The VLDB Journal (2013) 22:275–294
DOI 10.1007/s00778-012-0288-4

REGULAR PAPER

Computing weight constraint reachability in large networks

Miao Qiao · Hong Cheng · Lu Qin · Jeffrey Xu Yu · Philip S. Yu · Lijun Chang

Received: 11 March 2012 / Revised: 13 July 2012 / Accepted: 23 July 2012 / Published online: 28 August 2012

Abstract Reachability is a fundamental problem on large-scale networks emerging nowadays in various application domains, such as social networks, communication networks, biological networks, road networks, etc. It has been studied extensively. However, little existing work has studied reachability with realistic constraints imposed on graphs with real-valued edge or node weights. In fact, such weights are very common in many real-world networks, for example, the bandwidth of a link in communication networks, the reliability of an interaction between two proteins in PPI networks, and the handling capacity of a warehouse/storage point in a distribution network. In this paper, we formalize a new yet important reachability query in weighted undirected graphs, called *weight constraint reachability* (WCR) query that asks: is there a path between nodes a and b, on which each real-valued edge (or node) weight satisfies a range constraint. We discover an interesting property of WCR, based on which, we design a novel edge-based index structure to answer the WCR query in $O(1)$ time. Furthermore, we consider the case when the index cannot entirely fit in the memory, which can be very common for emerging massive networks. An I/O-efficient index is proposed, which provides constant I/O (precisely four I/Os) query time with $O(|V| \log |V|)$ disk-based index size. Extensive experimental studies on both real and synthetic datasets demonstrate the efficiency and scalability of our solutions in answering the WCR query.

Keywords Weight constraint reachability · Minimum spanning tree · Lowest common ancestor · Vertex coding · I/O-efficient index

M. Qiao · H. Cheng (✉) · L. Qin · J. X. Yu · L. Chang
Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, New Territories, Hong Kong
e-mail: hcheng@se.cuhk.edu.hk

M. Qiao
e-mail: mqiao@se.cuhk.edu.hk

L. Qin
e-mail: lqin@se.cuhk.edu.hk

J. X. Yu
e-mail: yu@se.cuhk.edu.hk

L. Chang
e-mail: ljchang@se.cuhk.edu.hk

P. S. Yu
Department of Computer Science,
University of Illinois at Chicago, Chicago, IL, USA

P. S. Yu
Computer Science Department,
King Abdulaziz University, Jeddah, Saudi Arabia
e-mail: psyu@cs.uic.edu

1 Introduction

Large networks are emerging nowadays in many application domains, such as social networks, communication networks, biological networks, road networks, etc. Among many types of graph queries, graph reachability is an important type of query, which asks whether there exists a path from one node to another in a directed graph. Graph reachability has been studied extensively in the literature [1,4–7,9,12–17,24–28,31–33] and has many potential applications. But most existing algorithms do not consider realistic constraints on graph reachability that are very common and challenging in real-world applications, except a few recent works [9,14,15,32] that consider adding categorical edge label constraint or distance constraint to reachability queries.

In fact, many real-world networks contain real-valued edge or node weights, for example, the bandwidth of a link in communication networks, the reliability of an interaction between two proteins in PPI networks, the handling capacity of a warehouse/storage point in a distribution network, etc. In many real-world applications, the answers to reachability queries are meaningful only if the edge or node weight is also captured in the reported path. Thus, in this paper, we study a new type of reachability query in weighted undirected graphs, called *weight constraint reachability* (WCR) query, which asks: is there a path between nodes a and b, on which each real-valued edge (or node) weight satisfies a range constraint, for example, $\geq x$, $\leq y$, or within $[x, y]$. The WCR query has many real application scenarios. Here we list several application examples.

Communication networks: Transmission of multimedia streams imposes a minimum-bandwidth requirement on all the links on a path to ensure end-to-end Quality-of-Service (QoS) guarantees [21]. A WCR query can find whether there is a feasible path between two nodes in a network, on which each link has a bandwidth $\geq x$, that is, a minimum-bandwidth requirement. The resulting path can support a rate of x bits per second for transmitting a stream, for example, audio or video, with a bandwidth guarantee.

Biological networks: In a PPI network, a node represents a protein, and an edge represents an interaction between two proteins with a real-valued weight to denote the reliability of the interaction. A WCR query can find whether there is a path between two proteins where the reliability of every interaction is $\geq x$. A lot of research has been proposed to find signaling pathways from PPI networks, for example, [2]. However, many false-positive candidates will be generated. The WCR query can be used to prune these false positives.

Phone call networks: From a phone call log, we can construct a phone call network, where a node represents a caller ID and an edge represents a phone call between two callers, labeled with the time stamp when the phone call is made. A WCR query can find whether there is a chain of calls between two callers, each of which is made during a time period $[t_1, t_2]$. This query can be useful for security reasons such as crime detection. Similarly, the WCR query can be applied to social networks for relationship analysis.

The WCR query also applies to node-weighted networks to ensure the weight of each node on the reported path satisfies a constraint. An example is given below.

Distribution networks: If a firm plans to ship its products from a factory to a retailer store located in distant locations, a WCR query can find whether there is a feasible delivery route between these two locations in the distribution network, on which each intermediate warehouse, storage point or distribution center has a proper handling capacity $\geq x$. This query can help facilitate delivery and distribution of the products and improve the operational efficiency of the supply chain.

To the best of our knowledge, there is no existing reachability study on the real-valued weight constraint. In the literature, [14,32] study categorical edge label-constraint reachability (LCR). In addition, regular path query (RPQ) [22], conjunctive RPQ (CRPQ) [10] and reachability query (RQ) [9] have been proposed for full or a subclass of regular expression constraint on categorical edge labels. The real-valued weight constraint we study is very different from those in [9,10,14,22,32] in the following aspects: (1) there is a total order among real-valued weights, but no order among categorical labels. A much more optimized solution can be designed for WCR by exploiting the total-order property; (2) the cardinality of a real-valued weight set is typically much larger than that of a categorical label set. A large weight set can substantially increase the indexing and query complexity of the existing methods. Take the Sampling-Tree method proposed in [14] for LCR query as an example, its index construction time grows exponentially with the number of distinct labels in a graph, and its query time increases linearly with the number of distinct labels. Given a large real-valued weight set, the WCR query can hardly be answered efficiently by directly applying Sampling-Tree. Other works including NP-Hard query RPQ [22], NPC query CRPQ [10], and $O(|V|^2)$ query time RQ [9] face the same problem for handling real-valued weight set.

In this paper, we aim to design efficient algorithms to answer the WCR query with a compact index. We mainly focus on the edge weight constraint and show that the node weight constraint can be easily reduced to the edge weight constraint. Given an undirected graph G and a WCR query, we exploit the cut property of minimum spanning tree (MST) to show that checking whether two nodes are reachable in G w.r.t. a constraint can be transformed to checking such reachability in an MST of G. This property serves as a building brick for designing a novel index, called Edge-Index. It organizes the MST edges hierarchically based on an elegant transformation of MST, so that we can answer a WCR query in $O(1)$ time.

Considering the networks emerging nowadays typically contain hundreds of millions of vertices or even more, the index size of Edge-Index in $O(|\Sigma||V|)$[1] (Σ is the edge weight set and V is the vertex set of the graph) may easily exceed the memory limit. Therefore, to answer the WCR query, we further design an I/O-efficient disk-based index, Balanced-Index, which is constructed by recursively adjusting an MST into a balanced tree. A nice property of the balanced tree is the $\log_2 |V|$ worst-case tree height, which effectively compresses the disk-based index size to $O(|\Sigma||V| \log |V|)$ while guaranteeing to answer the WCR query with exactly four I/Os. Our algorithm Balanced-Index proves to be I/O-efficient and highly scalable. This is a very significant contribution, as all existing algorithms

[1] The $O(|\Sigma||V|)$ space complexity is for handling the general bounded interval constraint $[x, y]$, $x, y \in \mathbb{R}$. For the half-bounded constraint $\geq x$ or $\leq y$, the complexity is $O(|V|)$.

on graph reachability in the literature are limited to main memory-based algorithms.

Our major contributions are summarized as follows.

– We study the weight constraint reachability (WCR) problem in weighted undirected graphs, which has many important applications in real-world networks. To the best of our knowledge, this is the first work to study the WCR problem.
– We design a novel Edge-Index as an efficient memory-based index to answer a WCR query by exploiting the cut property of minimum spanning tree. We also design a Balanced-Index as an I/O-efficient disk-based index, when the index is too large to fit in the memory. Remarkably, our in-memory algorithm Edge-Index achieves $O(1)$ query time and $O(|\Sigma||V|)$ index size, while our I/O-efficient algorithm using Balanced-Index uses four I/Os for query processing and $O(|\Sigma||V|\log|V|)$ disk space for indexing.
– We conducted extensive experiments on large real and synthetic networks. The query time of both Edge-Index and Balanced-Index algorithms is in microseconds and remains stable regardless of the network size, the density, the cardinality of the edge weight set or the weight distribution. Balanced-Index proves to be I/O-efficient and highly scalable for querying large networks. Finally, our query processing is at least three orders of magnitude faster than basic search approaches including DFS, BFS and bi-directional search.

The rest of this paper is organized as follows. Section 2 gives the preliminaries and problem definition. Section 3 introduces an efficient in-memory algorithm Edge-Index to answer the WCR query. Section 4 further proposes an I/O-efficient algorithm Balanced-Index. Section 5 shows our experimental results on large-scale real and synthetic graphs. We discuss related work in Sect. 6 and conclude our paper in Sect. 7.

2 Preliminaries

2.1 Edge weight constraint

We first consider an edge-weighted undirected graph $G = (V, E, \Sigma, w)$, where V is the set of vertices, E is the set of edges, $\Sigma \subset \mathbb{R}$ is the set of real-valued edge weights and $w : E \mapsto \Sigma$ is a function that assigns each edge $e \in E$ to a real-valued weight $w(e) \in \Sigma$. A path P between vertices u and v is denoted as $P(u, v) = (u, v_1, \ldots, v_{l-1}, v)$, where $\{u, v_1, \ldots, v_{l-1}, v\} \subseteq V$ and $\{(u, v_1), \ldots, (v_{l-1}, v)\} \subseteq E$. We say e belongs to P, denoted as $e \in P$, if e is an edge on P. The edge weight constraint reachability query is defined as follows.

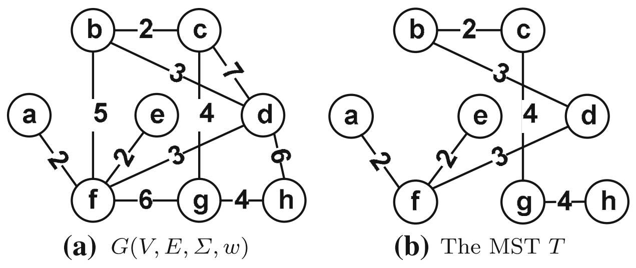


Fig. 1 An example graph G and its MST T

Definition 1 (*Edge weight constraint reachability* (EWCR)) Given a graph $G(V, E, \Sigma, w)$, an EWCR query is in the form of $q = (a, b, c)$, where $a, b \in V$ and c is a range constraint on edge weight, for example, $\geq x$, $\leq y$, or within $[x, y]$. q asks whether there is a path $P(a, b)$ between vertices a and b such that $\forall e \in P(a, b),\ w(e)$ satisfies the constraint c, for example, $w(e) \geq x$, or $w(e) \leq y$, or $w(e) \in [x, y]$, where $x, y \in \mathbb{R}$. If yes, we say a and b are reachable w.r.t. the edge weight constraint c.

Example 1 We use a running example throughout the paper. An undirected graph G is shown in Fig. 1a, where each edge has a real-valued weight. To answer an EWCR query $q = (a, g, \leq 4)$, we can find a path $P(a, g) = (a, f, d, b, c, g)$ in G such that $\forall e \in P(a, g),\ w(e) \leq 4$. Thus vertices a and g are reachable w.r.t. the constraint.

2.2 Node weight constraint

The WCR query also applies to node-weighted graphs, denoted as $G(V, E, \Sigma, w_n)$, where $w_n : V \mapsto \Sigma$ is a function that assigns each node $v \in V$ to a real-valued weight $w_n(v) \in \Sigma$. The node weight constraint reachability query is defined as follows.

Definition 2 (*Node weight constraint reachability* (NWCR)) Given a graph $G(V, E, \Sigma, w_n)$, an NWCR query is in the form of $q = (a, b, c)$, where $a, b \in V$ and c is a range constraint on node weight, for example, $\geq x$, $\leq y$, or within $[x, y]$. q asks whether there is a path $P(a, b)$ between vertices a and b such that $\forall v \in P(a, b),\ w_n(v)$ satisfies the constraint c, for example, $w_n(v) \geq x$, or $w_n(v) \leq y$, or $w_n(v) \in [x, y]$, where $x, y \in \mathbb{R}$. If yes, we say a and b are reachable w.r.t. the node weight constraint c.

Reducing NWCR *to* EWCR: An NWCR query on a node-weighted graph $G(V, E, \Sigma, w_n)$ can be reduced to an EWCR query, if we transform G to an edge-weighted graph $G'(V', E', \Sigma, w')$ as follows. For each edge in G, $e(a, b) \in E$, we create two weighted edges in E', that is, $e(a, v_{ab}) \in E'$ and $e(v_{ab}, b) \in E'$ with $w'(e(a, v_{ab})) = w_n(a)$ and $w'(e(v_{ab}, b)) = w_n(b)$. Here v_{ab} is a dummy nodeintroduced in V'. This transformation incurs a small constant-factor overhead on the edge set as $|E'| = 2|E|$. Then an NWCR query on G can be equivalently answered by an EWCR query on G'. Hence, we mainly focus on the EWCR query on edge-weighted graphs. For simplicity, in the following we use WCR to denote the edge weight constraint reachability problem.

More generally, our problem setting is not limited to real-valued edge or node weights, actually, it can be extended to any form of edge/node labels as long as they have a *total order*, for example, string labels with a total lexicographic order, and multidimensional features on edges/nodes given a function that maps a multidimensional feature to a *total ordered* one-dimensional value, for example, Hilbert curve [20]. Such graphs are commonplace in various scientific areas including bioinformatics and cheminformatics.

2.3 Two naive solutions

There are two naive approaches to answer the WCR query. One is online DFS or BFS search using the weight constraint to confine the search space. Starting from vertex a, we follow DFS or BFS order to recursively visit all adjacent vertices through edges whose weights satisfy the constraint, until we reach vertex b or have searched all reachable vertices without reaching b. The query time is $O(|V| + |E|)$, which is impractical for online query processing in a realtime system.

The other approach is to pre-compute the weight constraint reachability for all pairs $a, b \in V$. The query time is $O(1)$, but the space complexity is $O(|V|^2)$ for the constraint $\geq x$ or $\leq y$, or $O(|\Sigma||V|^2)$ for the constraint $[x, y]$, which severely limits the scalability.

The above two solutions are two extremes in terms of the query time complexity and index space complexity. In the following we will design novel index structures for efficient WCR query processing in large-scale networks.

3 An efficient memory algorithm

The user-specified edge weight constraint c can have various forms, such as a half-bounded interval, for example, $\geq x$, $\leq y$, or a bounded interval, for example, within $[x, y]$. It is not hard to see $\geq x$ and $\leq y$ are symmetric, thus without loss of generality we assume the constraint c has the form of $\leq y$. We will show later that our proposed algorithms can be easily extended to handle the bounded interval constraint $[x, y]$. We first show a property in WCR for query processing.

3.1 WCR property

Given a graph G and a WCR query $q(a, b, \leq y)$, for a path $P(a, b)$ in G, we denote the maximum edge weight on $P(a, b)$ as $\overline{P}(a, b) = \max_{e \in P(a,b)} w(e)$. If for *every* path $P(a, b)$ in G, $\overline{P}(a, b) > y$ holds, then we can conclude that a, b are not reachable w.r.t. the constraint $\leq y$. But it is too expensive to enumerate all possible paths $P(a, b)$ in G and check whether $\overline{P}(a, b) > y$ holds or not.

Let us consider the cut property [8] of minimum spanning tree (MST) of a graph, which states that *for any cut C in the graph, if the weight of an edge $e \in C$ is smaller than the weights of any other edges in C, then this edge e belongs to all MSTs of the graph.* Let T denote an MST of G. For a vertex pair (a, b), there is a unique tree path $P_T(a, b)$ between a and b on T. Let $\overline{P_T}(a, b) = \max_{e \in P_T(a,b)} w(e)$ denote the maximum edge weight on $P_T(a, b)$. Based on the cut property, we can derive the following lemma.

Lemma 1 *Given a graph $G(V, E, \Sigma, w)$, its MST T and an arbitrary vertex pair (a, b), for any $P(a, b)$ in G, $\overline{P_T}(a, b) \leq \overline{P}(a, b)$ holds, that is, $\overline{P_T}(a, b) \leq \min_{P(a,b)}\{\overline{P}(a, b)\}$.*

Proof For an arbitrary vertex pair (a, b), an a-b cut C_{ab} is a set of edges, the removal of which causes a and b to be in two disjoint components of G. Given an arbitrary path $P(a, b)$ in G and an arbitrary a-b cut C_{ab}, the intersection of $P(a, b)$ and C_{ab} must be non-empty, that is, $P(a, b) \cap C_{ab} \neq \emptyset$. Denote the minimum weight of a cut C_{ab} as $\underline{C_{ab}} = \min\{w(e)|e \in C_{ab}\}$. Then for any $e' \in P(a, b) \cap C_{ab}$, we have

$$\underline{C_{ab}} \leq w(e') \leq \overline{P}(a, b). \tag{1}$$

Let e_{max} denote the edge in $P_T(a, b)$ with the largest weight, that is, $e_{max} = \arg\max_{e \in P_T(a,b)} w(e)$. We can divide T into two disjoint components T_a and T_b by removing e_{max}. Here T_a and T_b represent the connected components in T containing a and b respectively. Then we obtain an a-b cut C^*_{ab} of G, $C^*_{ab} = \{e(u, v) \in E|u \in T_a, v \in T_b\}$ and $e_{max} \in C^*_{ab}$. Based on the cut property of MST, e_{max} must be the edge with the smallest weight in C^*_{ab}, that is,

$$w(e_{max}) = \underline{C^*_{ab}}, \tag{2}$$

as e_{max} belongs to T and no other edges in C^*_{ab} belong to T.

Combining (1) and (2), we prove

$$\overline{P_T}(a, b) = w(e_{max}) = \underline{C^*_{ab}} \leq \overline{P}(a, b)$$

for any path $P(a, b)$ in G.

From Lemma 1, we can derive the following theorem to answer a WCR query.

Theorem 1 *Two vertices a and b are reachable w.r.t. the weight constraint $\leq y$ in a graph G $\Leftrightarrow$ $\overline{P_T}(a, b) \leq y$ where T is the MST of G.*

Proof 1. $\Rightarrow$: As a and b are reachable w.r.t. the weight constraint $\leq y$ in the graph G through a path, denoted as $P(a, b)$, we have $\overline{P}(a, b) \leq y$ holds. By Lemma 1, we have $\overline{P_T}(a, b) \leq \overline{P}(a, b) \leq y$.

2. $\Leftarrow$: As $\overline{P_T}(a, b) \leq y$, we know $w(e) \leq y$ for each edge $e \in P_T(a, b)$. Thus, a and b are reachable w.r.t. the weight constraint $\leq y$ through path $P_T(a, b)$ in G.

Based on Theorem 1, a WCR query $q = (a, b, \leq y)$ can be processed as follows. We find the unique MST path $P_T(a, b)$ between a and b on T and compute the largest edge weight $\overline{P_T}(a, b)$. a and b are reachable w.r.t. the weight constraint if and only if $\overline{P_T}(a, b) \leq y$.

Example 2 For graph G in Fig. 1a, the MST T is shown in Fig. 1b. To answer a WCR query $q = (a, g, \leq 4)$, we find the unique tree path $P_T(a, g) = (a, f, d, b, c, g)$, such that $\forall e \in P_T(a, g)$, $w(e) \leq 4$. Thus vertices a and g are reachable w.r.t. the constraint.

An MST can be built in $O(|E|)$ time using Kruskal's algorithm [19] (with the union-find technique [8]). All edges in E can be sorted beforehand in $O(|E|)$ time using radix sort, since the edge weights are from a finite set. A straightforward approach takes $O(|V|)$ time to find the path $P_T(a, b)$ on T to answer a WCR query. Theorem 1 serves as a building brick for constructing an efficient index to answer a WCR query.

3.1.1 Connection with Gomory-Hu tree

The MST for answering a WCR query bears some similarity with the Gomory-Hu tree [11]. We discuss the connection between our MST and the Gomory-Hu tree here. The Gomory-Hu tree is defined as follows.

Definition 3 (*Gomory-Hu tree*) Given a weighted undirected graph $G = (V, E, w)$, denote the minimum capacity of an s-t cut by λ_{st}, for any $s, t \in V$. A tree $T = (V_T, E_T)$ with $V_T = V$ and $E_T \subset E$ is a Gomory-Hu tree of G if

$$\lambda_{st} = \min_{e \in P_T(s,t)} c(S_e, T_e), \forall s, t \in V$$

where

1. $P_T(s, t)$ is the path in T between $s, t \in V$,
2. S_e and T_e are the two connected components of $T \setminus \{e\}$ such that (S_e, T_e) forms an s-t cut in G, and
3. $c(S_e, T_e)$ is the capacity of the cut in G.

To encode the minimum cut capacity information into a Gomory-Hu tree T, for every edge $(a, b) \in E_T$, we can assign λ_{ab} as its weight. For a tree path $P_T(s, t)$, denote the minimum edge weight on $P_T(s, t)$ as $\underline{P_T}(s, t)$. By definition, $\underline{P_T}(s, t) = \lambda_{st}$. We have the following theorem.

Theorem 2 *Given a graph $G(V, E, w)$, we define G's λ-graph $G_\lambda = (V, V \times V, \lambda)$ as a complete graph whose edge weight on edge (s, t) is the minimum capacity of an s-t cut in G, denoted as λ_{st}. Let $\mathcal{P}(a, b) = \{P(a, b)\}$ be the set of all paths that connect a and b in G_λ, we have* $\lambda_{ab} = \max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\}$.

Proof First, the following triangle inequality holds:

$$\lambda_{ab} \geq \min\{\lambda_{ac}, \lambda_{bc}\}, \forall a, b, c \in V,$$

since c must be on one side of an a-b cut. Then, given any path in G_λ, $P(a, b) = (a, c_1, c_2, \ldots, c_l, b)$, we have

$$\begin{aligned}
\lambda_{ab} &\geq \min\{\lambda_{ac_l}, \lambda_{c_l b}\} \\
&\geq \min\{\lambda_{ac_{l-1}}, \lambda_{c_{l-1}c_l}, \lambda_{c_l b}\} \\
&\geq \ldots \\
&\geq \min\{\lambda_{ac_1}, \lambda_{c_1 c_2}, \ldots, \lambda_{c_l b}\} \doteq \underline{P}(a, b)
\end{aligned}$$

By considering all the paths $\mathcal{P}(a, b)$ in G_λ, we have $\lambda_{ab} \geq \max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\}$. As edge (a, b) itself is also a path between a and b, that is, $(a, b) \in \mathcal{P}(a, b)$, we have $\lambda_{ab} \leq \max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\}$. Thus we prove

$$\lambda_{ab} = \max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\}.$$

Theorem 3 *For any two nodes a, b in G, finding λ_{ab} on the Gomory-Hu tree is equivalent to calculating $\underline{P_T}(a, b)$ on the maximum spanning tree T of G_λ.*

Proof Since $\max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\}$ is a dual format of $\min_{P(a,b) \in \mathcal{P}(a,b)} \{\overline{P}(a, b)\}$ considered in Lemma 1, it is symmetric to prove for all the paths $\mathcal{P}(a, b)$ in G_λ,

$$\max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\} \leq \underline{P_T}(a, b),$$

where T is the maximum spanning tree of λ-graph G_λ. As $P_T(a, b) \in \mathcal{P}(a, b)$, we have

$$\max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\} \leq \underline{P_T}(a, b) \leq \max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\}.$$

Thus we prove

$$\underline{P_T}(a, b) = \max_{P(a,b) \in \mathcal{P}(a,b)} \{\underline{P}(a, b)\} = \lambda_{ab}.$$

From Theorem 3, we conclude that finding a minimum cut capacity on a Gomory-Hu tree is equivalent to applying our WCR technique on the maximum spanning tree of a *special graph* G_λ. But in the opposite direction, Gomory-Hu tree cannot be used to answer WCR queries on general graphs.

3.2 Novel edge-based indexing

In this section, we aim to design a novel index that can answer a WCR query in $O(1)$ time. According to Theorem 1, it is equivalent to solving the following problem: *Given an MST T, compute $\overline{P_T}(a, b)$ for any two vertices a and b in T in $O(1)$ time.*

Let $e_{\max}$ be the edge with the maximum edge weight $w_{\max}$ in T. If there are more than one edge with the same maximum weight w_{max}, we pick one of them arbitrarily. We have the following observations.

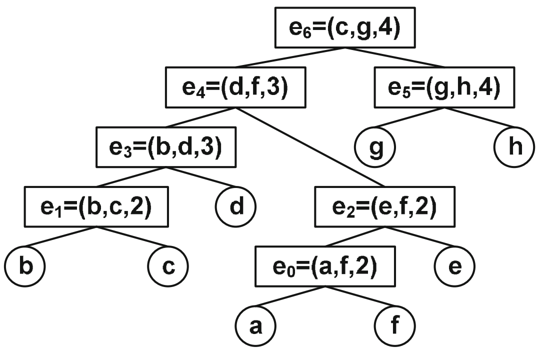


Fig. 2 The edge-based index tree $\mathcal{T}$ of MST T

1. After removing e_{max}, T becomes two disjoint subtrees T_1 and T_2.
2. For any vertices a in T_1 and b in T_2, $\overline{P_T}(a, b) = w_{max}$.
3. For any two vertices a and b in T_1 (or in T_2), $\overline{P_T}(a, b)$ can be similarly determined in T_1 (or in T_2).

Based on the above observations, for an MST T, we define its edge-based index tree as follows.

Definition 4 (*Edge-based index tree*) For an MST T, its edge-based index tree, denoted as $\mathcal{T}[T]$, is a vertex-labeled binary tree, such that: If T contains only one vertex, $\mathcal{T}[T]$ is a tree with a single vertex labeled 0. Otherwise, let e_{max} be an edge with the maximum edge weight w_{max} in T. Suppose after removing e_{max}, T becomes two disjoint subtrees T_1 and T_2. $\mathcal{T}[T]$ is recursively defined as follows.

– The root of $\mathcal{T}[T]$ is labeled w_{max}.
– The left subtree of $\mathcal{T}[T]$ is $\mathcal{T}[T_1]$.
– The right subtree of $\mathcal{T}[T]$ is $\mathcal{T}[T_2]$.

For any MST T, from the definition of $\mathcal{T}[T]$, each node $v \in V(T)$ corresponds to a unique leaf node in $\mathcal{T}[T]$ with a label 0, and each edge $e \in E(T)$ corresponds to a unique internal node in $\mathcal{T}[T]$ with a label $w(e)$. We use $L(v)$ to denote the label of any node v in $\mathcal{T}[T]$. If the context is obvious, we use $\mathcal{T}$ to denote $\mathcal{T}[T]$.

Example 3 Figure 2 shows the edge-based index tree $\mathcal{T}$ for the MST T shown in Fig. 1b. A leaf node in circle corresponds to a vertex $v \in V(T)$ and the letter in the circle denotes the vertex id v. The label of each leaf node in $\mathcal{T}$ is 0 and is not shown in the figure. An internal node with a triple (u, v, w) in rectangle corresponds to an edge $e(u, v) \in E(T)$ with $w = w(e(u, v))$. The label of an internal node in $\mathcal{T}$ is w. $\mathcal{T}$ organizes all 7 edges in T hierarchically.

Essentially, $\mathcal{T}$ is a delicate reorganization of all edges in MST T, and it supports computing $\overline{P_T}(a, b)$ in $O(1)$ time with little space overhead. $\mathcal{T}$ is stored in the memory as our index named Edge-Index. Next, we discuss query processing using Edge-Index, followed by the construction algorithm of Edge-Index.

Query processing: The edge-based index tree $\mathcal{T}$ has the following property: for an internal node $v \in V(\mathcal{T})$, its label $L(v) \leq L(r)$ where $r \in V(\mathcal{T})$ is any ancestor of v in $\mathcal{T}$. We have the following lemma.

Lemma 2 *Given an MST T of a graph $G(V, E, \Sigma, w)$ and its edge-based index tree $\mathcal{T}$, $\forall a, b \in V$, we denote the lowest common ancestor of a and b in $\mathcal{T}$ as $LCA_{\mathcal{T}}(a, b)$. Then we have*

$$\overline{P_T}(a, b) = L(LCA_{\mathcal{T}}(a, b))$$

Proof Let us consider the process to construct the edge-based index tree $\mathcal{T}$: We remove edges from the MST T one by one in the decreasing order of the edge weight. After removing a certain edge, the subtree that contains the edge becomes two disjoint subtrees and the removed edge becomes the root of a subtree in $\mathcal{T}$.

1. Let e be the first edge, whose removal separates a and b in two disjoint subtrees of T. It means e is the first node created in $\mathcal{T}$ that separates a and b, or in other words, $e = LCA_{\mathcal{T}}(a, b)$.
2. Let us consider the time before removing e and all edges with weights $> w(e)$ are removed. At that time, a and b are still in the same connected subtree in T, and e has the largest weight in the remaining edges. It means $\overline{P_T}(a, b) \leq w(e)$.
3. After removing e in T, a and b are disconnected, which means e is in $P_T(a, b)$. Thus $\overline{P_T}(a, b) \geq w(e)$.

From items 1–3, we can prove

$$\overline{P_T}(a, b) = w(e) = L(LCA_{\mathcal{T}}(a, b)).$$

Based on Lemma 2, the WCR query $q = (a, b, \leq y)$ can be processed as follows. On the edge-based index tree $\mathcal{T}$, we find the lowest common ancestor $LCA_{\mathcal{T}}(a, b)$ of a and b. If $\overline{P_T}(a, b) = L(LCA_{\mathcal{T}}(a, b)) \leq y$, then a and b are reachable w.r.t. the constraint; otherwise, they are not. The LCA query on an index tree $\mathcal{T}$ can be answered in $O(1)$ time with an $O(|V|)$ size index by transforming it into a range minimum query (RMQ). RMQ is defined as follows. Let A be an array of length n. For any indices $1 \leq i \leq j \leq n$, RMQ returns the index of the smallest element in the subarray $A[i, \ldots, j]$. The connection between LCA and RMQ is based on an observation—the LCA of nodes a and b is the shallowest node encountered between the visits to a and b during a depth-first search of a tree T. Literature [3] provides the technical details of the RMQ indexing and query processing. According to [3], the LCA index has $O(|V|)$ size and can be constructed in $O(|V|)$ time.

Algorithm 1 Edge-Index-Construct-Naive(T)

```
Input: An MST $T$
Output: An edge-based index tree $\mathcal{T}$ for $T$
1: if $T$ contains a single node then
2:     return a tree of a single node labeled 0;
3: $e_{max} \leftarrow$ an edge in $T$ with the maximum weight $w_{max}$;
4: remove $e_{max}$ from $T$ to generate two trees $T_1$ and $T_2$;
5: $\mathcal{T} \leftarrow$ a tree with root labeled $w_{max}$;
6: $\mathcal{T}.left \leftarrow$ Edge-Index-Construct-Naive($T_1$);
7: $\mathcal{T}.right \leftarrow$ Edge-Index-Construct-Naive($T_2$);
8: return $\mathcal{T}$;
```

Example 4 With the edge-based index tree $\mathcal{T}$ in Fig. 2, to answer the WCR query $q = (a, g, \leq 4)$, one has to find $LCA_{\mathcal{T}}(a, g)$, which is $e_6 = (c, g, 4)$. As $L(e_6) = 4 \leq 4$, a and g are reachable w.r.t. the constraint.

Index construction: Definition 4 gives a straightforward way to build the edge-based index tree $\mathcal{T}$ for an MST T in a top-down fashion. The index construction is shown in Algorithm 1 and is self-explanatory. Line 3 needs $O(|V|)$ time to find e_{max} with the maximum weight, and it is processed for $|V| - 1$ times using recursion. The total time complexity of Algorithm 1 is $O(|V|^2)$ for constructing Edge-Index. When the graph contains hundreds of millions of vertices, this time complexity is unacceptable.

Thus, we propose a novel linear-time method (in Algorithm 2) to build the edge-based index tree $\mathcal{T}$ from the MST T in *a bottom-up fashion*, in a way similar to Kruskal's algorithm [19].

In Algorithm 2, the edge-based index tree $\mathcal{T}$ is built with a function $R : V(\mathcal{T}) \mapsto V(\mathcal{T})$ which maps a node to its root. Initially, R maps each node to the node itself as the root (line 2–3). So all nodes in $\mathcal{T}$ are isolated. In each iteration, we pick an MST edge $e(a, b) \in E(T)$ in the ascending order of $w(e)$ and create an internal node with a label $w(e)$ in $\mathcal{T}$ (line 5–6). In line 7–8, Find-Root returns the current root of a and b in $\mathcal{T}$ respectively. Line 9–10 unions the two subtrees where a and b reside into one, by linking the two root nodes to the new internal node created in line 6. After inserting all edges $e \in E(T)$ as internal nodes in $\mathcal{T}$, the edge-based index tree $\mathcal{T}$ is constructed.

Algorithm 2 adopts the classical union-find algorithm [8]. The Find-Root procedure returns the current root of a node in $\mathcal{T}$. It adopts the path compression technique [8], thus the amortized time complexity for each Find-Root operation is $O(\alpha(|V|))$. Here $\alpha(n)$ is the inverse Ackermann function that is a very slowly growing function and can be considered as a small constant. As an indicator, $\alpha(2^{2^{2^{65536}}} - 3) = 4$. As we totally perform $O(|V|)$ Find-Root operations, the time for all Find-Root operations is $O(|V|)$. In addition, sorting all edges in $E(T)$ can be done in $O(|V|)$ time using radix sort since all edge weights are from a finite set. Hence, the overall

Algorithm 2 Edge-Index-Construct (T)

```
Input: An MST $T$
Output: An edge-based index tree $\mathcal{T}$ for $T$
1: $\mathcal{T} \leftarrow \emptyset$;
2: $R(v) \leftarrow v, \forall v \in V(T)$;
3: $R(e) \leftarrow e, \forall e \in E(T)$;
4: create a leaf node $v$ labeled 0 in $\mathcal{T}$, $\forall v \in V(T)$;
5: for $e(a, b) \in E(T)$ in ascending order of $w(e)$ do
6:     create an internal node $e$ labeled $w(e)$ in $\mathcal{T}$;
7:     $e.left \leftarrow$ Find-Root($a$);
8:     $e.right \leftarrow$ Find-Root($b$);
9:     $R(e.left) \leftarrow e$;
10:    $R(e.right) \leftarrow e$;
11: return $\mathcal{T}$;

12: Procedure Find-Root($v$)
13: if $R(v) = v$ then
14:     return $v$;
15: $R(v) \leftarrow$ Find-Root($R(v)$);
16: return $R(v)$;
```

time complexity of building an edge-based index tree $\mathcal{T}$ is $O(|V|)$.

Example 5 The edge-based index tree $\mathcal{T}$ in Fig. 2 is constructed from T in Fig. 1b as follows. Initially, all leaf nodes are isolated, and the root of each node is itself. We first sort the edges in $E(T)$ in the ascending order of their weights to be the edge sequence $e_0, \ldots, e_6$. In the first step, we create an internal node corresponding to $e_0(a, f, 2)$ and link nodes a and f as two children of the e_0 node. Next, we create an internal node corresponding to $e_1(b, c, 2)$ and link nodes b and c to it. In the third step, we create an internal node corresponding to $e_2(e, f, 2)$ and link node e and the current root of node f, that is, node $e_0(a, f, 2)$ to the e_2 node. This process iterates until we insert all 7 edges $e_0, \ldots, e_6$ as internal nodes into $\mathcal{T}$.

Lemma 3 *Constructing Edge-Index takes $O(|E|)$ time and $O(|V|)$ space. Using Edge-Index, the query time is $O(1)$.*

Proof We first build an MST from G in $O(|E|)$ time and then build the edge-based index tree and the LCA index in $O(|V|)$ time. Thus the time complexity for constructing Edge-Index is $O(|E| + |V|)$, or simplified as $O(|E|)$, as $|V| \leq |E| + 1$ usually holds, assuming G is a connected graph. The spaces for both the edge-based index tree and the LCA index are $O(|V|)$, and thus, the space complexity is $O(|V|)$. The query time is $O(1)$ for finding $LCA_{\mathcal{T}}(a, b)$ on $\mathcal{T}$.

3.3 Extension to other constraint formats

In the previous discussions, we assume the edge weight constraint has the form of $\leq y$. If the user-specified weight constraint is $\geq x$, Edge-Index can be applied similarly, since $\geq x$ and $\leq y$ are symmetric. The only change is to build a maximum spanning tree T' of G, instead of a minimum spanning tree. All the complexity results apply to the $\geq x$ constraint.

Algorithm 3 Query-Processing-Edge-Index $(\mathcal{T}_l, q)$

```
Input: Index Trees $\mathcal{T}_l$, $\forall l \in \Sigma$, and a WCR $q(a, b, [x, y])$
Output: Whether $a$ and $b$ are reachable w.r.t. $[x, y]$
1: $l \leftarrow \min_{l' \in \Sigma}\{l' \geq x\}$;
2: $r \leftarrow LCA_{\mathcal{T}_l}(a, b)$;
3: return $L(r) \leq y$;
```

When the weight constraint is a bounded interval $[x, y]$, we can show that Edge-Index can also be easily extended to handle this more general constraint.

3.3.1 When the weight constraint is $[x, y]$

In this section, we describe how to extend Edge-Index to handle the constraint $[x, y]$ respectively.

The y constraint can be handled in the same way. To satisfy the x constraint as well, we can simply remove all $e \in E$ with $w(e) < x$. For this purpose, we define the l-Graph.

Definition 5 (*l-graph*) Given $l \in \mathbb{R}$, we define the l-Graph $G_l(V, E_l, \Sigma_l, w)$ as a subgraph of $G(V, E, \Sigma, w)$, such that $E_l = \{e | e \in E, w(e) \geq l\}$ and $\Sigma_l = \{l' | l' \in \Sigma, l' \geq l\}$.

Index construction: For each $l \in \Sigma$, we first construct the MST T_l from the l-Graph G_l. A T_l may be a forest, due to the removal of edges with $w(e) < l$. It is trivial to handle this case by adding a virtual root. In the following, we assume T_l is connected for each l. For each MST T_l, $\forall l \in \Sigma$, we invoke Algorithm 2 to build an edge-based index tree $\mathcal{T}_l$ and build the LCA index for $\mathcal{T}_l$. Both the index tree $\mathcal{T}_l$, $\forall l \in \Sigma$, and the LCA index are kept in the memory.

Query processing: Given a query $q = (a, b, [x, y])$, we find $\mathcal{T}_l$ where $l = \min_{l' \in \Sigma}\{l' \geq x\}$. Then we compute $LCA_{\mathcal{T}_l}(a, b)$ on $\mathcal{T}_l$. a and b are reachable w.r.t. $[x, y]$ if and only if $\overline{P_{T_l}}(a, b) = L(LCA_{\mathcal{T}_l}(a, b)) \leq y$. Algorithm 3 lists the pseudo code for query processing by Edge-Index to handle the constraint $[x, y]$.

Lemma 4 *To handle a bounded interval constraint* $[x, y]$*, building Edge-Index takes* $O(|\Sigma||E|)$ *time and* $O(|\Sigma||V|)$ *space. Using Edge-Index, the query time is* $O(1)$.

We can see from Lemma 4, to handle a bounded constraint $[x, y]$, the index construction time and space complexities increase by a factor of $|\Sigma|$, but the query time complexity is the same as that for the constraint $\geq x$ or $\leq y$.

4 An I/O-efficient index

The indexing scheme Edge-Index assumes the index can entirely fit into the main memory, with a space complexity of $O(|\Sigma||V|)$ to handle the weight constraint $[x, y]$. When $|\Sigma|$ or $|V|$ is large which is very common for emerging massive graphs containing hundreds of millions of vertices, the index size may exceed the memory limit. In this section, we propose an I/O-efficient algorithm that builds compact disk-resident index for query processing with low I/O cost. We use the basic *external memory model* [30] that moves B data items continuously between external and internal memory as one I/O communication.

A straightforward implementation of the disk-based algorithm is to store all the $|\Sigma|$ MSTs on the disk. Given a query $q = (a, b, [x, y])$, the MST T_l where $l = \min_{l' \in \Sigma}\{l' \geq x\}$ is fetched into the memory to answer q. The disk-based index size is $O(|\Sigma||V|)$. However, it needs $O(|V|/B)$ I/O for query processing where B is the page size, as it fetches the whole MST T_l into the memory. Thus indexing MSTs directly on the disk is not I/O-efficient for query processing. Another approach is to store the edge-based index tree $\mathcal{T}_l$ as well as its LCA index, $\forall l \in \Sigma$ on the disk. However, answering queries using the disk-based LCA index is not I/O-efficient because of the complex structure used in the LCA index [3]. Our goal is to design a compact disk-based index based on the MSTs to reduce the I/O cost in query processing.

4.1 Vertex coding

For efficient storage and indexing, coding is a commonly used technique. To design an I/O-efficient disk index, we have an intuitive vertex coding scheme on an MST T as follows. We pick an arbitrary vertex $r \in V(T)$ as the root of T, thus T becomes a rooted MST. Given two vertices a and b, we denote the lowest common ancestor of a and b on such a tree as r_{ab}. Since r_{ab} lies on the tree path between a and b, we have

$$\overline{P_T}(a, b) = \max\{\overline{P_T}(a, r_{ab}), \overline{P_T}(b, r_{ab})\}.$$

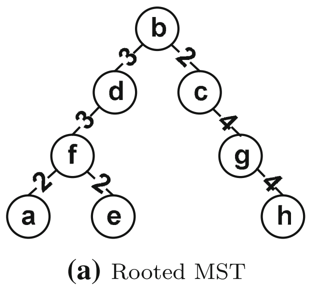
where $P_T(a, r_{ab})$ denotes the tree path between a and r_{ab}.

Based on the above equation, we generate a code for every vertex $v \in V(T)$. In the rooted MST T, we define the level of the root as 0, and the level of a child node increases that of its parent by 1. For any vertex v at level l of T, we find its ancestors r_i at level i, $0 \leq i \leq l - 1$. We also pre-compute the maximum edge weight $\overline{P_T}(v, r_i)$ on the path $P_T(v, r_i)$ from v to r_i on T. The code of a vertex v, code(v), is

$$\text{code}(v) = \{(r_i, \overline{P_T}(v, r_i)) | r_i \text{ is } v\text{'s ancestor at level } i, \; 0 \leq i \leq l - 1\}$$

The code is stored on the disk as index.

To answer a query $q = (a, b, [x, y])$, we retrieve the pages containing code(a) and code(b), from both of which we find the lowest common ancestor r_{ab} of a and b in the rooted MST. Thus $\overline{P_T}(a, b) = \max\{\overline{P_T}(a, r_{ab}), \overline{P_T}(b, r_{ab})\}$ can be determined by the two codes with no extra overhead. The I/O cost for a query is $O(h/B)$ on retrieving code(a) and



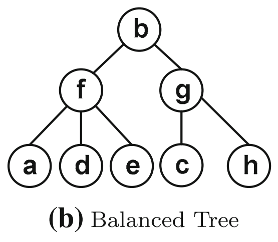


Fig. 3 Rooted MST and its balanced tree

code(b), where h is the tree height of the rooted MST (h is also the maximum number of ancestors). The code-based index uses $O(|V|h)$ space for one MST, or $O(|\Sigma||V|h)$ for $|\Sigma|$ MSTs.

Example 6 Figure 3a shows a rooted MST of T where vertex b is an arbitrarily picked root. To answer a query $q = (a, g, [2, 4])$, we retrieve

$$\text{code}(a) = \{(b, \overline{P_T}(a, b)),\ (d, \overline{P_T}(a, d)),\ (f, \overline{P_T}(a, f))\},$$
$$\text{code}(g) = \{(b, \overline{P_T}(g, b)),\ (c, \overline{P_T}(g, c))\}.$$

We can find $LCA(a, g) = b$. So

$$\overline{P_T}(a, g) = \max\{\overline{P_T}(a, b), \overline{P_T}(g, b)\} = 4.$$

The above coding method is not I/O efficient, because an MST may appear in arbitrary shape and thus the height h of an MST is $O(|V|)$ in the worst case, which causes $O(|V|/B)$ I/O cost for query processing and $O(|\Sigma||V|^2)$ index size in the worst case and is too expensive. To address this issue, we design a novel tree re-balancing technique to reorganize the MST into a balanced rooted tree, such that the height has an upper bound of $\log_2 |V|$. With the balanced tree, our disk-based algorithm only needs four I/Os for query processing and $O(|\Sigma||V| \log |V|)$ index size on the disk.

In the following, we will first introduce the tree re-balancing technique in Sect. 4.2 and then propose the disk-based index construction method in Sect. 4.3 and query processing algorithm in Sect. 4.4.

4.2 MST re-balancing

In the vertex coding scheme, the key to reduce the I/O cost in query processing and the disk-based index size is to make the height h of a rooted MST T as small as possible. However, when T is a chain, even if we pick the center node of T as its root, its height h is still as large as $\frac{|V|}{2}$. Thus, we need to re-balance T in order to make h small. Suppose we select a certain vertex $r \in V(T)$ as the root of T. Underneath the root r, T has $p \geq 1$ disjoint subtrees $T_1, T_2, \ldots, T_p$. For each subtree T_i, we select a vertex $r_i \in V(T_i)$ as the root of T_i for the balance purpose. This re-balancing process is recursively done in a top-down fashion to the leaf nodes. Although the re-balanced tree is no longer an MST, the vertex coding-based index and query processing can still be applied based on the following property.

Property 1 Once the root r of T is fixed, no matter how the subtrees $T_1, T_2, \ldots, T_p$ under r are re-balanced, for any two vertices $a \in V(T_i)$ and $b \in V(T_j)$ for $i \neq j$, their LCA in the re-balanced tree is still r, as a and b are from disjoint subtrees. Furthermore, r must lie on $P_T(a, b)$. The maximum weights $\overline{P_T}(a, r)$ and $\overline{P_T}(b, r)$ are pre-computed based on the MST paths $P_T(a, r)$ and $P_T(b, r)$ **in the original unbalanced MST**. $\overline{P_T}(a, b) = \max\{\overline{P_T}(a, r), \overline{P_T}(b, r)\}$ can be computed in the same way regardless of how the subtrees $T_1, T_2, \ldots, T_p$ are re-balanced.

The main purpose of the balanced tree is to reduce the tree height and thus reduce the code length and the disk-based index size. For an MST T, we define its balanced tree $\mathcal{B}[T]$.

Definition 6 (*Balanced tree*) For an MST T, its balanced tree, denoted as $\mathcal{B}[T]$, is a rooted tree such that

- The root node r of $\mathcal{B}[T]$ corresponds to a node in T.
- The height of $\mathcal{B}[T]$ is at most $\log_2 |V(T)|$.
- Underneath the root r, T has $p \geq 1$ disjoint subtrees $T_1, T_2, \ldots, T_p$, then the subtrees under root r in $\mathcal{B}[T]$ are $\mathcal{B}[T_1], \mathcal{B}[T_2], \ldots, \mathcal{B}[T_p]$.

If the context is obvious, we will use $\mathcal{B}$ to denote $\mathcal{B}[T]$.

Example 7 Figure 3b shows the balanced tree $\mathcal{B}$ for the unbalanced rooted MST in Fig. 3a. We pick node b as the tree root at level 0, and nodes f and g as the roots of the left and right subtrees at level 1. The tree height $h = 2 < \log_2 8 = 3$, as $|V(T)| = 8$. For vertices a and g, we can see that their lowest common ancestor in $\mathcal{B}$ is still vertex b, although the left and right subtrees are re-balanced.

Before introducing our algorithm to construct the balanced tree, we first study a special type of node in an MST T called *median node* which is defined as follows.

Definition 7 (*Median node*) Given an MST T, a node $v_m \in V(T)$ is a median node of T, if for each neighbor of v_m, that is, $\forall v' \in \{v | (v_m, v) \in E(T)\}$, $|V(T_{v'})| \leq \lfloor \frac{|V(T)|}{2} \rfloor$ holds, where $T_{v'}$ is the incident subtree of v_m rooted at a neighbor node v' and $|V(T_{v'})|$ is the number of nodes in $T_{v'}$.

We will select v_m as the root of a balanced tree $\mathcal{B}$. However, does the median node always exist? If yes, how to find it?

Algorithm 4 gives a constructive proof that such a median node always exists—the lowest node u with the subtree size satisfying $S_u = |V(T_u)| > \lfloor \frac{|V(T)|}{2} \rfloor$ is the median node. We

Algorithm 4 Find-Median-Node (T)

```
Input: An MST $T$
Output: The median node $v_m$ of $T$
1: Traverse $T$ from an arbitrarily picked root $r'$;
2: $v_m$ ← the lowest node $u$ with subtree rooted at $u$ satisfying $|V(T_u)| > \lfloor\frac{|V(T)|}{2}\rfloor$
3: return $v_m$;
```

Algorithm 5 Balanced-Tree-Construct (T)

```
Input: An MST $T$
Output: A balanced tree $\mathcal{B}$ for $T$
1: if $|V(T)| = 1$ then
2:     return a tree with the only node in $T$;
3: $r$ ← Find-Median-Node($T$);
4: $\mathcal{B}$ ← a tree of a single node $r$;
5: consider disjoint subtrees $T_1, T_2, \ldots, T_p$ under $r$ in $T$;
6: for $i = 1$ to $p$ do
7:     $\mathcal{B}_i$ ← Balanced-Tree-Construct($T_i$);
8:     add $\mathcal{B}_i$ to be a subtree under $r$ in $\mathcal{B}$;
9: return $\mathcal{B}$;
```

start from an arbitrarily picked root $r' \in V(T)$ to traverse the tree and find the median node. By 'lowest', we guarantee for each child node u_c of u, $|V(T_{u_c})| \le \lfloor\frac{|V(T)|}{2}\rfloor$ holds. We denote u's parent as u_p. After we re-root the tree on u, the size of subtree rooted at node u_p is

$$\begin{aligned}|V(T_{u_p})| &= |V(T)| - S_u\\ &\le |V(T)| - \lfloor\frac{|V(T)|}{2}\rfloor - 1\\ &\le \lfloor\frac{|V(T)|}{2}\rfloor\end{aligned}$$

Therefore, we prove for each neighbor v of u, the subtree rooted at v has a size satisfying $|V(T_v)| \le \lfloor\frac{|V(T)|}{2}\rfloor$. Thus u is a median node by definition. Algorithm 4 costs $O(|V(T)|)$ time to traverse the tree and find the median node.

Given an MST T, we choose the median node as the root and construct a balanced tree recursively based on Definition 6. The algorithm to construct the balanced tree $\mathcal{B}$ for T is shown in Algorithm 5 and is self-explanatory. We have the following theorem.

Theorem 4 *Given an MST T, the height of the balanced tree $\mathcal{B}[T]$ built is at most $\log_2|V(T)|$.*

Proof Let $h(\mathcal{B})$ be the height of the tree $\mathcal{B}$, and $h(n)$ be the maximum height of the balanced tree $\mathcal{B}[T]$ of any tree T with n nodes. Obviously, $h(n)$ is a nondecreasing function. In Algorithm 5, we find a median node r in T, whose removal from T generates p disjoint trees $T_1, T_2, \ldots, T_p$. Since r is a median node, we have $|V(T_i)| \le \lfloor\frac{|V(T)|}{2}\rfloor$, $\forall 1 \le i \le p$. From the construction of $\mathcal{B}$, we know

$$\begin{aligned}h(\mathcal{B}[T]) &= \max_{1\le i\le p} h(\mathcal{B}[T_i]) + 1\\ &\le \max_{1\le i\le p} h\left(\lfloor\frac{|V(T)|}{2}\rfloor\right) + 1 = h\left(\lfloor\frac{|V(T)|}{2}\rfloor\right) + 1\end{aligned}$$

Algorithm 6 Balanced-Index-Construct (G)

```
Input: A graph $G(V, E, \Sigma, w)$
Output: Balanced-Index for $G$
1: $\mathcal{I}$ ← an empty external code index;
2: $\mathcal{O}$ ← an empty external offset index;
3: for all $l \in \Sigma$ do
4:     $T_l$ ← MST for $G_l$;
5:     $\mathcal{B}_l$ ← Balanced-Tree-Construct($T_l$);
6:     for all $v \in V(\mathcal{B}_l)$ do
7:         code($v$) ← $\emptyset$;
8:         for all $v$'s ancestor $u$ in a top-down fashion do
9:             $P_{T_l}(v, u)$ ← the path from $v$ to $u$ on $T_l$;
10:            code($v$).Append(($u$, $\overline{P_{T_l}}(v, u)$));
11:        $\mathcal{O}$.Append($\mathcal{I}.offset$);
12:        $\mathcal{I}$.Append(code($v$));
```

Thus,

$$\begin{aligned}h(n) &= \max_{\forall T, s.t. |V(T)|=n} h(\mathcal{B}[T])\\ &\le h\left(\lfloor\frac{n}{2}\rfloor\right) + 1\\ &\le h\left(\lfloor\frac{n}{2^2}\rfloor\right) + 2\\ &\le \cdots \le h(1) + \log_2 n = \log_2 n\end{aligned}$$

Hence we prove $h(\mathcal{B}[T]) \le h(|V(T)|) \le \log_2|V(T)|$.

Lemma 5 *The time complexity for Algorithm 5 to construct the balanced tree $\mathcal{B}$ for tree T is $O(|V(T)|\log|V(T)|)$.*

Proof There are at most $\log_2|V(T)|$ levels in $\mathcal{B}$, and in each level, finding median nodes for all subtrees on the level takes at most $O(|V(T)|)$ time. The total time complexity is $O(|V(T)|\log|V(T)|)$.

4.3 Disk-based index construction

Our disk-based Balanced-Index includes two parts, namely a code index $\mathcal{I}$ and an offset index $\mathcal{O}$. $\mathcal{I}$ stores all the codes code(v), $\forall v \in T_l$ and $\forall l \in \Sigma$. For each node v, code(v) is a list of (key, $value$) pairs, where key is the node id for each ancestor of v in $\mathcal{B}[T_l]$ from the root to v, and $value$ is the maximum edge weight on the path from v to key in T_l, that is, $\overline{P_{T_l}}(v, key)$. $\mathcal{O}$ stores the offsets for all codes in $\mathcal{I}$, because codes are of different sizes, as nodes at different levels in $\mathcal{B}[T_l]$ have different number of ancestors.

Algorithm 6 constructs the Balanced-Index from a graph G. We first initialize $\mathcal{I}$ and $\mathcal{O}$ to be two empty lists. Then for each weight $l \in \Sigma$, we construct the balanced tree $\mathcal{B}_l$ from the MST T_l. For each node in $\mathcal{B}_l$, we calculate its code as described above. Note that the maximum edge weight $\overline{P_{T_l}}(v, u)$ on the path from v to v's ancestor u is calculated in T_l, not in $\mathcal{B}_l$. Finally, we append the code and the offset to $\mathcal{I}$ and $\mathcal{O}$ respectively.

Lemma 6 *Using Algorithm 6, Balanced-Index for graph G can be constructed using $O(|\Sigma||E| + |\Sigma||V|\log|V|)$ time*

Table 1 Complexity results for bounded interval constraint $[x, y]$

Methods	Index time	Index size	Query time
Edge-Index (memory-based)	$O(\|\Sigma\|\|E\|)$	$O(\|\Sigma\|\|V\|)$	$O(1)$
Balanced-Index (disk-based)	$O(\|\Sigma\|\|E\| + \|\Sigma\|\|V\| \log \|V\|)$	$O(\|\Sigma\|\|V\| \log \|V\|)$	4 I/Os

and $O(|\Sigma||V| \log |V|)$ disk space. The I/O cost to store the index is $O(\frac{|\Sigma||V| \log |V|}{B})$ where B is the page size.

Proof We first build $|\Sigma|$ MSTs in $O(|\Sigma||E|)$ time and $|\Sigma|$ balanced trees in $O(|\Sigma||V| \log |V|)$ time. This uses $O(|V|)$ memory space for processing one MST and balanced tree at a time. We also need $O(|\Sigma||V| \log |V|)$ time and disk space to calculate and store all the codes from the balanced trees, because each balanced tree has $|V|$ codes and each code has at most $\log_2 |V|$ $(key, value)$ pairs. Since creating $\mathcal{I}$ and $\mathcal{O}$ uses sequential I/Os, the total I/O cost is $O(\frac{|\Sigma||V| \log |V|}{B})$.

4.4 Query processing

We show how to process queries using the disk-based offset index $\mathcal{O}$ and code index $\mathcal{I}$ in Algorithm 7. Given a query $q = (a, b, [x, y])$, find $l = \min_{l' \in \Sigma}\{l' \geq x\}$. Suppose l is ranked r_l in Σ, and the id of node a in graph G is id_a. Let $p = r_l \times |V| + id_a$, we can get the offset of $\mathsf{code}(a)$ in $\mathcal{I}$ by retrieving the p-th element from index $\mathcal{O}$ with one I/O. Using the offset, we can retrieve $\mathsf{code}(a)$ from index $\mathcal{I}$, which contains at most $\log_2 |V|$ $(key, value)$ pairs. This operation needs one I/O since one page is enough to hold $\log_2 |V|$ pairs in a code for a very large $|V|$, that is, in the scale of $O(2^B)$, where B is the page size. We set $B = 4096$ bytes in our implementation. Similarly, we retrieve $\mathsf{code}(b)$ from index $\mathcal{I}$. We compare the i-th element in $\mathsf{code}(a)$ with the i-th element in $\mathsf{code}(b)$ one by one in a top-down fashion, until they are not referring to the same node. The last node with $\mathsf{code}(a)[i].key = \mathsf{code}(b)[i].key$ corresponds to the LCA of a and b in $\mathcal{B}_l$. We have

$$\overline{P_{T_l}}(a, b) = \max\{\mathsf{code}(a)[i].value, \mathsf{code}(b)[i].value\}$$

Thus we return true if $\overline{P_{T_l}}(a, b) \leq y$, and return false otherwise. We totally need four I/Os to retrieve the index entries for a and b to answer a WCR query. The offset index $\mathcal{O}$ contains $|\Sigma||V|$ offset values and is typically small enough to fit in the memory. Thus if $\mathcal{O}$ is in the memory, we only need two I/Os for query processing.

Algorithm 7 Query-Processing-Balanced-Index $(\mathcal{I}, \mathcal{O}, q)$

```
Input: The code index $\mathcal{I}$ and the offset index $\mathcal{O}$, and a WCR query $q(a, b, [x, y])$
Output: Whether $a$ and $b$ are reachable w.r.t. $[x, y]$
1: $l \leftarrow \min_{l' \in \Sigma}\{l' \geq x\}$;
2: $o_a \leftarrow \mathcal{O}$.Get-Offset$(l, a)$; code$(a) \leftarrow \mathcal{I}$.Get-Code$(o_a)$;
3: $o_b \leftarrow \mathcal{O}$.Get-Offset$(l, b)$; code$(b) \leftarrow \mathcal{I}$.Get-Code$(o_b)$;
4: $i \leftarrow \max\{j | \mathsf{code}(a)[j].key = \mathsf{code}(b)[j].key\}$;
5: return $\max\{\mathsf{code}(a)[i].value, \mathsf{code}(b)[i].value\} \leq y$;
```

Lemma 7 *Answering a WCR query with Balanced-Index by Algorithm 7 takes $O(\log |V|)$ time and four I/Os.*

Proof The I/O cost is on retrieving the offsets from $\mathcal{O}$ and the codes from $\mathcal{I}$. We need one I/O to retrieve each offset. The number of $(key, value)$ pairs in a code is at most $\log_2 |V|$ and is small enough to fit into one page, thus we can use one I/O to retrieve the code for each node. Thus we totally need four I/Os. After retrieving the codes into memory, in the worst case, we need to traverse all elements in the two codes once to find the LCA of a and b in $\mathcal{B}_l$, which needs $O(\log |V|)$ time.

The disk-based algorithm can be applied directly to solve the half-bounded constraint $\geq x$ and $\leq y$. The only difference is that we build only one balanced tree $\mathcal{B}$ from the MST of the graph G. Thus the index construction time is $O(|E| + |V| \log |V|)$ and the disk index size is $O(|V| \log |V|)$. The query time is the same as in Lemma 7. Finally, Table 1 summarizes the complexities of our proposed algorithms.

5 Experiments

In this section, we perform extensive experimental studies on real and synthetic datasets. We systematically test our memory algorithm Edge-Index and the I/O-efficient algorithm Balanced-Index. We report three performance measures, index construction time (IT), index size (IS), and query time (QT). All our algorithms are implemented in C++, and our experiments are performed on a machine with a 2.67 GHz CPU and 12 GB memory.

Besides the two algorithms we proposed, we also test the following baseline methods for comparison.

1. Naive-Search Methods: We take three basic search approaches, depth-first search (DFS), breadth-first search (BFS), and bi-directional search (BIS) [9], as memory-based baselines. Given a query $q = (a, b, [x, y])$, to find a valid path between a and b, Naive-Search only visits edges that satisfy the $[x, y]$ constraint. Being different only on the searching order, they share the same index time $O(|E| \log |E|)$ (for pre-sorting all graph edges according to the *start node id* of the edges, if the input is

not sorted), index size $O(|V|+|E|)$ (for storing the original graph G), and worst-case query time $O(|V|+|E|)$. We also use them as disk-based baselines by adapting them to external memory, denoted as Ext-DFS, Ext-BFS, and Ext-BIS, respectively. In the preprocessing phase, we need to pre-sort all edges in memory in $O(|E|\log|E|)$ time and then sequentially store the adjacency lists on disk with $O(|E|/B)$ I/Os. The disk-based index takes $O(|E|)$ size. In the online phase, when extending a node v, it needs $O(1+degree(v)/B)$ I/Os to fetch v's neighbors from disk. Thus the total number of I/Os for query processing is $O(\sum_{v\in V}(1+degree(v)/B)) = O(|V|+|E|/B)$ I/Os in the worst case.

2. MST-Index: MST-Index is a memory-based baseline, which builds $|\Sigma|$ MSTs in memory, denoted as T_l, $\forall l \in \Sigma$. Given a query $q=(a,b,[x,y])$, MST-Index finds $l=\min_{l'\in\Sigma}\{l'\geq x\}$ and computes $\overline{P_{T_l}}(a,b)$ on T_l. The index time and space complexities of MST-Index is the same as those of Edge-Index, but its query time is $O(|V|)$.
3. External-MST: External-MST is a disk-based baseline, which stores $|\Sigma|$ MSTs on disk. Given a query $q=(a,b,[x,y])$, External-MST finds $l=\min_{l'\in\Sigma}\{l'\geq x\}$ and fetches the MST T_l into the memory. Then it computes $\overline{P_{T_l}}(a,b)$ on T_l. It uses $O(|V|/B)$ I/Os for query processing.
4. Sampling-Tree [14] and 2-Label-Hop [32]: These two approaches handle label-constraint reachability (LCR) on directed graphs, where the labels appearing on the path from a node u to another node v should be a subset of a user-provided label set. In terms of problem hardness, LCR is more difficult than WCR. But since LCR is the closest to our WCR problem in the literature, we adapt Sampling-Tree and 2-Label-Hop to answer WCR query on undirected graphs for performance comparison.

5.1 Experiments on real datasets

In this experiment, we evaluate the performance of different methods on two real-world datasets. The first is the Facebook New Orleans network[2] [29] over a period of 2 years. A node represents a user, and an undirected edge denotes a user-to-user friendship link. Each edge has a weight denoting the UNIX timestamp with the time of link establishment. We generalize the timestamps into two granularities, hour and day. The two resulting networks are denoted as Facebook (h) and Facebook (day). The second graph is the USA road network,[3] a representative of very large-scale networks.

Table 2 Real network statistics

Network	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\Sigma\rvert$
Facebook (day)	63,731	440,384	862
Facebook (h)	63,731	440,384	19,657
USARN	23,947,347	29,166,672	12

A node represents an intersection or endpoint while an edge represents a road segment. We generate 12 weights from 10 to 120 with a step size of 10 and randomly assign a weight to each edge to represent the road speed limit. Table 2 lists the statistics of these real networks.

We generate and test 10,000 queries for each real network. All the queries have the constraint format of $[x,y]$, where x,y are randomly picked from Σ. Tables 3 and 4 show the index time (in s), index size (in GB) and average query time (in μs) of memory-based and disk-based algorithms respectively. Sampling-Tree and 2-Label-Hop cannot finish index construction on any of these datasets within 12 h. The comparisons are as follows.

Memory-based algorithms (Table 3) The query time of Edge-Index and MST-Index is the same on Facebook (day), but on USARN with 24 million nodes, the query time of MST-Index increases dramatically to 1382 μs, which is 345 times slower than Edge-Index. This is because the query time complexity of MST-Index is $O(|V|)$. The index time of Edge-Index is 1.75–3.95 times that of MST-Index, and the index size of Edge-Index is about 3 times larger, because Edge-Index needs to build edge-based index trees and LCA index. Both methods run out of memory on Facebook (Hour) on indexing due to the large number of weights, $|\Sigma| = 19,657$. Edge-Index uses 13.9–122.3 times longer index time and 12.9–220.0 times larger index size than Naive-Search due to the $|\Sigma|$ factor. But its query time is within 4 μs , **three orders of magnitude** faster than that of Naive-Search, which takes 1,098–32,462 μs.

Disk-based algorithms (Table 4) The query time of Balanced-Index is very stable on all networks, taking 11 or 18 μs. The query time of External-MST is orders of magnitude longer than that of Balanced-Index, especially on USARN with 24 million vertices, as it takes $O(|V|/B)$ I/Os to fetch an MST. The index size of Balanced-Index is 2.85–4.3 times larger than that of External-MST and the index time is 3–10 times longer. Both methods do not suffer from the memory limit as they build disk-based index. Balanced-Index's index size is 6.95–6843 times larger than that of Naive-Search, since the index size of Balanced-Index is linear with $|\Sigma|$, whereas that of Naive-Search is $O(|E|)$, not affected by $|\Sigma|$. But the query time of Balanced-Index is within 18 μs in all cases, **2,851–16,362 times** faster than that of Naive-Search, which takes 29,385–294,521 μs.

[2] http://socialnetworks.mpi-sws.org/data-wosn2009.html.

[3] http://www.dis.uniroma1.it/~challenge9/download.shtml.

Table 3 Memory-based algorithms on real dataset results (IT in s, IS in GB, QT in μs)

	Naive-Search					MST-Index			Edge-Index		
	IT	IS	QT			IT	IS	QT	IT	IS	QT
			DFS	BFS	BIS						
Facebook (day)	0.4	0.01	1,098	1,429	2,324	27.9	0.66	1	48.9	2.20	1
Facebook (h)	0.4	0.01	1,500	1,377	1,860	–	$>12G$	–	–	$>12G$	–
USARN	33.7	0.89	32,462	30,868	31,325	119.0	3.45	1,382	469.9	11.49	4

Table 4 Disk-based algorithms on real dataset results (IT in s, IS in GB, QT in μs)

	Naive-Search					External-MST			Balanced-Index		
	IT	IS	QT			IT	IS	QT	IT	IS	QT
			Ext-DFS	Ext-BFS	Ext-BIS						
Facebook (day)	0.6	0.01	31,368	48,152	45,405	41.8	0.66	772	125.6	2.98	11
Facebook (h)	0.5	0.01	35,325	57,366	47,533	922.3	15.03	749	3336.3	68.43	11
USARN	48.8	0.89	294,521	64,471	29,385	146.9	3.45	422,810	1425.9	6.19	18

Summary Our memory method Edge-Index has a very low and stable query time. But the memory index size can be a bottleneck for Edge-Index (also for baseline MST-Index). Naive-Search only stores the original graph which is compact, but its query time is three orders of magnitude longer. In fact, all the baseline methods, Naive-Search, MST-Index and External-MST, have a long query time especially when $|V|$ is large. In contrast, our disk-based Balanced-Index is very scalable, and the query time remains very low and stable regardless of the network size, the edge density, or the distinct weight number. The choice of algorithm depends upon the network size and the available memory.

5.2 Memory-based algorithms on synthetic datasets

To test the parameter sensitivity of different methods, we generate a collection of random graphs based on *Erdös-Rényi* model [18] by varying three parameters $|E|/|V|$, $|V|$ and $|\Sigma|$. $|E|/|V|$ is the density of the graph. The default values are $|E|/|V| = 2$, $|V| = 10^5$ and $|\Sigma| = 100$. We assign a random weight to each edge in the graphs. The edge weights follow a uniform distribution.

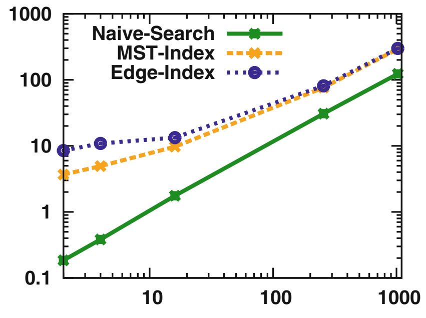
We first test our memory algorithm Edge-Index. For comparison, we also test the baselines, main memory Naive-Search (DFS, BFS and BIS) and MST-Index. The disk-based algorithm will be discussed separately in Sect. 5.3. We test 10,000 random queries on each graph and report index construction time (in seconds), index size (in GB) and the average query time (in μs). All the queries have the constraint format of $[x, y]$, where x, y are random real numbers from Σ.

Varying density: In this experiment, we vary the edge density $|E|/|V|$ from 2 to 1024 in log scale and fix $|V| = 10^5$ and $|\Sigma| = 100$. Figure 4a–c shows the index construction time, index size and query time in log scale, respectively.

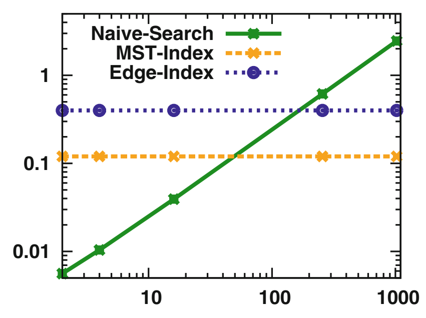
The index time of Edge-Index and MST-Index increases with the density $|E|/|V|$, or equivalently with $|E|$ as $|V|$ is fixed, since they both take $O(|\Sigma||E|)$ time to build MSTs. In addition, Edge-Index needs to build edge-based index trees and LCA index in $O(|\Sigma||V|)$ time, thus its index time is longer. But the margin between Edge-Index and MST-Index decreases with the increase of $|E|$, as the MST construction time dominates LCA index building time given a large $|E|$. The index time of Naive-Search is linear with $|E|$ since it is dominated by the loading cost of the graph, $|E|/B$ I/Os, which is much larger than $O(|E| \log |E|)$ sorting cost.

The index size of Edge-Index and MST-Index is not affected by the density. MST-Index uses 0.12 GB space for indexing MSTs, each of which contains $|V|$ vertices; Edge-Index uses 0.40 GB to store the edge-based index trees, each of which contains $\leq 2|V|$ vertices, as well as the LCA index. In contrast, the index size of Naive-Search increases linearly with $|E|$. When $|E|/|V| = 1024$, Naive-Search uses 6 times larger index size than Edge-Index.

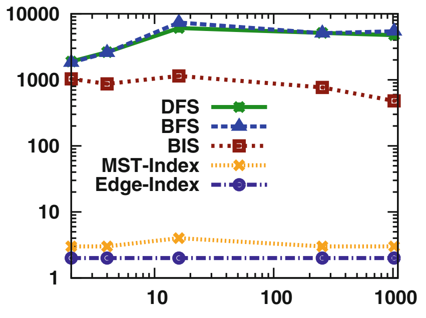
Edge-Index takes 2 μs for query processing via an $O(1)$ LCA operation. Thus its query time is not affected by the density. The query time of MST-Index is around 3–4 μs when the density changes. As it needs to search the MST online for the path between two query nodes, the query time



(a) Index Time (seconds)

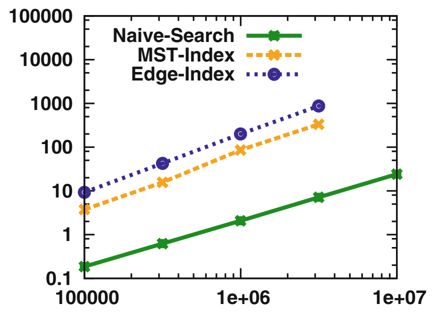


(b) Index Size (GB)

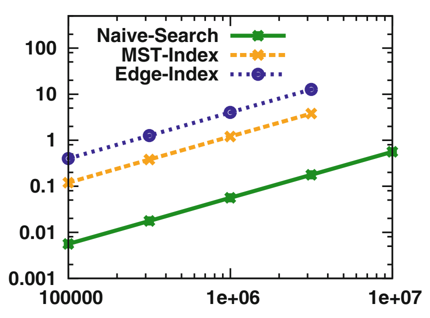


(c) Query Time (microseconds)

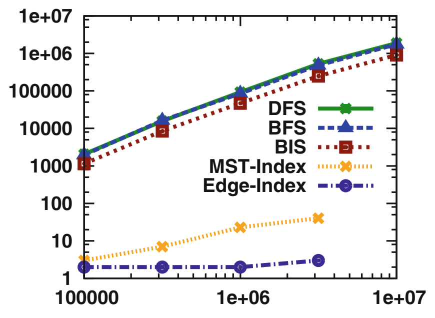
Fig. 4 Memory-based algorithms on synthetic datasets, varying $|E|/|V|$ from 2 to 1, 024, $|V| = 10^5$, $|\Sigma| = 100$



(a) Index Time (seconds)



(b) Index Size (GB)



(c) Query Time (microseconds)

Fig. 5 Memory-based algorithms on synthetic datasets, varying $|V|$ from 10^5 to 10^7, $|E|/|V| = 2$, $|\Sigma| = 100$

depends on the MST structure and how distant two query nodes are on the MST. The query time of Edge-Index is 915–3,682 times faster than that of DFS and BFS on graphs with different density. Although BIS slightly improves DFS and BFS (by 2–10 times) by reducing the searching space, it is still at least two orders of magnitude slower than Edge-Index.

Varying vertex number: In this experiment, we vary $|V|$ from 10^5 to 10^7 in log scale and fix $|E|/|V| = 2$ and $|\Sigma| = 100$. Figure 5a–c shows the index construction time, index size and query time in log scale, respectively.

The index time of all algorithms increases with $|V|$, as $|E|$ increases with $|V|$ given a fixed density. Since Edge-Index needs to build edge-based index trees and LCA index, its index time is about 2.6 times that of MST-Index, and 93 times longer than that of Naive-Search. The index size of all methods increases linearly with $|V|$. Edge-Index uses index size about 3.3 times more than MST-Index, and 71 times more than Naive-Search. When $|V| = 10^7$, the index size of both Edge-Index and MST-Index exceeds the 12 GB memory limit, but the index size of Naive-Search is much smaller by keeping the original graph only. The query time of Edge-Index remains 2 μs when $|V|$ increases, whereas that of MST-Index increases linearly with $|V|$ as the query time is $O(|V|)$. When $|V| = 10^{6.5}$, the query time of MST-Index is 20 times that of Edge-Index, which is a very big difference. The query time of Naive-Search increases linearly with $|V|$ and is three to five orders of magnitude longer than Edge-Index. When $|V| = 10^7$, it takes 1.5 s on average to answer one query.

Varying distinct weight number: In this experiment, we vary $|\Sigma|$ from 10^2 to 10^4 in log scale and fix $|E|/|V| = 2$ and $|V| = 10^5$. Figure 6a–c shows the index construction time, index size and query time in log scale, respectively.

The index time and size increase linearly with $|\Sigma|$ for both Edge-Index and MST-Index. The index time of Edge-Index is 2.4 times that of MST-Index, and the index size of Edge-Index is 3.3 times larger. Both methods run out of memory when $|\Sigma| = 10^4$. The query time of Edge-Index remains 2 μs, while that of MST-Index remains 3 μs. This demonstrates that our query processing is not affected by the weights appearing in the graph or in the query. The index time, index size and query time of Naive-Search remain stable when increasing $|\Sigma|$. Its query time is three orders of magnitude longer than that of Edge-Index and MST-Index.

Summary Edge-Index takes only 2 μs query time in all networks which is very stable and fast. The query time differ-

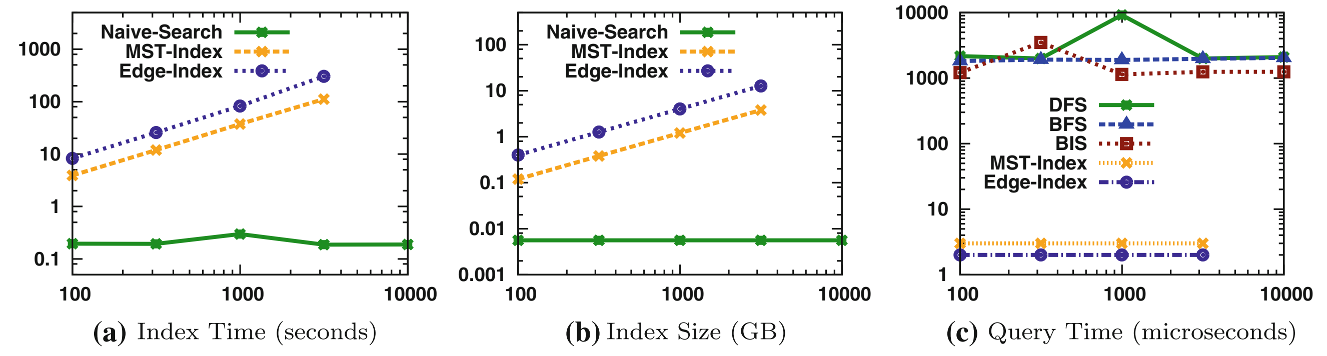


Fig. 6 Memory-based algorithms on synthetic datasets, varying $|\Sigma|$ from 10^2 to 10^4, $|E|/|V| = 2$, $|V| = 10^5$

Table 5 Online performance of memory-based algorithms

	Dependency	Response time (μs)
Naive-Search	Linear with $\|V\|$	481–1,892,083
MST-Index	Linear with $\|V\|$	3–57
Edge-Index	None	2

ence between Edge-Index and MST-Index increases with $|V|$ linearly and thus can be very significant when $|V|$ is large. Edge-Index uses 2–3 times more index time and space than MST-Index, which is a small overhead. When $|V| = 10^7$ or $|\Sigma| = 10^4$, both Edge-Index and MST-Index run out of the 12 GB memory limit. Naive-Search uses much smaller index time and size. But its query time is at least three orders of magnitude longer than that of Edge-Index, moreover, it increases linearly with $|V|$ and $|E|$. This is prohibitive for an online system. The online performance is summarized in Table 5.

5.3 Disk-based algorithms on synthetic data

We test our disk-based algorithm Balanced-Index on the same set of random graphs tested in Sect. 5.2. For comparison, we also test the following baseline methods, Naive-Search (Ext-DFS, Ext-BFS and Ext-BIS) and External-MST. Naive-Search stores sorted adjacency lists on disk and External-MST stores $|\Sigma|$ MSTs on disk. In query processing, Naive-Search traverses along edges that satisfy the $[x, y]$ constraint, while External-MST loads an MST T_l where $l = \min_{l' \in \Sigma}\{l' \geq x\}$ to answer a WCR query.

Varying density: In this experiment, we vary the edge density $|E|/|V|$ from 2 to 1024 in log scale and fix $|V|$ and $|\Sigma|$. Figure 7a–c shows the performance measures.

The index time of Balanced-Index is about 4 times that of External-MST when the density is small, that is, $|E|/|V| = 2-16$, but the margin decreases with the density increase. Balanced-Index uses 32 times longer index time than Naive-Search on average, but the margin decreases dramatically to less than 2 when $|E|/|V| \geq 100$.

The index size of External-MST is 0.12 GB under different density values. The index of Balanced-Index is about 4.5–7.3 times larger, as the vertex-coding-based index takes $O(|\Sigma||V| \log |V|)$ space. Naive-Search's index size increases linearly with $|E|$ and is 2.8 times larger than that of External-MST when $|E|/|V| = 1024$.

There is a very large margin between the query time of External-MST and Balanced-Index. External-MST takes 1635 μs, while Balanced-Index takes 23 μs, which

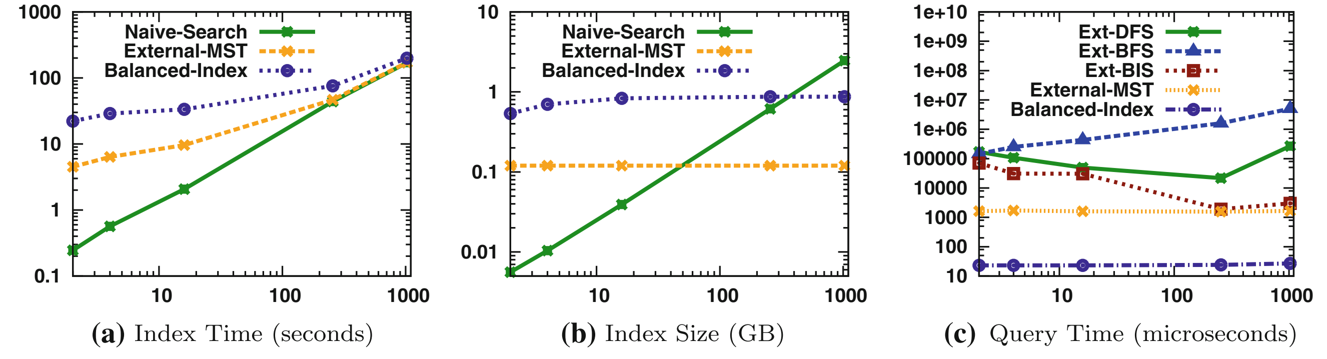
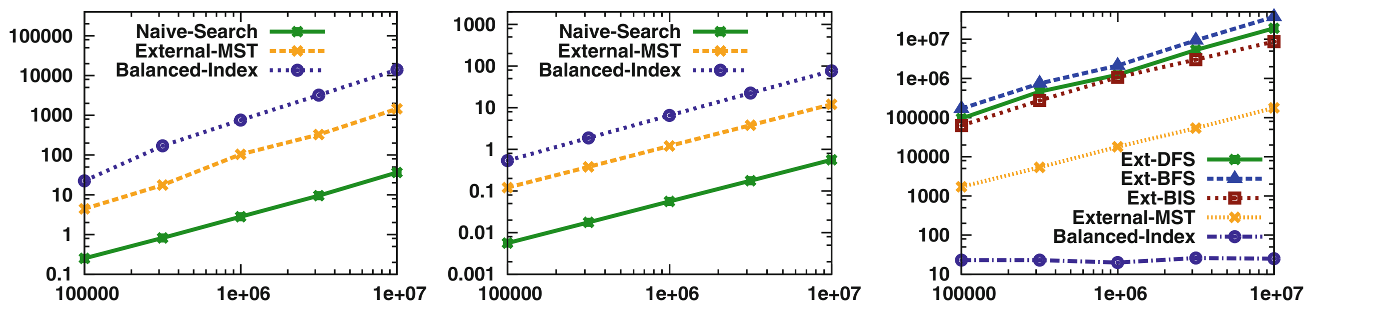
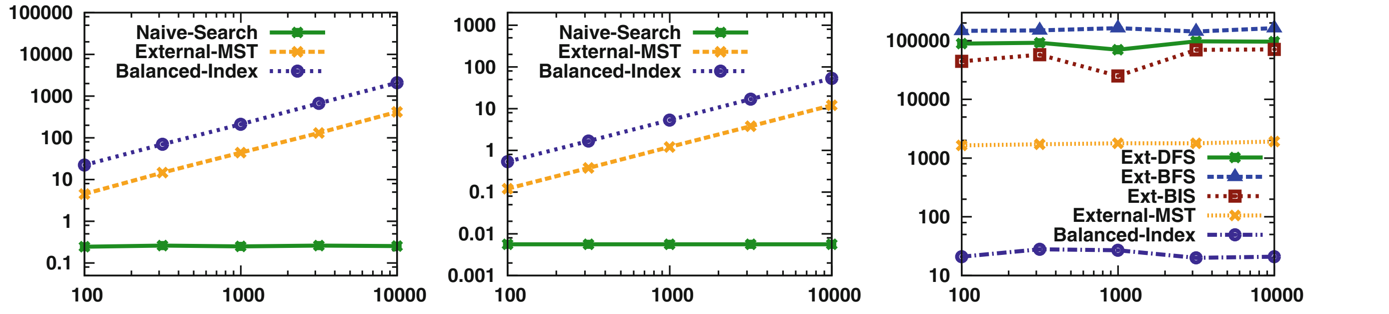


Fig. 7 Disk-based algorithms on synthetic datasets, varying $|E|/|V|$ from 2 to 1024, $|V| = 10^5$, $|\Sigma| = 100$



(a) Index Time (seconds) **(b)** Index Size (GB) **(c)** Query Time (microseconds)

Fig. 8 Disk-based algorithms on synthetic datasets, varying $|V|$ from 10^5 to 10^7, $|E|/|V| = 2$, $|\Sigma| = 100$



(a) Index Time (seconds) **(b)** Index Size (GB) **(c)** Query Time (microseconds)

Fig. 9 Disk-based algorithms on synthetic datasets, varying $|\Sigma|$ from 10^2 to 10^4, $|E|/|V| = 2$, $|V| = 10^5$

is 71 times faster. The query time of External-MST and Balanced-Index is not affected by the density. In contrast, the query time of Naive-Search varies with different densities. Ext-BIS is the most efficient among the three variants. On average, the query time of Naive-Search is three to four orders of magnitude longer than Balanced-Index.

Varying vertex number: In this experiment, we vary $|V|$ from 10^5 to 10^7 in log scale and fix $|E|/|V|$ and $|\Sigma|$. Figure 8a–c shows the performance measures.

The index time of Balanced-Index is about 5–10 times longer than that of External-MST, and its index size is about 4.5–6.4 times larger. The index time and index size of all the three methods increase near linearly with $|V|$. The query time of Balanced-Index remains 23 μs when $|V|$ increases. In contrast, the query time of External-MST and Naive-Search increases linearly with $|V|$, because External-MST takes $O(|V|/B)$ I/Os to load an MST, and Naive-Search uses $O(|V| + |E|/B)$ I/Os to search online. When $|V| = 10^7$, External-MST takes 176,901 μs to answer a query, 7,691 times slower than Balanced-Index. Naive-Search is even slower, with 2,722–764,514 times longer query time than Balanced-Index.

Varying distinct weight number: In this experiment, we vary $|\Sigma|$ from 10^2 to 10^4 in log scale and fix $|E|/|V|$ and $|V|$. Figure 9a–c shows the performance measures.

The index time and size increase linearly with $|\Sigma|$ for both External-MST and Balanced-Index. The index time of Balanced-Index is about 5 times longer than that of External-MST, while the index size of Balanced-Index is about 4.5 times larger. The query time of both methods remains very stable as $|\Sigma|$ increases. The query time of Balanced-Index is about 23 μs while that of External-MST is 1770 μs, which is 77 times slower. The index time and index size of Naive-Search is independent of $|\Sigma|$, which remain 0.25 s and 5.6 MB, respectively. Naive-Search's query time does not vary much with $|\Sigma|$ either. For comparison, DFS, BFS, and BIS take 89,000, 153,000 and 53,000 μs respectively for query processing on average, whereas Balanced-Index only takes 23 μs, which is three orders of magnitude faster.

Summary Balanced-Index scales to very large networks while at the same time providing 23 μs response time. It uses 5–10 times more index time and space than External-MST, which is a small overhead. Naive-Search has a low index time and size in most cases. But when the density $|E|/|V|$ is large, the index size of Naive-Search can be much larger than that of Balanced-Index. The index size of Balanced-Index is independent of $|E|/|V|$ or $|E|$. The query time of Naive-Search is three to four orders of magnitude longer than that of Balanced-Index. The online performance of disk-based approaches is summarized in Table 6.

5.4 Comparing Edge-Index with Sampling-Tree and 2-Label-Hop

We compare our solution with Sampling-Tree [14] and 2-Label-Hop [32] in this experiment. Sampling-Tree and 2-Label-Hop are memory algorithms to solve the label-constraint reachability on directed graphs. In terms of problem hardness, LCR is much more difficult than WCR. So adapting LCR solutions to answer WCR queries is unnecessarily complicated and thus is not very fair. Nevertheless, as LCR is the closest problem in the literature to WCR, we compare Edge-Index with Sampling-Tree and 2-Label-Hop in terms of index time, index size and average query time. We adapt both methods to handle undirected graphs. Given a query $q(a, b, [x, y])$, we rewrite it into the query format of Sampling-Tree and 2-Label-Hop by forming a categorical label set $A = \{l \in \Sigma | x \leq l \leq y\}$.

Random graphs with uniform weight distribution In this experiment, we generate a set of random graphs by varying the parameters $|E|/|V|$, $|V|$ and $|\Sigma|$. The default values are $|E|/|V| = 1.5$, $|V| = 1000$ and $|\Sigma| = 5$. The edge weights follow a uniform distribution. We report the performance in Table 7.

We vary the density $|E|/|V|$ from 1.0 to 3.0 with a step size of 0.5 and fix $|V| = 1000$ and $|\Sigma| = 5$. The index time, index size and query time of Edge-Index are not affected by the density. In contrast, the index time of Sampling-Tree and 2-Label-Hop increases with the density and is four to five orders of magnitude longer than that of Edge-Index. The index size of 2-Label-Hop is 50 times larger than that of Edge-Index on average. The query time of Sampling-Tree is two orders of magnitude longer than that of Edge-Index, and the query time of 2-Label-Hop is about 20 times longer.

When we vary $|V|$ and $|\Sigma|$, the index time and query time of Edge-Index remain stable, while the index size increases linearly with $|V|$ and $|\Sigma|$. In contrast, the index construction time of Sampling-Tree and 2-Label-Hop increases dramatically. When $|V| \geq 5000$ or $|\Sigma| \geq 10$, Sampling-Tree cannot finish index construction within 12 h. 2-Label-Hop cannot finish index construction within 12 h when $|\Sigma| \geq 15$. The index size of 2-Label-Hop is 150 times larger than that of Edge-Index. The query time of Sampling-Tree is two orders of magnitude longer than that of Edge-Index, and the query time of 2-Label-Hop is 12–23 times longer.

Random graphs with power law weight distribution In this experiment, we test random graphs with weights following a power law distribution with the parameter $\alpha = 2$ [23], the same setting as in [14]. According to the power law weight distribution, only a few weights appear frequently, while the majority of weights appear infrequently. The default values of the parameters are $|E|/|V| = 1.5$ and $|V| = 1000$. Under a power law distribution, the number of distinct weight values that actually appear in a network depends on $|E|$, thus we do not vary $|\Sigma|$ here. Table 8 shows the performance when we vary $|E|/|V|$ and $|V|$. The actual weight number $|\Sigma|$ is listed in the second column of Table 8.

First, we vary the density $|E|/|V|$ from 1.0 to 3.0 with a step size of 0.5 and fix $|V| = 1000$. We observe that the index time and index size of Edge-Index increase slightly, and the query time remains 1 μs. In contrast, the index time of Sampling-Tree and 2-Label-Hop is three to five orders of magnitude longer. The query time of Sampling-Tree is two orders of magnitude longer than that of Edge-Index and the query time of 2-Label-Hop is 16 times longer. When $|E|/|V| = 3.0$, Sampling-Tree cannot finish index construction within 12 h.

When we vary the vertex number $|V|$, the index time and index size of Edge-Index increase by at most 6.5 and 11 times respectively, as $|V|$, $|E|$ and $|\Sigma|$ all increase. The query time remains 1 μs. The index time of Sampling-Tree and 2-Label-Hop is three to four orders of magnitude longer than Edge-Index. Both index time and index size of Sampling-Tree and 2-Label-Hop increase with $|V|$. The query time of Sampling-Tree is three orders of magnitude longer than that of Edge-Index and the query time of 2-Label-Hop is 13 times longer.

Summary In the above experiments, the query time of Edge-Index remains 1 μs in all cases, which is one to three orders of magnitude faster than that of Sampling-Tree and 2-Label-Hop. In addition, both Sampling-Tree and 2-Label-Hop suffer from the index construction efficiency. In [14], the indexing time of Sampling-Tree is $O(n|V||E|\binom{|\Sigma|}{|\Sigma|/2}+n/n_0(|E|+|V|\log|V|))$, which increases exponentially with the label set size $|\Sigma|$, and also increases with $|V|$ and $|E|$. 2-Label-Hop[32] needs to pre-compute the local transitive closure in index construction. That is why, Sampling-Tree and 2-Label-Hop cannot finish index construction within 12 h in many cases, even for very small scale, for example, $|V| = 1000$ and $|\Sigma| = 15$. This demonstrates that both Sampling-Tree and 2-Label-Hop are not efficient to answer the WCR query.

6 Related work

Reachability query on directed graphs has been studied extensively with many algorithms proposed [1,4–7,9,12–17,24–28,31–33]. These algorithms usually design a certain type of coding for graph nodes so that reachability queries can be answered by checking the coding of the involved nodes. The codings include tree cover [1], chain cover [5,13], dual labeling [31], GRIPP index [27], path-tree cover [17], 2-hop cover [4,6,7,25,26], 3-hop [16], randomized interval label-

Table 6 Online performance of disk-based algorithms

	Dependency	Response time (μs)
Naive-Search	Linear with $\|V\|$	1,883–37,277,978
External-MST	Linear with $\|V\|$	1,573–176,901
Balanced-Index	None	23

Table 7 Edge-Index versus Sampling-Tree and 2-Label-Hop, uniform weights (IT in s, IS in MB, QT in μs)

	Edge-Index			Sampling-Tree			2-Label-Hop		
	IT	IS	QT	IT	IS	QT	IT	IS	QT
$\|E\|/\|V\|$									
1	0.01	0.2	1	72	0.1	86	24	4.55	22
1.5	0.01	0.2	1	339	0.1	106	225	9.74	18
2	0.01	0.2	1	627	0.3	193	574	10.88	20
2.5	0.01	0.2	1	737	0.4	158	1757	12.41	13
3	0.01	0.2	1	953	0.6	135	3773	13.81	13
$\|V\|$									
1000	0.01	0.2	1	339	0.1	106	226	9.45	20
2000	0.01	0.4	1	1500	0.3	362	1994	43.59	13
3000	0.01	0.6	1	3630	0.6	454	6299	94.88	12
4000	0.01	0.8	1	14432	1.4	565	17390	182.89	14
5000	0.02	1	1	$>12h$	–	–	34019	277.36	13
$\|\Sigma\|$									
5	0.01	0.2	1	339	0.1	106	228	9.59	23
10	0.01	0.4	1	$>12h$	–	–	28140	102.83	21
15	0.01	0.6	1	$>12h$	–	–	$>12h$	–	–
20	0.02	0.8	1	$>12h$	–	–	$>12h$	–	–
25	0.02	1	1	$>12h$	–	–	$>12h$	–	–

Table 8 Edge-Index versus Sampling-Tree and 2-Label-Hop, power law weights (IT in s, IS in MB, QT in μs)

	$\|\Sigma\|$	Edge-Index			Sampling-Tree			2-Label-Hop		
		IT	IS	QT	IT	IS	QT	IT	IS	QT
$\|E\|/\|V\|$										
1	14	0.02	1	1	125	0.1	311	12	2.90	20
1.5	17	0.02	1.2	1	154	0.4	716	49	3.41	17
2	20	0.02	1.4	1	2376	1.8	837	78	2.96	18
2.5	22	0.02	1.5	1	18078	6.2	663	114	2.68	12
3	24	0.03	1.7	1	$>12h$	–	–	148	2.63	12
$\|V\|$										
1000	17	0.02	1.2	1	154	0.4	716	44	3.45	15
2000	24	0.04	3.4	1	681	1.4	1252	404	15.22	12
3000	30	0.07	6.2	1	2579	5.5	1280	1114	31.61	12
4000	34	0.09	9.6	1	4309	10	3275	3110	61.45	12
5000	38	0.13	13.4	1	6633	17.4	3308	6106	96.03	13

ing [33], and bit vector compression of transitive closure [28].

Most existing algorithms do not consider vertex or edge label constraints except a few recent works [9,10,14,22,32]. Jin et al. [14] studies label-constraint reachability, given a set of categorical edge labels as the constraint. It utilizes the directed maximal weighted spanning tree and sampling techniques to compress the generalized transitive closure for the edge-labeled graphs. The index construction time increases exponentially with the label set size $|\Sigma|$ and thus is not scalable to handle a large label set. Xu et al. [32] solves the same problem by proposing a Dijkstra-like algorithm to compute path-label transitive closure. Mendelzon and Wood [22] proves RPQ, a path query with regular expression constraints, is NP-Hard; Florescu et al. [10] shows CRPQ is a NPC problem; and Fan et al. [9] studies adding a subclass of regular expressions (RQ) to specify the reachability via a path of certain edge types and of a possibly bounded length. Fan et al. [9] proposed two algorithms that answer a query in $O(|V|^2)$ time. One algorithm uses a matrix of shortest distances as index, and the other uses online bi-directional search to answer a query. Jin et al. [15] studies distance-constraint reachability in uncertain graphs and proposes two probabilistic estimators for the probabilistic reachability.

The existing reachability solutions cannot be directly or efficiently applied to answer the WCR query, as they focus on a different problem setting that does not have the total-ordering property on edge labels. In addition, all existing reachability algorithms in the literature are main memory-based algorithms that assume the index resides in the memory and do not consider the I/O cost in query processing. Our paper is the first to design a disk-based I/O-efficient algorithm to answer the WCR query.

7 Conclusion and future work

In this paper, we study a new type of reachability query, called weight constraint reachability WCR, on undirected graphs with real-valued edge or node weights, which is very common and has a wide range of real-world applications. We design two novel index structures for the memory and disk scenarios respectively. To answer a WCR query, we can guarantee $O(1)$ query time with the memory-based index Edge-Index and $O(1)$ I/O cost (exactly four I/Os) with the disk-based index Balanced-Index. Experimental results on real and synthetic graphs demonstrate that both the memory and disk-based approaches answer a query in microseconds with very compact index and efficient index construction. The disk-based algorithm is highly scalable to large networks and I/O-efficient in query processing.

Our work also opens several promising directions for future work on querying weighted graphs.

It is natural to integrate WCR with other existing queries. One, for example, is weight constraint *k-NN*, which reports a ranked list of nodes, each having a valid path to the query node under the weight constraints. The nodes are ranked according to a user-specified distance/similarity metric between the node and the query node, for example, the cosine of their multidimensional features. It is widely applicable in various scientific areas but not trivial to solve. Another one is weight constraint *keyword search*, which is queried on graphs with node labels. Another possibility is to combine WCR with other constraints such as the distance constraint to answer reachability queries.

From the system perspective, it is interesting to explore how to increase the throughput of a WCR querying system by proper caching mechanism under a main memory or disk budget. In addition, designing WCR algorithms for a distributed system and managing to reduce the network communication cost by graph partitioning is also a promising and useful extension.

Finally, maintaining update is also a viable direction, that is, adapting WCR algorithms to handle evolving graphs that allow node/edge insertion, deletion and modification.

Acknowledgments This work is supported by the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411211, 411310 and 419109, and by NSF through grants IIS-0905215, IIS-0914934, CNS-1115234, DBI-0960443 and OISE-1129076, Google Mobile 2014 Program, and KAU grant.

References

1. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: Proceedings of the 1989 ACM SIGMOD international conference on Management of data (SIGMOD 1989), pp. 253–262 (1989)
2. Bebek, G., Yang, J.: PathFinder: Mining signal transduction pathway segments from protein-protein interaction networks. BMC Bioinform. J. **8**, 335 (2007)
3. Bender, M. A., Farach-Colton, M.: The LCA problem revisited. In: LATIN 2000: Theoretical Informatics, volume 1776 of Lecture Notes in Computer Science, pp. 88–94. Springer, Berlin/Heidelberg
4. Bramandia, R., Choi, B., Ng, W.K.: On incremental maintenance of 2-hop labeling of graphs. In: Proceedings of the 17th international conference on World Wide Web (WWW 2008), pp. 845–854 (2008)
5. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In: Proceedings of the 24th International Conference on Data Engineering (ICDE 2008), pp. 893–902 (2008)
6. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P. S.: Fast computing reachability labelings for large graphs with high compression rate. In: Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008), pp. 193–204 (2008)
7. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002), pp. 937–946 (2002)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, New York (2001)

9. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y.: Adding regular expressions to graph reachability and pattern queries. In: Proceedings of the 27th International Conference on Data Engineering (ICDE 2011), pp. 39–50 (2011)
10. Florescu, D., Levy, A.Y., Suciu, D.: Query containment for conjunctive queries with regular expressions. In: Proceedings of the 1998 Symposium on Principles of Database Systems (PODS 1998), pp. 139–148 (1998)
11. Gomory, R.E., Hu, T.C.: Multi terminal network flows. J. Soc. Ind. Appl. Math. **9**, 551–571 (1961)
12. He, H., Wang, H., Yang, J., Yu, P. S.: Compact reachability labeling for graph-structured data. In: Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2005), pp. 594–601 (2005)
13. Jagadish, H.V.: A compression technique to materialize transitive closure. ACM Trans. Database Syst. **15**(4), 558–598 (1990)
14. Jin, R., Hong, H., Wang, H., Ruan, N., Xiang, Y.: Computing label-constraint reachability in graph databases. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD 2010), pp. 123–134 (2010)
15. Jin, R., Liu, L., Ding, B., Wang, H.: Distance-constraint reachability computation in uncertain graphs. Proc. VLDB Endowment (PVLDB 2011) **4**(9), 551–562 (2011)
16. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-HOP: a high-compression indexing scheme for reachability query. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD 2009), pp. 813–826 (2009)
17. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD 2008), pp. 595–608 (2008)
18. Johnsonbaugh, R., Kalin, M.: A graph generation software package. In: Proceedings of the 22nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 1991), pp. 151–154 (1991)
19. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proc. Amer. Math. Soc. **7**(1), 48–50 (1956)
20. Lawder, J.K., King, P.J.H.: Querying multi-dimensional data indexed using the hilbert space-filling curve. SIGMOD Rec. **30**(1), 19–24 (2001)
21. Ma, Q., Steenkiste, P.: On path selection for traffic with bandwidth guarantees. In: Proceedings of the 1997 International Conference on Network Protocols (ICNP 1997), pp. 191–202 (1997)
22. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM J. Comput. **24**(6), 1235–1258 (1995)
23. Newman, M.E.J.: Power laws, pareto distributions and zipf's law. Contemp. Phys. **46**, 323–351 (2005)
24. Roditty, L., Zwick, U.: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In: Proceedings of the 36th annual ACM symposium on Theory of computing (STOC 2004), pp. 184–191 (2004)
25. Schenkel, R., Theobald, A., Weikum, G.: HOPI: an efficient connection index for complex XML document collections. In: Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004), pp. 237–255 (2004)
26. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In: Proceedings of the 21th International Conference on Data Engineering (ICDE 2005), pp. 360–371 (2005)
27. TrißI, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD 2007), pp. 845–856 (2007)
28. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 2011), pp. 913–924 (2011)
29. Viswanath, B., Mislove, A., Cha, M., Gummadi, K.P.: On the evolution of user interaction in facebook. In: Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN 2009), pp. 37–42 (2009)
30. Vitter, J.S.: External memory algorithms and data structures. ACM Comput. Surv. **33**(2), 209–271 (2001)
31. Wang, H., He, H., Yang, J., Yu, P. S., Yu, J. X.: Dual labeling: answering graph reachability queries in constant time. In: Proceedings of the 22th International Conference on Data Engineering (ICDE 2006), pp. 75 (2006)
32. Xu, K., Zou, L., Yu, J. X., Chen, L., Xiao, Y., Zhao, D.: Answering label-constraint reachability in large graphs. In: Proceedings of the 2011 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2011), pp. 1595–1600 (2011)
33. Yildirim, H., Chaoji, V., Zaki, M.J.: GRAIL: scalable reachability index for large graphs. Proc. VLDB Endowment (PVLDB 2010) **3**(1), 276–284 (2010)