query can be answerd by "true" if these two queries have positive answers.

3 HR-INDEX FOR EVOVLING GRAPH

In this paper, we propose a novel index named HR-Index for historical reachability query on evolving graphs. The HR-Index for an evolving graph $\mathcal{G}$ is a single static graph constructed from $\mathcal{G}$ to integrate the historical reachability information. Every vertex in HR-Index is associated with a lifespan to indicate the snapshots containing this vertex. By HR-Index, a historical reachability query on an evolving graph can be regarded as a traditional reachability query on a static graph. In this section, we first introduce lifespan, SCC-table and ON-table, which are used for building HR-Index, and then introduce how to build HR-Index.

3.1 Lifespan

Given an evolving graph $\mathcal{G}$, some vertices or edges may exist in several snapshots with distinct time points in $\mathcal{G}$. For every vertex v_i in $\mathcal{G}$, the ***lifespan*** of v_i, denoted as $L(v_i)$, is the set of all the time points that v_i appears in the corresponding snapshots of $\mathcal{G}$, that is, for any $t_x \in L(v_i)$, v_i is in the snapshot G_x of $\mathcal{G}$ at the time point t_x. Similarly, we use $L(v_i, v_j)$ to denote the lifespan of the edge (v_i, v_j) in $\mathcal{G}$. For example, in Fig. 1, v_4 is in the snapshots G_0, G_1 and G_2, then we have $L(v_4) = \{t_0, t_1, t_2\}$.

In this paper, we use bitset technique to store the lifespan for every vertex and edge in $\mathcal{G}$. The bitset of a vertex v_i (or an edge (v_i, v_j)) is a string consisting of 0 and 1. If $t_x \in L(v_i)$ (or $t_x \in L(v_i, v_j)$), then the x-th character of bitset is 1, otherwise, it is 0. By bitset technique, the intersection and union operations for the lifespans can be converted into the logical-AND and logical-OR operations on bitsets, which can effectively reduce the computational cost for historical reachability query. For the example in Fig. 1, $\mathcal{G} = \{G_0, G_1, G_2, G_3\}$ and the lifespan of v_4 is $L(v_4) = \{t_0, t_1, t_2\}$, then the bitset of v_4 is 1110.

3.2 Strongly Connected Component Table

A ***S**trongly **C**onnected **C**omponent* (or SCC) of a directed graph G, denoted as S_i is a maximal strongly connected subgraph of G. If two vertices belong to the same SCC, then they are reachable from each other. By regarding every SCC as a new vertex, the original graph can be converted into a ***D**irected **A**cyclic **G**raph* (or DAG). Therefore, most of existing works about reachability on the static graphs only need to consider how to answer the reachability query on a DAG by precomputing SCCs.

Table 1. SCC-table for the evolving graph $\mathcal{G}$ in Fig. 1

Vertex set	SCC with lifespan
$\{v_1, v_2, v_3\}$	$(S_1, \{t_0, t_1, t_3\}),\ (S_5, \{t_2\})$
$\{v_4\}$	$(S_2, \{t_0, t_1, t_2\})$
$\{v_5, v_6, v_7, v_8\}$	$(S_3, \{t_0, t_1, t_2\})$
$\{v_9, v_{10}\}$	$(S_4, \{t_1, t_3\})$
$\{v_{11}\}$	$(S_5, \{t_2\})$

However, as the graph evolves over time, its strongly connected components change as well. Two vertices in the same SCC in snapshot G_x may be not reachable in the next snapshot with vertex (or edge) deletions. On the contrary, two distinct SCCs in G_x may be merged into a new SCC in the next snapshot. In this paper, we utilize the existing algorithm, e.g., Tarjans algorithm, to identify the SCCs for every snapshot in $\mathcal{G}$ and these SCCs are maintained in a *Strongly Connected Component Table* (or SCC-table for simplicity). Note that if a vertex v_i does not belong to any SCC, we also consider it as a SCC which only has v_i. Every SCC in $\mathcal{G}$ is also associated with a lifespan $L(S_i)$, which indicates that all the snapshots at the time points in $L(S_i)$ have the same SCC S_i. Therefore, every vertex v_i has a set of SCCs, denoted as $SI(v_i) = \{(S_j, L(S_j))|v_i \in S_j\}$, which indicates the SCCs including v_i with their lifespans in $\mathcal{G}$. The SCC information $SI(v_i)$ of all the vertices are maintained in SCC-table. Note that two different vertices v_i and v_j may have the same SCC information, i.e., $SI(v_i) = SI(v_j)$. To reduce the space cost, all the vertices with the same $SI(v_i)$ are maintained in the same row in the SCC-table.

Running Example: In Fig. 1, every SCC in G_0, G_1, G_2, G_3 is marked with a dashed line. For SCC S_1 consisting of v_1, v_2, v_3, it appears in G_0, G_1, G_3, then $L(S_1) = \{t_0, t_1, t_3\}$. The SCC-table of $\mathcal{G}$ is shown in Table 1. Note that v_1, v_2 and v_3 have the same SCC $\{(S_1, \{t_0, t_1, t_3\}), (S_5, \{t_2\})\}$, thus they are maintained in the first row in Table 1 together.

If v_i and v_j are in the same SCC at the time point t_x, then the reachability query between v_i and v_j can be answered from SCC-table straightforwardly. With the SCC-table, every snapshot in $\mathcal{G}$ can be converted into a directed acyclic snapshot by regarding every SCC as a vertex. Fig. 2 shows the new evolving graph consisting of the DAGs converted from $\mathcal{G}$ in Fig. 1, every vertex in Fig. 2 is a SCC in Fig. 1. For simplicity, we only discuss how to answer the disjunctive and conjunctive reachability query on the evolving graph $\mathcal{G}$ in which every snapshot is a directed acyclic graph. To distinguish from the vertices and edges in the original evolving graph, we use S_i and (S_i, S_j) to represent the vertex and edge in $\mathcal{G}$ respectively in the following.

3.3 Outgoing Neighbor Table

HR-Index integrates the historical reachability information of $\mathcal{G}$ into a static graph. To construct HR-Index, we utilize a table, named *Outgoing Neighbor Table* (or ON-table for simplicity), to maintain the outgoing neighbor information for every vertex S_i in $\mathcal{G}$. In ON-table, the outgoing neighbors for every vertex S_i can be grouped into two categories: ***instant outgoing neighbor*** and ***interval outgoing neighbor***. Next, we will introduce how to obtain ON-table from an evolving graph $\mathcal{G}$.

Given a vertex S_i in $\mathcal{G}$, we use $N^+(S_i)$ and $N^-(S_i)$ to represent the sets of outgoing neighbors and incoming neighbors with their lifespans respectively, that is,

$$N^+(S_i) = \{(S_j, L(S_i, S_j))|(S_i, S_j) \in \mathcal{G} \text{ on } L(S_i, S_j)\}$$
$$N^-(S_i) = \{(S_j, L(S_j, S_i))|(S_j, S_i) \in \mathcal{G} \text{ on } L(S_j, S_i)\}$$

$L(S_i, S_j)$ (or $L(S_j, S_i)$) is the lifespan of the edge (S_i, S_j) (or (S_j, S_i)) in $\mathcal{G}$. $(S_j, L(S_i, S_j)) \in N^+(S_i)$ means (S_i, S_j) is an outgoing edge of S_i for the time points in $L(S_i, S_j)$ and thus S_j is an outgoing

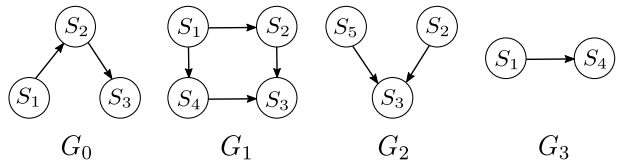


Fig. 2. The DAGs converted from $\mathcal{G}$ in Fig. 1

neighbor of S_i in $L(S_i, S_j)$. Similarly, $(S_j, L(S_j, S_i)) \in N^-(S_i)$ means S_j is an incoming neighbor of S_i in $L(S_j, S_i)$.

In ON-table, we only need to maintain the outgoing neighbors for every vertex S_i because an outgoing edge (S_i, S_j) of S_i in $L(S_i, S_j)$ must be an incoming edge of S_j in $L(S_i, S_j)$. The outgoing neighbors of every vertex S_i are grouped into two categories: instant outgoing neighbors and interval outgoing neighbors according to their corresponding outgoing edges at different time points. Let $L^-(S_i)$ denote the set of all the time points at which S_i has at least one incoming edge in G, that is,

$$L^-(S_i) = \bigcup_{(S_j, L(S_j,S_i)) \in N^-(S_i)} L(S_j, S_i) \tag{1}$$

For every outgoing neighbor $(S_j, L(S_i, S_j)) \in N^+(S_i)$, its lifespan $L(S_i, S_j)$ can be divided into two parts: instant part $L_1(S_i, S_j)$ and interval part $L_2(S_i, S_j)$, where

$$L_1(S_i, S_j) = L^-(S_i) \cap L(S_i, S_j) \tag{2}$$

and

$$L_2(S_i, S_j) = L(S_i, S_j) - L_1(S_i, S_j) \tag{3}$$

It indicates S_i has no incoming edge in G for $t \in L_2(S_i, S_j)$ and has at least one incoming edge for $t \in L_1(S_i, S_j)$. It is obvious that $L_1(S_i, S_j) \cap L_2(S_i, S_j) = \emptyset$ and $L_1(S_i, S_j) \cup L_2(S_i, S_j) = L(S_i, S_j)$. Therefore, every $(S_j, L(S_i, S_j)) \in N^+(S_i)$ also can be divided into two parts: instant part $(S_j, L_1(S_i, S_j))$ and interval part $(S_j, L_2(S_i, S_j))$. If $L_2(S_i, S_j) \neq \emptyset$, $(S_j, L_2(S_i, S_j))$ is maintained as an interval outgoing neighbor of S_i in ON-table. For the instant part $(S_j, L_1(S_i, S_j))$, if $L_1(S_i, S_j) \neq \emptyset$, then it is split into $|L_1(S_i, S_j)|$ instant outgoing neighbors maintained in the ON-table, where $|L_1(S_i, S_j)|$ is the number of the time points in $L_1(S_i, S_j)$, that is, every $(S_j, \{t_x\})$ for $t_x \in L_1(S_i, S_j)$ is maintained as an instant outgoing neighbor of S_i in ON-table. Let $N_1^+(S_i)$ and $N_2^+(S_i)$ denote the instant and interval outgoing neighbor set of S_i respectively, i.e.,

$$N_1^+(S_i) = \bigcup_{(S_j, L(S_i,S_j)) \in N^+(S_i)} \{(S_j, \{t_x\}) | t_x \in L_1(S_i, S_j)\} \tag{4}$$

and

$$N_2^+(S_i) = \bigcup_{(S_j, L(S_i,S_j)) \in N^+(S_i)} \{(S_j, L_2(S_i, S_j)\} \tag{5}$$

ON-table maintains instant outgoing neighbor set $N_1^+(S_i)$ and interval outgoing neighbor set $N_2^+(S_i)$ for every vertex S_i in $\mathcal{G}$.

Running Example: Table 2 shows the ON-table for the evolving graph in Fig. 2. Table 2 has three columns. The first column indicates the ID for every S_i in $\mathcal{G}$, the second and the third columns indicate the instant outgoing neighbor set $N_1^+(S_i)$ and interval outgoing neighbor set $N_2^+(S_i)$ of S_i respectively. For the example of the vertex S_2 in Fig. 2, it has an incoming neighbor S_1 at time point $\{t_0, t_1\}$ and an outgoing neighbor S_3 at time point $\{t_0, t_1, t_2\}$, thus $N^-(S_2) = \{(S_1, \{t_0, t_1\})\}$ and $N^+(S_2) = \{(S_3, \{t_0, t_1, t_2\})\}$. By Eq. (1), (2) and (3), we have $L^-(S_2) = \{t_0, t_1\}$, $L_1(S_2, S_3) = \{t_0, t_1\}$ and $L_2(S_2, S_3) = \{t_2\}$ respectively. Therefore, the instant outgoing neighbor set $N_1^+(S_2) = \{(S_3, \{t_0\}), (S_3, \{t_1\})\}$ and $N_2^+(S_2) = \{(S_3, \{t_2\})\}$ can be calculated by Eq. (4) and (5).

Table 2. The ON-table of the evolving graph $\mathcal{G}$ in Fig. 2

SCC	Instant outgoing neigbor	Interval outgoing neighbor
S_1	Null	$(S_2, \{t_0, t_1\}), (S_4, \{t_1, t_3\})$
S_2	$(S_3, \{t_0\}), (S_3, \{t_1\})$	$(S_3, \{t_2\})$
S_3	Null	Null
S_4	$(S_3, \{t_1\})$	Null
S_5	Null	$(S_3, \{t_2\})$

Note that Table 2 also keeps the complete historical reachability information for the evolving graph $\mathcal{G}$. Given an ON-table, we can reconstruct its corresponding evolving graph easily. Next, we will introduce how to construct HR-Index from ON-table which is a static graph keeping the complete historical reachability information for an evolving graph $\mathcal{G}$.

3.4 HR-Index Construction

The HR-Index of evolving graph $\mathcal{G}$, denoted as $H(\mathcal{G}) = (V_H, E_H)$, is a graph constructed from the ON-table of $\mathcal{G}$, where V_H and E_H are the sets of the vertices and edges in $H(\mathcal{G})$. To distinguish from the vertices in $\mathcal{G}$, we refer to a vertex in $H(\mathcal{G})$ as a "node". Every node and edge in $H(\mathcal{G})$ is in the form $\langle S_i, L_i\rangle$ and $(\langle S_i, L_i\rangle, \langle S_j, L_j\rangle)$ respectively, where S_i is a vertex in $\mathcal{G}$ and L_i is a lifespan derived from the ON-table. Note that there may be several distinct L_is for the same S_i, then there may be several nodes with the same S_i and there may exist the edge $(\langle S_i, L_i\rangle, \langle S_i, L_i'\rangle)$ in $H(\mathcal{G})$.

All the edges in $H(\mathcal{G})$ can be categorized into three cases: (1) created from the instant outgoing neighbor set in the ON-table; (2) created from the interval outgoing neighbor set in the ON-table; and (3) created by connecting two nodes in $H(\mathcal{G})$. We introduce how to create the edges for these three cases respectively.

Case (1): For every S_i, an edge $(\langle S_i, \{t_x\}\rangle, \langle S_j, \{t_x\}\rangle)$ is created for every $(S_j, \{t_x\}) \in N_1^+(S_i)$ in On-table. If node $\langle S_i, \{t_x\}\rangle$ or $\langle S_j, \{t_x\}\rangle$ does not exist in $H(\mathcal{G})$, we first create it. Obviously, these are $|N_1^+(S_i)|$ edges with at most $2|N_1^+(S_i)|$ nodes.

Case (2): For every S_i in the ON-table, we use $L^+(S_i)$ to denote the union of the lifespans $L_2(S_i, S_j)$ in $N_2^+(S_i)$, i.e.

$$L^+(S_i) = \bigcup_{(S_j, L(S_i,S_j)) \in N_2^+(S_i)} L_2(S_i, S_j) \tag{6}$$

We create a node $\langle S_i, L^+(S_i)\rangle$ if $L^+(S_i) \neq \emptyset$ and create the edge $(\langle S_i, L^+(S_i)\rangle, \langle S_j, L_2(S_i, S_j)\rangle)$ for every $(S_j, L(S_i, S_j)) \in N_2^+(S_i)$. Obviously, these are $|N_2^+(S_i)|$ edges with $|N_2^+(S_i)| + 1$ nodes. Note that every S_i has at most one $\langle S_i, L^+(S_i)\rangle$ in $H(\mathcal{G})$.

Case (3): For two nodes $\langle S_i, \{t_x\}\rangle$ and $\langle S_i, L\rangle$ of the same SCC S_i, if $|L| > 1$ and $t_x \in L$, then the edge $(\langle S_i, L\rangle, \langle S_i, \{t_x\}\rangle)$ will be created when it is not in $H(\mathcal{G})$.

From Case (1), (2) and (3), we get Lemma 3.1 straightforwardly.

Lemma 3.1: *For every edge $(\langle S_i, L_i\rangle, \langle S_j, L_j\rangle)$ in $H(\mathcal{G})$, we have $L_j \subseteq L_i$.* □

Lemma 3.2: *If an edge $(\langle S_j, L_j\rangle, \langle S_j, \{t_x\}\rangle)$ exists in $H(\mathcal{G})$, then there must exist an S_i such that (S_j, L_j) is an interval outgoing neighbor of S_i, i.e., $(S_j, L_j) \in N_2^+(S_i)$.* □

Proof. $(\langle S_j, L_j\rangle, \langle S_j, \{t_x\}\rangle)$ in $H(\mathcal{G})$ indicates $t_x \in L_j$ and $|L_j| > 1$. Therefore, the node $\langle S_j, L_j\rangle$ must be $\langle S_j, L^+(S_j)\rangle$ of S_j or $\langle S_j, L_2(S_i, S_j)\rangle$ for some S_i. On the other hand, both $\langle S_j, L_j\rangle$ and $\langle S_j, \{t_x\}\rangle$ in $H(\mathcal{G})$ means S_j has at least one incoming edge at time point t_x, and thus $\langle S_j, L_j\rangle$ cannot be $\langle S_j, L^+(S_j)\rangle$ of S_j, then (S_j, L_j) is an interval outgoing neighbor of S_i. □

The pseudo-code of building HR-Index $H(\mathcal{G})$ is shown in Algorithm 1. For every S_i in ON-table, Algorithm 1 creates the edges for the instant outgoing neighbors (line 3-9) as per discussion in

Algorithm 1: BUILD-HR-INDEX ($T_o(\mathcal{G})$)

Input: The ON-table $T_o(\mathcal{G})$ of the evolving graph $\mathcal{G}$
Output: The HR-Index $H(\mathcal{G})$ of the evolving graph $\mathcal{G}$

1 **Initialize:** $V_H \leftarrow \emptyset$ and $E_H \leftarrow \emptyset$;
2 **for** *each SCC* S_i *in* $T_o(\mathcal{G})$ **do**
3 **if** *the instant outgoing neighbor set* $N_1^+(S_i) \neq \emptyset$ **then**
4 **for** *each* $(S_j, \{t_x\}) \in N_1^+(S_i)$ **do**
5 **if** *the node* $(S_i, \{t_x\}) \notin V_H$ **then**
6 CREATE-NODE $(H(\mathcal{G}), \langle S_i, \{t_x\}\rangle)$;
7 **if** *the node* $(S_j, \{t_x\}) \notin V_H$ **then**
8 CREATE-NODE $(H(\mathcal{G}), \langle S_j, \{t_x\}\rangle)$;
9 $E_H \leftarrow E_H \cup (\langle S_i, \{t_x\}\rangle, \langle S_j, \{t_x\}\rangle)$;
10 **if** *the interval outgoing neighbor set* $N_2^+(S_i) \neq \emptyset$ **then**
11 $L^+(S_i) \leftarrow \bigcup_{(S_j, L(S_i,S_j)) \in N_2^+(S_i)} L_2(S_i, S_j)$;
12 CREATE-NODE $(H(\mathcal{G}), \langle S_i, L^+(S_i)\rangle)$;
13 **for** *each* $(S_j, L_2(S_i, S_j)) \in N_2^+(S_i)$ **do**
14 **if** *the node* $(S_j, L_2(S_i, S_j)) \notin V_H$ **then**
15 CREATE-NODE $(H(\mathcal{G}), \langle (S_j, L_2(S_i, S_j))\rangle)$;
16 $E_H \leftarrow E_H \cup (\langle S_i, L^+(S_i)\rangle, \langle (S_j, L_2(S_i, S_j))\rangle)$;
17 return $H(\mathcal{G}) = (V_H, E_H)$;

Algorithm 2: CREATE-NODE $(H(\mathcal{G}), \langle S_i, L\rangle)$

Input: HR-Index $H(\mathcal{G})$ and a new node $\langle S_i, L\rangle$
Output: HR-Index $H(\mathcal{G})$

1 $V_H \leftarrow V_H \cup \{\langle S_i, L\rangle\}$;
2 **if** $|L| = 1$ **then**
3 **for** *each node* $\langle S_i, L'\rangle$ *such that* $L \subset L'$ **do**
4 $E_H \leftarrow E_H \cup (\langle S_i, L'\rangle, \langle S_i, L\rangle)$;
5 **else**
6 **for** *each node* $\langle S_i, L'\rangle$ *whose* $|L'| = 1, L' \subset L$ **do**
7 $E_H \leftarrow E_H \cup (\langle S_i, L\rangle, \langle S_i, L'\rangle)$;
8 return $H(\mathcal{G}) = (V_H, E_H)$;

Case (1) and for the interval outgoing neighbors (line 10-16) as per discussion in Case (2). If node $\langle S_i, L\rangle$ does not exist in current $H(\mathcal{G})$, Algorithm 1 invokes CREATE-NODE $(H(\mathcal{G}), \langle S_i, L\rangle)$ to create the node $\langle S_i, L\rangle$. Algorithm 2 shows the pseudo-code of CREATE-NODE $(H(\mathcal{G}), \langle S_i, L\rangle)$. If L only includes one time point, then Algorithm 2 creates an edge $(\langle S_i, L'\rangle, \langle S_i, L\rangle)$ for every $\langle S_i, L'\rangle$ in $H(\mathcal{G})$ satisfying $L \subset L'$. On the other hand, if L includes more than one time points, i.e., $|L| > 1$, then Algorithm 2 creates an edge $(\langle S_i, L\rangle, \langle S_i, L'\rangle)$ for every $\langle S_i, L'\rangle$ in $H(\mathcal{G})$, where L' only includes one time point and $L' \subset L$. Algorithm 2 guarantees all the edges in Case (3) can be created.

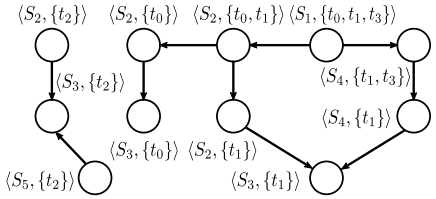


Fig. 3. HR-Index $H(\mathcal{G})$ of the evolving graph $\mathcal{G}$ in Fig. 2

Time and space complexity: The edge $(\langle S_i, \{t_x\}\rangle, \langle S_j, \{t_x\}\rangle)$ created in Case (1) essentially corresponds to an edge (S_i, S_j) in G_x of $\mathcal{G}$. By Lemma 3.2, every $\langle S_j, L_j\rangle$ ($|L_j| > 1$) created in Case (3) must be a $\langle S_j, L_2(S_i, S_j)\rangle$. It means there are at most $|L_2(S_i, S_j)|$ edges created as $(\langle S_j, L_j\rangle, \langle S_j, \{t_x\}\rangle)$ and every such edge corresponds to an edge (S_i, S_j) of G on $L_2(S_i, S_j)$. Therefore, the time and space cost to construct the edges in Case (1) and (3) are less than $O(|V_{\mathcal{G}}| + |E_{\mathcal{G}}|)$, where $|V_{\mathcal{G}}|$ and $|E_{\mathcal{G}}|$ are the vertex size and edge size of $\mathcal{G}$, i.e., $|V_{\mathcal{G}}| = \sum_{0\le x\le \|\mathcal{G}\|-1} |V_x|$ and $|E_{\mathcal{G}}| = \sum_{0\le x\le \|\mathcal{G}\|-1} |E_x|$. In Case (2), an edge is created for every interval outgoing neighbor $(S_j, L_2(S_i, S_j))$ and then there are at most $|N_2^+(S_i)|$ edges created for every S_i. Therefor, the time and space cost to construct $H(\mathcal{G})$ is $O(|V_{\mathcal{G}}| + |E_{\mathcal{G}}| + \lambda(\mathcal{G}))$, where $\lambda(\mathcal{G}) = \sum_{S_i\in\mathcal{G}} |N_2^+(S_i)|$. In Section 5, we propose two optimization techniques, redundant nodes deletion and SCC merging, which can reduce the time and space cost effectively.

Running Example: Fig. 3 illustrates the HR-Index $H(\mathcal{G})$ constructed from Table 2 for the evolving graph $\mathcal{G}$ in Fig. 2. $(S_2, \{t_0, t_1\})$ and $(S_4, \{t_1, t_3\})$ are two interval outgoing neighbors of S_1 in Table 2, then $(\langle S_1, \{t_0, t_1, t_3\}\rangle, \langle S_2, \{t_0, t_1\}\rangle)$ and $(\langle S_1, \{t_0, t_1, t_3\}\rangle, \langle S_4, \{t_1, t_3\}\rangle)$ are two edges created in Fig. 3. $(S_3, \{t_0\})$ and $(S_3, \{t_1\})$ are two instant outgoing neighbors of S_2, then $(\langle S_2, \{t_0\}\rangle, \langle S_3, \{t_0\}\rangle)$ and $(\langle S_2, \{t_1\}\rangle, \langle S_3, \{t_1\}\rangle)$ are also created in Fig. 3. We find the number of the nodes in Fig. 3 is less than the number of vertices in Fig. 2.

Lemma 3.3: *For two nodes $\langle S_i, L_i\rangle$ and $\langle S_i, L_i'\rangle$ in $H(\mathcal{G})$, if $L_i \cap L_i' \neq \emptyset$, i.e., there exists a time point $t_x \in L_i \cap L_i'$, then both $\langle S_i, L_i\rangle$ (for $|L_i| > 1$) and $\langle S_i, L_i'\rangle$ (for $|L_i'| > 1$) are interval outgoing neighbors in the ON-table. If S_i has an outgoing edge at time point t_x, then the node $\langle S_i, \{t_x\}\rangle$ must exist in $H(\mathcal{G})$.* □

Proof. If L_i or L_i' is exactly $\{t_x\}$, this lemma has been proved. Next, we consider both L_i and L_i' do not only include t_x. Because S_i has one $\langle S_i, L^+(S_i)\rangle$ at most, then $\langle S_i, L_i\rangle$ or $\langle S_i, L_i'\rangle$ is an interval outgoing neighbor in ON-table, which indicates S_i has incoming edge at time point t_x. Since $t_x \in L_i \cap L_i'$, neither $\langle S_i, L_i\rangle$ and $\langle S_i, L_i'\rangle$ can be $\langle S_i, L^+(S_i)\rangle$, otherwise it is in conflict with the fact S_i has incoming edge at time point t_x. Therefore, both $\langle S_i, L_i\rangle$ and $\langle S_i, L_i'\rangle$ are the interval outgoing neighbors. If S_i has an outgoing edge at time point t_x, the $\langle S_i, \{t_x\}\rangle$ is created for case (1), that is, $\langle S_i, \{t_x\}\rangle$ must exist in $H(\mathcal{G})$. □

Theorem 3.1: *The historical reachability of an evolving graph $\mathcal{G}$ is equivalent to its HR-Index $H(\mathcal{G})$. Specifically, for any two vertices S_i and S_j in an evolving graph $\mathcal{G}$, $S_i \rightsquigarrow S_j$ at time point t_x (or in snapshot G_x), if and only if there exist two nodes $\langle S_i, L_i\rangle$ and $\langle S_j, L_j\rangle$ in $H(\mathcal{G})$ such that $\langle S_i, L_i\rangle \rightsquigarrow \langle S_j, L_j\rangle$ and $t_x \in L_j$.* □

Proof. (1) We firt prove "⇒" direction. If $S_i \rightsquigarrow S_j$ at time point t_x, there must exist a path p from S_i to S_j in G_x. If p is an edge $S_i \to S_j$, S_j is an outgoing neighbor of S_i at time point t_x, then there must exist an edge $\langle S_i, \{t_x\}\rangle \to \langle S_j, \{t_x\}\rangle$ created from $N_1^+(S_i)$ or $\langle S_i, L^+(S_i)\rangle \to \langle S_j, L_2(S_i, S_j)\rangle$ created from $N_2^+(S_i)$ in ON-table of $\mathcal{G}$. If p is not an edge, without loss of generality, we suppose this path is $S_i \to S_1 \to \cdots \to S_l \to S_j$. In this case, every S_k ($1 \le k \le l$) has at least one incoming edge at time point t_x, according to the case (1) for ON-table constructing, there must exist edges $\langle S_k, \{t_x\}\rangle \to \langle S_{k+1}, \{t_x\}\rangle$ ($1 \le k \le l-1$) and $\langle S_l, \{t_x\}\rangle \to \langle S_j, \{t_x\}\rangle$ in $H(\mathcal{G})$. If S_i

Algorithm 3: QUERY-WHOLE-INTERVAL (v_i, v_j, I)

Input: Starting vertex v_i, ending vertex v_j, time point set I

Output: True or false for $v_i \overset{I_\vee}{\rightsquigarrow} v_j$ or $v_i \overset{I_\wedge}{\rightsquigarrow} v_j$

1 **if** *query type is disjunctive* $v_i \overset{I_\vee}{\rightsquigarrow} v_j$ **then**
2 **for** *each time point* $t_x \in I$ **do**
3 **if** QUERY-PERINSTANT (v_i, v_j, t_x) *is* ***True*** **then**
4 return **True**;
5 return **False**;
6 **if** *query type is conjunctive* $v_i \overset{I_\wedge}{\rightsquigarrow} v_j$ **then**
7 **for** *each time point* $t_x \in I$ **do**
8 **if** QUERY-PERINSTANT (v_i, v_j, t_x) *is* ***False*** **then**
9 return **False**;
10 return **True**;

has no incoming edge at t_x, then $\langle S_i, L^+(S_i)\rangle \rightarrow \langle S_1, L_2(S_i, S_1)\rangle$ and $\langle S_1, L_2(S_i, S_1)\rangle \rightarrow \langle S_1, \{t_x\}\rangle$ must exist in $H(\mathcal{G})$ according to the case (2) and (3) for ON-table constructing respectively. It means $\langle S_i, L^+(S_i)\rangle \rightarrow \langle S_1, L_2(S_i, S_1)\rangle \rightarrow \langle S_1, \{t_x\}\rangle \rightarrow \cdots \rightarrow \langle S_l, \{t_x\}\rangle \rightarrow \langle S_j, \{t_x\}\rangle$ is a path in $H(\mathcal{G})$, then $\langle S_i, L^+(S_i)\rangle \rightsquigarrow \langle S_j, \{t_x\}\rangle$. If S_i has at least one incoming edge at t_x, then the edge $\langle S_i, \{t_x\}\rangle \rightarrow \langle S_1, \{t_x\}\rangle$ must be created in $H(\mathcal{G})$ according to the case (1) for ON-table constructing. Therefore, the path $\langle S_i, \{t_x\}\rangle \rightarrow \langle S_1, \{t_x\}\rangle \rightarrow \cdots \rightarrow \langle S_l, \{t_x\}\rangle \rightarrow \langle S_j, \{t_x\}\rangle$ is in $H(\mathcal{G})$ and then we have $\langle S_i, \{t_x\}\rangle \rightsquigarrow \langle S_j, \{t_x\}\rangle$.

(2) We next prove "$\Leftarrow$" direction. If there exist two nodes $\langle S_i, L_i\rangle$ and $\langle S_j, L_j\rangle$ in $H(\mathcal{G})$ such that $\langle S_i, L_i\rangle \rightsquigarrow \langle S_j, L_j\rangle$ and $t_x \in L_j$, there must exist a path p from $\langle S_i, L_i\rangle$ to $\langle S_j, L_j\rangle$ in $H(\mathcal{G})$. Without loss of generality, suppose p is $\langle S_i, L_i\rangle \rightarrow \langle S_1, L_1\rangle \rightarrow \cdots \rightarrow \langle S_l, L_l\rangle \rightarrow \langle S_j, L_j\rangle$. By Lemma 3.1, we have $t_x \in L_j \subseteq L_l \subseteq \cdots \subseteq L_1 \subseteq L_i$. It means there exist the edges $S_i \rightarrow S_1 \rightarrow \cdots \rightarrow S_l \rightarrow S_j$ in G_x, that is, $S_i \rightsquigarrow S_j$ at time point t_x. □

Theorem 3.1 guarantees that the historical reachability query on $\mathcal{G}$ can be correctly answered by HR-Index $H(\mathcal{G})$.

4 QUERY PROCESSING

The reachability query has been well-studied on static graphs and most of the existing methods propose various indexes to improve querying efficiency significantly. HR-Index $H(\mathcal{G})$ is a single graph integrating complete and correct historical reachability information of $\mathcal{G}$. By Theorem 3.1, a historical reachability query on an evolving graph $\mathcal{G}$ can be transformed into a static reachability query on $H(\mathcal{G})$. Therefore, we construct an index on HR-Index $H(\mathcal{G})$ using the existing index methods for answering reachability queries on static graphs. In this paper, we adopt GRAIL[23], which is a time and space efficient index method for answering reachability queries on static graphs, to build index on $H(\mathcal{G})$. Next we will introduce how to answer the historical reachability query using $H(\mathcal{G})$ for an evolving graph $\mathcal{G}$.

Algorithm 3 shows the pseudo-code for answering disjunctive historical reachability query (line 1-5) and conjunctive historical reachability query (line 6-10). Given a query $v_i \overset{I_\vee}{\rightsquigarrow} v_j$ or $v_i \overset{I_\wedge}{\rightsquigarrow} v_j$, Algorithm 3 answers the query by invoking Algorithm 4 to check whether $v_i \rightsquigarrow v_j$ at every time point $t_x \in I$. For a disjunctive historical reachability query, if v_i can reach v_j at a certain time point

Algorithm 4: QUERY-PERINSTANT (v_i, v_j, t_x)

Input: Starting vertex v_i, ending vertex v_j, time point t_x
Output: True or false for $v_i \rightsquigarrow v_j$ at time point t_x

```
1  Let S_i and S_j be the SCCs including v_i and v_j respectively at time point t_x;
2  if S_i or S_j does not exist then
3      return False;
4  else if S_i = S_j then
5      return True;
6  else
7      if ⟨S_i, L+(S_i)⟩ ∈ V_H and t_x ∈ L+(S_i) then
8          for every ⟨S_j, L_j⟩ ∈ N+(⟨S_i, L+(S_i)⟩) do
9              if t_x ∈ L_j then
10                 return True;
11         return STATIC-QUERY (⟨S_i, L+(S_i)⟩, ⟨S_j, {t_x}⟩);
12     else
13         return STATIC-QUERY (⟨S_i, t_x⟩, ⟨S_j, {t_x}⟩);
```

$t_x \in I$, Algorithm 3 immediately returns true for the query, otherwise it returns false because it means v_i cannot reach v_j at any time point in I. For conjunctive historical reachability query, if v_i cannot reach v_j at a time point $t_x \in I$, Algorithm 3 immediately returns false for the query, otherwise v_i can reach v_j at every time point $t_x \in I$ and then Algorithm 3 returns true. Note that it can be very efficient to check $v_i \rightsquigarrow v_j$ for every t_x because the indexes and algorithms for static reachability query can be used on HR-Index $H(\mathcal{G})$. Therefore, the disjunctive and conjunctive historical reachability queries also will be efficient even though it may have to answer $v_i \rightsquigarrow v_j$ for every $t_x \in I$.

For answering the reachability query $v_i \rightsquigarrow v_j$ at time point t_x, Algorithm 4 first checks whether v_i and v_j are in the same SCC at t_x using SCC-table (line 1-5). If not, let S_i and S_j be the SCCs including v_i and v_j respectively, Algorithm 4 needs to query $S_i \rightsquigarrow S_j$ on $H(\mathcal{G})$ for two cases: (i) S_i has no incoming edge at t_x; and (ii) S_i has at least one incoming edge at t_x. In the first case, if $\langle S_i, L^+(S_i)\rangle \in V_H$ and $t_x \in L^+(S_i)$, which means S_i has at least one outgoing neighbor at time point t_x and $\langle S_i, L^+(S_i)\rangle$ is created in case (2) of HR-Index construction, then Algorithm 4 checks whether S_j is an outgoing neigbor of S_i at time point t_x (line 8-9). If so, then "true" is immediately returned as the result. Otherwise, Algorithm 4 invokes STATIC-QUERY to query $\langle S_i, L^+(S_i)\rangle \rightsquigarrow \langle S_j, \{t_x\}\rangle$ on $H(\mathcal{G})$ and returns this result, where STATIC-QUERY can be any existing method for answering reachability queries on static graphs. For the second case, Algorithm 4 only needs to query $\langle S_i, \{t_x\}\rangle \rightsquigarrow \langle S_j, \{t_x\}\rangle$ on $H(\mathcal{G})$ by invoking STATIC-QUERY. Note that in case (i), S_i may not have outgoing edge at time point t_x. In this case, t_x is not in $L^+(S_i)$ and thus Algorithm 4 will return "false" by line 13 because $\langle S_i, \{t_x\}\rangle$ does not exist or has no outgoing edge. Theorem 4.1 guarantees the correctness of Algorithm 4.

Theorem 4.1: *Algorithm 4 is correct for answering whether v_i can reach v_j in the snapshot G_x at time point t_x.* □

PROOF. We only need to prove that Algorithm 4 returns "true" if and only if S_i can reach S_j at time point t_x when v_i and v_j are in the distinct SCCs S_i and S_j. We first prove Algorithm 4

Algorithm 5: DELETE-REDUNDANT-NODE ($H(\mathcal{G})$)

Input: The orignal HR-Index $H(\mathcal{G})$
Output: The HR-Index $H(\mathcal{G})$ without redundant node

```
for each node ⟨S_i, {t_x}⟩ ∈ V_H do
    if all the nodes in N⁻(⟨S_i, {t_x}⟩) (N⁻(⟨S_i, {t_x}⟩) ≠ ∅) in H(G) are the senior-nodes of
      ⟨S_i, {t_x}⟩ then
        for each node ⟨S_j, L_j⟩ ∈ N⁻(⟨S_i, {t_x}⟩) do
            E_H ← E_H − {(⟨S_j, L_j⟩, ⟨S_i, {t_x}⟩)} ;
            for each node ⟨S_k, L_k⟩ ∈ N⁺(⟨S_i, {t_x}⟩) do
                E_H ← E_H ∪ (⟨S_j, L_j⟩, ⟨S_k, L_k⟩) ;
                E_H ← E_H − {(⟨S_i, {t_x}⟩, ⟨S_k, L_k⟩)} ;
    V_H ← V_H − {⟨S_i, {t_x}⟩} ;
return H(G) = (V_H, E_H);
```

returns "true" if S_i can reach S_j at time point t_x. If S_i has no incoming edge in G_x, $\langle S_i, L^+(S_i)\rangle$ satisfying $t_x \in L^+(S_i)$ must be in $H(\mathcal{G})$ since S_i can reach S_j at t_x. When (S_i, S_j) is an edge in G_x, $\langle S_i, L^+(S_i)\rangle$ must have an interval outgoing neighbor $\langle S_j, L_j\rangle$ and $t_x \in L_j$. When S_i reaches S_j via other vertices in G_x, $\langle S_j, \{t_x\}\rangle$ must be in $H(\mathcal{G})$ and $\langle S_i, L^+(S_i)\rangle$ can reach $\langle S_j, \{t_x\}\rangle$ in $H(\mathcal{G})$ by Theorem 3.1. If S_i has at least one incoming edge in G_x, $\langle S_i, \{t_x\}\rangle$ and $\langle S_j, \{t_x\}\rangle$ must be in $H(\mathcal{G})$. By the first direction proof of Theorem 3.1, there must exist a path from $\langle S_i, \{t_x\}\rangle$ to $\langle S_j, \{t_x\}\rangle$ in $H(\mathcal{G})$. For these cases, Algorithm 4 will return true. Next, if Algorithm 4 returns true, then there must exist a path from $\langle S_i, L_i\rangle$ to $\langle S_j, L_j\rangle$ in $H(\mathcal{G})$. By Theorem 3.1, S_i can reach S_j in the snapshot G_x. Therefore, Algorithm 4 must return the correct results. □

Time and space complexity: Algorithm 3 invokes Algorithm 4 at most $|I|$ times for every time point $t_x \in I$. Algorithm 4 answer the query by invoking the existing static reachability query algorithm once at most. We use $\alpha(H(\mathcal{G}))$ and $\beta(H(\mathcal{G}))$ denote the time and space costs of the existing static reachability algorithm on $H(\mathcal{G})$. Therefore, the time and space complexity of Algorithm 3 are $O(|I|\alpha(H(\mathcal{G})))$ and $O(\beta(H(\mathcal{G})))$.

5 OPTIMIZATION

In Section 3, we introduce what is HR-Index $H(\mathcal{G})$ and how to construct it for an evolving graph $\mathcal{G}$. In this section, we will propose two optimization techniques, redundant nodes deletion and SCC merging, to reduce the size of HR-Index $H(\mathcal{G})$. Because $H(\mathcal{G})$ essentially is a static graph, existing methods can be utilized for answering the historical reachability query on $H(\mathcal{G})$. It is obvious that the time cost for answering queries will be reduced with the decreasing of the size of $H(\mathcal{G})$.

5.1 Redundant nodes deletion

As per the discussion about HR-Index $H(\mathcal{G})$ in Section 3.4, there may exist several nodes in $H(\mathcal{G})$ with the same SCC ID. For example, in Fig. 3, $\langle S_2, \{t_0, t_1\}\rangle$ and $\langle S_2, \{t_0\}\rangle$ are two distinct nodes with the same SCC ID S_2. Some of these nodes can be deleted from HR-Index $H(\mathcal{G})$ and it does not affect the correctness of $H(\mathcal{G})$ for answering the historical reachability query. Such nodes are called ***redundant nodes*** of $H(\mathcal{G})$. Next, we will introduce how to find the redundant nodes and delete them from $H(\mathcal{G})$.

For two nodes $\langle S_i, \{t_x\}\rangle$ and $\langle S_j, L_j\rangle$ in $H(\mathcal{G})$, $\langle S_j, L_j\rangle$ is called a ***senior node*** of $\langle S_i, \{t_x\}\rangle$ if $S_i = S_j$ and $\{t_x\} \subsetneq L_j$. A node $\langle S_i, \{t_x\}\rangle$ must be a redundant node if it has at least one incoming

neighbor and all its incoming neighbors are senior nodes of $\langle S_i, \{t_x\}\rangle$. We use $N^-(\langle S_i, \{t_x\}\rangle)$ and $N^+(\langle S_i, \{t_x\}\rangle)$ to denote the set of incoming neighbors and outgoing neighbors of $\langle S_i, \{t_x\}\rangle$ in $H(\mathcal{G})$ respectively, that is,

$$N^-(\langle S_i, \{t_x\}\rangle) = \{\langle S_j, L_j\rangle | (\langle S_j, L_j\rangle, \langle S_i, \{t_x\}\rangle) \in E_H\}$$
$$N^+(\langle S_i, \{t_x\}\rangle) = \{\langle S_k, L_k\rangle | (\langle S_i, \{t_x\}\rangle, \langle S_k, L_k\rangle) \in E_H\}$$

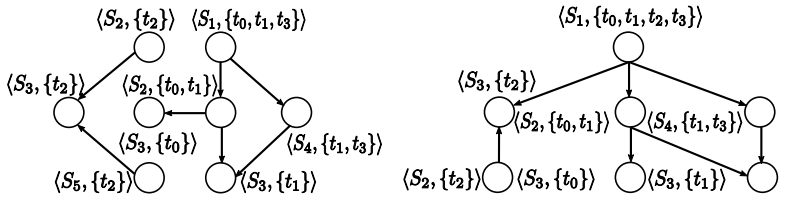
Thus $\langle S_i, \{t_x\}\rangle$ can be deleted from $H(\mathcal{G})$ after creating the edge $(\langle S_j, L_j\rangle, \langle S_k, L_k\rangle)$ for every $\langle S_j, L_j\rangle \in N^-(\langle S_i, \{t_x\}\rangle)$ and every $\langle S_k, L_k\rangle \in N^+(\langle S_i, \{t_x\}\rangle)$. The following lemma shows $H(\mathcal{G})$ is correct for the historical reachability query after deleting $\langle S_i, \{t_x\}\rangle$.

Lemma 5.1: *Given an HR-Index $H(\mathcal{G})$ and a node $\langle S_i, \{t_x\}\rangle$, if every incoming neighbor of $\langle S_i, \{t_x\}\rangle$ is a senior node of $\langle S_i, \{t_x\}\rangle$, the correctness of $H(\mathcal{G})$ for answering the historical reachability queries cannot be affected by deleting $\langle S_i, \{t_x\}\rangle$.* □

Proof. By Lemma 3.1, it is easy to know $L_k = \{t_x\}$ for every outgoing neighbor $\langle S_k, L_k\rangle$ of $\langle S_i, \{t_x\}\rangle$ in $H(\mathcal{G})$. It means S_i has an outgoing edge to S_k at time point t_x. For every senior node $\langle S_j, L_j\rangle$, connecting $\langle S_j, L_j\rangle$ to $\langle S_k, \{t_x\}\rangle$ also guarantees S_i has the outgoing edge to S_k at t_x. On the other hand, if S_i is reachable from some S' at time point t_x, because every incoming neighbor $\langle S_j, L_j\rangle \in N^-(\langle S_i, \{t_x\}\rangle)$ is a senior node of $\langle S_i, \{t_x\}\rangle$, there must exist a path from some $\langle S', L'\rangle$ to some $\langle S_j, L_j\rangle \in N^-(\langle S_i, \{t_x\}\rangle)$ in $H(\mathcal{G})$. Therefore the correctness of $H(\mathcal{G})$ cannot be affected for answering the historical reachability query after deleting $\langle S_i, \{t_x\}\rangle$ and creating $(\langle S_j, L_j\rangle, \langle S_k, L_k\rangle)$ for every $\langle S_j, L_j\rangle \in N^-(\langle S_i, \{t_x\}\rangle)$ and $\langle S_k, L_k\rangle \in N^+(\langle S_i, \{t_x\}\rangle)$. □

Algorithm 5 shows the pseudo-code of deleting redundant nodes from $H(\mathcal{G})$. For each node $\langle S_i, \{t_x\}\rangle$ in $H(\mathcal{G})$, Algorithm 5 first checks whether $\langle S_i, \{t_x\}\rangle$ is a redundant node or not (line 1-2). If it is, Algorithm 5 creates edges from every incoming neighbor to every outgoing neighbor of $\langle S_i, \{t_x\}\rangle$ and then remove $\langle S_i, \{t_x\}\rangle$ and its incoming and outing edges from $H(\mathcal{G})$ (line 3-8).

Note that Algorithm 4 may utilize $\langle S_i, \{t_x\}\rangle$ as a starting node to answer the reachability from S_i to S_j at time point t_x. However, $\langle S_i, \{t_x\}\rangle$ may be a redundant node deleted from $H(\mathcal{G})$. By Lemma 3.3 and Lemma 5.1, any node $\langle S_i, L\rangle$ satisfying $t_x \in L$ can be used as a starting node to answer the reachability query. On the other hand, a redundant node $\langle S_j, \{t_x\}\rangle$ also may be selected as a target node for answering the reachability from S_i to S_j at time point t_x. Because a redundant node has at least one incoming neighbor and all the incoming neighbors are its senior nodes, there must exist a senior node $\langle S_j, L_j\rangle$ of $\langle S_j, \{t_x\}\rangle$ satisfying $t_x \in L_j$, which preserves the reachability from S_i to S_j even thougth $\langle S_j, \{t_x\}\rangle$ is deleted from $H(\mathcal{G})$. If S_i can reach S_j, the senior node $\langle S_j, L_j\rangle$ must be an interval outgoing neighbor of $\langle S_i, L^+(S_i)\rangle$ since $|L_j| > 1$ and thus it returns true by line 7-10 in Algorithm 4 (we will explain it in the following example), otherwise it returns false by line 11 since $\langle S_j, t_x\rangle$ does not exist.



(a) Redundant nodes deletion (b) SCC merging

Fig. 4. The optimization of HR-Index

Running Example: Fig. 4(a) illustrates the HR-Index $H(\mathcal{G})$ after deleting redundant nodes in Fig. 3. In this example, $\langle S_2, t_0\rangle$, $\langle S_2, t_1\rangle$ and $\langle S_4, t_1\rangle$ are the redundant nodes in Fig. 3 and they can be

Algorithm 6: Build-Merging-Graph (T_s)

Input: The SCC-table T_s of $\mathcal{G}$
Output: A merging graph $G_M = \{V_M, E_M\}$
1 **Initialize:** $V_M \leftarrow S$, $E_M \leftarrow \emptyset$;
2 **for** *every two SCCs* S_i *and* S_j *in* V_M **do**
3 **if** $L(S_i) \cap L(S_j) = \emptyset$ **then**
4 $E_M \leftarrow E_M \cup \{(S_i, S_j)\}$;
5 return $G_M = (V_M, E_M)$;

removed from $H(\mathcal{G})$ safely. Consider a query whether S_1 can reach S_4 at time point t_1, even though $\langle S_4, t_1\rangle$ is redundant and deleted from $H(\mathcal{G})$, it has a senior node $\langle S_4, \{t_1, t_3\}\rangle$. Therefore, Algorithm 4 can return "true" by line 7-10 to check $\langle S_4, \{t_1, t_3\}\rangle$ is an outgoing neigbor of $\langle S_1, \{t_0, t_1, t_3\}\rangle$, i.e., $\langle S_1, L^+(S_1)\rangle$.

5.2 SCC Merging

In HR-Index $H(\mathcal{G})$, every SCC in the snapshots is regarded as a vertex and thus every snapshot in $\mathcal{G}$ is converted into a directed acyclic graph. Some SCCs may never appear in the same snapshot in the evolving graph $\mathcal{G}$. These SCCs can be merged into a SCC by assigning the same SCC ID. Note that the meaning of "SCC merging" is not to merge all the vertice in different SCCs into a new SCC but to assign the same SCC ID to these SCCs. By SCC merging, the storage cost of SCC-table, ON-table and the vertex size of $H(\mathcal{G})$ can be reduced, and thus the efficiency of historical reachability querying is also improved.

For two distinct SCCs S_i and S_j in an evolving graph $\mathcal{G}$, if their lifespans do not intersect, i.e., $L(S_i) \cap L(S_j) = \emptyset$, they can be assigned the same SCC ID such as S_k. The intuitive meaning behind SCC merging is that the same ID can be used for S_i in L_i and for S_j in L_j respectively and it will not cause conflict because $L(S_i) \cap L(S_j) = \emptyset$.

Theorem 5.1: *For any two SCCs S_i and S_j in $\mathcal{G}$ with lifespans $L(S_i)$ and $L(S_j)$, respectively. If $L(S_i) \cap L(S_j) = \emptyset$, the correctness of the historical reachability query cannot be affected when they are assigned the same SCC ID S_k.* □

Proof. Assume that vertex sets of S_i and S_j are V_i and V_j respectively. $L(S_i) \cap L(S_j) = \emptyset$ indicates S_i and S_j never appear in the same snapshot in $\mathcal{G}$. Although S_i and S_j are assigned the same SCC ID S_k, S_k still consists of V_i in L_i and consists of V_j in L_j. Obviously, the correctness of the historical reachability query cannot be affected if the vertex set corresponding to the SCCs with lifespans are maintained in SCC-table. □

Given a set $\mathcal{S}$ of SCCs, if $L(S_i) \cap L(S_j) = \emptyset$ for any pair of SCCs S_i and S_j in $\mathcal{S}$, then all the SCCs in $\mathcal{S}$ can be assigned the same ID S_k. $\mathcal{S}$ is called a *safe set* for SCC merging. Let S denote the set of all the SCCs in $\mathcal{G}$. There are several ways to partition S into different safe sets $\mathcal{S}$. A *safe partition* $P(S)$ of S is a collection of $\mathcal{S}$ such that (1) every $\mathcal{S} \in P(S)$ is a safe set for SCC merging; (2) $\mathcal{S} \cap \mathcal{S}' = \emptyset$ for $\forall \mathcal{S}, \mathcal{S}' \in P(S)$ and (3) $S = \cup_{\mathcal{S} \in P(S)} \mathcal{S}$. The smaller number of SCCs results in the less space cost of SCC-table, ON-table and vertex size of $H(\mathcal{G})$. Therefore, the optimal SCC merging is to find a $P(S)$ minimizing $|P(S)|$.

An optimal safe partition problem can be easily converted into a minimum graph coloring problem by constructing a "merging graph". A merging graph is an undirected graph , denoted as $G_M = (V_M, E_M)$, where $V_M = S$ is the vertex set representing all the SCCs in $\mathcal{G}$ and an edge $(S_i, S_j) \in E_M$ if and only if $L(S_i) \cap L(S_j) = \emptyset$. The pseudo-code of merging graph construction

Table 3. SCC-Table after SCC merging

Vertex set	SCC with lifespan
$\{v_1, v_2, v_3\}$	$(S_1, \{t_0, t_1, t_2, t_3\})$
$\{v_4\}$	$(S_2, \{t_0, t_1, t_2\})$
$\{v_5, v_6, v_7, v_8\}$	$(S_3, \{t_0, t_1, t_2\})$
$\{v_9, v_{10}\}$	$(S_4, \{t_1, t_3\})$
$\{v_{11}\}$	$(S_1, \{t_2\})$

Table 4. The new ON-Table after SCC merging

SCC	Instant outgoing neighbor	Interval outgoing neighbor
S_1	NULL	$(S_2, \{t_1, t_2\})$, $(S_3, \{t_3\})$, $(S_4, \{t_2, t_4\})$
S_2	$(S_3, \{t_1\})$, $(S_3, \{t_2\})$	$(S_3, \{t_3\})$
S_3	NULL	NULL
S_4	$(S_3, \{t_2\})$	NULL

is shown in Algorithm 6. A minimum graph coloring is an assignment of labels, called colors, to the vertices of a graph such that no two adjacent vertices share the same color and the number of colors is the minimum. The minimum graph coloring problem is a well-known NP-complete problem and has been well studied. To solve the minimum graph coloring problem, we use Welch Powell Algorithm which is a greedy algorithm to assign color to vertices iteratively. In each iteration, the algorithm assigns a selected color to as many uncolored non-adjacent vertices as possible, and vertices are colored in descending order of their degrees. Different colors are used in different iterations until all the vertices become colored. We assign the same SCC ID to the SCCs in merging graph G_M with the same color and then construct SCC-table, ON-table and HR-Index $H(\mathcal{G})$.

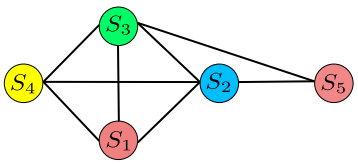


Fig. 5. The merging graph G_M of SCC table in Table 1

Running Example: Fig. 5 illustrates the coloring merging graph G_M constructed from Table 1. In this example, the lifespans of S_1 and S_5 are $L(S_1) = \{t_0, t_1, t_3\}$ and $L(S_5) = \{t_2\}$ respectively, $L(S_1) \cap L(S_5) = \emptyset$, then they are not adjacent in G_M and can be assigned with the same SCC ID S_1. The SCC-table, ON-table and HR-Index $H(\mathcal{G})$ after SCC merging is shown in Table 3, Table 4 and Fig. 4(b). From Table 3, we know v_1, v_2, v_3 are in the same SCC S_1 at t_0, t_1, t_2, t_3 and v_{11} is in S_1 only at t_2. The number of vertices in Fig. 4(b) decreases to 7, which is nearly the half of vertex size of the original evolving graph $\mathcal{G}$.

6 PERFORMANCE EVALUATION

In this section, we compare our method with TimeReach method proposed in [15] on eight real datasets. To the best of our knowledge, TimeReach is the state of the art method for answering the historical reachability query on evolving graphs. All the experiments are conducted on a cloud server with 2.5GHz Intel Xeon CPU and 128G main memory, running on Ubuntu 18.04.1 LTS.

Table 5. Dataset Information

Dataset	Number of vertices		Number of edges		Number of SCCs		Average number		$\|\mathcal{G}\|$
	First	Last	First	Last	First	Last	Vertex	Edge	
INL	9,978	65,535	29,504	104,824	1,299	1,931	17,385	63,262	43
AMA	261,056	403,394	915,010	3,387,388	41,924	1,588	297,652	2,228,770	12
WT	787,998	1,140,149	3,309,592	7,833,140	137	76,920	977,819	1,989,930	16
FBL	52,804	63,731	640,121	1,545,686	15,543	2,278	56,860	1,033,670	36
FBW	1,429	61,096	2,365	1,139,081	1,071	10,552	26,257	288,423	125
FBD	117	61,096	128	1,139,081	16	10,552	21,936	239,361	871
SOF	1,468,742	2,601,977	36,233,450	63,497,050	78,518	254,358	1,969,311	43,949,712	9
WY	7,290	1,892,691	9,249	39,953,145	6,140	51,104	859,613	14,186,300	7

6.1 Datasets and Experiment Setup

We utilize the following 8 real social network datasets obtained from three public links[1][2][3] to test our method.

Internet Links (INL) : This dataset includes 43 snapshots and every snapshot represents the connection between web pages in the corresponding time stamp. The time interval between two adjacent snapshots is one month.

Amazon (AMA) : This dataset includes 12 snapshots, which describes the evolution of the co-purchased relationship in Amazon mall in 2003.

Wiki Talk (WT) : It includes 16 snapshots and describes the evolution of the edited talk pages among users in Wikipedia.

FaceBook Links (FBL) : It has 36 snapshots, describing the evolution of user's friend relationship in one day on Facebook's New Orleans network. Every snapshot represents the situation of friend relationship between users at the corresponding time stamp.

Facebook Weeks (FBW) : This dataset describes the evolution of user's friend relationship in Facebook's New Orleans network. Unlike FaceBook Links, this dataset contains 125 snapshots, and the time interval between two adjacent snapshots is one week.

Facebook Days (FBD) : This dataset describes the evolution of user's friend relationship in Facebook's New Orleans network. Unlike FaceBook Links, this dataset contains 871 snapshots, and the time interval between two adjacent snapshots is one day.

Stack Overflow (SOF) : Stack Overflow is a temporal network of interactions on the stack exchange web site. This dataset includes 9 snapshots and describes the evolution of user interaction in the Stack Overflow.

Wikipedia Links Years (WY) : This dataset indicates the evolution of established connection among Wikipedia users in years. The dataset contains 7 snapshots, and the time interval between two adjacent snapshots is one year.

In the metadata of these social networks, every vertex and edge is with a time stamp, thus the snapshots of evolving graphs can be generated according to time granularity. For example, Facebook Weeks (FBW) and Facebook Days (FBD) consist of 125 weekly and 871 daily snapshots of the New Orleans Facebook friendship graph respectively. In every snapshot, the vertices and edges are with time stamps in the same week or the same day. All these datasets are treated as directed evolving graphs, i.e., if there is a record (v_i, v_j) in a snapshot file, there is a directed edge from vertice v_i to v_j in this snapshot. The statistics of these datasets are shown in Table 5. For every dataset, we show the number of vertices and edges, and the number of the strongly connected components (SCCs) in the first and last snapshots respectively. Different from TimeReach[15], the

[1]https://www.comp.hkbu.edu.hk/~db/book/community_search.html
[2]https://socialnetworks.mpi-sws.org/datasets.html
[3]http://socialnetworks.mpi-sws.mpg.de/data-wosn2009.html

Table 6. Statistics of SCC and HR-Index

Dataset	Number of SCCs		Number of vertices			Number of edges		
	HR	OP-HR	$\|V_{\mathcal{G}}\|$	HR	OP-HR	$\|E_{\mathcal{G}}\|$	HR	OP-HR
INL	19,210	1,931	747,555	48,178	35,534	2,720,266	65,780	65,331
AMA	73,737	41,924	3,571,824	96,252	92,242	26,745,240	141,582	138,175
WT	126,575	74,042	15,645,104	69,813	67,736	31,838,880	131,186	129,796
FBL	39,016	15,708	2,046,960	314,322	298,650	37,212,120	426,201	420,868
FBW	61,676	14,305	3,282,125	717,849	676,047	36,052,875	1,027,713	1,014,279
FBD	63,059	15,079	19,106,256	732,016	681,773	208,483,431	1,097,691	1,016,752
SOF	261,518	88,569	17,723,799	576,303	422,782	395,547,708	1,133,446	1,017,387
WY	310,251	163,216	6,017,291	1,706,992	1,451,873	99,304,100	2,526,458	2,385,126

SCCs only consisting of one vertex are also counted into the number of SCCs. The average number of vertices and edges, and the number of the snapshots (or length) of every dataset are also shown in Table 5.

Because HR-Index essentially is a static graph integrating complete and correct historical reachability information, then the existing index methods for static reachability queries can be used for HR-Index. The index on HR-Index can be regarded as a secondary-level index. In the experiments, we adopt GRAIL proposed in [23], which is with small index size and high query efficiency. Because HR-Index is equivalent to its corresponding evolving graph for answering historical reachability query, then we only need to maintain SCC-table and HR-Index instead of the original evolving graph $\mathcal{G}$.

We randomly generate 1000 pairs of vertices and query the historical reachability between every pair of vertices. The reported time is the average querying time on each dataset. We use HR, Op-HR and TRC to denote HR-Index, HR-Index with optimization and Time Reach Condensed method in [15] respectively.

Note that TRC is the optimized version of TimeReach method, which consumes less memory. The authors also propose another variant of TimeReach, named TimeReach Condensed 2-Hop (TRCH) in [15], which is a condensed variant using 2-hop index. However, the java code obtained from authors of this paper has a potential bug and out-of-memory errors on all our datasets for TRCH. Moreover, the description of TRCH in the paper is too simple to reproduce it for us. Thus we were unable to compare with TRCH method in our experiments. The authors encountered the same problem of TRCH code as they claimed in their paper [16] published in ICDE 2019.

At the beginning of query processing, for disjunctive reachability query, we firstly scan the whole query interval I to check whether there exists a time point $t_x \in I$ such that v_i and v_j are in the same SCC at t_x. If such t_x exists, then the reachability query can be answered by "true" directly. For conjunctive reachability query, we firstly scan query interval I to check whether both v_i and v_j exist in all the snapshots on I. If not, then the reachability query can be answered by "false" directly.

We are interested in the following aspects to evaluate the performance of HR-Index: (1) some statistics of SCC-table and HR-Index; (2) index constructing time; (3) index size; and (4) the querying time for disjunctive and conjunctive historical reachability query. We first present the experimental results of HR-Index without optimization techniques in Section 6.2.1. To validate the effectiveness of redundant nodes deletion and SCC merging, the experimental results of HR-Index with optimization techniques are shown in Section 6.2.2.

6.2 Experimental Results

6.2.1 HR-Index without optimization techniques

Exp-1. Statistics of SCC-table and HR-Index: Table 6 presents the number of SCCs in SCC-table, the number of vertices and edges in evolving graph $\mathcal{G}$ and that in its HR-Index respectively for every dataset. From Table 6, we find both $|V_H|$ and $|E_H|$ in HR-Index are far less than $|V_{\mathcal{G}}|$ and $|E_{\mathcal{G}}|$ in $\mathcal{G}$, where

$$|V_{\mathcal{G}}| = \sum_{0 \le x \le \|\mathcal{G}\|-1} |V_x| \text{ and } |E_{\mathcal{G}}| = \sum_{0 \le x \le \|\mathcal{G}\|-1} |E_x|$$

The reason for measuring the total number of vertices $|V_{\mathcal{G}}|$ and edges $|E_{\mathcal{G}}|$ for all the snapshots is that it is necessary to maintain all the snapshots for DFS method or construct index for every snapshot for answering historical reachability queries. However, our method only needs to maintain HR-Graph and SCC table. Therefore we compare the numbers of all the vertices and edges in all the snapshots with HR-Index. Take dataset WT as an example, the number of SCCs is 126,575, and the number of vertices and edges in HR-Index are only 69,813 and 131,186 respectively. Because the evolving graph is unnecessary for answering historical reachability, then we only need to maintain the SCC-table and HR-Index instead of maintaining the evolving graph which has 15,645,104 vertices and 31,838,880 edges.

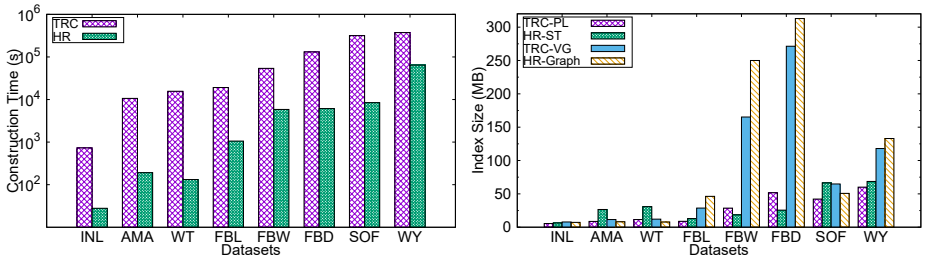
Exp-2. Index Construction Time: We investigate the index construction time by comparing HR-Index and TRC in Fig. 6(a). Note that our method adopts GRAIL to build a secondary-level index on HR-Index and then construction time for HR-Index is the sum of the time to construct SCC-table, HR-Index and GRAIL. As shown in the Fig. 6(a), the construction time for HR-Index are always less than TRC by at least an order of magnitude on seven datasets, even though we consider construction time of GRAIL. The main reason is that TRC needs to calculate the maximum weight bipartite matching for every two adjacent snapshots in evolving graphs, which results in more expensive construction time.

Exp-3. Index Size: We compare the index size of HR-Index and TRC in Fig. 6(b). In Fig. 6(b), TRC-PL and TRC-VG are the storage cost of posting list and version graph of TRC, where posting list is similar to SCC-table to maintain SCCs in evolving graphs. HR-ST and HR-Graph are the storage cost of SCC-table and HR-Index respectively. We do not present GRAIL in this figure because the maximum index size of GRAIL on our HR-Index is only 0.8Mb for these seven datasets and then we omit it.

From Fig. 6(b), we find the storage cost of SCC-table (HR-ST) is always slightly more than posting list (TRC-PL). It is because posting list is compressed by computing the maximum weight bipartite matching for every two adjacent snapshots in evolving graphs, which gives rise to high time consumption on posting list construction. On the other hand, we find the storage cost of HR-Index (HR-Graph) is less than version graph (TRC-VG) on datasets INL, AMA, WT and SOF but more than TRC-VG on datasets FBL, FBW, FBD and WY. It is because some nodes in HR-Index may be associated with the same SCC, e.g., $\langle S_i, L_i \rangle$ and $\langle S_i, L_i' \rangle$. We find the largest storage cost of HR-Index is 312.91Mb on FBD and TRC-VG is 271.36Mb for the same dataset.

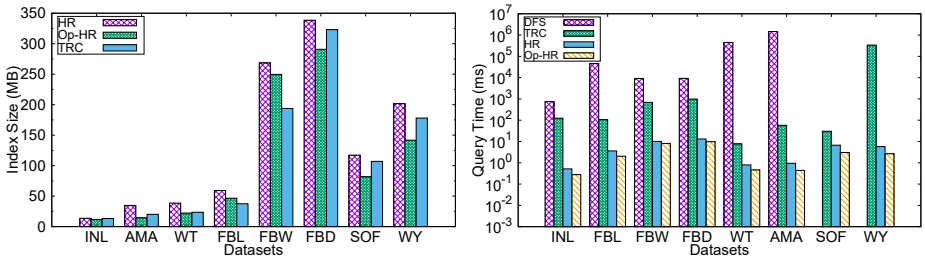
We find that the index sizes of HR-Graph on FBW and FBD are larger than that on SOF and WY, even though both $|V_H|$ and $|E_H|$ of HR-Graph on FBW and FBD are smaller than that on SOF and WY as shown in Table 6. The reason is FBW and FBD have more snapshots than SOF and WY, thus every node in HR-Graph of FBW and FBD needs a larger bitset to store more time points information.

For a clearer comparison, we present the total index size of our HR-Index (HR) and TimeReach (TRC) in Fig. 7(a). In this figure, HR is the total storage cost of SCC-table (HR-ST) and HR-Index (HR-Graph), and TRC is the total storage cost of posting list (TRC-PL) and version graph (TRC-VG). As shown in Fig. 7(a), the total index sizes of our HR-Index without optimization techniques on different datasets are always slightly larger than TRC but our method has at least an order of



(a) Index Construction time (b) Index size

Fig. 6. HR-Index without optimization



(a) Total Index Size (b) Query time for selected verter pairs

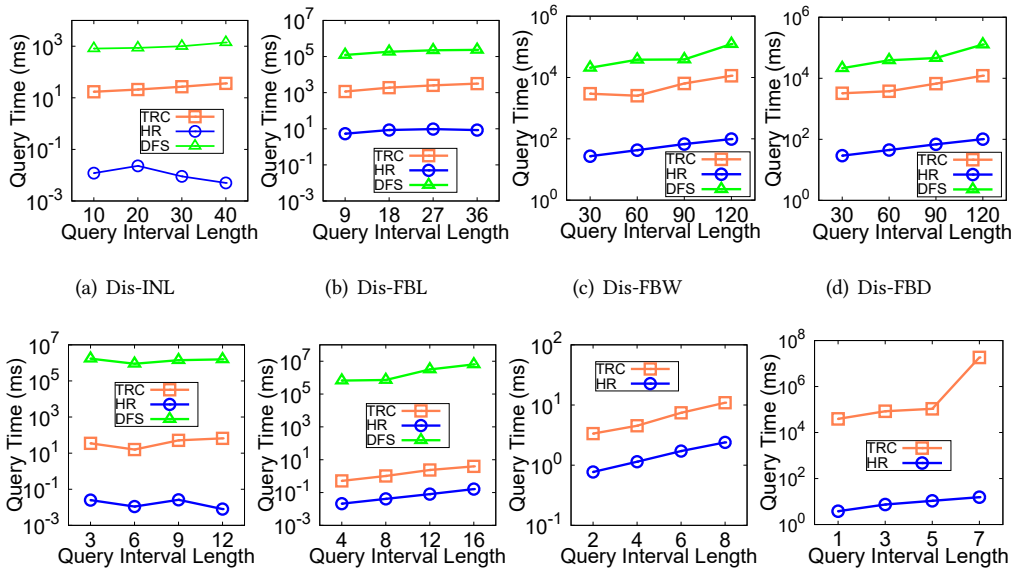
Fig. 7. Performance of HR-Index

magnitude improvement compared to TRC for both querying time and index construction time on these datasets. For example, the largest difference of total storage cost of HR and TRC is 74.8Mb (268.62MB-193.82MB) on FBW dataset, but the querying time and index construction time have been improved by at least one order of magnitude. It indicates our method is more suitable for scenarios where users prefer to pay a small space cost for larger time efficiency improvement. Moreover, by redundant nodes deleting and SCC merging, the total index sizes of HR-Index become smaller than TRC on several datasets, we will introduce that in Section 6.2.2.

It is worth noting that TRC needs to execute BFS or DFS traversal with checking time interval intersection on version graph to answer historical reachability query, but the existing static reachability techniques can be used on our HR-Index. The original static graph is unnecessary to maintain for some index methods answering static reachability queries, such as 2-Hop-based index. When these methods are used on HR-Index, the index size can be further reduced because HR-Index does not need to be stored. In addition, we find that the storage cost for some datasets, e.g., Amazon dataset, is less than that of Facebook Weeks dataset even though Amazon dataset has more vertices or edges. It is because the number of snapshots also affects the storage cost. Facebook Weeks has 125 snapshots but Amazon only has 12 snapshots.

Exp-4. Querying Time: We investigate the querying time for disjunctive reachability query in Fig. 8 and conjunctive reachability query in Fig. 9 by varying the query interval I. We compare our HR-Index method with TRC and traditional DFS traversal. For each dataset, we randomly generate 1000 pairs of vertices and query the historical reachability between every pair of vertices. The reported time is the average time on each dataset. In this experiment, if a method cannot answer the query within 48 hours, the querying time will not be shown in experiment results.

From Fig. 8 and Fig. 9, we find the querying time of disjunctive reachability increases marginally with the increasing of query interval length. For a disjunctive reachability query, it can return "false" when v_i cannot reach v_j on every snapshot in I. Therefore, the query execution times will increase with the expansion of query interval I. However, for INL and AMA datasets in Fig. 8(a)



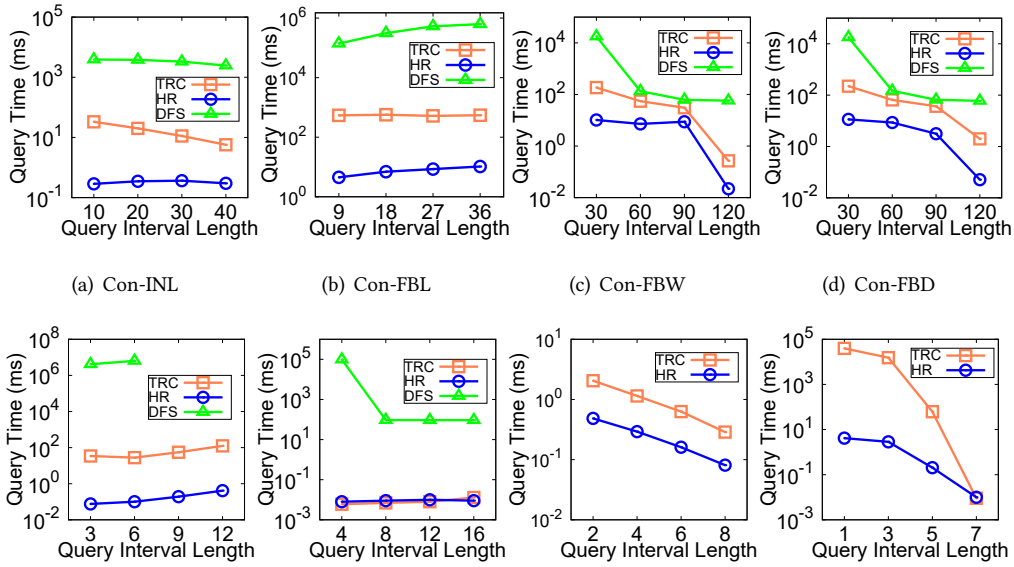
(a) Dis-INL (b) Dis-FBL (c) Dis-FBW (d) Dis-FBD

(e) Dis-AMA (f) Dis-WT (g) Dis-SOF (h) Dis-WY

Fig. 8. Query time of disjunctive historical reachability query on different datasets

and Fig. 8(e), we find the querying time of disjunctive reachability decreases with increasing of query interval length. The reason is, in these two experiments, the querying vertex pairs are not reachable in the snapshots in the smaller query interval but they are in the same SCC in some new snapshots when the query interval is expanded. At the beginning of the query processing, we firstly scan the query interval I to check whether there exists a time point $t_x \in I$ such that v_i and v_j are in the same SCC. Our algorithm cannot return "true" at the beginning of query processing for the queries on the smaller query interval, but it may return "true" directly for the expanded query interval. For example, v_i cannot reach v_j in all the snapshots on time interval $[10, 20]$ but they are in the same SCC at time point 25. When the query interval is $[10, 20]$, it needs to process reachability query for every snapshot in $[10, 20]$. However, when the query interval is $[10, 30]$, because we scan the query interval at the beginning and find v_i and v_j are in the same SCC at time point 25, thus the query can be answered by "true" directly. Therefore, the querying time on $[10, 30]$ is less than that on $[10, 20]$. On the other hand, we find the querying time of conjunctive reachability decreases with the length increasing of query interval. For conjunctive reachability, it can return "false" if v_i cannot reach v_j on one snapshot in I. At the beginning of query processing, we firstly check if both v_i and v_j exist in all the snapshots in I. If not, this query can be answered by "false" directly. Obviously, the longer query interval results in the less possibility that v_i and v_j exist in all the snapshots and then the querying time will be decreased.

As shown in Fig. 8 and Fig. 9, for both disjunctive and conjunctive reachability query, HR-Index can make at least an order of magnitude improvement compared to TRC on most datasets. However, in Fig. 9(f), we notice that the querying time of HR-Index and TRC is very close for conjunctive reachability query on the Wiki Talk dataset. It is because in this experiment, v_i and v_j always do not exist in all the snapshots or v_i and v_j are always in the same SCC. For these two cases, the query result can be returned directly without query processing on HR-Index or version graph. It is a special case that the querying time equals to the time to retrieve the posting list of TRC or SCC-table of our method.



(a) Con-INL (b) Con-FBL (c) Con-FBW (d) Con-FBD

(e) Con-AMA (f) Con-WT (g) Con-SOF (h) Con-WY

Fig. 9. Query time of conjunctive historical reachability query on different datasets

To avoid the effect of above special case, we conduct an experiment to evaluate querying time in Fig. 7(b). In this experiment, we select 1000 pairs of vertices such that v_i and v_j are in all the snapshots and they are not in the same SCC. We execute disjunctive and conjunctive reachability query for these vertex pairs and report the average querying time for every dataset. As shown in Fig. 7(b), we find HR-Index is always at least three orders of magnitude faster than TRC and it confirms that HR-Index has significant improvement on query efficiency.

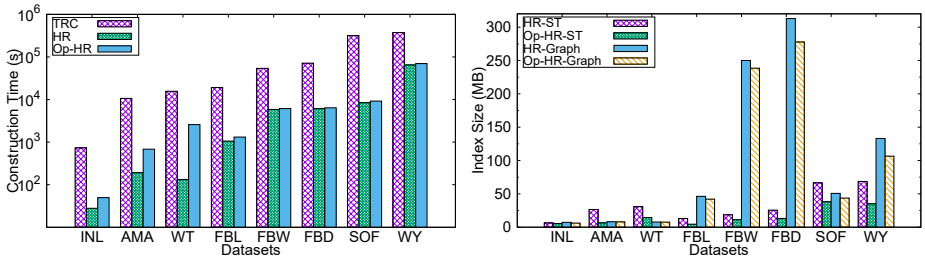
6.2.2 HR-Index with optimization techniques

Exp-5. Statistics of SCC-table and HR-Index: The number of SCCs after SCC merging and the number of vertices and edges after redundant nodes deleting are also shown in Table 6. For example, the number of SCCs decreases from 19,210 to 1,931 after SCC merging and the number of vertices in HR-Index decreases from 48,178 to 35,534 for INL dataset. The experimental results validate the effectiveness of redundant nodes deletion and SCC merging.

Exp-6. Index construction time: As shown in Fig. 10(a), even though redundant nodes deletion and SCC merging incur extra time cost, the index construction time of Op-HR does not increase significantly on all the datasets. We find Op-HR is still less than TRC by an order of magnitude. It is because our method only needs to compute minimum graph coloring for merging graph G_M once but TRC needs to compute the maximum weight bipartite matching for every two adjacent snapshots in evolving graphs.

Exp-7. Index Size: Fig. 10(b) illustrates the index size of Op-HR, where HR-ST and HR-Graph have been introduced in Exp-3. Op-HR-ST and Op-HR-Graph are the storage cost of SCC-table and HR-Index with optimization techniques, respectively. We find the storage cost of SCC-table is effectively reduced by SCC merging and the storage cost of Op-HR-Graph also decreases after SCC merging and redundant nodes deleting.

The main objective of redundant nodes deletion and SCC merging is to reduce the storage cost of SCC table and HR-Index. For SCC merging, it can effectively reduce the storage cost of SCC table. As shown in Fig. 10(b), the sizes of SCC-table without optimization are 26.6Mb, 30.8Mb,



(a) Index construction time (b) Index size

Fig. 10. HR-Index with optimization

12.9Mb, 18.6Mb, 25.6Mb, 66.5Mb and 68.6Mb on AMA, WT, FBL, FBW, FBD, SOF and WY datasets respectively, but these sizes are reduced to 6.4Mb, 14.6Mb, 4.5Mb, 11.2Mb, 13.1Mb, 38.1Mb and 35.2Mb for Op-HR on the same datasets. These experimental results show the storage cost of SCC-table can be reduced by nearly 50% with our SCC merging technique. On the other hand, we find the storage costs of HR-Index do not decrease very much after redundant node removing on some datasets. The reason is that the redundant nodes are a small proportion of HR-Graph for these datasets. However, the index construction time does not increase significantly but our redundant node deletion technique still can reduce the size of HR-Index from 133Mb to 106Mb (decreased by 20%) for WY dataset and reduce index sizes by about 10 percent for other datasets.

We also present the total storage cost of Op-HR in Fig. 7(a). By two optimization techniques, we find that the total index sizes of our method become smaller than TRC on several datasets. For example, the total storage costs of Op-HR are smaller than TRC on INL, AMA, WT, WY and SOF datasets. These experimental results validate that our method is more suitable for larger evolving graphs compared to the existing method.

Exp-8. Querying Time: Similar to Exp-4, we evaluate Op-HR by reporting average querying time of 1,000 selected pairs of vertices in Fig. 7(b). We find the querying time of Op-HR is less than HR because the number of vertices and edges of Op-HR is reduced from HR as shown in Table 6. In this experiment, we do not present the relative performance of our two optimization techniques with original HR-Index in Fig. 8 and Fig.9, because these two optimization techniques cannot affect query efficiency when two vertices are in the same SCC. Therefore, we only investigate the query efficiency with optimization for 1,000 selected pairs of vertices in Fig. 7(b) and the experimental results show the query efficiency also can be improved marginally.

7 RELATED WORK

Reachability query on static graphs has been well studied in the past decades [1, 3–8, 17–20, 23, 26, 27]. In recent years, some works investigate reachability query on dynamic graphs [14, 24, 27] or temporal graphs [2, 21, 22, 25]. For dynamic graphs, these works are to answer reachability query for the present status of evolving graphs instead of historical reachability query. They study how to update index incrementally for every deletion/insertion of vertex or edge. These methods do not consider historical reachability information and thus they incur expensive index updating cost for historical reachabilities on a set of snapshots. For temporal graphs, it can be regarded as a single graph in which every edge has a time label to indicate when it is built/exists. Most works propose various indexes based on 2-Hop labeling to answer temporal reachability query, which is to identify whether there is a time respecting path between two vertices. It is different from historical reachability query because a time respecting path requires time labels on edges to follow a non-decreasing order. Two vertices are temporal reachable but may not be reachable in a snapshot

of evolving graphs and thus these methods cannot be used for historical reachability queries. The work [21] defines a novel reachability model, called span-reachability, designed to relax the time order dependency and identify the relationship between entities in a given time period. For span reachability model, every edge is with one time point to indicate when it is connected and then the project graph is defined as a static graph containing all edges at times falling in the given interval. A span reachability query is to answer whether a vertex can reach another vertex on the project graph. A two-hop cover based index method is proposed in this work to answer span reachability queries. The index constructs a "bag" for every vertex which stores the reachability from or to other vertices under the limit of a given span length and uses pruning strategy and merge timestamps to reduce storage cost. When querying the reachability between two vertices, it can look up the corresponding bags to get the answer and uses sliding window approach to speed up. This method is also not suitable for historical reachability query (especially for conjunctive reachability queries) on evolving graphs because the lifespan of every vertex and edge in evolving graph is a set of discrete time intervals consisting a mass of time points. When this two-hop cover based index is used for historical reachability query on an evolving graph, it becomes the union of two-hop cover indexes of all snapshots, i.e., it is essentially equivalent to build a two-hop index for every snapshot in evolving graphs. Moreover, most of works on historical queries focus mainly on efficiently storing and retrieving the graph snapshots required for processing some kinds of queries such as shortest-path distance, closeness centrality, and graph diameter [10–13]. However, these methods are not to answer the historical reachability query. To the best of our knowledge, TimeReach [15] is the state-of-the-art method for the historical reachability query on evolving graphs. TimeReach constructs a version graph, in which every vertex and edge is with a set of time intervals indicating when this vertex or edge exists in an evolving graph, and the historical reachability query can be answered on it. The main disadvantage is TimeReach still needs to execute BFS or DFS traversal with checking time interval intersection on version graph to guarantee the correctness, because two vertices may not be reachable for some snapshots even though there is a path between them in version graph. Therefore the existing index for static reachability query cannot be used for version graph to improve query efficiency.

8 CONCLUSION

In this paper, we propose a novel index HR-Index for answering the historical reachability queries on evolving graphs. An HR-Index essentially is a static graph integrating complete and correct historical reachability information of the evolving graph. By HR-Index, a historical reachability query is equivalent to a static reachability query on HR-Index. Therefore, the existing method for static reachability query can be used on HR-Index straightforwardly and then query efficiency will be improved significantly. We also propose two optimization techniques to reduce the size of HR-Index effectively. We confirm the effectiveness and efficiency of our method through conducting extensive experiments on real-life datasets. The experimental results show our method has at least an order of magnitude improvement in both time and space efficiency compared to the state-of-the-art method.

9 ACKNOWLEDGEMENTS

This work is supported by the National Key Research and Development Program of China No. 2019YFB2101903, National Natural Science Foundation of China No. U22A2025 and No. 61972275, the State Key Laboratory of Communication Content Cognition Funded Project No. A32003.

REFERENCES

[1] Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-based Algorithms for Pattern Matching on DAGs. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 493–504. http://www.vldb.org/archives/website/2005/program/paper/wed/p493-chen.pdf

[2] Xiaoshuang Chen, Kai Wang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Ying Zhang. 2021. Efficiently Answering Reachability and Path Queries on Temporal Bipartite Graphs. *Proc. VLDB Endow.* 14, 10 (2021), 1845–1858. http://www.vldb.org/pvldb/vol14/p1845-chen.pdf

[3] Yangjun Chen and Yibin Chen. 2008. An Efficient Algorithm for Answering Graph Reachability Queries. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 893–902. https://doi.org/10.1109/ICDE.2008.4497498

[4] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 193–204. https://doi.org/10.1145/2463676.2465286

[5] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, David Eppstein (Ed.). ACM/SIAM, 937–946. http://dl.acm.org/citation.cfm?id=545381.545503

[6] Rodrigo Ferreira da Silva, Sebastián Urrutia, and Lars Magnus Hvattum. 2021. Extended high dimensional indexing approach for reachability queries on very large graphs. *Expert Syst. Appl.* 181 (2021), 114962. https://doi.org/10.1016/j.eswa.2021.114962

[7] H. V. Jagadish. 1990. A Compression Technique to Materialize Transitive Closure. *ACM Trans. Database Syst.* 15, 4 (1990), 558–598. https://doi.org/10.1145/99935.99944

[8] Ruoming Jin and Guan Wang. 2013. Simple, Fast, and Scalable Reachability Oracle. *PVLDB* 6, 14 (2013), 1978–1989. https://doi.org/10.14778/2556549.2556578

[9] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. 2008. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 595–608. https://doi.org/10.1145/1376616.1376677

[10] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 997–1008. https://doi.org/10.1109/ICDE.2013.6544892

[11] Georgia Koloniari, Dimitris Souravlias, and Evaggelia Pitoura. 2013. On Graph Deltas for Historical Queries. *CoRR* abs/1302.5549 (2013). arXiv:1302.5549 http://arxiv.org/abs/1302.5549

[12] Alan G. Labouseur, Paul W. Olsen, and Jeong-Hyon Hwang. 2013. Scalable and Robust Management of Dynamic Graph Data. In *Proceedings of the First International Workshop on Big Dynamic Distributed Data, Riva del Garda, Italy, August 30, 2013 (CEUR Workshop Proceedings, Vol. 1018)*, Graham Cormode, Ke Yi, Antonios Deligiannakis, and Minos N. Garofalakis (Eds.). CEUR-WS.org, 43–48. http://ceur-ws.org/Vol-1018/paper7.pdf

[13] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On Querying Historical Evolving Graph Sequences. *PVLDB* 4, 11 (2011), 726–737. http://www.vldb.org/pvldb/vol4/p726-ren.pdf

[14] Liam Roditty and Uri Zwick. 2016. A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time. *SIAM J. Comput.* 45, 3 (2016), 712–733. https://doi.org/10.1137/13093618X

[15] Konstantinos Semertzidis, Evaggelia Pitoura, and Kostas Lillis. 2015. TimeReach: Historical Reachability Queries on Evolving Graphs. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*, Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens (Eds.). OpenProceedings.org, 121–132. https://doi.org/10.5441/002/edbt.2015.12

[16] Neha Sengupta, Amitabha Bagchi, Maya Ramanath, and Srikanta Bedathur. 2019. ARROW: Approximating Reachability Using Random Walks Over Web-Scale Graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 470–481. https://doi.org/10.1109/ICDE.2019.00049

[17] Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1009–1020. https://doi.org/10.1109/ICDE.2013.6544893

[18] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 845–856. https://doi.org/10.1145/1247480.1247573

[19] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. 2006. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 75. https://doi.org/10.1109/ICDE.2006.53

[20] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability querying: an independent permutation labeling approach. *VLDB J.* 27, 1 (2018), 1–26. https://doi.org/10.1007/s00778-017-0468-3

[21] Dong Wen, Yilun Huang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2020. Efficiently Answering Span-Reachability Queries in Large Temporal Graphs. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1153–1164. https://doi.org/10.1109/ICDE48307.2020.00104

[22] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. 2016. Reachability and time-based path queries in temporal graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 145–156. https://doi.org/10.1109/ICDE.2016.7498236

[23] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2012. GRAIL: a scalable index for reachability queries in very large graphs. *VLDB J.* 21, 4 (2012), 509–534. https://doi.org/10.1007/s00778-011-0256-4

[24] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2013. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *CoRR* abs/1301.0977 (2013). arXiv:1301.0977 http://arxiv.org/abs/1301.0977

[25] Tianming Zhang, Yunjun Gao, Lu Chen, Wei Guo, Shiliang Pu, Baihua Zheng, and Christian S. Jensen. 2019. Efficient distributed reachability querying of massive temporal graphs. *VLDB J.* 28, 6 (2019), 871–896. https://doi.org/10.1007/s00778-019-00572-x

[26] Shuang Zhou, Pingpeng Yuan, Ling Liu, and Hai Jin. 2018. MGTag: a Multi-Dimensional Graph Labeling Scheme for Fast Reachability Queries. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1372–1375. https://doi.org/10.1109/ICDE.2018.00153

[27] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1323–1334. https://doi.org/10.1145/2588555.2612181

Received October 2022; revised January 2023; accepted February 2023