Online Shortestpath Computation using LTI-TR on Networks

P. Manju Latha
M.Tech Cse(Student)
Sri Vasavi Engineering College
Pedatadepalli,Tadepalligudem
Ap,India

D. Anjani Suputridevi
Asst.Professor
CSE Dept
Sri Vasavi Engineering College
Pedatadepalli, Tadepalligudem

***Abstract*:- Shortest path computation is one of the most common queries in location-based services that involve transportation networks. Motivated by appreciable challenges faced in the mobile network industry, we propose adopting the wireless broadcast model for such location-dependent applications. In this model the data are continuously transmitted on the air, while clients listen to the broadcast and process their queries locally. Although spatial problems have been considered in this environment, there exists no study on shortest path queries in road networks. In presented approach server will collect live traffic information and then announce them over wireless network. With this approach any number of clients can be added. This new approach called live traffic index-time reliant (LTI-TR) enables drivers to update their shortest path result by receiving only a small fraction of the index. The existing systems were infeasible to solve the problem due to their prohibitive maintenance time and large transmission overhead. LTI-TR is a novel solution for Online Shortest Path Computation on Time Reliant Network**

1. INTRODUCTION:

shortest path computation is an important function in modern car navigation systems and has been extensively studied in . This function helps a driver to figure out the best route from his cur-rent 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. Un-fortunately, 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).Nowadays, several online services provide live traffic data (by analyzing collected data from road sensors, traffic cameras, and crowdsourcing techniques), such as GoogleMap, Navteq , INRIX Traffic Information Provider , and TomTom NV , etc. These systems can calculate the snapshot shortest path queries based on current live traffic data; however, they do not report route 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 limitation appreciable in terms of network band-width and server loading. According to the Cisco Visual Networking Index forecast, global mobile traffic in 2010 was 237 pet bytes 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, the world's cellular networks need to provide 100 times the capacity in 2015 when compared to the networks in 2011. Furthermore, live traffic are updated frequently as these data may be collected by using crowd sourcing 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.raise the same concern in their work which processes spatial queries in wireless broadcast environments based on Euclidean distance metric.

Malviya et al. developed a client-server system for continuous monitoring of registered shortest path queries. For each registered query (s; t), the server first precalculate K different candidate paths from s to t. Then, the server periodically updates the travel times on these K paths based on the latest

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

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

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 transmit 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 appreciable, 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 transmit an air index over the wireless network (called index transmission model). 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 constitutes a set of pairwise minimum and maximum traveling costs between every two sub-partitions of the road map. However, these methods only solve appreciable the issue for the number of clients but not for the amount of live traffic updates. As reported in , 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 them 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 . To the best of our knowledge, this is the first work to give a through cost analysis on the hierarchical index techniques and apply stochastic process to optimize the index hierarchical structure.

LTI selectively fetches data in wireless transmit environments, which significantly reduce the tune-in cost. LTI efficiently maintains the index for live traffic circumstances by incorporating Dynamic Shortest Path Tree (DSPT) into hierarchial index techniques. In addition, a bounded version of DSPT is proposed to further reduce the broadcast overhead.

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.

2.RELATED WORKS

"Engineering Highway Hierarchies,"AUTHORS: P. Sanders and D. Schultes

Highway hierarchies exploit hierarchical properties inherent in real-world road networks to allow fast and exact point-to-point shortest-path queries. A fast preprocessing routine iteratively performs two steps: First, it removes edges that only appear on shortest paths close to source or target; second, it identifies low-degree nodes and bypasses them by introducing shortcut edges. The resulting hierarchy of highway networks is then used in a Dijkstra-like bidirectional query algorithm to considerably reduce the search space size without losing exactness. The crucial fact is that 'far away' from source and target it is sufficient to consider only high-level edges. Experiments with road networks for a continent show that using a preprocessing time of around 15 min, one can achieve a query time of around 1ms on a 2.0GHz AMD Opteron. Highway hierarchies can be combined with goal-directed search, they can be extended to answer many-to-many queries, and they can be used as a basis for other speed-up techniques (e.g., for transit-node routing and highway-node routing).

"Highway Hierarchies Hasten Exact Shortest Path Queries,"Authors: P. Sanders and D. Schultes

We present a new speedup technique for route planning that exploits the hierarchy inherent in real world road networks. Our algorithm preprocesses the eight digit number of nodes needed for maps of the USA or Western Europe in a few hours using linear space. Shortest (i.e. fastest) path queries then take around eight milliseconds to produce exact shortest paths. This is about 2 000 times faster than using Dijkstra algorithm.

"Dynamic Highway-Node Routing," Authors: D. Schultes and P. Sanders

We introduce a dynamic technique for fast route planning in large road networks. For the first time, it is possible to handle the practically relevant scenarios that arise in present-day navigation systems: When an edge weight changes (e.g., due to a traffic jam), we can update the preprocessed information in 2-40ms allowing subsequent fast queries in about one millisecond on average. When we want to perform only a single query, we can skip the comparatively expensive update step and directly perform a prudent query that automatically takes the changed situation into account. If the overall cost function changes (e.g., due to a different vehicle type), recomputing the preprocessed information takes typically less than two minutes. The foundation of our dynamic method is a new static approach that generalizes and combines several previous speedup techniques. It has outstandingly low memory requirements of only a few bytes per node.

"Shortest Path Algorithms: An Evaluation Using Real Road Networks, Authors: F. Zhan and C. Noon

The classic problem of finding the shortest path over a network has been the target of many research efforts over the years. These research efforts have resulted in a number of different algorithms and a considerable amount of empirical findings with respect to performance. Unfortunately, prior research does not provide a clear direction for choosing an algorithm when one faces the problem of computing shortest paths on real road networks. Most of the computational testing on shortest path algorithms has been based on randomly generated networks, which may not have the characteristics of real road networks. In this paper, we provide an objective evaluation of 15 shortest path algorithms using a variety of real road networks. Based on the evaluation, a set of recommended algorithms for computing shortest paths on real road networks is identified. This evaluation should be particularly useful to researchers and practitioners in operations research, management science, transportation, and Geographic Information Systems. The computation of shortest paths is an important task in many network and transportation related analyses. The development, computational testing, and efficient implementation of shortest path algorithms have remained important research topics within related disciplines such as operations.

EXISTING SYSTEM:

III. LTI

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 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).

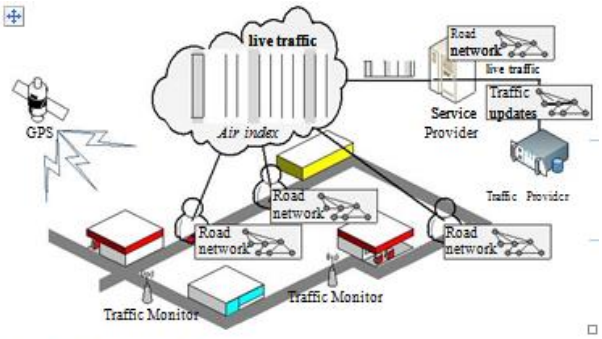


Fig. 3. LTI System Overview

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).

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

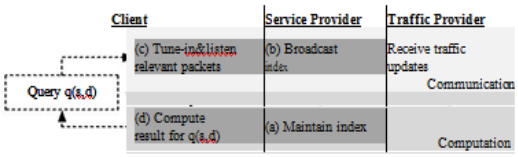


Fig. 4. Components in LTI

server side so that the traffic information is no longer live. This can reduce the maintenance time spent at component a.

(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 transmit cycle, but

also makes clients listen fewer packets in the transmit 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.

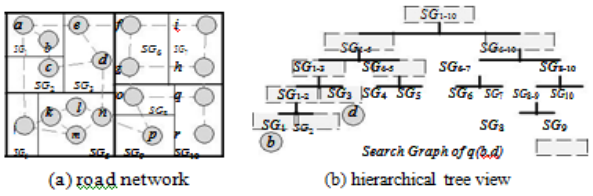
Inspired by these properties, LTI has relatively short tune-in cost (at client side), fast query response time (at client side), small broadcast size (at server side), and light index maintenance time (at server side) for OSP. As discussed in Section 2.2, the hierarchical index structures enable clients 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.

LTI CONSTRUCTION

Analysis of Hierarchical Index Structures

Hierarchical index structures (e.g., HiTi, HEPV, and Hub Indexing, TEDI) 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 sub graphs SG_i and organizes SG_i in a hierarchical fashion (i.e., tree). In Fig. 5, we illustrate a graph being partitioned

into 10 sub graphs (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_{SG1} = {a; b} and one edge E_{SG1} = {(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 $\acute{\Gamma}_{SG1\text{-}2}$ = {(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_{SGi} 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 sub graph. In Fig. 5, SG_5 has two border nodes[2] k and m so that SG_5 keeps a shortcut Δ_{SG5} = {(k; m)} and its corresponding weight.

To answer a shortest path query q(s; 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 Hi Ti 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. 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., $_{SG6\text{-}10}$ = f(f; p)g).

Index Construction

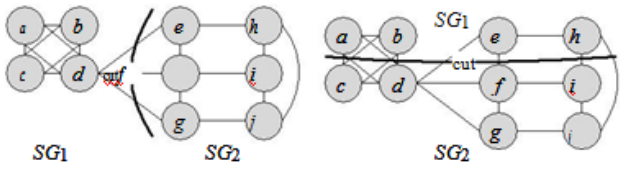
The above discussion shows that it is hard to find a hierarchical index structure I that achieves all optimization objectives. One possible solution is to relax the optimization objectives which makes them be the tuned factors of the problem. While the overhead of pre-computed information (O2) and the 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 $_{SGi}$ 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 sub graphs being created is that is a tuned parameter for controlling the number of sub graphs in this work. According to Eq. 2, minimizing the size of $_{SGi}$ is likely to minimize the overhead of I.

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

Stochastic based index construction. Intuitively, the size search graphs G^q (i.e., O3) is highly relevant to the index hierarchical structure. As a motivating example, the number 4 as the objective function so that we can heuristically reduce the number of border nodes. To construct an index, we recursively cut the sub graphs until we have enough partitions (i.e., the leaf 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.

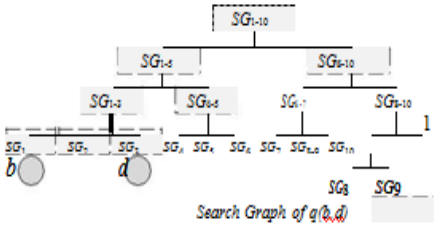


Fig. 7. Effect of hierarchical structure

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 Q or a universal query set U) and (2) the index statistics (e.g., shortcuts) are changed on different index hierarchical structures (which is necessarily recalculated based on the structure). Typically, these combinational 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, At every partitioning, we attempt to find the best structure for potential queries by the stochastic process. 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 .

In summary, a small may lead the index having large leaf entries while a large may lead the index having large number of index nodes, where these settings may degrade the query performance. Fortunately, is not a very sensitive parameter (cf. the studies in other hierarchical indexing techniques and our experiments), which can be decided by experimental studies in practice.

PROPOSED SYSTEM:

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.

Transmitting 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 "program"), 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 is one of the best transmitting schemes. Table 1 shows an example transmitting 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 transmitting schedule of all packets. In this example, the variables i and n in each header represent the last transmitted item and the total number of items. The server periodically transmits a sequence of packets (called as a transmit cycle)

We use a concrete example to demonstrate how a client receives her data from the transmit channel. Suppose that a client wishes to query for the data object o_5. First, the client tunes in the transmitt 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 transmitting 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 tuned time and the waiting time at the client side. In this transmitting 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. suggests to set m to the square root of the ratio of the data size to the index size.

LTI on Air

To transmit 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 transmit the latest weight of edges periodically.

Table 2 shows the format of a header/data packet in our model. id is the offset of the packet in the present transmit 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

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				...							
...	...															

transmit 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.

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 sub graphs. For instance, the edge weight of sub graph SG_1 are stored in the 2nd packet of a transmit 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 3rd 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 transmit cycle.

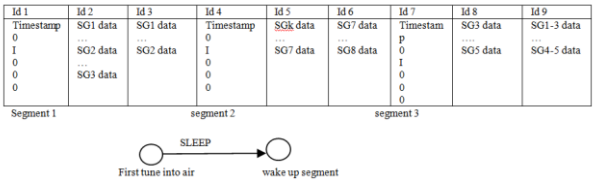


Fig 7:Receiving LTI from air index

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 transmit cycle. The navigation system identifies the sub graphs 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 transmitted(i.e., the 3rd packet of segment 3).

LTI Maintenance

In order to keep the freshness of the transmit 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 transmitted to the clients.

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

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 $_{SGi}$ are necessarily re-computed as the weight of other edges (i.e., E_{SGi} [$_{SGi}$) are directly derived from the updates. To boost the shortcut edge maintenance, we incorporates dynamic shortest path tree technique (DSPT) into the hierarchical index structures and reduce the overhead of DSPT by a bounded version (BSPT).

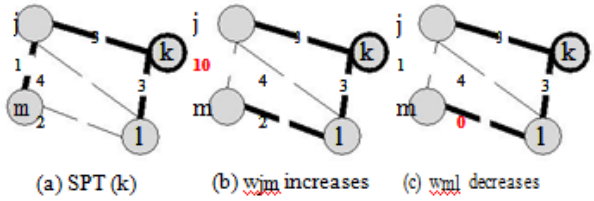


Fig. 10. Shortest path tree maintenance

Given a graph G = (V; E), a shortest path tree (SPT) rooted at a vertex r 2 V , denoted as SPT (r), is a tree with root r, and 8v 2 V fig, SPT (r) contains a shortest path from 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 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 -> l-> 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).

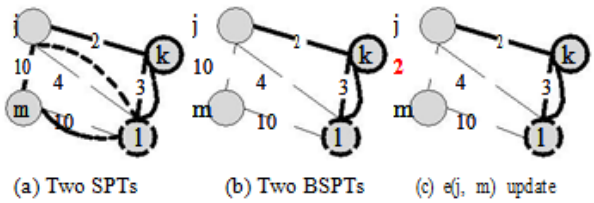


Fig. 11. Shortest path trees and updates

To keep the freshness of LTI, every sub graph is required to maintain its corresponding shortcut edges $_{SGi}$ according to live traffic circumstances. The weight of these

shortcuts can be maintained by the corresponding shortest path tree from each border node B_{SGi} Pruning ability of BSPT. The pruning ability of BSPT is highly relevant to the border node selection in each sub graph. In the worst case, BSPT performs as the same as a native 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).

Algorithm 2 : Client Algorithm

Algorithm Client(I:LTI; s:source; t:destination)
1: generate G^q from I based on s and d
2: listen to the channel for a header segment . Section
3: read the header segment 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 s to t) on G^q . Section 4.1

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
: update the sub graphs $\{SG_i\}$. Section 6
: broadcast the sub graphs $\{SG_i\}$. Section 5.2

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 s to a destination t. First, the client generates a search graph G^q based on s (i.e., current location) and 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 sub graphs $|SG_i|$ such that they are ready for transmitting. We develop an effective graph partitioning algorithm for minimizing the total size of sub graphs and study a combinatorial optimization for reducing the search space of shortest path queries in Section 4.2. In each transmit cycle, the server first collects live traffic updates from the traffic provider, updates the sub graphs $|SG_i|$ (discussed in Section 6), and eventually transmits them.

IV 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 transmits 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.we extend our solution on time reliant 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.

REFERENCES

[1] "Engineering Highway Hierarchies," AUTHORS: P. Sanders and D. Schultes
[2] "Highway Hierarchies Hasten Exact Shortest Path Queries,"AUTHORS: P. Sanders and D. Schultes
[3] "Dynamic Highway-Node Routing," AUTHORS: D. Schultes and P. Sanders
[4] "Shortest Path Algorithms: An Evaluation Using Real Road Networks, AUTHORS: F. Zhan and C. Noon
[5] "Location-Based Spatial Query Processing in Wireless Broadcast Environments, AUTHORS: W.-S. Ku, R. Zimmermann, and H. Wang