Towards Online Shortest Paths Computation

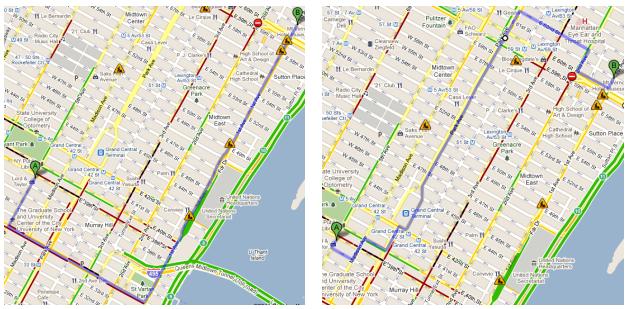
Leong Hou U, Hong Jun Zhao, Man Lung Yiu, Yuhong Li, and Zhiguo Gong

Abstract—The online shortest path problem aims at computing the shortest path based on live traffic circumstances. This is very important in modern car navigation systems as it helps drivers to make sensible decisions. To our best knowledge, there is no efficient system/solution that can offer affordable costs at both client and server sides for online shortest path computation. Unfortunately, the conventional client-server architecture scales poorly with the number of clients. A promising approach is to let the server collect live traffic information and then broadcast them over radio or wireless network. This approach has excellent scalability with the number of clients. Thus, we develop a new framework called *live traffic index* (LTI) which enables drivers to quickly and effectively collect the live traffic information on the broadcasting channel. An impressive result is that the driver can compute/update their shortest path result by receiving only a small fraction of the index. Our experimental study shows that LTI is robust to various parameters and it offers relatively short tune-in cost (at client side), fast query response time (at client side), small broadcast size (at server side), and light maintenance time (at server side) for online shortest path problem.

Index Terms—Spatial databases; Vehicle driving; Broadcasting

1 Introduction

Shortest path computation is an important function in modern car navigation systems and has been extensively studied in [1][2][3][4][5][6][7][8]. This function helps a driver to figure out the best route from his current position to destination. Typically, the shortest path is computed by offline data pre-stored in the navigation systems and the weight (travel time) of the road edges is estimated by the road distance or historical data. Unfortunately, road traffic circumstances change over time. Without live traffic circumstances, the route returned by the navigation system is no longer guaranteed an accurate result. We demonstrate this by an example in Fig. 1. Suppose that we are driving from `Lord & Taylor` (label A) to `Mt Vernon Hotel Museum` (label B) in `Manhattan,NY`. Those old navigation systems would suggest a route based on the pre-stored distance information as shown in Fig. 1(a). Note that this route passes through four road maintenance operations (indicated by maintenance icons) and one traffic congested road (indicated by a red line). In fact, if we take traffic circumstances into account, then we prefer the route in Fig. 1(b) rather than the route in Fig. 1(a).



(a) Shortest route using static weights (b) Shortest route using live traffic

Fig. 1. Two alternative shortest paths in *Manhattan, NY*

Nowadays, several online services provide live traffic data (by analyzing collected data from road sensors, traffic cameras, and crowdsourcing techniques), such as GoogleMap [9], Navteq [10], INRIX Traffic Information Provider [11], and TomTom NV [12], etc. These systems can calculate the *snapshot* shortest path queries based on current live traffic data; however, they do not report routes to drivers continuously due to high operating costs. Answering the shortest paths on the live traffic data can be viewed as a continuous monitoring problem in spatial databases, which is termed *online shortest paths computation* (OSP) in this work. To the best of our knowledge, this problem has not received much attention and the costs of answering such continuous queries vary hugely in different system architectures.

Typical client-server architecture can be used to answer shortest path queries on live traffic data. In this case, the navigation system typically sends the shortest path query to the service provider and waits the result back from the provider (called *result transmission model*). However, given the rapid growth of mobile devices and services, this model is facing scalability limitations in terms of network bandwidth and server loading. According to the Cisco Visual Networking Index forecast [13], global mobile traffic in 2010 was 237 petabytes per month and it grew by 2.6-fold in 2010, nearly tripling for the third year in a row. Based on a telecommunication expert [14], the world's cellular networks need to provide 100 times the capacity in 2015 when compared to the networks in 2011. Furthermore, live

- *L.H. U, H.J. Zhao, Y.H. Li, and Z.G. Gong are with the Department of Computer and Information Science, University of Macau, Macau. E-mails:* {ryanlhu,ma86569,yb27407,fstzgg}@umac.mo
- *M.L. Yiu is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong. E-mail:* csmlyiu@comp.polyu.edu.hk

traffic are updated frequently as these data may be collected by using crowdsourcing techniques (e.g., anonymous traffic data from Google map users on certain mobile devices). As such, huge communication cost will be spent on sending result paths on the this model. Obviously, the client-server architecture will soon become impractical in dealing with massive live traffic in near future. Ku et al. [15] raise the same concern in their work which processes spatial queries in wireless broadcast environments based on Euclidean distance metric.

Malviya et al. [16] developed a client-server system for continuous monitoring of registered shortest path queries. For each registered query $(\mathbf{s}, \mathbf{t})$, the server first precomputes K different candidate paths from $\mathbf{s}$ to $\mathbf{t}$. Then, the server periodically updates the travel times on these K paths based on the latest traffic, and reports the current best path to the corresponding user. Since this system adopts the client-server architecture, it cannot scale well with a large number of users, as discussed above. In addition, the reported paths are approximate results and the system does not provide any accuracy guarantee.

An alternative solution is to broadcast live traffic data over wireless network (e.g., 3G, LTE, Mobile WiMAX, etc.). The navigation system receives the live traffic data from the broadcast channel and executes the computation locally (called *raw transmission model*). The traffic data are broadcasted by a sequence of packets for each broadcast cycle. To answer shortest path queries based on live traffic circumstances, the navigation system must fetch those updated packets for each broadcast cycle. However, as we will analyze an example in Section 2.2, the probability of a packet being affected by 1% edge updates is 98.77%. This means that clients must fetch almost all broadcast packets in a broadcast cycle.

The main challenge on answering *live* shortest paths is *scalability*, in terms of the number of clients and the amount of live traffic updates. A new and promising solution to the shortest path computation is to broadcast an *air index* over the wireless network (called *index transmission model*) [17][18]. The main advantages of this model are that the network overhead is independent of the number of clients and every client only downloads a portion of the entire road map according to the index information. For instance, the proposed index in [17] constitutes a set of pairwise minimum and maximum traveling costs between every two sub-partitions of the road map. However, these methods only solve the scalability issue for the number of clients but not for the amount of live traffic updates. As reported in [17], the re-computation time of the index takes 2 hours for the San Francisco (CA) road map. It is prohibitively expensive to update the index for OSP, in order to keep up with live traffic circumstances.

Motivated by the lack of off-the-shelf solution for OSP, in this paper we present a new solution based on the *index transmission model* by introducing *live traffic index* (LTI) as the core technique. LTI is expected to provide relatively short tune-in cost (at client side), fast query response time (at client side), small broadcast size (at server side), and light maintenance time (at server side) for OSP. We summarize LTI features as follows.

- The index structure of LTI is optimized by two novel techniques, graph partitioning and stochastic-based construction, after conducting a thorough analysis the hierarchical index techniques [19][20][21]. To the best of our knowledge, this is the first work to give a thorough cost analysis on the hierarchical index techniques and apply stochastic process to optimize the index hierarchical structure. (Section 4)
- LTI selectively fetches data in wireless broadcast environments, which significantly reduce the tune-in cost. (Section 5)
- LTI efficiently maintains the index for live traffic circumstances by incorporating Dynamic Shortest Path Tree (DSPT) [22] into hierarchial index techniques. In addition, a bounded version of DSPT is proposed to further reduce the broadcast overhead. (Section 6)
- By incorporating the above features, LTI reduces the tune-in cost up to an order of magnitude as compared to the state-of-the-art competitors; while it still provides competitive query response time, broadcast size, and maintenance time. To the best of our knowledge, we are the first work that attempts to minimize all these performance factors for OSP.

The rest of the paper is organized as follows. We first introduce main performance factors for evaluating OSP and overview the state-of-the-art shortest path computation methods in Section 2. The system overview and objectives of our live traffic index (LTI) are introduced in Section 3. The LTI construction, LTI transmission, and LTI maintenance are subsequently discussed in Section 4, Section 5 and Section 6, respectively. We summarize our complete framework in Section 7 and evaluate LTI thoroughly in Section 8. Finally, our work is concluded in Section 9.

2 PRELIMINARY

2.1 Performance Factors

The main performance factors involved in OSP are: (i) tune-in cost (at client side), (ii) broadcast size (at server side), and (iii) maintenance time (at server side), and (iv) query response time (at client side).

In this work, we prioritize the tune-in cost as the main optimized factor since it affects the duration of client receivers into active mode and power consumption is essentially determined by the tuning cost (i.e., number of packets received) [17][23]. Shortening the duration of active mode is important since it enables the clients to receive more services simultaneously by selective tuning [24]. These services may include providing live weather information, delivering latest promotions in the surrounding area, and monitoring availability of parking slots at destination. If we minimize the tune-in cost of one service, then we reserve more tune-in time for other services.

The index maintenance time and broadcast size relate to the freshness of the live traffic information. The maintenance time is the time required to update the index

according to live traffic information. The broadcast size is relevant to the latency of receiving the latest index information. As the freshness is one of our main design criteria, we must provide reasonable costs for these two factors.

The last factor is the response time at client side. Given a proper index structure, the response time of shortest path computation can be very fast (i.e., few milliseconds on large road maps) which is negligible compared to access latency for current wireless network speed. The computation also consumes power but their effect is outweighed by communication. It remains, however, an evaluated factor for OSP.

2.2 Adaptation of Existing Approaches

In this section, we briefly discuss the applicability of the state-of-the-art shortest path solutions on different transmission models. As discussed in the introduction, the result transmission model scales poorly with respect to the number of clients. The communication cost is proportional to the number of clients (regardless of whether the server transmits live traffic or result paths to the clients). Thus, we omit this model from the remaining discussion.

2.2.1 Raw transmission model

Under raw transmission model, the traffic data (i.e., edge weights) are broadcasted by a set of packets for each broadcast cycle. Each header stores the latest timestamp of the packets, so that clients can decide which packets have been updated, and only fetch those updated packets in the current broadcast cycle. Having downloaded the raw traffic data from the broadcast channel, the following methods either directly calculate the shortest path or efficiently maintain certain data structure for the shortest path computation.

Uninformed search (e.g., Dijkstra's algorithm) traverses graph nodes in ascending order of their distances from the source $\mathbf{s}$, and eventually discovers the shortest path to the destination $\mathbf{t}$. Bi-directional search (BD) [3] reduces the search space by executing Dijkstra's algorithm simultaneously forwards from $\mathbf{s}$ and backwards from $\mathbf{t}$. As to be discussed shortly, bi-directional search can also be applied on some advanced index structures. However, the response time is relatively high and the clients may receive large amount of irrelevant updates due to the transmission model.

Goal directed approaches search towards the target by filtering out the edges that cannot possibly belong to the shortest path. The filtering procedure requires some pre-computed information. ALT [25] and arc flags (AF) [26] are two representative algorithms in this category.

ALT makes use of A* search, landmarks, and triangle inequality [27]. A few landmark nodes are selected and the distances between each landmark and every node are pre-computed. These pre-computed distances can be exploited to derive distance bounds for A* search on the graph. [28] proposes a lazy update paradigm for ALT (DALT) so that it can tolerate certain extents of edge weights changes on a dynamic graph. The distance bounds derived from the pre-computed information remain correct if no edge weight becomes lower than the initial weight used at the ALT construction. This lazy update paradigm significantly reduces the index maintenance cost.

Another well known goal directed approach is Arc flags (AF) that partitions the graph into m sub-graphs. For each edge e, it stores a bitmap B where $B[i]$ is set to *true* if and only if a shortest path to a node in the sub-graph i starts with e. During the Dijkstra execution, it only relaxes those edges for which the bitmap flag of the target node's sub-graph is *true*. AF provides reasonable speed-ups, but consume too much space for large road networks. The dynamic updates of AF (DAF) has been recently studied in [29]. However, the solution is not practical since the cost of updating the bitmap flags is exponential to the number of edge updates.

Dynamic shortest paths tree (DSPT) maintains a tree structure locally for efficient shortest path retrieval. [22] discusses how to maintain a correct shortest paths tree rooted at $\mathbf{s}$ after receive a set of edge weight updates to the graph $G = (V, E)$. Finding a shortest path from $\mathbf{s}$ to any node is computed at $O(|V|)$ time on the shortest paths tree. However, maintaining a tree structure for an edge update is similar to execute an Dijkstra in the worst case. In addition, DSPT requires to keep a tree structure in every network node. This overhead makes DSPT infeasible for OSP on the index transmission model.

2.2.2 Index transmission model

The index transmission model enables servers to broadcast an index instead of raw traffic data. We review the state-of-the-art indices for shortest path computation and discuss their applicability on this model.

Road map hierarchical approaches try to exploit the hierarchical structure to the road map network in a pre-processing step, which can be used to accelerate all subsequent queries. These speed-up approaches include *reach* [4], *highway hierarchies* (HH) [2][6], *contraction hierarchies* (CH) [30], and *transit-node routing* (TNR) [1].

Reach, HH, and CH are based on shortcut techniques [2][6], i.e., some paths in the original graph are represented by some shortcut edges. The shortcuts are identified out by exploiting the hierarchical structure (e.g., node ordering) on the road map network. To answer a query, a bi-directional search is executed on the overlay graph that constitutes of the shortcuts and some edges in the original graph. As the shortcuts are the only extra structure stored in the index, the construction is relatively fast as compared to other index approaches.

TNR is based on a simple observation that a driving path only passes one of a few important transit nodes. For each shortest path query (s, t), two transit node sets, $\overrightarrow{A}(s)$ and $\overleftarrow{A}(t)$, can be identified by the forward and backward searches from the source and the destination, respectively. The length of the shortest path (s, t) that passes at least one transit node is given by $\min\{dist(s,u) + dist(u,v) + dist(v,t) \mid u \in \overrightarrow{A}(s), v \in \overleftarrow{A}(t)\}$, where all involved distances can be directly looked up in the pre-computed

data structure. Note that if the shortest path that passes no transit node, then other shortest path algorithm is applied instead.

The hierarchical approaches can provide very fast query time as reported in [31]. However, the maintenance time could be very high as most of them have no efficient approach to update the pre-computed data structure. HH and CH can support dynamic weight updates [7] but the solution is limited to weight increasing cases. In [32], a theoretical approach has been proposed to update the overlay graphs, but the proposed methods have not been tested in practice. Again, none of these approaches is scalable using the index transmission model since the shortest path can only be computed on a complete index.

Hierarchical index structures provide another way to abstracting and structuring a topographical index in a hierarchical fashion. Hierarchical MulTi-graph model (HiTi) [21] is a representative approach in this category. The meaning of hierarchy in HiTi is the hierarchy of the index (i.e., tree structure) instead of the hierarchy of the road map (i.e., level of roads). By exploiting the hierarchical index structure, HiTi can support fast shortest path computation on a portion of entire index which can significantly reduce the tune-in cost on the index transmission model. However, prohibitive maintenance time and large broadcast size make it inapplicable to OSP on any transmission model.

Hierarchical Encoded Path View (HEPV) [20] and Hub indexing [19] share the same intuition of HiTi which divides large graph into smaller subgraphs and organize them in a hierarchical fashion by pushing up border nodes. However, both are infeasible for OSP since these approaches suffer from the excessive storage overhead for a large amount of pre-computed path information.

TEDI [33] applies a tree-based partitioning on the road graph such that each partition in the tree has a bounded number of sub-partitions. This method is applicable to unweighted graphs only; it is not applicable to typical road networks where the edges are weighted. Furthermore, there is no discussion on how to maintain the TEDI structure in presence of edge weight updates.

Oracle Sankaranarayanan et al. [34][35] focus on precomputing certain shortest path distances called *oracles* in order to answer approximate shortest path queries efficiently. These techniques bound the approximate path distance error to be ϵ times the shortest path distance. The distance oracle [34] can answer approximate shortest path distance query in $O(\log |V|)$ time, and it occupies $O(|V|/\epsilon^2)$ space. The path oracle [35] takes the same complexity, and it can compute an approximate shortest path in $O(k \log |V|)$ time, where k is the number of vertices on the path. The maintenance of these oracles regarding live traffic updates has not been studied in [34][35]. Also, these techniques do not provide exact results and incur high storage space at small ϵ.

Full pre-computation pre-computes the shortest paths between any two nodes in the road network, such as SILC [36] and distance index [37]. Even though these approaches offer fast query response time, the maintenance cost and size overhead become prohibitive on large road networks. Besides, as reported by [38], the performance of the full pre-computation approaches (i.e., SILC [36]) is not much superior to those road map hierarchical approaches (i.e., CH [30]).

Combination approaches integrate promising features from different index structure to support efficient shortest path computation. SHARC [39] and CALT [31] are two well studied combination approaches which integrate road map hierarchical approaches with AF and ALT, respectively. However, these complex index structures are either lack of efficient maintenance strategies or have huge size overhead which makes them inapplicable to OSP on any transmission model.

2.2.3 Summary

Except for hierarchical index structures, all methods on either raw or index transmission models suffer from a drawback that a few updates could affect a large portion of packets. We demonstrate this by a simple probabilistic analysis. Suppose that there are B packets in a broadcast cycle and U edges are updated. The probability of a specific packet being affected by the updates is $1-(1-1/B)^U$. For instance, suppose that the San Francisco (CA) bay area road network can be transmitted in 1,000 packets (443,604 edges in total), and there are 1% live traffic updates (i.e., ≈4,400 edges) in each processing cycle. Thus, the probability of a packet being affected is 98.77%. This means that almost every broadcast packet is updated by this small portion of edge updates.

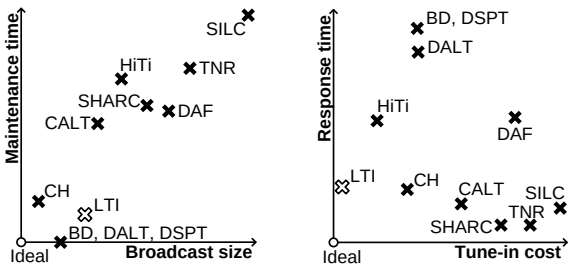
Fig. 2 illustrates the *relative* performance[1] to different cost factors (including tune-in cost, response time, broadcast size, and maintenance time) on two transmission models. Except that BD, DALT, and DSPT use the raw transmission model, other methods use the index transmission model. Except for HiTi [21] and LTI (our proposed method), the tune-in cost of all approaches is very close to the broadcast size as explained above. Based on the comparison in Fig. 2, LTI is the only method that supports relatively low tune-in cost (at client side), fast query response time (at client side), small broadcast size (at server side), and light index maintenance time (at server side) for OSP.

3 LTI Overview and Objectives

3.1 LTI Overview

A road network monitoring system typically consists of a service provider, a large number of mobile clients (e.g., vehicles), and a traffic provider (e.g., GoogleMap, NAVTEQ, INRIX, etc.). Fig.3 shows an architectural overview of this system in the context of our live traffic index (LTI) framework. The *traffic provider* collects the live traffic circumstances from the traffic monitors via techniques like road sensors and traffic video analysis. The *service provider*

1. based on the experimental results reported by [31] and [38]



(a) costs at service provider (b) costs at client

Fig. 2. Relative performance illustration

periodically receives live traffic updates from the traffic provider and broadcasts the live traffic index on radio or wireless network (e.g., 3G, LTE, Mobile WiMAX, etc.). When a *mobile client* wishes to compute and monitor a shortest path, it listens to the live traffic index and reads the relevant portion of the index for deriving the shortest path.

In this work, we focus on handling traffic updates but not graph structure updates. For real road networks, it is infrequent to have graph structure updates (i.e., construction of a new road) when compared to edge weight updates (i.e., live traffic circumstances). Thus, we assume that the graph structures are distributed to every client in advance (e.g., by monthly updates or at system boot-up) via typical transmission protocol (i.e., HTTP and FTP).

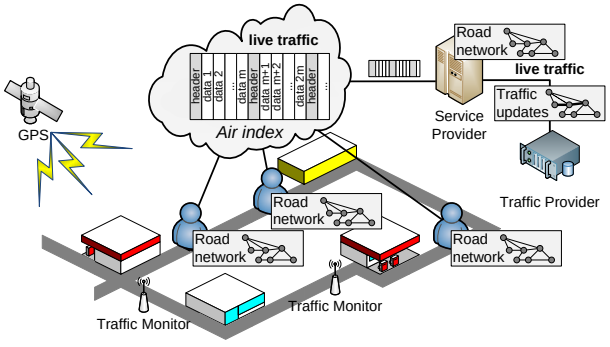


Fig. 3. LTI System Overview

In Fig.4, we illustrate the components and system flow in our LTI framework. The components shaded by gray color are the core of LTI. In order to provide live traffic information, the server maintains (component a) and broadcasts (component b) the index according to the up-to-date traffic circumstances. In order to compute the online shortest path, a client listens to the live traffic index, reads the relevant portions of the index (component c), and computes the shortest path (component d).

3.2 LTI Objectives

To optimize the performance of the LTI components, our solution should support the following features.

(1) Efficient maintenance strategy. Without efficient maintenance strategy, long maintenance time is needed at server side so that the traffic information is no longer *live*. This can reduce the maintenance time spent at component a.

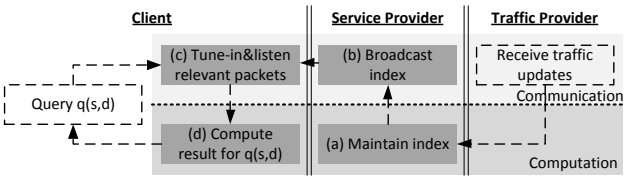


Fig. 4. Components in LTI

(2) Light index overhead. The index size must be controlled in a reasonable ratio to the entire road map data. This reduces not only the length of a broadcast cycle, but also makes clients listen fewer packets in the broadcast channel. This can save the communication cost at components b and c.

(3) Efficient computation on a portion of entire index. This property enables clients to compute shortest path on a portion of the entire index. The computation at component d gets improved since it is executed on a smaller graph. This property also reduces the amount of data received and energy consumed at component c.

Inspired by these properties, LTI has relatively short tune-in cost (at client side), fast query response time (at client side), small broadcast size (at server side), and light index maintenance time (at server side) for OSP. As discussed in Section 2.2, the *hierarchical index structures* enable clients to compute the shortest path on a portion of entire index. However, without pairing up with the first and second features, the communication and computation costs are still infeasible for OSP. To achieve these two features, in Section 4 and Section 6, we will discuss how to optimize the hierarchical structure and efficiently maintain the index according to live traffic circumstances.

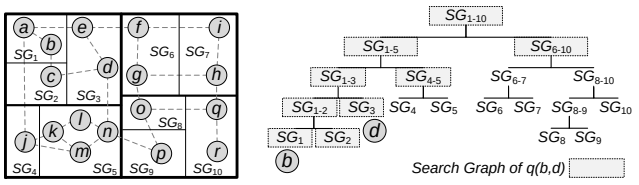
4 LTI Construction

In Section 4.1, we carefully analyze the hierarchical index structures and study how to optimize the index. In Section 4.2, we present a stochastic based index construction that minimizes not only the size overhead but also reduces the search space of shortest path queries. To the best of our knowledge, this is the first work to analyze the hierarchical index structures and exploit the stochastic process to optimize the index.

4.1 Analysis of Hierarchical Index Structures

Hierarchical index structures (e.g., HiTi [21], HEPV [20], and Hub Indexing [19], TEDI [33]) enable fast shortest path computation on a portion of entire index which significantly reduces the tune-in cost on the index transmission model. Given a graph $G = (V_G, E_G)$ (i.e., road network), this type of index structures partitions G into a set of small subgraphs SG_i and organizes SG_i in a hierarchical fashion (i.e., tree). In Fig. 5, we illustrate a graph being partitioned

into 10 subgraphs (SG_1, SG_2, ..., SG_{10}) and the corresponding hierarchical index structure.



(a) road network (b) hierarchical tree view

Fig. 5. Hierarchical index structure

Every leaf entry in a hierarchical structure represents a subgraph SG_i that consists of the corresponding nodes and edges from the original graph. For instance, SG_1 consists of two nodes $V_{SG_1} = \{a, b\}$ and one edge $E_{SG_1} = \{(a, b)\}$. A non-leaf entry stores the inter-connectivity information between the child entries. For instance, $SG_{1\text{-}2}$ stores a connectivity edge $\Gamma_{SG_{1\text{-}2}} = \{(b, c)\}$ between SG_1 and SG_2. To boost up the shortest path computation, the hierarchical index structures also keep some pre-computed information in the index entries. For instance, *shortcuts* Δ_{SG_i} are the most common type of pre-computed information in these indices, where a shortcut is the shortest path between two border nodes in a subgraph. In Fig. 5, SG_5 has two border nodes[2] k and m so that SG_5 keeps a shortcut $\Delta_{SG_5} = \{(k, m)\}$ and its corresponding weight.

To answer a shortest path query $q(\mathbf{s}, \mathbf{t})$ using the hierarchical structures, a common approach is to fetch the relevant entries from the index using a bottom-up execution fashion. For the sake of analysis, we use HiTi as our reference model in the remaining discussion. Our analysis can be adapted to other approaches since their execution paradigm shares the same principle.

In Fig. 5, the relevant entries of a shortest path query $q(b, d)$ are shaded in gray color. Besides the source and destination leaf entries (SG_1 and SG_3), we need to fetch the entries from two leaf entries towards the root entry ($SG_{1\text{-}2}$, $SG_{1\text{-}3}$, $SG_{1\text{-}5}$, and $SG_{1\text{-}10}$) and their sibling entries (SG_2, $SG_{4\text{-}5}$, and $SG_{6\text{-}10}$). The shortest path is computed on the *search graph* G^q (typically much smaller than G) which constitutes of the edges from the source and destination entries and the connectivity edges and shortcuts from other relevant entries. Note that the edges in G^q already secure the correctness of the shortest path query process [21]. As an example, suppose the shortest path of $q(b, d)$ passes through an edge in SG_6, this path must be revealed in the shortcut of $SG_{6\text{-}10}$ (i.e., $\Delta_{SG_{6\text{-}10}} = \{(f, p)\}$).

Cost analysis. The total space requirement of a hierarchical index I can be represented as follows.

$$|I| = \sum_{SG_i \in I} (|V_{SG_i}| + |E_{SG_i}| + |\Gamma_{SG_i}| + |\Delta_{SG_i}|) + tree \quad (1)$$

where V_{SG_i} and E_{SG_i} represent the nodes and edges in SG_i, respectively, Γ_{SG_i} represents the connectivity information between the child entries, Δ_{SG_i} represents the pre-computed information kept in SG_i, and $tree$ represents the hierarchical information of I. Since V_{SG_i}, E_{SG_i} and Γ_{SG_i} are directly derived from the original graph and $tree$ is negligible compared to G, the space requirement can be revised as the follows.

$$|I| \approx |G| + \sum_{SG_i \in I} |\Delta_{SG_i}| \quad (2)$$

To minimize the index broadcast size, it is more or less equivalent to minimize the size of Δ_{SG_i}. The simplest way is to partition the graph into multiple subgraphs such that the total size of Δ_{SG_i} is minimized. However, this may not optimize the query performance being discussed shortly.

In our problem, both tune-in cost and query response time are highly relevant to the size of the search graph G^q (i.e., search space). Given an index I and a query q, the search space of q can be represented by the relevant edge sets.

$$\mathcal{S}(I, q) = |E_{SG_\mathbf{s}} \cup E_{SG_\mathbf{t}} \cup \{\Gamma_{SG_i} \cup \Delta_{SG_i} : \forall SG_i \in G^{q-\mathbf{st}}\}| \quad (3)$$

where $SG_\mathbf{s}$ and $SG_\mathbf{t}$ represent the leaf entry of source and destination, respectively and $G^{q-\mathbf{st}} = G^q \setminus \{SG_\mathbf{s} \cup SG_\mathbf{t}\}$. To reduce $\mathcal{S}(I, q)$ for all possible queries, our goal is to find a hierarchical structure such that it minimizes **(O1)** the size of leaf entries $E_{SG_\mathbf{s}}$ and $E_{SG_\mathbf{t}}$, **(O2)** the overhead of pre-computed information Δ_{SG_i}, and **(O3)** the number of relevant entries G^q. However, these objectives are correlated to each other. For instance, to make $E_{SG_\mathbf{s}}$ and $E_{SG_\mathbf{t}}$ smaller, a simple way is to partition G into more subgraphs; however, it may increase the number of relevant entries G^q and the number of border nodes Δ_{SG_i}.

4.2 Index Construction

The above discussion shows that it is hard to find a hierarchical index structure I that achieves all optimization objectives. One possible solution is to relax the optimization objectives which makes them be the tunable factors of the problem. While the overhead of pre-computed information (**O2**) and the number of relevant entries (**O3**) cannot be decided straightforwardly, we decide to relax the first objective (i.e., minimizing the size of leaf entries) such that it becomes a tunable factor in constructing the index.

To minimize the overhead of pre-computed information (**O2**), we study a graph partitioning optimization that minimizes the index overhead Δ_{SG_i} through the entire index construction subject to a leaf entry constraint (**O1**). Subsequently, we propose a stochastic process to optimize the index structure such that the size of the query search graph G^q is minimized (**O3**).

Graph partitioning optimization. For the sake of discussion, we denote that the number of subgraphs being created is γ that is a tunable parameter for controlling the number of subgraphs [3] in this work. According to Eq. 2, minimizing the size of Δ_{SG_i} is likely to minimize the overhead of I. Obviously, our objective is to find a hierarchical index structure I such that

$$\mathcal{OBJ}(I) = \min_{SG_i \in I} \frac{\sum |\Delta_{SG_i}|}{\min\{|V_{SG_i}|\}} \quad (4)$$

2. node n is a border node in $SG_{4\text{-}5}$ but not in SG_5.

3. γ can be viewed as a parameter to control the size of subgraphs.

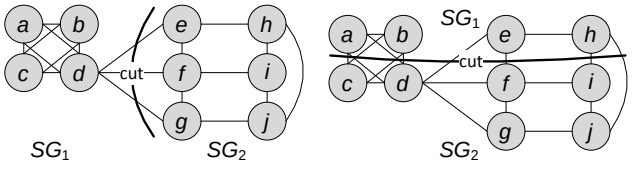
where $\min\{|V_{SG_i}|\}$ can be viewed as a normalized factor such that the objective function prefers balanced partitions.

We observe that minimizing Eq. 4 is similar to finding the best *Cheeger cut* [40] in a graph. A *Cheeger cut* is to remove some edges from a graph such that the graph is isolated into n subgraphs subject to an objective function:

$$\mathcal{OBJ}_{Cheeger}(G) = \min_{SG_1,...,SG_n \in G} \frac{Cut(\{SG_1, ..., SG_n\})}{\min\{|SG_1|, ..., |SG_n|\}} \quad (5)$$

where $Cut(\{SG_1, ..., SG_n\})$ is the number of edges between any two subgraphs.

We use an example to illustrate how *Cheeger cut* result can be viewed as a good result of I. Fig.6(a) shows a *Cheeger cut* on a graph where the cut value $Cut(\{SG_1, SG_2\})$ and the number of shortcut edges, $|\Delta_{SG_1}| + |\Delta_{SG_2}| = |\{\emptyset\}| + |\{(e,f),(e,g),(f,g)\}|$, are identical (i.e., 3). A large cut value is likely to produce more shortcut edges; in Fig.6(b), the cut value is 10 and there are 12 shortcuts.



(a) 3 shortcut edges (b) 24 shortcut edges

Fig. 6. The number of shortcut edges created by different *cuts*

Lemma 1: Given a cut having the cut value c and the maximum *cut* degree of border nodes d_{cut} (where the *cut* degree only counts on those edges being cut), the number of shortcuts produced by the cut is bounded by $\frac{c(c-1)}{2} + \frac{(c-d_{cut}+1)(c-d_{cut})}{2}$.

Proof: Given a cut having the cut value c and the maximum *cut* degree d_{cut}, the maximum number of border nodes in two subgraphs being created is c and $c-d_{cut}+1$, respectively. Thereby, the maximum number of shortcuts being created is $\frac{c(c-1)}{2} + \frac{(c-d_{cut}+1)(c-d_{cut})}{2}$. □

Lemma 1 provides a relationship between the *Cheeger* cut and our objective function. Practically, the effect of d_{cut} is negligible since d_{cut} is relatively small as compared to c. For instance, the average in/out degree of the San Francisco (CA) bay area vertices is only 2.54 (which means d_{cut} is smaller than 2.54 since a portion of these edges are interior edges.). Therefore, we claim that a partitioning result that minimizes Eq. 5 is likely to minimize Eq. 4 as well.

Finding the best *Cheeger cut* can be reduced to a quadratic discrete optimization problem [41]. Based on [41], a *cut* on a graph can be determined by the second smallest eigenvalue λ and its corresponding eigenvector $\mathcal{V}$. The problem becomes to decompose $\mathcal{V}$ into two subsets such that the objective function is minimized. To further improve the quality of each cut, we use Eq. 4 as the objective function so that we can heuristically reduce the number of border nodes. To construct an index, we recursively *cut* the subgraphs until we have enough partitions (i.e., the leaf entry constraint, γ). The pseudo code is omitted due to space limits.

Stochastic based index construction. The graph partitioning framework only returns a binary tree index I^{bi} that is constructed based on *Cheeger cut* sequences. However, I^{bi} only fulfills the first two objectives (i.e., minimizing the overhead $|\Delta_{SG_i}|$ subject to γ). Intuitively, the size of search graphs G^q (i.e., **O3**) is highly relevant to the index hierarchical structure. As a motivating example, the number of relevant entries of $q(b,d)$ is reduced from 9 to 8 if we remove one index node (e.g., SG_{1-2}) from the index tree in Fig. 5(b). The new index and the relevant entries are illustrated in Fig. 7.

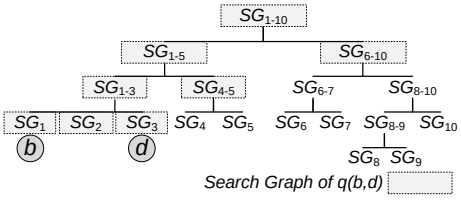


Fig. 7. Effect of hierarchical structure

Given the size of leaf entries γ, minimizing the size of search graph can be viewed as a problem of finding the best hierarchical index structure for potential queries. Finding the optimal hierarchical structure is challenging since (1) the performance of an index cannot be easily estimated (which should be estimated by a query workload $\mathcal{Q}$ or a universal query set $\mathcal{U}$) and (2) the index statistics (e.g., shortcuts) are changed on different index hierarchical structures (which is necessarily recalculated based on the structure). This problem is similar to those combinatorial optimization problems (e.g., hierarchical clustering [42]) that groups data into a set of hierarchical partitions such that the objective function is optimized. Typically, these combinatorial problems are solved by approximate solutions under reasonable response time. Thus, we propose a top-down approach that greedily decides the structure based on a stochastic estimation.

To estimate the average size of the search graphs, we apply a stochastic process, Monte Carlo, that relies on random sampling to obtain numerical results. In this work, the Monte Carlo process is to execute a set of randomly generated shortest path queries on a temporal index I' and estimates the average size of their relevant search graphs, $\text{avg}(\mathcal{S}(I'))$. For clarity, the stochastic process can be replaced by a query workload, $\mathcal{Q}$, based estimation which should offer more accurate estimation when $\mathcal{Q}$ is available.

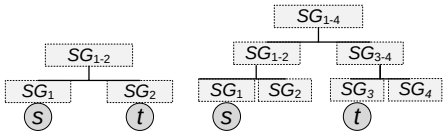
At every partitioning, we attempt to find the best structure for the potential queries by the stochastic process. More specifically, we assess the average size of the relevant search graphs, $\text{avg}(\mathcal{S}(I'))$, for different partitioning settings (i.e., varying k). Among all assessed partitioning, we attach the partitioning having the smallest relevant search graphs to the index. The construction terminates when we have enough leave entries (i.e., γ). Algorithm 1 shows the pseudo

codes of the partitioning algorithm based on the stochastic process.

Algorithm 1 Stochastic Partitioning Algorithm

```
PQ: a priority queue; I: index structure;
Algorithm partition(G:the graph, γ:the number of partitions)
1: (λ, V) := eigen(G) and n := root of I
2: insert (n, G, V, λ) into PQ in decreasing order to λ
3: while |PQ| < γ do
4:     (n, G, V, λ) := PQ.pop()
5:     for k:=2 to γ − |PQ| + 1 do
6:         decompose G into SG_1...SG_k s.t. Eq. 4 is minimized
7:         form a temporal index I' that attaches SG_1...SG_k
8:         if avg(S(I')) is better than best_S then
9:             update best_S and best_SG := {SG_1, ..., SG_k}
10:    attach best_SG as n's children
11:    for i:=1 to |best_SG| do
12:        insert (n_i, SG_i, V_i, λ_i) into PQ
13: return I
```

The effect of γ. In this work, LTI requires only one parameter γ to construct the index which is used to control the number of subgraphs being constructed. Our proposed techniques attempt to optimize the index (**O2 and O3**) subject to γ. Intuitively, similar to other hierarchical indices, the number of leaf entries, γ, not only affects the size of leaf entries but also the search performance.



(a) small γ (b) large γ

Fig. 8. The effect of γ on index construction

Fig. 8 illustrates a toy example that shows the effect of γ in different settings. Obviously, when γ is set to a large value, the index has small size of leaf entries (i.e., E_{SG_s} and E_{SG_t}) which boost the query processing due to the locality of relevant entries. However, it may increase the number of relevant entries to answer queries due to the hierarchy of the index. In summary, a small γ may lead the index having large leaf entries while a large γ may lead the index having large number of index nodes, where these settings may degrade the query performance. Fortunately, γ is not a very sensitive parameter (cf. the studies in other hierarchical indexing techniques [19][20][21] and our experiments), which can be decided by experimental studies in practice.

5 LTI TRANSMISSION

In this section, we present how to transmit LTI on the air index. We first introduce a popular broadcasting scheme called the $(1, m)$ interleaving scheme in Section 5.1. Based on this broadcasting scheme, we study how to broadcast LTI in Section 5.2 and how a client receives edge updates on air in Section 5.3.

5.1 Broadcasting Scheme

The broadcasting model uses radio or wireless network (e.g., 3G, LTE, Mobile WiMAX) as the transmission medium. When the server broadcasts a dataset (i.e., a "programme"), all clients can listen to the dataset concurrently. Thus, this transmission model scales well independent of the number of clients. A broadcasting scheme is a protocol to be followed by the server and the clients.

TABLE 1
The format of the $(1, m)$ interleaving scheme

header	data	header	data	header	data
i:0, n:6	o_1, o_2	i:2, n:6	o_3, o_4	i:4, n:6	o_5, o_6

The *(1,m) interleaving scheme* [23] is one of the best broadcasting schemes. Table 1 shows an example broadcasting cycle with $m = 3$ packets and the entire dataset contains 6 data items. First, the server partitions the dataset into m equi-sized data segments. Each *packet* contains a header and a data segment, where a *header* describes the broadcasting schedule of all packets. In this example, the variables i and n in each header represent the last broadcasted item and the total number of items. The server periodically broadcasts a sequence of packets (called as a broadcast cycle).

We use a concrete example to demonstrate how a client receives her data from the broadcast channel. Suppose that a client wishes to query for the data object o_5. First, the client tunes in the broadcast channel and waits until the next header is broadcasted. For instance, the client is listening to the header of the first packet, and finds out that the third packet contains o_5. In order to preserve energy, the client sleeps until the broadcasting time of that packet. Then, it wake-ups and reads the requested data item from the packet.

The query performance can be measured by the tuning time and the waiting time at the client side. The tuning time is the time for reading the packets. The waiting time is the time from the start time to the termination time of the query. In this broadcasting scheme, the parameter m decides the trade-off between tune-in size and the overhead. A large m favors small tune-in size whereas a small m incurs small waiting time. [23] suggests to set m to the square root of the ratio of the data size to the index size.

5.2 LTI on Air

To broadcast a hierarchical index using the (1,m) interleaving scheme, we first partition the index into two components: the index structure and the weight of edges. The former stores the index structure (e.g., graph vertices, graph edges, and shortcut edges) and the latter stores the weight of edges. In order to keep the freshness of LTI, our system is required to broadcast the latest weight of edges periodically [4].

Table 2 shows the format of a header/data packet in our model. `id` is the offset of the packet in the present broadcast cycle and `checksum` is used for error-checking of the header and data. Note that the packet does not store any offset information to the next broadcast cycle or

4. The hierarchical index structure is only affected by the connectivity / topology information (see Sections 4 and Sections 6). Thus, any changes in the weights would not affect the hierarchical index structure.

TABLE 2
Packet format on the air index

Offset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	id				checksum				...							
...	...															

broadcast segment. The offset can be matched up by the corresponding id since the structure of LTI is pre-stored at each client. In our model, the header packet stores a timestamp set T for checking new updates and data loss recovery.

5.3 Client Tune-in Procedures of Air LTI

We proceed to demonstrate how a client (i.e., driver) receives edge weights from the air index using the hierarchical structure. Fig. 9 shows the content of a broadcast cycle for a LTI structure in Fig.7. In this example, the air index uses a $(1, 2)$ interleaving scheme and each data packet stores the edge weight of different subgraphs. For instance, the edge weight of subgraph SG_1 are stored in the 2^{nd} packet of a broadcast cycle. Assume that a driver is moving from node b to node d and his navigation system first tunes-in to the air index at the 3^{rd} packet of segment 1. According to the search graph (as shown in Fig. 7) and the packet id, the navigation system falls into sleep for 1 segment transmission time. It wakes up and receives segment 3 where the search graph elements ($SG_{1\text{-}3}$ and $SG_{4\text{-}5}$) are located in. Note that the other search graph elements (SG_1, SG_2, and SG_3) in segment 1 can only be collected in the next broadcast cycle.

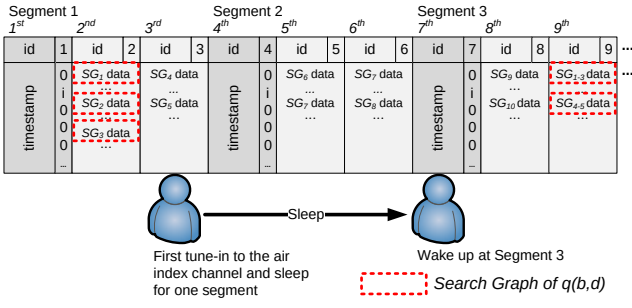


Fig. 9. Receiving LTI data from the air index

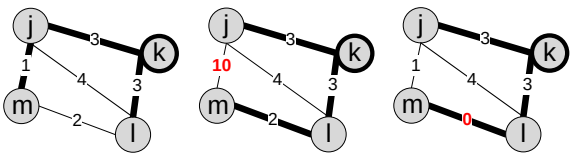
Suppose that there are two edge updates, including one graph edge (k, l) in SG_5 and one shortcut (j, n) in $SG_{4\text{-}5}$, in the next broadcast cycle. The navigation system identifies the subgraphs being updated by checking the timestamp set T in the header packet. Since the search graph G^q contains $SG_{1\text{-}3}$ and $SG_{4\text{-}5}$, the system tunes-in to the air index when the corresponding packets are broadcasted (i.e., the 3^{rd} packet of segment 3).

6 LTI Maintenance

In order to keep the freshness of the broadcasted index, the cost of index maintenance is necessarily minimized. In this section, we study an incremental update approach that can efficiently maintain the live traffic index according to the updates. As a remark, the entire update process is done at the service provider and there is no extra data structure being broadcasted to the clients.

There is a bottom-up update framework to maintain the hierarchical index structure in [21]. Their idea is to recompute the affected subgraphs starting from lowest level (i.e., leaf subgraphs) to root. Unfortunately, as shown in Section 2.2, a small portion of edge updates trigger updates in the majority of packets (i.e., subgraphs). Thus, the above update technique incurs high computational cost on updating the affected subgraphs.

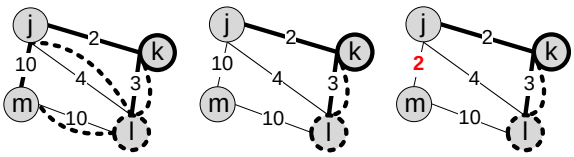
It is thus necessary to develop a more efficient update framework. For any weight update on the road edges, we observe that only shortcut edges Δ_{SG_i} are necessarily recomputed as the weight of other edges (i.e., $E_{SG_i} \cup \Gamma_{SG_i}$) are directly derived from the updates. To boost the shortcut edge maintenance, we incorporates dynamic shortest path tree technique (DSPT) [22] into the hierarchical index structures and reduce the overhead of DSPT by a bounded version (BSPT).



(a) $SPT(k)$ (b) w_{jm} increases (c) w_{ml} decreases

Fig. 10. Shortest path tree maintenance

Given a graph $G = (V, E)$, a shortest path tree (SPT) rooted at a vertex $\mathbf{r} \in V$, denoted as $SPT(\mathbf{r})$, is a tree with root $\mathbf{r}$, and $\forall v \in V - \{\mathbf{r}\}$, $SPT(\mathbf{r})$ contains a shortest path from $\mathbf{r}$ to v. In Fig.10(a), the shortest path tree of vertex k is highlighted by bold lines. Given a shortest path tree, a dynamic Dijkstra approach [22] is proposed for handling both weight increasing (Fig.10(b)) and decreasing cases (Fig.10(c)). The intuition of the algorithms is to find the affected local vertices and revise the shortest path tree using a Dijkstra like algorithm starting from the updated vertices. For instance, the weight of $e(m, l)$ is decreased from 2 to 0. Starting from the vertex m, a new path $m \rightarrow l \rightarrow k$, that is a better path from m to k, is found by the Dijkstra searching. Thereby, the update process revises the shortest path tree accordingly as shown in Fig.10(c).



(a) Two SPTs (b) Two BSPTs (c) $e(j, m)$ update

Fig. 11. Shortest path trees and updates

To keep the freshness of LTI, every subgraph is required to maintain its corresponding shortcut edges Δ_{SG_i} according to live traffic circumstances. The weight of these shortcuts can be maintained by the corresponding shortest path tree from each border node B_{SG_i}. Obviously,

the total space overhead of the shortest path trees is $\sum_{SG_i \in I} |B_{SG_i}| \cdot |V_{SG_i}|$. To reduce the space overhead and boost the maintenance process, we observe that not every edge in a $SPT(v)$ is necessarily kept for the maintenance process. We illustrate this by a concrete example in Fig.11(a). Suppose that only vertices k and l are the border nodes and $\Delta_{SG_i} = \{(l,k)\}$. We can say edge (j,m) is irrelevant to shortcut (l,k) since the distance from l to j or m is already longer than the distance from l to k. More specifically, changing the weight of edge (j,m) does not influence the shortest path from l to k in $SPT(l)$.

Definition 1 (Bounded shortest path tree): Given a subgraph $SG_i = (V_{SG_i}, E_{SG_i})$, a bounded shortest path tree (BSPT) rooted at a vertex $s \in V_{SG_i}$, denoted as $BSPT(s)$, is a tree with root $\mathbf{s}$, and $BSPT(\mathbf{s})$ contains a shortest path from $\mathbf{s}$ to $v \in V - \{\mathbf{s}\}$ subject to $d(\mathbf{s}, v) \leq \max_{v' \in \Delta_{SG_i}} d(\mathbf{s}, v')$.

Inspired by the discussion, we propose a variant shortest path tree, named as bounded shortest path tree $BSPT(v)$, in Def. 1. A shortest path starting from v is necessarily kept in a bounded shortest path tree $BSPT(v)$ if and only if the distance of the shortest path is shorter than the distance from v to every border node. In Fig.11(b), $BSPT(l)$ keeps only one shortest path $l \rightarrow k$. The shortest path $l \rightarrow m$ and $l \rightarrow j$ are dropped since the shortest distance of $l \rightarrow m$ and $l \rightarrow j$ is not shorter than the distance from l to border node k. Typically, $BSPT(v)$ is much smaller than $SPT(v)$ and it also boosts up computation efficiency due to smaller search space.

Lemma 2 (Relevance of weight updates): A bounded shortest path tree $BSPT(v)$ is affected by the weight update of an edge e if and only if e is adjacent to at least one vertex of $BSPT(v)$.

Proof: Assume to the contrary that a bounded shortest path tree, $BSPT(v)$, is affected by the weight updated of an edge $e(v_e, v'_e)$. In other words, based on Def. 1, the path distance of some vertex v' in $BSPT(v)$ becomes smaller if the weight of e is changed, i.e.,

$$\begin{aligned} d(v \rightarrow ... \rightarrow v') &> d(v \rightarrow ... v_e \rightarrow v'_e ... \rightarrow v') \\ &= d(v \rightarrow ... \rightarrow v_e) + d(v_e \rightarrow v'_e \rightarrow ... \rightarrow v') \\ &> d(v \rightarrow ... \rightarrow v_e) \end{aligned}$$

According to the assumption, e is not adjacent to $BSPT(v)$. Thereby,

$$\begin{aligned} d(v \rightarrow ... \rightarrow v') &> d(v \rightarrow ... v_i \rightarrow v_j ... \rightarrow v_e) \\ &= d(v \rightarrow ... v_i \rightarrow v_j) + d(v_j \rightarrow ... \rightarrow v_e) \end{aligned}$$

where $v_i \in BSPT(v)$ and $v_j \notin BSPT(v)$. However, it is in contradiction to Def. 1 since $d(v \rightarrow ... \rightarrow v') \leq \max_{v' \in \Delta_{SG_i}} d(s, v')$ but $d(v \rightarrow ... v_i \rightarrow v_j) > \max_{v' \in \Delta_{SG_i}} d(s, v')$. □

Based on Def. 1, we study Lemma 2 that provides the relevance of an edge update to a bounded shortest path tree. In the running example (Fig. 11(c)), the edge update (j,m) is relevant to $BSPT(k)$ but not $BSPT(l)$ since it is not adjacent to any vertex of $BSPT(l)$. In other words, $BSPT(l)$ is not necessary to maintain for this edge update. This lemma enables the system to omit update maintenance to those irrelevant bounded shortest path trees which can significantly reduce the maintenance cost. Besides, the maintenance does not increase any communication overhead since LTI only delivers the weight of border node pairs (i.e., Δ_{SG_i}) but not the shortest path trees on the index transmission model.

Pruning ability of BSPT. The pruning ability of BSPT is highly relevant to the border node selection in each subgraph. In the worst case, BSPT performs as the same as a naïve SPT if the borders are very far from each others. However, such cases rarely happen in LTI since the graph partitioning technique (Section 4.2) prefers a partitioning having small number of borders, which minimizes the change of the worst-case scenario. In our study, BSPT prunes 30% to 50% edges from the complete SPT for our evaluated datasets (Section 8).

7 Putting All Together

We are now ready to present our complete LTI framework, which integrates all techniques been discussed. A client can invoke Algorithm 2 in order to find the shortest path from a source $\mathbf{s}$ to a destination $\mathbf{t}$. First, the client generates a search graph G^q based on $\mathbf{s}$ (i.e., current location) and $\mathbf{d}$. When the client tunes-in the broadcast channel (cf. Section 5.2), it keeps listening until it discovers a header segment (cf. Figure 9). After reading the header segment, it decides the necessary segments (to be read) for computing the shortest path. These issues are addressed in Section 5.3. The client then waits for those segments, reads them, and update the weight of G^q. Subsequently, G^q is used to compute the shortest path in the client machine locally (cf. Figure 7 and Section 4.1). Note that Algorithm 2 is kept running in order to provide online shortest path until the client reaches to the destination.

We then discuss about the tasks to be performed by the service provider, as shown in Algorithm 3. The first step is devoted to construct the live traffic index; they are offline tasks to be executed once only. The service provider builds the live traffic index by partitioning the graph G into a set of subgraphs $\{SG_i\}$ such that they are ready for broadcasting. We develop an effective graph partitioning algorithm for minimizing the total size of subgraphs and study a combinatorial optimization for reducing the search space of shortest path queries in Section 4.2. In each broadcasting cycle, the server first collects live traffic updates from the traffic provider, updates the subgraphs $\{SG_i\}$ (discussed in Section 6), and eventually broadcasts them.

Algorithm 2 Client Algorithm

Algorithm $Client(I$:LTI, $\mathbf{s}$:source, $\mathbf{t}$:destination)
1: generate G^q from I based on $\mathbf{s}$ and $\mathbf{d}$
2: listen to the channel for a header segment
3: read the header segment ▷ Section 5.3
4: decide the necessary segments to be read ▷ Section 5.3
5: wait for those segments, read them to update the weight of G^q
6: compute the shortest path (from $\mathbf{s}$ to $\mathbf{t}$) on G^q ▷ Section 4.1

8 Experimental Evaluation

In this section, we empirically evaluate the performance of some representative algorithms using the broadcasting

Algorithm 3 Service Algorithm

Algorithm $Service(G$:graph)
1: construct I and $\{SG_i\}$ based on G ▷ Section 4.2
2: **for** each broadcast cycle **do**
3: collect traffic updates from the traffic provider
4: update the subgraphs $\{SG_i\}$ ▷ Section 6
5: broadcast the subgraphs $\{SG_i\}$ ▷ Section 5.2

architecture; we ignore the client-server architecture due to massive live traffic in near future (see Section 1). From our discussion in Section 2, bi-directional search (BD) [3], ALT on dynamic graph (DALT) [28], and dynamic shortest paths tree (DSPT) [22], are applicable to *raw transmission model*. On the other hand, contraction hierarchies (CH) [30], Hierarchical MulTi-graph model (HiTi) [21], and our proposed live traffic index (LTI) are applicable to *index transmission model*. We omit some methods (such as TNR [1], Quadtree [36], SHARC [39], and CALT [31]) due to their prohibitive maintenance time and broadcast size. In the following, we first describe the road map data used in experiments and describe the simulation of clients' movements and live traffic circumstances on a road map. Then, we study the performance of the above methods with respect to various factors.

Map data. We test with four different road maps, including New York City (NYC) (264k nodes, 733k edges), San Francisco bay area road map (SF) (174k nodes, 443k edges), San Joaquin road map (SJ) (18k nodes, 48k edges), and Oldenburg road map (OB) (6k nodes, 14k edges). All of them are available at [43] and [44].

Simulation of clients and traffic updates. We run the network-based generator [44] to generate the weight of edges. It initializes 100,000 cars (i.e., clients) and then generates 1,000 new cars in each iteration. It runs for 200 iterations in total, with the other generator parameters as their default values. The weight of an edge is set to the average driving time on it.

We adopt the approach in [28] to simulate live traffic updates. The initial weights of edges are assigned by the above network-based generator. In each iteration, we randomly select a set of edges subject to the update ratio δ and specific weight update settings. In our work, each weight update can be either a *light* traffic change, a *heavy* traffic change, or a road *maintenance*. The proportion of these update types are β, $\frac{1-\beta}{2}$, and $\frac{1-\beta}{2}$, respectively, where β is a ratio parameter. For each light traffic change, the edge weight is set to $\pm 20\%$ of the current weight. For each heavy traffic change, the weight is set to a large value by multiplying a weight factor ω (which is set to 5 by default). For each road maintenance, the weight is set to ∞. We reset the edge weight to its initial value if the edge weight is updated by heavy traffic or road maintenance after 10 iterations.

Implementation and evaluation platforms. All tested methods except CH [30] were implemented in Java. Experiments on the service provider were conducted on an Intel Xeon E5620 2.40GHz CPU machine with 18 GBytes memory, running Ubuntu 10.10; and experiments on the client were performed on an Intel Core2Duo 2.66GHz CPU machine with 4 GBytes memory, running Windows 7. Table 3 shows the ranges of the investigated parameters, and their default values (in bold). In each experiment, we vary a single parameter, while setting the others to their default values. For each method, we measure its performance in terms of *tune-in size*, *query response time*, *broadcast size*, and *index maintenance time* for all tested methods, and report its average performance over 2,000 shortest path queries. The response time is the query computation time at client and the maintenance time is the index maintenance time at service provider. In order to measure the exact transmission behavior, we use the number of packets received (broadcasted) by client (service provider) to represent the tune-in (broadcast) performance. Each packet size is of 128 bytes and the packet format can be found in Table 2. Each edge weight occupies 4 bytes. For Algorithm 1, we randomly generate 1,000 queries at each Monte Carlo estimation and we only partition the graph into 2 to 16 subgraphs at every partitioning for boosting up the construction time. As a remark, each subgraph/partition (in the HiTi and LTI methods) may span over multiple packets.

TABLE 3
Range of parameter values

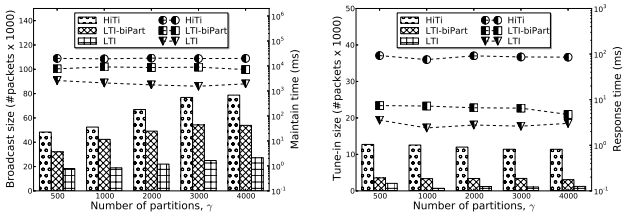
Parameter	Values
Road maps	NYC, **SF**, SJ, OB
Type of shortest paths σ	short, average, long, **mix**
Update ratio δ	1%, 2%, **5%**, 10%, 20%, 30%, 40%, 50%
Ratio of light traffic updates β	50%, 60%, 70%, **80%**, 90%
Weight changes of light traffic	$\pm 20\%$
Weight factor of heavy traffic, ω	2, **5**, 10, 15, 20
Number of HiTi partitions, γ	500, **1000**, 2000, 3000, 4000

8.1 Effectiveness of Optimizations

First, we evaluate the effectiveness of the optimizations proposed in Section 4. The fully optimized LTI is compared against to LTI-biPart (that is constructed by only the graph partitioning technique, described in Section 4.2) and HiTi [21] (which is the most representative model of hierarchical index structures). For fairness, we internally tune the HiTi graph model by varying the number of children subgraphs, and the 8-way regular partitioning is the best HiTi graph model among all testings.

Fig. 12 plots the performance of all three methods as a function of the number of partitions γ on the SF dataset. For the sake of saving space, we plot the costs at service provider (i.e., broadcast size and maintenance time) into one figure and plot the costs at client (i.e., tune-in size and response time) into another figure. The number of packets (left y-axis) is represented by bars, whereas the time (right y-axis) is represented by lines.

LTI is superior to LTI-biPart and HiTi for all four performance factors in Fig.12. As compared to HiTi, its maintenance time and response time are up to 14.7 and 21.1 times faster, respectively. The broadcast size and tune-in size are at least 2.4 and 6.4 times smaller than HiTi. It shows that our fully optimized LTI is very efficient and



(a) service provider (b) client

Fig. 12. Varying number of partitions, γ

performs vastly different from HiTi. In this work, we set γ to 1,000 since it performs the best in both HiTi and LTI. As shown in the figures, all performance factors are not very sensitive to γ which supports our claim in Section 4.2.

8.2 Scalability experiments

TABLE 4
Performance of different methods

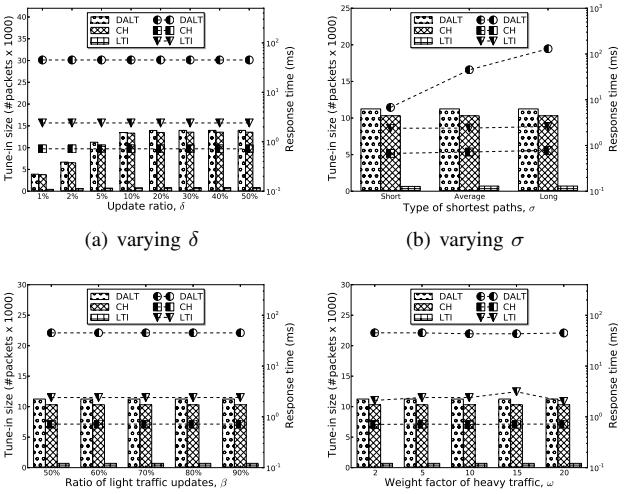
Client side: **T**une-in cost (#packets), **R**esponse time (ms)
Server side: **B**roadcast size (#packets), **M**aintenance time (ms)

City		Method					
		raw transmission model			index transmission model		
		BD	**DALT**	DSPT	**CH**	HiTi	**LTI**
NYC	T	18300.7	18300.7	18300.7	18617.9	26834.5	**704.7**
	R	72.39	54.53	374.93	**2.28**	157.11	4.26
	B	22930	22930	22930	**24802**	124870	30661
	M	-	-	-	15759.6	105451	**5575.4**
SF	T	11149.4	11149.4	11149.4	10212.1	12468.9	**602.8**
	R	89.74	45.03	94.51	**0.72**	75.89	2.40
	B	13863	13863	13863	**13453**	52377	18850
	M	-	-	-	5411.4	19264.4	**2094.9**
SJ	T	1191.2	1191.2	1191.2	624.0	1524.1	**331.1**
	R	5.20	1.98	9.37	**0.08**	10.72	1.37
	B	2525	2525	2525	**1370**	4602	2827
	M	-	-	-	276.3	735.3	**96.6**
OB	T	352.0	352.0	352.0	258.9	516.3	**118.4**
	R	1.11	0.45	31.12	**0.091**	3.66	0.68
	B	604	604	604	**348**	1336	666
	M	-	-	-	104.3	134.6	**16.0**

Next, we compare the discussed solutions on four different road maps. The result is shown in Table 4. Note that all methods on the raw transmission model have the same tune-in size and broadcast size. The only difference is the response time as it represents the local computation time for each client. Apart from BD and DALT, other methods require each client to maintain some index structures locally after receiving the live traffic updates. Thus, their response time is slower[5] than BD and DALT on the raw transmission model. Based on the response time, DALT is the best approach among the methods in this category.

Regarding the index transmission model, HiTi is obviously infeasible for online shortest path computation due to its prohibitive costs. Although CH has slightly better broadcast size and response time[6], we recommend LTI as the best approach due to its light tune-in cost and fast maintenance time. The tune-in size significantly affects the energy consumption and the duration of active mode at client receiver. The tune-in size of LTI is 2.19-26.41 and 2.97-25.97 times smaller than CH and DALT, respectively. Note that the margin becomes more significant on larger maps which demonstrates good scalability of our LTI framework. This is important since reducing the tune-in cost provides opportunity for clients to receive more services simultaneously by selective tuning. In addition, fast maintenance time keeps the freshness of the broadcasted index. The maintenance time of LTI is 2.58-6.5 times faster than CH while the broadcast size of LTI is just 23.6% and 40% larger than CH in NYC and SF, respectively.

In Section 1, we show that the present traffic providers report the traffic very frequently and megabit wireless networks (3G, LTE, Mobile WiMAX, etc.) are available. Therefore, the maintenance time of LTI (i.e., 2 and 5.5 seconds on SF and NYC, respectively) is affordable as compared to the live traffic update frequency and the broadcast overhead of LTI (i.e., around 35% larger than the raw data) is reasonable as the data is transmitted on the megabit wireless networks.



(a) varying δ (b) varying σ

(c) varying β (d) varying ω

Fig. 13. Scalability experiments (client)

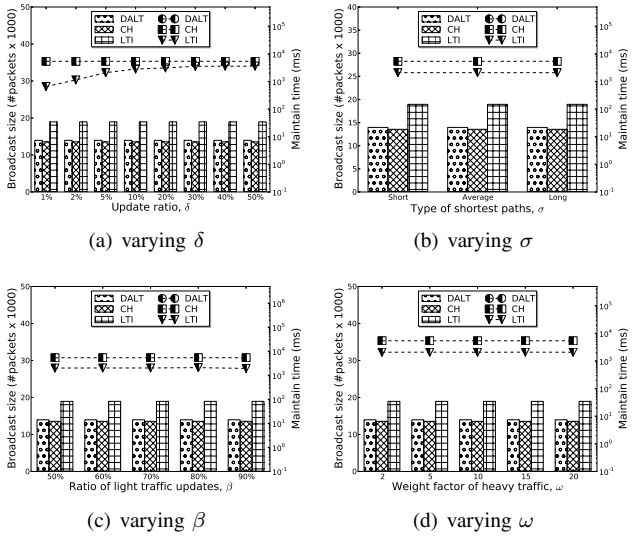
We omit HiTi from the remaining experiments as it is inferior to LTI. The remaining representative methods are: DALT on the raw transmission model, CH and LTI on the index transmission model. We evaluate the performance of these three methods as a function of different system settings in Fig.13. In Fig.13(a), the tune-in size of all methods grow with the update ratio δ, as well, the response time slightly increases since the search graph becomes larger. When $\delta = 20\%$, the number of necessary packets received by clients is 13847.2, 13390.12, and 727.28 for DALT, CH, and LTI respectively. DALT and CH almost receive the entire broadcast packets (i.e., 99.89% and 99.53%, respectively); this conforms with our edge-update probability analysis in Section 2.2. An impressive finding is that the client using LTI only receives 20.63% more packets as compared to $\delta = 10\%$. This shows that LTI is robust as the tune-in size only increases sub-linearly with the update ratio δ.

5. We omit the performance of CH, HiTi, and LTI on the raw transmission model since they are 2 orders of magnitude slower than DALT.

6. We use the codes provided by [30] to construct the CH index which is implemented in C++ instead of Java.

Fig.13(b) shows the tune-in size and response time of the methods on different type of shortest path queries σ. The type of queries is classified based on their length. Again, LTI has the lowest tune-in cost which is at least 16.9 times smaller than DALT and CH among all three types of queries. Note that only DALT is sensitive to various length of queries to the response time since the distance bounds derived from the pre-computed information become looser when the length of queries is longer.

We then study how the methods perform for different traffic circumstances. Fig.13(c) and Fig.13(d) shows the tune-in size and response time of the methods on two traffic update behaviors. For all three methods, the tune-in size and response time are not very sensitive to the ratio of traffic updates β and the weight factor of heavy traffic ω. Again, our LTI outperforms DALT and CH by an order of magnitude in terms of the tune-in size.



(a) varying δ (b) varying σ

(c) varying β (d) varying ω

Fig. 14. Scalability experiments (service provider)

TABLE 5
Broadcast cycle length at default settings

Methods	WCDMA time (s)			HSDPA time (s)		
	Broadcast	Main.	Tune-in	Broadcast	Main.	Tune-in
DALT	7.05	-	5.67	0.97	-	0.78
CH	6.84	5.41	5.19	0.94	5.41	0.71
LTI	9.59	2.01	0.31	1.31	2.01	0.04

Lastly, we demonstrate how the methods perform at service provider. Fig. 14 shows the broadcast size and maintenance time of the methods by varying δ, σ, β, and ω. For all testings, LTI is superior to CH in terms of maintenance time but produces around 40% more packets than CH. A more promising result is that the maintenance time of LTI is no longer sensitive to the update ratio when $\delta > 20\%$. This is because most of BSPTs are necessarily updated when the update ratio is around 20%. The subsequent updates (>20%) are more likely some incremental work in updating the BSPTs (i.e., traversing few more edges by the Dijkstra like algorithm) so that it becomes less sensitive to δ. To express the comparison in absolute terms, we show the time it takes to broadcast over a 1.92Mbps (WCDMA) and a 14Mbps (HSDPA) channel in Table 5, which are typical transmission rates in 3G networks and 3.5G networks. LTI takes 11.6s and 3.32s to complete a maintenance and broadcast cycle at WCDMA and HSDPA, respectively; while CH takes 12.25s and 6.35s to complete the same cycle, respectively. In addition, DALT and CH require the clients to tune-in the broadcast channel for ∼5s and ∼0.7s over WCDMA and HSDPA, respectively, which significantly affects the number of simultaneous services in the wireless broadcast environments. Although DALT does not bother any maintenance cost at service provider, the tune-in cost and response time of DALT makes it infeasible on the live traffic circumstance.

9 Conclusion

In this paper we studied online shortest path computation; the shortest path result is computed/updated based on the live traffic circumstances. We carefully analyze the existing work and discuss their inapplicability to the problem (due to their prohibitive maintenance time and large transmission overhead). To address the problem, we suggest a promising architecture that broadcasts the index on the air. We first identify an important feature of the hierarchical index structure which enables us to compute shortest path on a small portion of index. This important feature is thoroughly used in our solution, LTI. Our experiments confirm that LTI is a Pareto optimal solution in terms of four performance factors for online shortest path computation.

In the future, we will extend our solution on time dependent networks. This is a very interesting topic since the decision of a shortest path depends not only on current traffic data but also based on the predicted traffic circumstances.

Acknowledgment

This work was partially supported by SRG007-FST11-LHU and MYRG109(Y1-L3)-FST12-ULH from UMAC Research Committee and FDCT/106/2012/A3 from FDCT. Man Lung Yiu was supported by ICRG grant A-PL99 from the Hong Kong Polytechnic University.

References

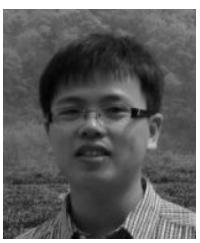
[1] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *ALENEX*, 2007.
[2] P. Sanders and D. Schultes, "Engineering highway hierarchies," in *ESA*, 2006, pp. 804–816.
[3] G. Dantzig, *Linear programming and extensions*, ser. Rand Corporation Research Study. Princeton, NJ: Princeton Univ. Press, 1963.
[4] R. J. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks," in *ALENEX/ANALC*, 2004, pp. 100–111.
[5] B. Jiang, "I/o-efficiency of shortest path algorithms: An analysis," in *ICDE*, 1992, pp. 12–19.
[6] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *ESA*, 2005, pp. 568–579.
[7] D. Schultes and P. Sanders, "Dynamic highway-node routing," in *WEA*, 2007, pp. 66–79.
[8] F. Zhan and C. Noon, "Shortest path algorithms: an evaluation using real road networks," *Transportation Science*, vol. 32, no. 1, pp. 65–73, 1998.

[9] "Google Maps," http://maps.google.com.
[10] "NAVTEQ Maps and Traffic," http://www.navteq.com.
[11] "INRIX inc. Traffic Information Provider," http://www.inrix.com.
[12] "TomTom NV," http://www.tomtom.com.
[13] "Cisco visual networking index: Global mobile data traffic forecast update, 2010-2015," 2011.
[14] D. Stewart, "Economics of wireless means data prices bound to rise," The Global and Mail, 2011.
[15] W.-S. Ku, R. Zimmermann, and H. Wang, "Location-based spatial query processing in wireless broadcast environments," *IEEE Trans. Mob. Comput.*, vol. 7, no. 6, pp. 778–791, 2008.
[16] N. Malviya, S. Madden, and A. Bhattacharya, "A continuous query system for dynamic route planning," in *ICDE*, 2011, pp. 792–803.
[17] G. Kellaris and K. Mouratidis, "Shortest path computation on air indexes," *PVLDB*, vol. 3, no. 1, pp. 747–757, 2010.
[18] Y. Jing, C. Chen, W. Sun, B. Zheng, L. Liu, and C. Tu, "Energy-efficient shortest path query processing on air," in *GIS*, 2011, pp. 393–396.
[19] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina, "Proximity search in databases," in *VLDB*, 1998, pp. 26–37.
[20] N. Jing, Y.-W. Huang, and E. A. Rundensteiner, "Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation," *IEEE TKDE*, vol. 10, no. 3, pp. 409–432, 1998.
[21] S. Jung and S. Pramanik, "An efficient path computation model for hierarchically structured topographical road maps," *IEEE TKDE*, vol. 14, no. 5, pp. 1029–1046, 2002.
[22] E. P. F. Chan and Y. Yang, "Shortest path tree computation in dynamic graphs," *IEEE Trans. Computers*, vol. 58, no. 4, pp. 541–557, 2009.
[23] T. Imielinski, S. Viswanathan, and B. R. Badrinath, "Data on air: Organization and access," *IEEE TKDE*, vol. 9, no. 3, pp. 353–372, 1997.
[24] J. X. Yu and K.-L. Tan, "An analysis of selective tuning schemes for nonuniform broadcast," *Data Knowl. Eng.*, vol. 22, no. 3, pp. 319–344, 1997.
[25] A. V. Goldberg and R. F. F. Werneck, "Computing point-to-point shortest paths from external memory," in *ALENEX/ANALCO*, 2005, pp. 26–40.
[26] M. Hilger, E. Köhler, R. Möhring, and H. Schilling, "Fast point-to-point shortest path computations with arc-flags," *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, vol. 74, pp. 41–72, 2009.
[27] A. V. Goldberg and C. Harrelson, "Computing the shortest path: search meets graph theory," in *SODA*, 2005, pp. 156–165.
[28] D. Delling and D. Wagner, "Landmark-based routing in dynamic graphs," in *WEA*, 2007, pp. 52–65.
[29] G. D'Angelo, D. Frigioni, and C. Vitale, "Dynamic arc-flags in road networks," in *SEA*, 2011, pp. 88–99.
[30] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *WEA*, 2008, pp. 319–333.
[31] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm," *ACM Journal of Experimental Algorithmics*, vol. 15, 2010.
[32] 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*, vol. 1, no. 4, pp. 709–736, 2008.
[33] F. Wei, "Tedi: efficient shortest path query answering on graphs," in *SIGMOD Conference*, 2010, pp. 99–110.
[34] J. Sankaranarayanan and H. Samet, "Query processing using distance oracles for spatial networks," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 8, pp. 1158–1175, 2010.
[35] J. Sankaranarayanan, H. Samet, and H. Alborzi, "Path oracles for spatial networks," *PVLDB*, vol. 2, no. 1, pp. 1210–1221, 2009.
[36] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *SIGMOD Conference*, 2008, pp. 43–54.
[37] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance indexing on road networks," in *VLDB*, 2006, pp. 894–905.
[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*, vol. 5, no. 5, pp. 406–417, 2012.
[39] R. Bauer and D. Delling, "Sharc: Fast and robust unidirectional routing," in *ALENEX*, 2008, pp. 13–26.
[40] T. Bühler and M. Hein, "Spectral clustering based on the graph - laplacian," in *ICML*, 2009, p. 11.
[41] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, 2000.
[42] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques, Second Edition*, 2nd ed. Morgan Kaufmann, 2006.
[43] "9th DIMACS Implementation Challenge - Shortest Paths," http://www.dis.uniroma1.it/challenge9/download.shtml.
[44] "Network-based Generator of Moving Objects," http://iapg.jade-hs.de/personen/brinkhoff/generator/.

Leong Hou U is now an Assistant Professor at the University of Macau. His research interest includes spatial and spatio-temporal databases, advanced query processing, web data management, information retrieval, data mining and optimization problems.

Hongjun Zhao graduated from University of Macau in 2012 under the supervision of Prof. Zhiguo Gong and Dr. Leong Hou U. He currently works as a senior software engineer in 30GROUP, China Electronics Technology Group Corporation (CETC), Chengdu, SC.

Man Lung Yiu is now an assistant professor in the Department of Computing, Hong Kong Polytechnic University. Prior to his current post, he worked at Aalborg University for three years starting in the Fall of 2006. His research focuses on the management of complex data, in particular query processing topics on spatiotemporal data and multidimensional data.



Yuhong Li is a PhD candidate at the Department of Computer and Information Science, University of Macau, under the supervision of Prof. Zhiguo Gong and Dr. Leong Hou U. His current research focuses on query processing on temporal and spatial data and high performance parallel computing.

Zhiguo Gong is currently an Associate Professor in the Department of Computer and Information Science, University of Macau, Macau, China. His research interests include databases, Web information retrieval, and Web mining.