World Wide Web (2024) 27:25
https://doi.org/10.1007/s11280-024-01248-5

Efficient processing of coverage centrality queries on road networks

Yehong Xu[1] · Mengxuan Zhang[2] · Ruizhong Wu[3] · Lei Li[1,3] · Xiaofang Zhou[1,3]

Received: 11 January 2023 / Revised: 28 August 2023 / Accepted: 21 November 2023

Abstract
Coverage Centrality is an important metric to evaluate vertex importance in road networks. However, current solutions have to compute the coverage centrality of all the vertices together, which is resource-wasting, especially when only some vertices centrality is required. In addition, they have poor adaption to the dynamic scenario because of the computation inefficiency. In this paper, we focus on the coverage centrality query problem and propose a method that efficiently computes the centrality of single vertices without relying on the underlying graph being static by employing the intra-region pruning, inter-region pruning, and top-down search. We further propose the bottom-up search and mixed search to improve efficiency. Experiments validate the efficiency and effectiveness of our algorithms compared with the state-of-the-art method.

Keywords Coverage centrality · Road networks · Shortest paths

1 Introduction

Centrality computation serves as a fundamental operation in a range of applications within road networks, including traffic monitoring and prediction [1], network maintenance and

✉ Yehong Xu
yxudi@connect.ust.hk

✉ Mengxuan Zhang
mengxuan.zhang@unt.edu

Ruizhong Wu
rwu601@connect.hkust-gz.edu.cn

Lei Li
thorli@ust.hk

Xiaofang Zhou
zxf@ust.hk

[1] The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

[2] Australian National University, 2601 Canberra, ACT, Australia

[3] The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China

Published online: 12 April 2024

assessment [2]. Compared to various metrics of centrality evaluation, *coverage centrality of a vertex s (denoted as CC(s))* [3–5] has a particularly high correlation with s's transportation ability and surrounding traffic condition. This is because CC is defined based on the shortest paths in a road network.

A road network is an undirected weighted graph $G(V, E, W)$ with the vertex set V (i.e., road intersections), the edge set $E \subseteq V \times V$ (i.e., road segments), and cost function $W : E \rightarrow \mathbf{R}^+$ that assigns a non-negative travel cost to each edge $(u, v) \in E$. We denote $n = |V|$, $m = |E|$, and $N(v)$ for the neighbors of $v \in G$. A path $p = \langle v_1, ..., v_k \rangle$ is a sequence of vertices where $(v_i, v_{i+1}) \in E$, $v_i \in V$. The length of a path p is defined as $l(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. Let $p_{s,t}$ denote any path between a vertex pair (s, t). The shortest path $\hat{p}_{s,t}$ is a path among all $p_{s,t}$ with the minimum length. *Coverage centrality (CC)* of one vertex is defined as the number of vertex pairs that have at least one shortest path passes it (as shown in (1)).

$$CC(v) = \sum_{s,t \in V, s \neq t \neq v} \delta_{s,t}(v) \tag{1}$$

where $\delta_{s,t}(v)$ is equal to 1 if at least one shortest path between s and t passing through v, otherwise 0.

In the context of the road network G, the travel cost associated with each edge can be interpreted as the dynamic travel time required to traverse that edge. In this scenario, a high value of $CC(s)$ indicates the importance of vertex s in terms of transportation within the graph G. In other words, the blockage of s can have a profound impact on the overall travel costs within G. Moreover, an increase in $CC(s)$ suggests an improvement in travel time when passing through vertex s, and vice versa. By closely monitoring $CC(s)$, we can obtain valuable insights into the traffic conditions related to vertex s, allowing us to implement more precise traffic management strategies. Therefore, its quick measurement is essential, especially for those urgent situations.

Existing solutions CC has been extensively studied, mainly including two branches of methods. One branch orders CC roughly in proportion with the vertex degree [6] in a hypergraph, which is constructed by the sampled vertices in the original graph. But it does not calculate the centrality directly, so it is out of this paper's consideration. Another branch focuses on *Betweenness Centrality (BC)*. The definition of BC is quite similar to that of CC except that $\delta_{st}(v)$ is defined as the ratio of shortest paths between s, t passing through v, relative to all shortest paths between s, t. The *Brandes* algorithm [7] is the fundamental BC algorithm, whose time complexity is $O(nm + n^2 logn)$ where $n = |V|$, $m = |E|$ are the vertex and edge number, respectively. *Brandes* is computationally expensive for large graphs and thus cannot support real-time query answering. *Subsequently,* there come other strategies that aim at improving the scalability of *Brandes* [8–13]. However, these strategies that either distribute or parallelize the computation could hardly apply to road networks, as analyzed in Section 2.2. Therefore, only *Brandes* could be used and extended to CC computation by ignoring the path number [5].

Motivation The network is dynamic with traffic conditions keep changing [14–19], obtaining vertices' CC values in real-time is quite useful. Typically, we are only interested in monitoring a small set of critical vertices over time, e.g., those that connect different parts of the road network. For other vertices, it is unnecessary to maintain their CC. However, existing BC-based methods that either maintain CC of all vertices or none of them are computationally expensive and cost-ineffective. Then a question comes naturally: why not focus on developing an online CC-answering method for *single* vertices that can be easily adapted to *dynamic*

road networks? Therefore, we aim to propose *Coverage Centrality Query Answering Framework* that relieves the heavy computation involved in *Brandes*-based methods. Instead, it is lightweight and can efficiently answer most *CC* queries in real-time.

Challenges Our problem is how to efficiently answer *CC* queries given a underlying road network $G(V, E, W)$, where a *CC* query is denoted as $q(s)$ $(s \in V)$. According to formula (1), to answer $q(s)$, we need to check every vertex pair (a, b) in G to see whether it has a shortest path that passes through v. We term the checking as the *dependency check* for (a, b). Thus, it takes $t = (n-1)(n-2)\tau$ time in total, where τ is the time of one dependency check. This naive calculation is obviously time-consuming. Approximately, t is around 25 hours for medium-size road network (with around 300, 000 vertices like *New York City* and *Beijing*) with τ in microsecond level. There comes our challenge: how to calculate accurate *CC* values efficiently.

Our idea We focus on speeding up *CC* calculation through reducing the number of vertex pairs that require dependency checks. Given a *CC* query $q(v)$, we initially need to check $(n-1)(n-2)$ vertex pairs. However, we find that some vertex pairs can be pruned: vertex pairs (a, b) that satisfy: $\exists \hat{p}(v, a)$, $\hat{p}(v, b)$ s.t. $\{v\} \subseteq \hat{p}(v, a) \cap \hat{p}(v, b)$. As this means that the concatenation of $\hat{p}(v, a)$, $\hat{p}(v, b)$ contains a cycle so that the shortest must not pass through v and (a, b) can be pruned. The *Shortest Path Tree (SPT)* rooted at v (denoted as T_v, Figure 1) is a perfect structure to organize starting from v to all other vertices. A *SPT* is formally defined below.

Definition 1 (***Shortest Path Tree (SPT)***) The shortest path tree rooted at $v \in V$ denoted as T_v is a spanning tree of G s.t. any simple path from v to another vertex u in T_v corresponds to one of the shortest paths $\hat{p}_{v,u}$ in G.

Spatially, a *SPT* T_v divides the network into multiple cone-shaped regions v [20] where the region denotes descendants of the same child of v in T_v. Vertex pairs (u, v) where u, v belong to the same region can be pruned, and we this as *intra-region pruning*. Nevertheless, many inter-region vertex pairs (i.e., vertex pairs with two endpoints in different regions) remain to check. We discuss their pruning methods in Section 3.3. The next problem is how we should check the unpruned vertex pairs. Our idea is to traverse the *SPT* and check

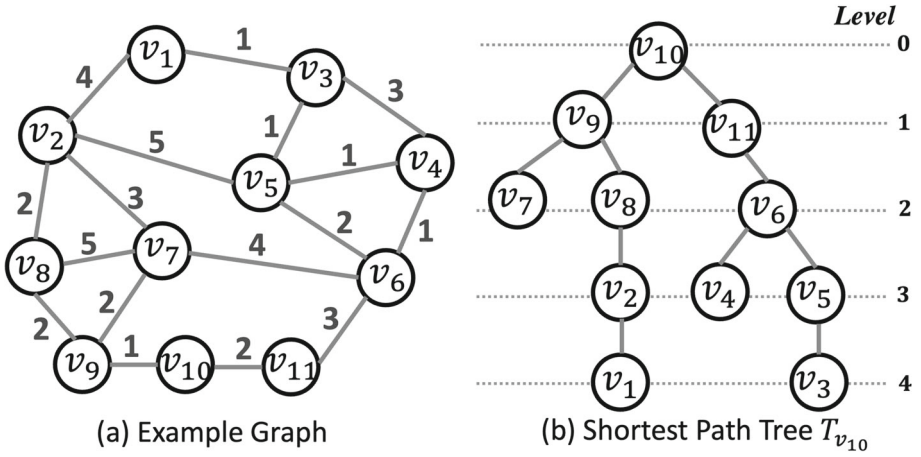

Figure 1 Example graph and its shortest path tree T_{v10}

vertex pairs along the traversal. We first propose the *top-down traversal* strategy, while it still incurs abundant dependency checks for vertices with high *CC* values. Thereby, we propose *bottom-up traversal* and *mixed traversal* for those vertices of high *CC* values.

Contributions To the best of our knowledge, we are the first to study the efficient centrality computation from the aspect of vertex pairs pruning and the first to support accurate centrality computation of single vertex. Our contributions are as follows:

- We utilize the *SPT* as the carrier of centrality calculation and propose *intra-region pruning* to prune those vertex pairs with endpoints in the same regions;
- We further propose *inter-region pruning* to prune those vertex pairs with endpoints in two different regions. Specifically, we propose *SPT* traversal strategies (top-down, bottom-up, and mix) for vertices of different centrality levels;
- We conduct extensive performance studies in real-life road networks, and the experimental results demonstrate the superiority of our approaches.

This work is an extension of [21] where we proposed the *top-down traversal* method. To scale to larger networks, we propose a new pruning technique (Lemma 4) based on previous ones, and further propose two new *bottom-up* and *mixed* traversal methods. New experiments on larger networks demonstrate the effectiveness of our new optimization techniques.

Organization Section 2 gives the related work. Section 3 introduces our computation framework which includes pruning techniques (Sections 3.2 and 3.3) and the top-down search (Section 3.4). Section 4 describes the improvements based on the computation framework, including the better search space management (Section 4.1), the bottom-up search (Section 4.2) and the mixed search (Section 4.3). Section 5 discusses experimental results. Section 6 concludes the entire paper.

2 Related work

The definition of Coverage Centrality *CC* was proposed in [22]. However, the authors did not provide a concrete algorithm for the computation. *CC* is equivalent to Betweenness Centrality *BC* [23, 24] when there is no more than one shortest path passing through one specific vertex for each vertex pair. In this section, we briefly summarize existing works about *CC* and *BC*, along with shortest path algorithms, as shortest path computation is a fundamental operation of our methods.

2.1 Coverage centrality

The only existing method that computes *CC* is what was proposed in [25, 26] used to derive a vertex order for relatively small hub-labeling index size. The authors assume that every shortest path is unique, and their idea is to compute the *SPT* rooted at every vertex, which takes $O(nm + n^2 logn)$ in time using *Dijkstra's* [27]. And the *CC* value of a vertex v is the sum of # descendants of v in all *SPT*s, which can be obtained by running the depth-first search from v over every *SPT*. The time complexity is $O(n^2 + mn)$. The overall time complexity of computing a vertex's *CC* value using this method is $O(nm + n^2 logn)$.

Top-k CC Yoshida [6] intended to find top-k vertices of the largest *CC* value by constructing a hypergraph H which consists of sampled vertices. It uses one vertex's degree in H to

approximate its *CC* value. Specifically, a greedy algorithm is proposed to find those vertices of the highest degree iteratively. In each iteration, the selected vertex and its incident edges are removed from H.

CC maximization *CC* maximization aims to find a set of arcs s.t. their insertions maximize the *CC* value of the target vertex or target group of vertices [28, 29]. It is an NP-hard problem. The proposed solutions prove that the selected edges are guaranteed to benefit the target vertex(s) with a high probability based on certain criteria. Their work does not compute a vertex's *CC* value.

2.2 Betweenness centrality

Exact BC The existing fastest algorithm for exact *BC* computation is *Brandes* [7]. It runs the *Dijkstra's* from each vertex v and constructs the corresponding directed acyclic graph (DAG), which encodes all the shortest paths starting from v. The BC value of one vertex is obtained by accumulating the paths' contributions to it through the backward propagation in each DAG. Its time complexity is $O(nm + n^2 logn)$, meaning it can hardly scale to large graphs.

Then there come several strategies to improve the scalability of *Brandes* by distributing [9, 30] or parallelizing [10, 31–34] the computation. Besides, another approach accelerates *BC* computation by identifying equivalent vertices and only considering one vertex of each equivalent group. Specifically, [8] decomposes the graph into multiple components by removing the articulation vertices. The *BC* value is obtained by summing up vertices' local *BC* values computed within each group. However, road networks in real life usually do not contain any articulation vertex. Daniel et al. [11], Suppa and Zimeo [35] decomposes the graph based on community finding algorithms [12]. Two vertices within the same group are considered equivalent if they have the same shortest distance and # shortest paths to the border vertices. Nevertheless, it is unlikely to happen in weighted graphs. In [13], vertices are treated equivalent if they have the same neighbors, s.t. the input graph can be compressed by contracting equivalent vertices into one super vertex. Then shortest paths within the super vertices and the compressed graph are considered separately in *BC* computation. But the equivalence condition only works in an unweighted graph. In summary, all these algorithms can hardly apply to road networks.

Approximated BC Another line of research scales up *BC* computation through approximation, which essentially trades the accuracy for efficiency by computing BC on sampled vertices [36]. Their main focus is to minimize the sample size but at the same time guarantee the approximation ratio. Riondato and Kornaropoulos [37] employed *Vapnik-Chervonenkis* dimension to greedily explore the tightest upper bound on the sample size [37] that guarantees accurate BC approximations. However, it needs some graph parameters to derive the sample size, such as the graph diameter (the longest shortest distance) which is computationally expensive. Then *Rademacher* average is used s.t. pre-computation of any parameter is not needed [38]. But it has to progressively enlarge the sample size until the guaranteed approximations of BC are obtained, which might drag down the efficiency. In addition, another drawback associated with these approximate methods is that the sample size could still be significant when the input graph is of considerable scale, which means the computation may stay inevitably slow.

Dynamic BC Many approaches update the *BC* values based on *Brandes*. One stream distinguishes potentially affected vertices and updates them by decomposing the graph based

on some data structures, such as independent *minimum union cycles* [39] and *biconnected components* [40]. Nevertheless, it works assuming that the updates would not affect the decomposition; otherwise, the BC must be recomputed from scratch. However, the assumption can hardly be satisfied. Another stream saves the intermediate results generated during *Brandes*'s computation, such as predecessor list and DAGs, to maintain the centrality value [41, 42]. When an update comes, the intermediate results are updated and recalculated for those affected BC values by identifying the changed shortest paths. However, they all have the same time complexity as that of *Brandes* and consume significant space. Besides, there have been many works in real-time graph analytics that accelerate different graph-based queries on large volume and high velocity streaming graphs [43, 44]. However, they do not directly support *BC* queries.

Top-k BC Besides computing BC values, comparing them and identifying top-k vertices with the highest values are of wide application. Because of the expensive computation of the existing BC algorithm *Brandes*, alternative centrality metrics similar to BC are proposed [45–47]. For instance, Lee et al. [48] made use of the novel property of biconnected components to derive the upper-bound of BC value of each vertex. But a road network can hardly be decomposed into multiple biconnected components as aforementioned. Fan et al. [49] proposed a deep learning method to learn the structural importance of vertices. However, it only applied to unweighted graphs.

2.3 Computing shortest paths

The shortest path algorithm is the building block of centrality computation. This paper uses it to compute the shortest distance between one given vertex pair. The fundamental shortest path algorithm is *Dijkstra's* [27], which finds the shortest path in *breadth-first search* manner with time complexity $O(m + nlogn)$. Then $A*$ [50, 51] is proposed to speed up it by directing the search space towards the destination. Nevertheless, both are not efficient enough, especially for a large graph. Then auxiliary information is stored to accelerate the computation, including shortcuts in *Contraction Hierarchy (CH)* [52] and labels in *Hub Labeling (HL)* [53, 54]. To be specific, *CH* computes the shortest path by traversing vertices in a bottom-up manner bidirectionally. *HL* calculates the shortest distance by summing up the distance label value and takes the minimum one as the shortest distance without graph traversal. Therefore, *HL* is generally more efficient in shortest distance computation, and we use it as our shortest path index.

3 Coverage centrality query answering framework

In this section, we introduce how to calculate the *CC* of a vertex s using the *SPT* rooted as s (i.e., T_s). We first demonstrate the computation framework (Section 3.1) and the associated characteristics that accelerate *CC* computation (Sections 3.2, 3.3). We then propose the algorithm in Section 3.4 that utilized the characteristics given the framework.

3.1 Overview

Given a road network G and a CC query $q(s)$, our framework employs (1) T_s, (2) a *HL*-based shortest distance index $\mathcal{L}$ of G and (3) a *CC* computation algorithm based on T_s. This

is because based on (1), obtaining *CC(s)* needs to perform abundant dependency checks. The time complexity to perform a dependency check is $O(nm + nlogn)$ using *Dijkstra's* algorithm [27] while a shortest distance algorithm *SPA* based on $\mathcal{L}$ (e.g. [54]) can improve the efficiency to $O(\tau)$ ($\tau \leq n$). Therefore, our framework employs $\mathcal{L}$ as a core component. Despite this, the time complexity of answering $q(s)$ is $O(n^2\tau)$ which remains impractical for real-time *CC* answering in a large road network. To address this problem, our framework further employs T_s as its structural characteristics help us directly identify the dependencies of many vertex pairs on s without the need for any computation, which are summarized as Lemmas 1 and 2. Thereby, we can reduce abundant dependency checks. We further propose a *CC* computation algorithm based on T_s in Section 3.4 to implement our idea.

Note that our *CC* computation method does not rely on the underlying graph being static, because we only borrow $\mathcal{L}$ and we assume that $\mathcal{L}$ always provided and kept up-to-date. We do not build any index ourselves, the 'index-free' nature of our method allows it to answer *CC* queries in dynamic graphs. When a *CC* query is issued, our method processes *CC* queries in the same manner regardless of whether the underlying road network is static or dynamic. Specifically, given a road network G and a query vertex s, we first construct T_s using *Dijkstra* algorithm [27] in real-time, then invoke a *CC* computation algorithm (e.g., Algorithm 1). Although these algorithms employ a shortest distance index $\mathcal{L}$ which requires updates when changes occur in G, the efficient maintenance of $\mathcal{L}$ falls outside the scope of this paper, and has been addressed in [46]. The time complexity of *Dijkstra* and Algorithm 1 is $O(m + nlogn)$ time and $O(n^2\tau)$. Therefore, the time complexity of our framework is $O(n^2\tau)$.

3.2 Intra-region pruning

Our framework employs T_s due to the underlying pruning power. In this section, we introduce the *intra-region* pruning strategy which predicates that *intra-region* vertex pairs of T_s must not depend on s and therefore do not need to be checked. Let us first define *region* as below.

Definition 2 (***Region***) Given the *SPT* T_s rooted at s, a region denotes the whole set of vertices in a subtree $T_s(v)$, $v \in cld(s)$.

Algorithm 1 Top-down traversal.

```
   Input: G, s
   Output: CC(s)
1  T_s ← ConSPT(G, s);                          ▷ Construct SPT rooted at s
2  CC(s) = 0; H ← {(u, v, 0)|∀u, v ∈ cld(s), u ≠ v};
3  while H is not empty do
4      u, v, mov ← H.pop();
5      dep ← DC(u, v, s);                       ▷ Dependency check
6      if dep then
7          CC(s) ← CC(s) + 1;
8          if mov = 0 then
9              foreach w ∈ cld(u) do
10                 H.insert((w, v, 0));
11         foreach w ∈ cld(v) do
12             H.insert((u, w, 1));
13 return CC(s);
```

where $T_s(v)$ denotes the subtree of T_s rooted at v and $cld(s)$ denotes the children tree nodes of s in T_s.

Let M_v be the number of v' children tree nodes, i.e. $M_v = |cld(v)|$. The vertex set V of G is partitioned into regions $T_s^1, \ldots, T_s^{M_s}$. Particularly, we call a vertex pair (u, w) a *intra-region* vertex pair if u and w are located in the same region or otherwise a *inter-region* pair. All intra-region vertex pairs can be safely pruned from dependency checking as they definitely do not depend on s according to the following lemma.

Lemma 1 *For any vertex pair in the same region of T_s, i.e. $u, w \in T_s(v)$ $(v \in cld(s))$, then $s \notin \hat{p}_{u,w}$.*

Proof We can prove it by contradiction. Assume that there exists a shortest path p_1 between u, w in G that contains s. Then the length of p_1 must equal $l(\hat{p_{u,s}}) + l(\hat{p_{s,w}})$. Meanwhile, since u and w are in the same subtree $T_s(v)$, there must exist a path p_2 between u, w that is the concatenation of $\hat{p}_{u,v}$, $\hat{p}_{v,w}$. Given that $l(\hat{u,v}) < l(\hat{u,s})$ and $l(\hat{v,w} < \hat{v,s})$, we have $l(p_2) < l(p_1)$. Thus, p_1 cannot be the shortest path between u, w; this contradicts our assumption. We proved that $s \notin \hat{p}_{u,w}$. □

Suppose # vertices in each region of T_s are $n_1, ..., n_{M_s}$, then there are $\sum_{1 \le i \le M_s} n_i^2$ vertex pairs being pruned from the dependency checking.

3.3 Inter-region pruning

Following the *intra-region pruning*, only the *inter-region* vertex pairs were left for dependency check. That is, we only need to check the vertex pair (u, v) with $u \in T_s^i, v \in T_s^j$ $(1 \le i < j \le M_s)$. How could we check those $\sum_{1 \le i < j \le M_s} n_i \times n_j$ vertex pairs efficiently? The naive solution is to check them one by one; however, some vertex pairs could avoid being checked if their *parent vertex pair*s (defined as follows) do not have the shortest path passing s, as illustrated in Lemma 2.

Definition 3 **(*Parent Vertex Pair*)** Given a vertex $v \in V$, we use $v.p$ to denote the parent of v in T_s. For a vertex pair(u, v), its right parent vertex pair is $(u, v.p)$, and its left parent vertex pair is $(u.p, v)$.

Lemma 2 *s does not constitute any shortest path between u, v (i.e., (u, v) does not* depend on *s) if the left (or right) parent vertex pair of (u, v) does not.*

Proof Without loss of generality, suppose the right parent vertex pair $(u, v.p)$ does not depend on s. We denote $p_{u,v.p}$ as a path concatenated by $\hat{p}_{u,s}$, $\hat{p}_{s,v.p}$, and $p_{u,v}$ is concatenated by $\hat{p}_{u,s}$, $\hat{p}_{s,v.p}$ and the edge $(v.p, v)$. Clearly, $p_{u,v.p}$ is a subpath of $p_{u,v}$. Since $p_{u,v.p}$ is not the shortest path, $p_{u,v}$ cannot be either, and the shortest paths between (u, v) do not pass s. □

3.4 Top-down traversal

Given Lemma 2, for a vertex pair (u, v), we could first check its parent vertex pairs. If the checked parent vertex pair depends on s, we then check (u, v); otherwise, (u, v) can be pruned. Thereby, we propose to check the *inter-region* vertex pairs by traversing the *SPT* in a top-down manner, and we call this way of vertex pair checking as *top-down traversal*. And we use *transmission route* to embody the traversal and define it as follows.

Definition 4 (***Transmission Route***) A transmission route Tra is to encode the sequence of movements to get from an inter-region vertex pair (u, v) ($u \in T_s^i, v \in T_s^j$ and $i < j$) to its descendant vertex pairs, where each movement is a downward hop in T_s^i or T_s^j via an edge in T_s. Without loss of generality, we use 0 (resp. 1) to denote the downward hop in T_s^i (resp. T_s^j).

For example, $(v_9, v_{11}) \xrightarrow{Tra} (v_7, v_{11}), (v_8, v_{11})$ with $Tra = \{0\}$ and $(v_9, v_{11}) \xrightarrow{Tra} (v_7, v_6), (v_8, v_6)$ with $Tra = \{10, 01\}$.

For the inter-region vertex pairs across two regions T_s^i, T_s^j ($i \neq j$), we initialize the traversal by first checking the vertex pair (v_i, v_j) where v_i (resp. v_j) is the root of T_s^i (resp. T_s^j), then push down the traversal by checking its *child vertex pairs* $\{(v_i', v_j)|\forall v_i' \in cld(v_i)\}$ and $\{(v_i, v_j')|\forall v_j' \in cld(v_j)\}$. It can be easily seen that every vertex pair between the two regions have the chance to be checked. Nevertheless, some vertex pairs could be checked more than once according to the following Lemma 3, which results in a tremendous redundant computations.

Let $h_s(v)$ denote the level of v in T_s ($h_s(s) = 0$).

Lemma 3 *Suppose that the dependency check on the vertex pair (u, v) ($u \in T_s^i, v \in T_s^j$ and $i \neq j$) propagates to vertex pair (u', v') with $h_s(u') - h_s(u) = a, h_s(v') - h_s(v) = b$, then (u', v') could be repeatedly checked for C_{a+b}^a times.*

Proof To traverse (u', v') downwards from (u, v), the movements in transmission route $(u, v) \xrightarrow{Tra} (u', v')$ contains a 0s and b 1s. The order of movements in T_s^i or T_s^j does not make any difference. Therefore, there are C_{a+b}^a different transmission routes s.t. (u', v') would be repeatedly checked for C_{a+b}^a times. □

The question then becomes how to skillfully avoid redundant dependency checks without the need to label if a vertex pair has been checked already.

Theorem 1 *For $(u, v) \xrightarrow{Tra} (u', v')$ with $h_s(u') - h_s(u) = a, h_s(v') - h_s(v) = b, a + b > 1$, if we exclude all Tras that contain the subsequence "10", then there left only one Tra.*

Proof We only need to prove that only one Tra exists that does not contain any subsequence "10". This is intuitively right because Tra can only be in the format $Tra = \{0\}^i\{1\}^j$ where $i, j \geq 0, i + j = a + b$. □

For example (Figure 1), in $(v_9, v_{11}) \xrightarrow{Tra} (v_2, v_6)$, Tra could be 100, 010, 001}. However, after applying the "10" restriction, Tra can only be 001.

We could avoid the repetitive dependency checks by complying with Theorem 1. Given Lemma 2, the traversal terminates when we finish checking child vertex pairs of vertex pairs that are visited by us depending on s. With all these theoretical directions, we illustrate our *top-down traversal* in Algorithm 1. Specifically, we first construct a shortest path tree T_s (line 1) and initialize a heap H by inserting into a triple $(u, v, 0)$ with $u, v \in cld(s)$ and $u \neq v$ (line 2). $DC(u, v, s)$ is the dependency check function and we use dep to denote the result. dep is true if s depends on (u, v); otherwise false. If s depends on (u, v), we insert its child vertex pair into H with the compliance to Theorem 1 (line $6 - 12$).

For example, given the query vertex v_{10}, the example graph G, and the *SPT* in Figure 1. Figure 2 shows all inter-region vertex pairs in T_{10} with movement choices (0 or 1 in blue color) of their parents. The *top-down traversal* strategy (Algorithm 1) first initializes the heap H with $(v_9, v_{11}, 0)$. Then it iteratively checks the dependency of vertex pairs (u, v) in

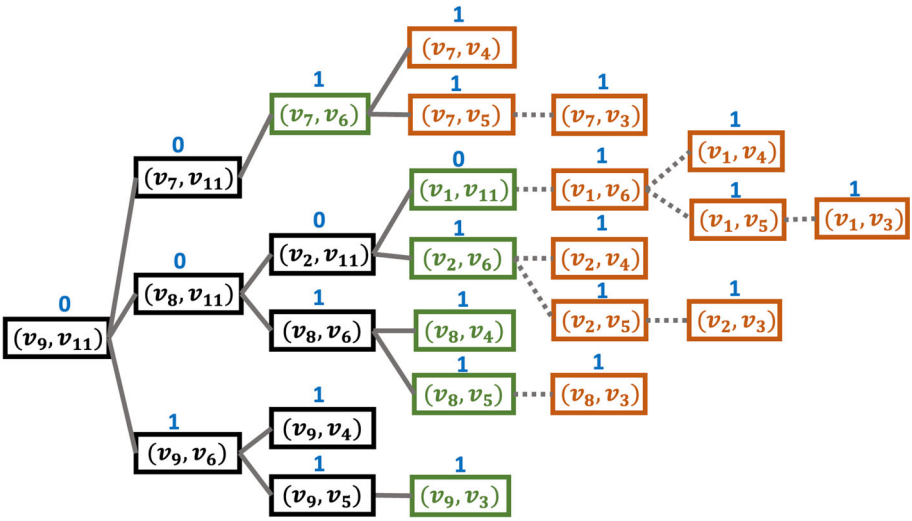

Figure 2 Illustration of the *top-down traversal* strategy implemented with Algorithm 1

H until H is empty. If (u, v) depends on v_{10} (denoted in black color), the algorithm first increments $CC(v_{10})$ by 1, then checks the movement label (mov) of (u, v). If $mov = 0$, all child vertex pairs of (u, v) are inserted into H (e.g. $(u, v) = (v_9, v_{11})$); whereas if $mov = 1$, only (u, w) $(\forall v \in cld(v))$ are inserted into H (e.g. $(u, v) = (v_9, v_6)$). For a vertex pair that does not depend on v_{10} (denoted in green color), none of its child vertex pairs is inserted into H (e.g., (v_2, v_{11})). These vertex pairs are denoted in orange color. Compared to performing dependency checks of (11-1)(11-2) = 90 vertex pairs in G, we only need to check 14 vertex pairs. Specifically, Lemma 1 prunes 40 vertex pairs (e.g. (v_9, v_7), (v_7, v_1), $(v_1 1, v_6)$), and Lemma 2 prunes 36 vertex pairs (e.g., those in orange color in Figure 2).

We can find that given $q(s)$, *top-down traversal* (Algorithm 1) has to check every vertex pair that depends on s. Namely, the number of dependency checks incurred by *top-down traversal* is at least $CC(s)$ which is infeasible for vertices with large CC values. This makes us think about other solutions that require fewer dependency checks.

4 Improved algorithms

Top-down traversal checks a parent vertex pair before its child vertex pairs. However, this can be unnecessary. Given Lemma 4 (introduced below), if we know a vertex pair (u, v) depends on the query vertex s, then its parent and ancestor vertex pairs must also depend on s.

Lemma 4 *Given a vertex pair (u, v) that depends on s, the left and right parent vertex pairs (if they exist) of (u, v) must also depend on s.*

Due to Lemma 4, an intuitive idea to improve *top-down traversal* is to start dependency check with medium-level vertex pairs instead of vertex pairs at level 1. To implement this idea, we can let all inter-regions vertex pairs initially labeled as "unpruned" and continuously check vertex pairs until none of them is labeled as "unpruned". Assume that the vertex pair (u, v) is under checking; if it depends on s, we first select a "unpruned" descendant vertex

pair of (u, v) as the next vertex pair to check, then mark all ancestor vertex pairs of (u, v) as "pass"; otherwise, we select an "unpruned" ancestor vertex pair of (u, v) and label all descendent vertex pairs of (u, v) as "dnt". Finally, we count # vertex pairs labeled as "pass" which is the value of $CC(s)$.

4.1 Search space management

The "unpruned" vertex pairs can be regarded as the dynamic search space when processing a *CC* query. Compared to the *top-down traversal* strategy, flexible checking methods may incur fewer dependency checks but have an overhead in maintaining the search space, which becomes our new problem. Here, we propose decomposing T_s into branches and managing search space in units of branch pairs instead of vertex pairs.

Definition 5 (***SPT Decomposition***) Given a query vertex s, *SPT* decomposition (Algorithm 2) turns T_s into a set of disjoint branches $\mathcal{B}$ s.t. every vertex $v \in V$ belongs to exactly one branch $B \in \mathcal{B}$. Specifically, each branch $B \in \mathcal{B}$ is a sequence of vertices $B = \langle v_0, ..., v_k \rangle$ that satisfy: (1) $v_0.p$ is either s or have more than 1 child (i.e. $|cld(u)| > 1$); (2) v_k is a leaf node in T_s; and (3) $\forall 0 \leq i < k, v_{i+1} = v_i.p$.

Consider a branch $B(x)$ starting with vertex x and an integer i $(0 \leq i < |B(x)|)$. Denote by $B(x)[i]$ the i-th vertex $v_i \in B(x)$ with $h(v_i) = h(x) + i$ $(i < |B(x)|)$. Denote by $B(x)[0, i]$ the vertices $v \in B(x)$ s.t. $h(x') \leq h(x) + i$. For example, given $B(v_{11}) = \langle v_{11}, v_6, v_5, v_3 \rangle$, $B(v_{11})[2] = v_5$, $B(v_{11})[0, 2] = \{v_{11}, v_6, v_5\}$.

Algorithm 2 SPT decomposition.

```
   Input: s, G
   Output: B
1  B ← φ;
2  GetBranches(s, G, B)                ▷ design of GetBranches is explained below;
3  return B;
4  Function GetBranches(s, G, &B)
5      T_s ← ConSPT(G, s), B ← φ;
6      FindSegs(s, B, B);
7  Function FindSegs(u, &B, &B)
8      if u ≠ s then
9          B ← B ∪ u;
10     foreach v ∈ cld(u) do
11         FindSegs(v, B, B);
12     if B ≠ φ then
13         B ← B ∪ B, B ← φ;
```

The time complexity of Algorithm 2 is $O(m + n)$. In addition to decomposing the *SPT* into disjoint branches, we give each branch a unique id, and for each branch, we record the ids of its parent and child branch(s). Let $B(u)$ $(u \in V)$ denote a branch starting at u. The parent branch $B(u).p$ of $B(u)$ satisfies that $u.p \in B(u).p$.

Definition 6 (***Parent Branch Pair***) Given a branch pair (B_i, B_j), its left (resp. right) parent branch pair $(B_{i'}, B_{j'})$ satisfies that $B_{i'} = B_i.p$ and $B_{j'} = B_j$ (resp. $B_{i'} = B_i$ and $B_{j'} = B_j.p$).

For example, given an *SPT* $T_{v_{10}}$ in Figure 1(b). After *SPT* decomposition, we obtain the following branches: $B(v_{11}) = \langle v_{11}, v_6, v_5, v_3\rangle$, $B(v_4) = \langle v_4\rangle$, $B(v_9) = \langle v_9, v_8, v_2, v_1\rangle$, $B(v_7) = \langle v_7\rangle$, where $B(v_4).p = B(v_{11})$, $B(v_7).p = B(v_9)$.

The size of a branch must be smaller than the height of the corresponding *SPT*, which is $O(logn)$. Thus, the number of branches is $O(\frac{n}{logn})$. If we manage search space in units of branch pairs, the size of search space becomes $O((\frac{n}{logn})^2)$, which is much less than $O(n^2)$ vertex pairs.

Algorithm 3 illustrates how to maintain the search space in units of branch pairs. Algorithm 3 is quite similar to Algorithm 1 in that the former one also (1) uses a heap H to store 'objects' to be checked, (2) visits a parent before their children 'objects', and (3) uses the same strategy to avoid repetitively inserting same 'objects' into H. What is new in Algorithm 3 is that (1) 'objects' in Algorithm 3 are branch pairs, not vertex pairs; (2) each branch pair is associated with a 'local search space' which is determined by corresponding *Deepest Depending Level* (*DDL*).

Definition 7 (***Deepest Depending Level (DDL)***) Given a query vertex s, a vertex u and a branch $B(v)$ ($s \neq u \neq v$), the deepest depending level *DDL* $DDL(u, v)$ of u is an integer i ($0 \leq i < |B(v)|$) used to refer to the vertex $B(v)[i]$ which satisfies: (1) $(u, B(v)[i])$ depends on s; and (2) if $i < |B(v)| - 1$, $(u_i, B(v)[i + 1])$ must not depend on s.

For example, given the example graph, the *SPT* $T_{v_{10}}$ in Figure 1, and the branches $B(v_{11})$, $B(v_7)$ where $B(v_{11}) = \langle v_{11}, v_6, v_5, v_3\rangle$ and $B(v_7) = \langle v_7\rangle$, the *DDL* of v_7 is 0. Because (v_7, v_{11}) depends on v_{10} while (v_7, v_6) does not, we have $DDL(v_7, v_{11}) = 0$; similarly, $DDL(v_{11}, v_7) = 0$. Whereas since (v_7, v_6) does not depend on v_{10}, v_6 does not have a valid *DDL* regarding $B(v_7)$, denoted by $DDL(v_6, v_7) = -1$.

The local search space of a branch pair $(B(u), B(v))$ represented as a pair of integers (lft, rgt) denotes the set of vertex pairs $\{(u', v')| \forall u' \in B(u)[0, left], v' \in B(v)[0, rgt]\}$ whose dependencies are unknown to us and needed to be checked. In other words, vertex pairs outside the range are pruned. The default local search space of a branch pair $(B(u), B(v))$ is $(|B(u)|-1, |B(v)|-1)$, but we want to reduce it as much as possible. Suppose that the *DDL* of u's parent $u.p$ regarding $B(v)$ is i ($0 \leq i < |B(v)|$) i.e., $DDL(u.p, B(v)) = i$. Given Lemma 2, the search space of $(B(u), B(v))$ can be confined to $(|B(u)| - 1, i)$. Enlightened by this, we make Algorithm 3 traverse branch pairs in a top-down order to obtain *DDLs* of vertices in a parent branch pair before determining the local search space of its child branch pairs.

Now we are ready to introduce Algorithm 3 in detail. It first decomposes T_s to get disjoint branches $\mathcal{B}$ (line 1) and initializes a heap H with penta-tuples $(B(u), lft, B(v), rgt, 0)$ with $u, v \in cld(s)$ and $lft = |B(u)| - 1, rgt = |B(v)| - 1$. Then it enters the main loop (lines 3-19). In each iteration, it first removes a penta-tuple $(B(u), lft, B(v), rgt, mov)$ from H (line 4); then, a *DDL* computation algorithm is called to compute *DDLs* of vertices in $B(u)[0, lft]$ and $B(v)[0, rgt]$ (line 5). The remaining is to retrieve child branch pairs of $(B(u), B(v))$ and determine their local search space (line 6-19). Specifically, for each vertex $u_i \in B(u)[0, lft]$, we get *DDL* j of u_i regarding $B(v)$. Then, the local search space of child branch pairs $(B(w), B(v))$ ($w \in cld(u_i)$) can shrink to $(|B(w)| - 1, j)$. Similarly, for each vertex $v_i \in B(v)[0, rgt]$, given the *DDL* of v_i regarding $B(u)$ is j, the local search space of child branch pairs $(B(u), B(w))$ ($w \in cld(v_i)$) becomes $(j, |B(w)| - 1)$.

Theorem 2 *Algorithm 3 correctly manages the search space during processing a CC query.*

Proof To prove Theorem 2, we need to show that (1) every pruned vertex pair does not depend on query vertex s; and (2) every branch pair is at most inserted to H once. We prove (1) by

Algorithm 3 CC computation paradigm over decomposed SPT.

Input: s, G
Output: $CC(v)$
$\mathcal{B} \leftarrow SPTDec(s, G)$; ▷ *SPT* decomposition
$H \leftarrow \{(B(u), |B(u)| - 1, B(v), |B(v)| - 1, 0) \mid \forall u, v \in cld(s), u \neq v\}$;
while $H \neq \phi$ **do**
  $B(u), lft, B(v), rgt, mov \leftarrow H.pop()$;
  ComputeDDLs($s, B(u), lft, B(v), rgt$); ▷ Call Algorithm 4 or Algorithm 5;
  if $mov = 0$ **then**
    foreach $u_i \in B(u)[0, lft]$ **do**
      $j \leftarrow DDL(u_i, v)$;
      if $j \neq -1$ **then**
        Continue;
      foreach $w \in cld(u_i)$ **do**
        $H.insert(B(w), |B(w)| - 1, B(v), j, 0)$;
  foreach $v_i \in B(v)[0, rgt]$ **do**
    $j \leftarrow DDL(v_i, u)$;
    if $j \neq -1$ **then**
      Continue;
    $CC(s) \leftarrow CC(s) + j + 1$;
    foreach $w \in cld(v_i)$ **do**
      $H.insert(B(u), j, B(w), |B(w)| - 1, 1)$;
return $CC(s)$;

contradiction. Suppose that a vertex pair (x, y) depends on s but is pruned. Then either the branch pair $(B(u), B(v))$ s.t. $x \in B(u), y \in B(v)$ is not inserted into H, or (x, y) is not included in the local search space of $(B(u), B(v))$. Both cases ensure that (x, y) does not depend on s, which contradicts our assumption. Whereas (2) can be proved by Theorem 1. □

Algorithm 3 allows the pruning information to spread and advance through the parent-child relationships between branch pairs. Another advantage of Algorithm 3 is that it confines dependency checks within each branch pair, simplifying our problem to *DDL* computation over a branch pair. In the following paragraphs, we propose two *DDL* computation strategies for branch pairs, i.e., the *bottom-up traversal* strategy and *mixed traversal* strategy.

4.2 Bottom-up traversal

This section introduces the *bottom-up traversal* strategy designed for vertices with high *CC* values.

Lemma 2 guarantees the continuity between vertex pairs in T_s that depend on vertex s. In other words, for a vertex v with a large $CC(v)$, only a few vertex pairs do not depend on v, and these vertex pairs must reside at bottom levels in T_v. The *top-down traversal* strategy that is based on Lemma 2 starts digging into T_v from level-1 inter-region vertex pairs. It does not stop until finds vertex pairs at bottom levels that do not depend on v. This strategy needs to check dependencies of at least $CC(v)$ vertex pairs, which is infeasible in practice.

Conversely, based on Lemma 4, given a query vertex v with large $CC(v)$, we can start with checking inter-region vertex pairs at the bottom level of T_v. We recursively check the parent vertex pair(s) of current vertex pair (u, v) under checking and do not stop until (u, v) *does* depend on v. We denote this strategy as the *bottom-up traversal* strategy.

Algorithm 4 illustrates the *bottom-up traversal* strategy in detail. Given the query vertex s, the branch pair $(B(u), B(v))$ and its local search space lft, rgt, Algorithm 4 first checks the dependency of $(B(u)[lft], B(v)[rgt])$. If the vertex pair (x, y) under checking does not depend on s, the algorithm checks their parent vertex pairs (line 6-8); else, it updates *DDL*s of x, y (line 10-13). This step is conducted recursively until no more vertex pair is checked. Then, the algorithm determines *DDL*s of vertices that have not been decided (lines 14-19). The time complexity of Algorithm 4 is $O(lft \times rgt)$.

Lemma 5 *Given a branch pair* $(B(u), B(v))$ *and two vertices* $u_i, u_j \in B(u)$ *where* $u_j = u_i.p$, *there must exist* $DDL(u_j, v) \geq DDL(u_i, v)$.

Theorem 3 *Algorithm 4 correctly determines DDLs of vertices in the given branch pair.*

Proof We can prove this by contradiction. Consider a branch pair $(B(u), B(v))$, its local search space (lft, rgt), and the query vertex s. Without loss of generality, suppose $B(u)[i]$ is the first vertex whose $DDL(B(u)[i], v)$ is mistakenly calculated by Algorithm 4. There are two cases:

Case 1: The wrong value j of $DDL(B(u)[i], v)$ is greater than its true value j'. Because $(B(u)[i], B(v)[j])$ does not depend on s, line 11 is never executed. So, if $i < lft$, $DDL(B(u)[i], v) \leftarrow j$ is executed at line 16. Namely, j is the value of $DDL(B(u)[i+1], v)$. However, based on Lemma 5, $j' > DDL(B(u)[i+1], v) = j$ which contradicts our assumption. If $i = lft$, then j' remains the default value -1, which is not greater than j', contradicting our assumption.

Case 2: j is smaller than j'. This would happen only if the vertex pair $(B(u)[i], B(u)[j'])$ is not inserted into H. However, this means that $(B(u)[i], B(v)[j'+1])$ depends on s which contradicts our assumption. □

Algorithm 4 Bottom-up traversal.

```
   Input: s, B(u), lft, B(v), rgt
   Output: DDLs of vertices in B(u), B(v)
1  DDL(x, v) ← −1, ∀x ∈ B(u); DDL(x, u) ← −1, ∀x ∈ B(v);
2  H ← {B(u)[lft], B(v)[rgt], 0};
3  while H is not empty do
4      x, y, mov ← H.pop();
5      dep ← DC(x, y, s)                                  ▷ Dependency check
6      if not dep then
7          if mov = 0 then
8              H.insert(x.p, y, 0);
9          H.insert(x, y.p, 1);
10     else if DDL(x, v) < h(y) − h(v) then
11         DDL(x, v) ← h(y) − h(v);
12     else if DDL(y, u) < h(x) − h(u) then
13         DDL(y, u) ← h(x) − h(u);
14 foreach i from lft − 1 to 0 do
15     if DDL(B(u)[i], v) = −1 then
16         DDL(B(u)[i], v) ← DDL(B(u)[i + 1], v);
17 foreach i from rgt − 1 to 0 do
18     if DDL(B(v)[i], u) = −1 then
19         DDL(B(v)[i], u) ← DDL(B(v)[i + 1], u);
```

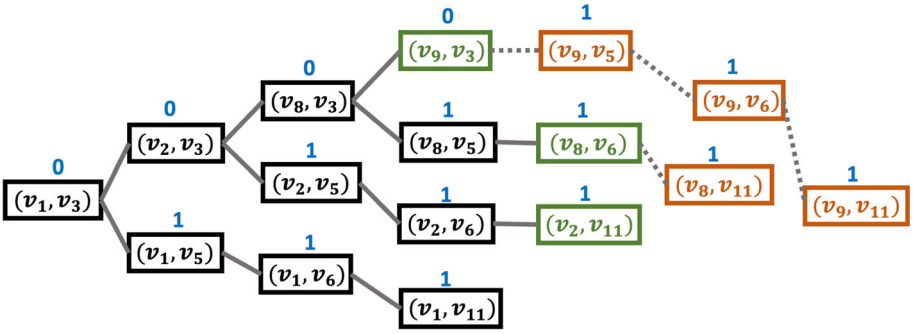

Figure 3 Illustration of *bottom-up traversal* strategy implemented with Algorithm 4

Figure 3 shows an example of running Algorithm 4. Consider two branches $B(v_{11}) = \langle v_{11}, v_6, v_5, v_3\rangle$, $B(v_9) = \langle v_9, v_8, v_2, v_1\rangle$ of the *SPT* in Figure 1, and suppose we are counting the number of vertex pairs (u, v) where $u \in B(v_{11})$ and $v \in B(v_9)$ that depend on v_{10}. Given the input v_{10}, $B(v_9)$, 3, $B(v_{11})$, 3, the algorithm first initializes the heap H with $(v_1, v_3, 0)$. In the main loop, since (v_1, v_3) does not depend on v_{10}, we insert its parent vertex pairs (v_2, v_3) and (v_1, v_5) in H. When (v_2, v_3) is removed from H, we insert both its parent vertex pairs into H as (v_2, v_3) is associated with $mov = 0$ (written in blue). Whereas when (v_1, v_5) is removed from H, we only insert one of its parent vertex pairs (v_1, v_6) into H, as which is associated with $mov = 1$. The same operations are repeated on every vertex pair removed from H that does not depend on v_{10} (denoted by black color). As for a vertex pair, e.g., (v_9, v_3) that depends on v_{10} (denoted by green color), we do not insert any of its parent vertex pairs into H. Instead, we set their *DDL*s, e.g. $DDL(v_9) \leftarrow 3$ and $DDL(v_3) \leftarrow 0$. When Algorithm 4 breaks from the while loop, all vertex pairs in Figure 3, except those written in orange, have once been inserted H. Besides, *DDL*s of v_9, v_3, v_8, v_6, v_2, v_{11} have been determined in the main loop. And $DDL(v_5)$ is set to 0 in line 19. Compared to performing dependency checks of 4×4 vertex pairs in $B(v_{11})$, $B(v_9)$, we only need to check 12 vertex pairs due to Lemma 4.

4.3 Mixed traversal

The *top-down traversal* strategy is only suitable for vertices with extremely small *CC* values, as it is based on the heuristic that only a few vertex pairs at top levels of T_s depend on the query vertex s; in contrast, the *bottom-up traversal* strategy is only suitable for vertices with extremely large *CC* values, as which is based on the heuristic that only a few vertex pairs at bottom levels of T_s do not depend on s. However, it is usually difficult to predict if a vertex has an extremely large or small CC value. In this case, we do not know which strategy should be used. Besides, most vertices have medium-level CC values that have no heuristic about vertex pairs from which levels in T_s begin not to depend on the query vertex T_s.

This section introduces the *mixed traversal* strategy that computes DDLs for vertices in the given branch pair without being based on any heuristic. To quickly find their DDLs, the *mixed traversal* strategy fully exploits both Lemmas 2 and 4. Given $(B(u), B(v))$, the local search space (lft, rgt) and the query vertex s. Suppose (u_i, v_{j-1}) depends on s and we are checking the dependency of (u_i, v_j) where u_i (resp. v_j) is the ith (resp. jth) vertex of $B(u)$ (resp. $B(v)$). If (u_i, v_j) depends on the query vertex s, the *mixed traversal* strategy continues

to find $DDL(u_i, v)$ by checking (u_i, v_{j+1}) (if $j < rgt$). If (u_i, v_j) does not depend on s, based on Lemma 2, (u_{i+1}, v_j) cannot depend on s; based on Lemma 4, (u_{i-1}, v_j) must depend on s. However, the dependency of (u_{i-1}, v_{j+1}) remains unknown. Therefore, the *mixed traversal* strategy then checks (u_{i-1}, v_{j+1}).

Algorithm 5 is the implementation of the *mixed traversal* strategy. Given $(B(u), B(v))$, the local search space (lft, rgt) and the query vertex s, the Algorithm 5 maintains a pointer $preJ$ (initialized as 0) which points to the deepest vertex in $B(v)$ that has ever been visited. The main loop (line 3-20) of the algorithm iterates over $B(u)$ from the lft-th to 0-th vertex. In every (say the i-th) iteration of the main loop, Algorithm 5 iterates over $B(v)$ from the $preJ$-th one, and does not stop until $B(v)[rgt]$ is reached or the vertex pair $(B(u)[i], B(v)[j])$ under checking does not depend on s. Specifically, if $(B(u)[i], B(v)[j])$ does not depend on s, Algorithm 5 sets DDLs for $B(u)[i]$ (line 9) and vertices in $B(v)$ that have been traversed in this iteration (line 10-11), and terminates current iteration; otherwise, the algorithm keeps on checking (u_i, v_{j+1}) until $j = rgt$ (line 12-17). The time complexity of Algorithm 5 is $O(lft \times rgt)$.

Theorem 4 *Algorithm 5 correctly computes DDLs of vertices of the given branch pair.*

Consider two branches $B(v_{11}) = \langle v_{11}, v_6, v_5, v_3 \rangle$, $B(v_9) = \langle v_9, v_8, v_2, v_1 \rangle$ of the *SPT* in Figure 1, and suppose we are counting the number of vertex pairs (u, v) where $u \in B(v_{11})$ and $v \in B(v_9)$ that depend on v_{10}. Given the input $v_{10}, B(v_9), 3, B(v_{11}), 3$, the algorithm first checks the dependency of (v_1, v_{11}). Because it does not depend on v_{10}, the algorithm then checks (v_2, v_{11}). Since (v_2, v_{11}) depends on v_{10}, (v_2, v_6) is checked the next. Then (v_8, v_6), (v_8, v_5), (v_8, v_3), (v_9, v_3). Compared to performing dependency checks of 4×4 vertex pairs in $B(v_{11})$, $B(v_9)$, we only need to check 7 vertex pairs.

Algorithm 5 Mixed traversal.

```
   Input: s, B(u), lft, B(v), rgt
   Output: DDLs of vertices in B(u), B(v)
1  DDL(x, v) ← −1, ∀x ∈ B(u); DDL(x, u) ← −1, ∀x ∈ B(v);
2  preJ ← 0;
3  foreach i from lft to 0 do
4      if preJ ≤ rgt then
5          j = preR, dep = true;
6          while j ≤ rgt and dep do
7              dep ← DC(B(u)[i], B(v)[j], s);
8              if not dep then
9                  DDL(B(u)[i], v) ← j − 1;
10                 for k from j − 1 to preJ do
11                     DDL(B(v)[k], u) ← i;
12             if dep then
13                 j ← j + 1;
14                 if j = rgt then
15                     DDL(B(u)[i], v) ← rgt;
16                     for k from j to preJ do
17                         DDL(B(v)[k], u) ← i;
18         preJ ← j;
19     else
20         DDL(B(u)[i], v) ← rgt;
```

Table 1 Real-world road networks

Name	Region	# Vertices n	# Edges m
DG	Dongguan	8,315	11,128
WH	Wuhan	21,560	30,008
SZ	Suzhou	46,094	62,190
SH	Shanghai	78,560	106,728
BK	Bangkok	154,352	187,364
NY	New York	264,346	730,100
BY	Bay Area	321,270	794,830
CL	Colorado	435,666	1,042,500

5 Experiments

5.1 Experiment setting

Datasets We test on eight real-world road networks [55] as shown in Table 1.

Queries We randomly sampled 1, 000 vertices from each road network as the query vertices. To inspect the *CC* distribution of the randomly selected vertices, we ranked all selected vertices by their *CC* values. We found the minimum and maximum *CC* values and the values at the four quartiles. Figure 4 shows the proportions of vertices in different quartiles. We can find those vertices in a larger road network are more likely to have larger *CC* values. Specifically, in small road networks (e.g., DG, WH, and SZ), more than 50 % of the query vertices are in the second quartile, and more than 80 % of them are in the third quartile; while

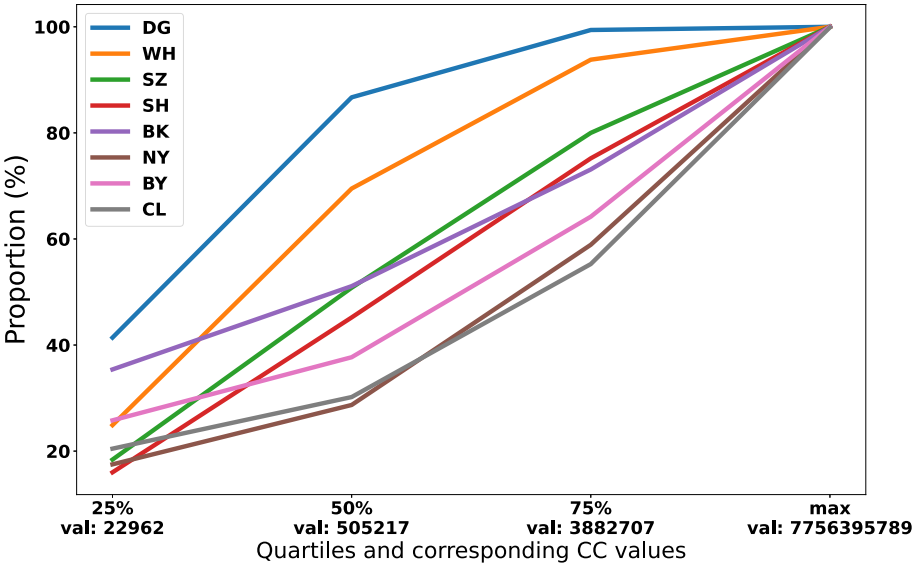

Figure 4 Proportions of *CC* values of the sampled vertices at each percentile from each road network

in large road networks (e.g., NY, BY, and CL), less than 30 % query vertices are in the second quartile.

Algorithms We evaluate the efficiency and scalability of traversal strategies, denoted by *TopD* (Algorithm 1), *BotU* (Algorithm 4), and *Mix* (Algorithm 5). Their dependency checks are facilitated by the shortest distance algorithm *PLL* [54]. As for the baseline, since no algorithm computes exact *CC* nor is there an algorithm for *CC* or *BC* query processing, we could only compare the efficiency of our algorithm with states-of-art algorithms that collectively compute *BC* values for all vertices in the given graph. Here, we chose *Brandes* [7] as our baseline.

Metrics We present the # of dependency checks and the running time to show the efficiency of our algorithms. Note that the running time is influenced by the *SPA* employed for dependency checks (*PLL* in this experiment), which can be improved if a more efficient *SPA* is adapted. As for the *Brandes* algorithm, we only present its running time.

Implementation All algorithms are implemented in C++ and compiled with GNU GCC 9.2.0 with full optimization and conducted on a machine with an Intel Xeon CPU with 2.20GHz and 1 TB main memory running Linux.

5.2 Experimental results

Query vertices of each road network are divided into four groups according to the percentile of their *CC* values among all selected vertices. We present the experimental results in each road network based on query groups to better demonstrate the scalability of our algorithms to vertices at different scales of *CC*. Specifically, sampled vertices in each road network are divided into four groups. The first, second, third, and fourth groups contain vertices whose *CC* values are ranked between 0-25 %, 25-50 %, 50-75 %, and 75+ % among all sampled vertices, respectively. We summarize the median *CC* value of different groups in each road network (Table 2) and find that even belonging to the same group, vertices from the larger road network have bigger *CC* values.

Table 3 presents the median # of dependency checks our algorithms performed over all groups of queries of each road network. We have the following observations: (1) The # of dependency checks our algorithms performed on each query group is much smaller than the corresponding road network's total # vertex pairs. This shows the efficacy of our pruning techniques. (2) The # of dependency checks of all the algorithms increases linearly with the increase of vertices' *CC* values. (3) the # of dependency checks of *TopD* is much more than

Table 2 Median *CC* value of each query group in different road networks

Groups	*0-25%*	*25-50%*	*50-75%*	*75%-max*
DG	215	110,966	943,737	5,002,428
WH	0	120,578	1,213,911	6,780,194
SZ	0	172,103	1,397,552	8,797,780
SH	0	190,402	1,379,240	10,594,334
BK	0	300,394	1,202,580	20,945,279
NY	0	264,344	1,446,144	19,229,518
BY	0	321,268	1,285,063	15,216,037
CL	0	435,664	1,439,550	14,684,851

Table 3 Median # of dependency checks performed by each algorithm on each query group of different road networks

Groups	*0-25%*			*25-50%*			*50-75%*			*75%-max*		
Algorithms	*TopD*	*BotU*	*Mix*	*TopD*	*BotU*	*Mix*	*TopD*	*BotU*	*Mix*	*TopD*	*BotU*	*Mix*
DG	224	198	75	111,439	48,556	36,663	945,443	226,232	278,827	5,005,134	630,798	1,434,856
WH	1	1	0	120,910	48,129	43,052	1,215,069	308,820	365,900	6,785,886	1,281,323	1,897,602
SZ	0	0	0	172,548	48,733	60,898	1,398,234	472,646	381,913	8,803,833	1,554,976	2,102,204
SH	0	0	0	190,537	48,098	74,890	1,380,364	551,980	388,996	10,600,670	2,941,600	2,751,209
BK	0	0	0	300,396	88,718	147,286	1,202,644	392,120	549,806	20,875,688	4,977,471	7,760,505
NY	0	0	0	264,344	76,854	101,232	1,447,162	740,474	528,686	19,238,083	7,520,415	6,577,305
BY	0	0	0	321,268	105,409	113,810	1,285,063	420,823	610,557	15,220,859	6,489,575	5,931,705
CL	0	0	0	435,664	125,310	125,434	1,439,664	337,342	567,786	14,689,052	6,908,770	7,494,322

Table 4 Median. running time (sec) of each algorithm on each query group of different road networks

Groups	*0-25 %*			*25-50%*			*50-75%*			*75%-max*			All
Algorithms	*TopD*	*BotU*	*Mix*	*TopD*	*BotU*	*Mix*	*TopD*	*BotU*	*Mix*	*TopD*	*BotU*	*Mix*	*Brandes*
DG	0	0	0	0	0	0	4	1	1	18	2	5	5
WH	0	0	0	1	0	0	8	2	2	42	7	10	52
SZ	0	0	0	3	1	1	25	7	7	142	23	30	238
SH	0	0	0	4	1	2	28	9	9	211	46	50	840
BK	0	0	0	9	3	4	39	12	18	526	122	180	5,234
NY	0	0	0	6	2	3	29	12	13	351	97	109	15,444
BY	0	0	0	6	3	3	28	10	15	316	101	118	23,895
CL	0	0	0	10	4	4	41	9	16	337	130	169	51,328

that of *BotU* and *Mixed* in all query groups. Specifically, the # of dependency checks of *TopD* are slightly larger than corresponding *CC* values. In contrast, the # of checks by *BotU* and *Mixed* are fewer than 1/3 of the *CC* values. This shows the stronger pruning power of *BotU* and *Mixed*. (4) *BotU* has stronger pruning power than *Mix* for vertices with medium to large *CC* values.

Table 4 presents the median running time of our algorithms over different query groups in all road networks. It shows that our methods are more efficient than *Brandes* in orders of magnitude. Especially for vertices in the first and second query groups that have small to medium *CC* values, the processing time of our methods is negligible compared to which of *Brandes*. Besides, the running time of our methods increases slowly, unlike *Brandes*, whose running time increases quadratically with the increasing size of the road network. This shows the much better scalability of our methods compared to *Brandes*. Among our algorithms, we find that in every road network, *TopD* method is the least efficient in all the query groups. Its running time can be 2 to 5 times larger than that of *BotU* and *Mix*. Besides, with the increase of *CC* values, the running time of *TopD* grows faster than which of *BotU* and *Mix*. *BotU* and *Mix* have similar efficiency in all road networks' first and second query groups. Whereas in the third and fourth groups, *BotU* is superior to *Mix* in terms of both efficiency and scalability. The superiority becomes increasingly evident with the increase of *CC* values.

5.3 Dynamic *CC* case study

We conducted a case study on a time-dependent Beijing road network consisting of 296,710 vertices and 774,660 edges with traffic condition changes every two hours. To perform the study, we randomly selected 1,000 vertices. For each selected vertex, denoted as v, we examined its *CC* every two hours from 2 : 00 to 24 : 00 in a day. We then calculated the standard deviation of the CC values for each vertex.

The results revealed that the average standard deviation of the selected vertices reached a staggering value of 7×10^7. Only 9% of the vertices maintained a constant *CC* throughout the entire observation period, indicating that the *CC* of most vertices changed significantly over time. To visualize these dynamics, we plotted the *CC* values of five selected vertices, representing each vertex with a different color, at all the test times (Figure 5). The figure emphasized how *CC* can differ completely over time.

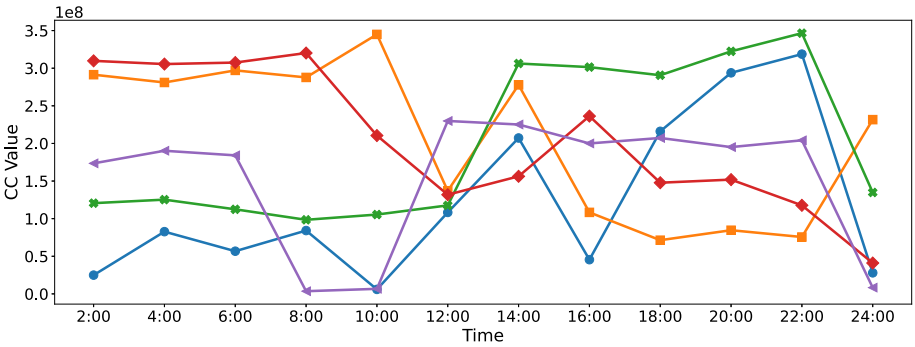

Figure 5 Dynamics of *CC* of five random vertices in a day

Therefore, to address the need for up-to-date *CC* of critical vertices in a road network, it is crucial to have a real-time *CC* query answering algorithm which can efficiently retrieve the *CC* value of the queried vertex at the time of the query. So, in terms of efficiency, our *CC* query takes 75 seconds in average to compute (medium query time is 15 seconds), while the *Brandes* algorithm takes more than 6 hours to finish. It should be noted that for *Brandes*, no result for any single vertex could be obtained before every vertex's *CC* is computed. Therefore, when it finishes computation, the results are outdated with traffic conditions have changed three times.

6 Conclusion

In this paper, we proposed the novel coverage centrality (*CC*) query. Initially, answering a *CC* query needs to perform a dependency check for every vertex pair in the road network to see if any of its shortest path passes the query vertex, which is infeasible for large road networks. Thereby, we aimed to accelerate the query processing. We proposed the computation framework which employs the shortest path tree (*SPT*) T_v rooted at the query vertex v as the carrier of the entire search space, and the pruning techniques to prune the unnecessary vertex pairs for *CC* computation in T_v. Then, we proposed the *top-down traversal* strategy to implement the computation framework, which processes *CC* queries in milliseconds for vertices with small CC values regardless of the size of the road network. Nevertheless, it cannot well scale to query vertices with large *CC* values. To enhance the scalability of those vertices, we further proposed the *bottom-up traversal* strategy and the *mixed traversal* strategy. The experimental results on extensive real-world road networks show the efficiency and scalability of our proposed methods in real-life applications.

Author Contributions Yehong Xu, Mengxuan Zhang, and Xiaofang Zhou wrote the main manuscript text. Yehong Xu, Mengxuan Zhang, and Lei Li proposed the algorithms. Yehong Xu, Ruizhong Wu, Lei Li conducted the experiments. Lei Li and Xiaofang Zhou summarized the experimental results, prepared the figures and tables, and reviewed the manuscript.

Funding Open access funding provided by Hong Kong University of Science and Technology. The research work described in this paper was supported by Natural Science Foundation of China (grant # 62202116 and # 62072125), Hong Kong Research Grants Council (grant # 16202722) and was partially conducted in the JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust.

Availability of Data and Materials The data sets used or examined during this study are available from the corresponding author on reasonable request.

Declarations

Ethical Approval Not applicable.

Competing interests The authors declare no competing interests.

References

1. Jiang, B.: Street hierarchies: a minority of streets account for a majority of traffic flow. Int. J. Geogr. Inf. Sci. **23**(8), 1033–1048 (2009)
2. Zhang, X., Miller-Hooks, E., Denny, K.: Assessing the role of network topology in transportation network resilience. J. Transp. Geogr. **46**, 35–45 (2015)
3. Rupi, F., Bernardi, S., Rossi, G., Danesi, A.: The evaluation of road network vulnerability in mountainous areas: a case study. Netw. Spat. Econ. **15**(2), 397–411 (2015)
4. Henry, E., Bonnetain, L., Furno, A., El Faouzi, N.-E., Zimeo, E.: Spatio-temporal correlations of betweenness centrality and traffic metrics. In: 2019 6th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS), pp. 1–10. IEEE (2019)
5. Li, Y., U, L.H., Yiu, M.L., Kou, N.M.: An experimental study on hub labeling based shortest path algorithms. Proceedings of the VLDB Endowment **11**(4), 445–457 (2017)
6. Yoshida, Y.: Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1416–1425 (2014)
7. Brandes, U.: A faster algorithm for betweenness centrality. J. Math. Sociol. **25**(2), 163–177 (2001)
8. Sariyüce, A.E., Saule, E., Kaya, K., Çatalyürek, Ü.V.: Shattering and compressing networks for betweenness centrality. In: Proceedings of the 2013 SIAM International Conference on Data Mining, pp. 686–694. SIAM (2013)
9. Hoang, L., Pontecorvi, M., Dathathri, R., Gill, G., You, B., Pingali, K., Ramachandran, V.: A round-efficient distributed betweenness centrality algorithm. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 272–286 (2019)
10. Madduri, K., Ediger, D., Jiang, K., Bader, D.A., Chavarria-Miranda, D.: A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–8. IEEE (2009)
11. Daniel, C., Furno, A., Goglia, L., Zimeo, E.: Fast cluster-based computation of exact betweenness centrality in large graphs (2021)
12. De Meo, P., Ferrara, E., Fiumara, G., Provetti, A.: Generalized louvain method for community detection in large networks. In: 2011 11th International Conference on Intelligent Systems Design and Applications, pp. 88–93. IEEE (2011)
13. Puzis, R., Zilberman, P., Elovici, Y., Dolev, S., Brandes, U.: Heuristics for speeding up betweenness centrality computation. In: 2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing, pp. 302–311. IEEE (2012)
14. Zhang, M., Li, L., Hua, W., Zhou, X.: Dynamic hub labeling for road networks. ICDE. IEEE (2021)
15. Mengxuan, Z., Lei, L., Wen, H., Xiaofang, Z.: Efficient 2-hop labeling maintenance in dynamic small-world networks. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 133–144. IEEE (2021)

16. Zhou, A., Wang, Y., Chen, L.: Butterfly counting on uncertain bipartite graphs. Proceedings of the VLDB Endowment **15**(2), 211–223 (2021)
17. Zhang, M., Li, L., Zhou, X.: An experimental evaluation and guideline for path finding in weighted dynamic network. Proceedings of the VLDB Endowment **14**(11), 2127–2140 (2021)
18. Li, L., Hua, W., Du, X., Zhou, X.: Minimal on-road time route scheduling on time-dependent graphs. Proceedings of the VLDB Endowment **10**(11), 1274–1285 (2017)
19. Li, L., Wang, S., Zhou, X.: Time-dependent hop labeling on road network. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 902–913. IEEE (2019)
20. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 43–54 (2008)
21. Xu, Y., Zhang, M., Wu, R., Li, L.: A top-down scheme for coverage centrality queries on road networks. In: Australasian Database Conference, pp. 37–49. Springer (2022)
22. Ishakian, V., Erdös, D., Terzi, E., Bestavros, A.: A framework for the evaluation and management of network centrality. In: Proceedings of the 2012 SIAM International Conference on Data Mining, pp. 427–438. SIAM (2012)
23. Freeman, L.C.: A set of measures of centrality based on betweenness. Sociometry, pp. 35–41 (1977)
24. Anthonisse, J.M.: The rush in a directed graph. Stichting Mathematisch Centrum. Mathematische Besliskunde, BN 9/71 (1971)
25. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths in road networks. In: International Symposium on Experimental Algorithms, pp. 230–241. Springer (2011)
26. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: European Symposium on Algorithms, pp. 24–35. Springer (2012)
27. Dijkstra, E.W., et al.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
28. D'Angelo, G., Olsen, M., Severini, L.: Coverage centrality maximization in undirected networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, no. 01, pp. 501–508 (2019)
29. Medya, S., Silva, A., Singh, A., Basu, P., Swami, A.: Group centrality maximization via network design. In: Proceedings of the 2018 SIAM International Conference on Data Mining, pp. 126–134. SIAM (2018)
30. Wang, W., Tang, C.Y.: Distributed computation of node and edge betweenness on tree graphs. In: 52nd IEEE Conference on Decision and Control, pp. 43–48. IEEE (2013)
31. Prountzos, D., Pingali, K.: Betweenness centrality: algorithms and implementations. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 35–46 (2013)
32. Bader, D.A., Madduri, K.: Parallel algorithms for evaluating centrality indices in real-world networks. In: 2006 International Conference on Parallel Processing (ICPP'06), pp. 539–550. IEEE (2006)
33. Cong, G., Makarychev, K.: Optimizing large-scale graph analysis on multithreaded, multicore platforms. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 414–425. IEEE (2012)
34. Edmonds, N., Hoefler, T., Lumsdaine, A.: A space-efficient parallel algorithm for computing betweenness centrality in distributed memory. In: 2010 International Conference on High Performance Computing, pp. 1–10. IEEE (2010)
35. Suppa, P., Zimeo, E.: A clustered approach for fast computation of betweenness centrality in social networks. In: 2015 IEEE International Congress on Big Data, pp. 47–54. IEEE (2015)
36. Bader, D.A., Kintali, S., Madduri, K., Mihail, M.: Approximating betweenness centrality. In: International Workshop on Algorithms and Models for the Web-Graph, pp. 124–137. Springer (2007)
37. Riondato, M., Kornaropoulos, E.M.: Fast approximation of betweenness centrality through sampling. Data Min. Knowl. Disc. **30**(2), 438–475 (2016)
38. Riondato, M., Upfal, E.: Abra: approximating betweenness centrality in static and dynamic graphs with rademacher averages. ACM Transactions on Knowledge Discovery from Data (TKDD) **12**(5), 1–38 (2018)
39. Lee, M.-J., Lee, J., Park, J.Y., Choi, R.H., Chung, C.-W.: Qube: a quick algorithm for updating betweenness centrality. In: Proceedings of the 21st International Conference on World Wide Web, pp. 351–360 (2012)
40. Jamour, F., Skiadopoulos, S., Kalnis, P.: Parallel algorithm for incremental betweenness centrality on large graphs. IEEE Trans. Parallel Distrib. Syst. **29**(3), 659–672 (2017)
41. Kourtellis, N., Morales, G.D.F., Bonchi, F.: Scalable online betweenness centrality in evolving graphs. IEEE Trans. Knowl. Data Eng. **27**(9), 2494–2506 (2015)
42. Green, O., McColl, R., Bader, D.A.: A fast algorithm for streaming betweenness centrality. In: 2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing. IEEE, pp. 11–20 (2012)

43. Jia, Y., Gu, Z., Jiang, Z., Gao, C., Yang, J.: Persistent graph stream summarization for real-time graph analytics. World Wide Web, pp. 1–21 (2023)
44. Tang, N., Chen, Q., Mitra, P.: Graph stream summarization: from big bang to big crunch. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1481–1496 (2016)
45. Kourtellis, N., Alahakoon, T., Simha, R., Iamnitchi, A., Tripathi, R.: Identifying high betweenness centrality nodes in large social networks. Soc. Netw. Anal. Min. **3**(4), 899–914 (2013)
46. Zhang, Q., Li, R.-H., Pan, M., Dai, Y., Wang, G., Yuan, Y.: Efficient top-k ego-betweenness search. arXiv preprint (2021) arXiv:2107.10052
47. Nakajima, K., Iwasaki, K., Matsumura, T., Shudo, K.: Estimating top-k betweenness centrality nodes in online social networks. In: 2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), pp. 1128–1135. IEEE (2018)
48. Lee, M.-J., Chung, C.-W.: Finding k-highest betweenness centrality vertices in graphs. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 339–340 (2014)
49. Fan, C., Zeng, L., Ding, Y., Chen, M., Sun, Y., Liu, Z.: Learning to identify high betweenness centrality nodes from scratch: a novel graph neural network approach. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, pp. 559–568 (2019)
50. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)
51. Li, L., Zhang, M., Hua, W., Zhou, X.: Fast query decomposition for batch shortest path processing in road networks. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1189–1200. IEEE (2020)
52. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: faster and simpler hierarchical routing in road networks. In: International Workshop on Experimental and Efficient Algorithms, pp. 319–333. Springer (2008)
53. Ouyang, D., Qin, L., Chang, L., Lin, X., Zhang, Y., Zhu, Q.: When hierarchy meets 2-hop-labeling: efficient shortest distance queries on road networks. In: Proceedings of the 2018 International Conference on Management of Data, pp. 709–724 (2018)
54. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 349–360 (2013)
55. Karduni, A., Kermanshah, A., Derrible, S.: A protocol to convert spatial polyline data to network formats and applications to world urban road networks. Scientific data **3**(1), 1–7 (2016)