Hierarchical and Dynamic *k*-Path Covers[*]

Takuya Akiba
Preferred Networks, Inc.
akiba@preferred.jp

Yosuke Yano
Recruit Holdings Co., Ltd.
yosukey@indeed.com

Naoto Mizuno
The University of Tokyo
mizuno@eps.s.u-tokyo.ac.jp

ABSTRACT

A metric-independent data structure for spatial networks called *k-all-path cover* (k-APC) has recently been proposed. It involves a set of vertices that covers all paths of size k, and is a general indexing technique that can accelerate various path-related processes on spatial networks, such as route planning and path subsampling to name a few. Although it is a promising tool, it currently has drawbacks pertaining to its construction and maintenance. First, k-APCs, especially for large values of k, are computationally too expensive. Second, an important factor related to quality is ignored by a prevalent construction algorithm. Third, an existing algorithm only focuses on static networks.

To address these issues, we propose novel k-APC construction and maintenance algorithms. Our algorithms recursively construct the layers of APCs, which we call the *k-all-path cover hierarchy*, by using vertex cover heuristics. This allows us to extract k-APCs for various values of k from the hierarchy. We also devise an algorithm to maintain k-APC hierarchies on dynamic networks. Our experiments showed that our construction algorithm can yield high solution quality, and has a short running time for large values of k. They also verified that our dynamic algorithm can handle an edge weight change within 40 ms.

Keywords

Graphs; road networks; spatial networks; indexing; route planning

1. INTRODUCTION

A new data structure called *k-all-path cover* (k-APC) has recently been developed for spatial networks [9] (the relevant article received the Best Paper Award at VLDB 2014). A k-APC is a subset of vertices that literally covers all paths of size k (see Section 3.1 for the formal definition). In graph theory, the problem of finding the minimum k-path vertex cover, an equivalent notion to k-APC, has been proposed and studied by Brešar et al. as a generalization of the vertex cover problem [5].

k-APC is attracting exceptional research interest because, while most previous index data structures are special-purpose, k-APC is general-purpose, i.e., it accelerates and facilitates many tasks on road networks. One application of APCs is route planning. It speeds up a simple Dijkstra search by conducting a search on *overlay graphs* of APCs (see Section 7). In particular, unlike other data structures, it is independent of a particular metric of a graph, and thus can also be used to accelerate personalized route planning on multi-cost networks [9,11]. Another example of its applications is subsampling paths. For example, when one visualizes a route on a road network, it is inefficient to retrieve too detailed a path. The property of k-APCs enables us to obtain a coarse path that contains at least one vertex of every k consecutive vertices along the original path (see the original paper [9] for a detailed description of applications).

Although the notion of k-APC seems to be natural and to have various potential applications, as stated above, algorithms for k-APC construction have not been intensively studied thus far. Funke et al. proposed a greedy construction algorithm [9], but it has a few issues:

- Constructing a k-APC for a large value of k (e.g., $k \geq 64$) is prohivitively expensive. Their algorithm needs to deal with all paths of size k, which can be exponential. Our experiments revealed that the running time of an algorithm increases drastically as k increases (in their experiments, Funke et al. set k to at most 40).
- Funke et al.'s algorithm tries to minimize only the number of vertices in covers. However, the number of edges in overlay graphs is also a dominant factor when using k-APCs in real applications (see Section 7).
- Funke et al. only focused on static networks. Although topological changes in road networks are relatively rare, edge weights (e.g., travel time) change from moment to moment due to the amounts of traffic and accidents. Handling such weight changes on networks should be necessary when one is also interested in the overlay graph of an APC.

Contribution

To address these problems in prevalent k-APC construction algorithms, we first propose a data structure called an *all-*

[*]Work done mostly while all authors were at National Institute of Informatics.

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA
© 2016 ACM. ISBN 978-1-4503-4073-1/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2983323.2983712

path cover hierarchy (APC hierarchy). Intuitively, the APC hierarchy of a graph consists of layers of recursively constructed k-APCs, and k-APCs of the original graph for various values of k can be easily extracted from an APC hierarchy. Therefore, our algorithm can be immediately employed for any application using k-APCs.

Our notion of APC hierarchies resolves the above issues and yields the following advantages:

- **Fast and scalable *k*-APC construction.** We propose a method to quickly construct k-APCs via our all-path cover hierarchies. We show that the algorithm can quickly construct k-APCs, especially for large values of k, which cannot be handled by prevalent algorithms.
- **Smaller overlay graphs.** We show by experiments that our algorithm constructs k-APCs of comparable quality in terms of the numbers of vertices, and that ours are better in terms of the sizes of overlay graphs. Moreover, one can control the trade-off between the size of a cover and that of the overlay graph by selecting a heuristic for our algorithm.
- **Fast dynamic maintenance.** We propose an algorithm to maintain all-path cover hierarchies under edge weight changes, which can quickly handle the weight change in country-level road networks. Our dynamic algorithm, which maintains the entire APC hierarchy, yields shorter update time than that required to maintain only an overlay graph.

The idea underlying our algorithm is to recursively construct a hierarchy of 2-APCs utilizing good heuristics for the vertex cover problem. It achieves efficient computation by avoiding the direct examinination of all paths of size k. The hierarchy also enables us to quickly handle dynamic updates by carefully propagating only the necessary edge weight changes to each layer.

We conducted experiments on real road networks of various sizes. The experimental results showed that *(i)* our construction algorithm is faster than the previous algorithm for moderate and large values of k, and generates higher quality of APCs, and *(ii)* our dynamic maintenance algorithm can handle an edge weight change within 40 ms, which is at most 50 times as fast as a naive algorithm.

Organization. The rest of this paper is organized as follows. We describe related work in Section 2, and formally introduce basic notions and briefly review an existing k-APC construction algorithm in Section 3. We describe our algorithm for constructing all-path cover hierarchy in Section 4 and discuss the dynamic maintenance of an all-path cover hierarchy in Section 5. We describe our experimental results in Section 6 and offer some conclusions in Section 8.

2. RELATED WORK

***k*-APC.** The notion of k-APC was introduced by Funke et al. as an index data structure for spatial networks [9]. They devised an algorithm for constructing k-APCs (see Section 3.3 for details) and proposed application scenarios for k-APCs. The problem of minimizing k-APCs has also been considered in [5] as a topic of theoretical interest. Theoretical properties including complexity, approximability and lower bounds of the size of a cover for various classes of graphs have been intensively studied in [5, 14]. Cheng et al. adopted a data structure equivalent to k-APCs for the k-hop reachability queries, which determine whether there is a path of length at most k between two given vertices [7].

Other Overlay Graphs. Although a k-APC itself is quite a new concept, it is rooted in a wealth of past research on shortest path computation and route planning. For the general (shortest-path) overlay graphs, Holzer et al. investigated the vertex selection method for multi-level overlay graphs [13]. A dynamic algorithm for general multi-level overlay graphs has also been proposed [6], but it is difficult to apply it to large networks because it involves dynamic shortest-path tree maintenance and, thus, requires quadratic space.

Route Planning on Single-cost Networks. Tao et al. proposed the notion of *k-skip cover* (or *k-shortest-path cover*), which is similar to k-APCs, and considered the problem of finding *k-skip shortest paths* instead of complete shortest paths. Many other data structures have been proposed for the shortest path problem on road networks, including the *contraction hierarchy* [12], *hub-based labeling* [2] and *pruned highway labeling* [3], to name a few. These indexing methods for shortest path queries have recently achieved satisfactory trade-offs between construction costs and query times for single-metric networks.

Route Planning on Multi-cost Networks. The problems of customizable route planning and personalized route planning, which consider not only travel time but also other criteria, have also been recently studied [8,10,11]. However, most past indexing methods for single-metric shortest path problems cannot be easily applied to these problems. One advantage of k-APCs is its independence of a fixed metric, which enables us to adopt k-APCs for the multi-cost shortest path problem. In particular, for personalized route planning on multi-cost road networks, Funke et al. attained a remarkable speed-up [9, 11] by applying k-APCs to personalized route planning. Thus methods based on k-APCs are state of the art.

3. PRELIMINARIES

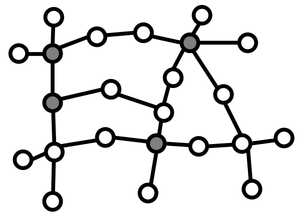
In this section, we first introduce the basic notions and notations used in this paper, and then describe the theoretical properties and the k-APC construction algorithm developed by Funke et al [9].

3.1 Definitions and Notation

Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. We always refer to a directed and simple graph when simply using the term "graph." In some parts of this paper, we handle an edge-weighted graph $G = (V, E, c)$, where $c : V \to \mathbb{R}_{\geq 0}$ is a weight function. Let $N^+_G(v)$ and $N^-_G(v)$ denote a set of out-neighbors and in-neighbors of vertex v in G, respectively. We also define $N_G(v)$ as $N^+_G(v) \cup N^-_G(v)$.

The definitions of k-APC and the overlay graph, which are the focuses of this study, are as follows:

Definition 1 (k-All-Path Cover)**:** *Let k be a positive integer. A k-all-path cover (k-APC) of a graph $G = (V, E)$ is a subset of vertices $C \subseteq V$ such that, for every simple path $\pi = (v_1, v_2, \ldots, v_k)$ of length $k - 1$, $C \cap \pi \neq \emptyset$ holds.*


(a) A graph and its 4-APC.

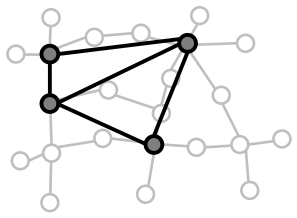

(b) Overlay graph induced by the 4-APC.

Figure 1: Example of a 4-APC and its overlay graph.

Table 1: Notation used in this paper. The table includes the terms defined in Section 4.

Symbol	Description
$G = (V, E)$	A directed graph
$G = (V, E, c)$	A directed and weighted graph
$N_G^+(v), N_G^-(v)$	The out-/in-neighbors of v in G
$N_G(v)$	$N_G^+(v) \cup N_G^-(v)$
$O(G, C)$	The overlay graph of G induced by C
$\mathcal{G} = (G_0, \dots, G_\ell)$	A sequence of graphs
$\mathcal{C} = (C_0, \dots, C_\ell)$	A sequence of k-all-path covers
$\mathcal{H} = (\mathcal{G}, \mathcal{C})$	A (k, ℓ)-all-path cover hierarchy

Definition 2 (Overlay Graph)**:** *The overlay graph $O(G, C) = (V_O, E_O)$ of graph $G = (V, E)$ induced by a subset of vertices $C \in V$ is the graph satisfying the following conditions: (1) The vertex set V_O is equal to C, and (2) the edge set E_O contains an edge (u, v) if and only if there exists a path from u to v on G that does not include a vertex in C other than the two end points.*

Example 1: *Figure 1 shows an example of an APC. The gray vertices in Figure 1a constitute a 4-APC, e.g., there exists no path of length three that does not contain a vertex in the APC. Figure 1b is the overlay graph induced by the APC.*

The definition of k-APCs for weighted graphs is the identical to that for unweighted graphs, except that we ignore edge weights. We define weighted overlay graph $O(G, C) = (V_O, E_O, c_O)$ for a weighted graph, so that $c_O(u, v)$ is the shortest distance among u-v paths satisfying Condition (2) in Definition 2.

Table 1 summarizes the notation in this paper, which also includes concepts to be introduced in Section 4.

Relation to shortest-path covers. When one is interested only in shortest paths on single-criterion networks, *k-skip covers*, or *k-shortest-path covers* (k-SPC), may be more suitable, which covers *all shortest paths of size k* instead of all paths [15]. The k-APC construction algorithm proposed by Funke et al. has also been proven to generate small k-SPCs than prior methods [9]. Similarly, our idea of APC hierarchies is also applicable to shortest-path covers with minor modification. However, we focus on k-APCs as they are more general purpose than k-SPCs, and are sufficiently scalable and small. Please also note that a k-APC is always a k-SPC.

3.2 Theoretical Properties

In [5, 9], the properties of k-APC were theoretically analyzed. Unfortunately, the problem of minimizing the size of k-APC has proven to be NP-hard for $k \geq 2$ by reduction from the vertex cover problem. Furthermore, the k-APC problem cannot be approximated for $k \geq 2$ in polynomial time within a factor of 1.3606 (unless P = NP), which is also inherited from the vertex cover problem [5]. The NP-hardness and inapproximability also apply to k-SPC, as pointed out in [9].

The upper bound on the size of an k-SPC has been discussed using the theory of VC dimensions. Please refer to [1, 9, 15] for details.

3.3 Reviewing the Algorithm Proposed by Funke et al.

Funke et al. introduced the notion of k-APCs and proposed a construction algorithm in their paper [9]. Here, we briefly review the algorithm.

Given a graph $G = (V, E)$, they first set cover C to the entire set of vertices V. Then they iterated through all vertices in a certain order and examined whether each vertex could be removed from C without violating the conditions of a k-APC. To check if the removal of a vertex v violated these conditions, they enumerated all paths containing v and no other vertices in C. They efficiently enumerated them by using a pruning technique, although the number of such paths can be exponential in their maximal lengths. This algorithm yielded a minimal k-APC due to its greedy nature, that is, no vertex could be removed from it.

For overlay graph construction, Dijkstra's algorithm was executed on each vertex in cover C. When vertex v in C was reached during the search from vertex r, they connected r and v in the overlay graph, stopped traversing edges from v, and moved on to the next vertex in the queue. The time complexity of the algorithm was $O(|V| (|V| \log |V| + |E|))$ in the worst case, since Dijkstra's algorithm was executed $|V|$ times. In most cases, however, the actual computation time was much shorter because the search space was limited by the cover for each execution of Dijkstra's algorithm.

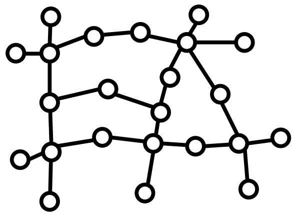
4. CONSTRUCTING *K*-PATH COVERS VIA HIERARCHIES

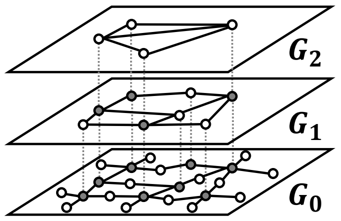
We describe our algorithm for constructing k-APCs in this section. It has been shown that the performance of Funke et al.'s algorithm [9] deteriorates as parameter k becomes large, since it enumerates paths of size less than k, the number of which can be exponential in k. Our algorithm aims to avoid such combinatorial explosions, and construct k-APCs quickly even when k is large, by using the novel idea of all-path cover hierarchies.

4.1 All-path Cover Hierarchy

We first introduce a few properties of APCs and define the notion of an *all-path cover hierarchy*, or an APC hierarchy. Our intuition is that an APC of an overlay graph, induced by another cover of the original graph, also forms a cover of the original graph. The following lemma states that this is the case.

Lemma 1: *Let $G = (V, E)$ be a graph and C be an a-APC of G. Let $G' = O(G, C)$ be the overlay graph of G and C' be a b-all-path cover of G'. Then C' forms an ab-APC of G.*


(a) Original graph G_0



(b) Layers of 2-APCs

Figure 2: Example of a $(2,2)$-APC hierarchy. Gray vertices in each layer form a 2-APC and correspond to the vertices in the upper layer.

Proof. We prove that any path on G of size ab contains at least one vertex in C'. Let $p = (v_1, v_2, \ldots, v_{ab})$ be any path on G of size ab. The path p contains at least b vertices in C, since each of b subpaths $p_i = (v_{ia+1}, v_{ia+2}, \ldots, v_{ia+a})$ $(0 \le i < b)$ of p (of size a) contains at least one vertex in C by the definition of a-APCs. Let $p' = (v_{c_1}, v_{c_2}, \ldots, v_{c_k})$ be an ordered set of all vertices in both C and p (of size k), where $c_i < c_j$ if $i < j$ holds. For any i $(1 \le i \le k-1)$, v_{c_i} and $v_{c_{i+1}}$ are adjacent in G' because there is a path between them in G such that it contains no other vertices in C. Thus, p' can be considered a path on G' of size k $(\ge b)$, and p' contains at least one vertex in C'. □

In Lemma 1, we have proved that we can obtain k-APC for large values of k by constructing APCs twice. Of course, this process can be performed as many times as we want. Based on this idea, we define a (b,ℓ)-APC hierarchy, which is a sequence of APCs and overlay graphs.

Definition 3 ((b,ℓ)-All-path Cover Hierarchy)**:** *Let b and ℓ be positive integers. A (b,ℓ)-All-path Cover Hierarchy $\mathcal{H} = (\mathcal{G}, \mathcal{C})$ of graph $G = (V,E)$ consists of a family of graphs $\mathcal{G} = (G_0, G_1, \ldots, G_\ell)$ and a family of sets of vertices $\mathcal{H} = (C_0, C_1, \ldots, C_\ell)$ satisfying the following conditions.*

1. *$G_0 = G$, $C_0 = V$,*
2. *C_i is a b-APC of G_{i-1} for $1 \le i \le \ell$.*
3. *$G_i = O(G_{i-1}, C_i)$ for $1 \le i \le \ell$,*

In short, there are conditions whereby C_i is always a b-APC of G_{i-1} and G_i is an overlay graph of G_{i-1} induced by C_i. An APC hierarchy can also be seen as the process of the recursive construction of an APC, where we repeatedly take a b-APC on one overlay graph and create another overlay graph of the next level.

The next theorem immediately follows Lemma 1:

Theorem 2: *Let $G = (V,E)$ be a graph and $\mathcal{H} = (\mathcal{G}, \mathcal{C})$ be a (b,ℓ)-APC hierarchy of G, where $\mathcal{G} = (G_0, G_1, \ldots, G_\ell)$ and $\mathcal{C} = (C_0, C_1, \ldots, C_\ell)$. Then C_i is a (b^i)-APC for $0 \le i \le \ell$. In particular, C_ℓ is a (b^ℓ)-APC of G.*

Example 2: *Figure 2 shows a (b,ℓ)-APC hierarchy, where $b = 2$ and $\ell = 2$. The layers of the graphs show G_0, G_1 and G_2 from bottom to top, and the gray colored vertices in each layer constitute a 2-APC of the graph in the layer. Note that the gray vertices form overlay graphs in the upper layer. The topmost layer is a $4(=2^2)$-APC of the original graph.*

Algorithm 1 The main part of our proposed algorithm.

```
Require: $G = (V, E)$, $k \in \mathbb{N}$
1: procedure ConstructKAPCHierarchy($G, k$)
2:     $G_0 \leftarrow G$, $C_0 \leftarrow V$
3:     $\ell \leftarrow \lfloor \log_2 k \rfloor$
4:     for $i \leftarrow 1, 2, \ldots, \ell$ do
5:         $C_i \leftarrow$ GetVertexCover($G_{i-1}$)
6:         $G_i \leftarrow$ GetOverlayGraph($G_{i-1}, C_i$)
7:     $\mathcal{H} \leftarrow ((G_0, G_1, \ldots, G_\ell), (C_0, C_1, \ldots, C_\ell))$
8:     return $\mathcal{H}$
```

4.2 Algorithm Overview

Thus far, we have seen the core idea underlying our algorithm, e.g., hierarchical construction of APCs. The algorithm is almost complete, but what values of k should we choose? We decided to set b to 2 in our algorithm for the following reasons:

- Algorithms and heuristics for finding a satisfactory 2-APC, which is equivalent to a vertex cover, have been extensively studied.
- We can maintain overlay graphs of 2-APCs by a simple algorithm, on dynamic networks with edge-weight modifications.
- It is easy to control the trade-off between the size of a cover and that of its overlay graph.

These three points will be discussed in detail in Section 4.3, Section 5, and Section 6.2, respectively.

We now introduce our algorithm for constructing k-APCs. Algorithm 1 shows the pseudocode of our proposed algorithm. It takes as input graph $G = (V,E)$ and the value of k. It returns a $(2,\ell)$-APC hierarchy $\mathcal{H} = (\mathcal{G}, \mathcal{C})$ such that C_ℓ is a 2^ℓ-APC, where ℓ is $\lfloor \log_2 k \rfloor$. (Note that a 2^ℓ-APC is also a k-all-path cover.) At the beginning of the algorithm, we set G_0 and C_0 to G and V, respectively. In the main loop, we iterate i from 1 to ℓ and construct a hierarchy layer by layer; C_i is set to a vertex cover of G_{i-1} and G_i is set to the overlay graph of G_{i-1} induced by C_i. Finally it returns a pair of sequences of overlay graphs and APCs as a hierarchy.

Time complexity. The time complexity of this algorithm, as well as the quality of the cover, largely depends on an algorithm that computes the vertex cover, GetVertexCover in Algorithm 1. We discuss vertex cover algorithms in Section 4.3. Let $T(n,m)$ be the time complexity of function GetVertexCover, where n and m are the numbers of vertices and edges, respectively. Then, the time complexity of the algorithm is

$$O\left(\sum_{i=0}^{\ell-1} \left(T\left(|V_i|, |E_i|\right) + |V_i||E_i|\right)\right).$$

Thus, the worst-case time complexity of the algorithm is $O((T(|V|, |V|^2) + |V|^3) \log k)$, where $T(n,m)$ is at most $O((n+m)\log n)$ when using vertex cover algorithms described in Section 4.3. However, our experiments in Section 6 suggest that the size of the overlay graph does not explode in practice, and that overlay graph construction occurs quite efficiently in most cases.

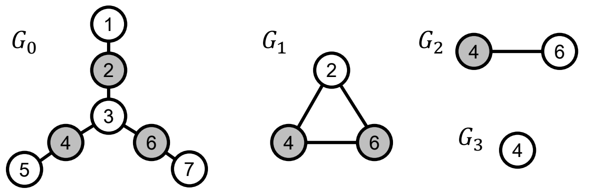


Figure 3: Example of a hierarchy that violates minimality. The graph G_3 is an 8-APC of G_0 calculated by our algorithm, but there is no path of size 8 in G_0.

Minimality. Unfortunately, this algorithm does not guarantee the minimality of the resulting APC unlike the greedy algorithm, even if a vertex cover of an overlay graph in each iteration is minimal. Figure 3 shows an example of $(2, 3)$-APC hierarchy that violates minimality. The four graphs correspond to G_0, G_1, G_2, and G_3, and the gray vertices in each graph represent its 2-APC (or vertex cover). In the figure, each APC is minimal, but the resulting 8-APC $C_3 = \{4\}$ is not because an empty set is already an 8-APC of the original graph G_0. However, even if APCs are not always minimal, as we confirmed in our experimental results, they are sufficiently small.

4.3 Vertex Cover Heuristics

As described in the previous section, we repeatedly compute vertex covers on overlay graphs in Algorithm 1. Although finding the minimum vertex cover is famously an NP-hard problem, several good heuristics for finding a small vertex cover have been proposed. We compare the following widely-accepted heuristics summarized in [4].

LR. Initialize C to an empty set. For each $v \in V$, add $N_G(v)$ to C if $v \notin C$.

LL. Initialize C to an empty set. For each $v \in V$, add v to C if there exists $u \in N_G(v)$ such that $u \notin C$.

ED. Initialize C to an empty set. For each $(u, v) \in E$, put u and v in C if neither u nor v is in C. This is a well-known 2-approximation algorithm for the vertex cover problem.

For these heuristics, the ordering of vertices or edges is arbitrary. Intuitively, high-degree vertices are likely to efficiently "cover" a graph. Thus, for LR(LL), we adopted increasing(decreasing) orders of degree, which we denote by LR-deg(LL-deg). We also adopted ordering by adaptive degree where we repeatedly chose the vertex with the highest degree and removed it from a graph. Heuristics with adaptive degree ordering are denoted by LR-ad or LL-ad. For ED, we sorted edges by larger value of degrees of two end points in decreasing order.

Let n and m be the numbers of vertices and edges. LR-deg and LL-deg works in $O(m + n \log n)$ time. LR-ad, LL-ad and ED works in $O((n+m) \log n)$ time. Section 6 shows how the choice of vertex cover heuristics affects time consumption and output quality.

5. DYNAMIC MAINTENANCE

In the networks that garner our focus in this study, namely road networks, edge insertions and deletions are not frequent. By constrast, the weights of edges are frequently updated as traffic changes from moment to moment. Therefore, we primarily discuss the dynamic maintenance of overlay graphs under edge-weight changes in this section.

We first design a general algorithm that assumes a k-APC and its overlay graph for a single value of k, and maintains them. We then propose an algorithm to maintain the entire APC hierarchy. Interestingly, this algorithm is faster than the general algorithm in practice, as it can effectively detect the outdated part of a graph by hierarchical propagation. We also briefly describe an algorithm that handles edge insertions and deletions in Section 5.3.

Algorithm 2 Naive edge-weight update algorithm for general k-APCs.

```
Require: $G = (V, E, c)$, $O(G, C) = (C, E_O, c_O)$, $e = (u, v) \in E$, $d \in \mathbb{R}_{\geq 0}$.
 1: procedure UpdateWeight($G, O(G, C), e, d$)
 2:     $c(e) \leftarrow d$
 3:     $B \leftarrow \emptyset$, $S \leftarrow \{u\}$
 4:     $Q \leftarrow$ an empty queue
 5:     Add $u$ to $Q$
 6:     while $Q$ is not empty do
 7:         $w \leftarrow$ Pop from $Q$
 8:         if $w \in C$ then
 9:             Add $w$ to $B$
10:         else
11:             for $z \in N^-_G(w)$ do
12:                 if $z \in S$ then continue
13:                 Add $z$ to $Q$ and $S$
14:     for $w \in B$ do
15:         $P \leftarrow$ a priority queue with an element $(0, w)$
16:         $H \leftarrow$ a map from $V$ to $\mathbb{R}_{\geq 0}$
17:         $H[w] \leftarrow 0$
18:         while $P$ is not empty do
19:             $(x, z) \leftarrow$ Pop from $P$
20:             if $x > H[z]$ then continue
21:             if $z \in C$ and $z \neq w$ then
22:                 $c_O(w, z) \leftarrow x$
23:             else
24:                 for $z' \in N^+_G(z)$ do
25:                     $x' \leftarrow x + d(z, z')$
26:                     Add $(x', z')$ to $P$
27:                     $H[z'] \leftarrow x'$
```

5.1 Algorithm for Non-hierarchical APC

We first provide a high-level overview of the general algorithm for edge-weight changes. When the weight of an edge is updated, the algorithm first searches for the *boundary* of the edge, which is a set of vertices in the APC. The formal definition of the boundary is as follows:

Definition 4 (Boundary)**:** *Let $G = (V, E)$ be a graph and $O(G, C) = (C, E_O)$ be its overlay graph induced by a k-all-path cover C.* The boundary *of an edge $e = (u, v) \in E$ is a set of vertices $B \subseteq C$ such that vertex $w \in C$ is in B if and only if there is at least one path from w to u that does not include a vertex in C other than the two endpoints.*

The boundary consists, in other words, of the vertices of an overlay graph that can potentially be affected by weight change. We reconstruct edges from the vertices in the boundary by Dijkstra's algorithm.

Algorithm 2 shows the pseudocode of the general update algorithm. The inputs of the algorithm are a graph $G = (V,$

$E, c)$, its overlay graph $O(G, C) = (V_O, E_O, c_O)$, a modified edge $e = (u, v)$, and its weight after change d. We first update $c(e)$ to d. In the first half of the algorithm (from line 3 to line 13), the algorithm computes the boundary vertices of e and stores them in B. In order to obtain the boundary, we conduct a breadth-first search (BFS) backwards from u. When we visit a vertex in cover C during the search, we add the vertex to B and stop traversing edges from the vertex.

In the latter half of the algorithm (from line 14 to line 27), we conduct a Dijkstra search from each vertex in B using a priority queue. Again, when we visit a vertex in cover C (excluding the root of the search), we stop traversing its neighbors and update the edge weight in the overlay graph from the root to the vertex.

Correctness. In order to prove the correctness of the algorithm, we show that it suffices to reconstruct the edges from the boundary vertices in an overlay graph.

Lemma 3 (Vertices affected by an update)**:** *Let $G = (V, E, c)$ be a graph and $O(G, C) = (C, E_O, c_O)$ be its overlay graph induced by a k-all-path cover C. Let $e = (u, v)$ be an edge to be modified, and B be the boundary of e. Let $G' = (V, E, c')$ be the graph after modification and $O(G') = (C, E_O, c'_O)$ the new overlay graph. Then, for any edge $e' = (u', v') \in E_O$, $u' \in B$ if $c_O(e') \neq c'_O(e')$.*

Proof. Note first that edge-weight modification does not change the structure of the overlay graph. An edge $(u', v') \in E_O$ in the overlay graph may need to change its weight only when at least one path from u' to v' in the original graph (satisfying Condition (2) in Definition 2) passes through e because the edge weights in the overlay graph are the shortest distances (under the condition) in the original graph. Assume that vertex $u' \in C$ is not in B; then, there is no path from u' to u that does not contain a vertex in the cover other than the two end points, from the definition of the boundary. This implies $c_O(u', v') = c'_O(u', v')$ for any vertex $v' \in C$ adjacent to u' because no path from u' to v' in G satisfying the condition contains the modified edge e. Therefore, if $c_O(u', v') \neq c'_O(u', v')$, u' should be in boundary B. □

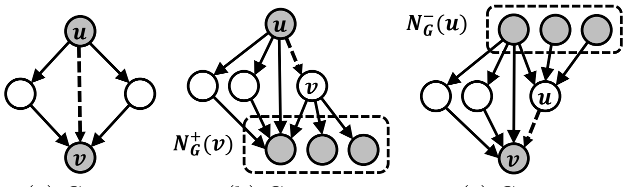
The correctness of the algorithm is immediate from Lemma 3.

Theorem 4 (Correctness of Algorithm 2)**:** *Algorithm 2 correctly maintains the overlay graph $O(G, C)$ for a weight change in the edge e in graph G.*

Time complexity. The time required for an update is dominated by the number of boundary vertices edges visited in searches because we conduct Dijkstra searches B times within the part of the graph separated by vertices in the cover. We can assume that these numbers are significantly smaller than $|V|$ and $|E|$ in most real road networks, but the worst-case time complexity is $O(|V|(|E| + |V| \log |V|))$.

5.2 Algorithm for APC Hierarchy

In this section, we consider maintaining the entire (b, ℓ)-all-path cover hierarchy instead of one overlay graph. Maintaining all overlay graphs in the hierarchy may appear inefficient at first glance, but we can in fact significantly reduce the number of edges to be checked, as we see in Section 6, by carefully propagating weight changes from a lower layer to an upper layer.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 4: Three types of edge-weight changes. Each figure shows the range of the graph that needs to be checked. The dashed arrows represent modified edges (u, v) and the vertices in APCs are in gray.

We first focus on two adjacent layers in the hierarchy, namely, $G_i = (C_i, E_i, c_i)$ and $G_{i+1} = (C_{i+1}, E_{i+1}, c_{i+1})$. Let us assume that the weight of edge $e = (u, v) \in E_i$ is modified. Of course we can update G_{i+1} by Algorithm 2, but we can do better when C_{i+1} is a 2-APC (recall that C_{i+1} forms a 2-APC of G_i). In this situation, an edge weight change can be categorized into the following three types. These three cases are also shown in Figure 4.

(Case 1) when $u \in C_{i+1}$ and $v \in C_{i+1}$. In this case, the boundary of $e = (u, v)$ is $\{u\}$, and the only vertex reachable from u through e is v. We only have to reconstruct the paths from u to v.

(Case 2) when $u \in C_{i+1}$ and $v \notin C_{i+1}$. In this case, the boundary of e is again $\{u\}$. The set of vertices reachable from u through e is $N^+_{G_i}(v)$ because all vertices in $N^+_{G_i}(v)$ belong to C_{i+1}. Thus we need to check the paths between $\{u\}$ and $N^+_{G_i}(v)$.

(Case 3) when $u \notin C_{i+1}$ and $v \in C_{i+1}$. This is symmetrical to the previous case. It suffices to check the paths between $N^-_{G_i}(u)$ and $\{v\}$.

To update all layers, we repeat the procedure above in a recursive manner. That is, when an edge in E_0 is modified, we first update G_1 by the procedure above. When we find an edge in E_1 that has changed weight due to the modification, we update G_2 using the same procedure, and so on.

Algorithm 3 shows the pseudocode of our proposed algorithm for APC hierarchy maintenance. UpdateHierarchical are given an APC hierarchy $\mathcal{H} = (\mathcal{G}, \mathcal{C})$, an integer j indicating the layer to be modified, an edge $e \in G_j$ to be modified, and edge weight d following the modification. We set j to 0 when we modify the original graph. It first checks whether the modification is necessary, i.e., whether $c_j(e) \neq d$. If so, it modifies the value of $c_j(e)$ to d. The recursion is then terminated if it is in the topmost layer. The following three "if" blocks (Line 5 to 9, Line 10 to 15, and Line 16 to 21) correspond to the three cases described above. In each case, the algorithm recurses for each updated edge. It should be noted that the algorithm checks whether the modification is necessary in each callee. Moreover, if the modification is unnecessary, the propagation is also stopped at that step, i.e., we do not recurse, which is important for efficiency.

Correctness. The correctness of Algorithm 3 is also straightforward. Let us think of the *reversed* boundary, by changing Definition 4 from "one path from w to u" to "one path from v to w." The reversed version of Lemma 3 then also holds;

Algorithm 3 Edge-weight update algorithm for APC hierarchies.

```
Require: $\mathcal{H} = (\mathcal{G}, \mathcal{C})$, $\mathcal{G} = (G_0, \ldots, G_\ell)$, $G_i = (V_i, E_i, c_i)$
    $\mathcal{C} = (C_0, \ldots, C_\ell)$, $0 \le j \le \ell$, $e = (u, v) \in E_j$, $d \in \mathbb{R}_{\ge 0}$.
 1: procedure UpdateHierarchical($\mathcal{H}, j, e, d$)
 2:     if $c_j(e) = d$ then return
 3:     $c_j(e) \leftarrow d$
 4:     if $j = \ell$ then return
 5:     if $u \in C_{j+1}$ and $v \in C_{j+1}$ then
 6:         $d_{\min} \leftarrow \infty$
 7:         for $z \in (N^+_{G_j}(u) \cap N^-_{G_j}(v)) \setminus C_{j+1}$ do
 8:             $d_{\min} \leftarrow \min\{d_{\min}, c_j(u, z) + c_j(z, v)\}$
 9:         UpdateHierarchical($\mathcal{H}, j + 1, e, d_{\min}$)
10:     else if $u \in C_{j+1}$ then
11:         for $w \in N^+_{G_j}(v)$ do
12:             $d_{\min} \leftarrow \infty$
13:             for $z \in (N^+_{G_j}(u) \cap N^-_{G_j}(w)) \setminus C_{j+1}$ do
14:                 $d_{\min} \leftarrow \min\{d_{\min}, (c_j(u, z) + c_j(z, w)\}$
15:             UpdateHierarchical($\mathcal{H}, j + 1, (u, w), d_{\min}$)
16:     else if $v \in C_{j+1}$ then
17:         for $w \in N^-_{G_j}(u)$ do
18:             $d_{\min} \leftarrow \infty$
19:             for $z \in (N^+_{G_j}(w) \cap N^-_{G_j}(v)) \setminus C_{j+1}$ do
20:                 $d_{\min} \leftarrow \min\{d_{\min}, (c_j(w, z) + c_j(z, v)\}$
21:             UpdateHierarchical($\mathcal{H}, j + 1, (w, z), d_{\min}$)
```

that is, it suffices to reconstruct the edges to the reversed boundary in an overlay graph. Algorithm 3 checks all paths between the boundary and reversed boundary for each of the three cases above, noting that the length of those paths is at most 2. A proof sketch goes as follows:

Theorem 5 (Correctness of Algorithm 3)**:** *Algorithm 3 correctly maintains the APC hierarchy $\mathcal{H}$ for a weight change in edge e in graph G_0.*

Proof. We prove the theorem by induction. When j is 0, Algorithm 3 simply updates the weight of edge e and G_0 is correctly maintained. Assuming that G_k is correctly maintained, Algorithm 3 should have been recursively called for each updated edge in G_k. For each updated edge in G_k, Algorithm 3 checks all edges in G_{k+1} that can be affected by the update in G_k and reconstructs them, as in the discussion above. Thus, G_{k+1} is also correctly maintained. □

Further optimization. Algorithm 3 (and also Algorithm 2, actually) can be more simple when we reduce edge weight (as in Algorithm 4). That is, we only need to check if paths containing the shortened edge will form a new shortest path. This asymmetry occurs because we can immediately see that the shortest path remains the shortest path when the weight of an edge in the path is reduced, whereas an old shortest path can cease to be one when the weight of an edge in it is increased. If a path using the shortened edge is shorter than a given shortest path, it is the new shortest path.

Extention to multi-cost networks. We note that the idea of propagating modifications to upper layers is applicable even when a target network has multiple metrics. In this situation, overlay graphs in a hierarchy will retain Pareto optimal paths in lower layers as multi-edges. Finding Pareto optimal paths requires extensive computation in general, but becomes simple when the length of paths is restricted to 2, as in APC hierarchies.

Algorithm 4 APC hierarchy update algorithm optimized for decreasing edge weight.

```
Require: $\mathcal{H} = (\mathcal{G}, \mathcal{C})$, $\mathcal{G} = (G_0, \ldots, G_\ell)$, $G_i = (V_i, E_i, c_i)$
    $\mathcal{C} = (C_0, \ldots, C_\ell)$, $0 \le j \le \ell$, $e = u, v \in E_j$, $d \in \mathbb{R}_{\ge 0}$.
 1: procedure UpdateHierarchicalDec($\mathcal{H}, j, e, d$)
 2:     if $c_j(e) \ge d$ then return
 3:     $c_j(e) \leftarrow d$
 4:     if $j = \ell$ then return
 5:     if $u \in C_{j+1}$ and $v \in C_{j+1}$ then
 6:         UpdateHierarchicalDec($\mathcal{H}, j + 1, e, d$)
 7:     else if $u \in C_{j+1}$ then
 8:         for $w \in N^+_{G_j}(v)$ do
 9:             $d' \leftarrow c_j(u, v) + c_j(v, w)$
10:             UpdateHierarchicalDec($\mathcal{H}, j + 1, (u, w), d'$)
11:     else if $v \in C_{j+1}$ then
12:         for $w \in N^-_{G_j}(u)$ do
13:             $d' \leftarrow c_j(w, u) + c_j(u, v)$
14:             UpdateHierarchicalDec($\mathcal{H}, j + 1, (w, v), d'$)
```

5.3 Edge Insertion and Deletion

We briefly explain algorithms to maintain an overlay graph against edge insertion and deletion.

Edge Insertion. When at least one end of an inserted edge is in the cover, one can easily maintain the overlay graph by reconstructing edges around the end vertex, as when decreasing the edge weight. When neither end is in the cover, we first add one to the cover as well as the overlay graph. We then update the overlay graph by reconstructing edges from the boundary of the edge.

Edge Deletion. One can virtually delete an edge by updating its weight to a sufficiently large value.

Note that these operations only add vertices to the cover and never remove any, which gradually deteriorates the quality of an APC. In this way, however, these operations are essentially the same with edge-weight update, and we can maintain the structure of the overlay graph as much as possible. We may also devise a complex algorithm that properly handles topological changes in a graph, but we think that this simple algorithm should be a decent option when the topology of a target network rarely changes. This issue can be resolved by periodically reconstructing the entire APC.

6. EXPERIMENTS

6.1 Experimental Setup

We conducted all experiments on a Linux server with two Intel Xeon X5650 processors (2.67 GHz) and 96GB of memory. We implemented our proposed algorithms and the algorithm by Funke et al. [9] in C++ (gcc 4.8.4). For the vertex ordering of Funke et al.'s algorithm, we used `comp-inc`, or the post-order of a DFS search, which was suggeted as the best heuristic in their experiments.

Table 2 summarizes the datasets we used in our experiments. We used road networks from the 9th DIMACS Im-

Table 2: Road networks used in our experiments. d_{ave} denotes the average degree of a network.

Name		$\|V\|$	$\|E\|$	d_{ave}
DIMACS	CAL	1,890,815	4,657,742	2.46
	Europe	18,010,173	42,188,664	2.34
	USA	23,947,347	58,333,344	2.44
OSM	Paris	1,200,020	2,618,613	2.18
	Australia	9,823,244	20,599,045	2.10
	Japan	42,488,715	90,089,140	2.12

plementation Challenge datasets[1] and those extracted from OpenStreetMap[2] (OSM). The network size varied from a few million to 90 million. The edge weight of the DIMACS datasets is travel time and that of the OSM datasets is physical distance.

6.2 Construction

6.2.1 *Construction Time*

Figures 5a and 5b show the construction time of k-APCs for varying values of k in USA and Japan. Note that these figures are log-log plots. For $k \leq 8$, Funke+ was the fastest algorithm among these methods. However, its construction time suddenly increased for $k = 32$, and it did not complete the task for $k = 64$ within an hour. This is because the running time of Funke+ was dominated by the number of paths of length k, which can be exponential.

For both networks, our proposed method successfully constructed 256-APCs with all vertex cover heuristics within 200 s. Of the vertex cover heuristics, LR-deg seemed to be the fastest one, but other heuristics also ran in reasonable time. Our method only runs a vertex cover algorithm $O(\log k)$ times, which is feasible even for very large k.

6.2.2 *Quality of Cover*

We examine the quality of the APCs generated by these methods. We mainly compared the size of the resulting APCs and the overlay graphs induced by the APCs, which has not been considered in past work.

Table 3 summarizes the quality of the resulting k-APCs for $k = 16$. The upper part of the table shows the numbers of vertices in k-APCs and the lower part shows the number of edges in their corresponding overlay graphs. Regarding the size of k-APC, Funke+ yielded the best performance on the DIMACS datasets, and LR-deg, LR-deg, and LL-ad generated approximately 1.3 times larger covers. By contrast, these heuristics outperformed Funke+ for OSM datasets, and worked particularly well for the Japan dataset. The method by Funke can potentially construct better APCs than our method because hierarchical APCs are only a special case of general APCs. Practically, however, our construction algorithms were shown to generate APCs of comparable sizes to those of Funke+. LL-deg and ED failed to generate small k-APCs. Although ED is a theoretically bounded heuristic for the vertex cover problem, it did not lead to a satisfactory performance for k-APC construction.

[1]http://www.dis.uniroma1.it/challenge9/
[2]http://www.openstreetmap.org/

For the numbers of edges of overlay graphs, LR-deg always generated the smallest overlay graphs, which were approximately 15% to 40% smaller than those of Funke+. LL-ad showed slightly worse results than LR-deg. For the DIMACS datasets, LL-deg constructed the second smallest overlay graphs in spite of its large APCs. These results imply that a small APC does not necessarily result in a small overlay graph. Constructing a k-APC that induces a small overlay graph should also be beneficial when users want to maintain an overlay graph or run their algorithm on an overlay graph.

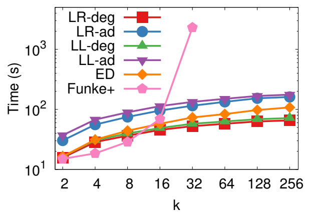
Figures 5c to 5h show the change in the size of covers, the size of overlay graphs, and the maximum degree of overlay graphs over the value of k in the USA and Japan networks. LR-deg, LR-ad, and LL-ad showed similar performance in terms of size of covers and constantly constructed small k-APCs. In Figures 5e and 5f, however, the size of the overlay graph increases with k for LR-ad. LR-deg and LL-ad constructed comparatively small overlay graphs compared to LR-ad and Funke+, but the sizes of the overlay graphs seemed to be saturated when k became large even for these methods. Figures 5g and 5h may partially explain the reason for this behavior. We can observe that the maximum degree of overlay graphs increases quite quickly for LR-ad, LR-deg, and LL-ad. Overlay graphs created by those methods seemed to contain some dense parts, which might have prevented the construction of a small vertex cover for the next layer.

6.3 Weight Update

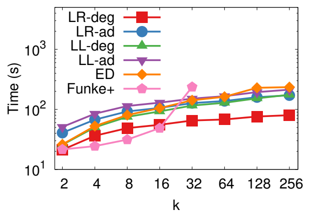
We also measured the performance of two edge-weight update algorithms, namely, the general algorithm (Algorithm 2) and Hierarchical Propagation (Algorithm 3). We employed the optimization technique discussed in Section 5.2 to both algorithms. In this experiment, we first constructed a hierarchical k-APC by LL-ad. We then randomly chose 10,000 edges as queries and updated the weights of the edges to half the original weight one by one (the decrease step). Finally, we updated the weights of the edges to the original weights (the increase step) and calculated the average time required for one update query for each step. Note that the general algorithm only updates the original graph and its overlay graph, but Hierarchical Propagation updates the entire hierarchy, that is, the layers of overlay graphs. We conducted the experiment for $k = 16$ and 256.

Table 4 summarizes the average time of weight updates for the two algorithms. HP stands for Hierarchical Propagation in the table. The left side shows the result for $k = 16$ and the right for $k = 256$. For each side, the increase and decrease columns show the average time required for the increase and decrease steps. The values in the ratio columns show the speed-up ratio of HP to the general algorithm. HP outperformed the general algorithm for all datasets except CAL. In particular, HP performed better in the increase step and for large values of k, and was at most 58 times faster than the general algorithm. HP required at most 40 ms for an edge weight update on average.

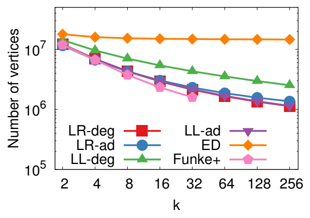
We expected that the decrease step was always easier than the increase step because edges that should be examined are fewer in the decrease step as we described in Section 5.2. However, this was sometimes not the case for HP. This might have been because HP prunes unnecessary scans of edges in the lower layers and propagates only necessary changes to



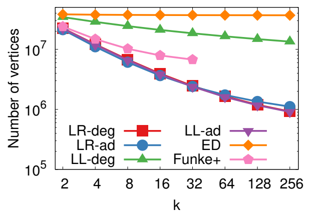
(a) Construction time (USA).



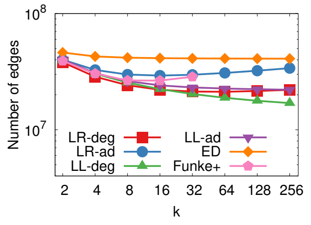
(b) Construction time (Japan).



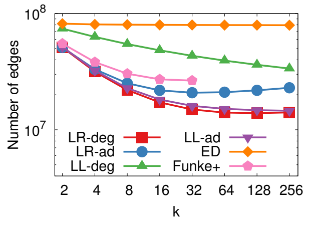
(c) Number of vertices (USA).



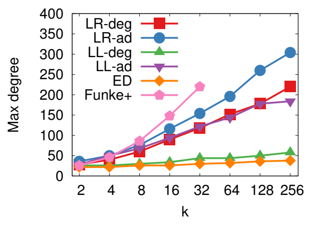
(d) Number of vertices (Japan).



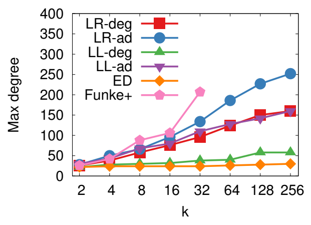
(e) Number of edges (USA).



(f) Number of edges (Japan).



(g) Max degree (USA).



(h) Max degree (Japan).

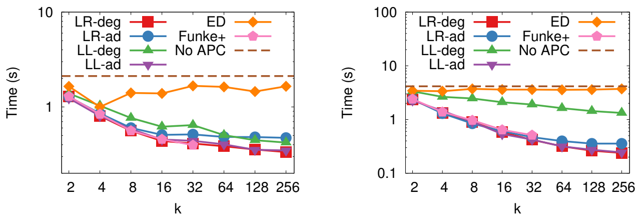
Figure 5: The performance of each method for varying values of k in USA and Japan: namely, construction time, the numbers of vertices in k-APCs, the numbers of edges in overlay graphs, and the maximum degree of overlay graphs. Some trials of Funke+ did not finish within an hour, and those points are not shown in the figures.

Table 3: Size of the resulting cover ($|C|$) and overlay graph ($|E_O|$) for each algorithm and dataset for $k = 16$.

	Dataset	Funke+	Hierarchical APC				
			LR-deg	LR-ad	LL-deg	LL-ad	ED
$\lvert C\rvert$	CAL	**192,409**	241,780	251,950	474,504	241,452	1,181,114
	Europe	**1,795,156**	2,372,150	2,441,853	4,886,137	2,372,924	10,676,518
	USA	**2,327,625**	2,921,614	3,033,414	5,398,901	2,938,373	14,994,034
	Paris	137,118	115,727	118,231	364,938	**113,900**	976,300
	Australia	1,347,137	858,287	**812,735**	4,578,880	841,935	8,451,682
	Japan	7,918,018	3,946,317	**3,656,519**	21,084,511	3,838,856	37,263,268
$\lvert E_O\rvert$	CAL	2,259,362	**1,903,322**	2,565,992	1,941,936	2,066,170	3,338,376
	Europe	19,147,972	**15,908,350**	21,010,563	17,531,103	17,110,582	28,595,950
	USA	26,490,560	**22,076,374**	29,268,514	22,374,876	24,098,004	41,315,910
	Paris	666,060	**555,376**	762,059	967,293	614,448	2,179,996
	Australia	4,521,446	**3,245,395**	4,097,776	10,219,513	3,470,355	17,909,600
	Japan	27,162,994	**17,206,565**	21,824,780	48,280,886	18,206,728	79,992,941

Table 4: Average update time for an edge-weight update query (μs). The times for increase and decrease steps are separately shown in the table. The ratio columns show the speed-up ratio of HP over the general one.

	$k = 16$						$k = 256$					
Dataset	Increase			Decrease			Increase			Decrease		
	General	HP	Ratio	General	HP	Ratio	General	HP	Ratio	General	HP	Ratio
CAL	67.1	**8.0**	8.37	**14.2**	17.5	0.81	820.2	**25.4**	32.28	101.3	**12.8**	7.89
Europe	53.0	**5.9**	9.06	12.0	**6.6**	1.82	737.6	**21.8**	33.77	106.0	**16.6**	6.39
USA	67.9	**7.5**	9.04	14.3	**7.7**	1.85	981.6	**39.5**	24.84	150.7	**36.5**	4.13
Paris	46.0	**3.8**	12.05	11.9	**4.4**	2.72	992.7	**17.4**	57.12	125.5	**8.3**	15.04
Australia	35.0	**3.6**	9.67	11.8	**4.1**	2.89	709.1	**12.1**	58.42	132.1	**10.0**	13.14
Japan	39.8	**5.5**	7.22	11.0	**6.6**	1.68	869.1	**20.4**	42.62	157.5	**17.9**	8.80



(a) USA (b) Japan

Figure 6: Average query time of route-planning queries accelerated by k-APCs and overlay graphs.

the upper layers, which possibly resulted in similar performance in both steps.

7. APPLICATION TO ROUTE PLANNING

In this section, we briefly show the usefulness of our k-APCs by applying them to route planning. In particular, we confirm the following two claims: *(1)* the number of edges (rather than vertices) in overlay graphs are an important performance factor in applications, and *(2)* k-APCs and their overlay graphs for large values of k (e.g., $k = 256$) are more useful than those for smaller values of k in certain cases. Note that the overlay graphs produced by the proposed methods have smaller numbers of edges than those by the previous method, and the proposed methods can construct k-APCs for large k, where the previous method times out.

Here, route planning is formalized as the standard point-to-point shortest-path problem under a single criterion, and we accelerate bi-directional Dijkstra searches by k-APCs and the overlay graphs. We report the average query time over 1,000 random queries.

Figure 6 shows the query time for varying values of k and the k-APC construction algorithms. It also includes results without APCs, from which we could confirm that APCs certainly accelerated query time. We can confirm the first claim by observing the query times for $k = 32$. For example, while the number of vertices in k-APCs constructed by the method in Funke+ was smaller than those of LR-deg, the query times of LR-deg were comparable to those of Funke+ because the overlay graphs of LR-deg had smaller numbers of edges than those of Funke+. The second claim can also be confirmed by comparing the query times for $k = 32$ and $k = 256$. The query time generally decreases when k increases. Funke+ did not have results for $k \geq 64$ as they failed to construct k-APCs within the time limit.

Please note that we do not claim that our bi-directional Dijkstra searches are competitive with state-of-the-art point-to-point shortest-path methods. On the contrary, unlike these methods, k-APCs are a general indexing scheme, and this problem has been studied as only of several problems to which k-APCs can be applied.

8. CONCLUSION

In this paper, we first introduced the notion of all-path cover hierarchies and proposed an algorithm for constructing k-all-path covers quickly via all-path cover hierarchies. Further, we developed the first practical algorithms for dynamicly maintaining overlay graphs and all-path cover hierarchies under edge-weight changes. Our experiments showed that for moderate values of k, our algorithm constructed comparable quality of APCs and smaller overlay graphs than the previous algorithm. For large values of k (e.g., $k \geq 64$), where the previous algorithm failed to construct APCs in one hour, our algorithm successfully constructed APCs within a few hundred seconds for country-level road networks. Moreover, the dynamic maintenance of the entire all-path cover hierarchy was at most 50 times faster than simply maintaining a k-APC in the experiments.

It is worth mentioning that the proposed method led the team including the authors to the victory at the 2016 SIGMOD Programming Contest, where the task was to process shortest-path queries on highly dynamic graphs.

Acknowledgement. This work was supported by JSPS Grant-in-Aid for Research Activity Startup (No. 15H06828) and JST, PRESTO.

9. REFERENCES

[1] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. Vc-dimension and shortest path algorithms. In *ICALP*, pages 690–699, 2011.

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

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

[4] E. Angel, R. Campigotto, and C. Laforest. Implementation and comparison of heuristics for the vertex cover problem on huge graphs. In *Experimental Algorithms*, pages 39–50. 2012.

[5] B. Brešar, F. Kardoš, K. Katrenič, and G. Semanišin. Minimum k-path vertex cover. *Discrete Applied Mathematics*, 159(12):1189–1195, 2011.

[6] F. Bruera, S. Cicerone, G. D'Angelo, G. D. Stefano, and D. Frigioni. Dynamic multi-level overlay graphs for shortest paths. *Mathematics in Computer Science*, 1(4):709–736, 2008.

[7] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.

[8] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *SEA*, pages 376–387, 2011.

[9] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. *PVLDB*, 7(10):893–902, 2014. Best Paper Award.

[10] S. Funke and S. Storandt. Polynomial-time construction of contraction hierarchies for multi-criteria objectives. In *ALENEX*, pages 41–54, 2013.

[11] S. Funke and S. Storandt. Personalized route planning in road networks. In *SIGSPATIAL*, 2015.

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

[13] M. Holzer, F. Schulz, and D. Wagner. Engineering multilevel overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, 2008.

[14] F. Kardoš, J. Katrenič, and I. Schiermeyer. On computing the minimum 3-path vertex cover and dissociation number of graphs. *Theoretical Computer Science*, 412(50):7009–7017, 2011.

[15] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *SIGMOD*, pages 421–432, 2011.