Approximate Shortest Distance Computing: A Query-Dependent Local Landmark Scheme

Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu, *Senior Member, IEEE*

Abstract—Shortest distance query is a fundamental operation in large-scale networks. Many existing methods in the literature take a landmark embedding approach, which selects a set of graph nodes as landmarks and computes the shortest distances from each landmark to all nodes as an embedding. To answer a shortest distance query, the precomputed distances from the landmarks to the two query nodes are used to compute an approximate shortest distance based on the triangle inequality.
In this paper, we analyze the factors that affect the accuracy of distance estimation in landmark embedding. In particular we find that a globally selected, query-independent landmark set may introduce a large relative error, especially for nearby query nodes. To address this issue, we propose a query-dependent local landmark scheme, which identifies a local landmark close to both query nodes and provides a more accurate distance estimation than the traditional global landmark approach. We propose efficient local landmark indexing and retrieval techniques, which achieve low offline indexing complexity and online query complexity. Two optimization techniques on graph compression and graph online search are also proposed, with the goal of further reducing index size and improving query accuracy. Furthermore, the challenge of immense graphs whose index may not fit in the memory leads us to store the embedding in relational database, so that a query of the local landmark scheme can be expressed with relational operators. Effective indexing and query optimization mechanisms are designed in this context. Our experimental results on large-scale social networks and road networks demonstrate that the local landmark scheme reduces the shortest distance estimation error significantly when compared with global landmark embedding and the state-of-the-art sketch-based embedding.

Index Terms—local landmark embedding, least common ancestor, local search, graph compression, query optimization.

1 Introduction

As the size of graphs that emerge nowadays from various application domains is dramatically increasing, the number of nodes may reach the scale of hundreds of millions or even more. Due to the massive size, even simple graph queries become challenging tasks. One of them, the shortest distance query, has been extensively studied during the last four decades. Querying shortest paths or shortest distances between nodes in a large graph has important applications in many domains including road networks, social networks, communication networks, the Internet, etc. For example, in road networks, the goal is to find shortest routes between locations; in social networks, the goal is to find the closest social relationships such as friendship or collaboration between users; while in the Internet, the goal is to find the nearest server in order to reduce access latency for clients. Although classical algorithms like breadth-first search (BFS), Dijkstra's algorithm [1], and A^* search [2], [3], [4] can compute the exact shortest paths in a network, the massive size of modern information networks and the online nature of such queries make it infeasible to apply the classical algorithms online. On the other hand, it is space inefficient to precompute and store the shortest paths between all pairs of nodes, as it requires $O(n^3)$ space to store the shortest paths and $O(n^2)$ space to store the distances for a graph with n nodes.

Recently, there have been many different methods [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15] for estimating the shortest distance between nodes based on graph embeddings. A commonly used embedding technique is *landmark embedding*, where a set of graph nodes is selected as *landmarks* [6], [12], [14] (also called *reference nodes* [11], [15], *beacons* [8], or *tracers* [5]) and the shortest distances from a landmark to all the other nodes in a graph are precomputed. Such precomputed distances can be used online to provide an approximate distance between two graph nodes based on the triangle inequality.

In this paper, we revisit the landmark embedding approach. According to the findings in the literature [5], [12], the problem of selecting the optimal landmark set is NP-hard, by a reduction from the classical NP-hard problems such as vertex cover or minimum K-center [16]. As a result, the existing studies use random selection or graph measure based heuristics such as degree, betweenness centrality, closeness centrality, coverage, etc. Despite various heuristics which try to optimize landmark selection, all the existing methods follow the triangulation based distance estimation, which estimates the shortest distance between a pair of query nodes as the sum of their distances to a landmark. As the landmark selection step is query independent, the landmark set provides a single global view for all possible queries which could be diameter apart or close by. Thus it is hard to achieve uniformly good performance on all queries. As a consequence, the landmark

• *M. Qiao, H. Cheng, L. Chang and J. X. Yu are with the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong.*
E-mail: {mqiao, hcheng, ljchang, yu}@se.cuhk.edu.hk

embedding approach may introduce a large relative error, especially when the landmark set is distant from both nodes in a query but the two nodes themselves are nearby. For example, in a US road network with 24 million nodes and 58 million edges, landmark embedding (with 50 randomly selected landmarks) has a maximum relative error of 68 for one query among $10,000$ random queries we tested.

This observation motivates us to find a query-dependent "local landmark" which is close to both query nodes for a more accurate distance estimation. In contrast, the original landmarks are called "global landmarks". In this paper, we propose a *query-dependent local landmark scheme*, which identifies a local landmark specific to a pair of query nodes. Then the distance between the two query nodes is estimated as the sum of their shortest distances to the local landmark, which is much closer than the global one. The query-dependent local landmark scheme is expected to reduce the distance estimation error in principle, compared with the traditional global landmark embedding.

Challenges. The key challenges of the *query-dependent local landmark scheme* lie in the following aspects. First, efficient local landmark indexing and retrieval techniques are needed. We cannot afford expensive online computation to find a query-specific local landmark, as it would significantly increase the query processing time. Second, the shortest distance from a query node to a local landmark needs to be efficiently computed. This distance should not be computed from scratch at the query time. These two factors are crucial to achieve efficient online query processing. Third, the embedding index should be compact. The estimation accuracy improvement and the query processing efficiency should not be achieved at the expense of an increase in the offline indexing complexity.

Bearing these goals in mind, we propose a *shortest path tree* (SPT) based local landmark scheme, where the shortest path tree rooted at node r is a tree composed of the shortest paths from r to all the other nodes. The SPTs rooted at global landmarks help to select the *query-dependent local landmarks* between two query nodes. Our main contributions are summarized as follows.

- In the traditional landmark embedding, we find that the query-independent global landmark selection introduces a large relative error, especially for nearby query nodes which are distant from the global landmarks. In light of this, we propose a *query-dependent local landmark scheme* which finds a local landmark close to both query nodes to improve the distance estimation accuracy. The local landmark scheme proves to be a robust embedding solution that substantially reduces the dependency of query performance on the global landmark selection strategy.
- Given a query node pair, the proposed local landmark scheme finds a local landmark, which is defined as the *least common ancestor* (LCA) of the two query nodes in the SPT rooted at one of the global landmarks. An $O(1)$ time algorithm for finding the LCA on an SPT is introduced. We show that the shortest path tree based local landmark scheme can significantly improve the distance estimation accuracy, without increasing the offline embedding or the online query complexity.
- Facing the challenge of immense graphs whose index may not fit in the memory, we also study to store the embedding in relational database, so that a query of the local landmark scheme can be expressed with relational operators.
- We performed extensive experiments on large-scale social networks and road networks with both memory and relational database implementations. Experimental results show that our local landmark scheme significantly reduces the average relative error to the scale of $0 - 10^{-3}$, which is orders of magnitude better than global landmark embedding, and several times better than the state-of-the-art sketch-based embedding TreeSketch [14].

2 Preliminary Concepts

Consider a weighted undirected graph $G = (V, E, w)$, where V is a set of vertices, E is a set of edges, and $w : E \mapsto \mathbb{R}^+$ is a weighting function mapping an edge $(u, v) \in E$ to a positive real number $w(u, v) > 0$, which measures the length of (u, v). We denote $n = |V|$ and $m = |E|$. For a pair of vertices $a, b \in V$, we use $\delta(a, b)$ to denote the shortest distance between a and b, and $P(a, b) = (a, v_1, v_2, \ldots, v_{l-1}, b)$ to denote the shortest path, where $\{a, v_1, \ldots, v_{l-1}, b\} \subseteq V$ and $\{(a, v_1), (v_1, v_2), \ldots, (v_{l-1}, b)\} \subseteq E$.

2.1 Landmark Embedding

Given a pair of query nodes (a, b), to efficiently estimate an approximate shortest distance between a and b, a commonly adopted approach is *landmark embedding*. Consider a set of nodes $S = \{l_1, \ldots, l_k\} \subseteq V$ which are called *landmarks*. For each $l_i \in S$, we compute the shortest distances to all nodes in V. Then for every node $v \in V$, we can use a k-dimensional vector $\overrightarrow{D}(v) = \langle \delta(l_1, v), \delta(l_2, v), \ldots, \delta(l_k, v) \rangle$ to represent its distances to the k landmarks. This is called landmark embedding, which can be used to compute an approximate shortest distance between nodes a and b based on the triangle inequality as

$$\widetilde{\delta}(a, b) = \min_{l_i \in S} \{\delta(l_i, a) + \delta(l_i, b)\} \tag{1}$$

This general embedding approach has been widely used in many existing methods in the literature.

2.2 Landmark Selection

In the landmark embedding approach, a key question is how to select the landmark set S from V, as the landmarks can heavily influence the estimation accuracy of shortest distance queries. However, selecting the optimal set of landmarks has been proven to be NP-hard, by a reduction from the classical NP-hard problems such as vertex cover [12] or minimum K-center [5]. Due to the hardness of the landmark selection problem, previous studies (e.g., [5], [12], [15]) proposed various heuristics, including random selection, degree, centrality, and coverage based selection heuristics. But the performance of different heuristics heavily depends on the graph properties, e.g., degree distribution, diameter, etc. There is no heuristic that excels in all kinds of graphs.

2.3 Factors on Embedding Performance

Here we briefly discuss the factors that affect the performance of landmark embedding.

A globally selected query-independent landmark set: Most existing methods select a single set of global landmarks which are independent of queries. Such a query-independent landmark set provides a single global view for all possible queries which could be diameter apart or close by, thus it cannot achieve uniformly good performance on all queries. The landmark set can only provide a very rough distance estimation for a query, especially when it is distant from both query nodes, and the two query nodes are close by, as shown in Figure 1.

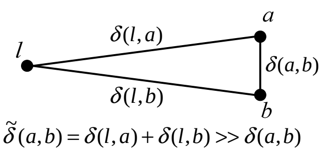

l
a
b
$\delta(l,a)$
$\delta(a,b)$
$\delta(l,b)$
$\widetilde{\delta}(a,b) = \delta(l,a) + \delta(l,b) >> \delta(a,b)$

Fig. 1. Distance Estimation with a Landmark

The number of landmarks k: In general, increasing the number of landmarks k will improve the performance of landmark embedding. An extreme case is $k = |V|$ which leads to zero estimation error. This actually corresponds to computing all pair shortest paths as an embedding. As a side effect, increasing k will cause an increase of the query processing time and the index size, as the query complexity is $O(k)$ and the index space complexity is $O(kn)$. Thus, increasing k is not an efficient or scalable solution to improve the embedding performance.

2.4 A Query-Dependent Local Landmark Scheme

In this paper we propose a novel framework, called *local landmark scheme*, for estimating the shortest distance with a small number of query-dependent local landmarks. In this framework, the problem is formulated as follows.

Problem Statement: Given an arbitrary query node pair (a, b) and a global landmark set S, our goal is to identify a query-specific local landmark which is closer to the true shortest path $P(a, b)$ than any global landmark in a graph G. The approximate shortest distance between a and b is computed as the sum of their distances to the local landmark.

The local landmark can be defined in an abstract way as:

Definition 1 (Query-Dependent Local Landmark): Given a global landmark set S and a query (a, b), a query-dependent local landmark function is

$$L_{ab}(S) : V^k \mapsto V$$

which maps S to a vertex in V called a local landmark.

With the local landmark, we can estimate a shortest distance of query (a, b) as

$$\widetilde{\delta}^L(a, b) = \delta(L_{ab}(S), a) + \delta(L_{ab}(S), b) \qquad (2)$$

Let us see an example.

Example 1: Figure 2 shows an example graph with the global landmark set $S = \{l_1, l_2, l_3\}$. In this graph, a solid line between two nodes represents an edge of unit length, while a dashed line between two nodes represents a path with zero or more intermediate connecting nodes and thus a length no less than one.

For a pair of query nodes (a, b), the path in bold (a, e, f, g, b) is the shortest path between a and b. The shortest paths from the global landmark l_1 to a and b are $(l_1, \ldots, c, e, a)$ and $(l_1, \ldots, c, d, g, b)$, respectively. Based on l_1, the estimated shortest distance between a and b is

$$\widetilde{\delta}(a, b) = \delta(l_1, a) + \delta(l_1, b) \qquad (3)$$

But if we have the shortest distances $\delta(c, a)$ and $\delta(c, b)$, we can have a more accurate distance estimation based on c than that based on l_1:

$$\widetilde{\delta}^L(a, b) = \delta(c, a) + \delta(c, b) \qquad (4)$$

as we have $\widetilde{\delta}(a, b) = \widetilde{\delta}^L(a, b) + 2\delta(l_1, c)$.

As opposed to the concept of global landmark, we call node c a local landmark with respect to query nodes a, b.

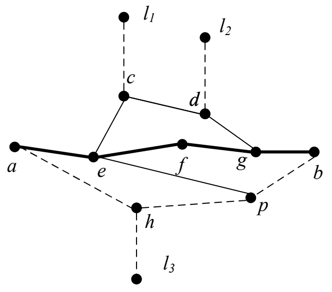


Fig. 2. Local Landmarks

3 Shortest Path Tree Based Local Landmark

In landmark embedding, the shortest distances from each global landmark $l \in S$ to all vertices in V are precomputed for the embedding purpose. To preserve more delicate information, we can further consider the *shortest path tree* (SPT) rooted at each global landmark $l \in S$.

Definition 2 (Shortest Path Tree): Given a graph $G = (V, E, w)$, the shortest path tree rooted at a vertex $r \in V$ is a spanning tree of G, such that the path from the root r to each node $v \in V$ is a shortest path between r and v, and the path length is the shortest distance between r and v.

Figure 3 shows an SPT rooted at the global landmark l_1 according to the graph in Figure 2. An SPT not only contains the shortest distance information from the tree root, it also preserves the more delicate structure information on how the two query nodes are connected. Based on the tree structure, we can identify a node, e.g., c, which is closer to nodes a and b than l_1. Based on this intuition, we propose an SPT based local landmark function.

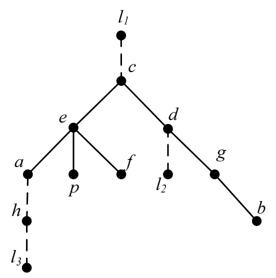


Fig. 3. Shortest Path Tree Rooted at l_1

3.1 SPT Based Local Landmark Function

In Example 1, we find that the distance estimation based on node c, i.e., $\widetilde{\delta}^L(a,b) = \delta(c,a) + \delta(c,b)$ is a tighter upper bound than that based on l_1. If we look at the SPT rooted at l_1 in Figure 3, it is not hard to find that c is the *least common ancestor* (LCA) of a and b.

Definition 3 (Least Common Ancestor): Let T be a rooted tree. The least common ancestor of two nodes u and v in T, denoted as $LCA_T(u,v)$, is the node furthest from the root that is an ancestor of both u and v.

In light of this, we propose an SPT based local landmark function, which returns the LCA of the query nodes a, b in the SPT rooted at each landmark $l \in S$.

Definition 4 (SPT Based Local Landmark Function): Given a global landmark set S and a query (a,b), the SPT based local landmark function is defined as:

$$L_{ab}(S) = \arg \min_{r \in \{LCA_{T_l}(a,b) | l \in S\}} \{\delta(r,a) + \delta(r,b)\}$$

where $LCA_{T_l}(a,b)$ denotes the least common ancestor of a and b in the SPT T_l rooted at $l \in S$.

We can show that the distance estimation with the SPT based local landmarks is more accurate, or at least the same accurate as that with the global landmark set.

Theorem 1: Given a global landmark set S, $\forall a,b \in V$, we have

$$\delta(a,b) \leq \widetilde{\delta}^L(a,b) \leq \widetilde{\delta}(a,b).$$

Proof: First, $\delta(a,b) \leq \widetilde{\delta}^L(a,b)$ holds according to the triangle inequality.

Next, $\forall l \in S$, $r = LCA_{T_l}(a,b)$, we have

$$\delta(l,a) + \delta(l,b) = \delta(r,a) + \delta(r,b) + 2\delta(r,l)$$

As $\delta(r,l) \geq 0$, we have

$$\delta(r,a) + \delta(r,b) \leq \delta(l,a) + \delta(l,b)$$

Consequently,

$$\begin{aligned}\widetilde{\delta}^L(a,b) &= \min_{l \in S}\{\delta(LCA_{T_l}(a,b),a) + \delta(LCA_{T_l}(a,b),b)\} \\ &\leq \min_{l \in S}\{\delta(l,a) + \delta(l,b)\} \\ &= \widetilde{\delta}(a,b)\end{aligned}$$

□

To efficiently calculate $\widetilde{\delta}^L(a,b)$, $a \neq b \in V$, we have:

$$\begin{aligned}\widetilde{\delta}^L(a,b) &= \min_{l \in S}\{\delta(LCA_{T_l}(a,b),a) + \delta(LCA_{T_l}(a,b),b)\} \qquad (5) \\ &= \min_{l \in S}\{\delta(l,a) + \delta(l,b) - 2\delta(l, LCA_{T_l}(a,b))\}\end{aligned}$$

When $LCA_{T_l}(a,b)$, $\forall l \in S$ is known, Eq.(5) can be computed in $O(|S|)$ time. For each $l \in S$, it simply looks up three embedded distances $\delta(l,a)$, $\delta(l,b)$ and $\delta(l, LCA_{T_l}(a,b))$.

3.2 LCA Computation

We introduce the techniques in [17] for efficiently finding the LCA of two nodes in an SPT T in $O(1)$ time with an $O(n)$ size index, where n is the number of nodes in T. A closely related problem to LCA is *Range Minimum Query* (RMQ), the solution to which leads to the solution to LCA.

Definition 5 (Range Minimum Query): Let A be an array of length n, namely $A[1,\ldots,n]$. For indices $1 \leq i \leq j \leq n$, $RMQ_A(i,j)$ returns the index of the smallest element in the subarray $A[i,\ldots,j]$.

This linkage between the two problems is based on the following observation. $LCA_T(a,b)$ is the shallowest node encountered between the visits to a and b during a depth first search of a tree T. Based on this linkage, we can reduce the LCA problem to RMQ as follows:

1) Perform a depth first search on the tree T and record the node label sequence $trace[1,\ldots,2n-1]$ in the Euler Tour of the tree. The Euler Tour of T traverses each of the $n-1$ edges in T twice, once in each direction, during a DFS traversal. Therefore, the array $trace$ has a length $2n-1$. For example, for the shortest path tree in Figure 3, $trace = (l_1,\ldots,c,e,a,\ldots,h,\ldots,l_3,\ldots,h,\ldots,a,e,p,e,f,e, c,d,\ldots,l_2,\ldots,d,g,b,g,d,c,\ldots,l_1)$.
2) Record the level of the nodes in T in an array $L[1,\ldots,2n-1]$, where $L[i]$ is the level of the node $trace[i]$. We define the level of the root as 0, and the level of a child increases that of its parent by 1.
3) For the n nodes in the tree T, record the timestamp of each node in $stamp[1,\ldots,n]$, where $stamp[a] = \min_i\{trace[i] = a\}$, $a \in V$. Here the "timestamp" refers to a counter of consecutively increasing integers starting from 0. The $stamp$ array records the time when a node is visited for the first time.

With the above transformation, without loss of generality, for two nodes a, b, let $stamp[a] < stamp[b]$, then

$$\begin{aligned}LCA_T(a,b) &= trace[\arg \min_{stamp[a] \leq i \leq stamp[b]} L[i]] \qquad (6) \\ &= trace[RMQ_L(stamp[a], stamp[b])]\end{aligned}$$

The query $RMQ_L(stamp[a], stamp[b])$ finds the index of the node with the smallest level, i.e., the shallowest node, in the subarray $L[stamp[a], stamp[b]]$. According to [17], an RMQ query can be answered in $O(1)$ time with an index of size $O(n)$. We need to perform the DFS traversal $|S|$ times, one for each landmark, thus it takes $O(|S|n)$ processing time and $O(|S|n)$ index space in total.

3.3 Complexity Analysis

We analyze the online query complexity and the offline embedding complexity of LLS.

Online Query Time Complexity: The query is based on Eq.(5). For each global landmark $l \in S$, there are three lookup operations to retrieve the embedded distances, which take $O(1)$ time. In addition, there is an LCA query which can be answered in $O(1)$ time by RMQ. For all global landmarks in S, the query time complexity is $O(|S|)$.

Offline Embedding Space Complexity: The space requirement of LLS can be partitioned into three parts: (1) embedded distances from each global landmark to every node in the graph in $O(|S|n)$ space; (2) shortest path trees and the corresponding *trace*, L and *stamp* arrays for all global landmarks in $O(|S|n)$ space; and (3) RMQ index tables for all global landmarks in $O(|S|n)$ space. Combining the above three factors, the offline embedding space complexity of LLS is $O(|S|n)$.

Offline Embedding Time Complexity: Given a global landmark set S, the time complexity to compute the single-source shortest paths from a landmark by Dijkstra's algorithm [1] is $O(m + n \log n)$. It can be simplified to $O(n \log n)$ when the graph G is sparse. We also have to build the RMQ index in $O(n)$ time for each global landmark in S. Thus the total embedding time complexity is $O(|S|n \log n)$.

When compared with GLS, it is not hard to verify that our LLS has the same online query complexity and offline embedding complexity, although our complexities have slightly larger constant factors.

3.4 Extending LLS to Directed Graphs

Our LLS method is mainly designed for undirected graphs. Here we briefly discuss how to extend it to handle directed graphs. For each landmark l, we build two SPTs rooted at l. One is a backward tree TB_l, where the tree path $P_{TB_l}(v, l)$ is the shortest path from v to l for $v \in V$. The other is a forward tree TF_l, where the tree path $P_{TF_l}(l, v)$ is the shortest path from l to v for $v \in V$. Given a query (x, y), for landmark l, we first retrieve two tree paths, $P_{TB_l}(x, l)$ and $P_{TF_l}(l, y)$, then find one of their common nodes with the smallest distance estimation, i.e., $\min_{v \in P_{TB_l}(x,l) \cap P_{TF_l}(l,y)} \{d(x, v) + d(v, y)\}$, as a local landmark candidate. Finally, we report

$$\widetilde{d}(x, y) = \min_{l \in S} \{ \min_{v \in P_{TB_l}(x,l) \cap P_{TF_l}(l,y)} \{d(x, v) + d(v, y)\}\}$$

as the approximate distance. Using hashing techniques, the query time is $O(\Sigma_{l \in S}(|P_{TB_l}(x, l)| + |P_{TF_l}(l, y)|))$ where $|P_{TB_l}(x, l)|$ and $|P_{TF_l}(l, y)|$ denote the number of hops in the corresponding paths. The embedding uses $O(|S|n)$ space to store the forward and backward SPTs for all landmarks in S.

4 Optimization Techniques

In this section, we propose two additional techniques, *graph compression* and *local search* to further optimize the performance of our local landmark scheme. Graph compression aims to reduce the embedding index size by compressing the graph nodes, and local search performs limited scope online search to improve the distance estimation accuracy.

4.1 Index Reduction with Graph Compression

As we have shown, the embedding index takes $O(|S|n)$ space, which is a linear function of the graph node number n. Thus we can effectively reduce the embedding index size if the graph nodes can be compressed. Towards this goal, we propose graph compression techniques which reduce some simple local graph structures with low-degree nodes to a representative node. Our compression techniques are lossless, thus do not sacrifice the distance query accuracy.

4.1.1 Graph Compression and Index Construction

We first define two types of special graph nodes, i.e., *tree node* and *chain node*.

Definition 6 (Graph Incident Tree and Tree Node): A tree $T = (V_T, E_T, r)$ with the root r is a graph incident tree on a graph $G = (V, E)$ if (1) $V_T \subset V$, $E_T \subset E$; and (2) for any path $P(u, v)$ between $u \in V_T$ and $v \in V - V_T$, P must go through the tree root r. A graph incident tree is maximal if it is not contained in another graph incident tree. The nodes $V_T - \{r\}$ are called tree nodes and the root r is the entry node of all tree nodes in T.

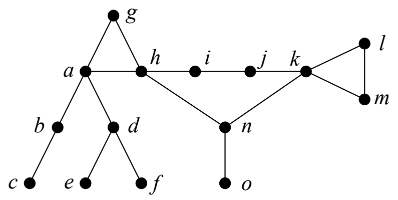


Fig. 4. Graph Compression Example

For example, in Figure 4, the tree with nodes a, b, c, d, e, f is a maximal graph incident tree, where the root a is the *entry node* and nodes b, c, d, e, f are the *tree nodes*. A graph incident tree can be simply discovered by recursively removing graph nodes with degree 1 until the entry node is met (with degree > 1). As the entry node is the only access point for a tree node to connect to the rest of the graph, it is sufficient to keep the entry node as a representative. The graph incident tree is thus compressed to the entry node by removing all the tree nodes. In addition, the distance from each tree node to the entry node is saved in an array. After we remove all tree nodes, we next identify the chain nodes.

Definition 7 (Chain Node): Given a graph $G = (V, E)$, a chain node $v \in V$ is a non tree node with $degree(v) = 2$.

If we trace through the two edges incident on a chain node respectively, the two nodes which are first encountered through the chain with a degree greater than 2 are called *end nodes*. The two end nodes may be identical when a cycle exists. For example in Figure 4, nodes i and j are *chain nodes* with *end nodes* h and k. We will remove the chain nodes and the incident edges, and then connect the two end nodes with a new edge, whose length is equal to the length of the chain. The distances from a chain node to both end nodes are saved in an array.

After we remove the tree nodes and chain nodes and their incident edges from a graph $G = (V, E)$, we obtain a compressed graph $G' = (V', E')$. Then the index structures including the embedded distances, shortest path trees and RMQ index tables are constructed on top of G', instead of G. As $|V'| < |V|$, the index size can be effectively reduced.

One point worth noting is that, as a graph incident tree is compressed to a single entry node, the tree structure is lost. When given two query nodes from the same tree, their LCA cannot be identified from the compressed graph G'. For example, for nodes e, f in Figure 4, their LCA d cannot be identified from the compressed graph, as d has been removed as a tree node. In order to handle such queries, we select one global landmark $l \in V$ and build the embedding index including the shortest path tree and RMQ table on the *original graph*; and select the other global landmarks from V' and build the index on the *compressed graph*. For any two tree nodes on the same graph incident tree, e.g., e, f, their LCA is the same in all shortest path trees rooted at any global landmarks. Thus it is sufficient to build the full index on the original graph for one global landmark only. The space complexity is $O(n + (|S| - 1)n')$, which is smaller than $O(|S|n)$ on the original graph.

4.1.2 Query Processing on Compressed Graph

Given a query (a, b), if $a, b \in V'$, the local landmark based approximate shortest distance $\widetilde{\delta}^L(a, b)$ can be estimated by Eq.(5) in the same way as in the original graph; otherwise, if at least one of a, b is a tree node or chain node and not in V', then

$$\widetilde{\delta}^L(a, b) = \min_{\substack{r_a \in map(a),\\ r_b \in map(b)}} \{\delta(a, r_a) + \widetilde{\delta}^L(r_a, r_b) + \delta(b, r_b)\} \tag{7}$$

where $map(a)$ contains the representative nodes for a, defined as follows: (1) if a is a tree node, $map(a)$ contains the corresponding entry node; (2) if a is a chain node, $map(a)$ contains the two end nodes; and (3) if $a \in V'$, $map(a)$ is a itself. The intermediate query $\widetilde{\delta}^L(r_a, r_b)$ can be answered by Eq.(5), as $r_a, r_b \in V'$, according to the definition of $map()$.

There are two special cases to be handled separately.

1) If a, b are tree nodes from the *same* graph incident tree, the query $\widetilde{\delta}^L(a, b)$ can be answered with the local landmark from the index on the original graph. For example, for query (e, f) in Figure 4, $\widetilde{\delta}^L(e, f) = \delta(e, d) + \delta(f, d)$, where d is the LCA of e and f.
2) If a, b are two chain nodes with the same end nodes, then $\widetilde{\delta}^L(a, b) = \min\{d, |\delta(a, r) - \delta(b, r)|\}$, where d is estimated by Eq.(7), and r is either one of the two end nodes. For example, for query (i, j) in Figure 4, there are two paths $P_1 = (i, j)$ and $P_2 = (i, h, n, k, j)$ between i and j. The distance $\widetilde{\delta}^L(i, j)$ is estimated by the shorter one among P_1 and P_2.

Our graph compression is lossless. Thus in all the above cases, the estimated distance based on the compressed graph will be the same as that based on the original graph.

4.2 Improving Accuracy by Local Search

In this subsection, we propose an online *local search* (LS) technique which performs a limited scope local search on the graph and may find a shortcut with a smaller distance than that based on LLS. Given a query (a, b), for each global landmark $l \in S$, we can find $LCA_{T_l}(a, b)$ in T_l rooted at l. The shortest path between a query node and a local landmark $LCA_{T_l}(a, b)$ can also be obtained from the corresponding SPT T_l. If we trace the shortest paths from a to all the LCAs (similarly from b to all the LCAs), we can form two *partial shortest path trees* rooted at a and b respectively, e.g., T_a and T_b in Figure 5 following Example 1. A leaf node in such trees must be an LCA; while it is also possible an LCA is an intermediate node, if it lies on the shortest path from a query node to another LCA, e.g., the intermediate node c in T_a in Figure 5 (a).

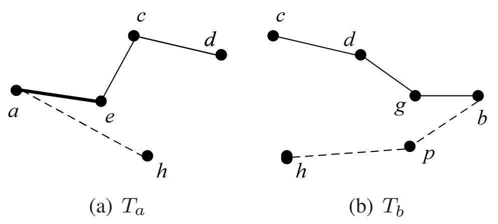


Fig. 5. Partial Shortest Path Trees

The local search expands a partial shortest path tree T by a width of c, i.e., for each node in T, its neighbors within c hops in the graph are included in the expanded tree T^c. For the two expanded trees T_a^c and T_b^c rooted at the query nodes, the common nodes of T_a^c and T_b^c act as bridges to connect the two query nodes. We will find a path connecting the two query nodes through a bridge with the smallest distance. If the distance is smaller than the estimation $\widetilde{\delta}^L(a, b)$ by LLS, we will report this local search distance as a more accurate estimation for the query (a, b). Figure 6 shows the 1-hop expanded trees T_a^1 and T_b^1, where the 1-hop neighbors of every tree node in T_a and T_b are included. The yellow shade illustrates (in an abstract way) that each node in the dashed path is also expanded to include its 1-hop neighbor. Based on the expanded trees there are four paths connecting a and b, i.e., (a, e, c, d, g, b), (a, e, f, g, b), $(a, e, p, \ldots, b)$ and $(a, \ldots, h, \ldots, p, \ldots, b)$. As (a, e, f, g, b) has the shortest distance between a and b, we return the distance as the answer.

Algorithm 1 shows the pseudocode of the local search. Lines 2-3 build two partial shortest path trees rooted at a and b respectively to all the local landmarks. Lines 4-5 expand the two trees to include the neighbors within c hops for each tree node. $dist_{T_a^c}(a, r)$ in line 8 represents the path length from a to r in the expanded tree T_a^c.

The setting of the search width c is a trade-off between the accuracy and the online query cost. If we set c to 2 or above, for graphs with high average degree, e.g., social networks, the online search space will explode and the query time will become too long. On the other hand, if

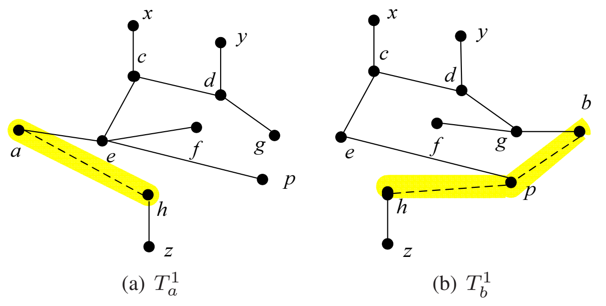


Fig. 6. 1-Hop Expanded Trees T_a^1 and T_b^1

Algorithm 1 Local Search

Input: A query (a, b) and the expansion width c.
Output: The shortest distance of a path.

```
1: $LCA \leftarrow \{LCA_{T_l}(a, b) | l \in S\}$
2: $T_a \leftarrow partial\_SPT(a, LCA)$
3: $T_b \leftarrow partial\_SPT(b, LCA)$
4: $T_a^c \leftarrow Tree\_Expansion(T_a, c)$
5: $T_b^c \leftarrow Tree\_Expansion(T_b, c)$
6: $dist \leftarrow \infty$
7: for $r \in T_a^c \cap T_b^c$ do
8:     if $dist_{T_a^c}(a, r) + dist_{T_b^c}(b, r) < dist$ then
9:         $dist \leftarrow dist_{T_a^c}(a, r) + dist_{T_b^c}(b, r)$
10: return $dist$
```

we set $c = 0$ without edge expansion, it will be faster but much less accurate than the $c = 1$ case (note that LS with $c = 0$ is more accurate than LLS or the same, since it searches a shortcut between two query nodes by connecting two partial trees directly). Thus, in our experiment, we set $c = 1$ to achieve a good balance.

4.2.1 Comparing Local Search with TreeSketch

TreeSketch [14] is a sketch-based method for shortest distance/path estimation. It also uses online search to improve the accuracy. The main differences between TreeSketch and LS include different search order and stop condition. In TreeSketch, the sketch of node s denoted as T_s is a tree rooted at s. For a query $q = (s, d)$, TreeSketch performs a bi-directional search on T_s and T_d, and the expansion follows a breadth-first search order on each side. Let V_{BFS} and V_{RBFS} denote the sets of visited nodes from two sides respectively. Consider $u \in T_s$ and $v \in T_d$ that are two nodes under expansion in the current iteration. TreeSketch checks if there is an edge from u to a node in V_{RBFS} or from a node in V_{BFS} to v. If yes, then an s-d path is found and added to a queue Q. Denote the length of the shortest path in Q as l_{min}, the algorithm terminates if $dist(s, u) + dist(v, d) \geq l_{min}$. This early stop condition may miss a better shortcut and thus return a less optimal answer. Figure 7 is an example. l_1 and l_2 are two landmarks. For query (s, d), all the solid arrows are edges in $T_s \cup T_d$ while the dashed arrows are not. The solid curves are paths with one or more edges in T_s or T_d. During the search process, when b is visited, $V_{BFS} = \{j, a, f\}$ and $V_{RBFS} = \{e, c, b\}$, TreeSketch finds the first shortcut

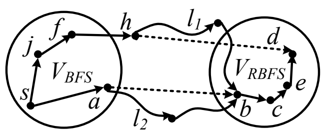


Fig. 7. Comparing Local Search with TreeSketch

$p = p(s, a) \circ (a, b) \circ p(b, d) = (s, a, b, c, e, d)$, and updates $l_{min} = 5$. TreeSketch finds $dist(s, f) + dist(b, d) = 5 = l_{min}$ and then terminates the search. In contrast, LS can find a better shortcut $p^* = (s, j, f, h, d)$ with length 4 by local search, but p^* is missed by TreeSketch due to its early stop mechanism.

5 Local Landmark Scheme on Relational Database

Although the proposed memory based LLS and LS methods can estimate an approximate shortest distance between two nodes efficiently, the index size in $O(|S|n)$ increases linearly with the graph size. For very large graphs, the index may become too large to fit in the memory. This limitation motivates us to consider a scalable disk-based index. In this section, we propose to build a disk-based index on relational database (RDB) due to its powerful indexing and query optimization mechanisms. In the following, we will study how to design a database schema to store the index and how to use RDB features to optimize the query performance.

5.1 LLS on Relational Database

We first study how to implement the local landmark scheme on relational database, denoted as LLS_{db}. To distinguish, the memory based LLS described before is denoted as LLS_{mem}.

5.1.1 Database Schema For LLS_{db}

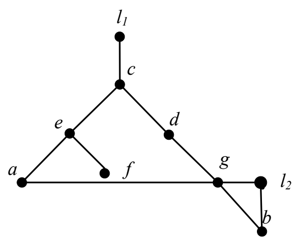
Recall the local landmark scheme estimates a shortest distance of a query (a, b) as $\widetilde{\delta}^L(a, b) = \delta(L_{ab}(S), a) + \delta(L_{ab}(S), b)$ in Eq.(2). If we store $\delta(L_{ab}(S), a)$ and $\delta(L_{ab}(S), b)$ in a relational table, they can be retrieved to answer a shortest distance query. Thus we create a table, called Tbl_D, with $Tbl_D\_schema = (s, t, d)$ where $s, t \in V$ are two nodes and $d = \delta(s, t)$ is the true shortest distance from s to t. (s, t) is designated as the primary key to support efficient selection and join operations on them.

We populate Tbl_D to build a local landmark embedding as follows. On an SPT rooted at a global landmark l, consider a path $P(v, l) = (v, v_1, v_2, \ldots, v_t, l)$ for any $v \in V$, we calculate $\delta(v, v_i) = \delta(v, l) - \delta(v_i, l)$ and insert the tuple $\langle v, v_i, \delta(v, v_i)\rangle$ into Tbl_D, for $i = 1, 2, \ldots, t$. Algorithm 2 shows how to populate Tbl_D using SPTs and the shortest distance $\delta(v, l)$, $\forall v \in V$ and $\forall l \in S$.

Example 2: Figure 8 shows a graph with two global landmarks l_1 and l_2. Based on the SPTs rooted at l_1 and l_2 respectively, we can build Tbl_D on the right, which only lists tuples with attribute $s = a$ or $s = b$ here.

Algorithm 2 Constructing $\mathsf{Tbl_D}$

```
Input: SPTs for landmark set $S$ and $\delta(v,l), v \in V, l \in S$
1: for $\forall v \in V, l \in S$ do
2:     Obtain $P(v,l) = (v, v_1, v_2, \ldots, v_t, l)$ from SPT $T_l$;
3:     for $i = 1, \ldots, t$ do
4:         $\delta(v, v_i) \leftarrow \delta(v,l) - \delta(v_i, l)$;
5:         Insert tuple $\langle v, v_i, \delta(v, v_i)\rangle$ into $\mathsf{Tbl_D}$;
```



s	t	d
a	e	1
a	c	2
a	l_1	3
a	g	1
a	l_2	2
b	g	1
b	d	2
b	c	3
b	l_1	4
b	l_2	1

Fig. 8. Example for Building $\mathsf{Tbl_D}$

5.1.2 $\mathsf{LLS_{db}}$ *Query*

For $\mathsf{LLS_{db}}$, we express the shortest distance query in relational algebra as follows.

$$\widetilde{\delta}_{\mathsf{LLS_{db}}}(x,y) \leftarrow \mathcal{G}_{\min(t_1.d+t_2.d)}(\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D}))) \quad (8)$$

which is composed of a join operation and an aggregation operation. Firstly, it selects two groups of tuples where $s = x$ and $s = y$ and renames them as t_1 and t_2 respectively, then it joins t_1 and t_2 using the condition $t_1.t = t_2.t$. Such $t_1.t$ is a local landmark. In $\mathsf{LLS_{db}}$, any node can be a local landmark, as long as it satisfies the condition $t_1.t = t_2.t$. This is different from $\mathsf{LLS_{mem}}$ which restricts the local landmarks as LCAs of two query nodes. Finally, the aggregation operator $\mathcal{G}_{\min}$ finds the minimum $t_1.d + t_2.d$ over all joined tuples.

5.1.3 *Accuracy Analysis*

$\mathsf{LLS_{db}}$ is more accurate, or at least the same accurate as $\mathsf{LLS_{mem}}$, i.e.,

$$\widetilde{\delta}_{\mathsf{LLS_{db}}}(x,y) \leq \widetilde{\delta}_{\mathsf{LLS_{mem}}}(x,y)$$

The proof can be sketched as follows. $\mathsf{LLS_{mem}}$ uses LCAs on SPTs as local landmarks. For any global landmark $l \in S$, we denote $LCA_{T_l}(x,y)$ as r. Obviously, tuples $\langle x, r, \delta(x,r)\rangle$ and $\langle y, r, \delta(y,r)\rangle$ are in $\mathsf{Tbl_D}$ and they satisfy the $t_1.t = t_2.t$ condition. Thus the LCAs are a subset of local landmarks considered in $\mathsf{LLS_{db}}$. Under the aggregation operator $\mathcal{G}_{\min}$, we prove $\widetilde{\delta}_{\mathsf{LLS_{db}}}(x,y) \leq \widetilde{\delta}_{\mathsf{LLS_{mem}}}(x,y)$. Example 3 shows one such case.

Example 3: In Figure 8, there are two global landmarks l_1 and l_2. Given a query (a,b), $LCA_{T_{l_1}}(a,b) = c$, and $LCA_{T_{l_2}}(a,b) = l_2$. Therefore, by $\mathsf{LLS_{mem}}$ we have

$$\widetilde{\delta}_{\mathsf{LLS_{mem}}}(a,b) = \min\{\delta(a,c)+\delta(b,c), \delta(a,l_2)+\delta(b,l_2)\} = 3$$

In comparison, in $\mathsf{LLS_{db}}$, we have two tuples in $\mathsf{Tbl_D}$, $\langle a, g, 1\rangle$ and $\langle b, g, 1\rangle$, by joining which we have a more accurate estimation as

$$\widetilde{\delta}_{\mathsf{LLS_{db}}}(a,b) = 1 + 1 < 3 = \widetilde{\delta}_{\mathsf{LLS_{mem}}}(a,b)$$

g is a local landmark providing a shortcut between a, b.

For the memory based LS, if we set $c = 0$, it will report the same estimated distance as $\mathsf{LLS_{db}}$, since both of them essentially find shortcuts between two query nodes by joining two partial trees without edge expansion. From this perspective, it is not surprising that $\mathsf{LLS_{db}}$ achieves a better accuracy than $\mathsf{LLS_{mem}}$.

5.2 Local Search on Relational Database

We now study how to implement local search on relational database, denoted as $\mathsf{LS_{db}}$.

5.2.1 *Database Schema For* $\mathsf{LS_{db}}$

We create another table $\mathsf{Tbl_G}$ which stores the graph edges E and serves for online expansion. We define the schema $Tbl_G\_schema = (s,t,d)$ where $e(s,t) \in E$ represents an edge and $d = w(s,t)$ is the edge weight. (s,t) is designated as the primary key to support efficient selection and join operations on them. For a graph $G(V,E,w)$, we insert all edges in E and a self-loop for each node in V to $\mathsf{Tbl_G}$:

$$\mathsf{Tbl_G} \leftarrow \{\langle u, v, w(u,v)\rangle | (u,v) \in E\} \cup \{\langle v, v, 0\rangle | v \in V\}$$

The purpose of adding self-loops will be made clear shortly.

5.2.2 $\mathsf{LS_{db}}$ *Query*

We define two slightly different local search queries: $\mathsf{unidirectional\text{-}expansion}$, denoted as $\mathsf{LS_{dbu}}$, and $\mathsf{bidirectional\text{-}expansion}$, denoted as $\mathsf{LS_{dbb}}$. We first express $\mathsf{LS_{dbu}}$ in relational algebra as follows.

$$\widetilde{\delta}_{\mathsf{LS_{dbu}}}(x,y) \leftarrow \mathcal{G}_{\min(t_1.d+\mathsf{Tbl_G}.d+t_2.d)}(\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=\mathsf{Tbl_G}.s} \mathsf{Tbl_G} \bowtie_{\mathsf{Tbl_G}.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D}))) \quad (9)$$

It is composed of two join operations. The tuples selected by $\sigma_{s=x}(\mathsf{Tbl_D})$ correspond to a set of shortest paths originating from x to some graph nodes $t_1.t$. The first join operator expands these shortest paths from x by one edge with the condition $t_1.t = \mathsf{Tbl_G}.s$, and the second join operator connects the expanded paths with the shortest paths originating from y with the condition $\mathsf{Tbl_G}.t = t_2.t$. Conversely if we perform the second join operation before the first join, this corresponds to expanding the shortest paths originating from y and then connecting them with the shortest paths from x. The final result will be the same regardless of the join order. Here we do not explicitly specify a join order and leave it to DBMS. The aggregation operator $\mathcal{G}_{\min}$ finds the minimum $t_1.d + \mathsf{Tbl_G}.d + t_2.d$ over all joined tuples.

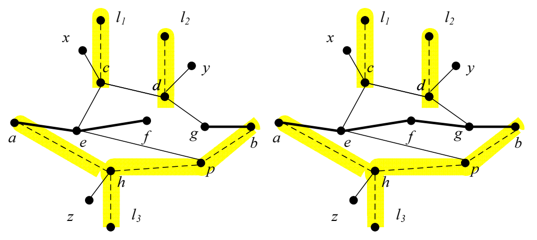
Note that if t_1 has a tuple $\langle x, v, \delta(x,v)\rangle$ and t_2 has a tuple $\langle y, v, \delta(y,v)\rangle$, $v \in V$, then $\mathsf{LS_{dbu}}$ will join these two tuples through an intermediate tuple $\langle v, v, 0\rangle$ corresponding to a self-loop on v and generate a distance of $\delta(x,v) + 0 + \delta(y,v)$. This actually performs no edge expansion and is the same as $\mathsf{LLS_{db}}$. Thus the purpose of including self-loops in $\mathsf{Tbl_G}$ is to allow both edge expansion and no expansion, with the hope to generate more accurate distance estimations.

Next we define the bidirectional-expansion $\mathsf{LS_{dbb}}$ in relational algebra as follows.

$$\begin{aligned}\widetilde{\delta}_{\mathsf{LS_{dbb}}}(x,y) \quad \leftarrow \quad & \mathcal{G}_{\min(t1.d+g1.d+g2.d+t2.d)} \\ & (\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=g_1.s} \rho_{g_1}(\mathsf{Tbl_G}) \\ & \bowtie_{g_1.t=g_2.s} \rho_{g_2}(\mathsf{Tbl_G}) \\ & \bowtie_{g_2.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D}))) \qquad (10)\end{aligned}$$

$\mathsf{LS_{dbb}}$ expands the shortest paths from both x and y by one edge respectively through the first and third join operators, and connects the expanded paths by the second join operator. It is similar to the memory based local search with the search width $c = 1$. We do not consider further edge expansions, as that would lead to an explosive number of joined tuples.

Here we use one example to illustrate the local search process unidirectional-expansion and bidirectional-expansion.



(a) $\mathsf{LS_{dbu}}$ (b) $\mathsf{LS_{dbb}}$

Fig. 9. unidirectional-expansion and bidirectional-expansion

Example 4: $\mathsf{LS_{dbu}}$ performs unidirectional-expansion and evaluates all possible paths between a and b in Figure 9(a). $\mathsf{LS_{dbb}}$, on the other hand, performs bidirectional-expansion and evaluates all possible paths in Figure 9(b). The yellow shade in figures illustrates (in an abstract way) that each node in the dashed path is also expanded to include its 1-hop neighbor. In this example, $\mathsf{LS_{dbb}}$ can find the true shortest path $P(a,b) = (a,e,f,g,b)$. The search space of $\mathsf{LS_{dbu}}$ is a subset of that of $\mathsf{LS_{dbb}}$, thus $\mathsf{LS_{dbu}}$ is less accurate than $\mathsf{LS_{dbb}}$ but with shorter response time.

$\mathsf{LS_{dbb}}$ contains three join operations. The built-in query optimization in RDB may generate a query plan by specifying a join order. It may reduce the intermediate joined results only, but cannot reduce the total number of final joined tuples before aggregation. The final joined results can be huge, thus cause an unacceptable query response time. Let us see an example as follows.

Example 5: Figure 10 shows three tables t_1, g_1 and $t_1 \bowtie_{t_1.t=g_1.s} g_1$. t_1 contains shortest paths between x and nodes $a_1, a_2, \ldots, a_p$, each of which can be joined with a tuple in g_1 leading to the same node m. Thus the join result contains p tuples denoting paths between x and m with different distances $t_1.d + g_1.d$ (the distance column $t_1.d + g_1.d$ is omitted in Figure 10 due to lack of space). Similarly, Figure 11 shows tables g_2, t_2 and $g_2 \bowtie_{g_2.t=t_2.t} t_2$. The join result contains q tuples denoting paths between m and y with different distances $g_2.d + t_2.d$ (the distance column $g_2.d + t_2.d$ is omitted in Figure 11 due to lack of space).

When we join these two intermediate results using the condition $g_1.t = g_2.s$, we totally get $p \times q$ tuples denoting different paths between x and y. By aggregation on $\min(t_1.d + g_1.d + g_2.d + t_2.d)$, only one tuple with the smallest distance will be returned. Thus all but one of these tuples are a waste of effort. We should try to reduce the intermediate result size before the final join.

s	t	d
x	a_1	d_1
x	a_2	d_2
...	...	...
x	a_p	d_p

t_1

s	t	d
a_1	m	d'_1
a_2	m	d'_2
...	...	...
a_p	m	d'_p

g_1

$t_1.s$	$t_1.t$	$g_1.t$
x	a_1	m
x	a_2	m
...	...	...
x	a_p	m

$t_1 \bowtie_{t_1.t=g_1.s} g_1$

Fig. 10. Table t_1 Joins g_1

s	t	d
m	b_1	e_1
m	b_2	e_2
...	...	...
m	b_q	e_q

g_2

s	t	d
y	b_1	e'_1
y	b_2	e'_2
...	...	...
y	b_q	e'_q

t_2

$g_2.s$	$g_2.t$	$t_2.s$
m	b_1	y
m	b_2	y
...	...	...
m	b_q	y

$g_2 \bowtie_{g_2.t=t_2.t} t_2$

Fig. 11. Table g_2 Joins t_2

Optimization of $\mathsf{LS_{dbb}}$: With careful analysis, we optimize $\mathsf{LS_{dbb}}$ as follows:

$$\begin{aligned}\widetilde{\delta}_{\mathsf{LS_{dbb}}}(x,y) \quad \leftarrow \quad & \mathcal{G}_{\min(d1+t2.d+g2.d)} \\ & ((_{g_1.t}\mathcal{G}_{\min(t_1.d+g_1.d)} \textbf{ as } d1 \\ & (\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=g_1.s} \rho_{g_1}(\mathsf{Tbl_G}))) \\ & \bowtie_{g_1.t=g_2.s} (\rho_{g_2}(\mathsf{Tbl_G}) \\ & \bowtie_{g_2.t=t_2.t} \rho_{t_2}(\sigma_{s=y}(\mathsf{Tbl_D})))) \qquad (11)\end{aligned}$$

The major optimization is the early evaluation on $\rho_{t_1}(\sigma_{s=x}(\mathsf{Tbl_D})) \bowtie_{t_1.t=g_1.s} \rho_{g_1}(\mathsf{Tbl_G})$ by the inner aggregation operation $_{g_1.t}\mathcal{G}_{\min(t_1.d+g_1.d)}$. The aggregation operation groups the joined tuples by their destination $g_1.t$, and for each distinct $g_1.t$, the tuples with non-minimum distance on $t_1.d + g_1.d$ are eliminated, as these tuples cannot lead to a path with the minimum distance, when further joining with g_2 and t_2. In Figure 10, this means we only keep one tuple $\langle x, a_i, m, d_i + d'_i\rangle$ for m in the joined table where $d_i + d'_i$ is the minimum among all tuples. This tuple will join with tuples in table $g_2 \bowtie_{g_2.t=t_2.t} t_2$ in Figure 11 and produce only q joined tuples, instead of $p \times q$. Finally the outer aggregation operation will return the tuple with the minimum distance on $(t_1.d + g_1.d + g_2.d + t_2.d)$. This optimization greatly boosts the query efficiency by 5 to 70 times in our experiment.

Interestingly, we do not apply the same inner aggregation on the join results of $g_2 \bowtie_{g_2.t=t_2.t} t_2$. Actually we have tested that possibility and it turns out to be slower than the current version. The reason is when one side is aggregated, the join becomes an injection; then aggregating the other side will not reduce the number of tuples generated/evaluated but increase the overhead of inner aggregation evaluation.

5.2.3 *Accuracy Analysis*

In terms of the distance estimation accuracy, we have

$$\widetilde{\delta}_{\mathsf{LS_{dbb}}}(x,y) \leq \widetilde{\delta}_{\mathsf{LS_{dbu}}}(x,y) \leq \widetilde{\delta}_{\mathsf{LLS_{db}}}(x,y)$$

The proof can be sketched as follows. $\mathsf{LS_{dbu}}$ can subsume $\mathsf{LLS_{db}}$ by joining two tuples $\langle x, v, \delta(x,v)\rangle$, $\langle y, v, \delta(y,v)\rangle$ in $\mathsf{Tbl_D}$, for an arbitrary $v \in V$, through a self-loop tuple $\langle v, v, 0\rangle$, which generates an estimated distance $\delta(x,v) + 0 + \delta(y,v)$. Similarly, $\mathsf{LS_{dbb}}$ subsumes $\mathsf{LLS_{db}}$ through joining the above two tuples with a self-loop twice, i.e., joining $\langle x, v, \delta(x,v)\rangle$, $\langle v, v, 0\rangle$, $\langle v, v, 0\rangle$ with $\langle y, v, \delta(y,v)\rangle$. $\mathsf{LS_{dbb}}$ subsumes $\mathsf{LS_{dbu}}$ through joining with a self-loop and a graph edge, i.e., joining $\langle x, v, \delta(x,v)\rangle$, $\langle v, v, 0\rangle$, $\langle v, u, \delta(v,u)\rangle$, with $\langle y, u, \delta(y,u)\rangle$. Based on the aggregation operator $\mathcal{G}_{\min}$, we prove the result. But in terms of query complexity, $\mathsf{LS_{dbb}}$ is the highest, and $\mathsf{LLS_{db}}$ is the lowest. The three RDB approaches $\mathsf{LLS_{db}}$, $\mathsf{LS_{dbu}}$ and $\mathsf{LS_{dbb}}$ are trade-offs between query time and accuracy. A user can choose one method based on his needs.

6 Experiment

We compare our query-dependent local landmark scheme with global landmark embedding. We present extensive experimental results in terms of accuracy, query efficiency and index size on six large networks. All algorithms were implemented in C++ and tested on a Windows server using one 2.67 GHz CPU and 128 GB memory.

6.1 Dataset Description

We use four social networks or webgraphs: Slashdot[1], Google Webgraph[2], Youtube [18], and Flickr [18], and two road networks: NYRN[3] and USARN[3]. Table 1 lists the network statistics. $|V|$ and $|E|$ represent the node and edge numbers in the original graph, while $|V'|$ and $|E'|$ represent the numbers in the compressed graph. As we can see, our proposed graph compression technique effectively reduces the node number by $38\% - 73\%$. Our embedding index is constructed on the compressed graph. We also sample $10,000$ node pairs in each network and show the average shortest distance $\overline{\delta}$.

1. http://snap.stanford.edu/data/soc-Slashdot0902.html
2. http://snap.stanford.edu/data/web-Google.html
3. http://www.dis.uniroma1.it/∼challenge9/download.shtml

TABLE 1
Network Statistics

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V'\rvert$	$\lvert E'\rvert$	$\overline{\delta}$
Slashdot	77,360	905,468	36,012	752,478	4.1146
Google	875,713	5,105,039	449,341	4,621,002	7.4607
Youtube	1,157,827	4,945,382	313,479	4,082,046	5.3317
Flickr	1,846,198	22,613,981	493,525	18,470,294	5.5439
NYRN	264,346	733,846	164,843	532,264	27km
USARN	23,947,347	58,333,344	7,911,536	24,882,476	1522km

6.2 Comparison Methods and Metrics

We compare the following embedding methods: (1) Global Landmark Scheme (GLS), (2) Local Landmark Scheme (LLS) and (3) Local Search (LS) with $c = 1$. For global landmark selection, we use random selection and closeness centrality based selection [12]. We use two landmark set sizes $k = 20$ and $k = 50$ in our experiments.

We use the relative error $\frac{|\widetilde{\delta}(s,t)-\delta(s,t)|}{\delta(s,t)}$ to evaluate the quality of an estimated distance for a query (s,t). As it is expensive to exhaustively test all node pairs in a large network, we randomly sample $10,000$ node pairs in each graph as queries and compute the average relative error on the sample set. In addition, we test the query processing time and the embedding index size.

In the following, we first compare the memory based implementations of different methods, and then the relational database based implementations.

6.3 Memory-based Implementations

6.3.1 *Average Relative Error*

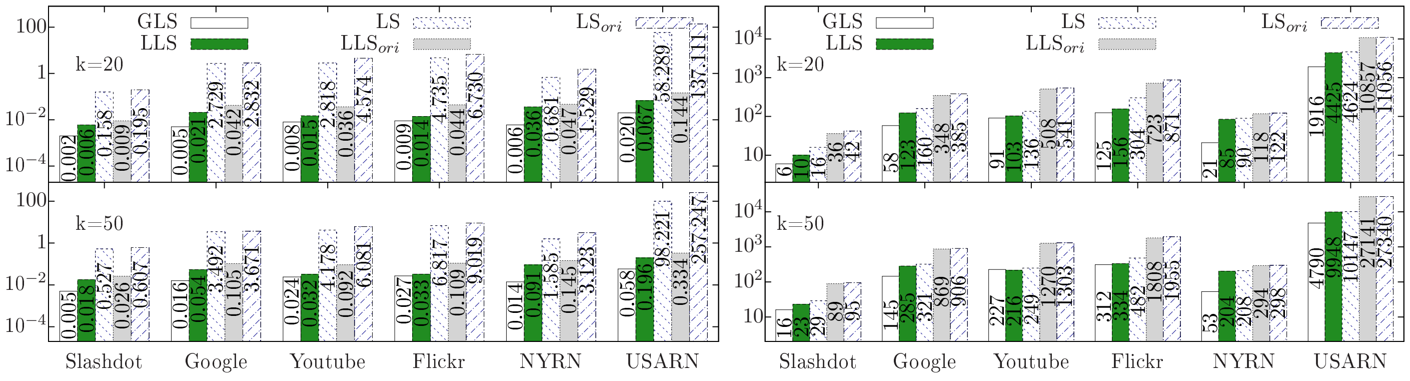
Table 2 shows the average relative error (AvgErr) of GLS, LLS and LS with different global landmark sets selected by Random and Centrality.

LLS reduces the AvgErr of GLS by a large margin in all cases. Under Random landmark selection strategy, the AvgErr of LLS is one order of magnitude smaller than that of GLS on most graphs; while under Centrality, LLS reduces the AvgErr by 40% compared with GLS on average. Furthermore, in most cases the AvgErr of LLS with $k = 20$ landmarks is even lower than that of GLS with $k = 50$ landmarks, and at the same time, LLS ($k = 20$) has a smaller embedding index size than GLS ($k = 50$) (see the GLS and LLS bars in Figure 12(b)). This result demonstrates that selecting more global landmarks for GLS (e.g., $k = 50$) and using more index space do not necessarily achieve a better estimation accuracy than LLS ($k = 20$). Thus simply selecting more landmarks for GLS may not be an effective solution, as the main bottleneck of GLS is caused by the query-independent landmark embedding.

LS achieves the best performance in all cases. Its AvgErr is between 0 and the scale of 10^{-3} in most cases. In particular, in social networks the average distance is usually very small, according to the famous rule of “six degrees of separation”. Thus by a local search with 1-hop expansion, the expanded trees rooted at both query nodes are very likely to intersect, which helps to find a very short path, or even the shortest path, between the query nodes.

TABLE 2
Average Relative Error

		$k=20$						$k=50$					
		SlashD	Google	Youtube	Flickr	NYRN	USARN	SlashD	Google	Youtube	Flickr	NYRN	USARN
Random	GLS	0.6309	0.5072	0.6346	0.5131	0.1825	0.1121	0.4535	0.4750	0.4549	0.4559	0.1188	0.0632
	LLS	0.1423	0.0321	0.0637	0.0814	0.0246	0.0786	0.0727	0.0142	0.0391	0.0444	0.0103	0.0241
	LS	0.0000	0.0046	0.0009	0.0001	0.0071	0.0090	0.0000	0.0022	0.0003	0.0001	0.0042	0.0030
Centrality	GLS	0.1520	0.0426	0.0595	0.0567	0.6458	1.5599	0.1385	0.0245	0.0461	0.0524	0.6133	0.7422
	LLS	0.1043	0.0290	0.0489	0.0503	0.1536	0.4708	0.0663	0.0140	0.0334	0.0284	0.1533	0.4505
	LS	0.0001	0.0074	0.0010	0.0003	0.1479	0.4703	0.0000	0.0037	0.0005	0.0000	0.1455	0.4483



(a) Online Query Time in Milliseconds

(b) Index Size in MB

Fig. 12. Query Time and Index Size Comparison

One point worth noting is that, the state-of-the-art techniques for computing shortest paths and shortest distances on road networks have achieved controlled error rate and low complexities, with the aid of coordinates [19], [20]. As LLS and LS are designed for general graphs, the performance improvement by our methods is more significant on social networks than on road networks.

6.3.2 Online Query Time

Figure 12(a) shows the query time in milliseconds of different methods in log scale – GLS, LLS and LS on the compressed graph, LLS$_{ori}$ and LS$_{ori}$ on the original uncompressed graph. According to our analysis in Section 3.3, the online query complexity is $O(|S|)$ for both GLS and LLS. Even in the largest graph USARN with 24 million nodes, it only costs 0.196 milliseconds for LLS to process one query when $k = 50$. In most cases, LLS is $2-4$ times slower than GLS, which is a very small factor.

As LS performs online tree expansion and search in query processing, the query time largely depends on the network size. For example, it costs 0.158 milliseconds to process one query in Slashdot, but 58 milliseconds in USARN when $k = 20$, as the node number of USARN is about 310 times larger than that of Slashdot.

We can also observe that LLS reduces the query time of LLS$_{ori}$ by 23% – 70%. The main reason is that graph compression reduces the index size, thus increases the locality of memory access and reduces the amortized time per memory access. Similarly, LS reduces the query time of LS$_{ori}$ by 32% on average, as the graph compression prevents LS from expanding the partial tree to tree/chain nodes unnecessarily.

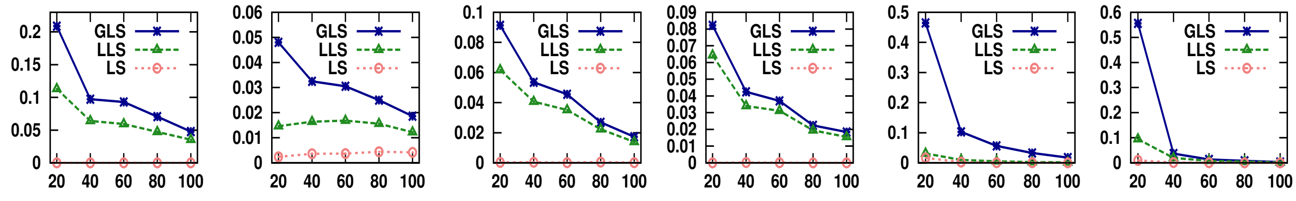
6.3.3 Index Size

Figure 12(b) shows the index size in MB of different methods in log scale – GLS, LLS and LS on the compressed graph, LLS$_{ori}$ and LS$_{ori}$ on the original uncompressed graph. The index size of LLS is about 2 times that of GLS, as LLS needs to store extra information including shortest path trees and RMQ index tables. LS uses a little extra space compared with LLS, as LS needs to store the original graph in memory for edge expansion and local search. Nevertheless, we can see that our LLS and LS methods use moderate index sizes even for very large networks, e.g., an index of 10 GB (when $k = 50$) on USARN with 24 million nodes. We can also observe that the index size of LLS and LS is significantly smaller than that of LLS$_{ori}$ and LS$_{ori}$, and it is reduced by 63% on average, which is consistent with the graph compression ratio in Table 1.

In terms of the index construction time of LLS, it takes less than 1 minute in most cases, while the longest one takes 354.7 seconds on USARN when $k = 50$.

6.3.4 Distance Sensitive Relative Error

We evaluate the performance of GLS, LLS and LS on queries in different shortest distance ranges. For each network, we sort the 10,000 sample queries in the increasing order of their actual shortest distances and find the 20th, 40th, 60th, 80th and 100th percentiles of the shortest distance. Based on this, we partition the 10,000 queries into 5 intervals, each containing 20 percent of the queries. We evaluate the AvgErr in Figure 13 for queries whose shortest distances fall into the five intervals respectively. For each network we adopt the best landmark selection heuristic, i.e., Centrality for social networks and Random for road networks. We set $k = 50$.



(a) Slashdot (b) Google (c) Youtube (d) Flickr (e) NYRN (f) USARN

Fig. 13. Average Relative Error on Queries with Shortest Distance in Different Ranges

We observe that LLS outperforms GLS in all distance ranges on all networks. The improvement is most significant for queries whose shortest distances are within the 20th percentile. This demonstrates that LLS can provide very accurate estimation for nearby query nodes while GLS cannot. In particular, we observe that on the two road networks, the improvement of LLS over GLS within the 20th percentile is about 10 times or more, which is very substantial. This is because the road networks have large diameters. If two query nodes are nearby but distant from global landmarks, GLS will provide a very inaccurate distance estimation. In contrast, social networks usually have a fairly small diameter, which guarantees that the global landmarks will not be too far from the query nodes. Therefore, on the social networks, the improvement by LLS is around 2 times within the 20th percentile.

The performance of LS remains very stable in all distance ranges. The AvgErr of LS is zero or close to zero on most networks. We also observe that, on dense networks with large average degrees, e.g., Slashdot and Flickr, local search with neighbor expansion is particularly effective in finding the (nearly) shortest paths. The AvgErr of LS on Slashdot is zero in all distance ranges.

6.4 Relational Database based Implementations

In this experiment, we study the performance of local landmark scheme implemented on relational database. Specifically we compare LLS_{db}, LS_{dbu} and LS_{dbb} and report relative error, query time, index construction time and index size. 20 randomly selected global landmarks are used. We use Oracle Database 11g, Edition release 11.2.0.1.0 and connect to it through ODBC interface. The disk quota for Oracle system is 100 GB. We find in our experiments that the index size for USARN exceeds the 100 GB disk quota, so we only report results on the other five networks.

6.4.1 *Average Relative Error*

Figure 14 shows the average relative error of LLS_{db}, LS_{dbu} and LS_{dbb} on five networks. LS_{dbb} has the most accurate estimation. It has zero error on SlashDot and Youtube, and average error at 10^{-4}–10^{-3} scale on the other networks. The online bidirectional search in LS_{dbb} reduces the error of LLS_{db} by 93% on social networks on average. LS_{dbb} also outperforms LS_{dbu}. The reduction of relative error is the most remarkable on SlashDot, Google, and Flickr, i.e., $100\%, 76\%$, and 99%, respectively. In addition, LS_{dbu} has the second best performance. It outperforms LLS_{db} with a reduction of relative error by 76% on social networks on average. Note that the error on NYRN is the same for all three methods. This shows that on a network with a large diameter, local search within 1 or 2-hop neighborhood does not help find a shortcut. But on social networks with much smaller diameters, local search can reduce the relative error substantially.

In Section 5.1.3 we have proved that LLS_{db} has better precision than LLS_{mem}. When we compare the average error of LLS_{db} and LLS_{mem} (in Table 2, under 20 random global landmarks), we find LLS_{db} reduces the relative error by 31% on average. In particular, on NYRN dataset, the average error of LLS_{db} is only $\frac{1}{3}$ that of LLS_{mem}. This result verifies that LLS_{db} is more accurate than LLS_{mem}.

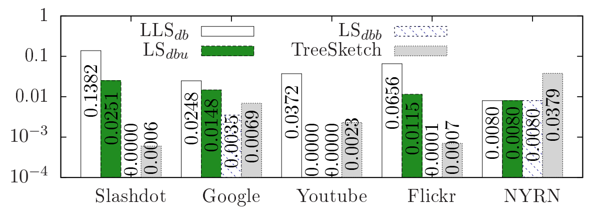


Fig. 14. Average Relative Error of RDB Algorithms

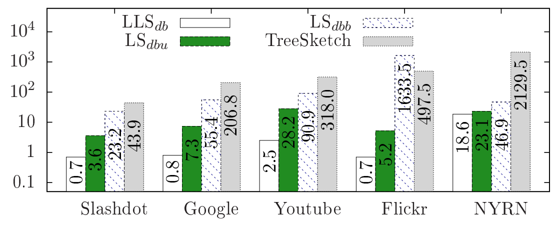


Fig. 15. Online Query Time (ms) of RDB Algorithms

6.4.2 *Online Query Time*

We report the query time of the three approaches in Figure 15 in milliseconds. LLS_{db} uses the least query time. The query time of LS_{dbu} is 6.67 times that of LLS_{db} on average. As a disk-based solution with fairly good precision, LS_{dbu} only uses 13.5 ms query time on average on the five networks, which is very cost effective. As for LS_{dbb}, it spends 8 times longer query time than LS_{dbu} on most networks. Still it spends less than 90.9 ms on 4 out of 5 graphs for query processing, which is also quite small for a disk-based algorithm.

6.4.3 Index Size

Figure 16 reports the index size on the disk in MB. $\mathsf{LS_{dbu}}$ and $\mathsf{LS_{dbb}}$ use the same disk space since both of them use tables $\mathsf{Tbl_D}$ and $\mathsf{Tbl_G}$, thus we use $\mathsf{LS_{db}}$ to represent both of them. $\mathsf{LS_{db}}$ uses slightly more space than $\mathsf{LLS_{db}}$ since it has to store the $\mathsf{Tbl_G}$ table. The index size of all social networks are within 1 GB in most cases. However, the index size of NYRN is as large as 8.9 GB although it is a small network with only 264K nodes. This is because the diameter of NYRN is large while that of social networks is a small value (less than 8 on the networks in our experiment). The index time of $\mathsf{LLS_{db}}$ and $\mathsf{LS_{db}}$ shows a similar trend with their index size, as almost all the index time is spent on database insertion operations on $\mathsf{Tbl_D}$ and $\mathsf{Tbl_G}$.

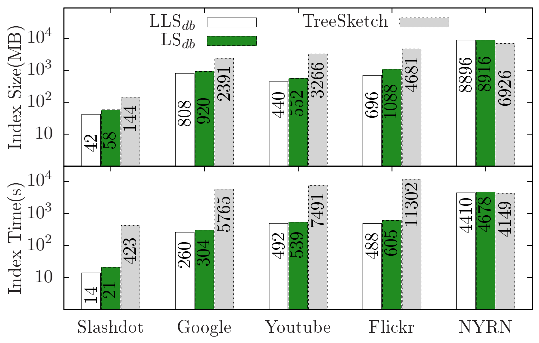


Fig. 16. Index Size and Index Time of RDB Algorithms

6.4.4 Comparison with TreeSketch

In this experiment, we compare $\mathsf{LS_{dbb}}$ with TreeSketch [14], a sketch-based method implemented in RDF graph database provided by the authors. Figure 14 shows that $\mathsf{LS_{dbb}}$ reduces the average error of TreeSketch by a large margin. $\mathsf{LS_{dbb}}$ has zero error on SlashDot and Youtube. On the other three networks, the average error of $\mathsf{LS_{dbb}}$ is 2–7 times smaller than that of TreeSketch. Figure 15 shows that the query time of $\mathsf{LS_{dbb}}$ is 2–45 times shorter than that of TreeSketch on four out of five graphs. Furthermore, Figure 16 shows the index time and index size of both $\mathsf{LS_{dbb}}$ (denoted as $\mathsf{LS_{db}}$) and TreeSketch, from which we can see $\mathsf{LS_{dbb}}$ uses much smaller index size and shorter index time on most graphs than TreeSketch. One reason for the smaller index size is that we apply our graph compression technique in $\mathsf{LS_{dbb}}$, but TreeSketch does not have the compression. We also find the implementation of TreeSketch incurs an unnecessary index overhead due to the way RDF3x constructs all possible index configurations which are not really needed for the shortest path estimation. In conclusion, $\mathsf{LS_{dbb}}$ outperforms TreeSketch in all aspects.

7 Related Work

Graph embedding techniques have been widely used to estimate the distance between two nodes in a graph in many applications including road networks [7], [11], social networks and web graphs [10], [12], [13], [14], [15] as well as the Internet [5], [6]. Shahabi et al. [7] utilize Linial, London and Robinovich (LLR) embedding to estimate the distance between two nodes. Kriegel et al. [11] propose a hierarchical reference node embedding approach which organizes reference nodes in multiple levels for a better scalability. Potamias et al. [12] formulate the reference node selection problem to selecting nodes with high betweenness centrality. [5] proposes an architecture, called IDMaps, which measures and disseminates distance information on the global Internet. [8] defines a notion of slack – a certain fraction of all distances that may be arbitrarily distorted as a performance guarantee based on randomly selected reference nodes. [9] and its follow-up studies [13], [14] provide a $(2k-1)$-approximate distance estimation with $O(kn^{1+1/k})$ memory for any integer $k \geq 1$. To summarize, the major differences between the above methods lie in the following aspects: (1) landmark selection – some [8], [9], [10], [13], [14] select landmarks randomly, while others [5], [6], [11], [12] use heuristics; (2) landmark organization – some methods organize landmarks in multiple levels [9], [11], [13], [14], while other methods use a flat landmark embedding; and (3) an error bound or not – [8], [9], [13], [14], [15] analyze the error bound of the estimated distances, while most of the other methods have no error bounds or guarantees of the estimated distances.

There have been a lot of studies on computing shortest paths and processing k-nearest neighbor queries in spatial networks. Papadias et al. [21] use the Euclidean distance as a lower bound to prune the search space and guide the network expansion for refinement. Kolahdouzan and Shahabi [22] propose to use first order Voronoi diagram to answer KNN queries in spatial networks. Jagadish et al. [23] compress high-dimensional data points to one-dimensional values based on a set of well selected reference nodes and then apply range search using a B^+ tree index to answer KNN queries. Hu et al. [24] propose an index, called distance signature, which associates approximate distances from one object to all the other objects in the network, for distance computation and query processing. Samet et al. [25] build a shortest path quadtree to support k-nearest neighbor queries in spatial networks. For a spatial network of dimension d, [19], [20] can retrieve an ε-approximation distance estimation in $O(\log n)$ time using an index termed path-distance oracle of size $O(n \cdot \max(s^d, \frac{1}{\epsilon}^d))$. [26] proposes TEDI, an indexing and query processing scheme for the shortest path query based on tree decomposition. Based on incremental construction of a shortest path tree, [27] monitors a type of nearest neighbor query, called continuous detour query (CDQ), on road networks.

8 Conclusions

In this paper, we propose a novel shortest path tree based local landmark scheme, which finds a node close to the query nodes as a query-specific local landmark for a triangulation based shortest distance estimation. Specifically, a local landmark is defined as the LCA of the query nodes in a shortest path tree rooted at a global landmark. Efficient algorithms for indexing and retrieving LCAs are introduced, which achieve low offline indexing complexity and online query complexity. This strategy significantly reduces the distance estimation error, compared with global landmark embedding. We also study the local landmark scheme on relational database for better scalability. Extensive experimental results on large-scale social networks and road networks demonstrate the effectiveness and efficiency of the proposed local landmark scheme.

Acknowledgment: This work is supported by the Hong Kong Research Grants Council General Research Fund (GRF) Project No. CUHK 411211, 411310, 419109 and 418512.

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Systems Science and Cybernetics SSC4*, vol. 4, no. 2, pp. 100–107, 1968.

[3] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," in *SODA*, 2005, pp. 156–165.

[4] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Reach for A*: Efficient point-to-point shortest path algorithms," in *Workshop on Algorithm Engineering and Experiments*, 2006, pp. 129–143.

[5] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, "IDMaps: A global internet host distance estimation service," *IEEE/ACM Trans. Networking*, vol. 9, no. 5, pp. 525–540, 2001.

[6] T. S. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *INFOCOM*, 2002, pp. 170–179.

[7] C. Shahabi, M. Kolahdouzan, and M. Sharifzadeh, "A road network embedding technique for k-nearest neighbor search in moving object databases," in *GIS*, 2002, pp. 94–100.

[8] J. Kleinberg, A. Slivkins, and T. Wexler, "Triangulation and embedding using small sets of beacons," in *FOCS*, 2004, pp. 444–453.

[9] M. Thorup and U. Zwick, "Approximate distance oracles," *Journal of the ACM*, vol. 52, no. 1, pp. 1–24, 2005.

[10] M. J. Rattigan, M. Maier, and D. Jensen, "Using structure indices for efficient approximation of network properties," in *KDD*, 2006, pp. 357–366.

[11] 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.

[12] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *CIKM*, 2009, pp. 867–876.

[13] A. D. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy, "A sketch-based distance oracle for web-scale graphs," in *WSDM*, 2010, pp. 401–410.

[14] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum, "Fast and accurate estimation of shortest paths in large graphs," in *CIKM*, 2010.

[15] M. Qiao, H. Cheng, and J. X. Yu, "Querying shortest path distance with bounded errors in large graphs," in *SSDBM*, 2011.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[17] M. A. Bender and M. Farach-Colton, "The LCA problem revisited," in *LATIN 2000: Theoretical Informatics*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, vol. 1776, pp. 88–94.

[18] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *IMC*, 2007, pp. 29–42.

[19] J. Sankaranarayanan, H. Samet, and H. Alborzi, "Path oracles for spatial networks," *PVLDB*, pp. 1210–1221, 2009.

[20] J. Sankaranarayanan and H. Samet, "Distance oracles for spatial networks," in *ICDE*, 2009, pp. 652–663.

[21] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network database," in *VLDB*, 2003, pp. 802–813.

[22] M. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004, pp. 840–851.

[23] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, R. Zhang, "iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search," *TODS*, vol. 30, issue 2, pp. 364–397, 2005.

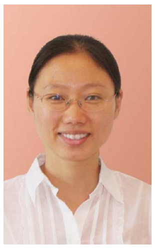
[24] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance indexing on road networks," in *VLDB*, 2006, pp. 894–905.

[25] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *SIGMOD*, 2008, pp. 43–54.

[26] F. Wei, "TEDI: Efficient shortest path query answering on graphs," in *SIGMOD*, 2010, pp. 99–110.

[27] S. Nutanong, E. Tanin, J. Shao, R. Zhang, K. Ramamohanarao, "Continuous Detour Queries in Spatial Networks," *TKDE*, vol. 24, no. 7, pp. 1201–1215, 2012.

Miao Qiao Miao Qiao received her BE in computer science and technology from Shanghai Jiao Tong University in 2009. She is currently a Ph.D. student in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. Her major research interests include large-scale graph indexing and query, and graph algorithms in relational databases.



Hong Cheng Hong Cheng is an Assistant Professor in the Department of Systems Engineering and Engineering Management at the Chinese University of Hong Kong. She received her Ph.D. degree from University of Illinois at Urbana-Champaign in 2008. Her research interests include data mining, database systems, and machine learning. She received research paper awards at ICDE'07, SIGKDD'06 and SIGKDD'05, and the certificate of recognition for the 2009 SIGKDD Doctoral Dissertation Award. She is a recipient of the 2010 Vice-Chancellor's Exemplary Teaching Award at the Chinese University of Hong Kong.

Lijun Chang Lijun Chang received his B.Eng. in computer science and technology from Renmin University of China in 2007, and Ph.D. in Systems Engineering and Engineering Management from Chinese University of Hong Kong in 2011. He is currently a postdoctoral research fellow at Chinese University of Hong Kong. His research interests include graph exploration, uncertain data management, and keyword search.

Jeffrey Xu Yu Jeffrey Xu Yu received the BE, ME, and PhD degrees in computer science, from the University of Tsukuba, Japan, in 1985, 1987, and 1990, respectively. Currently he is a professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include graph mining, graph database, keyword search, and query processing and optimization. He is a senior member of the IEEE, a member of the IEEE Computer Society, and a member of ACM.