Finding Top-k Optimal Sequenced Routes
—Full Version

Huiping Liu*, Cheqing Jin*, Bin Yang†, Aoying Zhou*

hlui@stu.ecnu.edu.cn  {cqjin, ayzhou}@dase.ecnu.edu.cn  byang@cs.aau.dk

†Department of Computer Science, Aalborg University, Denmark

*School of Data Science and Engineering, East China Normal University, China

Abstract—Motivated by many practical applications in logistics and mobility-as-a-service, we study the top-k optimal sequenced routes (KOSR) querying on large, general graphs where the edge weights may not satisfy the triangle inequality, e.g., road network graphs with travel times as edge weights. The KOSR querying strives to find the top-k optimal routes (i.e., with the top-k minimal total costs) from a given source to a given destination, which must visit a number of vertices with specific vertex categories (e.g., gas stations, restaurants, and shopping malls) in a particular order (e.g., visiting gas stations before restaurants and then shopping malls).

To efficiently find the top-k optimal sequenced routes, we propose two algorithms PruningKOSR and StarKOSR. In PruningKOSR, we define a dominance relationship between two partially-explored routes. The partially-explored routes that can be dominated by other partially-explored routes are postponed, which leads to a smaller searching space and thus improves efficiency. In StarKOSR, we further improve the efficiency by extending routes in an A* manner. With the help of a judiciously designed heuristic estimation that works for general graphs, the cost of partially explored routes to the destination can be estimated such that the qualified complete routes can be found early. In addition, we demonstrate the high extensibility of the proposed algorithms by incorporating Hop Labeling, an effective label indexing technique for shortest path queries, to further improve efficiency. Extensive experiments on multiple real-world graphs demonstrate that the proposed methods significantly outperform the baseline method. Furthermore, when k = 1, StarKOSR also outperforms the state-of-the-art method for the optimal sequenced route queries.

This is a full version of “Finding Top-k Optimal Sequenced Routes” [20], to appear in IEEE ICDE 2018.

I. INTRODUCTION

Optimal sequenced route (OSR) querying [32, 33], a.k.a., generalized shortest path querying [29], aims at finding a route with minimum total cost (e.g., travel distance or travel time), passing through a number of vertex categories (e.g., restaurants, banks, gas stations) in a particular order (e.g., visiting banks before restaurants). This problem has many practical applications in route planning [14, 19], crisis management, supply chain management, video surveillance, mobility-as-a-service [13], and logistics [29, 32]. However, it is often the case that the optimal sequenced route with the minimum total cost may not be the best choice for all users since different users may have different personal preferences [10, 28, 36].

Consider the example shown in Figure 1, where a vertex represents a point-of-interest and is associated with a category, e.g., shopping mall (MA), restaurant (RE), or cinema (CI) and edge weights represent travel costs, e.g., travel time or fuel consumption. Suppose that Alice plans a trip which starts from location s and wishes passing through a shopping mall, a restaurant, and then a cinema and finally reaching destination t. This plan can be formalized with an OSR query with category sequence (MA, RE, CI). The optimal sequenced route for Alice is s → a → b → d → t with a cost of 20. However, if Alice prefers restaurant e to restaurant b, route s → a → e → d → t with a cost of 21 is more preferable. In addition, if the shopping mall at vertex c has sale promotions, route s → c → b → d → t with a cost of 22 can also be a good candidate. In these cases, returning only the optimal sequenced route may not sufficiently satisfy users’ varying preferences. This motivates us to study the top-k optimal sequenced routes (KOSR) querying that returns k routes that satisfy the given category order and have the k least total costs.

In this paper, we focus on finding the top-k optimal sequenced routes in general graphs, where edge weights may not satisfy triangle inequality. Unfortunately, the KOSR problem on general graphs has not been addressed carefully before, though the OSR problem has been extensively studied. In [32], the progressive neighbor exploration algorithm PNE is proposed to solve the OSR problem on general graphs. In [29], a dynamic programming based algorithm GSP is formulated, which outperforms PNE significantly and is considered as the state-of-the-art for solving the OSR problem on general graphs.

However, by simply extending existing solutions for the OSR problem, it is unlikely, if it is not impossible, to achieve efficient solutions for the KOSR problem. In particular, dynamic programming based GSP is unable to be extended to solve the KOSR problem due to lack of sufficient information for other sequenced routes. Although PNE can be extended to handle the KOSR problem by iteratively finding the next optimal sequenced route, the efficiency is low since all partially explored sequenced routes whose costs are less than the cost of the k-th optimal sequenced route must be examined, whereas
most of them can be avoided being extended.

It is non-trivial to devise an efficient solution for solving KOSR due to two challenges. The first is how to filter unnecessary partially explored sequenced routes when exploring the graph. To conquer this challenge, we propose a dominance relationship between two partially explored sequenced routes \( r \) and \( r' \). If \( r \) dominates \( r' \), the optimal (i.e., least-cost) feasible sequenced route extended from \( r \) is always better than that of \( r' \). Thus, the exploring of routes that are extended from \( r' \) can be postponed until a complete sequenced route extended from \( r \) occurs in the result set. Furthermore, inspired by A* algorithm \([21]\), we estimate the cost of each partially explored sequenced route to the destination, and explore the partially explored routes according to their estimated total costs, which further reduces the searching space.

The second challenge is how to efficiently find the \( i \)-th nearest, not merely the nearest, neighbor in a category, as this operation is invoked frequently when solving KOSR. For example, recall that we may want to recommend the top-3 optimal sequenced routes to Alice in Figure 1. More than one nearest neighbors in category MA for vertex \( s \), i.e., \( a \) and \( c \), are required to be explored. A simple and intuitive implementation of the operation is to apply Dijkstra’s algorithm, which is however very costly. To overcome this weakness, we build an inverted index for each category by employing hop labeling technique \([11, 2, 4, 5, 9]\) on the original graph in an off-line manner. In this way, the \( i \)-th nearest neighbor in a category can be identified efficiently in an on-line manner by simply looking up the inverted label index.

To the best of our knowledge, this is the first comprehensive work to study the KOSR problem. The paper makes four contributions. First, we propose a dominance relationship between partially explored sequenced routes and develop an algorithm based on the dominance relationship to reduce the searching space significantly when solving the KOSR problem. Second, we propose a heuristic method that is able to estimate the minimal total cost of partially explored sequenced routes, which enables the develop of an \( \text{A}^* \) like algorithm to further reduce the searching space for solving the KOSR problem. Third, we propose an inverted index which facilitates the operation that identifies the \( i \)-th nearest neighbor in a category for a given vertex, which improves the efficiency of both algorithms. Finally, we report on a comprehensive empirical study over different real-world graphs, showing that the proposed algorithms significantly outperform the baseline method for KOSR and the state-of-the-art method for OSR.

### II. RELATED WORK

We categorize relevant studies on sequenced route querying in Table 1. This categorization considers three different aspects. First, we consider whether the algorithms work for general graphs. When edge weights represent Euclidean distances between vertices, the edge weights satisfy the triangle inequality. We call such graphs Euclidean graphs. When edge weights represent other costs such as travel times and fuel consumption \([13, 35]\), the edge weights do not necessarily satisfy triangle inequality anymore. We call such graphs general graphs. Note that Euclidean distance and indexing structures based on the Euclidean space, such as R-trees, cannot be utilized in general graphs. The proposed algorithms in this paper work for general graphs. Second, we consider whether the algorithms support returning the top-\( k \) optimal sequenced routes. Most existing studies only work for the case when only the top-1 optimal sequenced route is required. Third, we consider whether a specific category order is given.

Table 1. Categorization of Sequenced Route Queries

| Euclidean Graphs | General Graphs |
|------------------|----------------|
| \text{Specific order:} & [12] \text{Specific order:} & [29, 32, 33] |
| \text{Arbitrary order:} & [8, 26, 27] \text{Arbitrary order:} & [7, 30] |
| \text{Specific order:} & [22, 25, 31] \text{Arbitrary order:} & 0 |
| \text{Specific order:} & This paper \text{Arbitrary order:} & 0 |

The optimal sequenced route querying \([22, 33]\), a.k.a., the generalized shortest path querying \([29]\), is the most relevant problem. \([32]\) is the first work that addresses the problem, in which three algorithms are proposed, namely LORD, R-LORD and PNE. The first two algorithms, LORD and R-LORD, are designed for edge weights in Euclidean spaces where R-trees can be utilized to enable efficient query processing. The PNE algorithm works for general graphs. In this paper, we extend PNE to solve the KOSR problem, which is regarded as the baseline method. \([33]\) tries to improve the efficiency of optimal sequenced route querying on general graphs by preconstructing a series of additively weighted Voronoi diagrams (AWVD). However, this approach requires a prior knowledge of the category sequence in a query, thus limiting its applicability for online queries, because it is prohibitive to pre-construct AWVDs for all possible category sequences. \([29]\) addresses the optimal sequenced route queries on general graphs by using a dynamic programming formulation. In their formulation, the optimal costs of all vertices in each category from the start and passing through all the categories before them are computed by using a transition function between consecutive categories. In their solutions, contraction hierarchy technique \([15]\) is utilized to compute the optimal costs of the vertices in the next category according to above recurrence. Though efficient, this approach cannot be extended to KOSR queries, because the transition function only suits the optimal cost.

Group optimal sequenced routes problem \([22, 23, 31]\) is also relevant to KOSR. Given a group of users with different sources and destinations and a set of ordered categories, group optimal sequenced routes querying aims to find the top-\( k \) optimal sequenced routes that pass through the categories in order and minimize the aggregate travel costs of the group. Specifically, when the group only has one user, then the problem becomes the KOSR problem. However, all existing methods are based on Euclidean space. Thus, they cannot be applied in general graphs.

\([7, 8, 26, 27]\) study the problem on finding the optimal route that visits a given set of categories, but without a specific category order. Sometimes, additional constraints, such as partial order \([8, 27]\) and budget limit \([7]\), are also considered. Such problems are NP-hard and can be reduced to generalized traveling salesman problem \([30]\). Therefore, approximate methods are proposed to solve such problems. Due
to different problem natures, above methods cannot be directly applied for KOSR. Other advanced routing strategies \cite{10, 11, 12, 13, 14}, e.g., skyline routing \cite{17, 18, 19}, stochastic routing \cite{20, 21, 22}, and personalized routing \cite{10, 23}, are also different from KOSR.

III. PRELIMINARIES

We formalize the KOSR problem and introduce baseline solution. Frequent notations are summarized in Table I.

A. Problem Definition

Definition 1 (Graph): A directed weighted graph \( G(V, E, F, W) \) includes a vertex set \( V \) and an edge set \( E \subseteq V \times V \). Category function \( F : V \rightarrow 2^S \) takes as input a vertex \( v \in V \) and returns a set of categories \( F(v) \), where \( S \) denotes a set of all possible categories. Weight function \( W : E \rightarrow R^+ \) takes as input an edge \((u, v)\) and returns a non-negative cost of the edge \( W((u, v)) \), e.g., the travel time when traversing edge \((u, v)\).

For example, in Figure 1 we have \( S = \{ MA, RE, CI \} \), \( F(a) = \{ MA \} \), and \( W((s, a)) = 8 \). Note that the edge weights can be arbitrary and may not satisfy the triangle inequality.

Definition 2 (Route): A route \( P_{s,t} \) from vertex \( s \) to vertex \( t \) in graph \( G \) is a sequence of vertices, where each two adjacent vertices are connected by an edge, denoted by \( P_{s,t} = (v_0 = s, v_1, \ldots, v_q = t) \). Let \( w(P_{s,t}) = \sum_{0 \leq i < q} W((v_i, v_{i+1})) \) be the weight, or cost, of route \( P_{s,t} \) and \( |P_{s,t}| \) be the size of route \( P_{s,t} \) which equals to the number of vertices in route \( P_{s,t} \).

Definition 3 (Category Sequence): A category sequence \( C = \{C_1, C_2, \ldots, C_i\} \) represents an order in which each category must be visited, where each \( C_i \in C \), \( 1 \leq i \leq j \), represents a specific category in category set \( S \), and each \( C_i \) corresponds to a vertex set \( V_{C_i} = \{v \in V \mid C_i \subseteq F(v)\} \). We refer \( |C| \) and \( |C_i| \) to the size of the category sequence \( C \) and the size of \( V_{C_i} \), respectively.

Definition 4 (Feasible Route): Given a source-destination pair \((s, t)\), and a category sequence \( C = \{C_1, C_2, \ldots, C_i\} \), a route \( P_{s,t} = (v_0 = s, v_1, \ldots, v_q = t) \) is feasible if and only if there exists a subsequence of vertices \( \langle v_r, v_{r+1}, \ldots, v_t \rangle \) from \( P_{s,t} \), such that \( 0 < r_1 \leq r_2 \leq \cdots \leq r_j < q \) and for \( 1 \leq i \leq j \), \( v_{r_i} \in V_{C_i} \) or \( C_i \subseteq F(v_{r_i}) \). We call \( (s, v_{r_1}, v_{r_2}, \ldots, v_{r_j}, t) \) the witness\(^1\) of \( P_{s,t} \) w.r.t. category sequence \( C \), denoted as \( P_{s,t,C} \).

In many cases, there exist multiple feasible routes for a given source-destination pair and a category sequence. We distinguish two feasible routes according to their witnesses. This means that if two feasible routes share the same witness \((s, t, C)\) a category sequence, they are regarded as the same feasible route and only the route with smaller cost is considered. Formally, for a witness \( P_{s,t,C} = (v_0, v_1, \ldots, v_q) \), its cost \( w(P_{s,t,C}) \) is defined as \( w(P_{s,t,C}) = \sum_{0 \leq i < q} \text{dis}(v_i, v_{i+1}) \), where \( \text{dis}(v_i, v_{i+1}) \) is the least cost from vertex \( v_i \) to \( v_{i+1} \).

Definition 5 (KOSR query): Given a graph \( G \), the top- \( k \) optimal sequenced routes (KOSR) query is a quad-tuple \((s, t, C, k)\), where \( s, t \in V \) denotes a source-destination pair, \( C \) is a category sequence, and \( k \) is a positive integer. The query returns a set of \( k \) different feasible routes \((s, t, C)\), \( \Psi = \{P_{s,t,1}^1, P_{s,t,2}^2, \ldots, P_{s,t,k}^k\} \), such that there does not exist any other feasible route \( P'_{s,t} \) in \( G \) where \( P'_{s,t} \notin \Psi \wedge w(P'_{s,t}) < \max_{1 \leq i \leq k} w(P_{s,t,C}) \).

Example 1: Consider the graph \( G \) in Figure 1 the KOSR query \((s, t, \langle MA, RE, CI \rangle, 3)\) returns \( \Psi \{\langle s, a, b, d, t \rangle, \langle s, a, c, d, t \rangle, \langle s, c, b, d, t \rangle\} \) that includes routes with costs of \( 20, 21, \) and \( 22 \). There does not exist another feasible path whose cost is smaller than \( 22 \).

To simplify later discussion, we focus on identifying the witnesses of top- \( k \) optimal sequenced routes, rather than identifying the actual routes. However, given the witness, its actual route can be easily reconstructed. For simplicity, all routes we discuss in the following sections refer to witnesses unless stated otherwise. Moreover, given a category sequence \( C \), we introduce two dummy categories \( C_0 = \{s\} \) and \( C_{|C|+1} = \{t\} \) to include the source vertex \( s \) and destination vertex \( t \).

B. Baseline Solution

Since OSR can be considered as a special case of KOSR where \( k \) is set to 1, we present PNE and GSP, two state-of-the-art methods for solving OSR, and we present the baseline KPNE, which is extended from PNE, for solving KOSR.

1) PNE: The progressive neighbor exploration (PNE) algorithm \cite{30} is able to find the optimal sequenced route in general graphs. Algorithm 1 shows the sketch of PNE. During the processing, a priority queue is maintained for partially explored routes (witnesses). At each iteration, the route \((v_0, v_1, \ldots, v_{q-1}, v_q)\) with minimal cost in the priority queue is chosen to be examined, where \( v_i \in V_{C_i} \) for each \( 1 \leq i \leq q \). To extend from the route, we need to consider vertices in the next category \( C_{q+1} \). Instead of extending the route via all its neighbors in category \( C_{q+1} \), only the nearest neighbor \( v_{q+1} \) of \( v_q \), such that \( \text{dis}(v_q, v_{q+1}) = \arg \min_{v \in C_{q+1}} \text{dis}(v_q, v) \), is considered. Moreover, to guarantee the correctness, another candidate route derived from \((v_0, v_1, \ldots, v_{q-1}, v_q)\) is incrementally generated by extending \((v_0, v_1, \ldots, v_{q-1}, v_q)\) via \( v_{q-1} \)'s next nearest neighbor \( v'_{q-1} \) in \( C_{q} \), such that \( v'_q \neq v_q \) and \( \text{dis}(v_{q-1}, v'_q) \geq \text{dis}(v_{q-1}, v_q) \). The algorithm returns the optimal route as it passes through all categories in order and reaches the destination. Since a vertex’s neighbors in the next category \( C_i (1 \leq i \leq j) \) can be as many as \( |C_i| \), it is impractical to compute the least costs from the vertex to all its neighbors. By progressively extending route via its nearest neighbors and generating candidate route derived from it, PNE carefully examines all the possible partially explored candidate routes on demand to find the optimal sequenced route. It is possible to extend PNE to solve KOSR problem, we only need
As a result, the total number of routes to be examined by KPNE can be \( O \left( \prod_{i=1}^{k-1} C_i \right) \) times of forward search (Dijkstra’s algorithm based search by contraction hierarchy) and backward search (DFS based) in the worst case, the number of examined partially explored candidate routes in Section IV-A which reduces the searching space. Moreover, we demonstrate the extensibility of the proposed method by incorporating an optimization technique that is able to find the \( i \)-th nearest neighbor in a category for a given vertex efficiently. Subsequently, we further reduce the searching space by integrating a heuristic estimation in an An* manner in Section IV-B.

A. Dominance Based Algorithm

We first illustrate the intuition of the route dominance relationship. Consider a KOSR query \((s, t, \langle MA, RE, CT \rangle, 2)\) in Figure 1. In order to find the first optimal sequenced route \((s, a, b, d, t)\) with the cost of 20 (shorten as \((s, a, b, d, t)\) and \((s, c, b)\)), KPNE will attempt to examine and extend \((s, a, b, d, t)\) and \((s, c, b)\), because both \((s, a, b)\) and \((s, c, b)\) have a smaller cost than \((s, a, b, d, t)\). However, there is no need to extend \((s, c, b)\) to find \((s, a, b, d, t)\), because the cost of the optimal feasible route extended from \((s, c, b)\) won’t be smaller than that of \((s, a, b)\) i.e., \((s, a, b, d, t)\). Hence, \((s, c, b)\) can be excluded to be extended until the optimal sequenced route \((s, a, b, d, t)\) is found. In this case, we say \((s, c, b)\) is dominated by \((s, a, b)\). Next, we formally define the dominance relationship.

Definition 6 (Dominance): Consider a given category sequence \(C = \langle C_1, \ldots, C_i \rangle\) and two partially explored candidate routes (witnesses) \(P_1 = \langle s, v_1^{\top}, \ldots, v_1^l \rangle\) and \(P_2 = \langle s, v_2^{\top}, \ldots, v_2^l \rangle\) \((1 \leq q \leq j)\). If \(v_1^q \neq v_2^q\) and \(w(P_1) \leq w(P_2)\) holds, \(P_1\) dominates \(P_2\) w.r.t. \(C\), denoted as \(P_1 \prec_C P_2\).

Lemma 1: Given a KOSR query \((s, t, C = \langle C_1, \ldots, C_i \rangle, k)\) and two partially explored routes \(P_1\) and \(P_2\), if \(P_1 \prec_C P_2\), then \(w(P_1) \leq w(P_2)\), where \(P_1^*\) and \(P_2^*\) are the optimal feasible routes that are extended from \(P_1\) and \(P_2\), respectively.

Proof: Suppose \(P_1 = \langle s, v_1^{\top}, \ldots, v_1^j \rangle\), \(P_2 = \langle s, v_2^{\top}, \ldots, v_2^j \rangle\) \((1 \leq q \leq j)\) and \(P_1^* = \langle s, v_1^{\top}, \ldots, v_1^j, v_1^j, \ldots, v_j, t \rangle\), since \(P_1^*\) is the optimal feasible route extended from \(P_1\), \(P^* = \langle v_1^q, v_1^q, \ldots, v_1^q, v_1^q, v_1^q, v_1^q, \ldots, v_j, t \rangle\) must be the optimal sequenced route for category sub-sequence \(\langle C_q^\top, \ldots, C_i \rangle\) from \(v_1^q\) to \(t\). Because \(P_1 \prec_C P_2\), we have \(v_1^q = v_2^q\) and \(w(P_1) \leq w(P_2)\). Thus, \(P_2^*\) can be represented by \(\langle s, v_1^q, \ldots, v_1^q, v_1^q, v_1^q, v_1^q, \ldots, v_j, t \rangle\), then \(w(P_1^*) = w(P_1) + w(P)\) and \(w(P_2^*) = w(P_2) + w(P)\), since \(w(P_1) \leq w(P_2)\), we have \(w(P_1^*) \leq w(P_2^*)\).

According to Lemma 1, there is no need to extend the dominated partially explored routes until the optimal feasible route extended from their dominating route become one of the top-\(k\) optimal sequenced routes. This is because the partially explored candidate routes that are dominated by other partially explored candidate routes with smaller costs can never be extended to be the next optimal sequenced routes before their dominating routes. On the other hand, after an optimal sequenced route is found, we need to reconsider its corresponding dominated routes, so that they can be extended to be the next optimal sequenced routes. Based on the dominance relationship, we propose PruningKOSR method (Algorithm 2).

To check the dominance relationship and maintain the dominated routes, for each vertex \(v\), we introduce two hash
Algorithm 2: PruningKOSR(G, s, t, C, k)

Input: Graph G(V, E), source-destination pair s, t ∈ V, category sequence C = {C₀, ..., Cᵣ}, and k.
Output: The top-k optimal sequenced routes.
1 ∀v ∈ V, initialize v.HTᵣ<C and v.HTᵣ>C;
2 \( \Psi \leftarrow \emptyset \);
3 Priority queue \( Q \) ← \{((s, 1))\}; // (route, x) /* route, x */
4 WHILE \( Q \) is not empty and \( |\Psi| < k \) do
5 \( p = ((v₀, v₁, \ldots, vᵣ−1, vᵣ), x) \) ← \( Q \).extractMin();
6 IF \( q = |C| + 1 \) THEN
7 \( \Psi \leftarrow \Psi \cup \{p\}; \\
8 // reconsider dominated routes
9 FOR each \( i = 0 \) to \( q - 1 \) DO
10 \( p' = ((v₀, v₁, \ldots, vᵣ, x) \) ← vᵣ.HTᵣ>C.getValue(i + 1);
11 \( Q \).insert(p', vᵣ.HTᵣ>C.getValue(i + 1).extractMin());
12 \( vᵣ.HTᵣ>C \).remove(i + 1);
13 ELSE
14 // pruning dominated routes
15 IF \( |p| \notin vᵣ.HTᵣ<C.KeyThen \)
16 \( vᵣ.HTᵣ>C \).add(p, ((v₀, v₁, \ldots, vᵣ), x));
17 \( \psi ← \text{FindNN}(vᵣ, Cᵣ+1, 1); \\
18 \( Q \).insert(((v₀, v₁, \ldots, vᵣ−1, vᵣ+1), 1));
20 ELSE
21 \( vᵣ.HTᵣ>C \).getValue(p).insert(p);
22 IF \( q > 0 \) THEN
23 \( v_q' ← \text{FindNN}(v_q−1, C_q, x + 1); \\
24 \( Q \).insert(((v₀, v₁, \ldots, vᵣ−1, vᵣ', x + 1));
25 return \( \Psi \);

Tables in the form of \( \{\text{key}, \text{value}\} \) pairs. One is \( \text{HT}_ᵣ<C \) for dominating routes, where \( \text{key} \) is the size of the partially explored dominating route that has been extended at \( v \), and the \( \text{value} \) is the route itself. Another one is \( \text{HT}_ᵣ>C \) for dominated routes, where \( \text{key} \) represents the size of dominated route, and \( \text{value} \) is a priority queue for the routes with the size of \( \text{key} \) that have reached \( v \) and been dominated, the dominated routes are ordered according to their costs in an ascending order. We also maintain a result set \( \Psi \) for the top-k optimal sequenced routes and a global priority queue \( Q \) for partially explored routes (winners) sorted by their costs in an ascending order. Moreover, for each route \( p = (v₀, v₁, \ldots, vᵣ−1, vᵣ) \), we introduce an additional attribute \( x \) to indicate that \( vᵣ \) is the \( x \)-th nearest neighbor of \( vᵣ−1 \) in category \( Cᵣ \) when generating \( p \). Initially, only the source with \( x = 1 \) is added to the queue \( Q \). Then we begin a loop until \( Q \) is empty or the top-k optimal sequenced routes have been found.

**Pruning dominated routes:** At each iteration, the route with the minimum cost is chosen to be examined. If it already reaches the destination, we add it to the result set and reconsider the dominated routes (lines 6–12). Otherwise, we check whether it is dominated. For a route \( p = (v₀, v₁, \ldots, vᵣ−1, vᵣ) \) to be examined, if \( p \) is the first route with size \( |p| \) that reaches vertex \( vᵣ \), we add \( p \) to the \( \text{HT}_<C \) of \( vᵣ \) and extend it via \( vᵣ \)'s nearest neighbor in category \( Cᵣ+1 \) (lines 14–17). Otherwise, if its size \( |p| \) is in the \( \text{HT}_<C \) of \( vᵣ \), it means that another route with size \( |p| \) and smaller cost has been reached and extended at \( vᵣ \), so that \( p \) is dominated. According to Lemma 1 there is no need to extend \( p \) anymore, therefore, we insert it into the \( \text{HT}_>C \) of \( vᵣ \) instead of the priority queue \( Q \) (line 19). Subsequently, we generate a new candidate route derived from \( p \). Since the candidate route via the \( x \)-th nearest neighbor of \( vᵣ−1 \) has been generated in previous iterations, we need to find \( vᵣ−1 \)'s \( (x + 1) \)-th nearest neighbor in category \( Cᵣ \), \( vᵣ' \), by invoking algorithm FindNN, and create candidate route \( (v₀, v₁, \ldots, vᵣ−1, vᵣ') \) with incremental \( x \) and insert it into the priority queue (lines 20–22).

**Reconsider dominated routes:** After an optimal sequenced route \( p \) has been found, we need to reconsider the partially explored routes that are dominated by sub-routes of \( p \), since these routes now can possibly be extended to be the next optimal sequenced route. Therefore, for each vertex \( vᵣ \) in \( p \), if \( (v₀, \ldots, vᵣ) \) dominates the routes with size of \( i + 1 \) in the \( \text{HT}_>C \) of \( vᵣ \), we only reconsider the dominated route \( p' \) with the lowest cost in the \( \text{HT}_>C \) of \( vᵣ \), because other routes in \( \text{HT}_>C \) of \( vᵣ \) are dominated by \( p' \). This also explains why we use a priority queue as the \( \text{value} \) in hash table \( \text{HT}_>C \). Since \( p' \)'s \( x + 1 \) nearest neighbor has been computed after it is dominated, we set its \( x \) to '-' (which means there is no need to generate candidate route that is derived from \( p' \)) and re-add it to the priority queue (lines 10–11). Meanwhile, we remove \( (v₀, \ldots, vᵣ) \) from the \( \text{HT}_<C \) of \( vᵣ \), so that the next candidate route that reaches \( vᵣ \) can be extended (line 12).

**Lemma 2:** Algorithm 2 returns the correct result for a KOSR query.

**Proof:** To find the next optimal sequenced route, all
Lemma 3 shows the time complexity of Algorithm 2.

\[
\text{Algorithm FindNN complexity (to KPNE, the searching space is reduced from exponential to polynomial)}
\]

In category \(RE\), we insert \((s, c, b)\) into the \(HT_{\infty C}\) of \(b\), instead, we insert \((s, c, b)\) into the \(HT_{\infty C}\) of \(b\), and generate candidate route \((s, c, e)\) via \(c\)’s nearest neighbor \(e\) in category \(RE\). At step 9, the first optimal sequenced route \((s, a, b, d, t)\) is found. Since both \((s, c, b)\) and \((s, a, c, e, d)\) in \(HT_{\infty C}\) of \(b\) and \(d\) respectively, we re-add them into the queue with \(x=p^\prime\) and remove the corresponding dominating routes from \(HT_{\infty C}\). Finally, at step 13, the second optimal sequenced route \((s, a, b, d, t)\) is found, and we return \{(s, a, b, d, t), (s, a, c, e, d, t)\} as the result.

By pruning the dominated routes and the candidate routes derived from them, both the capacity of the priority queue and the searching space are reduced, which improves the efficiency. Given a KOSR query \((s, t, C, k)\), to find the first optimal sequenced route, for each vertex \(v\) in \(C_i (0 \leq i \leq |C|)\), at most one route with size \((i+1)\) (plus the source) is extended at line 16 in Algorithm 2, and at most \(|C|\) – 1 candidate routes can be generated via \(v\)’s next nearest neighbors in category \(C_{i+1}\) (line 21).

As a result, in the worst case, the number of routes to be examined by Algorithm 2 for the first optimal sequenced route is \(\sum_{0 \leq i \leq |C|} |C_i| \cdot |C|\), in which \(\sum_{0 \leq j \leq |C|} |C_j|\) routes are extended. Then, for each of the next \(k-1\) optimal sequenced routes, at most \(|C|\) dominated routes are reconsidered once an optimal sequenced route is found, which results in at most \(\sum_{2 \leq i \leq |C|+1} |C_i|\) examined routes, and in which at most \(|C|\) routes are extended at \(|C|\) different categories, respectively. That is, to find the top-k optimal sequenced routes, at most \(\sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1+(k-1)\cdot \sum_{2 \leq i \leq |C|+1} |C_i|\) partially explored routes need to be examined, in which \(\sum_{0 \leq j \leq |C|} |C_j|\) and \((k-1)\cdot |C|\) routes are extended. Compared to KPNE, the searching space is reduced from exponential complexity \(\sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1 \cdot \prod_{1 \leq j \leq |C_j|} |C_j|\) down to polynomial complexity \(\sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1 \cdot (k-1)\cdot \prod_{2 \leq i \leq |C|+1} |C_i|\) which, at most \(|C|\) routes are extended at \(|C|\) different categories, respectively.

Lemma 3: Given a KOSR query \((s, t, C, k)\), let \(M = \sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1+(k-1)\cdot \sum_{2 \leq i \leq |C|+1} |C_i|\) be the complexity of Algorithm 2, \(N = \sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1+(k-1)\cdot |C|\) be the time complexity of Algorithm FindNN.

\[
\text{Lemma 3: Given a KOSR query (s, t, C, k), let M =} \sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1+(k-1)\cdot \sum_{2 \leq i \leq |C|+1} |C_i|, N = \sum_{0 \leq j \leq |C|} |C_j| \cdot |C|+1+(k-1)\cdot |C|, \text{the time complexity of Algorithm FindNN is } O(M\rho + M\log N), \text{ where } \rho \text{ is the time complexity of Algorithm FindNN.}
\]

Proof: Since at most \(M\) partially explored candidate routes are generated during the process of Algorithm 2, which means Algorithm FindNN will be called \(M\) times at most, in which, at most \(N\) routes are extended via the nearest neighbor.

So that the complexity of this part is \(O(M\rho)\). In addition, each time we examine a candidate route from the priority queue, if the route is extended via the nearest neighbor, two candidate routes are generated in total, in this case, the capacity of the priority queue will be increased by 1. Otherwise, if the route is dominated, then it cannot be extended and only one candidate route is generated via the nearest neighbor, and the capacity of the priority queue will not change. Since at most \(N\) candidate routes are extended via their nearest neighbors, the capacity of the priority queue is at most \(N\).

As a result, the complexity of the maintenance of the priority queue is \(O(M\rho + M\log N)\).

Finding the \(x\)-th nearest neighbor. Next, we introduce how to find the \(x\)-th nearest neighbor, the core operation FindNN in PruningKOSR. A straightforward way to find the \(x\)-th nearest neighbor of vertex \(v\) in category \(C_{i+1}\) is that by using Dijkstra’s search. We start from \(v\) and extend vertices via their adjacent vertices until the \(x\)-th vertex in \(V_{C_{i+1}}\) is settled. However, each time we find the \(x\)-th nearest neighbor, Dijkstra’s search actually finds the top-\(x\) nearest neighbors from scratch, which results in duplicate search effort throughout the graph. Moreover, since FindNN is frequently invoked, frequent Dijkstra’s searches on large graphs are practically inefficient. Hence, a more efficient method without duplicate searches is called for. To this end, we propose a method to incorporate the use of 2-hop labeling technique, to find the \(x\)-th nearest neighbor.

Given a directed weighted graph \(G(V, E)\), for each vertex \(v \in V\), 2-hop labeling maintains two labels \(L_{in}(v)\) and \(L_{out}(v)\). In particular, \(L_{in}(v)\) consists of a set of label entries in the form of \((u, d_{u,v})\), where \(u \in V\) is a vertex that is able to reach \(v\), and \(d_{u,v}=\text{dis}(u, v)\). Similarly, \(L_{out}(v)\) consists of a set of label entries in the form of \((u', d_{u',v})\), where \(u' \in V\) is a vertex that can be reached by \(v\), and \(d_{u',v} = \text{dis}(u, v)\). Note that \(L_{in}(v)\)’s entries may only contain a subset of vertices that can reach \(v\); similarly, \(L_{out}(v)\)’s entries may only contain a subset of vertices that can be reached by \(v\). In addition, the labels must satisfy the cover property: for any two vertices \(s\) and \(t\), there exists a vertex \(u\) on the shortest path from \(s\) to \(t\) that belongs to both \(L_{out}(s)\) and \(L_{in}(t)\). Based on which, to answer a least cost query from \(s\) to \(t\), we compute as follows:

\[
dis(s, t) = \min\{d_{s,u}+d_{u,t} | (u, d_{s,u}) \in L_{out}(s), (u, d_{u,t}) \in L_{in}(t)\}
\]

Hence, the least cost from \(s\) to \(t\) can be computed by scanning \(L_{out}(s)\) and \(L_{in}(t)\) to find their matching label entries. If the label entries in each label set are sorted by their vertices, then we can compute \(\text{dis}(s, t)\) in \(O(|L_{out}(s)|+|L_{in}(t)|)\) time using a merge-join like algorithm.

We note that building the 2-hop labeling with the minimal size (where the size of the index is defined as \(\sum_{v \in V} |L_{in}(v)|+|L_{out}(v)|\)) while satisfying the cover property is NP-hard [9]. Thus, existing methods [1], [2], [4], [9] are all heuristic to approximate the minimal 2-hop labeling index. Alternatively, we may use an all-pairs shortest path algorithm
Let’s find the nearest neighbor of \( x \). Since the label entries in the inverted label index are sorted in an ascending order, with the inverted label index \( IL(v) \in IL(C_{i+1}) \);

- **Input:** Vertex \( v_i \), category \( C_{i+1} \), integer \( x \).
- **Output:** The \( x \)-th nearest neighbor of \( v_i \) in \( C_{i+1} \).

| \( v_i \) | \( IL(v) \) |
|---|---|
| 1 | \((a, 0), (s, 8), (t, 10)\) |
| 2 | \((b, 0), (s, 13), (t, 12)\) |
| 3 | \((c, 0), (s, 10), (t, 15)\) |
| 4 | \((d, 3), (d, 0), (e, 3), (s, 13), (t, 13)\) |
| 5 | \((e, 0), (s, 14), (t, 10)\) |
| 6 | \((f, 10), (f, 0), (s, 24), (t, 20)\) |
| 7 | \((s, 0), (t, 25)\) |
| 8 | \((t, 0)\) |

To improve the efficiency of \( FindNN \), we construct an inverted label index for each category, so that we can quickly identify the matching label entries between \( v_i \) and all vertices in \( C_{i+1} \).

The inverted label index for a category \( C_i \), denoted as \( IL(C_i) \), consists of label elements \( IL(u) \), where \( u \in V \) is the vertex in the label entry belongs to \( L_{in}(u) \) for each \( u \in V \). That is, \( IL(v) \) consists of a list of label entries \( (u, d_{u,v}) \) such that \( u \in V \) and \( d_{u,v} \in L_{in}(u) \), and all label entries in \( IL(v) \) are sorted by their costs, i.e., \( d_{u,v} \), in an ascending order. With the inverted label index \( IL(C_i) \), for each label entry \( (u, d_{u,v}) \in L_{out}(v) \), the vertices with matching label entry in \( C_i \) can be found in \( IL(u) \in IL(C_i) \).

Since the label entries in the inverted label index are sorted, to find the \( x \)-th nearest neighbor of \( v \) in category \( C_i \), only one label entry in each \( IL(u) \) needs to be checked.

| Inverted label | IL(s) | Label entries |
|---|---|---|
| \((a, 0)\) | \((a, 0)\) |
| \((c, 0)\) | \((a, 8), (c, 10)\) |
| \((t, 0)\) | \((c, 15), (a, 33)\) |

We maintain an array list \( NL \) for \( v_i \) to keep its nearest neighbors that have been found. Moreover, to avoid searching from scratch every time, we maintain the candidate label entries \( (u', d_{u',v'}) \) in matching inverted label \( IL(u') \) such that \( (u', d_{u',v'}) \in L_{out}(v_i) \) that have been found so far into a priority queue \( NQ \) and all entries are sorted by \( d_{u',v'} \) in an ascending order. In addition, to keep the entry position that we have scanned for each \( IL(v) \), we introduce a hash table structure \( KV \), where \( key \) is vertex \( u' \) and \( value \) is the entry position of \( IL(u') \). \( NL \) and \( NQ \) are all global variables and initialized to be empty. By using above data structures, to find the \( x \)-th nearest neighbor of \( v_i \) in \( C_{i+1} \), we can start from last nearest neighbor searching instead of finding the top-\( x \) nearest neighbors from scratch, so that no overlapping search is needed. Specifically, if the \( x \)-th nearest neighbor is in \( NL \), it can be retrieved and returned (lines 4–5). Otherwise, for the first time to find the 1st nearest neighbor of \( v_i \), we retrieve all the matching inverted labels \( IL(u') \in IL(C_{i+1}) \), then insert the first label entry of each \( IL(u') \) into \( NQ \) and initialize \( KV \) (lines 6–10). Subsequently, we get the minimal label entry \((a, (c, 10))\) in \( NQ \) which is the next nearest neighbor (line 11). In addition, we add the next label in \( IL(u') \) into \( NQ \) and update its entry position for latter nearest neighbor search (lines 12–16). Since nearest neighbors are incrementally needed, the next nearest neighbor will be the \( x \)-th nearest neighbor, so we add it to \( NL \) and return (lines 17–18).

Example 5: Consider the inverted label index in Table \( V \), Table \( V \) and Table \( IV \) we find the 2nd nearest neighbor of \( s \) in category \( MA \). Let’s follow Example \( IV \) since the 1st nearest neighbor of \( s \) is \( a \) from \( IL(s) \), after finding \( a \), \( NL = \{(a, 8), (c, 10), (c, 15)\} \), and \( KV = \{(s, 2), (t, 1)\} \). Hence, we get the minimal label \((c, 10)\) in \( NQ \). Because all the labels in \( IL(s) \) are scanned, we set the entry position of \( IL(s) \) to
optimal feasible route extended from p instead of their real costs in an A∗ order. By adding a parent vertex to each partially explored candidate route, the number of candidate routes can be reduced. To this end, for each partially explored candidate route p, we heuristically estimate the cost of the optimal feasible route extended from p, so that we can examine routes according to their estimated costs instead of their real costs in an A∗ manner.

Given a KOSR query ⟨s, t, C, k⟩, for a partially explored candidate route (witness) p = ⟨v0 = s, v1, · · · , vi⟩, the optimal feasible route extended from p can be represented as p′ = ⟨s, v1, · · · , vi, vj+1, · · · , v|C|, t⟩, so that w(p′) = w(p) + w(v1, vj+1, · · · , v|C|). That is, we need to estimate the cost of {v1, vj+1, · · · , v|C|} which is the optimal sequenced route starts from vi and passes through p’s remaining categories and reaches the destination t. We say that a heuristic estimation h for a route P is admissible if h(P) ≤ w(P). Recall that dis(u, v) returns the least cost from vertex u to vertex v along all possible routes from u to v, and it can be easily computed by 2-hop labeling. Thus, we have dis(vi, t) ≤ w(vi, vj+1, · · · , v|C|), which means dis(vi, t) is an admissible estimation of the cost of route {v1, vj+1, · · · , v|C|}. Therefore, the estimated cost of p′ is w(p) + dis(vi, t). By applying this target-directed estimation, we propose another improved method StarKOSR. Instead of ordering the routes, i.e., p = ⟨v0, v1, · · · , vi⟩, in the priority queues (Q and priority queues in HT×C) by their real costs, i.e., w(p), in StarKOSR, we order routes by their estimated costs, i.e., w(p) + dis(vi, t), so that the optimal feasible routes can be progressively found.

The detail process of StarKOSR is almost the same as Algorithm 2 except for FindNN. Since we examine routes by their estimated costs, instead of finding the x-th nearest neighbor of vertex vi, we find vi’s neighbor vj+l in category Cj+l such that dis(vi, vj+l) + dis(vj+l, t) is the x-th least among all vertices in Cj+l, we call vj+l the x-th nearest estimated neighbor of vi. To this end, we devise the algorithm FindNN (Algorithm 4).

```
Algorithm 4: FindNN(vi, Cj+l, x)

Input: Vertex vi, category Cj+l, integer x.
Output: Vertex vj+l in Cj+l such that dis(vi, vj+l) + dis(vj+l, t) is the x-th least.
1 ENL ← list of vi’s estimated neighbors in Cj+l that have been found;
2 ENQ ← priority queue of vi for the candidate neighbors in Cj+l;
3 ln ← vi’s last nearest neighbors that have been computed in Cj+l;
4 if |ENL| ≥ x then
5 return ENL[x];
6 while |ENL| = 0 ∧ |ENQ| = 0 ∨ (ln ≠ NULL ∧ dis(vi, ln) < dis(vi, v) + dis(v, t)), where v is the vertex in ENQ with minimal dis(vi, v) + dis(v, t) do
7 if ln ≠ NULL then
8 ENQ.insert(ln);
9 ln ← FindNN(vi, Cj+l, |ENL| + |ENQ| + 1);
10 vj+l ← ENQ.extractMin();
11 ENL.add(vj+l);
12 return vj+l;
```

Given vertex vi, category Cj+l and integer x, Algorithm 4 finds vi’s x-th nearest estimated neighbor, vj+l, in Cj+l. To avoid computing vi’s x-th nearest estimated neighbor multiple times, we maintain an array list ENL of vi to keep the nearest estimated neighbors that have been computed. Moreover, to continuously compute the next nearest estimated neighbors, we maintain a priority queue ENQ of vi for candidate neighbors that have been considered so far and sort the neighbors (v) by their estimated costs (dis(vi, v) + dis(v, t)) in an ascending order. Meanwhile, we store the last nearest neighbor of vi that have been computed into variable ln, so that we can start from last nearest estimated neighbor searching instead of computing from scratch every time. We note that ENL, ENQ and ln are global variables and initialized to be empty or NULL. Then if the x-th nearest estimated neighbor of vi has been computed, we can retrieve it from ENL instead of recomputing it (lines 4–5). Otherwise, we find the x-th nearest estimated neighbor for the first time. Instead of checking all vertices in Cj+l to find the x-th nearest estimated neighbor, we incrementally find the next nearest neighbor ln of vi in Cj+l, by calling FindNN (line 9), if dis(vi, ln) is greater than the minimal cost dis(vi, v) + dis(v, t) in ENQ, then v has the minimal estimated cost among all remaining vertices in Cj+l, because other vertices, say v′, have not been checked hold dis(vi, v′) ≥ dis(vi, ln) ≥ dis(vi, v) + dis(v, t), which means that their estimated cost cannot be less than that of v. Since FindNN is incrementally called, the next nearest estimated neighbor is the x-th nearest estimated neighbor. Finally, we add v to ENL and return it as the result (lines 10–12).

Lemma 4: Algorithm StarKOSR returns the correct result for a KOSR query ⟨s, t, C = ⟨C1, · · · , Cj⟩, k⟩.

Proof: Suppose the examined route from the priority
FindNN the searching space and improves the efficiency of KOSR. In
comparing to Example 2. That is, StarKOSR further reduces and no dominated routes exist. Finally, the second optimal nearest estimated neighbor of step 6, the first optimal sequenced route $b$ $\langle s, a, b, d, t \rangle (20), 1, (s, a, b, f) (22), 1)$, $\langle s, a, b, d, t \rangle (20), 1, (s, a, e) (21), 2, (s, c, b) (22), 1)$, $\langle s, a, b, f, j \rangle (43), 2)$, $\langle s, a, c, d, f \rangle (21), 2, (s, c, b) (22), 1, (s, a, c, f) (27), 2$, $\langle s, a, c, f, j \rangle (43), 2)$.

queue at each iteration is $P = \langle s, v_1, \ldots, v_i \rangle \ (1 \leq i \leq |C|)$ and $v_j \in V_{C_i}$ for $1 \leq j \leq i$, we first prove the total estimated cost of $P$, i.e., $w(P) + \text{dis}(v_i, t)$, is minimal in all possible partially explored routes. For all partially explored routes in the priority queue, it is trivial that they have greater total estimated costs than $P$. For the possible partially explored routes that have not been generated, they must have greater total estimated costs than the routes in the priority queue, since we always generate nearest estimated neighbor for each examined routes. Next, we prove that $P$ is the optimal route from $s$ to $v_i$ and passes through category sequence $\langle C_1, \ldots, C_i \rangle$. Suppose there is a route $P' = \langle s, v'_1, \ldots, v'_q, \ldots, v'_j \rangle (q < i)$ such that $w(P') + \text{dis}(v'_q, \ldots, v'_j) < w(P)$, then we have $w(P') + \text{dis}(v'_q, \ldots, v'_j) < w(P') + \text{dis}(v'_q, \ldots, v'_j) + \text{dis}(v_i, t) < w(P) + \text{dis}(v_i, t)$, which means that the total estimated cost of $P'$ is smaller than $P$, which is a contradiction. Thus, $P$ is the optimal route from $s$ to $v_i$ and passes through category sequence $\langle C_1, \ldots, C_i \rangle$. Hence, when $v_i$ equals to destination $t$, we have found an optimal route from $s$ to $t$ and passes through category sequence $C$. In summary, StarKOSR returns the correct result for a KQSR query.

Example 6: Reconsider Figure [1] and suppose the KOSR query is $\langle s, t, (MA, RE, CI) \rangle$. Table [X] lists the routes in the priority queue $Q$ at each step by applying StarKOSR algorithm. The first route is $\langle s \rangle$, and we extend $\langle s \rangle$ by finding the 1st nearest estimated neighbor of $s$ in category $MA$. Initially, the $h$ of $s$ is NULL, ENQ and ENL are empty, we find the 1st nearest neighbor of $s$ which is $a$ with $\text{dis}(s, a) + \text{dis}(a, t) = 8 + 12 = 20$, then we add $a$ to ENQ and continue to find the 2nd nearest neighbor $c$ with $\text{dis}(s, c) + \text{dis}(c, t) = 10 + 7 = 17$. Because $\text{dis}(s, c) = 10 < \text{dis}(s, a) + \text{dis}(a, t) = 8 + 12 = 20$, $c$ is added to ENQ. Since $s$ has no nearest neighbor in $MA$, $h = NULL$ and $c$ is the 1st nearest estimated neighbor of $a$ (step 2), thus, we have $h = NULL$, $\text{ENQ} = \{a\}$ and $\text{ENL} = \{c\}$ after $c$ is returned. Subsequently, we extend $\langle s, c \rangle$ via $b$ and generate candidate route $\langle s, a \rangle$ by finding the 2nd nearest estimated neighbor of $s$ in category $MA$ (step 3). At step 6, the first optimal sequenced route $\langle s, a, b, d, t \rangle$ is found and no dominated routes exist. Finally, the second optimal feasible route $\langle s, a, c, d, t \rangle$ is found at step 9.

As we can see from Example 6, 4 steps are reduced compared to Example [2]. That is, StarKOSR further reduces the searching space and improves the efficiency of KOSR. In StarKOSR, though FindNN may be called multiple times as we attempt to find the $x$-th nearest estimated neighbor by applying FindNEN, however, to find the next optimal feasible route, the total times of calling FindNN by StarKOSR is significantly less than that by PruningKOSR. We address this as follows:

Table VI: Running Example of StarKOSR for Fig.[X]

| Step | Routes (route(estimated cost), x) in priority queue Q |
|------|------------------------------------------------------|
| 1    | $\langle (s, v_1, \ldots, v_i) \rangle \ (1 \leq i \leq |C|)$ |
| 2    | $\langle s, c \rangle (17), 1$ |
| 3    | $\langle s, a \rangle (20), 1, \langle s, c, b \rangle (22), 1$ |
| 4    | $\langle s, a, b \rangle (20), 1, \langle s, c, b \rangle (22), 1$ |
| 5    | $\langle s, a, b, d \rangle (20), 1, \langle s, a, c \rangle (21), 2, \langle s, c, b \rangle (22), 1$ |
| 6    | $\langle s, a, b, d, t \rangle (20), 1, \langle s, a, e \rangle (21), 2, \langle s, c, b \rangle (22), 1, \langle s, a, b, f \rangle (43), 2$ |
| 7    | $\langle s, a, c \rangle (21), 2, \langle s, c, b \rangle (22), 1, \langle s, a, b, f \rangle (43), 2$ |
| 8    | $\langle s, a, c, d \rangle (21), 2, \langle s, c, b \rangle (22), 1, \langle s, a, b, f \rangle (43), 2$ |
| 9    | $\langle s, a, b, d, t \rangle (20), 1, \langle s, a, c \rangle (21), 2, \langle s, c, b \rangle (22), 1, \langle s, a, b, f \rangle (43), 2$ |

Figure 2. The searching space of different methods

suppose we examine route $P = \langle v_0, \ldots, v_i \rangle$, and find the $x$-th nearest estimated neighbor $v_{i+1}$ of $v_i$ by calling FindNN $j$ times, that is $j$ nearest neighbors of $v_i$ have been found and $\text{dis}(v_i, NN_j) < \text{dis}(v_i, v_{i+1}) + \text{dis}(v_{i+1}, t)$ for each nearest neighbor $NN_j$, $1 \leq j < i$. If $w(p) + \text{dis}(v_i, v_{i+1}) + \text{dis}(v_{i+1}, t) < w(P)$, where $w(P)$ is the cost of the next optimal feasible route $P$, then $w(p) + \text{dis}(v_i, NN_j) < w(P)$, that is, to find $P$, $j - 1$ candidate routes $\langle v_0, \ldots, v_i, NN_j \rangle$ should be examined in PruningKOSR by calling FindNN $j$ times. In this case, both methods call FindNN the same times. On the other hand, if $w(p) + \text{dis}(v_i, v_{i+1}) + \text{dis}(v_{i+1}, t) \geq w(P)$, then $\langle v_0, \ldots, v_i, v_{i+1} \rangle$ won’t be examined and subsequently, all possible candidate routes derived from $\langle v_0, \ldots, v_i, v_{i+1} \rangle$ can never be considered before $P$ is found, which in turn reduces the searching space of StarKOSR. Thus, in summary, the times of calling FindNN by StarKOSR is significantly less than that by PruningKOSR.

Remarks. Figure 2 illustrates the searching space of different methods for the optimal sequential route. Since KPNE examines all possible candidate routes that with smaller costs than the optimal sequential route, its searching space (Figure 2[a]) is a whole circle whose radius is the cost of the optimal sequenced route from source $s$ to destination $t$, and each route in the circle will be examined.

In PruningKOSR, for each category $C_i$, at most $|C_i|$ routes are extended due to dominance relations, which results in at most $|C_i||C_{i+1}|$ candidate routes can be examined at category $C_{i+1}$, which is the area of each dark ring in Figure 2(b). As a result, the searching space (area) of PruningKOSR is reduced compared to KPNE, and the pruned space consists of the routes that are dominated and the candidate routes derived from them.

For StarKOSR, since we consider the whole cost of the route from source to destination by using target-directed strategy, the partially explored candidate routes that are far away from the destination are further pruned, as a result, the area of each ring in Figure 2(c) gets smaller compared to Figure 2(b). Since the estimated whole costs of the partially explored routes are not greater than the real costs of their corresponding optimal sequenced routes, and as we extend routes along the category sequence, the estimated whole costs become larger and closer to the real optimal cost. As a result, at the beginning, loose estimated cost (may not contain the required categories) enables more candidate routes to be examined and the searching space (area) increases. Subsequently, when the estimated costs get tighter and are closer and closer to the real optimal cost and finally equal to the real optimal cost, more and more routes whose estimated costs are greater than the optimal cost are filtered and the searching space (area) shrinks until the optimal sequenced route is found.
C. Extensions

In this section, we extend our work in several aspects.

**Variants of KOSR:** The KOSR query can have different variants, which can be solved by extending our proposal. For KOSR on *unweighted graphs*, we simply set the weights of all edges to 1. For *undirected graphs*, for each vertex \( v \), \( L_{in}(v) \) and \( L_{out}(v) \) are identical in the 2-hop labeling, and thus only one label is needed. In the case that source is not required, we can additionally add all vertices in the first category instead of the source to the priority queue (line 3 in Algorithm 2). In the case that destination is not required, since the estimated cost to the destination cannot be applied, the StarKOSR method will not work, but PruningKOSR still works. In the case that people want to find the KOSR according to their personal preferences, for example, you may want the restaurant (RE) in your sequenced route to be an Italian restaurant, such constraint can be easily met by finding the \( x \)-th nearest Italian restaurant in category RE. Specifically, these constraints can be added to line 15 of Algorithm 3.

**Handling dynamic updates:** We distinguish two different kinds of updates: the graph structure updates and the category updates.

For the graph structure updates, we adopt existing methods \cite{3, 6, 38} to update label index. After label entries are inserted into or removed from the label index (update the cost of existing label entries can be regarded as a remove-insert operation), the corresponding inverted label indexes should also be updated accordingly.

For the category updates, which means the category set \( F(v) \) of vertex \( v \) has changed. If a new category \( C_i \) is inserted into \( F(v) \), add \( v \) into \( V_{C_i} \), and for each entry \((u, d_{a,u}, v)\) in \( L_{in}(v) \), we find the inverted label \( IL(u) \in IL(C_i) \), and insert \((v, d_{u,v})\) into \( IL(u) \) by using a binary search for \( d_{u,v} \). On the other hand, if a category \( C_i \) is removed from \( F(v) \), we delete \( v \) from \( V_{C_i} \), and for each entry \((u, d_{u,v})\) in \( L_{in}(v) \), find inverted label \( IL(U) \in IL(C_i) \), then retrieve and remove \((v, d_{u,v})\) from \( IL(U) \). Since the inverted label index is in order, above operations take \( O(|L_{in}(v)| \log |C_i|) \) time, which is efficient.

**Disk-based query answering:** In the case that the label index cannot fit into memory, we store the indexes (including label indexes and inverted label indexes) into disk according to categories. Specifically, indexes in each category \( C_i \) are stored as follows: \( IL(C_i) \), \( L_{out}(v) \), and \( L_{in}(v) \) for each \( v \in V_{C_i} \). For each KOSR query \((s, t, C, k)\), we locate the beginning of the index of each category \( C_i \) by a disk-based \( B^+ \)-tree, and load the inverted label index \( IL(C_i) \) as well as \( L_{out}(v) \) for each \( v \in V_{C_i} \). For source \( s \) (or destination \( t \), we first locate its category that it belongs to, then locate and load its label index \( L_{out}(s) \) (or \( L_{in}(t) \)). Overall, \(|C| + 4 \) disk seek operations are needed.

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

**Datasets:** We use five real-world graphs with varying sizes. In particular, CAL, NYC, COL, and FLA are graphs representing the road networks of California, New York City, Colorado, and Florida, respectively. \( G^+ \) is the social network from Google+. Table VII gives the sizes in terms of the cardinalities of both vertex and edge sets.

\[
\begin{array}{|c|c|c|}
\hline
\text{Dataset} & |V| & |E| \\
\hline
\text{CAL} & 68,345 & 68,990 \\
\text{NYC} & 980,632 & 1,280,981 \\
\text{COL} & 435,666 & 1,057,066 \\
\text{FLA} & 1,070,376 & 2,687,902 \\
\text{G}^+ & 107,614 & 13,673,453 \\
\hline
\end{array}
\]

In particular, CAL is a weighted, undirected graph where edge weights represent the distances of the corresponding roads. In addition, 47,298 vertices in \( G \) are associated with 63 different categories. NYC is a weighted, undirected road network downloaded from OpenStreetMap\(^2\). In addition, we also get the POI dataset of New York from OpenStreetMap\(^2\). Specifically, the POI dataset contains 30,382 points of interest in New York that belong to 135 different categories. For each POI, we find its nearest vertex in the road network and regard the category of the vertex as the category of the POI.

Graphs COL and FLA are weighted directed graphs, where edge weights represent the travel time of roads. Graph \( G^+ \) is an unweighted, directed graph where all edge weights are set to 1. Since no categorical information is associated with the vertices in these graphs, we generate categories for the vertices using both uniform and zipfian distributions. In particular, we follow \cite{29} to generate uniform distributions. We fix the number of vertices in each category with parameter \(|C_i|\), and then uniformly assign a category to vertices. We generate uniform categories for COL, FLA, and \( G^+ \), which is used as the default setting in the following experiments. Next, following \cite{32}, we generate 100 categories for FLA with Zipfian distribution, and we use a parameter factor \( f \geq 1 \) to control the skewness of the distributions, the greater the \( f \) is. The less skew the distributions are. For example, when \( f = 1.2 \), the smallest category size is 23, and the largest category size is 139,717.

**Queries:** For each KOSR query \((s, t, C, k)\), we randomly select a source-destination pair, a category sequence with size \(|C|\), and an integer \( k \). Then, we issue the query on all graphs. In each experiment, 50 random query instances are constructed and the average query time is reported. If a query cannot stop within 3,600 seconds, or fails due to out of memory exception, we denote its corresponding query time as INF. We vary important parameters according to Table VII, where default parameter settings are shown in bold.

| Parameter | Values |
|-----------|--------|
| \(|C_i|\) | 5,000, 10,000, 15,000, 20,000 |
| \(|C|\) | 2, 4, 6, 8, 10 |
| \(k\) | 10, 20, 30, 40, 50 |

**Label index:** We adopt the pruned landmark labeling method \cite{2}, which achieves good performance and is easy to implement, to precompute the label index for each graph in Table VII. Based on the label index, we then construct the inverted

\[\text{http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm}\]
\[\text{http://openstreetmap.org}\]
\[\text{http://www.dis.uniroma1.it/challenge9/download.shtml}\]
\[\text{http://snap.stanford.edu/data/index.html}\]
label index for each category in the graph. Table IX shows the preprocessing results on different graphs under default parameter settings. For large graphs, e.g., FLA, the index sizes may be too large to fit into main memory. To contend with this, we store the indexes on disks. Alternatively, labeling compression method [12] can be applied to further reduce the index sizes.

| Graph Time [H:M] | Avg. | Avg. | Index Size |
|------------------|------|------|------------|
| CAL              | 0:1  | 122.90 | 95.24MB |
| NYC              | 0:52 | 122.90 | 14.11GB |
| COL              | 0:9  | 1,101.06 | 5.05GB |
| FLA              | 2:42 | 1,495.84 | 18.25GB |
| G+               | 0:2  | 335.53  | 230.47MB |

| Graph Time [H:M] | Avg. | Avg. | Index Size |
|------------------|------|------|------------|
| CAL              | 0:1  | 9543.48 | 49.05MB |
| NYC              | 0:1  | 10863.24 | 12.61GB |
| COL              | 0:3  | 98,345.80 | 2.53GB |
| FLA              | 0:14 | 186,763.43 | 9.18GB |
| G+               | 0:1  | 39,621.12 | 113.32MB |

**Methods:** We consider the following methods for answering KOSR queries: (1) GSP: the state-of-the-art algorithm to find the optimal sequenced route (k = 1). (2) KPNE: the KPNE algorithm (Section II-B) by using Algorithm FindNN to find the nearest neighbors. (3) PK: our algorithm PruningKOSR by using dominance relationship to filter temporarily unnecessary routes (Section IV-A). (4) SK: our algorithm StarKOSR by using the target-directed strategy to find the optimal feasible routes (Section IV-B). (5) SK-DB: StarKOSR with label indexes resident on disks. (6) KPNE-Dij, PK-Dij, SK-Dij: the KPNE, PruningKOSR, and StarKOSR algorithms by using Dijkstra’s search to find the nearest neighbors rather than using Algorithm FindNN.

**Evaluation Criteria:** We evaluate the performance of different methods in three different aspects: the query run-time, the number of examined routes (witnesses), and the number of (next) nearest neighbor (shorten as NN) queries executed by calling Algorithm FindNN, where the number of hits in the NL list (line 5 at Algorithm 3) is not included.

**Implementation details:** All algorithms are implemented in Java 1.6 and run on a Windows 10 machine with 3.2GHz CPU, and 32 GB memory.

**B. Experimental Results**

We first evaluate the efficiency of different methods (except for GSP) on different graphs for answering KOSR queries under the default parameter settings, and then evaluate the effects of parameters by varying their values. Finally, when k = 1, we test the performance of our methods against the state-of-the-art method GSP for answering OSR queries.

**Overall performance under default parameter settings.** Figures 3(a) and 3(c) show the performance of different methods on different graphs. The run-times of the methods on different graphs are illustrated in Figure 3(a). Since KPNE examines all possible candidate routes in the searching space, both KPNE and KPNE-Dij are not well performed and they cannot return the results on larger graphs with large category size, i.e., COL, FLA, and G+ within 3,600 seconds. Compared to KPNE, by reducing the searching space, both PK and SK are able to return the results on all graphs. Since SK further filters partially explored routes that are far away from the destination by using a target-directed cost estimation strategy, it performs nearly two orders of magnitude faster than PK on COL and FLA, 4 (or 3) times faster than PK on CAL (or NYC), and 7 times faster than PK on G+. In addition, by comparing the costs of the routes in the result set of different methods on CAL and CAL, our methods, i.e., PK and SK, have the same results as KPNE, which also verifies the correctness of our methods. On the other hand, since the time complexities of our methods are independent of the graph size, but are dependent on the size of category sequence C and the category size |C|, both PK and SK have steady query run-times on larger graphs such as COL and FLA. Moreover, PK and SK perform orders of magnitude faster than PK-Dij and SK-Dij, respectively, because Algorithm FindNN performs efficient NN queries by using inverted label indexes. Since SK-DB needs additional time to load label indexes into memory and initialize them for each query, it takes more time than SK. However, it still outperforms PK where all indexes are always resident in main memory.

For G+, since its edge weights are all 1 and its diameter is only 6, the partially explored routes and nearest neighbors tend to have similar costs, which leads to a larger searching space for both PK and SK, as a result, both PK and SK take much more time to find the top-k optimal sequenced routes. Moreover, since Dijkstra’s search on unweighted graph explores much more vertices and edges, all KPNEDij, PK-Dij and SK-Dij cannot return the results on G+.

Table X shows the distributions of the run-times of our methods on graph FLA. Clearly, the NN queries dominate the query run-time of both methods. Since lots of candidate routes are examined in PK, the maintenance of the priority queue in PK costs more time than does SK. On the other hand, SK
needs to compute the least cost to the destination to estimate the total cost for a partially explored route, which takes some time. While PK does not spend any time since it does not estimate the total cost. However, the time on cost estimation is only a small portion of the overall query time.

Figure 3 shows the searching space of SK at different categories on different graphs. Initially, only one route (source $s$) is examined at category 0. Then the number of examined routes increases along the category sequence, because the estimated costs are loose and more candidate routes are enabled to be examined. As the estimated costs are closer and closer to the real least costs, the number of examined routes quickly decreases and the searching space shrinks. Finally, only 30 routes are examined at the last category (i.e., for destination $t$). The searching space begins to decrease at the 3rd or 4th category or even earlier, which is very efficient. Figure 3 is also consistent with the intuition shown in Figure 2(c).

Next, we show performance while varying important parameters. Due to the space limitation, we only report experimental results on small graph CAL with real categories and large graph FLA with synthetic categories.

**Effect of $k$.** Figures 3(d) and 3(e) show the effect of parameter $k$. On large graph FLA, KPNE, KPNE-Dij, and PK-Dij cannot return the results within 3,600 seconds, even when $k = 10$, due to larger searching space or too many NN queries by Dijkstra’s search. On small graph CAL, all methods are able to compute the results and KPNE, PK, SK are much more efficient than KPNE-Dij, PK-Dij, SK-Dij, respectively. Both Figures 3(d) and 3(e) show that SK and SK-DB greatly outperform other methods in different $k$ due to much fewer examined routes and NN queries. Note that all methods perform steadily with different $k$, meaning that they are scalable w.r.t. $k$ and are able to process KOSR with large $k$s. This is because the top-$k$ optimal sequenced routes tend to have similar costs, and once we find the 1st optimal sequenced route, other optimal sequenced routes are also considerably covered in its searching space. As a result, fewer NN queries are needed to find the other optimal sequenced routes when $k$ rises, therefore, the query time only increases slightly, which is also consistent with the time complexity analysis in Lemma 3. Figure 3 shows the performance of different methods with small $k$ on CAL and FLA. The query time of all methods slightly change as $k$ slowly increases, and our proposed algorithms also outperform existing algorithms.

**Effect of $|C|$.** The performance of different methods on FLA and CAL by varying the size of category sequence $|C|$ is shown in Figures 3(f) and 3(g). When $|C| = 2$, KPNE is able to return the results on FLA as the searching space is small. However, KPNE-Dij and PK-Dij still cannot compute the results on FLA within 3,600 seconds due to too many Dijkstra’s searches on large graphs for KPNE and PK. As $|C|$ increases, the searching space of KPNE increases exponentially, KPNE fails to return the results on FLA when $|C| \geq 4$. On small graph CAL, KPNE-Dij cannot return results when $|C| \geq 8$. Although the searching spaces and run-times of PK and SK (SK-DB) increase as $|C|$ gets larger due to greater $M$ and $N$ in Lemma 3, SK (SK-DB) greatly outperforms PK in all settings. In addition, the run-time of SK (SK-DB) increases more slowly than PK. A larger $|C|$ means more label indexes need to be loaded into memory and initialized by SK-DB. As a result, SK-DB needs more disk accesses and thus a higher overhead compared to SK.

**Effect of $|C_i|$.** Figure 3(h) shows the performance of different methods on FLA by varying the size of vertices in each category, i.e., $|C_i|$. We only report experiments on the largest graph FLA as we do not generate categories for CAL. Due to the huge searching space, KPNE, KPNE-Dij, and PK-Dij cannot return the results even when $|C_i| = 5,000$. Obviously, the performance of both PK and SK deteriorates as $|C_i|$
increases, because the time complexity of the two methods increases as $|C_i|$ increases according to Lemma\ref{lemma:sk}. Intuitively, a larger $|C_i|$ means more vertices in each category and thus more routes to be examined. Clearly, SK is more efficient than PK due to much fewer NN queries. As $|C_i|$ increases, the runtime increasing trend of SK(-DB) is slower than that of PK, which means SK(-DB) is more scalable w.r.t. $|C_i|$.

**Zipfian category distribution.** Figure\ref{fig:zipfian} illustrates the results of different $f$ with $|C| = 6, k = 30$ on FLA. Clearly, our methods greatly outperform baseline KPNE in all settings. It shows that the query time of PK increases as $f$ gets larger, and KPNE cannot return the results when $f \geq 1.4$. This is because a larger $f$ means less skew distribution. As a result, the number of partially explored routes to be examined between consecutive categories, i.e., $|C_i| \cdot |C_{i+1}|$ in the worst case (see Lemma\ref{lemma:sk}), gets larger for less skew distribution since $|C_i|$ and $|C_i| + 1$ tend to be similar in this case. Hence, more time is needed to find the top-$k$ optimal sequenced routes. Moreover, since SK filters much more routes, it greatly outperforms PK.

**Performance for the OSR queries.** By setting $k = 1$, the KOSR problem becomes the OSR problem. We evaluate the performance of the state-of-the-art OSR method, GSP, and our proposed methods. Figure\ref{fig:osr} shows the run-time of different methods on different graphs. Clearly, the state-of-the-art method GSP outperforms KPNE(Dij), PK-Dij and SK-Dij on all graphs, and GSP also outperforms PK on graphs with large category size, i.e., COL, and FLA, because GSP only requires $O(|C|)$ graph searches to find the optimal sequenced route, while PK needs much more examined routes and NN queries on these graphs. However, on graph with small category size, i.e., CAL and NYC, PK is more efficient than GSP due to fewer examined routes and NN queries. In all settings, SK and SK-DB are more efficient than GSP; since SK and SK-DB have much smaller searching space by using the target-directed cost estimation strategy, and achieve very efficient NN query by using inverted label indexes. For $G^+$, we cannot build the contraction hierarchy structure for GSP on $G^+$ in 3 days, thus GSP cannot return the results on $G^+$. In addition, the run-time of GSP is dependent on the graph sizes. As the graph size increases, GSP takes longer time. In contrast, the runtime of SK(-DB) is independent of the graph sizes, meaning that it has better scalability w.r.t. the graph sizes.

VI. CONCLUSION AND OUTLOOK

In this paper, we study the top-$k$ optimal sequenced routes problem. We propose efficient algorithms based on a novel route dominance relationship and a target-directed cost estimation strategy using hop labeling techniques. Extensive experiments on real world graphs demonstrate that the proposed algorithms are efficient.

As a future work, we plan to fill the gaps as shown in Table\ref{table:comparison} to solve the KOSR querying when partial or arbitrary category orders or personal preferences for categories are allowed on both Euclidean and general graphs.

**ACKNOWLEDGEMENT**

Our research is supported by the National Key Research and Development Program of China (2016YFB1000905), NSFC (61532021, U1501252, 61702423, and 61772327).

**REFERENCES**

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings. In *ESA*, pages 24–35, 2012.
[2] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pages 349–360, 2013.
[3] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, pages 237–248, 2014.
[4] M. A. Babenko, A. V. Goldberg, A. Gupta, and V. Nagarajan. Algorithms for hub label optimization. *TALG*, 13(1):1–17, 2016.
[5] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering*, pages 19–80, 2016.
[6] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *WWW*, pages 845–854, 2008.
[7] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *PVLDB*, 5(11):1136–1147, 2012.
[8] H. Chen, W. Ku, M. Sun, and R. Zimmermann. The partial sequenced route query with traveling rules in road networks. *GeoInformatica*, 15(3):541–569, 2011.
[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *Siam Journal on Computing*, 32(5):937–946, 2002.
[10] J. Dai, B. Yang, C. Guo, and Z. Ding. Personalized route recommendation using big trajectory data. In *ICDE*, pages 543–554, 2015.
[11] J. Dai, B. Yang, C. Guo, C. S. Jensen, and J. Hu. Path cost distribution estimation using trajectory data. *PVLDB*, 10(3), 2017.
[12] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *Proc. SEA*, pages 18–29, 2013.
[13] Z. Ding, B. Yang, Y. Chi, and L. Guo. Enabling smart transportation systems: A parallel spatio-temporal database approach. *IEEE Trans. Computers*, 65(5):1377–1391, 2016.
[14] Z. Ding, B. Yang, B. R. Güting, and Y. Li. Network-matched trajectory-based moving-object database: Models and applications. *IEEE Trans. Intelligent Transportation Systems*, 16(4):1918–1928, 2015.
[15] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.
[16] C. Guo, C. S. Jensen, and B. Yang. Towards total traffic awareness. *SIGMOD Record*, 43(3):18–23, 2014.
[17] C. Guo, B. Yang, O. Andersen, C. S. Jensen, and K. Torp. Ecomark 2.0: empowering eco-routing with vehicular environmental models and actual vehicle fuel consumption data. *GeoInformatica*, 19(3):567–599, 2015.

**TABLE I**

| Method    | CAL | NYC | COL | FLA | G+ |
|-----------|-----|-----|-----|-----|----|
| SK        |     |     |     |     |    |
| KP        |     |     |     |     |    |
| GSP       |     |     |     |     |    |
[18] C. Guo, B. Yang, O. Andersen, C. S. Jensen, and K. Torp. Ecosky: Reducing vehicular environmental impact through eco-routing. In ICDE, pages 1412–1415, 2015.
[19] C. Guo, B. Yang, J. Hu, and C. S. Jensen. Learning to route with sparse trajectory sets. In ICDE, page 12 pages, 2018.
[20] C. Guo, B. Yang, J. Hu, and C. S. Jensen. Finding Top-k Optimal Sequenced Routes. In ICDE, page 12 pages, 2018.
[21] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2):100–107, 1968.
[22] T. Hashem, S. Barua, M. E. Ali, L. Kulik, and E. Tanin. Efficient computation of trips with friends and families. In CIKM, pages 931–940, 2015.
[23] T. Hashem, T. Hashem, M. E. Ali, and L. Kulik. Group trip planning queries in spatial databases. In SSTD, pages 259–276, 2013.
[24] J. Hu, B. Yang, C. Guo, and C. S. Jensen. Risk-aware path selection with time-varying, uncertain travel costsa time series approach. VLDB Journal, to appear, 2018.
[25] J. Hu, B. Yang, C. S. Jensen, and Y. Ma. Enabling time-dependent uncertain eco-weights for road networks. GeoInformatica, 21(1):57–88, 2017.
[26] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng. On trip planning queries in spatial databases. In SSTD, pages 273–290, 2005.
[27] J. Li, Y. D. Yang, and N. Mamoulis. Optimal route queries with arbitrary order constraints. TKDE, 25(5):1097–1110, 2013.
[28] H. Liu, C. Jin, B. Yang, and A. Zhou. Finding top-k shortest paths with diversity. TKDE, 30(3):488–502, 2018.
[29] M. N. Rice and V. J. Tsotras. Engineering generalized shortest path queries. In ICDE, pages 949–960, 2013.
[30] M. N. Rice and V. J. Tsotras. Parameterized algorithms for generalized traveling salesman problems in road networks. In SIGSPATIAL, pages 114–123, 2013.
[31] S. Samrose, T. Hashem, S. Barua, M. E. Ali, M. H. Uddin, and M. I. Mahmud. Efficient computation of group optimal sequenced routes in road networks. In MDM, pages 122–127, 2015.
[32] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. VLDB J., 17(4):765–787, 2008.
[33] M. Sharifzadeh and C. Shahabi. Processing optimal sequenced route queries using voronoi diagrams. GeoInformatica, 12(4):411–433, 2008.
[34] B. Yang, J. Dai, C. Guo, and C. S. Jensen. Pace: A PAth-CEntric paradigm for stochastic path finding. VLDB Journal, online first, 2017.
[35] B. Yang, C. Guo, C. S. Jensen, M. Kaul, and S. Shang. Stochastic skyline route planning under time-varying uncertainty. In ICDE, pages 136–147, 2014.
[36] B. Yang, C. Guo, Y. Ma, and C. S. Jensen. Toward personalized, context-aware routing. VLDB Journal, 24(2):297–318, 2015.
[37] B. Yang, Q. Ma, W. Qian, and A. Zhou. TRUSTER: trajectory data processing on clusters. In DASFAA, pages 768–771, 2009.
[38] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In SIGMOD, pages 1323–1334, 2014.
[39] S. Shang, K. Zheng, C. S. Jensen, B. Yang, P. Kalnis, G. Li, and J. Wen, “Discovery of path nearby clusters in spatial networks,” IEEE Trans. Knowl. Data Eng., vol. 27, no. 6, pp. 1505–1518, 2015.