Efficient Verification of Shortest Path Search via Authenticated Hints

Man Lung Yiu [#1], Yimin Lin [*2], Kyriakos Mouratidis [*3]

[#]*Department of Computing, Hong Kong Polytechnic University*
Hung Hom, Hong Kong
[1]csmlyiu@comp.polyu.edu.hk

[*]*School of Information Systems, Singapore Management University*
80 Stamford Road, Singapore 178902
[2]yimin.lin.2007@phdis.smu.edu.sg
[3]kyriakos@smu.edu.sg

***Abstract*— Shortest path search in transportation networks is unarguably one of the most important online search services nowadays (e.g., Google Maps, MapQuest, etc), with applications spanning logistics, spatial optimization, or everyday driving decisions. Often times, the owner of the road network data (e.g., a transport authority) provides its database to third-party query services, which are responsible for answering shortest path queries posed by their clients. The issue arising here is that a query service might be returning sub-optimal paths either purposely (in order to serve its own purposes like computational savings or commercial reasons) or because it has been compromised by Internet attackers who falsify the results. Therefore, for the above applications to succeed, it is essential that each reported path is accompanied by a proof, which allows clients to verify the path's correctness.**

This is the first study on shortest path verification in outsourced network databases. We propose the concept of *authenticated hints*, which is used to reduce the size of the proofs. We develop several authentication techniques and quantify their tradeoffs with respect to offline construction cost and proof size. Experiments on real road networks demonstrate that our solutions are indeed efficient and lead to compact query proofs.

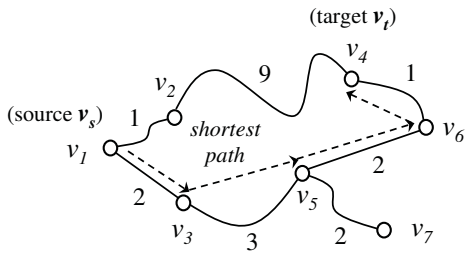


Fig. 1. Shortest path example

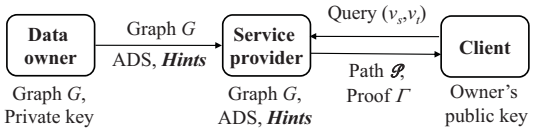


Fig. 2. Outsourcing architecture

I. Introduction

A road network is modeled as a graph G, whose nodes represent road junctions and whose edges correspond to road segments. The weight of an edge typically reflects the travel distance, the driving time, or the toll fee of the respective road segment. Given a source node v_s and a target node v_t on such a graph G, the *shortest path query* returns the path between v_s and v_t, along which the sum of edge weights is minimal. Figure 1 shows an example of a graph where the weight of each edge corresponds to its length and is indicated by a number next to the edge. Suppose that the source node is v_1 and the target node is v_4. The shortest path between them is $v_1 \rightarrow v_3 \rightarrow v_5 \rightarrow v_6 \rightarrow v_4$ with total cost (i.e., total length) $2 + 3 + 2 + 1 = 8$.

Shortest path search is crucial to a wide range of applications. For example, taxi drivers want to find shortest paths to the target locations specified by passengers. Logistics companies need to find shortest paths (in a transportation network) for quickly delivering packages from senders to receivers. Besides the above daily business operations, shortest path computation also finds many applications for personal use. Prior to a hiking trip or a bike tour, a group of people wish to find the shortest path from their gathering point to a particular target location.

Usually, the road network data belong to and are maintained by a commonly trusted government or transport authority, referred to as the *data owner*. On the other hand, the aforementioned business and personal needs are attended to by third-party online *service providers* (such as Google Maps, MapQuest, etc), who download the network information from the owner and use it to compute shortest paths on demand. The issue here is that returned paths may be incorrect for a variety of reasons. Firstly, the service provider may be returning sub-optimal paths for profit purposes (e.g., favorably selecting paths that pass by the gas stations of a certain chain), or in order to save computational resources (e.g., employing directional heuristics that report fast but approximate results). Furthermore, even if the provider is legitimate, its servers may be infiltrated by attackers via the Internet; recent studies show that multi-step intrusions into online servers are becoming increasingly common [1]. Should an attacker take over a

server, he may be falsifying the results to serve his own purposes or to simply be a nuisance to the clients and the service provider (e.g., sending clients to non-existing roads or damaging the service provider's reputation). This risk becomes more severe as the service providers are gradually using the low-cost cloud computing environment, where many security concerns remain unresolved.

The aforementioned reasons necessitate the development of mechanisms that will allow clients to verify the correctness of the returned paths. Specifically, a client should be able to verify that: (i) the returned path $\mathcal{P}$ is *possible*, i.e., $\mathcal{P}$ is an existing path in the original graph G of the data owner, and (ii) $\mathcal{P}$ is the *shortest* among all possible paths in G between the source and target nodes. To enable this, we take the following approach. The data owner constructs an authenticated data structure (ADS) on top of the network data and uploads them together to the provider. The provider returns parts of this authentication information to the clients in the form of a correctness *proof*, along with each shortest path reported. This process is depicted in Figure 2.

Interestingly, the definition of the shortest path itself makes its verification a challenging problem. A naïve solution would be to generate all possible paths between the source node and the target node, and show that none of them can be shorter than the server's reported path. Unfortunately, this approach incurs a prohibitively high communication overhead between the service provider and the client. In this paper, our challenge is to minimize the size of the proof, while it is still sufficient for verification. To achieve this, we propose that the data owner pre-computes and authenticates some auxiliary information called *authenticated hints* from the network. The hints are collected during query processing and inserted into the proof. We develop a spectrum of authenticated hints with different tradeoffs in terms of offline construction cost and proof size.

To the best of our knowledge, this is the first study on the shortest path verification problem. Most of the existing work on outsourced database authentication focuses on verifying the results of range queries in relational databases [2], [3], [4], [5], [6], and cannot be applied to our problem. Spatial verification methods, on the other hand, explicitly assume the vector space model [7], [5]. Goodrich et al. [8] do consider graph data, but study only the verification of connectivity queries. This is a different problem from ours; all the nodes are connected in the network of Figure 1, but no shortest paths may be deduced from this information.

The rest of the paper is organized as follows. Section II provides a necessary background and surveys related work. Section III formalizes the addressed problem and presents our general framework. Section IV describes two basic solutions: one using no pre-computation, and one using full pre-computation. Section V designs methods that reduce the amount of pre-computation required (for scalability with network size), without sacrificing the conciseness of the proof. Section VI experimentally evaluates our solutions, while Section VII concludes the paper with directions for future work.

II. Related Work

In this section we review the cryptographic primitives that underlie our approach, and survey related work in the areas of database authentication and shortest path computation.

A. Cryptographic Primitives

One-way hash function: Such a function H maps a message m of arbitrary length into a fixed-length output $H(m)$, which is called *message digest*. It is fast to compute $H(m)$, but it is computationally infeasible to find a message that maps to a given digest. SHA-1 [9] is a commonly used one-way hash function.

Cryptographic signature: A message owner creates a pair of a private and a public key; the former is kept secret and the latter is publicly distributed. A message can be signed by its owner using his private key. The integrity and ownership of the message can then be verified by any recipient using the signature and the owner's public key. The most widely used public-key signature algorithm is RSA [10].

Merkle Hash Tree (MHT): The MHT [11] is a structure used for set membership verification. It is a binary tree, where each node is the digest of the concatenation of its two children; the leaf level contains the hashes of the set's elements (messages). The MHT root is signed by the set's owner. The integrity and ownership of an element (message) can be verified using the element itself and a *proof*. The proof contains the signed root and the sibling nodes (hashes) of the path from the root down to the element. The message is deemed authentic if its digest combined with the proof hashes leads to an MHT root that matches the owner's root signature. The MHT idea can be applied to arbitrary DAGs [12].

B. Query Result Verification

The outsourced database model [13] includes three entities: *data owner*, *service provider* and *clients*. The data owner outsources its data to the service provider, who is responsible for answering the clients' queries (e.g., equality and range selection on a specific search key). Along with the answer to each query, the provider returns a proof that allows the client to verify that the answer includes those and only those data tuples that the owner's database would return, and that these tuples have not been tampered with. There are two general approaches to achieve this:

Signature chaining [14], [15], [16]: The owner first sorts the data tuples. For each tuple, he generates a signature over its contents and the tuples immediately to its left and to its right. The proof for an answer contains the signature of every returned tuple. The chaining achieved by signing consecutive triples of tuples ensures the completeness of the result and the authenticity of each returned tuple. To reduce the size of the proof, the service provider may aggregate all signatures into one [17].

MHT authentication: An MHT is embedded into the data index (typically a B^+-tree). The MHT structure follows that of the data index. The proof for a query includes the signed MHT root, two boundary tuples (i.e., the tuples immediately to the

left and right of the query range), and the left and right sibling digests of the left and right boundary tuple respectively. Verification at the client side includes reconstructing the MHT root digest (by combining the hashes of the result tuples with those in the proof) and checking whether it matches the owner's root signature. [4] includes an efficient implementation of an MHT-authenticated B^+-tree, and demonstrates its superiority over signature chaining. [7] further boosts performance, by separating the authentication information from the data index, in order to increase B^+-tree fanout and use a binary MHT (so that proof size is near-minimal). MHT techniques have been used to verify spatial [7], [5], continuous [18], [19], XML [20], and text search [21] queries.

The only authentication work on graph queries belongs to the MHT category; [8] considers connectivity queries in graphs, i.e., it verifies whether two nodes are connected in the graph and if so, it can additionally return *a path* (not the shortest) between them. The data owner computes a spanning tree for each connected component in the graph, and authenticates the resulting "forest" (set of spanning trees). Two nodes are connected if they are in the same spanning tree, and a path can always be found between them in the tree. This method is inapplicable to shortest path queries, because paths in the spanning tree are in general not the shortest, and even if they happen to be, no proof can be produced for this.

C. Shortest Path Computation

The shortest path between a source node v_s and a target node v_t in a graph G is the path between v_s and v_t with the minimum sum of edge weights along it. Below we review shortest path computation schemes, categorizing them based on the amount of pre-computation required.

No pre-computation: A general and commonly used method is Dijkstra's algorithm [22]. Initially, nodes adjacent to v_s are pushed into a min-heap with their graph distance from v_s (i.e., the weights of the corresponding edges) as sorting key. The top node v in the heap is iteratively popped, and *expanded*; i.e., its adjacent nodes v' that have not been encountered before are en-heaped with key equal to the key of v plus the weight of edge (v, v'). The process stops when v_t is popped; the shortest path is formed by tracing backwards the expansions that lead to v_t. The A* algorithm [23] requires that a lower bound $LB(v, v_t)$ can be computed for the graph distance between an encountered node v and the target node v_t. The only difference from Dijkstra's algorithm is that the key of each en-heaped node v is increased by $LB(v, v_t)$. This leads to a smaller search space and an earlier termination. *Bi-directional search* [24] is a paradigm which can be integrated with other methods. The basic idea is to initiate two concurrent graph expansions at the source and at the target node. The shortest path is computed when the two expansions meet.

Partial pre-computation: Partial pre-computation methods accelerate ad-hoc shortest path queries by pre-computing and materializing some shortest path information. In *arc-flag* [25] the graph nodes are first partitioned. Every edge is assigned a bit-vector (*flag*), where each bit corresponds to a partition. In the flag of edge (v_i, v_j), the bit for a partition is set to 1 only if there is at least one node v in that partition where the shortest path from v_i to v passes through this edge. Given a target node v_t, search only considers edges whose bit for v_t's partition is 1.

Landmark A* [26], [27] chooses some landmarks (anchor nodes) and pre-computes for each node v the graph distance from v to all landmarks. The distances to the landmarks form a distance vector. Given the distance vectors of two nodes, a lower bound can be derived for their graph distance. This bound is then used by A* algorithm to guide the search.

In HiTi [28] the graph is partitioned using a (Euclidean) grid of cells. The resulting subgraphs are recursively grouped into higher level subgraphs, thus forming a subgraph tree. All distances between the subgraph boundary nodes are computed and stored in the upper level. A shortest path is computed by A* algorithm, starting from the lowest level cell where the source node resides, ascending to the root of the tree and then descending the tree until the target node is reached. HEPV [29] works similarly to HiTi.

Other schemes include [30] and [31], two embedding methods, which materialize node distances from selected sets of nodes and edges respectively. The former supports only approximate answers. The latter is theoretical in nature, requiring integer weights and several graph properties to hold.

Full pre-computation: Full pre-computation schemes materialize the shortest paths between any two nodes in the graph. The *shortest path quadtree* scheme [32] stores for each node v a colored-quadtree, built on the Euclidean coordinates of the other graph nodes. The nodes v' for which the shortest path (from v) passes through the same incident edge of v are assigned the same color. In *distance index* [33] the graph distance spectrum is partitioned into a number of categories. Each node v stores the distance category for each other node v' in the graph, along with the next node information on the shortest path from v to v'. Full pre-computation methods are only applicable to small graphs, due to their high pre-computation cost and storage overhead.

III. Problem Setting and Framework

We first describe the problem setting, and then we develop a subgraph authentication technique, which will be employed as a functional component in subsequent sections. Table I summarizes the notation used in the paper.

TABLE I
SUMMARY OF NOTATION

Symbol	Description
$H(\cdot)$	Secure hash function
v	A node in the node set V
(v_i, v_j)	An edge in the edge set E
$W(v_i, v_j)$	Weight of edge (v_i, v_j)
$dist(v_i, v_j)$	Shortest path distance from v_i to v_j
$\Phi(v)$	Extended-tuple of node v
v_s, v_t	The source and target nodes respectively
Γ_T, Γ_S	The integrity and shortest path proofs respectively

A. Problem Setting

Architectural Framework

We assume the traditional three-party model (illustrated in Figure 2) that comprises a data owner, a service provider, and the clients. Let $G = (V, E, W)$ be a weighted graph, where V is the set of nodes, E is the set of edges, and W is a function that maps each edge $(v_i, v_j) \in E$ to a non-negative weight. We focus on general road networks in which the edge weights could represent measures other than Euclidean distances (e.g., they could be toll fees, driving times, etc). Therefore, Euclidean distance lower bounds are inapplicable to our target networks.

The data owner possesses a graph G and generates a public-private key pair. He then builds an authenticated data structure (ADS) on G. Next, he pre-computes some auxiliary attributes on G, called *authenticated hints*, which are utilized for accelerating the verification process discussed later. In the last step, he signs on the ADS, and then sends the graph G, the ADS, and the authenticated hints to the service provider.

At query time, the client submits a shortest path query (v_s, v_t) to the service provider, where v_s and v_t denote the source and target nodes respectively. The service provider runs Algorithm 1 to compute the shortest path and the proof. First, it applies the shortest path algorithm $algo_{sp}$ of its choice to compute a result path $\mathcal{P}_{rslt} : v_{z_0}, v_{z_1}, \cdots, v_{z_k}$. The path distance of $\mathcal{P}_{rslt}$ is defined as:

$$dist(\mathcal{P}) = \sum_{i \in [1,k]} W(v_{z_{i-1}}, v_{z_i})$$

Next, the service provider examines the authenticated hints and ADS for generating the query proofs Γ_S and Γ_T (details are elaborated shortly). Upon receiving the result path $\mathcal{P}_{rslt}$ and the proofs Γ_S, Γ_T, the client checks whether the returned path $\mathcal{P}_{rslt}$ satisfies both proofs.

Algorithm 1 Service Provider Task

algorithm Service_Provider_Task(Source v_s, Target v_t)
1: $\mathcal{P}_{rslt}$:= Shortest_Path_Search($algo_{sp}, v_s, v_t$);
2: Γ_S := SP_Proof_Generate($hints, \mathcal{P}_{rslt}$);
3: Γ_T := Integrity_Proof_Generate(ADS, Γ_S);
4: **return** $\mathcal{P}_{rslt}$, and the proofs Γ_S, Γ_T, to the client;

Proof Notions

The query proof consists of two components, namely, a *shortest path proof* Γ_S and an *integrity proof* Γ_T:

- Γ_S is used to prove that $\mathcal{P}_{rslt}$ is the *shortest* possible path, i.e., $\nexists \mathcal{P}'$ such that $dist(\mathcal{P}') < dist(\mathcal{P}_{rslt})$.
- Γ_T is used to prove that both Γ_S and $\mathcal{P}_{rslt}$ are *authentic*, i.e., $v_{z_0} = v_s$, $v_{z_k} = v_t$, and $(v_{z_{i-1}}, v_{z_i}) \in E$ for each $i \in [1, k]$, where E is the original edge set.

Specifically, at Line 2 of Algorithm 1, the service provider visits the shortest path $\mathcal{P}_{rslt}$, collects necessary items from the authenticated hints $hints$, and inserts them into Γ_S. At Line 3, the service provider identifies from the ADS the necessary items to be included into Γ_T for proving the integrity of $\mathcal{P}_{rslt}$ and Γ_S.

We distinguish between two different types of the shortest path proof Γ_S: subgraph proof, and distance proof. A *subgraph proof* Γ_S^{SG} refers to a subgraph G' of the original graph G, such that the client is guaranteed to obtain the actual distance $dist(v_s, v_t)$ (for verification) by running a specific shortest path algorithm on G'. On the other hand, a *distance proof* Γ_S^{DT} contains the actual distance value $dist(v_s, v_t)$ as pre-computed by the data owner.

The crux of our verification framework is that a path is correct if (i) the reported path passes through actual nodes whose connectivity information has not been tampered with (as proven by Γ_T), and (ii) the distance of the returned path coincides with the $dist(v_s, v_t)$ computed by Γ_S (either directly by a distance proof or indirectly by a subgraph proof).

Application Requirements

Our goal is to design a verification solution with the following two desirable characteristics:

- The proof size should be as small as possible.
- The offline construction cost and storage overhead of authenticated hints should be low.

B. Subgraph Authentication via a Merkle Tree

The methods where the shortest path proof is a subgraph proof rely on proving to the user that the corresponding subgraph G' contains correct and complete node/connectivity information, i.e., that G' contains only *existing* nodes and that the *full* adjacency information for each of them is *accurately* reported. On the other hand, in methods where the shortest path proof is a distance proof, the integrity of nodes comprising the path needs to be proven. Our approach is based on the MHT authentication paradigm. Note that here we assume that Γ_S is given; Γ_S formation is discussed in subsequent sections.

Merkle Tree on Graph Nodes

The data of each node v, including its adjacency information, are encapsulated into an extended-tuple $\Phi(v)$. Specifically, $\Phi(v)$ consists of (i) the attributes of v (e.g., node identifier $v.id$, and geo-coordinates $v.x, v.y$), and (ii) the adjacent node v' and edge weight $W(v, v')$ for each edge incident to v, i.e.:

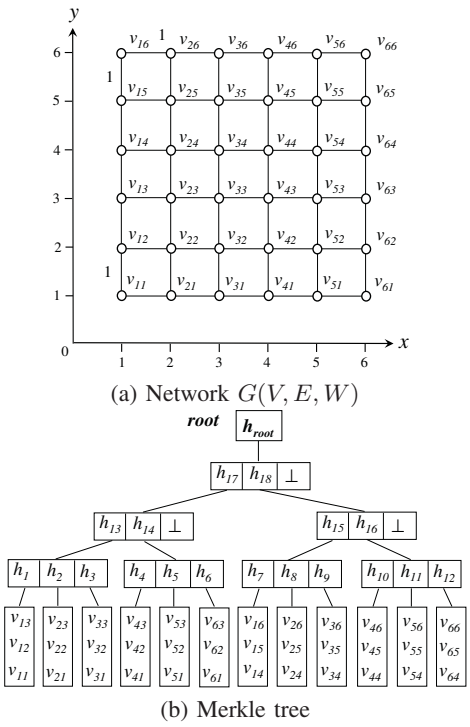
$$\Phi(v) = \langle v.id, v.x, v.y, \{\langle v', W(v, v')\rangle \mid (v, v') \in E\}\rangle \quad (1)$$

For instance, in Figure 3a, the extended-tuple of v_{16} is: $\Phi(v_{16}) = \langle 16, 1.0, 6.0, \{\langle 26, 1.0\rangle, \langle 15, 1.0\rangle\}\rangle$. In case graph G is not a spatial network, its coordinates $v.x, v.y$ are replaced by `null` values.

The purpose of our desired network certification ADS would be achieved if it would be able to authenticate the extended-tuple $\Phi(v)$ of each node v in the subgraph G' (or in the path) that corresponds to the shortest path proof Γ_S.

Let $H(\cdot)$ be a secure hash function. The digest of a node $v \in V$ is defined as $H(\Phi(v))$. By imposing a particular graph-node ordering $\mathcal{O}$ (to be elaborated shortly), the data owner is able to build a Merkle tree on $\Phi(v)$. For example, the Merkle tree in Figure 3b is constructed from

the network in Figure 3a. Hash entry h_1 is computed as $H(\ H(\Phi(v_{11})) \circ H(\Phi(v_{12})) \circ H(\Phi(v_{13}))\ )$, where $\circ$ is the concatenation operator. In a similar fashion, hash entry h_{13} is defined as $H(h_1 \circ h_2 \circ h_3)$. Note that the root entry h_{root} needs to be signed by the data owner before the Merkle tree is sent to the service provider.



(a) Network $G(V, E, W)$

(b) Merkle tree

Fig. 3. Example of a network and its certification

Subgraph Authentication

Having the shortest path proof Γ_S, the service provider generates the integrity proof Γ_T as follows. According to [11], a hash entry h_i is inserted into the integrity proof Γ_T if: (i) the subtree of h_i contains no tuple $\Phi(v)$ in Γ_S, and (ii) the parent hash entry of h_i does not satisfy condition (i).

Let us consider the example in Figure 3 again. Suppose that the service provider wishes to send the client a subgraph G' that contains nodes v_{32}, v_{33}, and v_{42} (and their incident edges). Thus, it builds set $\Gamma_S = \{\Phi(v_{32}), \Phi(v_{33}), \Phi(v_{42})\}$. It then examines the Merkle tree and generates the integrity proof as: $\Gamma_T = \{H(\Phi(v_{31})), H(\Phi(v_{41})), H(\Phi(v_{43})), h_1, h_2, h_5, h_6, h_{18}\}$. At the client side, sets Γ_S and Γ_T are combined to reconstruct the root hash h_{root}, which is then checked against the signed root hash. If it matches, then integrity is proven.

The size of the integrity proof Γ_T depends on the graph-node ordering $\mathcal{O}$ used in the Merkle tree. A desirable ordering should preserve the proximity of graph nodes in the network. The following are possible ordering choices:

- Random ordering of nodes.
- Hilbert ordering of nodes.
- Spatial partitioning (e.g., kd-tree) ordering of nodes.
- Depth-first ordering of nodes.
- Breadth-first ordering of nodes.

As we will see in the experiments, Hilbert ordering and depth-first ordering succeed in preserving the proximity of graph nodes, and consequently lead to a smaller proof size than the other orderings.

Having elaborated the computation of the integrity proof Γ_T, we turn our focus on building the shortest path proof Γ_S in the subsequent sections.

IV. Basic Solutions

This section presents basic solutions for computing the shortest path proof Γ_S. For the sake of convenience, we overload the use of Γ for representing Γ_S.

A. Dijkstra Subgraph Verification (DIJ)

We first develop a method called Dijkstra subgraph verification (DIJ). It employs a subgraph proof Γ_S^{SG} for Γ.

Proof Computation

Let v_s and v_t be the source node and the target node respectively. Recall that the extended-tuple $\Phi(v)$ of a node v consists of both the attributes of v and the full information of its incident edges. The crucial question is for which nodes should $\Phi(v)$ be inserted into the proof Γ. If every $\Phi(v)$ is inserted into Γ, then Γ contains sufficient information for verification but its size is too large. We aim at generating a proof that contains sufficient, yet no unnecessary, information for verification.

Lemma 1 shows that the service provider can generate a valid proof Γ by including the $\Phi(v)$ of each node v that is within distance $dist(v_s, v_t)$ from v_s.

Lemma 1: **Dijkstra subgraph containment.** If $\Gamma = \{\Phi(v) \mid v \in V, dist(v_s, v) \leq dist(v_s, v_t)\}$, then the shortest path distance from v_s to v_t (computed by Dijkstra's algorithm) in the subgraph defined by Γ is the same as the shortest path distance $dist(v_s, v_t)$ in the original graph G.

Proof: Since all edge weights are non-negative, Dijkstra's algorithm visits the nodes in ascending order of their distances from v_s (according to [22]). Thus, proof Γ contains all nodes required by Dijkstra's algorithm for the computation of the shortest path distance from v_s to v_t. ■

Shortest Path Verification

Note that a malicious service provider could remove some tuples from the shortest path proof Γ and then insert their corresponding digests into the integrity proof. This way, the integrity proof remains correct but the modified Γ is no longer a valid shortest path proof. It is essential that the client's verification method is able to check the validity of Γ.

First, the client applies Dijkstra's algorithm on the subgraph defined by proof Γ, in order to compute the shortest path distance $dist(v_s, v_t)$. The proof is said to be valid if each node required by Dijkstra's algorithm can be found in Γ. If (i) Γ is valid, and (ii) the shortest path distance $dist(v_s, v_t)$ on Γ is the same as the distance of the path reported by the service provider, then the path is deemed correct.

Example
We illustrate the computation of proof Γ with the example in Figure 4. Suppose that the weight of each edge equals to 1.0. The source node $v_s = v_{33}$ and the target node $v_t = v_{44}$ are shown in black.

Observe that the shortest path distance from v_{33} to v_{44} is 2.0. Thus, each node v that is within distance 2.0 from $v_s = v_{33}$ will have its extended-tuple $\Phi(v)$ inserted into the proof. Note that the incident edges of those nodes (shown as bold edges) can be found from $\Phi(v)$. Overall, the proof contains the extended-tuples of nodes: $v_{33}, v_{34}, v_{23}, v_{32}, v_{43}, v_{35}, v_{24}, v_{13}, v_{22}, v_{31}, v_{42}, v_{53}, v_{44}$ (shown in gray and black colors).

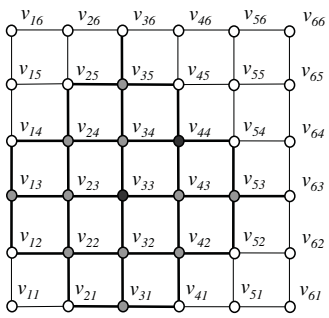


Fig. 4. The subgraph proof of DIJ with $v_s = v_{33}$ and $v_t = v_{44}$

B. Fully Materialized Distances (FULL)

In this section we present the FULL method, which utilizes materialized network distances among all pairs of graph nodes. Unlike the DIJ method described earlier, FULL employs a distance proof Γ_S^{DT} for Γ, which is much more concise than the subgraph proof used in DIJ.

Building the Authenticated Structure
First, the data owner applies the Floyd-Warshall algorithm to compute the shortest path distance $dist(v_i, v_j)$ for each pair of nodes $v_i, v_j \in V$. This pre-computation's time complexity is $O(|V|^3)$ and the total number of materialized distances is $O(|V|^2)$; note that both complexities explode with the number of nodes and, thus, FULL is only feasible for relatively small road networks.

Next, these distances are stored as tuples in the form of $\langle v_i.id, v_j.id, dist(v_i, v_j)\rangle$ in a Merkle B-tree, using $(v_i.id, v_j.id)$ as the composite key. Observe that the height of the tree is $O(f \cdot log_f|V|^2) = O(f \cdot log_f|V|)$, where f is the fanout of tree nodes.

In addition to the road network Merkle tree described in Section III-B, the above *distance Merkle tree* is also an ADS and it has to be signed by the data owner and uploaded to the service provider.

Proof Computation
Let v_s and v_t be the source and target nodes respectively. The service provider first inserts tuple $\langle v_s.id, v_t.id, dist(v_s, v_t)\rangle$ into the proof Γ. Along the tree path from the above tuple to the Merkle tree root node, the digests of sibling nodes are also inserted into Γ. Thus, the size of proof Γ is $O(f \cdot log|V|)$.

Shortest Path Verification
The client first combines the digests in Γ into a root digest, and then checks whether it matches the signed root digest of the Merkle tree. If so, then the client checks whether value $dist(v_s, v_t)$ is identical to the distance of the path returned by the service provider. In case both conditions are satisfied and Γ_T successfully proves the integrity of all path nodes, the reported shortest path is considered correct.

V. Towards Practical Authenticated Hints

Although the basic solutions DIJ and FULL presented in Section IV do succeed in providing verification means for shortest path queries, they suffer from serious performance limitations. In particular, DIJ produces very large proofs (which translates to high communication cost), while FULL is impractical for large networks because its pre-computation cost and storage overhead become prohibitively high ($O(|V|^3)$ time and $O(|V|^2)$ space respectively, which are unaffordable even for moderately sized networks). The above shortcomings motivate us to devise practical verification solutions; the methods presented in this section require the data owner to pre-compute a small amount of authenticated hints, and yet enable the service provider to generate proofs of small size.

A. Landmark-based Verification Method (LDM)

We now present the landmark-based verification method (LDM). It involves three parameters: the number c of landmarks, the number b of bits for distance quantization, and the threshold ξ for distance compression. LDM employs a subgraph proof Γ_S^{SG} for Γ.

This method exploits landmarks [26], [27] for deriving tight lower bound distances among the nodes, thus effectively reducing the proof size. Nevertheless, a large number of landmarks would also incur high overhead in the proof size. To tackle this challenge, we apply distance quantization and compression techniques for substantially reducing the size of landmark information per node, by sacrificing only a small portion of its utility.

Review on the Landmark Approach
We first briefly describe the notation used in the landmark approach [26], [27] for shortest path computation.

Let $s_1, s_2, \cdots, s_c$ be the (chosen) landmark nodes, where c is the number of landmarks used. Concrete methods for choosing landmarks can be found in [26], [27]. The *landmark distance vector* of a node v is defined according to its shortest path distances to the landmarks:

$$\Psi(v) = \langle dist(s_1, v), dist(s_2, v), \cdots, dist(s_c, v)\rangle \quad (2)$$

The lower bound distance between two nodes v and v' is defined as follows:

$$dist_{LB}(v, v') = \max_{i\in[1,c]} |dist(s_i, v) - dist(s_i, v')| \quad (3)$$

Theorem 1 shows that the lower bound distance $dist_{LB}(v, v')$ is always less than or equal to the actual distance $dist(v, v')$ between v and v'.

Theorem 1: **Lower bound property (from [26], [27]).** It holds that $dist_{LB}(v, v') \leq dist(v, v')$ for any $v, v' \in V$.

Figure 5a depicts an example of a road network. Suppose that nodes v_2 and v_7 are chosen as landmarks. The distances of each node to the landmarks are shown in Figure 5b. The lower bound distance between nodes v_3 and v_8 is computed as: $dist_{LB}(v_3, v_8) = \max\{|1-9|, |7-3|\} = \max\{8, 4\} = 8$. Note that $dist_{LB}(v_3, v_8) \leq dist(v_3, v_8)$, as $dist(v_3, v_8) = 10$.

Digest Hash Verification Framework

In order to capture the landmark distance vector $\Psi(v)$, we re-define the extended-tuple $\Phi(v)$ of a node v to include $\Psi(v)$ as follows:

$$\Phi(v) = \langle v.id, v.x, v.y, \Psi(v), \{\langle v', W(v, v')\rangle \mid (v, v') \in E\}\rangle \quad (4)$$

By utilizing the landmark-based lower bound distance $dist_{LB}(\cdot)$, Lemma 2 shows how to generate proof Γ in the LDM method.

Lemma 2: **A* subgraph containment.** If $\Gamma = \{\Phi(v), \Phi(v') \mid (v, v') \in E, v \in V, dist(v_s, v) + dist_{LB}(v, v_t) \leq dist(v_s, v_t)\}$, then the shortest path distance computed on Γ by the A* search is identical to the shortest path distance $dist(v_s, v_t)$ on the original graph G.

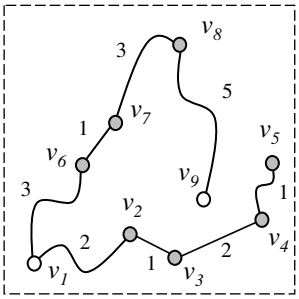
Proof: To compute the shortest path between v_s and v_t, the A* search needs to access the graph nodes v satisfying $dist(v_s, v) + dist_{LB}(v, v_t) \leq dist(v_s, v_t)$. For each adjacent node v' of the node v, the landmark vector of v' will also be examined (by the A* search) to check whether v' satisfies the above condition. Thus, proof Γ must contain both $\Phi(v)$ and $\Phi(v')$. ■

The example in Figure 5 illustrates how to build the shortest path proof Γ. Suppose that the source and target nodes are $v_s = v_1$ and $v_t = v_9$ respectively. Note that the shortest path distance $dist(v_1, v_9)$ is 12. For node v_2, we derive $dist(v_s, v_2) + dist_{LB}(v_2, v_t) = 2 + 14 = 16 > 12$, so v_2 needs not be inserted into Γ at this stage. Similarly, nodes v_3, v_4, v_5 are not inserted into Γ. On the other hand, nodes v_1, v_6, v_7, v_8, v_9 satisfy the inequality $dist(v_s, v) + dist_{LB}(v, v_t) \leq dist(v_1, v_9)$, so the extended-tuples of those nodes and their adjacent nodes are inserted into Γ. For instance, v_2 is now inserted into Γ because it is adjacent to v_1. In summary, the proof consists of extended-tuples of nodes: $v_1, v_6, v_2, v_7, v_8, v_9$.

At the client side, the verification procedure presented in Section IV-A may also be applied in LDM, with the difference that Dijkstra's algorithm must be replaced by A* search using the landmark-based lower bound distance $dist_{LB}(\cdot)$.

Quantization of Distance Vectors

A closer look reveals that the distance vector $\Psi(v)$ incurs significant overhead in the size of the extended-tuple $\Phi(v)$ in the proof. To reduce the size of the proof, we propose to quantize each landmark distance by a binary number of b bits, where the value of parameter b is decided by the data owner.



(a) Network

Node v	$dist(v_2, v)$	$dist(v_7, v)$
v_1	2	4
v_2	0	6
v_3	1	7
v_4	3	9
v_5	4	10
v_6	5	1
v_7	6	0
v_8	9	3
v_9	14	8

(b) Landmark distances

Fig. 5. Example of LDM with $v_s = v_1$ and $v_t = v_9$

Let D_{max} be an upper bound value of all the landmark distances. The quantized increment value λ is defined as:

$$\lambda = \frac{D_{max}}{2^b - 1}$$

The data owner then converts each landmark distance $dist(s_i, v)$ into the following quantized distance $dist_b(s_i, v)$, which can be represented using b bits:

$$dist_b(s_i, v) = \lambda \cdot \texttt{round}\left(\frac{dist(s_i, v)}{\lambda}\right) \quad (5)$$

where function `round` returns the nearest integer to its input value.

Lemma 3 shows that distance $dist^{loose}_{LB}(v, v')$ (of Equation 6) is always lower than or equal to the original lower bound distance $dist_{LB}(v, v')$.

$$dist^{loose}_{LB}(v, v') = \max\{\ 0,\ -\lambda + \max_{i \in [1,c]} |dist_b(s_i, v) - dist_b(s_i, v')|\ \} \quad (6)$$

Lemma 3: **Quantized distance lower bound.** Given any pair of nodes v and v', it holds that $dist^{loose}_{LB}(v, v') \leq dist_{LB}(v, v')$.

Proof: Let $\delta_i = dist_b(s_i, v) - dist(s_i, v)$. According to Equation 5 and the properties of the `round` function, we have: $|\delta_i| \leq 0.5 \cdot \lambda$. Similarly, we let $\delta'_i = dist_b(s_i, v') - dist(s_i, v')$, and derive: $|\delta'_i| \leq 0.5 \cdot \lambda$.

In Equation 3, term $|dist(s_i, v) - dist(s_i, v')|$ can be re-written as $|(dist_b(s_i, v) - dist_b(s_i, v')) + (\delta'_i - \delta_i)|$, which is lower-bounded by $|dist_b(s_i, v) - dist_b(s_i, v')| - |\delta'_i| - |\delta_i|$, using the triangular inequality. Thus, we have: $|dist(s_i, v) - dist(s_i, v')| \geq |dist_b(s_i, v) - dist_b(s_i, v')| - \lambda$.

By applying the above derivation for each $i \in [1, c]$, we obtain: $dist^{loose}_{LB}(v, v') \leq dist_{LB}(v, v')$. ■

As an example, we demonstrate how to quantize the landmark distances in Figure 5b. Value $D_{max} = 14$ is the maximum of all landmark distances. Assuming that $b = 3$, we have $\lambda = 14/7 = 2$. The resulting quantized landmark distances of the nodes are shown in Figure 6a. For instance, the original distance vector of v_4 is $\langle 3, 9\rangle$, which is quantized to $\langle 2 \cdot \texttt{round}(3/2), 2 \cdot \texttt{round}(9/2)\rangle = \langle 4, 10\rangle$. Its binary representation is $\langle 010_2\lambda, 101_2\lambda\rangle$.

Compression of Distance Vectors

We propose a distance compression technique to further reduce

Node v	$dist_b(v_2, v)$		$dist_b(v_7, v)$	
v_1	2	$001_2\lambda$	4	$010_2\lambda$
v_2	0	$000_2\lambda$	6	$011_2\lambda$
v_3	2	$001_2\lambda$	8	$100_2\lambda$
v_4	4	$010_2\lambda$	10	$101_2\lambda$
v_5	4	$010_2\lambda$	10	$101_2\lambda$
v_6	6	$011_2\lambda$	2	$001_2\lambda$
v_7	6	$011_2\lambda$	0	$000_2\lambda$
v_8	10	$101_2\lambda$	4	$010_2\lambda$
v_9	14	$111_2\lambda$	8	$100_2\lambda$

(a) Quantized distances, $\lambda = 2$

Node v	$dist_b(v_2, v)$	$dist_b(v_7, v)$
v_1	$(\theta, \epsilon) = (v_2, 2)$	
v_2	0	6
v_3	$(\theta, \epsilon) = (v_2, 2)$	
v_4	4	10
v_5	$(\theta, \epsilon) = (v_4, 0)$	
v_6	6	2
v_7	$(\theta, \epsilon) = (v_6, 2)$	
v_8	10	4
v_9	14	8

(b) Compressed distances, $\xi = 2$

Fig. 6. Example of quantized and compressed distances in LDM

the size of the distance vector $\Psi(v)$ in the extended-tuple $\Phi(v)$ in the proof. The data owner specifies a threshold parameter ξ for distance compression, which in turn controls the tightness of lower bounds derived from the compressed distances. The quantized distance difference between nodes v and v' is defined as:

$$\epsilon(v, v') = \max_{i \in [1,c]} |dist_b(s_i, v) - dist_b(s_i, v')|$$

In our distance compression algorithm, each node v is associated with (i) a reference node $v.\theta$, and (ii) a compression error $v.\epsilon$. It is an iterative greedy algorithm that attempts to maximize the number of nodes whose distance vectors can be represented by others within an error no larger than ξ. In each iteration, it first finds a node v_{rep} such that the cardinality of set $S = \{v' \in V \mid \epsilon(v', v_{rep}) \leq \xi\}$ is maximized, and then represents the distance vectors of S by using that of v_{rep}. The above procedure is repeated on the remaining uncompressed nodes in V, until no further compression is possible.

The following lemma shows that, after applying our compression method, we need to replace the lower bound distance $dist_{LB}^{loose}(v, v')$ by distance $dist_{LB}^{loose}(v.\theta, v'.\theta) - (v.\epsilon + v'.\epsilon)$ in the proof computation and verification steps.

Lemma 4: **Compressed distance lower bound.** For any pair of nodes v and v', it holds that $dist_{LB}^{loose}(v.\theta, v'.\theta) - (v.\epsilon + v'.\epsilon) \leq dist_{LB}^{loose}(v, v')$, where $v.\epsilon = \epsilon(v, v.\theta)$ and $v'.\epsilon = \epsilon(v', v'.\theta)$.

Proof: By the triangular inequality, we have: $\epsilon(v.\theta, v'.\theta) \leq \epsilon(v.\theta, v) + \epsilon(v, v') + \epsilon(v', v'.\theta)$. Thus we obtain: $\epsilon(v.\theta, v'.\theta) \leq \epsilon(v, v') + v.\epsilon + v'.\epsilon$. Note that $dist_{LB}^{loose}(v, v')$ can be re-written as $\max\{0, \epsilon(v, v') - \lambda\}$.

By subtracting λ from both sides of the above inequality, we have: $dist_{LB}^{loose}(v.\theta, v'.\theta) \leq dist_{LB}^{loose}(v, v') + v.\epsilon + v'.\epsilon$. Thus, the lemma is proven. ∎

According to our compression algorithm described earlier, we know that values $v.\epsilon$ and $v'.\epsilon$ (in Lemma 4) are upper bounded by the value of parameter ξ. In other words, ξ determines the tightness of lower bounds derived from the compressed distances.

Figure 6b shows an example of compressing the quantized distances obtained from Figure 6a. Suppose that the distance compression threshold is set to $\xi = 2$. For instance, node v_2 is a representative node and nodes v_1, v_3 satisfy $\epsilon(v', v_2) \leq \xi$. Thus, we set $v_1.\theta = v_2$ and $v_1.\epsilon = \epsilon(v_1, v_2) = 2$. Similarly, we set $v_3.\theta = v_2$ and $v_3.\epsilon = 2$. The other representative nodes are v_4 and v_6; they are used for compressing the distance vectors of v_5 and v_7 respectively. Note that nodes v_8 and v_9 are not compressed as they lie too far away from any representative node.

B. Hyper-graph Verification Method (HYP)

Our last verification scheme is the hyper-graph verification method (HYP). To reduce the communication cost, HYP first generates a subgraph proof Γ_S^{SG} on a concise coarse graph and then produces a distance proof Γ_S^{DT} on the original fine graph.

In HYP we use: (i) an adapted version of the graph node Merkle tree presented in Section III-B (in order to certify the nodes and edges in the road network), and (ii) a distance Merkle tree for the hyper-edges in the HiTi graph [28] reviewed next (which helps authenticate the shortest path distance between the source and the target node).

Review on HiTi Graph

The HiTi graph is a multi-level hierarchical structure, which was proposed originally for efficient shortest path distance computation. Performance investigation in [28] has shown that a 2-level HiTi graph achieves similar query performance to higher-level HiTi graphs. Thus, in subsequent discussions we focus on its 2-level version.

Figure 7a illustrates how to build the HiTi graph. First, the nodes in the network are partitioned into grid cells C_i based on their coordinates. Consider a node v in cell C_i. Node v is called a *border node* if it is adjacent to a node in another cell C_j. Otherwise, it is called an *inner node*. For instance, in cell C_{11}, v_2 and v_3 are border nodes (shown in gray color) whereas v_1 is an inner node.

Given any two border nodes v and v', we define a *hyper-edge* $E^*(v, v')$ between them[1], whose weight $W^*(v, v')$ is set to the shortest path distance $dist(v, v')$. Solutions of [28] can be applied for pre-computing those $W^*(v, v')$ values and storing them into a disk-based index. In the example of Figure 7a, nodes v_2, v_3, v_{22}, v_{24} are border nodes; the hyper-edges connecting them are: $E^*(v_2, v_3)$, $E^*(v_2, v_{22})$, $E^*(v_2, v_{24})$, $E^*(v_3, v_{22})$, $E^*(v_3, v_{24})$, $E^*(v_{22}, v_{24})$. For ease of illustration, the other hyper-edges are not shown here.

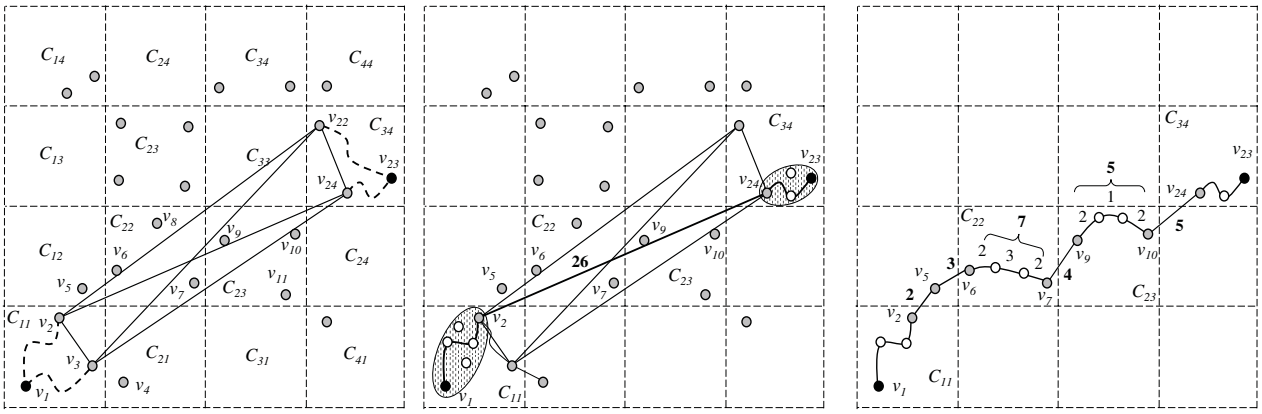
Digest Hash Verification Framework

As mentioned above, network information is certified using a modified version of the graph node Merkle tree. The tree is built identically to Section III-B, the difference being that the extended-tuple $\Phi(v)$ of a graph node v is re-defined as:

$$\Phi(v) = \langle v.id, v.x, v.y, \{\langle v', W(v, v')\rangle \mid (v, v') \in E\}, v.c, v.is\_border \rangle \quad (7)$$

where $v.c$ is the cell identifier of v, and $v.is\_border$ indicates whether v is a border node or not. Note that the definition of $\Phi(v)$ is not affected by the existence of a HiTi graph.

[1] An important difference from [28] is that we now maintain a hyper-edge for any pair of border nodes, not just for borders within the same cell.



(a) Hyper-graph of the network (b) Subgraph proof on a coarse graph (c) Distance proof on the original fine graph

Fig. 7. Example of HYP with $v_s = v_1$ and $v_t = v_{23}$

The second ADS utilized in HYP is a *distance Merkle tree* which certifies the hyper-edges and their weights. This is a Merkle B-tree materialized in a fashion similar to the FULL method.

Subgraph Proof on a Concise Coarse Graph

Let v_s and v_t be the source and target nodes respectively. The service provider first generates the proof in a coarse graph (see Figure 7b).

In the first step, the service provider conceptually formulates a coarse graph G_{coarse} which contains the extended-tuple $\Phi(v)$ for every node v that satisfies either $v.c = v_s.c$ or $v.c = v_t.c$. Consequently, the tuples in G_{coarse} cover all edges in the source and target cells, as well as all hyper-edges connecting the border nodes in those two cells.

By applying Theorem 2 (proven in [28]), we infer that the shortest path distance (between v_s and v_t) on G_{coarse} is the same as the actual distance $dist(v_s, v_t)$ (on the original graph G).

Theorem 2: **Border node passage (from [28]).** Let $\mathcal{P}$ be the actual shortest path from the source node v_s to the target node v_t. Let v_i be a node on $\mathcal{P}$ (if any) such that $v_i.c \neq v_s.c$ and $v_i.c \neq v_t.c$. It holds that within cell $v_i.c$ there exist two border nodes v_h and v_j such that v_i appears after v_h and v_i appears before v_j on the path $\mathcal{P}$. The shortest path distance from v_h to v_j is identical to the hyper-edge weight $W^*(v_h, v_j)$.

Based on the above theorem, the server includes the following information into the shortest path proof: (1) the extended tuples of all nodes in the source cell and the target cell, (2) each hyper-edge $E^*(v_{bs_i}, v_{bt_j})$ where v_{bs_i} is any border node in the source cell and v_{bt_j} is any border node in the target cell. Then, the server generates the integrity proof for (1) and (2) separately, by using the road network Merkle tree and the distance Merkle tree respectively.

Upon receiving the proof, the client is able to run Dijkstra's algorithm on G_{coarse}'s source cell and target cell, using the original graph's nodes/edges which are within the source cell and target cell. This step verifies the distances from the source node to any border node of the source cell and the distances from any border node of the target cell to the target node. After that, the client combines these distances with the weights of hyper-edges that connect border nodes between the source and target cells. This way, the client is able to obtain the actual shortest path length between v_s and v_t, and compare it against the length of the path reported by the service provider. Figure 7b shows an example of the coarse proof. The proof includes (certified information about) the hyper-edges between the set of source border nodes (v_2 and v_3) and the set of target border nodes (v_{22} and v_{24}), i.e., hyper-edges $E^*(v_2, v_{22})$, $E^*(v_2, v_{24})$, $E^*(v_3, v_{22})$, $E^*(v_3, v_{24})$. The proof also contains (the certified information of) all inner nodes in the source and target cells.

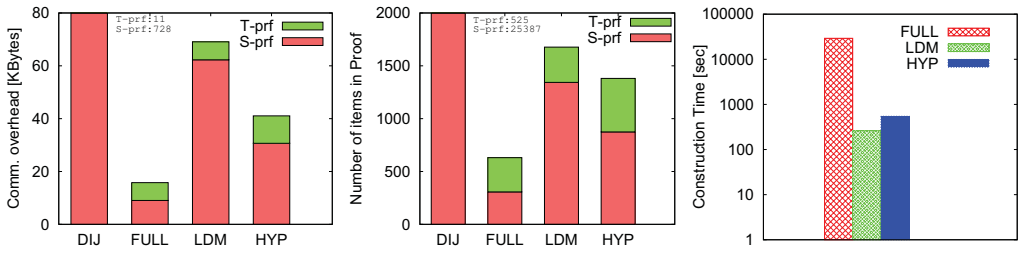
The shaded regions (in cells C_{11} and C_{34}) correspond to the subgraphs in the proof for the cells of the source and target nodes. The shortest path on the hyper-graph, i.e., hyper-edge $E^*(v_2, v_{24})$, is indicated by a bold line. The weight of this hyper-edge (26) is shown above it in bold.

Distance Proof on the Original Fine Graph

Having verified the correctness of the coarse proof, in the second step, the service provider generates a fine proof as the integrity proof of all nodes on the shortest path. The client just needs to check whether the path distance of the fine proof is the same as the shortest path distance of the coarse proof. If so, then the shortest path returned by the service provider is considered correct.

Figure 7c depicts an example of the fine proof. It contains the inner nodes (in white color) of the shortest path in intermediate cells. Let us consider the detailed sub-path between v_2 and v_{24}. The client can check that the sum of weights along the sub-path is equal to the weight of hyper-edge $W^*(v_2, v_{24}) = 26$. Thus, the detailed sub-path is regarded as correct. The main advantage of the HYP method is that there is no need to produce subgraph proofs for nodes in intermediate cells, thus significantly reducing the total size of the proof.

In practice, both the coarse and the fine proof are combined into a single proof, which can be sent to the client in a single



(a) Communication overhead (b) Number of items in Γ_S and Γ_T (c) Construction time

Fig. 8. Performance comparison under default setting

step. This way, the size of the integrity proof is reduced.

VI. Experimental Study

In this section we study the performance of DIJ, FULL, LDM and HYP in terms of the communication overhead (i.e., the cumulative size of shortest path proof and integrity proof) and the offline construction time (i.e., the time required to pre-compute the authenticated hints of each approach). We found out that the proof generation cost at the service provider and the proof verification cost at the client are roughly proportional to the proof size. Thus, we do not report those measurements here, but need to mention that client verification takes less than 100msec for FULL, LDM and HYP, and around 1.5sec for DIJ in our default setting. In the experiments we test the scalability of all methods under different network sizes, graph-node orderings, Merkle tree fanouts, and query ranges.

A. Experiment Setup

Table II shows the parameters used in our empirical study; the default values are shown in bold. In each experiment, we vary one parameter and set the others to their default values. We use four different real spatial network datasets, obtained from `http://www.maproom.psu.edu/dcw/`. They are: DE (28,867 nodes and 30,429 edges), ARG (85,287 nodes and 88,357 edges), IND (149,566 nodes and 155,483 edges) and NA (175,813 nodes and 179,179 edges). We normalize each network so that its nodes' x and y coordinates fall into a $[0..10,000]$ range. We investigate the effect of five graph-node ordering methods: breadth-first (**bfs**), depth-first (**dfs**), Hilbert ordering (**hbt**), kd-tree based (**kd**) and random (**rand**).

The query range (with default value 2,000) represents the shortest path distance used for the query workload. Specifically, we generate a workload with 100 source-target (v_s, v_t) pairs, such that the shortest path distance between the source node v_s and the target node v_t is as close to the query range as possible. Besides the parameters shown in Table II, LDM does involve additional settings; we fix the distance compression threshold ξ and the number of quantization bits b to 50.0 and 12 respectively. Due to lack of space, the effect of ξ and b on the performance of LDM is not studied here.

B. Performance Study

Performance comparison under default setting

In this experiment we set all the parameters to their default values and study the performance of the four methods. Figure 8a shows the communication overhead (i.e., the size of proofs in KBytes), while Figure 8b shows the total number of items in Γ_S and Γ_T. Since DIJ results in very large communication overhead, we truncate its bars and present its measurements as plain numbers next to the bars. We decompose the communication overhead into two parts: the size of the shortest path proof Γ_S (S-prf, lower part of each bar) and that of the integrity proof Γ_T (T-prf, upper part of each bar). DIJ incurs a huge total communication overhead which is about 10 times that of LDM, 18 times of HYP and 40 times of FULL; the reason is that DIJ expands all the nodes which are within the shortest path distance between v_s and v_t, and includes them into the proof. Unlike DIJ, the other three methods benefit from the pre-computed information and thus avoid thorough expansion. FULL incurs the smallest communication overhead since its Γ_S proof verifies a single value (in the distance Merkle B-tree), while Γ_T verifies only the extended-tuples of nodes along the shortest path (in the network Merkle tree). On the other hand, since HYP shares the advantages of FULL on the coarse graph (which is much smaller than the original graph), it incurs a smaller communication overhead than LDM. Figure 8c shows the offline construction time (in logarithmic scale); DIJ is omitted since it requires no pre-computation of authenticated hints. FULL is about 60 to 150 times slower than LDM and HYP, which confirms our claims in Section IV-B that it is impractical for large road networks. HYP has longer construction time than LDM because it materializes the distances between any pair of border nodes.

TABLE II
Experiment Parameters

Parameter	Value Range
Dataset	DE, ARG, **IND**, NA
Graph-node ordering	bfs, dfs, **hbt**, kd, rand
Query range (x1000)	0.25, 0.5, 1, **2**, 4, 8
Merkle tree fanout	**2**, 4, 8, 16, 32
Number of landmarks (c) - LDM	50, 100, **200**, 400, 800
Number of cells (p) - HYP	25, 49, 100, **225**, 400, 625

Performance comparison under different datasets

In Figure 9 we compare the performance of our verification methods for different datasets. Figure 9a shows that the relative performance of FULL, LDM and HYP in terms of communication overhead is similar for different datasets. DIJ always incurs very large communication overheads, while the

proofs of LDM and HYP are much smaller and only slightly larger than FULL. Figure 9b depicts the offline construction time (in logarithmic scale). When the network size increases, the pre-computation time of FULL explodes due to its $O(|V|^3)$ complexity.

Effect of graph-node ordering methods

Next, we experiment with different graph-node orderings (see Table II). Unlike conventional database query verification, in which the result is either a single tuple (equality search) or a set of tuples (range search) residing contiguously in the authenticated structure (e.g., a Merkle B-tree), in shortest path query verification, the nodes involved in the verification are not necessarily placed close to each other in the Merkle tree. Figure 10 shows the proof size of our methods for different graph-node orderings. **rand** leads to the largest communication overhead, while **bfs** is the second worst ordering. **hbt**, **kd** and **dfs** demonstrate similar performance with each other. The reason is that **hbt**, **kd** and **dfs** all preserve to a decent (and similar) degree the locality of the network, which implies that proof items tend to share many sibling digests in the internal levels of the corresponding Merkle tree.

Effect of Merkle tree fanout

This experiment investigates the impact of the Merkle tree fanout, i.e., of the number of children each Merkle tree node can have. Figure 11a shows that the proof size of each method increases when the fanout increases. The reason is that the larger the fanout, the larger the number of sibling Merkle tree digests that need to be included in Γ_T. All methods achieve their best performance when the Merkle tree fanout is 2. On the other hand, the relative performance of the methods is not affected by the Merkle tree fanout. LDM and HYP consistently outperform DIJ in terms of communication overhead (about 10-18 times smaller), while they are just 2.5-4 times worse than FULL.

Effect of query range

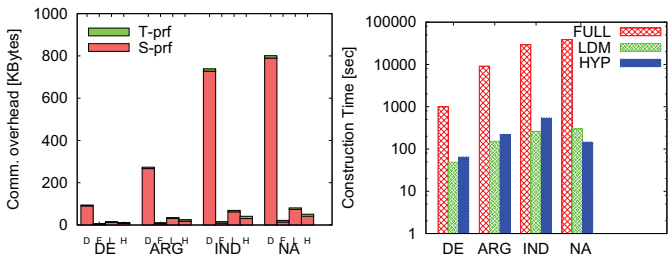
Next, we study the effect of query range on the communication overhead. A larger query range leads to a larger search space, a larger number of edges in the reported shortest path, and thus an increase in proof size. Figure 11b shows that as the query range expands, all the methods' communication overhead increases, but the performance gap between HYP and FULL shrinks from 2.8 (query range=1,000) to 1.7 times (query range=8,000). On the other hand, the proof of LDM is from 3.5 (query range=1,000) to 6.6 times (query range=8,000) larger than FULL. Note that the proof of DIJ is 3.9MBytes for query range=8,000, which is a prohibitive communication burden for most applications.

Performance evaluation of LDM

Next, we focus on LDM, and specifically on the effect of the number of landmarks c. Figure 12a plots the communication overhead versus c. When the number of landmarks increases, the proof size decreases, because the more the landmarks, the tighter the distance lower bound, and thus the smaller the search space. Figure 12b shows that the offline construction time of LDM is slightly superlinear to c.

Performance evaluation of HYP

Our final experiment evaluates the performance of HYP in terms of communication overhead and offline construction time with respect to the number of cells p used to create the HiTi graph. The increase of p means a decrease in the extent of the cells and a decrease in the number of border nodes in each cell. Thus, it results in smaller search spaces in the source cell and the target cell (for generating Γ_S). Also, the number of hyper-edges between source cell and target cell decreases dramatically. Hence, the communication overhead decreases with p (see Figure 13a). On the other hand, as shown in Figure 13b, the construction time increases in a sublinear way.



(a) Communication overhead (b) Construction time

Fig. 9. Effect of the data distribution

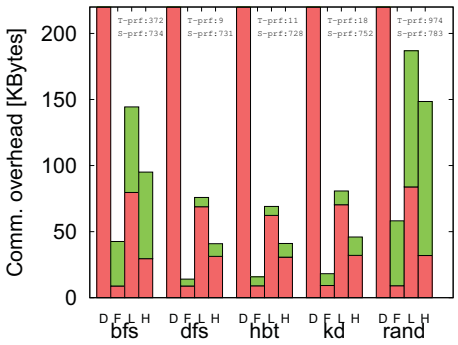
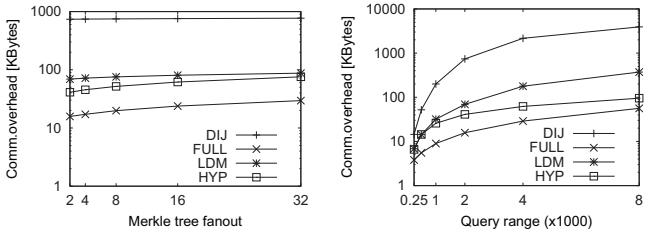


Fig. 10. Effect of the graph-node ordering

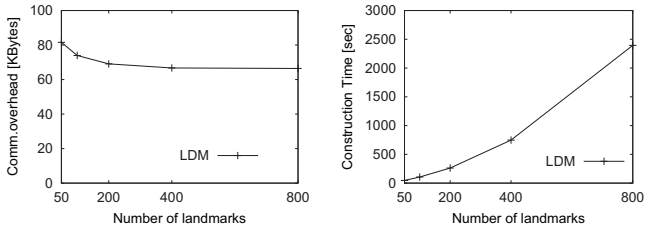


(a) vs. Merkle tree fanout (b) vs. query range

Fig. 11. Effect of the Merkle tree fanout and of the query range

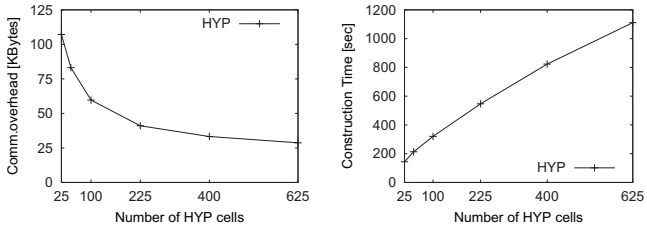
Summary

In summary, the experimental evaluation shows that although DIJ requires no pre-computation, it incurs the largest communication overhead. On the other hand, FULL has the smallest



(a) Communication overhead (b) Construction time

Fig. 12. Effect of number of landmarks



(a) Communication overhead (b) Construction time

Fig. 13. Effect of number of cells

proof, but it is impractical for large networks due to its excessive pre-computation cost. LDM and HYP achieve graceful tradeoffs between pre-computation time and communication overhead. We expect HYP's tradeoff to be more desirable than LDM for most real applications.

VII. Conclusion

In this paper we introduce the shortest path verification problem and identify its important applications in online search services. After examining the shortcomings of two basic verification solutions (DIJ and FULL), we propose the concept of authenticated hints and develop two methods (LDM and HYP) for efficient shortest path verification. LDM utilizes distance quantization and distance compression techniques for reducing the overall proof size, whereas HYP exploits a 2-level graph structure for generating a compact proof. Experiment results suggest that both LDM and HYP strike a good balance between offline construction time and proof communication overhead, with HYP typically being preferable over LDM. A promising future direction is to develop a model for estimating the proof size for shortest path verification.

References

[1] L. Wang, S. Noel, and S. Jajodia, "Minimum-Cost Network Hardening using Attack Graphs," *Computer Communications*, vol. 29, no. 18, pp. 3812–3824, 2006.

[2] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine, "Authentic Data Publication over the Internet," *J. Comput. Secur.*, vol. 11, no. 3, pp. 291–314, 2003.

[3] W. Cheng, H. Pang, and K.-L. Tan, "Authenticating Multi-dimensional Query Results in Data Publishing," in *DBSec*, 2006.

[4] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic Authenticated Index Structures for Outsourced Databases," in *SIGMOD*, 2006, pp. 121–132.

[5] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios, "Spatial Outsourcing for Location-based Services," in *ICDE*, 2008.

[6] S. Papadopoulos, D. Papadias, W. Cheng, and K.-L. Tan, "Separating Authentication from Query Execution in Outsourced Databases," in *ICDE*, 2009, pp. 1148–1151.

[7] K. Mouratidis, D. Sacharidis, and H. Pang, "Partially Materialized Digest Scheme: An Efficient Verification Method for Outsourced Databases," *VLDBJ*, vol. 18, no. 1, pp. 363–381, 2009.

[8] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. F. Cohen, "Authenticated Data Structures for Graph and Geometric Searching," in *CT-RSA*, 2003, pp. 295–313.

[9] *National Institute of Standards and Technology. Secure Hashing. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html*.

[10] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[11] R. C. Merkle, "A Certified Digital Signature," in *CRYPTO*, 1989.

[12] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A General Model for Authenticated Data Structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, 2004.

[13] H. Hacigümüs, S. Mehrotra, and B. R. Iyer, "Providing Database as a Service," in *ICDE*, 2002.

[14] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan, "Verifying Completeness of Relational Query Results in Data Publishing," in *SIGMOD*, 2005.

[15] M. Narasimha and G. Tsudik, "Authentication of Outsourced Databases Using Signature Aggregation and Chaining," in *DASFAA*, 2006, pp. 420–436.

[16] H. Pang, J. Zhang, and K. Mouratidis, "Scalable verification for outsourced dynamic databases," *PVLDB*, vol. 2, no. 1, pp. 802–813, 2009.

[17] E. Mykletun, M. Narasimha, and G. Tsudik, "Signature Bouquets: Immutability for Aggregated/Condensed Signatures," in *ESORICS*, 2004, pp. 160–176.

[18] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-Infused Streams: Enabling Authentication of Sliding Window Queries On Streams," in *VLDB*, 2007, pp. 147–158.

[19] S. Papadopoulos, Y. Yang, and D. Papadias, "CADS: Continuous Authentication on Data Streams," in *VLDB*, 2007, pp. 135–146.

[20] E. Bertino, B. Carminati, E. Ferrari, B. M. Thuraisingham, and A. Gupta, "Selective and Authentic Third-Party Distribution of XML Documents," *IEEE TKDE*, vol. 16, no. 10, pp. 1263–1278, 2004.

[21] H. Pang and K. Mouratidis, "Authenticating the Query Results of Text Search Engines," *PVLDB*, vol. 1, no. 1, pp. 126–137, 2008.

[22] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[23] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[24] B. Dantzig, *Linear Programming and Extensions*. Princeton University Press, 1962.

[25] M. Hilger, E. Kohler, R. Mohring, and H. Schilling, "Fast Point-to-Point Shortest Path Computations with Arc-Flags," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 2008.

[26] A. V. Goldberg and C. Harrelson, "Computing the Shortest Path: A* Search Meets Graph Theory," in *SODA*, 2005, pp. 156–165.

[27] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt, "Hierarchical Graph Embedding for Efficient Query Processing in Very Large Traffic Networks," in *SSDBM*, 2008, pp. 150–167.

[28] 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.

[29] 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.

[30] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh, "A Road Network Embedding Technique for k-Nearest Neighbor Search in Moving Object Databases," in *ACM-GIS*, 2002, pp. 94–10.

[31] S. Gupta, S. Kopparty, and C. V. Ravishankar, "Roads, Codes and Spatiotemporal Queries," in *PODS*, 2004, pp. 115–124.

[32] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable Network Distance Browsing in Spatial Databases," in *SIGMOD*, 2008, pp. 43–54.

[33] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance Indexing on Road Networks," in *VLDB*, 2006, pp. 894–905.