I/O Efficient Label-Constrained Reachability Queries in Large Graphs

Long Yuan
Nanjing University of Science and Technology
longyuan@njust.edu.cn

Xia Li
The University of New South Wales
xia.li@unsw.edu.au

Zi Chen*
Nanjing University of Aeronautics and Astronautics
zichen@nuaa.edu.cn

Xuemin Lin
Shanghai Jiaotong University
xuemin.lin@gmail.com

Xiang Zhao
National University of Defense Technology
xiangzhao@nudt.edu.cn

Wenjie Zhang
The University of New South Wales
wenjie.zhang@unsw.edu.au

ABSTRACT

Computing the reachability between two vertices in a graph is a fundamental problem in graph data analysis. Most of the existing works assume that the edges in the graph have no labels, but in many real application scenarios, edges naturally come with edge-labels, and label constraints may be placed on the edges appearing on a valid path between two query vertices. Therefore, we study the label-constrained reachability (LCR) queries in this paper, where we are given a source vertex s, a target vertex t, a label set Δ, and the goal is to check whether there exists any path from s to t such that all the labels of edges on the path belong to Δ.

A plethora of methods have been proposed in the literature to support the LCR queries. All these methods take the assumption that the graph is resident in the main memory of a machine. Nevertheless, the graphs in many real application scenarios are generally big and may not reside in memory. In these cases, existing methods suffer from serious scalability problem, i.e., result in huge I/O costs. Motivated by this, in this paper, we study the I/O efficient LCR query problem and aim to efficiently answer the LCR queries when the graph cannot fit in the main memory. To achieve this goal, we propose a reduction-based indexing approach. We introduce two elegant graph reduction operators which aims to reduce the size of the graph loaded in memory while preserving the LCR information among the remaining vertices. With these two operators, we devise an index named LCR-Index and propose algorithms to adaptively construct the index based on the available memory. Equipped with LCR-Index, we can answer a LCR query by only scanning the LCR-Index sequentially. Experiments demonstrate our query processing algorithm can handle graphs with billions of edges.

PVLDB Reference Format:
Long Yuan, Xia Li, Zi Chen, Xuemin Lin, Xiang Zhao, Wenjie Zhang. I/O Efficient Label-Constrained Reachability Queries in Large Graphs. PVLDB, 17(10): 2590 - 2602, 2024.
doi:10.14778/3675034.3675049

* Zi Chen is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675049

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://www.dropbox.com/scl/fo/ngye29akb1ukk55s0elw5/h?rlkey=wfgazy8oqsmo07kr5rbk0j70d&dl=0.

1 INTRODUCTION

The reachability query which asks if one vertex can reach another vertex or not is one of the fundamental problems in graph analysis [39, 40]. This seemingly simple but very challenging problem has attracted extensive research for decades [8, 11, 14, 20–22, 41–43, 45, 49–51, 58, 62]. Most of the existing works assume that the edges in the graphs have no labels. However, in many real applications, not all edges are the same, i.e., edges are associated with labels to denote different types of relationships between vertices [7, 56, 57]. For example, in social networks, two persons may have different types of relationships, such as "parentOf", "friendOf", "employeeOf", "colleagueOf", etc. Therefore, many applications place constraints on the edges appearing on a valid path when determining the reachability between two vertices, which leads to the study of label-constrained reachability (LCR) queries [6, 19, 37, 44, 64]. Formally, given a source vertex s, a target vertex t, and a label set Δ, a LCR query checks whether there exists a path from s to t, where the label of each edge on the path belongs to Δ.

Applications. LCR queries can be used in many real application scenarios, for example,

(A) Relationship discovery in social network. In a social network, a vertex represents an entity (e.g., user, poster, organization) and there is an edge if two entities are related. These relationships could be different types like "brotherOf", "friendOf", "employeeOf", "colleagueOf", "like", "follow". A LCR query can be used to determine if two persons are related via a series of given relationships. For example, a common practice in social network analysis for counter-terrorism is to check if two suspects are connected by certain types of relationships such as "friendOf", "brotherOf" [13].

(B) Metabolic chain reaction in metabolic networks. In metabolic networks, each vertex represents a chemical compound and a direct edge indicates a chemical reaction from one compound to another where the edge label records the enzymes which control the reaction. One of the basic questions is to determine whether there is a

certain pathway between two compounds that can be active or not under a set of enzymes, which can be modeled as LCR queries [19].

(C) One of the most important operators for the language of regular path queries. Regular path queries have been extensively studied [26, 31, 36] and are supported in practical graph query languages such as SPARQL, PGQL, and openCypher [44]. A LCR query can be can be described by the regular expression $(l_1|l_2|\dots|l_k)^*$, where $\Delta = \{l_1, l_2, \dots, l_k\}$, | denotes alternation, and $*$ denotes the Kleen star. LCR query is one of the most important operators in regular path queries. An analysis of SPARQL query logs on Wikidata17 knowledge graph reveals that 65% of the regular path queries can be expressed by LCR queries [3].

Motivation. Due to its wide application, LCR queries have attracted considerable attention in recent years and a plethora of methods have been proposed in the literature to accelerate the LCR query processing [6, 19, 37, 44, 64]. All the existing methods assume that graphs are resident in memory. However, as graphs continue to grow to have billions of vertices and edges, the size of many real-world graphs exceeds the capacity of the main memory, which leads to serious scalability problems for existing methods. On the other hand, people have started to leverage the massive storage to keep the large-sized edge data for holding a large-scale graph at low cost and devise various scalable I/O efficient techniques to achieve high performance graph processing, and remarkable results have been obtained in other graph problems [9, 53, 59, 61]. A natural question, therefore, is whether it is possible to devise an I/O efficient solution to the LCR queries? Motivated by this, in this paper, we study I/O efficient algorithms for LCR queries when the input graph G cannot be entirely held in main memory.

Our approach. Nevertheless, it is challenging to design such an I/O efficient algorithm for LCR queries due to the inherent poor locality caused by irregular access pattern of graph data [12, 35]. To address this challenge, we propose a new reduction-based indexing framework for I/O efficient LCR query processing. Since the large size of graphs leads to the insufficiency of main memory, the main idea of our framework is to make the size of the input graph smaller than that of the available memory by reduction while keep the LCR information in the reduced graph as much as possible. The benefits of this idea are twofolds: 1) as the size of the graph is reduced, a smaller graph facilitates the design of an I/O efficient algorithm intuitively; 2) since the graph is reduced based on the size of main memory, the available memory can be fully used as well. Following this idea, we propose the concept of LCR preserved graph (LCR-PG) which has a smaller size than the input graph while preserves the LCR information among the vertex pairs in it, and two graph reduction operators are designed accordingly. With the graph reduction operators, our new framework generates a series of LCR-PGs as the index. Since LCR-PG preserves LCR information of the input graph, we can answer a given query just through the corresponding LCR-PG, which means the given query can be processed with limited memory. However, to make the new framework practically applicable, the following issues need to be addressed: (1) How can a good LCR-PG be obtained in an I/O efficient manner? and (2) How can the LCR-PGs be used to answer the given query I/O efficiently?

Contributions. In this paper, we answer these questions and make the following contributions:

(1) The first work for I/O efficient LCR *query processing.* In this paper, we aim to answer the LCR queries in massive graphs by considering the I/O issues when the memory size is inadequate. To the best of our knowledge, this is the first work to study the I/O efficient LCR query processing problem.

(2) A new reduction-based LCR *query processing framework.* Our main idea to address the problem is through graph reduction. We introduce the concept of LCR preserved graph and propose two graph reduction operators, VR and ER, to reduce the vertex and edge of the graph, respectively. We discuss how to use these two graph reduction operators to generate a series of LCR preserved graphs as index, and answer the query through the corresponding LCR preserved graph.

(3) I/O efficient indexing and query processing algorithms. Following the reduction-based framework, we propose I/O efficient algorithms to construct the index and process the given query. Remarkably, our query processing algorithm can answer a LCR query by scanning the index on the disk sequentially once. We also provide theoretical I/O complexity analysis of the proposed algorithms.

(4) Extensive performance studies on large datasets. We conduct extensive performance studies using real-world graphs. The experimental results demonstrate that our query processing algorithm can handle graphs with billions of edges and achieve up to 3 orders of magnitude speedup compared with the existing solutions.

2 RELATED WORK

In this section, we review the related work from three categories, namely, label-constrained reachability queries, reachability queries on unlabelled graphs, and other I/O efficient graph algorithms.

Label-constrained reachability query. In the literature, extensive research efforts have been devoted to the label-constrained reachability queries. A direct approach to answering the query is based on online search as follows: starting from s and t, we traverse the graph in BFS manner following the edges with labels in Δ. If these two traverses can meet at a vertex, then t is reachable from t regarding Δ. Otherwise, t is not reachable from t regarding Δ. [19] is the first work on the index-based approach for LCR queries. It devises a tree-based index framework to process the label-constrained reachability queries. Following the idea of [19], [64] leverages the strongly connected components in the graph to further improve the query processing performance. [44] proposes a landmark-based index by utilizing the reachability information of important vertices. Recently, P2H+ [37] designs a 2-hop labelling index for the label-constrained reachability queries and shows that the proposed method outperforms [44] in terms of the query processing. [6] extends P2H+ [37] to support the dynamic update of the graphs. Nevertheless, all the existing methods assume that the input graphs reside in main memory, and thus cannot be directly used to address our problem due to the large amount of I/Os [34]. Then, another natural question raised is whether is it possible to extend these approaches for I/O efficient setting to address our problem? Unfortunately, the answer is negative. For example, I/O efficient BFS algorithms for directed graphs have been investigated in the literature [4, 23, 25] and can be adjusted for our problem. The-state-of-art I/O efficient BFS algorithm for directed graphs [23] uses a buffered repository tree [4] to record the edges that point to previously seen nodes and an external queue [46] to conduct the traversal. However, it needs $\Omega(n + m/B)$ I/Os [23], where n/m represents the number of vertices/edges of the graph and B represents the block transfer size between main memory and disk. Thus, it is impracticable for real applications [2]. [63] alleviates the I/O issue by allowing all the vertices of the graph to be kept in the memory, but the query processing performance is still poor as evaluated in our experiments. For the index-based approaches, the index structures of these methods are generally large and the index construction algorithms are complex, which makes them infeasible to be extended to support efficient LCR queries when the input graphs are stored on external storage devices. Take the state-of-the-art approach P2H+ in this category as an example. P2H+ adopts a 2-hop labelling strategy and answers a given LCR query $q(s, t, \Delta)$ by only iterating the elements attached in s and t in the index. However, the space consumption of the P2H+ index structure is $O(n^2 \cdot 2^{|\Sigma|})$, where $|\Sigma|$ is the number of the edge labels in the graph. Clearly, this approach is not scalable for large graphs even though the index structure could be stored on the disk. Moreover, in the procedure of the index construction, a series of BFS explorations have to be performed, which further makes this approach inapplicable in our I/O efficient scenario as analyzed above. Therefore, we have to design a new I/O efficient method from scratch for LCR queries.

Reachability query on unlabelled graphs. Reachability query on unlabelled graphs has been studied in the literature. The classic approach to answering the reachability is BFS. However, this approach is inefficient as it may traverse the entire network. Therefore, index-based approaches are studied to tackle this problem [8, 11, 14, 15, 20–22, 41–43, 45, 48–51, 62]. [52] provides a comprehensive survey on the reachability query on unlabelled graphs. However, these approaches just consider the unlabelled graphs and they cannot be used directly to answer our queries.

I/O efficient graph algorithms. With the proliferation of graph applications [5, 27, 29, 32, 33], several graph algorithms focusing on I/O efficiency have been proposed in the literature. [9] describes an I/O efficient algorithm for the core decomposition in massive networks. [60] studies an I/O efficient algorithm to compute the strongly connected components in a graph in the semi-external model and [59] extends the algorithm to the external memory model. I/O efficient algorithms for the triangle enumeration problem are presented in [18]. And I/O efficient algorithms for the maximal clique enumeration related problems are proposed in [10, 53]. The I/O efficient algorithms for the k-truss decomposition problem and k-edge connected component decomposition problem are investigated in [47] and [54, 55], respectively. An I/O efficient semi-external algorithm for DFS is proposed in [61]. [24] devises an I/O efficient algorithm for subgraph enumeration. However, all these I/O efficient algorithms are designed for other graph problems and cannot be used for LCR queries.

3 PROBLEM DEFINITION

Let $G = (V, E, \Sigma, \lambda)$ be a directed edge-labeled graph where V is a set of vertices, E is a set of directed edges, Σ is a set of edge labels, and $\lambda : E \to \Sigma$ is a function that assigns each edge a label $l \in \Sigma$. We use $n = |V|$ and $m = |E|$ to denote the number of vertices and edges in the graph, respectively. For each vertex $v \in V$, the in-neighbors (resp. out-neighbors) of v, denoted as $\mathsf{nbr}^-(v, G)$ (resp. $\mathsf{nbr}^+(v, G)$), is defined as $\mathsf{nbr}^-(v, G) = \{u | (u, v) \in E\}$ (resp. $\mathsf{nbr}^+(v, G) = \{u | (v, u) \in E\}$). The in-degree (resp. out-degree) of a vertex $v \in V$, denoted by $\deg^-(v, G)$ (resp. $\deg^+(v, G)$), is the number of in-neighbors (resp. out-neighbors) of v, i.e., $\deg^-(v, G) = |\mathsf{nbr}^-(v, G)|$ (resp. $\deg^+(v, G) = |\mathsf{nbr}^+(v, G)|$). We call the in-neighbors and out-neighbors together of a vertex v as its neighbors and the number of neighbors as its degree, denoted by $\mathsf{nbr}(v, G)$ and $\deg(v, G)$, respectively. For each directed edge $e = (u, v) \in E$, we use $\lambda((u, v), G)$ to denote its associated label. In the graph G, a path is a sequence of vertices $p_{v_1,v_k} = (v_1, v_2, \cdots, v_k)$, where $(v_i, v_{i+1}) \in E$ for each $1 \leq i < k$. Given a path p in G, the path label of p, denoted by $\ell(p)$, is defined as $\ell(p) = \cup_{(u,v)\in p}\lambda((u, v), G)$. Given two paths p_{v_1,v_i} and p_{v_i,v_k}, the concatenation of p_{v_1,v_i} and p_{v_i,v_k} generates a path p_{v_1,v_k} with $\ell(p_{v_1,v_k}) = \ell(p_{v_1,v_i}) \cup \ell(p_{v_i,v_k})$.

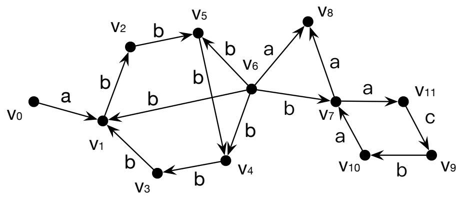


Figure 1: LCR **Reachability Query**

Definition 3.1: (Label-Constrained Reachable) Given two vertices s, t in a graph $G = (V, E, \Sigma, \lambda)$ and a set of edge labels $\Delta \subseteq \Sigma$, let $\mathcal{P}$ be the set of all paths from s to t in G, t is label-constrained reachable from s, denoted by $s \mapsto_{\Delta} t(G)$, if there is a path $p \in \mathcal{P}$ such that $\ell(p) \subseteq \Delta$. □

For simplicity, we omit G in the notations hereafter if the context is self-evident. For any vertex v, v is label-constrained reachable from itself regarding any $\Delta \subseteq \Sigma$.

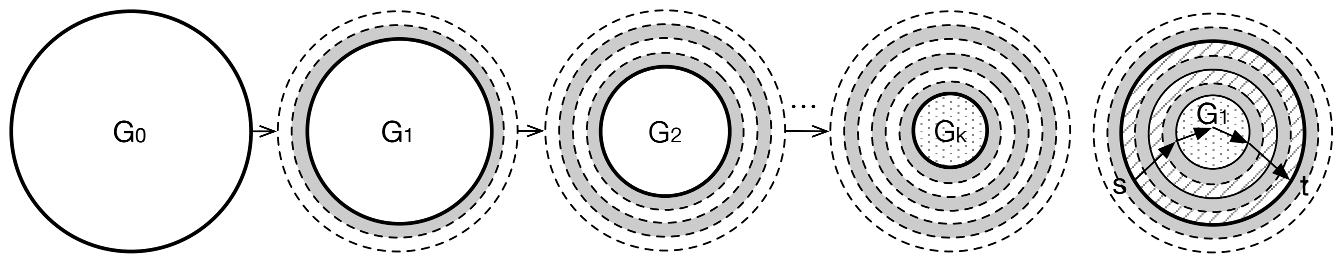
Example 3.1: Consider the graph shown in Figure 1, $\Sigma = \{a, b, c\}$, and the label of each edge is shown beside the corresponding edges. For example, $\lambda((v_0, v_1)) = a$. For path $p_{v_0,v_2} = \{v_0, v_1, v_2\}$, $\ell(p_{v_0,v_2}) = \{a, b\}$ as $\lambda((v_0, v_1)) = \{a\}$ and $\lambda((v_1, v_2)) = \{b\}$. For vertices v_0 and v_5, if $\Delta = \{a, b\}$, then $v_0 \mapsto_{\{a,b\}} v_5$ as there exists a path $p_{v_0,v_5} = \{v_0, v_1, v_2, v_5\}$, $\ell(p_{v_0,v_5}) = \{a, b\} \subseteq \Delta = \{a, b\}$. □

Problem Statement. Given two vertices s, t in a graph G and a set of edge labels Δ, a label-constrained reachability (LCR) query $q(s, t, \Delta)$ asks whether t is label-constrained reachable from s regarding Δ.

Since the graphs in real applications can be massive, in this paper, we assume that the graph G cannot reside entirely in the main memory. We aim to design I/O efficient algorithms where the input graph resides in the disk to answer the label-constrained reachability queries. When analyzing the I/O complexity of our algorithms, we use the standard I/O complexity notations in [1] as follows: M is the main memory size and B is the disk block size. The I/O complexity to scan N elements is $\mathsf{scan}(N) = \Theta(\frac{N}{B})$, and the I/O complexity to sort N elements is $\mathsf{sort}(N) = O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$.

4 A REDUCTION-BASED FRAMEWORK

In this section, we present the general idea of our algorithm. As discussed in Section 1, the main reason leading to the insufficiency



(a) Indexing Phase (b) Query Processing Phase

Figure 2: Main Framework of Our Approach

of memory is the input graph is too large, which means if we can reduce the input graph to the size that can fit in memory while the reachability information can be recovered based on the reduced graph, then the problem is totally addressed. Following this idea, our I/O efficient solution to the LCR queries contains two phases: reduction-based indexing phase and index-based query processing phase. Figure 2 illustrates the main framework of our idea.

Phase 1. Reduction-based indexing. In the reduction-based indexing phase, we aim to reduce the input graph size as much as possible while preserving the LCR reachability information to facilitate the query processing in phase 2. To achieve this seemingly contradictory goal, we generate a series of LCR preserved graphs $G_0, G_1, \ldots, G_k$, which are defined as follows:

Definition 4.1: (LCR **Preserved Graph)** Given a graph G, a LCR Preserved Graph (LCR-PG) G' is a graph with $V(G') \subset V(G)$ such that $\forall s,t \in V(G')$, if $s \mapsto_\Delta t(G')$, then $s \mapsto_\Delta t(G)$, where $V(G')/V(G)$ denotes the vertices of G'/G. □

Specifically, starting from $G_0 = G$, we generate G_i based on G_{i-1} by two reduction operators (denoted by dashed ring and shadowed ring in Figure 2 (a)), where $1 \leq i \leq k$, and guarantee that G_i is a LCR-PG of G_{i-1}. The indexing phase stops when G_k can be loaded in the main memory. The benefits of this approach are twofolds: (1) since $V(G_{i+1}) \subset V(G_i)$, we can often obtain a smaller graph as the indexing procedure continues, while G_{i+1} is the LCR-PG of G_i that means the LCR reachability information could be kept. (2) since the indexing phase stops when G_k can be loaded in the memory which means it can use the available memory effectively, and the performance can be improved naturally from the availability of extra memory.

Phase 2. LCR-PG-**based query processing.** According to Definition 4.1, to answer a label-constrained reachability query $q(s, t, \Delta)$, if s and t are in G_k, then we can directly answer the query based on G_k. Otherwise, we can construct the LCR-PG G_i containing s and t first, and then obtain the reachability information on LCR-PG accordingly.

Challenges. Following the above approach, the LCR queries can be answered with limited memory obviously. However, to make our approach practically applicable, the following challenges should be addressed when designing our algorithm:

- Reducibility: for a given graph G_i in the indexing phase, the size of the graph should be able to be reduced continuously to meet the limited memory condition.
- LCR preservability: for a given graph G_i generated in the indexing phase, the label-constrained reachability information regarding two vertices s and t in G_i should be retained for the LCR query processing.
- I/O efficiency: since the available memory is limited, all the algorithms should be able to be implemented in an I/O efficient manner.

5 INDEX CONSTRUCTION AND QUERY PROCESSING

In this section, we focus on how to address the challenges discussed above. We first introduce the main idea of the indexing approach and show how to guarantee the reducibility and LCR preservability during the indexing. Then, we present our techniques to construct the index in an I/O efficient manner. At last, we present the query processing algorithm based on the proposed index.

5.1 Overview of Index Construction Algorithm

Our index is based on the LCR dominating label set, which is defined as follows:

Definition 5.1: (LCR **dominating label set)** Given a graph G, for an edge (u, v), a LCR dominating label set of (u, v), denoted by $\Phi((u,v),G) = (\phi_1, \ldots, \phi_n)$, is a set of edge label sets such that $\phi_i \subseteq \Sigma$ and $\nexists\, \phi' \in \Phi((u,v),G)$ with $\phi' \subseteq \phi_i$ for every $1 \leq i \leq n$. □

Obviously, for each edge (u, v), the edge label $\lambda((u,v),G)$ assigned by λ in G is a special case of LCR dominating label set.

Definition 5.2: (LCR **dominating label set concatenation operator $\oplus$)** Given the LCR dominating label sets $\Phi((u,v),G)$ of (u,v) and $\Phi((v,w),G)$ of (v,w), let C be the set of all concatenations of $\phi_{u,v}$ and $\phi_{v,w}$ after removing repeated labels, where $\phi_{u,v} \in \Phi((u,v),G)$ and $\phi_{v,w} \in \Phi((v,w),G)$, the LCR dominating label set concatenation of $\Phi((u,v),G)$ and $\Phi((v,w),G)$ is defined as $\Phi((u,v),G) \oplus \Phi((v,w),G) = \{\phi \in C | \nexists\, \phi' \in C, \phi' \subseteq \phi\}$. □

Definition 5.3: (LCR **dominating label set union operator $\uplus$)** Given two LCR dominating label sets $\Phi((u,v),G)$ and $\Phi'((u,v),G)$ of (u,v), the LCR dominating label set union of $\Phi((u,v),G)$ and $\Phi'((u,v),G)$ is defined as: $\Phi((u,v),G) \uplus \Phi'((u,v),G) = \{\phi \in \{\Phi((u,v),G) \cup \Phi'((u,v),G)\} | \nexists\, \phi' \in \{\Phi((u,v),G) \cup \Phi'((u,v),G)\}, \phi' \subseteq \phi\}$. □

For the ease of presentation, Definition 5.3 only shows two operands for $\uplus$ operator, but it supports more than two operands in the same way. Based on the LCR dominating label set, we define the

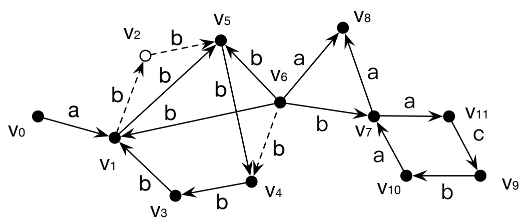


Figure 3: $\mathsf{VR}(v_2, G)$ **and** $\mathsf{ER}((v_6, v_4), G)$**)**

LCR vertex reduction operator and LCR edge reduction operator, which lay the foundation of our indexing construction.

Definition 5.4: (LCR **vertex reduction operator** $\mathsf{VR}(v, G)$**)** Given a graph G, for a vertex $v \in V$, the LCR vertex reduction operator applied on v in G, denoted by $\mathsf{VE}(v, G)$, transforms G into a new graph G' as follows: the vertex v and its incident edges are removed first. Then, for every pair of $u \in \mathsf{nbr}^-(v, G)$ and $w \in \mathsf{nbr}^+(v, G)$, if $(u, w) \notin E(G)$, a new edge (u, w) with LCR dominating label set $\Phi((u, w), G') = \Phi((u, v), G) \oplus \Phi((v, w), G)$ is inserted. Otherwise, the LCR dominating label set of (u, w) is updated with $\Phi((u, w), G') = \Phi((u, w), G) \uplus (\Phi((u, v), G) \oplus \Phi((v, w), G))$. □

Definition 5.5: (LCR **edge reduction operator** $\mathsf{ER}((u, v), G)$**)** Given a graph G, for an edge (u, v), the LCR edge reduction operator applied on (u, v) in G, denoted by $\mathsf{ER}((u, v), G)$, transforms G into a new graph G' as follows: (u, v) is removed from G if there exist two edges $(u, w) \in E(G)$ and $(w, v) \in E(G)$ such that for each $\phi \in \Phi((u, v), G)$, there exists $\phi' \in \Phi((u, v), G) \oplus \Phi((v, w), G)$ with $\phi' \subseteq \phi$. Otherwise, G' keeps the same as G. □

Example 5.1: Consider the graph G shown in Figure 1, Figure 3 shows $\mathsf{VR}(v_2, G)$ and $\mathsf{ER}((v_6, v_4), G))$ on G, which are illustrated by dash lines. Specifically, when applying LCR vertex reduction operator on v_2, as $\mathsf{nbr}^-(v_2, G) = \{v_1\}$ and $\mathsf{nbr}^+(v_2, G) = \{v_5\}$, and $\lambda((v_1, v_2)) = \{b\}$, $\lambda((v_2, v_5)) = \{b\}$. Then vertex v_2 together with (v_1, v_2) and (v_2, v_5) are removed from G and a new edge (v_1, v_5) with LCR dominating label set $\Phi((v_1, v_2), G) \oplus \Phi((v_2, v_5), G) = \{b\}$ is inserted. When applying LCR edge reduction operator on (v_6, v_4), as $\lambda((v_6, v_5)) = \{b\}$ and $\lambda((v_5, v_4)) = \{b\}$, $\Phi((v_6, v_5), G) \oplus \Phi((v_5, v_4), G) = \{b\} \subseteq \Phi((v_6, v_4), G) = \{b\}$, then (v_6, v_4) is removed from G. □

With the vertex reduction operator and edge reduction operator, we are ready to show our index construction algorithm, which is based on the following definition:

Definition 5.6: (Maximal independent set) Given a graph G, the independent set of G is a set of vertices in G such that no two vertices in the set are adjacent in G. A maximal independent set (MIS) is an independent set that is not a subset of another independent set in G. □

The main framework of the index construction algorithm is shown in Algorithm 1. For a given graph G, G_0 is initialized as G and i and k are initialized as 0 (line 1). After that, the index construction algorithm iteratively generates a series of graphs $G_1, G_2, \ldots, G_k$ where G_i is generated based on G_{i-1} with a smaller number of nodes (line 2-8). In each iteration, the maximal independent set I_i of G_i is computed first (line 3). Then, for each $v \in I_i$, the vertex reduction operator is applied on v in G_i (line 4-5). When the vertex

Algorithm 1 $\mathsf{LCRIndexCons}(G)$

```
1: $i, k \leftarrow 0$; $G_i \leftarrow G$;
2: while $G_i$ cannot be loaded in main memory do
3:   $I_i \leftarrow \mathsf{MIS}(G_i)$;
4:   for each $v \in I_i$ do
5:     $G_i \leftarrow \mathsf{VR}(v, G_i)$;
6:   for each $(u, v) \in E(G_i)$ do
7:     $G_i \leftarrow \mathsf{ER}((u, v), G_i)$;
8:   $G_{i+1} \leftarrow G_i$; $i \leftarrow i + 1$; $k \leftarrow i$;
```

reduction finishes, the edge reduction operator is applied on each edge (u, v) in G_i (line 6-7). The index construction stops when G_k can be loaded in the main memory (line 2).

Example 5.2: Figure 4 shows the procedures of Algorithm 1 to construct the index. It finishes the constructions in two iterations. In first iteration, the maximal independent set $I_0 = \{v_0, v_2, v_3, v_8, v_9\}$ of G_0 is computed, which is shown by circle in Figure 4(a). Then, it conducts the vertex reduction on the vertices in I_0 and edge reduction on the edge (v_6, v_1) and (v_6, v_4) to obtain G_1. In the second iteration, the maximal independent set $I_1 = \{v_1, v_6, v_{10}\}$ of G_1 is computed, which is shown in Figure 4(c). After that, it conducts the vertex reduction on the vertices in I_1 and no edge can be reduced following Definition 5.5. The reduced graph G_2 is shown in Figure 4(d). Assume G_2 can be loaded in the memory, and the index construction finishes. □

Following the procedure of Algorithm 1, it is obvious that the index satisfies the reducibility property due to the vertex/edge reduction in line 5/7. Meanwhile, the index also satisfies the LCR preservability property, which is proved as follows:

Definition 5.7: (Maximal independent set order) Given a vertex v in graph G, let $G_0, G_1, \ldots, G_k$ be the graphs generated during the indexing phase, the maximal independent set order of v, denoted by $o(v)$, is defined as the maximum i such that $v \in V(G_i)$. □

Theorem 5.1: *Given two vertices u and v in graph G, for a set of edge labels $\Delta \subseteq \Sigma$, if there is a path p from u to v in G with $\ell(p) \subseteq \Delta$, then, there is a path p' from u to v in G_i such that $\ell(p') \subseteq \ell(p) \subseteq \Delta$, where $i \leq \min\{o(u), o(v)\}$.*

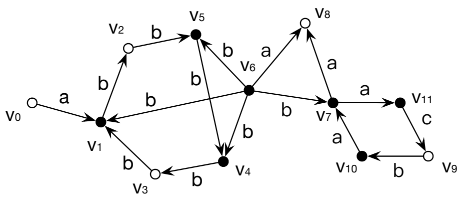
According to Theorem 5.1, the LCR preservability is also guaranteed regarding the index generated by Algorithm 1.

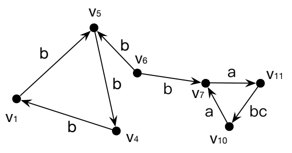
5.2 I/O Efficient Index Construction

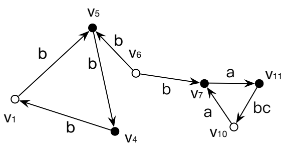
As shown in Algorithm 1, the key procedures of the indexing phase are maximal independent set computation, vertex reduction, and edge reduction. In the following, we will introduce them in order.

5.2.1 I/O Efficient Vertex Reduction.

I/O efficient maximal independent set computation. Intuitively, the size of the maximal independent set I_i computed in line 3 of Algorithm 1 should be as large as possible to reduce the number of iterations of Algorithm 1, which lead to the maximum independent set problem. No matter how desirable, the maximum independent set problem is known to be NP-hard [38], and for any $\epsilon > 0$, there is no polynomial-time $n^{1-\epsilon}$ approximation algorithm for the maximum independent set problem [17]. On the other hand,


(a) Iteration 1 (MIS Computation, G_0)


(b) Iteration 1 (VR/ER, G_1)


(c) Iteration 2 (MIS Computation, G_1)

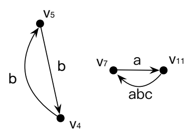

(d) Iteration 2 (VR/ER, G_2)

Figure 4: Index Construction Procedure

Algorithm 2 IOMIS(G)

```
1:  reassign a new vertex id to each vertex $v$ such that for two vertices $u$ and $v$, if $u \prec v$, the new id of $u$ is less than $v$;
2:  $G'$ ←transforms the edges in $G$ with the new vertex id, and reverse the direction of an edge $(u, v)$ if the new id of $u$ is bigger than that of $v$.
3:  sort the edge $(u, v) \in E(G')$ based on the non-decreasing order of the new id of $u$;
4:  I/O efficient Priority Queue $Q \leftarrow \emptyset$;
5:  for each vertex $v$ in the increasing order of their new ids do
6:    pop all the elements $(v, I(v'))$ related to $v$ in $Q$;
7:    if $\exists I(v') = 1$ then $I(v) \leftarrow 0$; else $I(v) \leftarrow 1$;
8:    for each out-neighbor $u$ of $v$ in $G'$ do
9:      push element $(u, I(v))$ into $Q$;
10: return $I \leftarrow$ all the vertices with $I(v) = 1$;
```

[16] shows that show that for general graphs of bounded degree $\mathcal{D}$, the greedy algorithm produces $\frac{\mathcal{D}+1}{3}$-factor approximation result for the maximum independent set problem and [28] experimentally shows that greedy algorithm works well on real graphs in practice. Therefore, we leverage the greedy-based I/O efficient maximal independent set algorithm in the literature [28, 30] for our index construction whose pseudocode is shown Algorithm 2.

Briefly, Algorithm 2 processes the vertices in the non-decreasing order of their degrees (the vertex with smaller id comes first if two vertices have the same degree). Then, it selects the vertex whose processed neighbors are not in the maximal independent set as a member of the maximal independent set. Specifically, given a graph G, the algorithm first define an order $\prec$ of the vertices such that $u \prec v$ if (1) $\deg(u) < \deg(v)$; (2) $\deg(u) = \deg(v)$ and $\mathrm{id}(u) < \mathrm{id}(v)$. Then, it pre-processes the input graph by: (1) reassigning a new vertex id to each vertex v such that for two vertices u and v, if $u \prec v$, the new id of u is less than v (line 1). (2) transforming the edges of G with the new vertex id, and reversing the direction of an edge (u, v) if the new id of u is bigger than that of v (line 2). (3) sorting the edge $(u, v) \in E(G')$ based on the non-decreasing order of the new id of u (line 3). After the pre-processing, the $\prec$ relationship between two vertices incident to an edge can be directly determined by their new ids. Then, it initializes an I/O efficient minimum priority queue [46] (line 4) and iterates the vertices in the increasing order of their new ids (line 5). For each vertex v, it uses $I(v) = 1/0$ to indicate v is in/not in the maximal independent set. It pops all the elements $(v, I(v'))$ in Q that maintains the information about whether the processed neighbors of v are in the maximal independent set or not (line 6). If there exists at least one $I(v')$ is 1, then it sets $I(v)$ as 0. Otherwise, $I(v)$ is set as 1 (line 7). After that, for each out-neighbor u of v, $(u, I(v))$ with the priority of the new id of u is further pushed into Q (line 8-9). The procedure finishes all the vertices have been processed (line 5). At last, the vertices with $I(v) = 1$ are returned as the maximal independent set (line 12). As shown in [30], the I/O complexity of Algorithm 2 to compute the maximal independent set for a given graph G is $O(\mathrm{sort}(G))$.

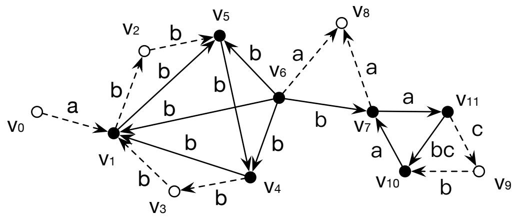


Figure 5: I/O Efficient MIS and Vertex Reduction on G_0

Example 5.3: Consider the graph G shown in Figure 1, Figure 5 shows the maximal independent set of G computed by Algorithm 2, which is illustrated by circle. Following the definition of $\prec$, we have $v_0 \prec v_2 \prec v_3 \prec v_8 \prec v_9 \prec v_{10} \prec v_{11} \prec v_4 \prec v_5 \prec v_7 \prec v_6 \prec v_1$ (For the clearness of presentation, we omit the pre-processing in line 1-3 of Algorithm 2 and assume we have known the $\prec$ relationship between two vertices). As v_0 has the smallest order and there is no element related to v_0 in Q, $I(v_0) = 1$ and $(v_1, 1)$ is pushed in Q. v_2, v_3, v_8, v_9 are processed similarly. For v_{10}, as $(v_{10}, 1)$ is pushed in Q when processing v_9, $I(v_{10}) = 0$. When Algorithm 2 finishes, the computed maximal independent set contains $\{v_0, v_2, v_3, v_8, v_9\}$. □

I/O efficient vertex reduction. For the vertex reduction, if we perform the reduction on each vertex individually following Definition 5.4, it needs at least $O(|I|)$ I/Os, which is inefficient. Moreover, as the vertex reduction involves new edge generation and edge elimination, this approach may process the same edge several times repeatedly, which would lead to unfruitful I/Os. To address these problems, we propose an I/O efficient vertex reduction algorithm that processes the vertices in I as a whole and integrates the operations on the related edges accordingly, which is shown in Algorithm 3.

Given the maximal independent set I of G, it first sorts the vertices in the increasing order of their ids (line 2). Then, the edges (u, v) of G are sorted based on $\{\mathrm{id}(u), \mathrm{id}(v)\}$ and stored as E^+(line 3). Similarly, it reverses the edges in G, sorts these edges in the same way as line 3, and stores them as E^- (line 4). After that, for each vertex v, it retrieves the in-neighbors and out-neighbors of v by sequentially scanning E^- and E^+, and generates the edge (u, w) with labels $\Phi((u, v), G) \oplus \Phi((v, w), G)$ following Definition 5.2 (line 6-11). The newly generated edges are stored in $\mathcal{S}$ (line 10), and the edges that have a vertex incident to vertices in the maximal

Algorithm 3 IOVertexReduction(I, G)

```
1: $\mathcal{S} \leftarrow \emptyset, \mathcal{E} \leftarrow \emptyset$;
2: $V \leftarrow$ sort$(v \in I)$ based on $\{\mathsf{id}(v)\}$;
3: $E^+ \leftarrow$ sort$((u,v) \in E(G))$ based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$;
4: $E^- \leftarrow$ reverse the direction of edges in $G$ and sort these edges as line 3;
5: for each $v \in V$ do
6:   if $(v,u) \in E^-$ by scan$(E^-)$ then
7:     for each $(v,u) \in E^-$ by scan$(E^-)$ do
8:       $\mathcal{E} \leftarrow \mathcal{E} \cup (u,v)$;
9:       if $(v,w) \in E^+$ by scan$(E^+)$ then
10:        $\Phi((u,w),G) \leftarrow \Phi((u,v),G) \oplus \Phi((v,w),G)$; $\mathcal{S} \leftarrow \mathcal{S} \cup (u,w)$;
11:        $\mathcal{E} \leftarrow \mathcal{E} \cup (v,w)$;
12:   else for each $(v,w) \in E^+$ by scan$(E^+)$ do $\mathcal{E} \leftarrow \mathcal{E} \cup (v,w)$;
13: $\mathcal{E}' \leftarrow$ sort$((u,v) \in \mathcal{E})$ based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$;
14: $E'^+ \leftarrow$ remove edges $(u,v) \in E^+$ that is also in $\mathcal{E}'$ by scan$(E^+)$ and
    scan$(\mathcal{E})$;
15: $\mathcal{S}' \leftarrow$ sort$((u,v) \in \mathcal{S})$ based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$;
16: $E''^+ \leftarrow$ add edges $(u,v) \in \mathcal{S}'$ to $E'^+$ and update $\Phi((u,v),G)$ following
    $\uplus$ if $(u,v) \in E'^+$ by scan$(\mathcal{S}')$ and scan$(E'^+)$;
17: return $G'(V(G) - I, E''^+)$;
```

independent set are stored in $\mathcal{E}$ (line 8, 11, 12). Then, the edges $(u,v) \in \mathcal{E}$ are sorted based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$ and removed from E^+ (line 13-14). At last, the edges $(u,v) \in \mathcal{S}$ are sorted based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$ (line 15) and added into E'^+ by scanning scan$(\mathcal{S}')$ and scan(E'^+). If (u,v) are also in E'^+, $\Phi((u,v),G)$ is updated by unionizing the LCR dominating label set of (u,v) in S and E'^+ (line 16). The newly generated graph is returned in line 17.

Example 5.4: Continue Example 5.3 in which maximal independent set $I = \{v_0, v_2, v_3, v_8, v_9\}$ is computed. To conduct the vertex reduction, Algorithm 3 first sorts the vertices in I by their ids, namely $V = \{v_0, v_2, v_3, v_8, v_9\}$. Then, E^+/E^- is obtained by sorting the edges (u,v) based on $\{\mathsf{id}(u), \mathsf{id}(v)\}/\{\mathsf{id}(v), \mathsf{id}(u)\}$, namely $E^+ = \{(v_0, v_1, a), (v_1, v_2, b), (v_1, v_5, b), (v_2, v_5, b), \ldots, (v_{11}, v_9, c)\}$ and $E^- = \{(v_1, v_0, a), (v_1, v_3, b), (v_1, v_4, b), (v_1, v_6, b), \ldots, (v_{11}, v_7, a)\}$. Then it scans V, E^+ and E^- simultaneously to perform the vertex reduction. v_0 does not have any in-neighbors, which is a trivial case, and only (v_0, v_1, a) is added into $\mathcal{E}$. For v_2, its in-neighbor is v_1 by E^- and its out-neighbor is v_5, then it sets $\Phi((v_1, v_5), G) = \Phi((v_1, v_2), G) \oplus \Phi((v_2, v_5)) = \{b\}$, and adds (v_1, v_5, b) into $\mathcal{S}$, and (v_1, v_2, b), (v_2, v_5, b) are inserted into $\mathcal{E}$. The remaining vertices in V are processed similarly. We have $\mathcal{E} = \{(v_0, v_1, a), (v_1, v_2, b), (v_2, v_5, b), (v_3, v_1, b), (v_4, v_3, b), (v_6, v_8, a), (v_7, v_8, a), (v_9, v_{10}, b), (v_{11}, v_9, c)\}$, $\mathcal{S} = (v_1, v_5, b), (v_4, v_1, b), (v_{11}, v_{10}, bc)$. After that, the edges in $\mathcal{E}$ are sorted and removed from E^+, which leads to E'^+. At last, the edges in $\mathcal{S}$ are sorted and inserted in E'^+. The subgraph induced by the solid edge in Figure 5 is the returned graph when Algorithm 3 finishes. □

5.2.2 *I/O Efficient Edge Reduction.*

For the edge reduction, it is obvious that applying the edge reduction operator on each edge following Algorithm 1 directly lead to $O(m)$ I/Os at least, which is inefficient. To reduce the I/Os, we have to design an algorithm that can fully utilize every I/O when handling the edge reduction. Nevertheless, different from the in-memory algorithm which knows the whole input graph, it is challenging to design such an I/O efficient algorithm as the memory is limited and the global information about the input graph is unknown. Consider

Algorithm 4 IOEdgeReduction(G)

```
1: $E^+ \leftarrow$ sort$((u,v) \in E(G))$ based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$;
2: $E^- \leftarrow$ sort$((u,v) \in E(G))$ based on $\{\mathsf{id}(v), \mathsf{id}(u)\}$;
3: $S \leftarrow \emptyset$; $S^* \leftarrow \emptyset$; $\mathcal{E} \leftarrow \emptyset$;
4: for each $v \in V(G)$ do
5:   $S \leftarrow S \cup \{v\}$; $S^* \leftarrow S^* \cup \{v\} \cup \mathsf{nbr}^-(v,G) \cup \mathsf{nbr}^+(v,G)$;
6:   if $(\frac{\overline{\mathsf{deg}} \cdot |S^*|}{2} \geq cM)$ then
7:     scan$(E^+)$ and scan$(E^-)$ to extract $G_{S^*}$;
8:     for each $u \in V(G_{S^*})$ do
9:       for each $v \in \mathsf{nbr}^+(u, G_{S^*})$ do
10:        for each $w \in \mathsf{nbr}^+(v, G_{S^*})$ do
11:          if $w \in \mathsf{nbr}^+(u, G_{S^*})$ then
12:            for each $\phi \in \Phi((u,v), G_{S^*})$ do
13:              if $\nexists\, \phi' \in \Phi((u,w), G_{S^*}) \oplus \Phi((w,v), G_{S^*})$ s.t. $\phi' \subseteq \phi$ then
14:                break;
15:            mark $(u,v)$ as removed from $G_{S^*}$;
16:            $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u,v)\}$;
17:   $S \leftarrow \emptyset$; $S^* \leftarrow \emptyset$;
18: sort$((u,v) \in \mathcal{E})$ based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$;
19: remove edges in $\mathcal{E}$ from $E^+$ by scan$(\mathcal{E})$ and scan$(E^+)$;
20: return $G(V(G), E^+)$;
```

the graph shown in Figure 6 (a), which is the result graph after the vertex reduction shown in Example 5.4, Assume that we conduct the edge reduction following the order (v_6, v_1), (v_6, v_4), (v_6, v_5). Ideally, $(v_6, v_1)/(v_6, v_4)$ are reduced due to (v_6, v_4) and $(v_4, v_1)/(v_6, v_5)$ and (v_5, v_4), and (v_6, v_5) is kept to guarantee the LCR preservability. However, in the I/O efficient environment, it is quite possible that these three edges are not processed in memory as a whole. In this case, (v_6, v_5) may also be reduced since the reduction of $(v_6, v_1)/(v_6, v_4)$ is unknown when processing (v_6, v_5). As a result, the LCR information between v_6 and $v_1/v_4/v_5$ is lost. To address this problem, we first define:

Definition 5.8: (Seed Vertex Set Ego-Subgraph) Given a graph G, let S be a set of vertices in G, the ego vertex set of S, denoted by S^*, is defined as $S^* = S \cup \{u | u \in \{\mathsf{nbr}^-(u) \cup \mathsf{nbr}^+(u)\}\}$. The vertex set ego-subgraph, denoted by G_{S^*}, is defined as the induced subgraph in G by S^*. □

According to Definition 5.8, we have the following lemma:

Lemma 5.1: *Given a set of vertices S in G, let (u,v) be an edge in G with at least one incident vertex in S, if there exist two edges (u,w) and (w,v) in G such that $\forall \phi \in \Phi((u,v),G)$, $\exists\, \phi' \in \Phi((u,w),G) \oplus \Phi((w,v),G)$ with $\phi' \subseteq \phi$, then there exist two edges (u,w) and (w,v) in G_{S^*} such that $\forall \phi \in \Phi((u,v),G_{S^*})$, $\exists\, \phi' \in \Phi((u,w),G_{S^*}) \oplus \Phi((w,v),G_{S^*})$ with $\phi' \subseteq \phi$.*

Proof: This lemma can be directed approved by Definition 5.5 and Definition 5.8. □

Algorithm. According to Lemma 5.1 and Definition 5.5, it is clear that for the edges with at least one incident vertex in S, we can conduct the edge reductions for these edges on G_{S^*} instead of G. Therefore, we can partition the vertices of G into disjoint subsets whose ego-subgraph can be loaded in the main memory and perform the edge reduction in the main memory accordingly. Following this idea, our I/O efficient edge reduction algorithm is shown in Algorithm 4.

Specifically, it first sorts the edges (u, v) in the given graph G based on $\{\mathsf{id}(u), \mathsf{id}(v)\}$ (line 1) and $\{\mathsf{id}(v), \mathsf{id}(u)\}$ (line 2), which can be used to obtain the out-neighbors and in-neighbors of a vertex easily. After that, it initializes S, S^*, and $\mathcal{E}$ as empty (line 3). Then, it continuously adds a new vertex v in S, and its neighbors in S^* based on the size of G_{S^*} and the available main memory M (line 4-6). In line 6, we use the average degree $\overline{\deg}$ of vertices in G to estimate the size of G_{S^*} and parameter $0 < c < 1$ is used to ensure the estimated G_{S^*} size is less than M. For the case that S^* leads to $|G_{S^*}| > M$, we can further split S^* into smaller sets and construct G_{S^*} accordingly. G_{S^*} can be obtained by $\mathsf{scan}(E^+)$ and $\mathsf{scan}(E^-)$ (line 7). Currently, G_{S^*} is kept in main memory, and we only store the out-neighbors in G_{S^*} to fully utilize the available memory. For each vertex u in G_{G^*}, it iterates each out-neighbor v of u (line 9), and if w is the out-neighbor of both u and v (line 10) while for each $\phi \in \Phi((u, v), G_{S^*})$, there exists $\phi' \in \Phi((u, v), G_{S^*}) \oplus \Phi((v, w), G_{S^*})$ such that $\phi' \subseteq \phi$ (line 12-13), it means (u, v) can be reduced based on Definition 5.5. Therefore, (u, v) is marked as removed in G_{S^*} and added into $\mathcal{E}$ (line 15-16). When all the edges in G have been loaded in the main memory and processed once (line 4), the edges stored in $\mathcal{E}$ are sorted (line 18) and removed from E^+ (line 19). The reduced graph is returned in line 20.

Example 5.5: Figure 6 shows the procedure of Algorithm 4 in the iteration 1 during the index construction. Assume that the main memory is large enough to store 7 edges. Algorithm 4 first selects $\{v_1, v_4, v_5, v_6\}$ as S, which is illustrated in shadow. The vertex set ego-subgraph G_{S^*} of S is the subgraph induced by the solid lines in Figure 6 (b). It conducts the edge reduction for the edges (v_1, v_5), (v_4, v_1), (v_5, v_4), (v_6, v_1), (v_6, v_4), (v_6, v_5), (v_6, v_1) and (v_6, v_4) are reduced, which are shown in dashed line. It continues to select (v_7, v_{10}, v_{11}) as S and no more edges can be reduced. Figure 6 (c) shows the final edge reduction result of iteration 1. □

Lemma 5.2: *Give two vertices u and v in a graph G, let G' be the output graph of Algorithm 4, for a set of edge labels $\Delta \subseteq \Sigma$, if there is an path p from u to v in G with $\ell(p) \subseteq \Delta$, then, there is a path p' from u to v in G' such that $\ell(p') \subseteq \ell(p) \subseteq \Delta$.*

Optimization. In Algorithm 4, we perform the reduction for each edge in every iteration. However, except the first iteration, performing the reduction for every edge in the remaining iterations involves lots of unnecessary computations. Reconsider the procedure of Algorithm 2 and Definition 5.5, an edge (u, v) can be reduced possibly if and only if the in-neighbors/out-neighbors of u/v are changed due to the vertex reduction. In other words, when conducting the edge reduction, we do not need to consider these edges (u, v) that the in-neighbors/out-neighbors of u/v are not changed after the vertex reduction. Following this idea, instead of conducting the edge reduction based on the output of Algorithm 3, we can record the newly added or updated edges in line 14 of Algorithm 3 and denote them as E_{new}. Based on the above discussion, only the edges in E_{new} could lead to the reduction of edges. Therefore, we take the subgraph induced by the vertices incident to E_{new} and their neighbors as the input of Algorithm 4, and we can not only guarantee the correctness of the edge reduction but also significantly improve the edge reduction performance.

Algorithm 5 IOLCRIndexCons(G)

```
1: $i \leftarrow 0$; $G_i \leftarrow G$;
2: while $G_i$ cannot be loaded in main memory do
3:   $I_i \leftarrow$ IOMIS($G_i$);
4:   $G' \leftarrow$ IOVertexReduction($G'$);
5:   $G' \leftarrow$ IOEdgeReduction($G'$);
6:   $G_{i+1} \leftarrow G'$; $i \leftarrow i + 1$; $k \leftarrow i$;
```

5.2.3 I/O Efficient Index Construction.

Following the above procedures, the I/O efficient algorithm to construct the index is straightforward, which is shown in Algorithm 5. Starting from $i = 0$, the algorithm iteratively computes the maximal independent set of G_i and applies the vertex reduction and edge reduction on G_i. The construction terminates when G_k can fit in the main memory. The correctness of Algorithm 5 can be directly obtained based on Theorem 5.1 and Lemma 5.2. Moreover, we have:

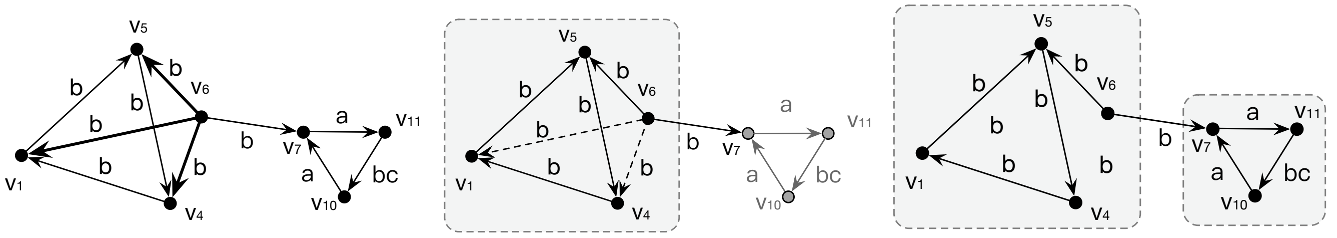
Theorem 5.2: *Given a graph G, the I/O complexity of Algorithm 5 is $O(\Sigma_{i=0}^{k}(\mathsf{sort}(G_i) + \kappa_i \cdot \mathsf{scan}(G_i)))$, where κ_i is the number of seed vertex sets generated by Algorithm 4 for G_i.*

Proof: To construct the index, Algorithm 5 generates a series of LCR preserved graphs $G_0, G_1, \ldots, G_k$. For each LCR-PG G_i, the I/O complexity of maximal independent set computation is $O(\mathsf{sort}(G_i))$ as shown in [30]. For vertex reduction, Algorithm 3 needs to sort the edges of G_i first and then scans the edges of G_i to conduct the vertex reduction, which takes $O(\mathsf{sort}(G_i))$ I/Os. For edge reduction, Algorithm 4 needs to scan G_i once to obtain the seed vertex set ego-subgraph for a seed vertex set, and thus it needs $O(\kappa_i \cdot \mathsf{scan}(G_i))$ I/Os to conduct the edge reduction for G_i. Therefore, the total I/Os to construct the index is $O(\Sigma_{i=0}^{k}(\mathsf{sort}(G_i) + \kappa_i \cdot \mathsf{scan}(G_i)))$.

Remark. In Algorithm 5, we conduct vertex reduction before edge reduction. However, we can also conduct edge reduction before vertex reduction as well. For a specified reduction round, conducting vertex reduction before edge reduction can reduce more edges generated by vertex reduction while conducting edge reduction before vertex reduction can lead to a simplified input graph for vertex reduction, which means both reduction orders have their own advantages. On the other hand, following Algorithm 5, the reduction can be conducted in several rounds, and thus the performance difference is leveled out if the reduction procedure is considered as a whole. Therefore, the reduction orders of vertex and edge have little effect on the effectiveness of our proposed algorithm. Here, we follow the reduction order of vertex and edge for convention. For Theorem 5.2, $O(\mathsf{scan}(G_i)) = O(\frac{|E(G_i)|}{B})$, where B is the block transfer size between main memory and disk and current operating systems generally use 8KB as default size [46]. Consider κ_i is usually not very large and it only involves sequential reads, $O(\kappa_i \cdot \mathsf{scan}(G_i))$ is better than $O(m)$. Moreover, as discussed in Section 4, the general framework of Algorithm 5 can also be adjusted to optimize other graph algorithms to handle the scenario that main memory is limited if I/O efficient reduction operators that guarantee the reducibility and algorithm-specific property preservability can be designed.

5.3 Index Structure on Disk

The index construction algorithm generates a series of LCR preserved graphs as shown in Algorithm 5. According to Theorem 5.1,



(a) Challenges of Edge Reduction (b) Seed Vertex Set Ego-Subgraph (c) Edge Reduction Result

Figure 6: I/O Efficient Edge Reduction

we can answer a given LCR query $q(s,t,\Delta)$ based on $G_{\min\{o(s),o(t)\}}$, where $o(s)/o(t)$ is the maximal independent set order of s/t following Definition 5.7. However, directly storing these graphs on the disk as an index is insufficient for an I/O efficient LCR query processing as this approach only transforms the problem from query processing on G to query processing on $G_{\min\{o(s),o(t)\}}$ and in worst case $G_{\min\{o(s),o(t)\}}$ could be G. To address this problem, we devise a new index in which the edges in the generated LCR preserved graphs are reorganized to facilitate the I/O efficient query processing. Specifically, it contains two parts:

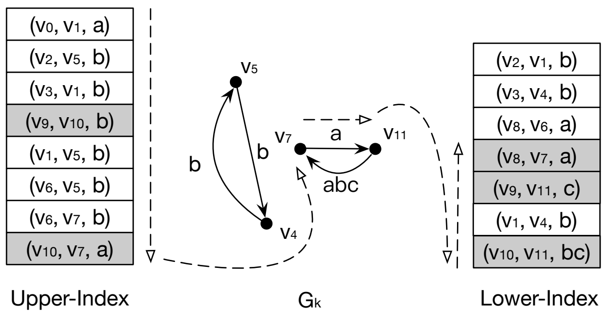
- Upper-Index: Upper-Index stores the vertices $v \in \{V(G) \setminus V(G_k)\}$ following the maximal independent set order. Moreover, it also stores the out-neighbors v' of v in $G_{o(v)}$ and the corresponding LCR dominating label set $\Phi((v,v'),G_{o(v)})$.
- Lower-Index: Lower-Index stores the vertices in $v \in \{V(G) \setminus V(G_k)\}$ following the maximal independent set order. Moreover, it also stores the in-neighbors v' of v in $G_{o(v)}$ and the corresponding LCR dominating label set $\Phi((v',v),G_{o(v)})$.

Example 5.6: Following the index construction procedure illustrated in Figure 4, Figure 7 shows the corresponding Upper-Index and Lower-Index of G. Take Lower-Index as an example, the maximal independent set order of v_0, v_2, v_3, v_8, v_9 is 0, and that of v_1, v_6, v_{10} is 1. Therefore, Lower-Index stores the in-neighbors of v_2, v_3, v_8, v_9 and the labels in the corresponding LCR dominating label set, namely (v_2,v_1,b), (v_3,v_4,b), (v_8,v_6,a), (v_8,v_7,a), (v_9,v_{11},c) (v_0 has no in-neighbors in G_0), in G_0 sequentially. Following that, (v_1,v_4,b), (v_{10},v_{11},bc) in G_1 (v_6 has no in-neighbors in G_1) are stored for the same reason. The Upper-Index is processed in the same way. □

Following the index construction procedure presented in Section 5.2, it is clear that Lower-Index and Upper-Index can be obtained by line 6 and line 8 of Algorithm 3. Therefore, it does not affect the time complexity analysis in Theorem 5.2.

5.4 Query Processing

For the ease of presentation, we denote Lower-Index and Upper-Index as $\mathcal{I}_l$ and $\mathcal{I}_u$, respectively, and call G_k, $\mathcal{I}_l$, and $\mathcal{I}_u$ together as the LCR-Index afterwards. With LCR-Index, the simplest case for the LCR query $q(s,t,\Delta)$ is $s,t \in V(G_k)$. In this case, it is clear that t is reachable from s regarding Δ in G if and only if t is reachable from s regarding Δ in G_k following the LCR preservability of G_k as proved in Theorem 5.1, therefore, we can answer the query by a BFS search on G_k following the edges with label in Δ directly. As this case does not involve I/Os when processing the query, it is the



Upper-Index
(v0, v1, a)
(v2, v5, b)
(v3, v1, b)
(v9, v10, b)
(v1, v5, b)
(v6, v5, b)
(v6, v7, b)
(v10, v7, a)

Lower-Index
(v2, v1, b)
(v3, v4, b)
(v8, v6, a)
(v8, v7, a)
(v9, v11, c)
(v1, v4, b)
(v10, v11, bc)

Figure 7: Lower-Index, Upper-Index **on Disk and** G_k **in Memory**

most efficient case. When s and t are not in G_k, we need to leverage the Lower-Index and Upper-Index to answer the query. Following the structure of Lower-Index and Upper-Index, we have:

Lemma 5.3: *Given a graph G and its corresponding* Lower-Index *$\mathcal{I}_l$/*Upper-Index *$\mathcal{I}_u$, for two vertices $u,v \in \mathcal{I}_l/\mathcal{I}_u$, if u/v precedes v/u in $\mathcal{I}_l/\mathcal{I}_u$, then $o(u) \le o(v)$.*

Proof: This lemma can be directly proved following the definition of $\mathcal{I}_l$ and $\mathcal{I}_u$. □

Lemma 5.4: *Given a graph G and its corresponding* Lower-Index *$\mathcal{I}_l$/*Upper-Index *$\mathcal{I}_u$, for each edge $(u,v) \in \mathcal{I}_l/\mathcal{I}_u$, $o(u) \le o(v)$.*
Proof: This lemma can be proved similarly as Lemma 5.3. □

According to Lemma 5.3 and Lemma 5.4, the vertices and edges are stored in order based on their maximal independent set order, which implies any graph G_i ($0 \le i < k$) can be easily obtained by scanning $\mathcal{I}_l$ and $\mathcal{I}_u$ sequentially. Considering the LCR preservability of G_i, it means we can answer the given LCR query via sequential scan of $\mathcal{I}_l$ and $\mathcal{I}_u$ without any random I/Os (Note that $\mathcal{I}_l/\mathcal{I}_u$ stores the in-neighbors/out-neighbors of the vertices, respectively, therefore, both $\mathcal{I}_l$ and $\mathcal{I}_u$ are needed).

Algorithm. Following the above idea, our I/O efficient query processing algorithm, IOQuery, is shown in Algorithm 6. Given a LCR query $q(s,t,\Delta)$, it maintains a hash table $\mathcal{H}$ to record the LCR reachable from s regarding Δ (line 1). It first scans the items in $\mathcal{I}_u$ until s is encountered (line 2) and pushes the out-neighbors u of s such that there is a $\phi \in \Phi(s,u)$ with $\phi \subseteq \Delta$ in $\mathcal{H}$ (line 3-5). Then, it continues the sequential scan of $\mathcal{I}_u$, and for each edge (u,v) encountered during the scan, if u has been pushed in $\mathcal{H}$ and there is a $\phi \in \Phi(u,v)$ with $\phi \subseteq \Delta$ (line 6-7), if v is the query vertex t, then t is LCR reachable from s regarding Δ and true is returned (line 8). Otherwise, v is also pushed in $\mathcal{H}$ (line 9). When all the edges in $\mathcal{I}_u$ has ben explored, it initializes a empty queue Q in line 10 and pushes the vertices in $\mathcal{H}$ in Q (line 11-12). Since G_k can be loaded

Algorithm 6 IOQuery$(s, t, \Delta, G_k, \mathcal{I}_l, \mathcal{I}_u)$

```
1: $\mathcal{H} \leftarrow \emptyset$;
2: scan the edges in $\mathcal{I}_u$ until encountering $s$;
3: for each edge $(s, u) \in \mathcal{I}_u$ do
4:   if $\exists\, \phi \in \Phi(s, u)$ s.t. $\phi \subseteq \Delta$ then
5:     if $u \notin \mathcal{H}$ then push $u$ into $\mathcal{H}$;
6: for each edge $(u, v) \in \mathcal{I}_u$ by scanning $\mathcal{I}_u$ do
7:   if $u \in \mathcal{H}$ and $\exists\, \phi \in \Phi(u, v)$ s.t. $\phi \subseteq \Delta$ then
8:     if $v = t$ then return true;
9:     if $v \notin \mathcal{H}$ then push $v$ into $\mathcal{H}$;
10: $Q \leftarrow \emptyset$;
11: for each $v \in \mathcal{H}$ do
12:   push $v$ in $Q$;
13: while $Q \neq \emptyset$ do
14:   pop $v$ from $Q$;
15:   for each out-neighbor $w$ of $v$ in $G_k$ do
16:     if $\exists\, \phi \in \Phi(v, w)$ s.t. $\phi \subseteq \Delta$ then
17:       if $v = t$ then return true;
18:       if $w \notin \mathcal{H}$ then push $w$ into $\mathcal{H}$; push $w$ into $Q$;
19: for each edge $(u, v)$ in the reverse order in $\mathcal{I}_l$ do
20:   if $v \in \mathcal{H}$ and $\exists\, \phi \in \Phi(u, v)$ s.t. $\phi \subseteq \Delta$ then
21:     if $u = t$ then return true;
22:     if $u \notin \mathcal{H}$ then push $u$ into $\mathcal{H}$;
23: return false;
```

in the main memory, it just conducts a BFS manner search starting the vertices in $\mathcal{H}$ via the edges (v, w) in G_k that has $\phi \in \Phi(v, w)$ with $\phi \subseteq \Delta$ (line 13-18). If t is reached during the search of G_k, it returns true in line 17. Otherwise, it scans $\mathcal{I}_l$ in reverse following the vertices in $\mathcal{H}$ in a similar manner as that in $\mathcal{I}_u$ (line 19-23). If t is encountered, it returns true (line 21). Otherwise, it returns false in line 23.

Example 5.7: Consider the LCR query $q(v_9, v_8, \{a, b\})$, Figure 7 also shows the procedures of Algorithm 6 to answer the query, which are illustrated by the dashed line arrow. Specifically, it first scans the Upper-Index $\mathcal{I}_u$ and finds v_9 is located in the fourth item of $\mathcal{I}_u$. As $\Phi(v_9, v_{10}) = \{b\} \subseteq \{a, b\}$, v_{10} is pushed into $\mathcal{H}$. It continues to scan $\mathcal{I}_u$ and encounters (v_{10}, v_7) with $\Phi(v_{10}, v_7) = \{a\} \subseteq \{a, b\}$, then v_7 is added into $\mathcal{H}$. After that, it conducts the BFS search from v_7 on G_k and encounters v_{11} with $\Phi(v_7, v_{11}) = \{a\} \subseteq \{a, b\}$, thus v_{11} is pushed in $\mathcal{H}$. After that, it scans the Lower-Index $\mathcal{I}_l$. It first encounters v_{10} and $\Phi(v_{10}, v_{11}) = \{bc\} \nsubseteq \{a, b\}$. Then, it encounters v_9 with $\Phi(v_9, v_{11}) = \{c\} \nsubseteq \{a, b\}$. At last, it encounters v_8, and $\Phi(v_8, v_7) = a \subseteq \{a\}$. As v_8 is the target query vertex of q, it means v_8 is LCR reachable from v_9 regarding $\{a, b\}$, and true is returned. □

Theorem 5.3: *Given a* LCR *query $q(s, t, \Delta)$, Algorithm 6 answers q correctly.*

Theorem 5.4: *Given a* LCR *query $q(s, t, \Delta)$, Algorithm 6 answers q in $O(\text{scan}(|\mathcal{I}_l| + |\mathcal{I}_u|))$ I/Os.*

Proof: Following the procedures of Algorithm 6, to answer q, it only sequentially scans the Upper-Index and Lower-Index once and no other I/Os are needed. Therefore, the I/O complexity of Algorithm 6 is $O(\text{scan}(|\mathcal{I}_l| + |\mathcal{I}_u|))$. The theorem holds. □

Remark. Based on Algorithm 6, we also keep the vertices encountered during the query processing in memory, which means we need extra memory besides G_k. However, even in worst case, the number of such vertices can be bounded by $O(n)$, which is significantly smaller than the number of edges in the graph. Actually, the number of such vertices is fewer in practice as verified in our experiment. Therefore, we can construct a G_k smaller than the available memory by adjusting the value of c in line 6 Algorithm 5 and reserve some memory for Q and $\mathcal{H}$ for query processing.

6 EXPERIMENT

In this section, we compare our algorithms with the state-of-the-art I/O efficient methods for label-constrained reachability queries. All experiments are conducted on a machine with an Intel Xeon, and 8 GB main memory running Linux.

Table 1: Datasets used in Experiments

Dataset	Name	$\|V\|$	$\|E\|$	d_{max}	d_{avg}	$\|\mathcal{I}_u\| + \|\mathcal{I}_l\|$
DBLP	DB	326,186	1,615,400	238	5.0	129MB
Google	GO	875,713	5,105,039	6,332	5.8	237MB
UK-2005	UK	39,459,925	936,364,282	1,776,852	23.7	5.1GB
IT-2004	IT	41,291,594	1,150,725,436	1,326,745	27.9	5.7GB
Twitter	TW	41,652,230	1,468,365,182	2,997,487	70.5	8.7GB
SK-2005	SK	50,636,154	1,949,412,601	8,563,808	38.5	10.8GB
Wikidata	WK	113,782,967	1,376,148,923	96,140,796	12.1	8.1GB
YAGO	YG	177,319,207	1,472,473,745	79,168,293	8.3	8.9GB

Datasets. We evaluate our algorithms on eight real-world graphs, which are demonstrated in Table 1. GO is downloaded from SNAP (http://snap.stanford.edu/data/index.html). DB, UK, IT, TW and SK are downloaded from LAW (http://law.di.unimi.it/datasets.php). YG is downloaded from https://yago-knowledge.org/downloads/yago-4-5 and WK is downloaded from https://www.wikidata.org/wiki/Wikidata:Database_download. Among the graphs, YG and WK have natural edge labels, and for rest of graphs without edge labels, we generate label for each edge following the approach used in [37, 44] with $|\Sigma| = 8$.

Algorithms. We compare the following four algorithms:

- EM-BFS: external-memory BFS based algorithm [23] where the vertices and edges cannot fit in the main memory and only the edges with label in Δ are explored on the disk during the search.
- SEM-BFS: semi-external memory BFS based algorithm [63] in which the vertices are kept in the main memory and the search is performed by scanning the edges with label in Δ on the disk.
- P2H+: the state-of-the-art index-based in-memory LCR query processing algorithm [37].
- LCR-Index: our proposed I/O efficient algorithm (Algorithm 6).

All the algorithms are implemented in C++ and complied in GCC 8.3.1 with -O3 flag. In the experiments, we set the default memory limit as 2GB and default $|\Delta| = 4$. For LCR-Index, we keep running the index construction algorithm (Algorithm 5) until G_k can fit inside the available memory. The time cost is measured as the amount of wall-clock time elapsed during the program's execution. If a query cannot be processed in 10,000 seconds, we denote the processing time as OT.

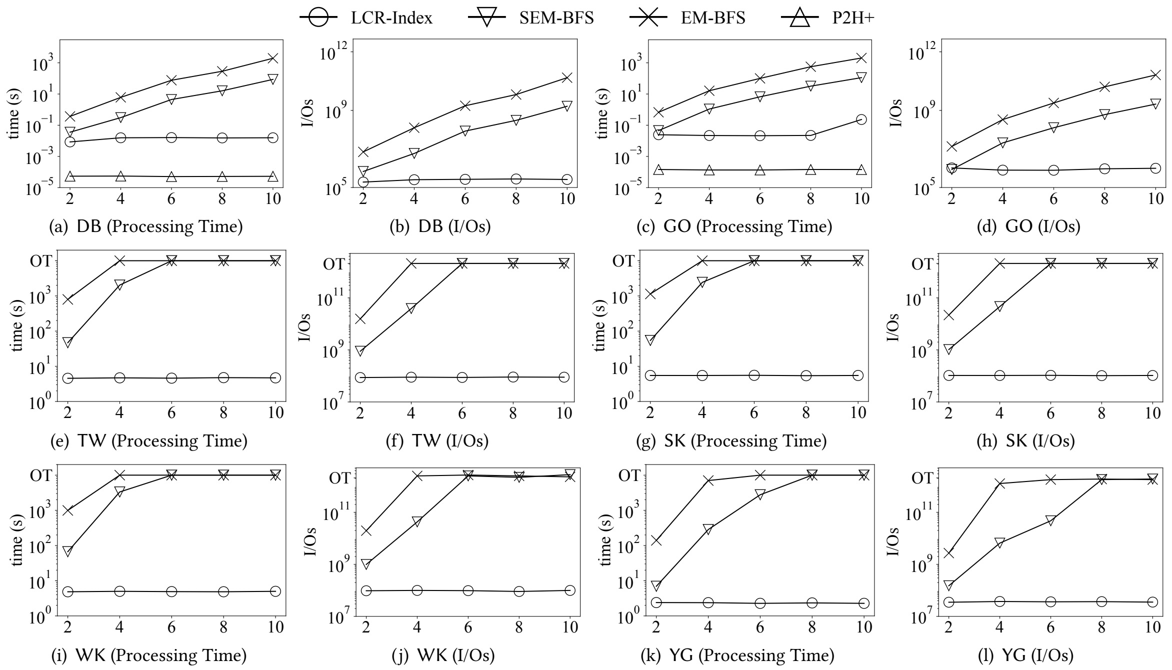


Figure 8: Processing time and I/Os when varying query vertices distance

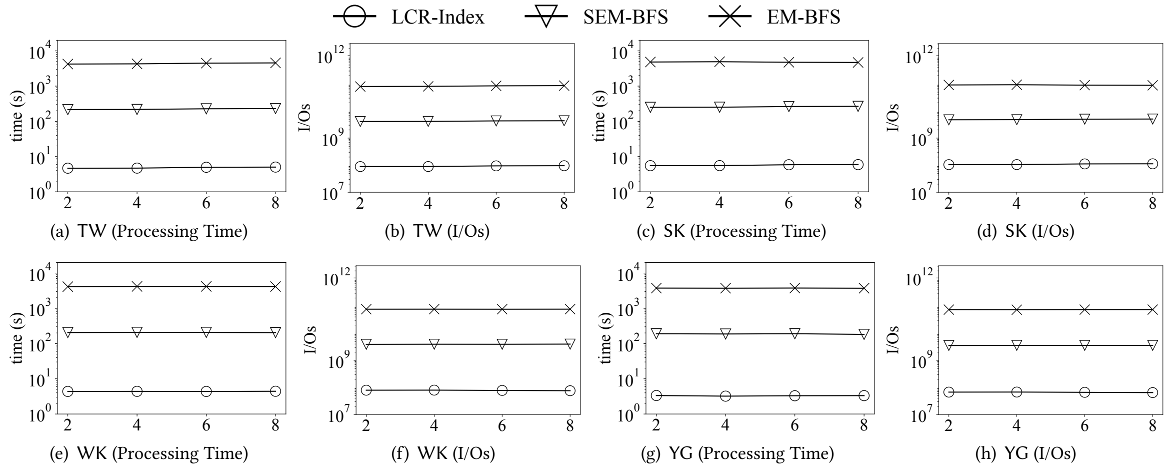


Figure 9: Processing time and I/Os when varying $|\Delta|$

Exp-1: Efficiency when varying query vertex distance. In this experiment, we evaluate the query performance by varying the distance between two vertices in the query. For each dataset, we randomly generate 5 groups of queries and each group contains 100 LCR queries in which the distance between two query vertices are from 2 to 10. We report the average time and I/Os to process each query and Figure 8 shows the results. Note that P2H+ works entirely in memory, thus, there is no I/Os associated with P2H+.

As shown in Figure 8, for the three algorithms designed to handle the graphs that cannot be kept in the main memory, namely, EM-BFS, SEM-BFS, and LCR-Index, LCR-Index always outperforms the baseline algorithms EM-BFS and SEM-BFS, and the performance gap increases as the distance grows. This is because as the query distance increases, the search space to answer the query increases. As a result, the I/Os involved in EM-BFS and SEM-BFS increases. However, due to the introduction of the index structure, LCR-Index can significantly reduce such I/Os, which is also verified by the reported I/Os in Figure 8. Moreover, on the billion-scale datasets, both baseline algorithms run out of time when the query distance becomes large, while LCR-Index can always answer the queries efficiently, which is consistent with the theoretical analysis shown in Theorem 5.4. Regarding P2H+, which stores the entire graph in memory, it demonstrates exceptional performance as long as the graph is small enough to fit within the available memory. However, when the graph size increases (i.e., on all graphs except DB and GO), it fails to compute the result because it exhausts the available memory (Thus, the results on these datasets are not shown in Figure 8). Based on the results, it is clear our algorithm is not only

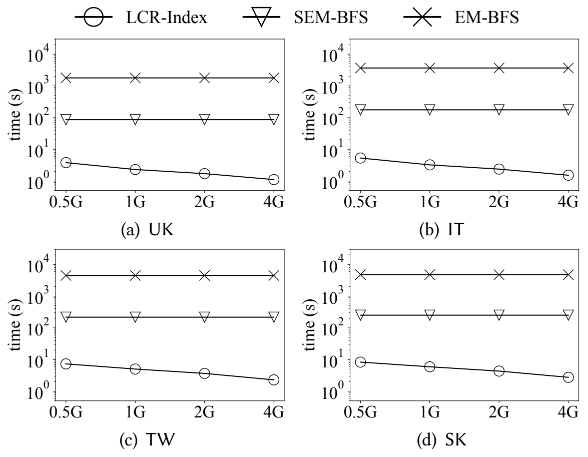


Figure 10: Processing time when varying M

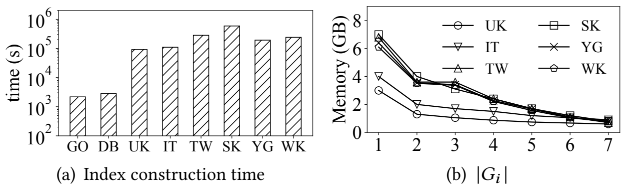


Figure 11: Index construction evaluation

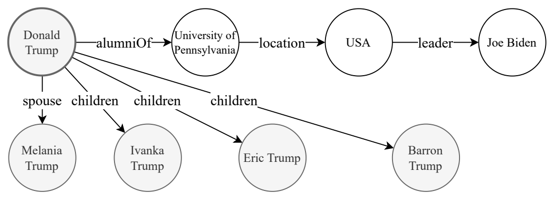


Figure 12: Case study

I/O efficient in answering the LCR queries but is also able to handle billion-scale graphs with limited memory resources.

Exp-2: Efficiency when varying $|\Delta|$. In this experiment, we evaluate the query efficiency by varying $|\Delta|$ of queries. To do this, we randomly generate 4 groups of queries and each group has 100 queries. For each query $q = (s, t, \Delta)$ in group i, the size of Δ is $2i$, and s can reach t following the edges with label in Δ. The average processing time and I/Os for each query is shown in Figure 9.

As shown in Figure 9, it is clear that LCR-Index always outperforms EM-BFS and SEM-BFS. The reasons are the same as discussed in Exp-1. Moreover, when the size of Δ increases, the processing time for all evaluated algorithms remains relatively constant. This is because a larger $|\Delta|$ leads to an increase in the number of potential vertices explored during the query process. However, this is offset by a corresponding decrease in the path length to reach the target vertex from the source, resulting in a quicker path discovery. As a result, the overall processing time remains stable.

Exp-3: Efficiency when varying M. In this experiment, we evaluate the efficiency of the query processing algorithms as the available memory M increases from 0.5 GB to 4 GB on four large datasets. For LCR-Index, we build the index for each dataset and process the queries under these varying memory constraints. We randomly generate 100 queries and report the average processing time for each query in Figure 10. Due to the limited space, I/Os are not shown afterwards, but the trends are similar to Exp-1 and Exp-2.

As shown in Figure 10, when M increases, the processing time of LCR-Index decreases. This improvement is attributed to the increased capacity of memory to accommodate a larger size of G_k, thus reducing the I/Os associated with accessing data from $\mathcal{I}_u$ and $\mathcal{I}_l$. In contrast, the performance of SEM-BFS and EM-BFS algorithms remains relatively consistent, regardless of the increased memory size. This is primarily because, even at a memory size of 4 GB, the sheer scale of the billion-scale graphs prevents them from being fully accommodated in the memory. Moreover, these two methods lack the systematic ability to fully use the available memory. Consequently, these algorithms still incur a large number of I/Os, leading to relatively stable processing time.

Exp-4: Index construction evaluation. In this experiment, we evaluate the time to construct the index on all datasets and report the size of G_i ($1 \leq i \leq 7$) as the index construction progresses on four large datasets to show the effectiveness of our proposed reduction techniques. The results are shown in Figure 11. Moreover, the size of $|\mathcal{I}_u| + |\mathcal{I}_l|$ is also reported in Table 1.

As shown in Figure 11 (a), the index construction time increases as the size of the input graph increases. This is because the larger the input graph is, the more I/Os are required to construct the index. For the size of G_i, it is clear that the size of the input graph is significantly reduced as shown in Figure 11 (b). Moreover, Figure 11 (b) shows that the size of G_i decreases sharply in the first three iterations. After that, the size of G_i decreases stably. This is because at first, lots of vertices are in the computed maximal independent set. However, as the construction progresses, the newly generated G_is become dense, and the size of computed maximal independent set decreases consequently. As a result, fewer vertices and edges can be reduced in the later iterations. Regarding the index size, Table 1 shows that the index structure stored on the disk is comparatively not large considering the graph size due to our reduction methods.

Exp-5: Case study on social networks analysis with LCR queries. Figure 12 presents a real-world example of LCR queries on social network analysis. In Figure 12, part of "Donal Trump"'s social network is shown. Assume that we want to find the relatives of "Donald Trump", if we do not consider the labels on the edges, "Joe Biden" could be possible misidentified as a relative of "Donal Trump" due to their reachability through path "Donald Trump" -[alumniOf]->"University of Pennsylvania"-[location]->"USA"-[leader]->"Joe Biden". On the other hand, if we use LCR queries with Δ = {children, spouse}, we can avoid the above issue and obtain the correct result. Based on this case study, it is clear that LCR queries provide the users with the capacity for fine-grained precise social network analysis.

7 CONCLUSION

In this paper, we study the I/O efficient LCR query problem that aims to answer the LCR query efficiently when the graphs are too big to reside in memory. We propose a reduction-based indexing approach and devise an index named LCR-Index. Based on the LCR-Index, we propose an efficient query processing algorithm with a theoretical I/O bound. We conduct experiments on real-world datasets and the results demonstrate the efficiency of our proposed algorithms.

REFERENCES

[1] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory bfs implementations. In *Proceedings of ALENEX*, pages 3–12, 2007.

[3] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *Proc. VLDB Endow.*, 11(2):149–161, 2017.

[4] A. L. Buchsbaum, M. H. Goldwasser, and S. Venkatasubramanian. On external memory graph traversal. In *Proceedings of SODA*, pages 859–860, 2000.

[5] L. Chen, Y. Gao, Y. Zhang, C. S. Jensen, and B. Zheng. Efficient and incremental clustering algorithms on star-schema heterogeneous graphs. In *Proceedings ICDE*, pages 256–267, 2019.

[6] X. Chen, Y. Peng, S. Wang, and J. X. Yu. DLCR: efficient indexing for label-constrained reachability queries on large dynamic graphs. *Proc. VLDB Endow.*, 15(8):1645–1657, 2022.

[7] Z. Chen, B. Feng, L. Yuan, X. Lin, and L. Wang. Fully dynamic contraction hierarchies with label restrictions on road networks. *Data Science and Engineering*, 8(3):263–278, 2023.

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

[9] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *Proceedings of ICDE*, pages 51–62, 2011.

[10] J. Cheng, Y. Ke, A. W.-C. Fu, J. X. Yu, and L. Zhu. Finding maximal cliques in massive networks. *TODS*, 36(4):1–34, 2011.

[11] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *Proceedings of EDBT*, pages 961–979, 2006.

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. 1995.

[13] P. Choudhary and U. Singh. A survey on social network analysis for counter-terrorism. *International Journal of Computer Applications*, 112(9):24–29, 2015.

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

[15] Y. Gao, T. Zhang, L. Qiu, Q. Linghu, and G. Chen. Time-respecting flow graph pattern matching on temporal graphs. *IEEE TKDE*, 33(10):3453–3467, 2020.

[16] M. Halldórsson and J. Radhakrishnan. Greed is good: Approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of the STOC*, pages 439–448, 1994.

[17] J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of FOCS*, pages 627–636, 1996.

[18] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *Proceedings of SIGMOD*, pages 325–336, 2013.

[19] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *Proceedings of SIGMOD*, pages 123–134, 2010.

[20] R. Jin, N. Ruan, S. Dey, and J. X. Yu. SCARAB: scaling reachability computation on large graphs. In *Proceedings of SIGMOD*, pages 169–180, 2012.

[21] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.

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

[23] I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84, 2002.

[24] H. Kim, J. Lee, S. S. Bhowmick, W. Han, J. Lee, S. Ko, and M. H. A. Jarrah. DUALSIM: parallel subgraph enumeration in a massive graph on a single machine. In *Proceedings of SIGMOD*, pages 1231–1245, 2016.

[25] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of SPDP*, pages 169–176, 1996.

[26] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *Proceedings of ICDE*, pages 74–85, 2012.

[27] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation in bipartite graphs. *The VLDB Journal*, 29(5):1075–1099, 2020.

[28] Y. Liu, J. Lu, H. Yang, X. Xiao, and Z. Wei. Towards maximum independent sets on massive graphs. *PVLDB*, 8(13):2122–2133, 2015.

[29] C. Ma, J. Li, K. Wei, B. Liu, M. Ding, L. Yuan, Z. Han, and H. V. Poor. Trusted ai in multiagent systems: An overview of privacy and security for distributed learning. *Proceedings of the IEEE*, 111(9):1097–1132, 2023.

[30] A. Maheshwari and N. Zeh. I/o-efficient algorithms for graphs of bounded treewidth. *Algorithmica*, 54(3):413–469, 2009.

[31] W. Martens and T. Trautner. Evaluation and enumeration problems for regular path queries. In *Proceedings of ICDT*, volume 98, pages 19:1–19:21, 2018.

[32] L. Meng, Y. Shao, L. Yuan, L. Lai, P. Cheng, X. Li, W. Yu, W. Zhang, X. Lin, and J. Zhou. A survey of distributed graph algorithms on massive graphs. *CoRR*, abs/2404.06037, 2024.

[33] L. Meng, L. Yuan, Z. Chen, X. Lin, and S. Yang. Index-based structural clustering on directed graphs. In *Proceedings ICDE*, pages 2831–2844, 2022.

[34] U. Meyer and P. Sanders. *Algorithms for memory hierarchies: advanced lectures*, volume 2625. Springer Science & Business Media, 2003.

[35] A. Mukkara, N. Beckmann, and D. Sanchez. Cache-guided scheduling: Exploiting caches to maximize locality in graph processing. *AGP'17*, 2017.

[36] A. Pacaci, A. Bonifati, and M. T. Özsu. Regular path query evaluation on streaming graphs. In *Proceedings SIGMOD*, pages 1415–1430, 2020.

[37] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang. Answering billion-scale label-constrained reachability queries within microsecond. *PVLDB*, 13(6):812–825, 2020.

[38] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.

[39] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, 2017.

[40] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.*, 29(2-3):595–618, 2020.

[41] R. Schenkel, A. Theobald, and G. Weikum. HOPI: an efficient connection index for complex XML document collections. In *Proceedings of EDBT*, pages 237–255, 2004.

[42] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: flexible and efficient reachability range assignment for graph indexing. In *Proceedings of ICDE*, pages 1009–1020, 2013.

[43] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theory Computer Science*, 58:325–346, 1988.

[44] L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *Proceedings SIGMOD*, pages 345–358, 2017.

[45] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of SIGMOD*, pages 913–924, 2011.

[46] J. S. Vitter. *Algorithms and Data Structures for External Memory.* Now Publishers Inc, 2008.

[47] J. Wang and J. Cheng. Truss decomposition in massive networks. In *Proceedings of VLDB*, pages 812–823, 2012.

[48] K. Wang, M. Cai, X. Chen, X. Lin, W. Zhang, L. Qin, and Y. Zhang. Efficient algorithms for reachability and path queries on temporal bipartite graphs. *The VLDB Journal*, pages 1–28, 2024.

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

[50] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of CIKM*, pages 1601–1606, 2013.

[51] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *VLDB J.*, 21(4):509–534, 2012.

[52] J. X. Yu and J. Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. 2010.

[53] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Diversified top-k clique search. *The VLDB Journal*, 25(2):171–196, 2016.

[54] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. I/O efficient ECC graph decomposition via graph reduction. *PVLDB*, 9(7):516–527, 2016.

[55] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.*, 26(2):275–300, 2017.

[56] J. Zhang, L. Yuan, W. Li, L. Qin, and Y. Zhang. Efficient label-constrained shortest path queries on road networks: A tree decomposition approach. *PVLDB*, 15(3):686–698, 2021.

[57] J. Zhang, L. Yuan, W. Li, L. Qin, Y. Zhang, and W. Zhang. Label-constrained shortest path query processing on road networks. *VLDB J.*, 33(3):569–593, 2024.

[58] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen. Efficient distributed reachability querying of massive temporal graphs. *The VLDB Journal*, 28:871–896, 2019.

[59] Z. Zhang, L. Qin, and J. X. Yu. Contract & expand: I/O efficient sccs computing. In *Proceedings of ICDE*, pages 208–219, 2014.

[60] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/O efficient: computing sccs in massive graphs. *The VLDB Journal*, 24(2):245–270, 2015.

[61] Z. Zhang, J. X. Yu, L. Qin, and Z. Shang. Divide & conquer: I/O efficient depth-first search. In *Proceedings of SIGMOD*, pages 445–458, 2015.

[62] Z. Zhang, J. X. Yu, L. Qin, Q. Zhu, and X. Zhou. I/O cost minimization: reachability queries processing over massive graphs. In *Proceedings of EDBT*, pages 468–479, 2012.

[63] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of FAST*, pages 45–58, 2015.

[64] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Information Systems*, 40:47–66, 2014.