Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees

Dian Ouyang[†,§], Long Yuan[†*], Lu Qin[‡], Lijun Chang[§], Ying Zhang[‡], Xuemin Lin[♮]
[†] School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China
[§] The University of Sydney, Australia
[‡] CAI, FEIT, University of Technology, Sydney, Australia
[♮] The University of New South Wales, Australia
[†]longyuan@njust.edu.cn;[§]{dian.ouyang, lijun.chang}@sydney.edu.au;
[‡]{lu.qin, ying.zhang}@uts.edu.au;[♮]lxue@cse.unsw.edu.au

ABSTRACT

Computing the shortest path between two vertices is a fundamental problem in road networks that is applied in a wide variety of applications. To support efficient shortest path query processing, a plethora of index-based methods have been proposed in the literature, but few of them can support dynamic road networks commonly encountered in practice, as their corresponding index structures cannot be efficiently maintained when the input road network is dynamically updated. Motivated by this, we study the shortest path index maintenance problem on dynamic road networks in this paper. We adopt Contraction Hierarchies (CH) as our underlying shortest path computation method because of its outstanding overall performance in pre-processing time, space cost, and query processing time and aim to design efficient algorithms to maintain the index structure, *shortcut index*, of CH when the input road network is dynamically updated. To achieve this goal, we propose a shortcut-centric paradigm focusing on exploring a small number of shortcuts to maintain the *shortcut index*. Following this paradigm, we design an auxiliary data structure named SS-Graph and propose a shortcut weight propagation mechanism based on the SS-Graph. With them, we devise efficient algorithms to maintain the *shortcut index* in the streaming update and batch update scenarios with non-trivial theoretical guarantees. We experimentally evaluate our algorithms on real road networks and the results demonstrate that our approach achieves 2-3 orders of magnitude speedup compared to the state-of-the-art algorithm for the streaming update.

PVLDB Reference Format:
Dian Ouyang, Long Yuan, Lu Qin, Lijun Chang, Ying Zhang, Xuemin Lin. Efficient Shortest Path Index Maintenance on Dynamic Road Networks with Theoretical Guarantees. *PVLDB*, 13(5): 602-615, 2020.
DOI: https://doi.org/10.14778/3377369.3377371

1. INTRODUCTION

Computing the shortest path between two locations is one of the most fundamental problems in road networks with many applications such as GPS navigation, route planning services and POI recommendation [11, 28, 31, 40, 33]. Typically, a road network can be modelled as a weighted graph $G = (V, E, \phi)$, where each vertex $v \in V$ represents a junction, each edge $e \in E$ represents a road segment between two junctions, and the weight of an edge $\phi(e)$ represents the transit time between two junctions. Given a source vertex s and a destination vertex t, a shortest path query asks for the path with the shortest network distance from s to t in G.

The classic approach to answering the shortest path queries is Dijkstra's algorithm [18]. However, given Dijkstra's algorithm may traverse the entire network when the two query vertices are far apart, this approach cannot satisfy the real-time requirements for the shortest path queries on large networks. As a result, indexing-based approaches are studied to tackle this problem. A plethora of index-based methods for shortest path queries on road networks have been proposed in the literature, such as ALT [21], hierarchical MulTi (HiTi) [24], highway hierarchy (HH) [35], contraction hierarchy (CH) [19], customizable route planning (CRP) [17] and arterial hierarchy (AH) [50]. Of these methods, CH has been highly successful due to its outstanding overall performance in pre-processing time, space cost, and query processing time [38, 29]. Index-based methods to compute the shortest distance for a given query are also studied [12, 7, 8, 9, 32], but they cannot retrieve the concrete shortest paths efficiently and therefore, are not considered.

Motivation. Although existing index-based methods are efficient to compute the shortest path, most of them assume that the input road networks are static. Unfortunately, in real road networks, the vertex set V and the edge set E are usually static (as road construction/removal seldom happens in a city), but the edge weights (transit time) are dynamically updated due to the change of traffic conditions. For example, the commercial live map service providers, such as TomTom and INRIX, update their road traffic information every 1 minute currently to reflect the real traffic conditions [6, 36]. Furthermore, this update frequency cannot fully satisfy the accuracy requirement in practice yet and these service providers are seeking crowdsourcing-based approach to obtain real-time traffic information [1, 2]. By collecting the travel information of App users, the crowdsourcing-based approach can obtain real-time traffic information. However, the crowdsourcing-based approach usually generates a huge volume of new records at high speed. According to [49], in 2010, Beijing has approximately 67,000 licensed taxis generating over 1.2 million trips records per day. It means nearly 13.8 new updates are generated per second on average. According to [47], in 2012, the approximately 13,000 taxis in the New

*Dian Ouyang and Long Yuan are the joint first authors. Long Yuan is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377371

York City generate more than half a million taxi trip records per day, which means nearly 5.78 new updates are generated per second on average. With the proliferation of ridesharing service, such as Uber and DiDi, the number of registered drivers rapidly grows in these years [4, 5]. Accordingly, the number and the generating speed of update records significantly increase as well. The high dynamics of edge weight makes existing index-based approaches inapplicable for real applications, as their index structures do not allow efficient updates when the edge weight changes.

To handle the dynamics of edge weight, $\mathsf{DCH}_{\mathsf{vcs}}$ [20] is proposed to incrementally maintain the index structure, *shortcut index*, of CH. $\mathsf{DCH}_{\mathsf{vcs}}$ follows a vertex-centric paradigm and imitates the procedure of CH to update the *shortcut index* from vertex perspectives. Although it can reduce the computation compared with reconstructing the *shortcut index* from scratch, it has two drawbacks: (1) Theoretically, $\mathsf{DCH}_{\mathsf{vcs}}$ shares the trivial worst-case time complexity with reconstructing the index from scratch. (2) Practically, $\mathsf{DCH}_{\mathsf{vcs}}$ entails prohibitive time-cost to maintain the *shortcut index* for an edge weight update, as shown in our experiment (details in Section 3). Considering road networks in the physical world are typically large and edge weights are updated frequently as presented above, $\mathsf{DCH}_{\mathsf{vcs}}$ is inefficient for real-world applications.

Motivated by this, we aim to design efficient index maintenance algorithms to support the shortest path queries on dynamic road networks. We adopt CH as our underlining shortest path query processing method for its outstanding overall performance in preprocessing time, space cost, and query processing time. Our objective is to overcome the drawbacks of $\mathsf{DCH}_{\mathsf{vcs}}$ and achieve progress on both theoretical and practical aspects simultaneously in the *shortcut index* maintenance.

In the literature, time-dependent road networks are also studied. In a time-dependent road network, each edge (u, v) is assigned with a function f which specifies the time $f(t)$ needed to reach v from u via edge (u, v) when starting at time t [16]. Time-dependent road network assumes that the edge weight updates are known beforehand and given as a function of time. As a result, it is unable to handle sudden and unpredictable edge weight update, such as traffic incident which happens frequently in practice [3]. Moreover, obtaining a function which can precisely reflect the edge weight change over time is hard [48]. On the other hand, dynamic graph model does not have the assumption that the edge weight updates are known beforehand and the edge weight updates come and process in an online fashion, which means the sudden and unpredictable edge weight update can be easily handled. Therefore, we use dynamic graph model in this paper.

Our Idea. While $\mathsf{DCH}_{\mathsf{vcs}}$ maintains the *shortcut index* in a vertex-centric paradigm, we observe that when the weight of an edge in the given road network is updated, the topological structure of the corresponding *shortcut index* stays the same and only very few shortcut weights in the *shortcut index* are changed (For consistency, edges in the *shortcut index* are called shortcuts) as shown in Exp-5 of Section 6. This reveals the opportunity to incrementally maintain the *shortcut index* from shortcut perspectives, by exploring only a small number of shortcuts in the *shortcut index*.

When the weight of an edge is updated, let Δ be the shortcuts with weight change in the *shortcut index*. As discussed above, the topological structure of the *shortcut index* stays the same and $|\Delta|$ is small in practice. Thus, we aim to explore the shortcuts only related to Δ to achieve high efficiency. To reach this goal, we design an auxiliary data structure named SS-Graph, to track the weight dependence relationships of the shortcuts in the *shortcut index*. Based on the SS-Graph, we devise efficient algorithms with tight bounds related to $|\Delta|$ to maintain the *shortcut index*.

Contribution. In this paper, we make the following contributions:

(1) A new paradigm to maintain the shortcut index for dynamic road networks. In this paper, we adopt CH as our underlying shortest path query processing method and aim to efficiently maintain the *shortcut index* of CH for dynamic road networks. Instead of using the vertex-centric paradigm employed by the state-of-the-art approach, we propose a shortcut-centric paradigm to maintain the *shortcut index* based on the observation that the weights of very few shortcuts are changed when the weight of an edge is updated in the road network.

(2) Efficient algorithms to maintain the shortcut index with non-trivial theoretical guarantees. Following the shortcut-centric paradigm, we design the SS-Graph and shortcut weight propagation mechanism. Based on these, we propose efficient algorithms to maintain the *shortcut index* in both streaming update scenarios and batch update scenarios with non-trivial theoretical guarantees. To the best of our knowledge, this is the first solution to the *shortcut index* maintenance with such tight bounds. We also explore techniques to maintain the *shortcut index* without the materialized SS-Graph to further reduce memory consumption.

(3) Extensive performance studies on real road networks. We conduct extensive performance studies on eight real road networks. The experimental results demonstrate that our proposed approach can achieve 2-3 orders of magnitude speedup compared with the state-of-the-art algorithm for the streaming update.

2. PRELIMINARIES

2.1 Shortest Path Queries

Let $G = (V, E, \phi)$ be a road network (i.e., a degree-bounded connected and weighted graph) where $V(G)$ is the set of vertices, $E(G)$ is the set of edges, and $\phi : E(G) \rightarrow \mathbb{R}^+$ is a function that assigns each edge a positive number as its weight. We use $n = |V(G)|$ and $m = |E(G)|$ to denote the number of vertices and edges in the road network, respectively. For each vertex $v \in V(G)$, the neighbors of v, denoted as $\mathsf{nbr}(v, G)$, is defined as $\mathsf{nbr}(v, G) = \{u|(u, v) \in E(G)\}$. The degree of a vertex $v \in V(G)$, denoted by $\mathsf{deg}(v, G)$, is the number of neighbors of v, i.e., $\mathsf{deg}(v, G) = |\mathsf{nbr}(v, G)|$. For each edge $e = (u, v) \in E(G)$, we use $\phi((u, v), G)$ to denote its associated weight. A path is a sequence of vertices $p = (v_1, v_2, \cdots, v_k)$ where $(v_i, v_{i+1}) \in E(G)$ for each $1 \leq i < k$. The weight of a path p, denoted as $\phi(p, G)$, is defined as $\phi(p, G) = \Sigma_{i=1}^{k-1} \phi((v_i, v_{i+1}), G)$. Given two vertices $s, t \in V(G)$, the shortest path p between s and t is a path starting from s and ending at t with the minimum $\phi(p, G)$ and the shortest distance of s and t in G, denoted by $dist_G(s, t)$, is the weight of any shortest path between s and t. Given a road network G, a shortest path query $q = (s, t)$ returns the shortest path between s and t in G, where $s, t \in V(G)$. For simplicity, we omit G in the notations if the context is self-evident. For the ease of explanation, we consider G as an undirected graph in this paper, and our techniques can be easily extended to handle directed graphs.

2.2 Contraction Hierarchies

Given a road network G, CH is defined in two phases. Next, we explain these two phases in detail.

Phase 1: Shortcut Index Construction. In the first phase, CH focuses on constructing the *shortcut index* G', which is based on the vertex contraction operator $\ominus$.

Definition 2.1: (Vertex contraction operator $\ominus$) Given a road network G, a total vertex order γ and a vertex v, the vertex contraction operator applied on v in G, denoted by $G \ominus v$, transforms

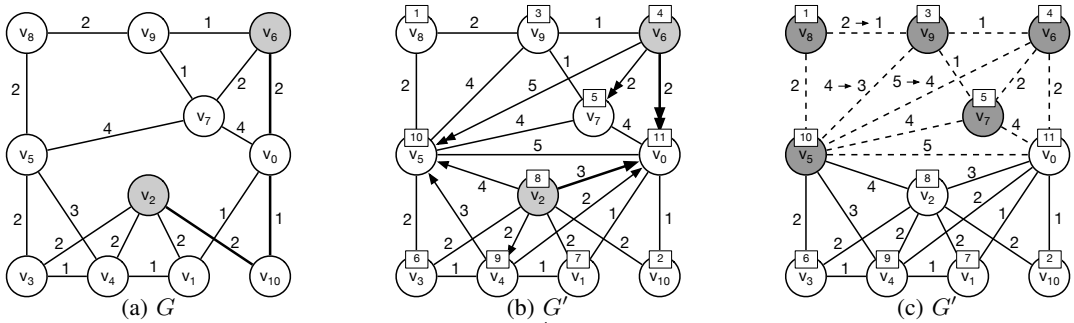

Figure 1: A road network G, its *shortcut index* G' and $\mathsf{DCH}_{\mathsf{vcs}}$ for (v_8, v_9) with edge weight decrease

Algorithm 1 ShortcutIndexConstruction

Input: A road network $G(V, E)$ and a total vertex order γ
Output: The shortcut index G' of G

```
1: G' ← G;
2: for each vertex v ∈ V(G') in the predefining total order γ do
3:    G' ← G' ⊖ v;
4: return G';
5: procedure operator ⊖(G, γ, v)
6:    G ← G;
7:    for all pair of vertices u, w ∈ nbr(v, G) with γ(u) > γ(v) and
      γ(w) > γ(v) do
8:       if (u, w) ∉ E(G) then
9:          insert edge (u, w) with φ(u, w) ← φ(u, v) + φ(v, w) in G;
10:      else if φ(u, w) > φ(u, v) + φ(v, w) then
11:         update φ(u, w) ← φ(u, v) + φ(v, w) in G;
12:   return G;
```

G into $\mathcal{G}$ as follows: For every pair of neighbors u and w of v with $\gamma(u) > \gamma(v)$ and $\gamma(w) > \gamma(v)$, if $(u, w) \notin E(G)$, a new edge (u, w) with weight $\phi((u, w)) = \phi((u, v)) + \phi((v, w))$ is inserted. Otherwise, if $\phi((u, v)) + \phi((v, w)) < \phi((u, w))$, the weight of (u, w) is updated with $\phi((u, v)) + \phi((v, w))$. □

CH first assigns a total order γ of vertices by assigning each vertex v a rank $\gamma(v)$ based on their importance. Then, CH examines each vertex following γ. For each vertex v, CH applies the vertex contraction operator on v. The shortcut index construction phase terminates after all the vertices are examined. The edges in G whose weight are not decreased along with all the edges generated by $\ominus$ form the index structure of CH. The index structure is called *shortcut index* and we denote it as G'. For the ease of distinction, we refer the edges in the *shortcut index* as *shortcuts* and the edges in the given road network as *edges*. The detailed algorithm of shortcut index construction is shown in Algorithm 1.

Phase 2: Query Processing. For a shortest path query $q = (s, t)$, CH answers the query using the bidirectional Dijkstra's algorithm on G' [34] with some minor modifications. Specifically, it starts two instances of Dijkstra's algorithm simultaneously from s and t in G', respectively. During the search, it only considers shortcuts connecting a visited vertex v to an unvisited vertex v' with a higher rank than v, i.e., $\gamma(v) < \gamma(v')$. The search terminates when all keys in the respective priority queue of Dijkstra's algorithm instance are larger than the tentative shortest distance and the shortest path p on G' is obtained. After obtaining p on G', CH outputs the shortest path on G by unpacking the shortcuts in p.

Example 2.1: Figure 1 (a) shows a road network G with 11 vertices and 18 edges. The weight of the edge $\phi(e)$ is indicated beside each edge. Assume the vertices are ranked as $v_8 < v_{10} < v_9 < v_6 < v_7 < v_3 < v_1 < v_2 < v_4 < v_5 < v_0$. We mark the rank of each vertex in a box beside each vertex in Figure 1 (b). The *shortcut index* G' of G is shown in Figure 1 (b). To answer the shortest path query $q = (v_2, v_6)$, CH starts two instances of Dijkstra's algorithm from v_2 and v_6 on G', respectively, which are shown in arrow-headed line (start from v_2) and double-arrow-headed line (start from v_6) in Figure 1 (b). These two traversals finally meet at v_0, and we have the shortest path $p = \{v_2, v_0, v_6\}$ on G' with weight 5. Then, we replace the shortcut (v_2, v_0) in p by (v_2, v_{10}) and (v_{10}, v_0) and the shortest path from v_2 to v_6 on G is $\{v_2, v_{10}, v_0, v_6\}$. □

2.3 Problem Statement

Now, we provide a formal definition of the problem studied in this paper: Given a road network G, compute the *shortcut index* G' of G when the edge weights of G are dynamically updated. We distinguish two different edge weight update scenarios for different real application requirements [15]: (1) *Streaming update*, in which edge weight updates are continuously arriving and the processing occurs on each edge weight update. (2) *Batch update*, in which a batch of edge weight updates arrives each time and the processing is conducted based on the arrived batch of edges.

For a given road network, we *assume that the total order of vertices γ is fixed*. This assumption is reasonable and practical since the total vertex order is generally determined by the importance of the vertices and the edge weight updates preserve the topology of the road network, which have little influence on the importance of the vertices. This assumption is also adopted in the-state-of-the approach [20].

3. STATE-OF-THE-ART ALGORITHM

The state-of-the-art algorithm to maintain the *shortcut index* is $\mathsf{DCH}_{\mathsf{vcs}}$ [20], which focuses on the streaming update.

A vertex-centric algorithm. $\mathsf{DCH}_{\mathsf{vcs}}$ (vcs denotes vertex-centric for streaming update) adopts a vertex-centric paradigm and maintains G' by leveraging $\ominus$. Specifically, it contains two steps:

Step 1. Affected Vertex Identification. When the weight of an edge $e = (u, v)$ in G changes, without loss of generality, let w be another neighbor of u and assume that $\gamma(u) < \gamma(v)$ and $\gamma(u) < \gamma(w)$. Based on Phase 1 of CH, the weight of shortcut (v, w) may change due to the vertex contraction operation on u. Recursively, the weight change of (v, w) may further affect the weight of shortcuts incident to v for the same reason. Therefore, starting from the shortcut e with weight change, $\mathsf{DCH}_{\mathsf{vcs}}$ explores the shortcuts generated upon e in a depth-first search manner on G'. For an explored shortcut (u', v'), the incident vertex with $\min\{\gamma(u'), \gamma(v')\}$ is recorded as an affected vertex.

Step 2. Vertex Recontraction. After all the affected vertices are obtained, $\mathsf{DCH}_{\mathsf{vcs}}$ updates the weight of e and applies the vertex contraction operator $\ominus$ on all the vertices identified in Step 1 following the total vertex order γ. The new generated *shortcut index* is the *shortcut index* of the updated road network.

Example 3.1: Consider the road network G and its *shortcut index* G' shown in Figure 1 (a) and (b), respectively. When the weight of (v_8, v_9) changes from 2 to 1, Figure 1 (c) illustrates the procedure of $\mathsf{DCH}_{\mathsf{vcs}}$ to maintain G'. In Step 1, it first identifies affected vertices. Since the weight of (v_8, v_9) changes and $\gamma(v_8) < \gamma(v_9)$, v_8 is identified as an affected vertex. Then, it explores the shortcuts generated upon (v_8, v_9) in a depth-first search manner and further identifies other affected vertices. As (v_9, v_5) is generated upon (v_8, v_9) and $\gamma(v_9) < \gamma(v_5)$, v_9 is identified as an affected vertex. Recursively, v_6, v_7 and v_5 are also identified as affected vertices. Therefore, the affected vertices regarding the weight update of (v_8, v_9) are $\{v_8, v_9, v_6, v_7, v_5\}$, which are marked with dark grey in Figure 1 (c). In Step 2, $\mathsf{DCH}_{\mathsf{vcs}}$ recontracts all the affected vertices following γ with $\ominus$. In Figure 1 (c), the checked shortcuts of $\mathsf{DCH}_{\mathsf{vcs}}$ are marked with dash lines and the original and updated weights for the shortcuts with weight change are shown near the shortcuts. For example, the weight of shortcut (v_5, v_9) changes from 4 to 3, when performing $\ominus$ on v_8 with the updated weight of (v_8, v_9). □

Theorem 3.1: *Given the shortcut index G' of G, for an edge weight update in G, the time complexity of $\mathsf{DCH}_{\mathsf{vcs}}$ to maintain G' is $O(n \cdot d_{\max}^2)$, where $d_{\max}$ is the maximum degree of G.*

4. STREAMING UPDATE ALGORITHM

4.1 A Shortcut-Centric Paradigm

$\mathsf{DCH}_{\mathsf{vcs}}$ adopts a vertex-centric paradigm and maintains the *shortcut index* G' from the vertex perspectives. However, the vertex-centric paradigm is not suitable for maintaining G'. It causes two kinds of unnecessary computation in $\mathsf{DCH}_{\mathsf{vcs}}$: (1) In Step 1, $\mathsf{DCH}_{\mathsf{vcs}}$ considers all the vertices incident to the explored shortcuts generated upon the edge e as the affected vertices. However, not all of the weights of these explored shortcuts will change after the weight update of e. (2) In Step 2, for an identified vertex u, all pairs of u's neighbors with a higher rank than u regarding γ are checked due to the vertex contraction operation applied on u. However, some shortcut weights may remain the same after the weight of e changes, which means checking all pairs of neighbors of u with a higher rank than u in Step 2 of $\mathsf{DCH}_{\mathsf{vcs}}$ creates lots of unnecessary computations. Example 4.1 shows these problems.

Example 4.1: Reconsider the procedure of $\mathsf{DCH}_{\mathsf{vcs}}$ to handle the edge weight update of (v_8, v_9) shown in Figure 1. In Step 1, $\mathsf{DCH}_{\mathsf{vcs}}$ identifies $\{v_8, v_9, v_6, v_7, v_5\}$ as affected vertices. However, since the weight of (v_7, v_5), (v_7, v_0), and (v_5, v_0) does not change after the weight change of (v_8, v_9), v_7 and v_5 are mistakenly identified as affected vertices. In Step 2, when applying $\ominus$ on v_9, $\mathsf{DCH}_{\mathsf{vcs}}$ also checks the neighbor pair v_6 and v_7 of v_9. Obviously, checking neighbor pair v_6 and v_7 is unnecessary computation when maintaining G', since the weight of (v_9, v_6) and (v_9, v_7) does not change after the weight update of (v_8, v_9). □

Based on above analysis, the vertex-centric paradigm is not suitable for maintaining G'. On the other hand, revisiting Example 4.1, we can observe that the topological structure of G' does not change when the weight of an edge e changes and the essence of maintaining G' is to correct the weights of shortcuts with weight change after the weight update of e. Meanwhile, as shown in Exp-5 in Section 6, the number of shortcuts with weight change after the weight update of e is small in practice. Therefore, in this paper, we adopt a shortcut-centric paradigm and maintain G' from the shortcut perspectives rather than the vertex perspectives. Specifically, when the weight of e is updated, instead of identifying the affected vertices as $\mathsf{DCH}_{\mathsf{vcs}}$, we identify the affected shortcuts with weight change caused by the weight update of e. For these affected shortcuts, we correct their weights based on the new road network after the weight update of e. In this way, we can totally avoid the unnecessary computation involving in $\mathsf{DCH}_{\mathsf{vcs}}$ caused by the vertex-centric paradigm. However, to make our idea applicable in practice, the following issues need to be addressed: (1) how to efficiently identify the affected shortcuts, and (2) how to efficiently correct the weights of these affected shortcuts. In the following section, we will address these two issues.

4.2 Running Time Bounded Algorithms

For a road network G, we first prove that the topological structure of its *shortcut index* G' stays the same when the weight of an edge in G changes. For brevity, we use $G_{\oplus(e,k)}$ to denote the road network after updating the weight of an edge e to k and $G'_{\oplus(e,k)}$ to denote the corresponding shortcut index of $G_{\oplus(e,k)}$. We have:

Lemma 4.1: *Given a road network G and its corresponding shortcut index G', after updating the weight of an edge e to k in G, $V(G') = V(G'_{\oplus(e,k)})$ and $E(G') = E(G'_{\oplus(e,k)})$.*

According to Lemma 4.1, G' and $G'_{\oplus(e,k)}$ shares the same topological structure when the weight of an edge e in G is updated. Therefore, to efficiently maintain the *shortcut index* G', we only need to concentrate on updating the weights of shortcuts whose weights in G' and $G'_{\oplus(e,k)}$ are different. However, since the weight of a shortcut depends on the weights of other shortcuts in the *shortcut index*, it's hard to identify these shortcuts and update their weights directly. To address this problem, we first define:

Definition 4.1: (Supporting Shortcut Pair) Given a *shortcut index* G', for a shortcut $(u, v) \in E(G')$, let (u, w), (v, w) be another two shortcuts in G', we call (u, w) and (v, w) are a supporting shortcut pair of (u, v) if $\gamma(w) < \gamma(u)$ and $\gamma(w) < \gamma(v)$. □

Definition 4.2: (SS-Graph) Given a *shortcut index* G', the SS-Graph (Shortcut Supporting Graph) G^* of G' is a directed graph that contains two types of vertices: (1) Shortcut type vertices V_s, each such vertex v_s corresponds to a shortcut s in G' and the weight of v_s is equal to the weight of s. (2) Supporting relation type vertices V_r, each such vertex v_r corresponds to a supporting relation instance between a shortcut s and one of its supporting shortcut pair s_1 and s_2. For each supporting relation type vertex v_r, we connect three directed edges $\langle v_r, v_s\rangle$, $\langle v_{s_1}, v_r\rangle$ and $\langle v_{s_2}, v_r\rangle$ in G^*, where v_s, v_{s_1} and v_{s_2} are the corresponding shortcut type vertices of shortcuts s, s_1 and s_2, respectively. □

As shown in Lemma 4.1, for a given road network G, when the weight of an edge in G changes, the topological structure of its *shortcut index* G' does not change. Following Definition 4.2, it is clear that the topological structure of G^* is only determined by the topological structure of the given *shortcut index* G'. Therefore, for a road network G, the topological structure of the corresponding SS-Graph G^* does not change either when the weight of an edge in G changes. For this reason, hereafter, when we talk about the road network G, *shortcut index* G' and SS-Graph G^*, we assume that their topological structures match each other unless specified.

In an SS-Graph G^*, if there is a directed edge $\langle u, v\rangle$ in G^*, we call u is an in-neighbor of v and v is an out-neighbor of u. For each vertex $v \in V(G^*)$, we use $\mathsf{nbr}^-(v, G^*)$ and $\mathsf{nbr}^+(v, G^*)$ to denote the set of its in-neighbors and out-neighbors in G^*. Given a road network G, a *shortcut index* G' and the SS-Graph G^* of G', for a shortcut $s \in G'$ and its corresponding shortcut type vertex $v_s \in G^*$, we use $\phi(s, G)$ to denote the weight of s in G if $s \in G$ (if $s \notin G$, $\phi(s, G)$ is $+\infty$), $\phi(s, G')$ to denote the weight of s in G' and $\phi(v_s, G^*)$ to denote weight of v_s in G^*. We define:

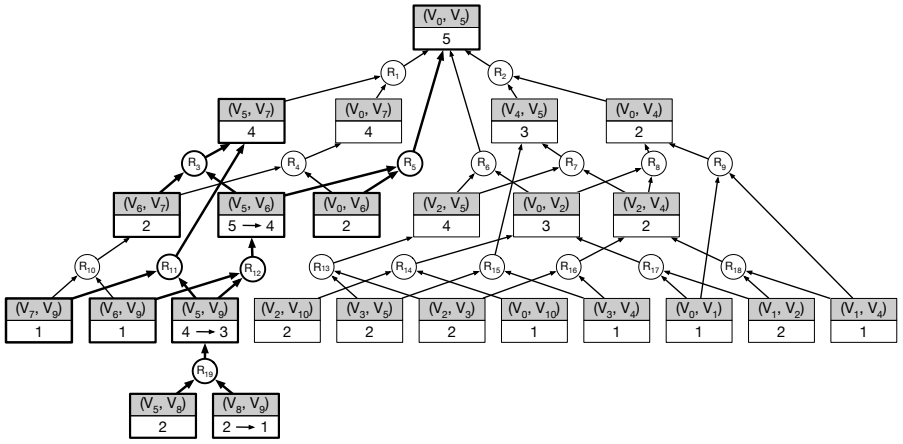

Figure 2: SS-Graph G^* and $\mathsf{DCH_{scs}}$-WDec for (v_8, v_9) with weight decrease

Definition 4.3: (Minimum Weight Property) Given a road network G, a *shortcut index* G' and the SS-Graph G^* of G', for a shortcut $s \in G'$ and its corresponding shortcut type vertex $v_s \in G^*$, let $v_{r_1}, v_{r_2}, \cdots, v_{r_n}$ be the in-neighbors of v_s and $v_{s_{11}}, v_{s_{12}}, v_{s_{21}}, v_{s_{22}}, \cdots, v_{s_{n1}}, v_{s_{n2}}$ be the in-neighbors of $v_{r_1}, v_{r_2}, \cdots, v_{r_n}$ in G^*, respectively, we say v_s satisfies the minimum weight property if $\phi(v_s, G^*) = \min\{\phi(s, G),\ \phi(v_{s_{11}}, G^*) + \phi(v_{s_{12}}, G^*), \phi(v_{s_{21}}, G^*) + \phi(v_{s_{22}}, G^*), \cdots, \phi(v_{s_{n1}}, G^*) + \phi(v_{s_{n2}}, G^*)\}$. □

Lemma 4.2: *Given a road network G, a shortcut index G' and the SS-Graph G^* of G', G' is the shortcut index of G if and only if all the shortcut type vertices v_s in G^* satisfy the minimum weight property.*

As the weight of a shortcut in G' directly corresponds to the weight of a shortcut type vertex in G^* according to Definition 4.2, the problem of maintaining the *shortcut index* G' when the weight of an edge in G changes is equivalent to maintaining the minimum weight property for all shortcut type vertices in G^* based on Lemma 4.2. Therefore, we redefine our problem as follows:

Definition 4.4: (Problem Definition*) Given a road network G and its SS-Graph G^*, we aim to adjust $\phi(v_s, G^*)$ for all shortcut type vertices in G^* to make them satisfy the minimum weight property when the weight of an edge e in G is updated to k. □

To address this problem, we have the following lemma:

Lemma 4.3: *Given a road network G and the SS-Graph G^*, when the weight of an edge e in G changes, the shortcut type vertices unreachable from v_e in G^* satisfy the minimum weight property, where v_e is the corresponding shortcut type vertex of e in G^*.*

Therefore, when the weight of an edge e in G is updated, we do not need to consider the shortcut type vertices unreachable from v_e in G^*. However, all the other shortcut type vertices may violate the minimum weight property. The remaining problem is how to identify these shortcut type vertices and adjust their $\phi(v_s, G^*)$.

Shortcut Weight Propagation Mechanism on G^*. According to Definition 4.3, for a shortcut type vertex v_s, $\phi(v_s, G^*)$ may violate the minimum weight property if and only if at least one of the values in $\phi(s, G)$, $\phi(v_{s_{11}}, G^*) + \phi(v_{s_{12}}, G^*)$, $\cdots$, $\phi(v_{s_{n1}}, G^*) + \phi(v_{s_{n2}}, G^*)$ changes, where $s, v_{s_{11}}, \cdots, v_{s_{n2}}$ follows Definition 4.3. In addition, when $\phi(v_s, G^*)$ changes, let v_r be one of the out-neighbors of v_s and v'_s be the out-neighbor of v_r. The change of $\phi(v_s, G^*)$ may further cause the shortcut type vertices v'_s to violate the minimum weight property and we need to adjust $\phi(v'_s, G^*)$ consequently. Therefore, our shortcut weight propagation mechanism works as follows: for a shortcut type vertex v_s whose $\phi(v_s, G^*)$ may change, we first determine whether $\phi(v_s, G^*)$ needs to be adjusted based on the the minimum weight property. If $\phi(v_s, G^*)$ changes, we notify the out-neighbors v'_s of v_s's out-neighbors as the candidate shortcut type vertices that $\phi(v'_s, G^*)$ may need to be further adjusted. Following this shortcut propagation mechanism, we can iteratively adjust $\phi(v_s, G^*)$ until all the shortcut type vertices satisfy the minimum weight property.

Shortcut weight propagation mechanism achieves the goal of identifying and adjusting $\phi(v_s, G^*)$ not satisfying the minimum weight property. However, using the weight propagation mechanism alone may propagate incorrect shortcut weight and futile notifications will be introduced. For example, when we adjust $\phi(v_s, G^*)$ for a shortcut type vertex v_s, if one of the value in $\phi(v_{s_{11}}, G^*), \cdots, \phi(v_{s_{n2}}, G^*)$ has not been correctly adjusted, the adjusted $\phi(v_s, G^*)$ may not be the final correct value. If we notify the out-neighbors v'_s of v_s's out-neighbors with this incorrect $\phi(v_s, G^*)$ and adjust $\phi(v'_s, G^*)$ accordingly, then, $\phi(v'_s, G^*)$ may not be its final correct value. Therefore, we need further notifications regarding v'_s to adjust $\phi(v'_s, G^*)$ correctly, which means the previous notification is futile. On the other hand, based on Definition 4.3, for v_s, if $\phi(v_{s_{11}}, G^*), \cdots, \phi(v_{s_{n2}}, G^*)$ have been correctly adjusted before v_s notifies its out-neighbors, then $\phi(v_s, G^*)$ can be adjusted correctly and the futile notification problem can be avoided. Inspired by this, we define the shortcut type vertex priority based on the total vertex order γ:

Definition 4.5: (Shortcut Type Vertex Priority) Given the SS-Graph G^* of a shortcut index G', let v_s and v'_s be two shortcut type vertices in G^* and their corresponding shortcuts in G' are (u, v) and (u', v'), respectively. Without loss of generality, assume that $\gamma(u) < \gamma(v)$ and $\gamma(u') < \gamma(v')$. We define v_s has a higher priority than v'_s if:

- $\gamma(u) < \gamma(u')$, or
- $\gamma(u) = \gamma(u')$ and $\gamma(v) < \gamma(v')$

□

If we process the shortcut type vertices following above priority, we can guarantee that $\phi(v_{s_{11}}, G^*), \cdots, \phi(v_{s_{n2}}, G^*)$ have been correctly adjusted before v_s notifies its out-neighbors. With the shortcut weight propagation mechanism and shortcut type vertex priority, we are ready to introduce our algorithm to maintain G^*.

4.2.1 Edge Weight Decrease Case

Our algorithm to handle edge weight decrease case, $\mathsf{DCH_{scs}}$-WDec (scs denotes shortcut-centric for streaming update), is shown in Algorithm 2. $\mathsf{DCH_{scs}}$-WDec uses a priority queue Q to store the shortcut type vertices with $\phi(v_s, G^*)$ changed and the processing priority of shortcut type vertices in Q follows Defini-

Algorithm 2 DCH$_{scs}$-WDec (G^*, G, e, k)

```
1: PriorityQueue Q ← ∅; φ(e, G) ← k;
2: if φ(v_e, G*) > k then
3:    φ(v_e, G*) ← k; Q.push(v_e);
4: while Q ≠ ∅ do
5:    v_s ← Q.pop();
6:    for each v_r ∈ nbr+(v_s) do
7:       v'_s ← the other in-neighbor of v_r except v_s;
8:       v''_s ← nbr+(v_r);
9:       if φ(v''_s, G*) > φ(v_s, G*) + φ(v'_s, G*) then
10:         φ(v''_s, G*) ← φ(v_s, G*) + φ(v'_s, G*);
11:         if v''_s ∉ Q then
12:            Q.push(v''_s);
```

tion 4.5. The priority queue is initialised as $\emptyset$ (line 1). When the weight of an edge e in G is decreased to k, if $\phi(v_e, G^*) > k$ (v_e denotes the corresponding shortcut type vertex of e in G^*), which means $\phi(v_e, G^*)$ does not satisfy the minimum weight property, it adjusts $\phi(v_e, G^*)$ to k and pushes v_e into Q (line 3). After that, it iteratively notifies other shortcut type vertices that may violate the minimum weight property to adjust their $\phi(v_s, G^*)$ following the shortcut weight propagation mechanism (line 4-12). Specifically, DCH$_{scs}$-WDec first pops out the shortcut type vertex v_s from Q and iterates the out-neighbors of v_s (line 5-6). For each out-neighbor v_r of v_s, it retrieves the other in-neighbor v'_s of v_r and the unique out-neighbor v''_s of v_r (line 7-8). If $\phi(v''_s, G^*) > \phi(v_s, G^*) + \phi(v'_s, G^*)$ (line 9), which means $\phi(v''_s, G^*)$ does not satisfy the minimum weight property, it updates $\phi(v''_s, G^*)$ to $\phi(v_s, G^*) + \phi(v'_s, G^*)$ (line 10) and pushes v''_s into Q (line 11-12). DCH$_{scs}$-WDec terminates when Q is empty (line 4).

Example 4.2: Recall the road network G in Figure 1 (a) and consider the weight of (v_8, v_9) decreases from 2 to 1. Figure 2 shows the procedure of DCH$_{scs}$-WDec to maintain G^*. In Figure 2, each rectangle represents a shortcut type vertex. For a shortcut type vertex, its corresponding shortcut in G' and weight are shown in the top and bottom of the rectangle, respectively. As $\phi((v_8, v_9), G)$ decreases from 2 to 1 and $\phi(v_{(v_8,v_9)}, G^*)$ is bigger than 1, $\phi(v_{(v_8,v_9)}, G^*)$ is adjusted from 2 to 1 and $v_{(v_8,v_9)}$ is pushed into Q. Then, $v_{(v_8,v_9)}$ is popped from Q and we check whether $\phi(v_{(v_5,v_9)}, G^*)$ needs to change following the out-neighbor of $v_{(v_8,v_9)}$. Since $\phi(v_{(v_5,v_9)}, G^*) = 4$ is bigger than $\phi(v_{(v_5,v_8)}, G^*) + \phi(v_{(v_8,v_9)}, G^*) = 3$, $\phi(v_{(v_5,v_9)}, G^*)$ is adjusted from 4 to 3 and $v_{(v_5,v_9)}$ is pushed into Q. The procedure continues iteratively until Q is empty. □

Theorem 4.1: *Given a road network G, when the weight of an edge e in G decreases to k, Algorithm 2 computes $G'_{\oplus(e,k)}$ correctly.*

Performance Guarantees. Now, we arrive at the first main result:

Theorem 4.2: *Given a road network G, when the weight of an edge e in G decreases to k, the time complexity of Algorithm 2 to compute $G'_{\oplus(e,k)}$ is $O(|\Delta| \cdot (\log |\Delta| + \mathsf{deg}'_{\mathsf{max}}) + 1)$, where Δ represents the shortcuts s whose $\phi(s, G')$ and $\phi(s, G'_{\oplus(e,k)})$ are different and $\mathsf{deg}'_{\mathsf{max}}$ is the maximum degree of G^*.*

To compute $G'_{\oplus(e,k)}$, we have to explore the shortcuts in Δ at least. Meanwhile, as shown in Theorem 4.2, the number of explored vertices in Algorithm 2 can be bounded by the number of shortcut type vertices for the shortcuts in Δ and their 1-hop neighbors in G^*. In worst case, $|\Delta|$ could be the number of shortcuts in the *shortcut index*. However, as shown in our experiment (Exp-5), $|\Delta|$ could be very small in practice. Therefore, Algorithm 2 is efficient regarding computing $G'_{\oplus(e,k)}$.

Algorithm 3 DCH$_{scs}$-WIncDirect (G^*, G, e, k)

```
1: PriorityQueue Q ← ∅; φ(e, G) ← k;
2: φ ← minWeight(G*, v_e);
3: if φ(v_e, G*) > φ then
4:    φ(v_e, G*) ← φ; Q.push(v_e);
5: while Q ≠ ∅ do
6:    v_s ← Q.pop();
7:    for each v_r ∈ nbr+(v_s) do
8:       v'_s ← nbr+(v_r); φ ← minWeight(G*, v'_s);
9:       if φ(v'_s, G') > φ then
10:         φ(v'_s, G') ← φ; if v'_s ∉ Q then Q.push(v'_s);
11: procedure minWeight(G*, v_s)
12:    φ ← φ(s, G);
13:    for each v_r ∈ nbr−(v_s) do
14:       let v_{s1} and v_{s2} be the two in-neighbors of v_r;
15:       if φ > φ(v_{s1}, G*) + φ(v_{s2}, G*) then
16:          φ ← φ(v_{s1}, G*) + φ(v_{s2}, G*);
17:    return φ;
```

4.2.2 Edge Weight Increase Case

A 3-hop neighbors bounded algorithm. Following the shortcut weight propagation mechanism, we can directly obtain an algorithm for edge weight increase case (Algorithm 3).

DCH$_{scs}$-WIncDirect uses a priority queue Q to store the shortcut type vertices with $\phi(v_s, G^*)$ changed and initialises it as $\emptyset$ (line 1). After updating the weight of edge e with k (line 1), it checks whether $\phi(v_e, G^*)$ satisfies the minimum weight property (v_e denotes the corresponding shortcut type vertex of e in G^*). If $\phi(v_e, G^*)$ is larger than the minimum weight ϕ computed by procedure minWeight, DCH$_{scs}$-WIncDirect updates $\phi(v_e, G^*)$ with ϕ and pushes it into Q (line 3-4). Then, it iteratively propagates the shortcut weight and further notifies the shortcut type vertices through its out-neighbor. DCH$_{scs}$-WIncDirect terminates when Q is empty (line 5). Procedure minWeight computes $\min\{\phi(s, G), \phi(v_{s_{11}}, G^*) + \phi(v_{s_{12}}, G^*), \cdots, \phi(v_{s_{n1}}, G^*) + \phi(v_{s_{n2}}, G^*)\}$ for the given shortcut type vertex v_s following Definition 4.3. It first retrieves $\phi(s, G)$ as the minimum weight ϕ (line 12). Then, for each in-neighbor v_r of v_s (line 13), it computes the sum of $\phi(v_{s_1}, G^*)$ and $\phi(v_{s_2}, G^*)$, where v_{s_1} and v_{s_2} are the two in-neighbors of v_r. If the sum is less than the ϕ, ϕ is updated with the sum (line 15-16). It returns the minimum weight in line 17.

Algorithm 3 is intuitive for edge weight increase case. However, for an edge weight increase in G, the number of explored vertices in Algorithm 3 cannot be bounded by the number of shortcut type vertices for the shortcuts in Δ and their 1-hop neighbors in G^* as Algorithm 2. This is because two types of shortcut type vertices are pushed in Q in Algorithm 3, namely:

- Type-1: The set of shortcut type vertices that the weight of corresponding shortcuts in G' and $G'_{\oplus(e,k)}$ are different, i.e., the corresponding shortcut type vertices for the shortcuts in Δ.
- Type-2: The set of shortcut type vertices that are (1) 2-hop out-neighbors of Type-1 vertices; and (2) not Type-1 vertices.

In Algorithm 3, every v_s in Q is processed by minWeight. As a result, for each Type-2 shortcut type vertex, its 2-hop in-neighbors are explored in line 14 because we do not know whether v_s is a Type-2 vertex before invoking minWeight. Since a Type-2 shortcut type vertex is a 2-hop neighbor of a Type-1 shortcut type vertex, the 4-hop neighbors of some Type-1 vertices need to be explored. This means the number of explored shortcut type vertices in Algorithm 3 is only bounded by the number of shortcut type vertices for the shortcuts in Δ and their 3-hop neighbors in G^*.

A 1-hop neighbors bounded algorithm. As discussed above, the reason that Algorithm 3 has to push the Type-2 shortcut type vertices into Q is that Algorithm 3 cannot determine whether

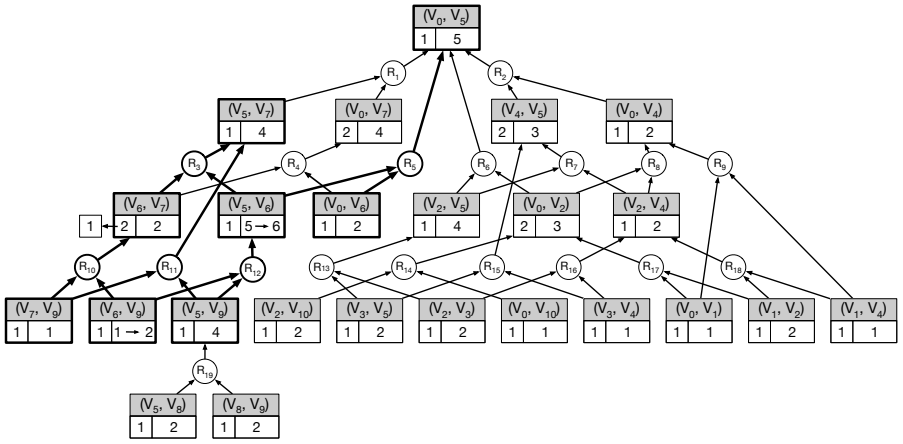

Figure 3: SS-Graph G^* and $\mathsf{DCH_{scs}}$-WInc for (v_6, v_9) with weight increase

$\phi(v_s, G^*)$ will change without invoking minWeight. In other words, if we can determine whether $\phi(v_s, G^*)$ will change in $O(1)$ time, then we can avoid exploring the neighbors of Type-2 shortcut type vertices and the number of explored shortcut type vertices can be bounded as Algorithm 2. Inspired by this, we define:

Definition 4.6: (Shortcut Weight Counter) Given a shortcut type vertex v_s, the shortcut weight counter of v_s, denoted by $c_\phi(v_s)$, records the number of values in $\phi(s, G)$, $\phi(v_{s_{11}}, G^*) + \phi(v_{s_{12}}, G^*), \cdots, \phi(v_{s_{n1}}, G^*) + \phi(v_{s_{n2}}, G^*)$ that equals the value of $\phi(v_s, G^*)$, where $s, v_{s_{11}}, \ldots$ follows Definition 4.3. □

Lemma 4.4: *Given a shortcut type vertex v_s in G^*, $c_\phi(v_s) \geq 1$.*

According to Lemma 4.4, when the weight of an edge e in G increases, if we can maintain $\phi(s, G)$, $\phi(v_{s_{11}}, G^*), \ldots, \phi(v_{s_{n2}}, G^*)$ and $c_\phi(v_s)$ correctly as their definitions, then, when we process v_s, if $c_\phi(v_s)$ is still not less than 1, we know immediately that the value of $\phi(v_s, G^*)$ will not change after the weight increase of e; otherwise, the current value of $\phi(v_s, G^*)$ will definitely change. In this way, we can determine whether $\phi(v_s, G^*)$ will change after the weight increase of e in $O(1)$ time. The remaining problem is how to maintain them without introducing extra theoretical time complexity. We show it in Algorithm 4.

The framework of Algorithm 4 is similar to that of Algorithm 3. It utilizes a priority queue Q to store the shortcut type vertices that need to be processed (line 1). Then, it iteratively pops the shortcut type vertex v_s out from Q (line 8), and propagates the change of $\phi(v_s, G^*)$ to its out-neighbors and further pushes the shortcut type vertices whose $\phi(v_s, G^*)$ are updated into Q (line 9-12). The procedure terminates when Q is empty (line 7). Different from Algorithm 3, we introduce $c_\phi(v_s)$ and a new procedure updateWeight in Algorithm 4. For each shortcut type vertex v_s whose weight may change, we first check whether $c_\phi(v_s)$ is less than 1 (line 5 and line 11). If it is less than 1, we invoke updateWeight to compute the new value of $\phi(v_s, G^*)$ and update $c_\phi(v_s)$ regarding the new value of $\phi(v_s, G^*)$ (line 8).

Procedure updateWeight is used to compute the new value of $\phi(v_s, G^*)$ and maintain $c_\phi(v_s)$ accordingly. Specifically, for a shortcut type vertex v_s, it first iterates the out-neighbors of v_s (line 14) and for each out-neighbor v_r of v_s, it retrieves the two in-neighbors v_{s_1} and v_{s_2} of v_s and the out-neighbor v_{s_3} of v_r (line 15). If $\phi(v_{s_3}, G^*) = \phi(v_{s_1}, G^*) + \phi(v_{s_2}, G^*)$, $c_\phi(v_{s_3})$ decreases by 1 according to Definition 4.6 (line 16-17). After that, updateWeight computes the new value of $\phi(v_s, G^*)$ and maintains $c_\phi(v_s)$ regarding the new value of $\phi(v_s, G^*)$ by iterating the in-neighbors of v_s and computing the minimum weight based on Definition 4.3 (line 19-25).

Algorithm 4 $\mathsf{DCH_{scs}}$-WInc (G^*, G, e, k)

```
1:  PriorityQueue Q ← ∅;
2:  if φ(v_e, G*) = φ(e, G) then
3:      c_φ(v_e) ← c_φ(v_e) − 1;
4:  φ(e, G) ← k;
5:  if c_φ(v_e) < 1 then
6:      Q.push(v_e);
7:  while Q ≠ ∅ do
8:      v_s ← Q.pop(); updateWeight(G*, v'_s);
9:      for each v_r ∈ nbr+(v_s) do
10:         v'_s ← nbr+(v_r);
11:         if c_φ(v'_s) < 1 then
12:             if v'_s ∉ Q then Q.push(v'_s);
13: procedure updateWeight(G*, v_s)
14:     for each v_r ∈ nbr+(v_s) do
15:         v_{s1}, v_{s2} ← nbr−(v_r); v_{s3} ← nbr+(v_r);
16:         if φ(v_{s3}, G*) = φ(v_{s1}, G*) + φ(v_{s2}, G*) then
17:             c_φ(v_{s3}) ← c_φ(v_{s3}) − 1;
18:     φ ← φ(s, G); c_φ(v_s) ← 1;
19:     for each v_r ∈ nbr−(v_s) do
20:         v_{s1}, v_{s2} ← nbr−(v_r);
21:         if φ > φ(v_{s1}, G*) + φ(v_{s2}, G*) then
22:             φ ← φ(v_{s1}, G*) + φ(v_{s2}, G*); c_φ(v_s) ← 1;
23:         else if φ = φ(v_{s1}, G*) + φ(v_{s2}, G*) then
24:             c_φ(v_s) ← c_φ(v_s) + 1;
25:     φ(v_s, G*) ← φ;
```

Note that we can easily extend Algorithm 2 to maintain the shortcut weight counter for the weight decrease case without affecting its time complexity. We omit the details for brevity.

Example 4.3: Figure 3 shows the SS-Graph G^* with shortcut weight counter for the road network G in Figure 1 (a). For each shortcut type vertex, Figure 3 shows its shortcut weight counter and its weight in the left side and right side at the bottom of each corresponding rectangle. Take $v_{(v_0,v_7)}$ as an example, since $\phi((v_0, v_7), G) = 4$ and $\phi(v_{(v_6,v_7)}, G^*) + \phi(v_{(v_0,v_6)}, G^*) = 4$, $c_\phi(v_{(v_0,v_7)})$ is 2 and $\phi(v_{(v_0,v_7)}, G^*)$ is 4. When the weight of (v_6, v_9) increases from 1 to 2, the dark lines in Figure 3 highlight the procedure of $\mathsf{DCH_{scs}}$-WInc to maintain G^*. As $\phi((v_6, v_9), G)$ increases from 1 to 2 and $\phi(v_{(v_6,v_9)}, G^*)$ is 1, $c_\phi(v_{(v_6,v_9)})$ decreases by 1 and is less than 1, updateWeight is invoked with $v_{(v_6,v_9)}$. In updateWeight, $c_\phi(v_{(v_6,v_7)})$ decreases from 2 to 1, $c_\phi(v_{(v_5,v_6)})$ decreases from 1 to 0 (not shown in Figure 3 as it is an intermediate state), $\phi(v_{(v_6,v_9)}, G^*)$ increases from 1 to 2 and $c_\phi(v_{(v_6,v_9)})$ is 1 regarding the new value of $\phi(v_{(v_6,v_9)}, G^*)$. Since the value of $\phi(v_{(v_6,v_9)}, G^*)$ changes, $v_{(v_6,v_9)}$ is pushed into Q. The procedure continues iteratively until Q is empty. □

Theorem 4.3: *Given a road network G, when the weight of an edge e in G increases to k, Algorithm 4 computes $G'_{\oplus(e,k)}$ correctly.*

Performance Guarantees. We arrived at the second main result:

Theorem 4.4: *Given a road network G, when the weight of an edge e in G increases to k, the time complexity of Algorithm 4 to compute $G'_{\oplus(e,k)}$ is $O(|\Delta| \cdot (\log|\Delta| + \mathsf{deg}'_{\mathsf{max}}) + 1)$.*

4.3 Extensions

Extension for directed road networks. Our techniques can be extended to support directed road networks. For a directed road network G, CH contracts a vertex v similarly as a undirected road network as follows: CH contracts v by inserting a directed shortcut $\langle u, w\rangle$ (resp. updating the weight of $\langle u, w\rangle$) between any pair of u and w where $u \in \mathsf{nbr}^-(v)$, $w \in \mathsf{nbr}^+(v)$, $\gamma(u) > \gamma(v)$ and $\gamma(w) > \gamma(v)$. Therefore, we extend Definition 4.1 and Definition 4.2 for directed road networks as follows:

Given a shortcut index G' for a directed road network G, for a directed shortcut $\langle u, w\rangle \in E(G')$, let $\langle u, v\rangle$ and $\langle v, w\rangle$ be another two directed shortcuts in G', we call $\langle u, v\rangle$ and $\langle v, w\rangle$ are a supporting shortcut pair of $\langle u, w\rangle$ if $\gamma(v) < \gamma(u)$ and $\gamma(v) < \gamma(w)$. For the SS-Graph G^* of G', each shortcut type vertex in G^* corresponds to a directed shortcut in G' and each supporting relation type vertex in G^* corresponds to a supporting shortcut pair instance in G'. The shortcut type vertices and supporting relation type vertices are connected the in the same way as Definition 4.2. For the edge weight updates on the directed road network G, we just run our algorithms on G^* for the directed road network and we can guarantee that G^* is maintained correctly. The correctness of the algorithms can be proved similarly as the undirected case.

Extension for vertex/edge update. Our techniques can be extended to support vertex/edge update. Since a vertex deletion/insertion can be regarded as a sequence of edge deletions/insertion, we mainly focus on edge update here.

Regarding the edge deletion, our techniques can handle it directly. Given a deleted edge e, we can treat the deletion of e as the weight of e increases to $+\infty$. Thus, we can use our algorithm for edge weight increase to address this case directly.

Regarding the edge insertion, assume that we insert an edge $e = (u, v)$ with weight k in G, if (u, v) is already a shortcut in G', then, we just treat it as the weight of (u, v) decrease to k and use Algorithm 2 to address this case directly. If (u, v) is not a shortcut in G' (assume $\gamma(u) < \gamma(v)$), then, we first add a shortcut type vertex for (u, v) with weight k in G^* (resp. a shortcut (u, v) with weight k in G'). After that, for each $w \in \mathsf{nbr}(u)$ with $\gamma(u) < \gamma(w)$, if there is a shortcut type vertex for (v, w) in G^*, then we check whether $\phi(v_{(v,w)}, G^*) \leq \phi(v_{(u,v)}, G^*) + \phi(v_{(u,w)}, G^*)$. If $\phi(v_{(v,w)}, G^*) \leq \phi(v_{(u,v)}, G^*) + \phi(v_{(u,w)}, G^*)$, we do nothing and continue our procedure. Otherwise, we treat it as the weight of (v, w) decreases to $\phi(v_{(u,v)}, G^*) + \phi(v_{(u,w)}, G^*)$ and invoke Algorithm 2 to adjust the weight of $v_{(v,w)}$ and other shortcut type vertices whose weight may be changed due to the weight decrease of $v_{(v,w)}$. If there is no shortcut type vertex for (v, w) in G^*, then we add a shortcut type vertex for (v, w) with weight $\phi(v_{(u,v)}, G^*) + \phi(v_{(u,w)}, G^*)$ in G^* (resp. a shortcut (v, w) with weight $\phi((u, v), G') + \phi((u, w), G')$ in G') and recursively process (v, w) following the above procedure. We finish the processing when all the neighbors w of u with $\gamma(u) < \gamma(w)$ are processed.

4.4 Algorithms without Materialized G*

In the above discussion, we always assume that G^* has been materialized and all the algorithms are designed based on the materialized G^*. However, the space consumption of G^* could be large for big road networks. In this section, we remove this assumption and propose efficient algorithms without materialized G^*.

Algorithm 5 $\mathsf{DCH}^+_{\mathsf{scs}}$-$\mathsf{WDec}$ (G', $e = (u, v)$, k)

```
1: PriorityQueue Q ← ∅; φ((u,v), G) ← k;
2: if φ((u,v), G') > k then                    // w.l.o.g., assume γ(u) < γ(v)
3:    φ((u,v), G') ← k; Q.push((u,v));
4: while Q ≠ ∅ do
5:    (u,v) ← Q.pop();
6:    for each w ∈ nbr(u) and γ(u) < γ(w) and w ≠ v do
7:       if γ(w) < γ(v) then
8:          if φ((w,v), G') > φ((u,w), G') + φ((u,v), G') then
9:             φ((w,v), G') ← φ((u,w), G') + φ((u,v), G');
10:            if (w,v) ∉ Q then Q.push((w,v));
11:      else
12:         if φ((v,w), G') > φ((u,w), G') + φ((u,v), G') then
13:            φ((v,w), G') ← φ((u,w), G') + φ((u,v), G');
14:            if (v,w) ∉ Q then Q.push((v,w));
```

Space consumption of G^*. We first analyze the space consumption of G^* from theoretical aspect.

Lemma 4.5: *Given G^* of a shortcut index G', the space of G^* is $O(|E(G')| + \sum_{v \in V(G')}(|\mathsf{Nbr}(v)| \cdot (|\mathsf{Nbr}(v)| - 1)/2))$, where $\mathsf{Nbr}(v) = \{u | u \in \mathsf{nbr}(v) \wedge \gamma(u) > \gamma(v)\}$.*

According to Lemma 4.5, the space consumption of G^* is much larger than that of G'. We also compare the space consumption of G^* and G' for the datasets used in our experiments (Exp-4, Section 6). The experimental results confirm the theoretical analysis in Lemma 4.5. For example, on *CAL*, G^* consumes 12.7 times more space than G'. The large space consumption of G^* limits the scalability of our algorithm to handle big road networks. In this section, we propose efficient algorithms to handle the edge weight update without introducing any extra memory consumption. For brevity, we only show the weight decrease case and the proposed techniques can be easily extended to other cases.

Revisiting Algorithm 2, for a shortcut type vertex v_s, we leverage G^* to retrieve its out-neighbor v_r in line 6. For the computed v_r regarding v_s, we leverage G^* to retrieve its in-neighbors and out-neighbor in line 7-8. Suppose v_s represents the shortcut (u, v) in G'. Essentially, these operations in Algorithm 2 regarding v_s are equal to retrieving the shortcuts that (u, v) supports and the shortcut forming a supporting shortcut pair with (u, v). According to Definition 4.1, we can retrieve these shortcuts for (u, v) based on G' directly, which means we can achieve the same goal of Algorithm 2 based on G' without the materialized G^*.

Algorithm. Following the above idea, our new algorithm is shown in Algorithm 5. Algorithm 5 shares the same framework with Algorithm 2. The differences locate in line 6-14. In Algorithm 5, for a shortcut (u, v), to retrieve all the shortcuts that (u, v) supports, we iterate all the neighbors w of u with $\gamma(w) > \gamma(u)$ and the shortcut $(v, w)/(w, v)$ are the shortcuts that (u, v) supports based on Definition 4.1 (line 6). For $(v, w)/(w, v)$, another shortcut except (u, v) supporting it is (u, w). We retrieve it in this way in line 8-9 and line 12-13.

5. BATCH UPDATE ALGORITHM

5.1 A Direct Batch Update Algorithm

Since a batch of weight updates can be treated as a series of streaming weight updates, a direct approach to process the batch weight update is to leverage the techniques designed for the streaming update. The algorithm for batch weight decrease update is shown in Algorithm 6. Since the pseudocode is self-explanatory,

Algorithm 6 $\mathsf{DCH_{scb}}$-WDec $(G^*, \{(e_1, k_1), \ldots, (e_n, k_n))\})$

```
1: PriorityQueue Q ← ∅;
2: for each e_i do
3:    φ(e_i, G) ← k_i;
4:    if φ(v_{e_i}, G*) > k then
5:       φ(v_{e_i}, G*) ← k; Q.push(v_{e_i});
6: line 4-12 of Algorithm 2;
```

we omit the detailed description for brevity. The batch weight increase update can be handled similarly and the pseudocode is omitted for the same reason. We have the following theorem:

Theorem 5.1: *Given a road network G and a batch of edge weight decrease/increase updates U, G' can be maintained correctly in $O(|U| + |\Delta| \cdot (\log |\Delta| + \mathsf{deg'_{max}}))$, where $|U|$ is the number of edge weight decrease/increase updates in U.*

At first glance, Algorithm 6 is an ideal algorithm for batch update. Besides the inevitable exploration of the edges in U, the number of explored vertices in Algorithm 6 can be bounded by the number of shortcut type vertices for the shortcuts in Δ and their 1-hop neighbors and the time complexity of Algorithm 6 has a similar form to that of Algorithm 2 and Algorithm 4. However, for the batch update case, $|\Delta|$ could be large. Consequently, the factor $\log |\Delta|$ in Theorem 5.1 could be large as well, which limits the efficiency of Algorithm 6 for batch update. In the following, we aim to eliminate the $\log |\Delta|$ factor in the time complexity of Algorithm 6 to further improve the efficiency to handle batch update.

5.2 An Improved Batch Update Algorithm

In this section, we present our approach for the batch update. In Algorithm 6, we follow the shortcut type vertex priority and use a priority query Q to store the shortcut type vertices with weight change. With the help of Q, for a shortcut type vertex v_s, we can guarantee that $\phi(v_{s_{11}}, G^*), \cdots, \phi(v_{s_{n2}}, G^*)$ have been correctly adjusted before v_s notifies its out-neighbors as discussed in Section 4.2. However, due to the introduction of Q, the factor $\log |\Delta|$ is involved in the time complexity of Algorithm 6. Therefore, to eliminate the $\log |\Delta|$ factor, we have to design a new structure with which we can still guarantee that the order of processing shortcut type vertices with weight change to replace Q.

Recall that to guarantee that $\phi(v_{s_{11}}, G^*), \cdots, \phi(v_{s_{n2}}, G^*)$ have been correctly adjusted before v_s notifies its out-neighbors, we only need to guarantee that we have an order in which v_s ranks after $v_{s_{11}}, \cdots, v_{s_{n2}}$ and process the shortcut type vertices based on the order. Meanwhile, it is clear that G^* is a directed acyclic graph. Hence, we define:

Definition 5.1: (SS-Graph Level) Given a SS-Graph G^*, for a shortcut type vertex v_s in G^*, the SS-Graph level of v_s, denoted by $l(v_s)$, is defined as $l(v_s) =$

$$\begin{cases} \max\{l(v'_s) | v'_s \in \cup_{v_r \in \mathsf{nbr^+}(v_s)} \mathsf{nbr^+}(v_r)\} + 1, & \mathsf{nbr^+}(v_s) \neq \emptyset \\ 1, & \mathsf{nbr^+}(v_s) = \emptyset \end{cases}$$

□

Lemma 5.1: *Given a SS-Graph G^*, for a shortcut type vertex v_s, let v'_s be a shortcut type vertex positioned after v_s in any directed path in G^*, then, $l(v'_s) < l(v_s)$.*

Following Lemma 5.1, if we process the shortcut type vertices in non-ascending order of SS-Graph levels, we can still guarantee that $\phi(v_{s_{11}}, G^*), \cdots, \phi(v_{s_{n2}}, G^*)$ have been correctly updated when v_s is processed, as they have higher levels than v_s. The remaining problem is how to implement the shortcut weight propagation mechanism efficiently based on SS-Graph levels. According to the procedure of shortcut weight propagation mechanism, only shortcut type vertices with big SS-Graph levels notify the shortcut type vertices with small SS-Graph levels. Therefore, we utilize a bucket array to store the shortcut type vertices with weight change based on their SS-Graph levels and process shortcut type vertices in the bucket array in non-ascending order of their SS-Graph levels. When a new shortcut type vertex is notified by the shortcut weight propagation mechanism, we distribute it in the corresponding bucket based on its level and iteratively continue the procedure until all $\phi(v_s, G^*)$ are adjusted.

Algorithm 7 $\mathsf{DCH^+_{scb}}$-WDec $(G^*, U = \{(e_1, k_1), \ldots, (e_n, k_n))\})$

```
1: BinArray B ← ∅;
2: for each e_i ∈ U do
3:    φ(e_i, G) ← k;
4:    if φ(v_{e_i}, G*) > k then
5:       φ(v_{e_i}, G*) ← k;
6:       B[l(v_{e_i})] ← B[l(v_{e_i})] ∪ v_{e_i};
7: i = argmax_{i∈ℕ+}{B[i] ≠ ∅};
8: while B ≠ ∅ do
9:    for each v_s ∈ B[i] do
10:      for each v_r ∈ nbr+(v_s) do
11:         v'_s ← the other in-neighbor of v_r except v_s;
12:         v''_s ← nbr+(v_r);
13:         if φ(v''_s, G*) > φ(v_s, G*) + φ(v'_s, G*) then
14:            φ(v''_s, G*) ← φ(v_s, G*) + φ(v'_s, G*);
15:            if v''_s ∉ B[l(v''_s)] then
16:               B[l(v''_s)] ← B[l(v''_s)] ∪ v''_s;
17:   i ← i − 1;
```

Algorithm. Following the above idea, we show the algorithm to handle the batch weight decrease in Algorithm 7. As our techniques can be directly extended for batch weight increase case, the pseudocode for the batch weight increase is omitted for clearness.

Instead of the priority queue Q, Algorithm 7 uses a bucket array B to store the shortcut type vertices with weight change and B is initialized as $\emptyset$ (line 1). For each edge e_i with weight decrease, it first updates e_i's weight in G as k (line 3) and if $\phi(v_{e_i}, G^*) > k$, where v_{e_i} is the corresponding shortcut type vertex of e_i, $\phi(v_{e_i}, G^*)$ is updated with k and v_{e_i} is inserted into the $l(v_{e_i})$-th bucket of B (line 4-6). After that, Algorithm 7 iteratively conducts the shortcut weight update propagation through B. Specifically, it first computes the maximum level i such that the corresponding $B[i]$ is not empty (line 7). Then, for each shortcut type vertex v_s (line 9), it inserts all the shortcut type vertices whose weights may change due to the weight change of v_s in B similar to Algorithm 2 and continues the procedure (line 10-17). It terminates when B is empty (line 8). For the SS-Graph level, we can compute it based on Definition 5.1 directly. Note that for a specific SS-Graph G^*, the level of a shortcut type vertex is fixed and we only need to compute it once for G^*.

Theorem 5.2: *Given a road network G, for a batch of edge weight decrease/increase updates U, G' can be correctly maintained in $O(|U| + \beta + |\Delta| \cdot \mathsf{deg'_{max}})$, where β is the maximum SS-Graph level of G^*.*

6. EXPERIMENT

In this section, we compare our algorithms with the state-of-the-art methods. All experiments are conducted on a machine with an Intel Xeon 2.8GHz CPU (10 cores) and 256 GB main memory running Linux (Red Hat Linux 4.4.7, 64bit).

Datasets. We use eight publicly available real road networks from DIMACS (`http://www.dis.uniroma1.it/challenge9/download.shtml`). In each road network, ver-

Table 1: Datasets used in Experiments

DataSet G	Region	$\|V(G)\|$	$\|E(G)\|$	β
NY	New York City	264,346	733,846	767
COL	Colorado	435,666	1,057,066	558
FLA	Florida	1,070,376	2,712,798	580
CAL	California and Nevada	1,890,815	4,657,742	939
E-US	Eastern US	3,598,623	8,778,114	1,372
W-US	Western US	6,262,104	15,248,146	1,489
C-US	Central US	14,081,816	34,292,496	2,982
US	United States	23,947,347	58,333,344	3,976

tices represent intersections between roads, edges correspond to roads or road segments and the weight of an edge is the transit time between two vertices. Table 1 provides the details about these datasets. Table 1 also shows the value of β in Theorem 5.2 for each dataset and it is clear that β is small in practice.

Algorithms. We implement and compare the following algorithms. All the algorithms are implemented in C++ 11, using g++ complier with -O3 optimization.

- $\mathsf{DCH}_{\mathsf{scs}}$-WDec/$\mathsf{DCH}_{\mathsf{scs}}$-WInc: our algorithm to handle streaming weight decrease/increase with materialised G^*.
- $\mathsf{DCH}^{+}_{\mathsf{scs}}$-WDec/$\mathsf{DCH}^{+}_{\mathsf{scs}}$-WInc: our algorithm to handle streaming weight decrease/increase without materialised G^*.
- $\mathsf{DCH}_{\mathsf{scb}}$-WDec/$\mathsf{DCH}_{\mathsf{scb}}$-WInc: the direct algorithm for batch weight decrease/increase update.
- $\mathsf{DCH}^{+}_{\mathsf{scb}}$-WDec/$\mathsf{DCH}^{+}_{\mathsf{scb}}$-WInc: the improved algorithm for batch weight decrease/increase update.
- $\mathsf{DCH}_{\mathsf{vcs}}$/$\mathsf{DCH}_{\mathsf{vcb}}$: the state-of-the-art algorithm for streaming/batch update. Note that [20] only discusses the streaming update. In our experiment, we implement $\mathsf{DCH}_{\mathsf{vcb}}$ based on $\mathsf{DCH}_{\mathsf{vcs}}$ in a similar way of Algorithm 6.
- CRP: The algorithm for shortest path query proposed in [17]. It also supports the dynamic maintenance of its index structure. [1]
- Rebuild: rebuild the *shortcut index* from scratch.

Exp-1: Efficiency for streaming weight decrease update. In this experiment, we compare the efficiency of the algorithms for streaming weight decrease update. To test the efficiency, we generate 9 groups of weight decrease instances for each dataset as follows: for the group i, we randomly select 1000 edges in the road network. For each selected edge e, we decrease its weight to $(1.0 - 0.i) \times \phi(e)$. We report the average processing time for the 1000 weight decrease instances for each group and the results are shown in Figure 4.

According to the results shown in Figure 4, we make the following observations. First, for all test cases, CRP consumes the most time. $\mathsf{DCH}_{\mathsf{vcs}}$ is faster than CRP but consumes much more time than our algorithms. As shown in Figure 4, $\mathsf{DCH}_{\mathsf{vcs}}$ consumes nearly 2-3 orders of magnitude more time than $\mathsf{DCH}_{\mathsf{scs}}$-WDec and $\mathsf{DCH}^{+}_{\mathsf{scs}}$-WDec for all test cases on all datasets. This is because $\mathsf{DCH}_{\mathsf{vcs}}$ shares the same time complexity with reconstructing the *shortcut index* from scratch for an edge weight decreases and it involves lots of unfruitful computation in the maintenance. Second, $\mathsf{DCH}_{\mathsf{scs}}$-WDec and $\mathsf{DCH}^{+}_{\mathsf{scs}}$-WDec consume similar time and their average processing times are much smaller than that of $\mathsf{DCH}_{\mathsf{vcs}}$. This is because $\mathsf{DCH}_{\mathsf{scs}}$-WDec and $\mathsf{DCH}^{+}_{\mathsf{scs}}$-WDec share the same time complexity and they avoid the unfruitful computation in the maintenance of $\mathsf{DCH}_{\mathsf{vcs}}$ due to the introduction of shortcut-centric paradigm. Another thing that needs to mention is the running time of CRP in our paper is slower than that in [17]. The following three factors together lead to the difference of the running time: (1) As CRP is implemented by ourselves, some hidden tricks in [17] may be missed in our implementation. (2) CRP involves several parameters closely related to the query and update efficiency, such as the number of levels and cell sizes. However, [17] does not provide clear guidelines to set these parameters for a given dataset. Therefore, we set these parameters uniformly for all datasets without tuning. It is also a possible source leading to the different running times of our paper and that of [17]. (3) The experimental environments of [17] and our paper are different, which has a significant effect on the running time of the algorithm. We also evaluate the query efficiency of CRP and CH and the results are shown in Table 4 of the long version of our paper [2]. Based on our experimental results regarding query efficiency, not only the running time of CRP in this paper is 2 orders of magnitude slower than that in [17], but the running time of CH in our paper is also 1∼2 orders of magnitude slower than that in [17]. It means the experimental environments affect the running time of both algorithms. On the other hand, CH is 1∼2 orders of magnitude faster than CRP regarding the query efficiency in both papers. From this point, the experiment results of [17] are consistent with our paper.

Exp-2: Efficiency for streaming weight increase update. In this experiment, we evaluate the performance of the algorithms for streaming weight increase update. To test the efficiency, we generate 9 groups of weight increase instances for each dataset as follows: for the group i, we randomly select 1000 edges. For each selected edge e, we increase its weight to $(i + 1.0) \times \phi(e)$. We report the average processing time for the 1000 weight increase instances for each group and Figure 5 shows the results.

Figure 5 shows that CRP consumes the most time in most cases while $\mathsf{DCH}_{\mathsf{vcs}}$ is faster than CRP but consumes much more time than our algorithms. $\mathsf{DCH}_{\mathsf{scs}}$-WInc and $\mathsf{DCH}^{+}_{\mathsf{scs}}$-WInc consumes similar time for each update and outperform $\mathsf{DCH}_{\mathsf{vcs}}$ by 2-3 orders of magnitude. The reasons are similar as presented in Exp-1.

Exp-3: Efficiency for batch weight decrease/increase update. In this experiment, we evaluate the efficiency of the algorithms regarding batch update. We select 0.001% to 100% edges from each dataset randomly as a batch and decrease/increase the weight of each selected edge e to $0.9 \times \phi(e)$/$2.0 \times \phi(e)$. We report the processing time of the algorithms and show the results in Figure 6 and Figure 7, respectively.

Figure 6 shows that the processing time of each algorithm increases as the number of edges in the batch increases. For our proposed algorithms, $\mathsf{DCH}^{+}_{\mathsf{scb}}$-WDec consumes less time than $\mathsf{DCH}_{\mathsf{scb}}$-WDec and the processing time gap increases as the number of edges in the batch increases. For $\mathsf{DCH}_{\mathsf{vcb}}$, it consumes much more time than our proposed algorithms when the number of edge in the batch is small, but the gap decreases as the number of edges in the batch increase. When the percentage of selected edges is more than 10^{-1}, $\mathsf{DCH}_{\mathsf{vcb}}$ even outperforms $\mathsf{DCH}_{\mathsf{scb}}$-WDec on *C-US*. This is because as the number of edges in the batch increases, the number of shortcuts with weight update also increases, which weakens the advantages of $\mathsf{DCH}_{\mathsf{scb}}$-WDec against $\mathsf{DCH}_{\mathsf{vcb}}$. However, $\mathsf{DCH}^{+}_{\mathsf{scb}}$-WDec always outperforms $\mathsf{DCH}_{\mathsf{vcb}}$ benefiting from the introduction of bucket array. In addition, even when the weight of all the edges in the road network are updated, $\mathsf{DCH}^{+}_{\mathsf{scb}}$-WDec has a similar processing time as Rebuild. The results show that $\mathsf{DCH}^{+}_{\mathsf{scb}}$-WDec is very suitable for batch update. Figure 7 shows that $\mathsf{DCH}^{+}_{\mathsf{scb}}$-WInc outperforms $\mathsf{DCH}_{\mathsf{scb}}$-WInc and $\mathsf{DCH}_{\mathsf{vcb}}$ in all cases and has a similar processing time as Rebuild when all the weight of all the edges are updated.

[1] We implement CRP by ourselves as the code of [17] cannot be provided publicly. We set its parameters uniformly for all datasets.

[2] https://www.dropbox.com/s/3s7n9u1sv315785/DynSDist.pdf?dl=0

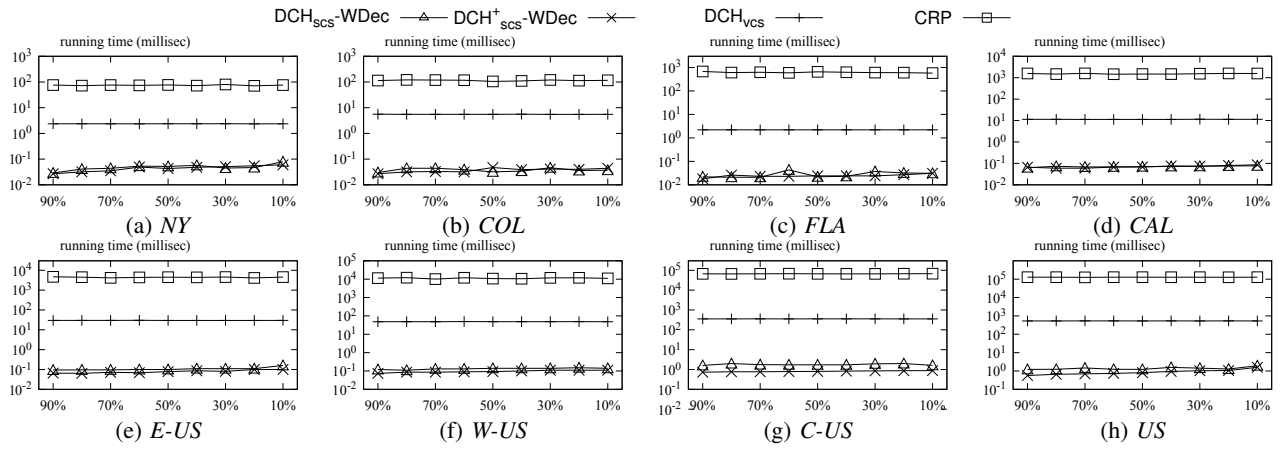

Figure 4: Processing Time for Streaming Update - Weight Decrease (Varying the percentage of weight decrease for an edge)

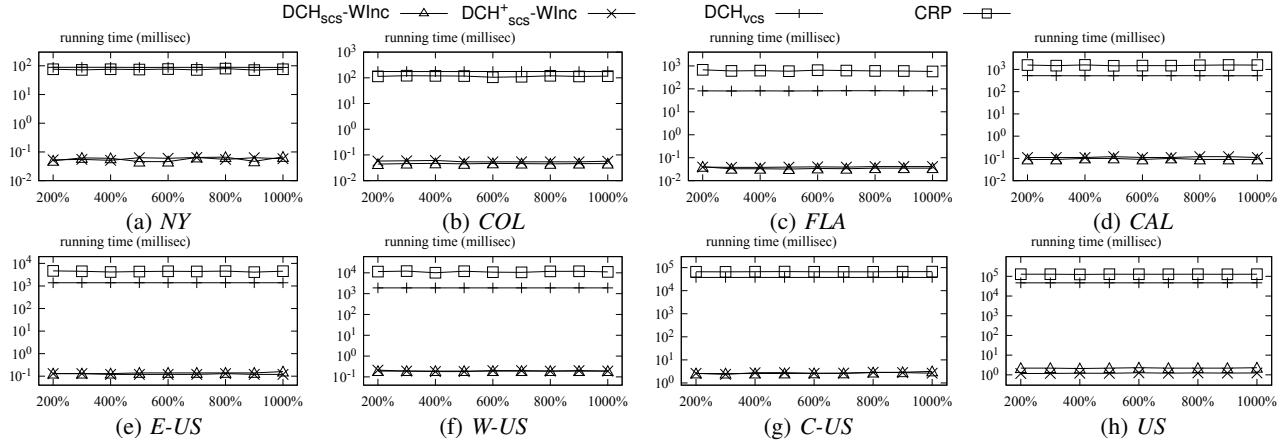

Figure 5: Processing Time for Streaming Update - Weight Increase (Varying the percentage of weight increase for an edge)

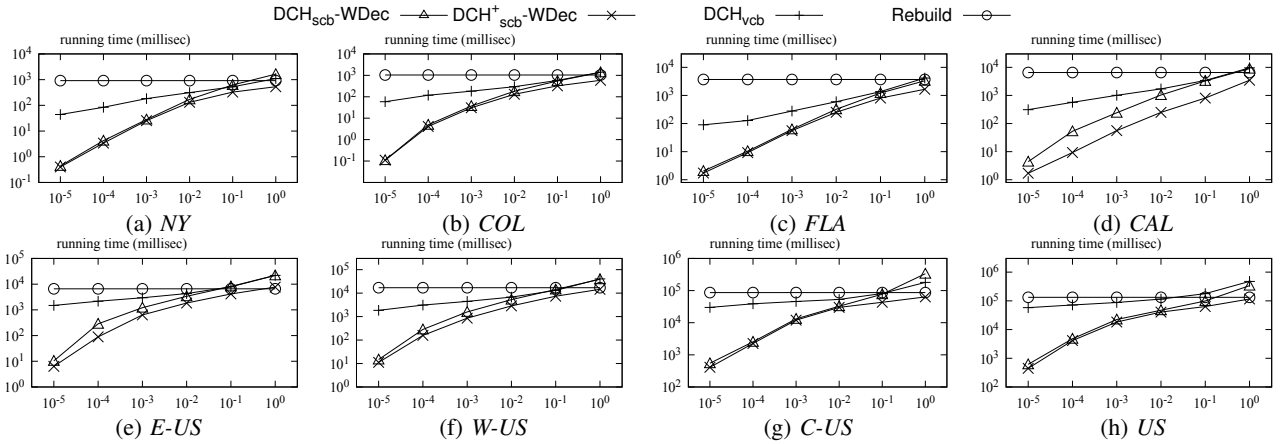

Figure 6: Processing Time for Batch Update - Weight Decrease (Varying the percentage of edges with weight decrease)

Table 2: Space consumption of G' vs G^*

Dataset	*NY*	*COL*	*FLA*	*CAL*	*E-US*	*W-US*	*C-US*	*US*
$\|G'\|$ (MB)	9.22	9.44	23.97	44.1	84.4	142	358	590
$\|G^*\|$ (MB)	146	121	209	562	1,390	2,085	12,379	18,199

Exp-4: Space consumption of G' and G^*. In this experiment, we report the size of G' and G^* for each dataset and the results are shown in Table 2. As shown in Table 2, the size of G^* is much larger than that of G'. The results are consistent with our theoretical analysis in Lemma 4.5 and verify the necessity of proposing the techniques in Section 4.4.

Table 3: φ for each update

Dataset	*NY*	*COL*	*FLA*	*CAL*	*E-US*	*W-US*	*C-US*	*US*
φ	61.6	46.8	34.8	72.1	66.2	77.9	187.8	172.4

Exp-5: φ for each update. Table 3 shows the φ (average number of shortcuts with weight change) when the weight of an edge in the road network is update. To compute φ, we randomly choose 1,000 edges from each road network. For each chosen edge, we randomly update its weight as Exp-1 or Exp-2, compute G' for the updated road network and record the number of shortcuts with weight update. As shown in Table 3, when the weight of an edge

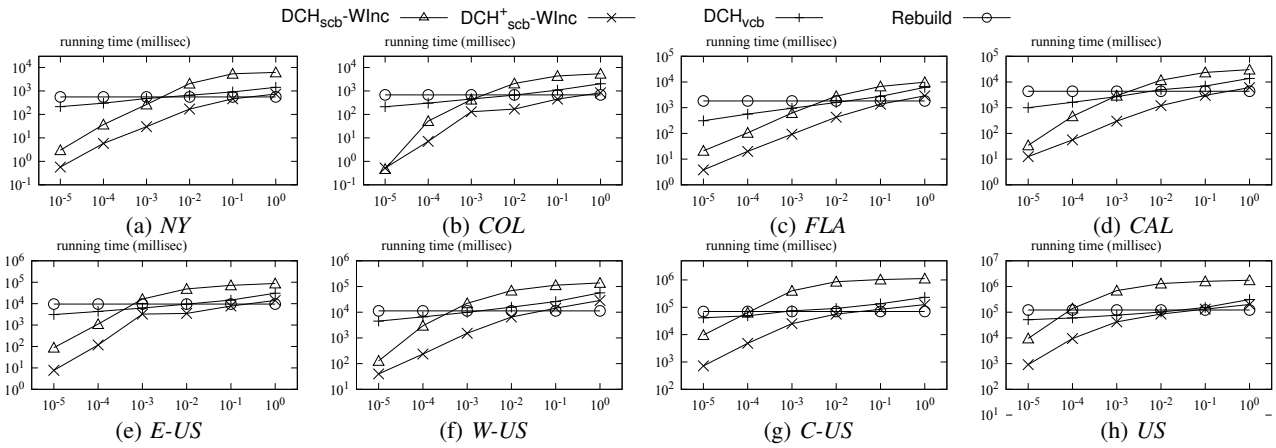

Figure 7: Processing Time for Batch Update - Weight Increase (Varying the percentage of edges with weight increase)

is updated, the number of shortcuts with weight change for each update is very small in practice.

7. RELATED WORK

Shortest path in static road networks. With the proliferation of graph applications, research efforts have been devoted to many fundamental problems in managing and analyzing graph data [41, 42, 43, 44, 45, 39, 30]. Among them, shortest path queries are one of the most fundamental problem in road networks. The classic approach is Dijkastra's algorithm [18] but it may result in a long running time. Therefore, index-based approaches are studied. ALT uses landmark to pre-compute some shortest distances for accelerating A* searching [21]. HiTi accelerates query processing by dividing the road network and constructing hierarchical structure [24]. HH [35] prunes search space to accelerate query. CRP [17] supports the personalized cost functions. CH [19] and AH [50] are two state-of-the-art hierarchy approaches for shortest path query on road networks. CH constructs an index structure named *shortcut index* by recursively contracting the vertex based on a total vertex order [19]. AH divides road networks into grids using 2-dimensional spatial properties of the road networks and follows a similar query processing procedure of CH. As evaluated in [29], AH can be a little faster than CH in query processing time but involves larger index size and longer pre-processing time. Related to the shortest path queries, computing the shortest distance between two given query vertices is also extensively studied [7, 8, 23, 9, 10, 12, 32]. [11] provides a comprehensive survey on shortest path/distance algorithms. [38] and [29] experimentally evaluate the practical performance of the representative algorithms.

Shortest path in dynamic road networks. Although many index-based methods for the shortest path queries are proposed, most of them assume the input road network is static. In [20], DCH$_{vcs}$ is proposed to maintain the index structure of CH for dynamic networks, which is introduced in Section 3 and used as the baseline solution in our experiment. Besides DCH$_{vcs}$, CRP [17] also supports the dynamic maintenance of its index structure and we also compare it with our algorithms in the experiment. [46] studies the dynamic shortest path problem in distributed environments. Since it considers the distributed environments, its objective and proposed techniques are totally different from ours. [22] studies the dynamic edge-constrained short path query problem, which considers the edges associated with multiple labels.

Shortest path in time-dependent road networks. In the literature, shortest path in time-dependent road networks is also studied. In a time-dependent road network $G = (V, E)$, each edge (u, v) is assigned with a function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ which specifies the time $f(t)$ needed to reach v from u via edge (u, v) when starting at time t [16]. As discussed in Section 1, time-dependent road network model is unable to handle sudden and unpredictable edge weight update, we adopt the dynamic graph model in this paper. Given a time-dependent road network G, for a source vertex s, a target vertex t, and a departure time t, the time dependent shortest path query returns the path from s to t with the fastest travel time in G when departing at time t. Given a source vertex s, a target vertex t, and a departure time interval I, travel time profile query returns the departing time $t \in I$, at which it takes the fastest travel time from s to t, and the corresponding shortest path.

[13] first proposes an index-based approach named TCH by generalizing CH to support the shortest path/travel time profile queries in time-dependent road networks. [14] further reduces the huge space requirement of TCH by adopting the approximated shortcuts technique. By selecting a set of landmark vertices and computing the travel-time summaries from landmarks towards all reachable vertices, [25] devises the FLAT oracle to speedup the shortest path queries in time-dependent road networks. Following the landmark-based approach, [27] creates a hierarchy of landmark sets to further reduce the query processing time. [26] experimentally evaluates the performance of FLAT and HORN. Recently, [37] proposes a height-balanced tree structured index, called TD-G-Tree, to answer the shortest path/travel time profile queries. [11] also surveys recent advances on the shortest path related problems on time-dependent road networks.

8. CONCLUSION

In this paper, we study the shortest path index maintenance problem on dynamic road networks. We adopt CH as our underlying shortest path computation method and aim to efficiently maintain the *shortcut index* of CH when the edge weights are updated. We propose a shortcut-centric paradigm focusing on exploring a small number of shortcuts to maintain the *shortcut index*. Following the paradigm, we design efficient algorithms to maintain the *shortcut index* for streaming update and batch update with non-trivial theoretical guarantees. The experimental results show the efficiency of our proposed algorithms.

Acknowledge. Long Yuan is supported by NSFC61902184 and NSF of Jiangsu Province BK20190453. Lu Qin is supported by ARC DP160101513. Lijun Chang is supported by ARC DP160101513 and FT180100256. Ying Zhang is supported by ARC DP180103096 and FT170100128. Xuemin Lin is supported by 2018YFB1003504, NSFC61232006, ARC DP180103096 and DP170101628

9. REFERENCES

[1] Crowdsourcing Transportation Systems Data, Michigan Department of Transportation. Available at https://www.michigan.gov/documents/mdot/02-14-2015_Crowd_Sourced_Mobile_Applications_483062_7.pdf.

[2] https://www.tomtom.com/automotive/products-services/connected-services/tomtom-traffic/.

[3] Road traffic injuries. https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries.

[4] Statistical Report on Internet Development in China, China Internet Network Information Center. Available at https://cnnic.com.cn/IDR/ReportDownloads/201807/P020180711391069195909.pdf.

[5] Taxi and Ridehailing Usage in New York City. https://toddwschneider.com/dashboards/nyc-taxi-ridehailing-uber-lyft-data/.

[6] TomTom Real Time Traffic Information, TomTom White Paper. Available at https://www.tomtom.com/lib/img/REAL_TIME_TRAFFIC_WHITEPAPER.pdf.

[7] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of SEA*, pages 230–241, 2011.

[8] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of ESA*, pages 24–35, 2012.

[9] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings of ALENEX*, pages 147–154, 2014.

[10] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In *Proceedings of SEA*, pages 55–66, 2013.

[11] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. 2016.

[12] H. Bast, S. Funke, and D. Matijevic. Ultrafast shortest-path queries via transit nodes. In *Proceedings of DIMACS, The Shortest Path Problem Workshop*, pages 175–192, 2006.

[13] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments*, pages 97–105, 2009.

[14] G. V. Batz, R. Geisberger, S. Neubauer, and P. Sanders. Time-dependent contraction hierarchies and approximation. In *Proceedings of SEA,*, pages 166–177, 2010.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[16] K. L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, 14(3):493 – 498, 1966.

[17] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[18] E. Djikstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[19] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of WEA*, pages 319–333, 2008.

[20] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[21] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of SODA*, pages 156–165, 2005.

[22] M. S. Hassan, W. G. Aref, and A. M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In *Proceedings of SIGMOD*, pages 1183–1197, 2016.

[23] R. Jin, N. Ruan, Y. Xiang, and V. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of SIGMOD*, pages 445–456, 2012.

[24] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE TKDE*, 14(5):1029–1046, 2002.

[25] S. C. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. D. Zaroliagis. Analysis and experimental evaluation of time-dependent distance oracles. In *Proceedings of ALENEX*, pages 147–158, 2015.

[26] S. C. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. D. Zaroliagis. Engineering oracles for time-dependent road networks. In *Proceedings of ALENEX*, pages 1–14, 2016.

[27] S. C. Kontogiannis, D. Wagner, and C. D. Zaroliagis. Hierarchical oracles for time-dependent networks. *CoRR*, abs/1502.05222, 2015.

[28] H. Li, Y. Ge, R. Hong, and H. Zhu. Point-of-interest recommendations: Learning potential check-ins from friends. In *Proceedings of SIGKDD*, pages 975–984, 2016.

[29] Y. Li, L. H. U, M. L. Yiu, and N. M. Kou. An experimental study on hub labeling based shortest path algorithms. *PVLDB*, 11(4):445–457, 2017.

[30] B. Liu, L. Yuan, X. Lin, L. Qin, W. Zhang, and J. Zhou. Efficient (α, β)-core computation: an index-based approach. In *Proceedings of WWW*, pages 1130–1141, 2019.

[31] Y. Liu, T. Pham, G. Cong, and Q. Yuan. An experimental evaluation of point-of-interest recommendation in location-based social networks. *PVLDB*, 10(10):1010–1021, 2017.

[32] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu. When hierarchy meets 2-hop-labeling: efficient shortest distance queries on road networks. In *Proceedings of SIGMOD*, pages 709–724, 2018.

[33] D. Ouyang, L. Yuan, F. Zhang, L. Qin, and X. Lin. Towards efficient path skyline computation in bicriteria networks. In *Proceedings of International Conference on Database Systems for Advanced Applications*, pages 239–254, 2018.

[34] I. Pohl. Bidirectional and heuristic search in path problems. Technical report, 1969.

[35] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of ESA*, pages 568–579, 2005.

[36] A. Sharma, V. Ahsani, and S. R. and. Evaluation of opportunities and challenges of using inrix data for real-time performance monitoring and historical trend assessment (2017). Reports and White Papers. 24. Available at https://lib.dr.iastate.edu/ccee_reports/24.

[37] Y. Wang, G. Li, and N. Tang. Querying shortest paths on time dependent road networks. *PVLDB*, 12(11):1249–1261, 2019.

[38] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.

[39] X. Wu, L. Yuan, X. Lin, S. Yang, and W. Zhang. Towards efficient k-tripeak decomposition on large graphs. In *Proceedings of International Conference on Database Systems for Advanced Applications*, pages 604–621, 2019.

[40] G. Xu and Y. Xu. *GPS: theory, algorithms and applications*. Springer, 2016.

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

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

[43] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Effective and efficient dynamic graph coloring. *PVLDB*, 11(3):338–351, 2017.

[44] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. I/O efficient ECC

graph decomposition via graph reduction. *VLDB Journal.*, 26(2):275–300, 2017.

[45] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang. Index-based densest clique percolation community search in networks. *IEEE TKDE*, 30(5):922–935, 2018.

[46] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, and H. T. Shen. Distributed shortest path query processing on dynamic road networks. *The VLDB Journal*, 26(3):399–419, 2017.

[47] J. Zhang, C. Kamga, H. Gong, and L. Gruenwald. U^2sod-db: a database system to manage large-scale ubiquitous urban sensing origin-destination data. In *Proceedings of SIGKDD Workshop on Urban Computing, 2012*, pages 163–171, 2012.

[48] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: Concepts, methodologies, and applications. *ACM TIST*, 5(3):38:1–38:55, 2014.

[49] Y. Zheng, Y. Liu, J. Yuan, and X. Xie. Urban computing with taxicabs. In *Proceedings of the 13th ACM International Conference on Ubiquitous Computing*, pages 89–98, 2011.

[50] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of SIGMOD*, pages 857–868, 2013.