Finding Top-k Optimal Sequenced Routes

Huiping Liu*, Cheqing Jin*✉, Bin Yang†, Aoying Zhou*
hpliu@stu.ecnu.edu.cn {cqjin, ayzhou}@dase.ecnu.edu.cn byang@cs.aau.dk
*School of Data Science and Engineering, East China Normal University, China
†Department of Computer Science, Aalborg University, Denmark

***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 being extended, 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.

I. Introduction

Optimal sequenced route (OSR) querying [28], [29], a.k.a., generalized shortest path querying [25], 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 planing [13], [17], crisis management, supply chain management, video surveillance, mobility-as-a-service [12], and logistics [25], [28]. 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 [9], [24], [32].

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 $\langle MA, RE, CI\rangle$. The optimal sequenced route for Alice is $s \rightarrow a \rightarrow b \rightarrow d \rightarrow t$ with a cost of 20.

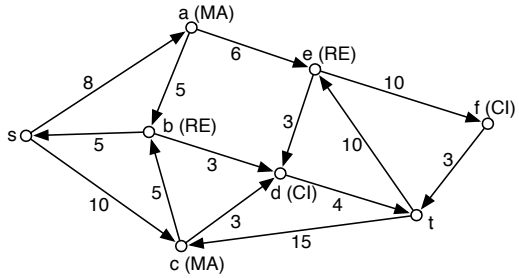


Figure 1. A road network graph G

However, if Alice prefers restaurant e to restaurant b, route $s \rightarrow a \rightarrow e \rightarrow d \rightarrow t$ with a cost of 21 is more preferable. In addition, if the shopping mall at vertex c has sale promotions, route $s \rightarrow c \rightarrow b \rightarrow d \rightarrow t$ 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 [28], the progressive neighbor exploration algorithm *PNE* is proposed to solve the OSR problem on general graphs. In [25], 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 [18], 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 label index for each category by employing hop labeling technique [2]–[5], [8] 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 A* like algorithm to further reduce the searching space for solving the KOSR problem. Third, we propose an inverted label 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 I. 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 [16], [31], 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 I clearly shows that this paper is the first comprehensive study for addressing the sequenced route problem on general graphs, with specific category orders, and $k \geq 1$, i.e., the top-k optimal sequenced route (KOSR) problem.

The optimal sequenced route querying [28], [29], a.k.a., the generalized shortest path querying [25], is the most relevant problem. [28] 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. [29] tries to improve the efficiency of optimal sequenced route querying on general graphs by pre-constructing 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. [25] 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 [14] 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.

Table I. Categorization of Sequenced Route Queries

	Euclidean Graphs	General Graphs
$k = 1$	Specific order: [28] Arbitrary order: [7], [22], [23]	Specific order: [25], [28], [29] Arbitrary order: [6], [26]
$k \geq 1$	Specific order: [19], [20], [27] Arbitrary order:$\emptyset$	Specific order: This paper Arbitrary order:$\emptyset$

Group optimal sequenced routes problem [19], [20], [27] 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.

[6], [7], [22], [23] 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 [7], [23] and budget limit [6], are also considered. Such problems are NP-hard and can be reduced to generalized traveling salesman problem [26]. 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 [15], e.g., skyline routing [16], [31], stochastic routing [10], [21], [30], and personalized routing [32], are also different from KOSR.

III. Preliminaries

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

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

Table II. NOTATION

Notation	Meaning
$P_{s,t}$	A route from s to t
C	A category sequence $C = \langle C_1, \cdots, C_j \rangle$
$\|C\|$	The number of categories in category sequence C
V_{C_i}	The vertex set of category C_i
$\|C_i\|$	The number of vertices that belong to category C_i, i.e., $\|V_{C_i}\|$
$P_{s,t,C}$	Witness $P_{s,t,C=\langle C_1,\cdots,C_j\rangle} = \langle s, v_1, \cdots, v_j, t \rangle$, such that $v_i \in V_{C_i}$ for $1 \leq i \leq j$
$\|P\|$	The number of vertices in route or witness P
$w(P)$	The weight of route or witness P
$dis(v_i, v_j)$	The least cost from vertex v_i to v_j
k	Top k results are needed

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 $P_{s,t} = \langle v_0 = s, v_1, \cdots, v_q = t \rangle$. 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 $C = \langle C_1, C_2, \cdots, C_j \rangle$ 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 $V_{C_i} = \{v | v \in V \bigwedge C_i \in F(v)\}$. 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 $C = \langle C_1, C_2, \cdots, C_j \rangle$, a route $P_{s,t} = \langle v_0 = s, v_1, \cdots, v_q = t \rangle$ is feasible if and only if there exists a subsequence of vertices $\langle v_{r_1}, v_{r_2}, \cdots, v_{r_j} \rangle$ 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 $C_i \in F(v_{r_i})$. We call $\langle s, v_{r_1}, v_{r_2}, \cdots, v_{r_j}, t \rangle$ 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 w.r.t a category sequence, they are regarded as the same feasible route and only the route with smaller cost is considered. Formally, for a witness $P_{s,t,C} = \langle v_0, v_1, \cdots, v_q \rangle$, its cost $w(P_{s,t,C})$ is defined as $w(P_{s,t,C}) = \sum_{0 \leq i < q} dis(v_i, v_{i+1})$, where $dis(v_i, v_{i+1})$ 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 w.r.t C, $\Psi = \{P^1_{s,t}, P^2_{s,t}, \cdots, P^k_{s,t}\}$, such that there does not exist any other feasible route $P'_{s,t}$ in G where $P'_{s,t} \notin \Psi \bigwedge w(P'_{s,t}) < \max_{1 \leq i \leq k} w(P^i_{s,t})$.

[1]Note that a witness may not represent a route according to Definition 2 as consecutive vertices in a witness may not be connected by an edge.

Example 1: Consider the graph G in Figure 1, the KOSR query $(s, t, \langle MA, RE, CI \rangle, 3)$ returns Ψ=$\{\langle s, a, b, d, t \rangle$, $\langle s, a, e, d, t \rangle$, $\langle s, c, b, d, t \rangle\}$ 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 first present PNE, one of state-of-the-art methods for solving OSR. Then we present the baseline KPNE, which is extended from PNE, for solving KOSR. Another state-of-the-art method GSP for OSR is included in the full version [1].

The progressive neighbor exploration (PNE) algorithm [28] 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 $\langle v_0, v_1, \cdots, v_{q-1}, v_q \rangle$ 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 $dis(v_q, v_{q+1}) = \arg\min_{v \in V_{C_{q+1}}} dis(v_q, v)$, is considered. Moreover, to guarantee the correctness, another candidate route derived from $\langle v_0, v_1, \cdots, v_{q-1}, v_q \rangle$ is incrementally generated by extending $\langle v_0, v_1, \cdots, v_{q-1} \rangle$ via v_{q-1}'s next nearest neighbor v'_q in C_q, such that $v'_q \neq v_q$ and $dis(v_{q-1}, v'_q) \geq dis(v_{q-1}, v_q)$. 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 to add a result set and each time we find an optimal sequenced route (line 5), it will be added to the result set, when the result set consists of k routes or the priority queue is already empty, the set will be returned as the result of KOSR. We refer to this method for solving KOSR as *KPNE*.

Although KPNE which is extended from [28] is able to solve KOSR on general graphs, it is inefficient since all partially explored candidate routes whose costs are smaller than the cost of the k-th optimal sequenced route must be examined. In the worst case, the number of examined partially explored candidate routes at category C_i can reach $\prod_{1 < j \leq i} |C_j|$, as a result, the total number of routes to be examined by KPNE can be $\sum_{1 \leq i \leq |C|+1} \prod_{1 \leq j \leq i} |C_j|$, which is too huge to process on large graphs.

Algorithm 1: $PNE(G, s, t, C)$

Input: Graph $G(V, E)$, source-destination pair $s, t \in V$, category sequence $C = \langle C_1, \cdots, C_j \rangle$.
Output: The optimal sequenced route.

Priority queue $Q \leftarrow \{\langle s \rangle\}$;
while $|Q| > 0$ **do**
  $\langle v_0 = s, v_1, \cdots, v_{q-1}, v_q \rangle \leftarrow Q$.extractMin();
  if $q = |C| + 1$ **then**
    return $\langle v_0, v_1, \cdots, v_{q-1}, v_q \rangle$;
  // extend route.
  $v_{q+1} \leftarrow v_q$'s nearest neighbor in category C_{q+1};
  Q.insert($\langle v_0, v_1, \cdots, v_q, v_{q+1} \rangle$);
  // generate candidate route.
  if $q > 0$ **then**
    $v'_q \leftarrow v_{q-1}$'s next nearest neighbor in C_q;
    // $v'_q \neq v_q \bigwedge dis(v_{q-1}, v'_q) \geq dis(v_{q-1}, v_q)$
    Q.insert($\langle v_0, v_1, \cdots, v_{q-1}, v'_q \rangle$);

IV. Proposed solutions for KOSR

In this section, we propose two efficient methods to solve KOSR. We first describe a method based on the route *dominance* relationship to filter unnecessary 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 A* manner in Section IV-B.

A. Dominance Based Algorithm

We first illustrate the intuition of the route dominance relationship. Consider a KOSR query $(s, t, \langle MA, RE, CI \rangle, 2)$ in Figure 1. In order to find the first optimal sequenced route $\langle s, a, b, d, t \rangle$ with the cost of 20 (shorten as $\langle s, a, b, d, t \rangle(20)$), KPNE will attempt to examine and extend $\langle s, a, b \rangle(13)$ and $\langle s, c, b \rangle(15)$, because both $\langle s, a, b \rangle$ and $\langle s, c, b \rangle$ have a smaller cost than $\langle s, a, b, d, t \rangle$. However, there is no need to extend $\langle s, c, b \rangle$ to find $\langle s, a, b, d, t \rangle$, because the cost of the optimal feasible route extended from $\langle s, c, b \rangle$ won't be smaller than that of $\langle s, a, b \rangle$ (i.e., $\langle s, a, b, d, t \rangle$). Hence, $\langle s, c, b \rangle$ can be excluded to be extended until the optimal sequenced route $\langle s, a, b, d, t \rangle$ is found. In this case, we say $\langle s, c, b \rangle$ is *dominated* by $\langle s, a, b \rangle$. Next, we formally define the dominance relationship.

Definition 6 (Dominance): Consider a given category sequence $C = \langle C_1, \cdots, C_j \rangle$ and two partially explored candidate routes (witnesses) $P_1 = \langle s, v_1^1, \cdots, v_q^1 \rangle$ and $P_2 = \langle s, v_1^2, \cdots, v_q^2 \rangle$ $(1 \leq q \leq j)$. If $v_q^1 = v_q^2$ 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 $(s, t, C{=}\langle C_1, \cdots, C_j \rangle, k)$ 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 $P_1{=}\langle s, v_1^1, \cdots, v_q^1 \rangle$, $P_2{=}\langle s, v_1^2, \cdots, v_q^2 \rangle$ $(1 \leq q \leq j)$ and $P_1^*{=}\langle s, v_1^1, \cdots, v_q^1, v_{q+1}, \cdots, v_j, t \rangle$, since P_1^* is the optimal feasible route extended from P_1, $P = \langle v_q^1, v_{q+1}, \cdots, v_j, t \rangle$ must be the optimal sequenced route for category sub-sequence $\langle C_{q+1}, \cdots, C_j \rangle$ from v_q^1 to t. Because $P_1 \prec_C P_2$, we have $v_q^2 = v_q^1$ and $w(P_1) \leq w(P_2)$, thus, P_2^* can be represented by $\langle s, v_1^2, \cdots, v_q^2, v_{q+1}, \cdots, v_j, t \rangle$, 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 $w(P_1^*) \leq w(P_2^*)$. ■

Algorithm 2: $PruningKOSR(G, s, t, C, k)$

Input: Graph $G(V, E)$, source-destination pair $s, t \in V$, category sequence $C = \langle C_1, \cdots, C_j \rangle$, and k.
Output: The top-k optimal sequenced routes.

$\forall v \in V$, initialize $v.\mathbf{HT}_{\prec_C}$ and $v.\mathbf{HT}_{\succ_C}$;
$\Psi \leftarrow \emptyset$;
Priority queue $Q \leftarrow \{(\langle s \rangle, 1)\}$; /* $(route, x)$ */
while Q *is not empty and* $|\Psi| < k$ **do**
  $p = (\langle v_0, v_1, \cdots, v_{q-1}, v_q \rangle, x) \leftarrow Q$.extractMin();
  if $q = |C| + 1$ **then**
    $\Psi \leftarrow \Psi \bigcup \{p\}$;
    // reconsider dominated routes
    for *each* $i = 1 \cdots q - 1$ **do**
      if $\langle v_0, \cdots, v_i \rangle = v_i.\mathbf{HT}_{\prec_C}.getValue(i + 1)$ **then**
        $p' = (\langle v_0, v'_1, \cdots, v_i \rangle, x) \leftarrow$ $v_i.\mathbf{HT}_{\succ_C}$.getValue$(i + 1)$.extractMin();
        Q.insert$(p' = (\langle v_0, v'_1, \cdots, v_i \rangle, -))$;
        $v_i.\mathbf{HT}_{\prec_C}$.remove$(i + 1)$;
  else
    // pruning dominated routes
    if $|p| \notin v_q.\mathbf{HT}_{\prec_C}.KeySet$ **then**
      $v_q.\mathbf{HT}_{\prec_C}$.add$(|p|, \langle v_0, \cdots, v_q \rangle)$;
      $v_{q+1} \leftarrow$ FindNN$(v_q, C_{q+1}, 1)$;
      Q.insert$((\langle v_0, v_1, \cdots, v_q, v_{q+1} \rangle, 1))$;
    else
      $v_q.\mathbf{HT}_{\succ_C}$.getValue$(|p|)$.insert$(p)$;
    if $q > 0$ **then**
      $v'_q \leftarrow$ FindNN$(v_{q-1}, C_q, x + 1)$;
      Q.insert$((\langle v_0, v_1, \cdots, v_{q-1}, v'_q \rangle, x + 1))$;
return Ψ;

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 tables in the form of $(key, value)$ pairs. One is $\mathbf{HT}_{\prec_C}$ for dominating routes, where key is the size of the partially

explored dominating route that has been extended at v, and the $value$ is the route itself. Another one is $\mathbf{HT}_{\succ_C}$ for dominated routes, where key represents the size of dominated route, and $value$ is a priority queue for the routes with the size of 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 Ψ for the top-k optimal sequenced routes and a global priority queue Q for partially explored routes (witnesses) sorted by their costs in an ascending order. Moreover, for each route $p = \langle v_0, v_1, \cdots, v_{q-1}, v_q \rangle$, we introduce an additional attribute x to indicate that v_q is the x-th nearest neighbor of v_{q-1} in category C_q 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 $p = \langle v_0, v_1, \cdots, v_{q-1}, v_q \rangle$ to be examined, if p is the first route with size $|p|$ that reaches vertex v_q, we add p to the $\mathbf{HT}_{\prec_C}$ of v_q and extend it via v_q's nearest neighbor in category C_{q+1} (lines 14–17). Otherwise, if its size $|p|$ is in the $\mathbf{HT}_{\prec_C}$ of v_q, it means that another route with size $|p|$ and smaller cost has been reached and extended at v_q, so that p is dominated. According to Lemma 1, there is no need to extend p anymore, therefore, we insert it into the $\mathbf{HT}_{\succ_C}$ of v_q 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 v_{q-1} has been generated in previous iterations, we need to find v_{q-1}'s $(x+1)$-th nearest neighbor in category C_q, v'_q, by invoking algorithm FindNN, and create candidate route $\langle v_0, v_1, \cdots, v_{q-1}, v'_q \rangle$ 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 v_i in p, if $\langle v_0, \cdots, v_i \rangle$ dominates the routes with size of $i+1$ in the $\mathbf{HT}_{\succ_C}$ of v_i (line 9), we only reconsider the dominated route p' with the least cost in the $\mathbf{HT}_{\succ_C}$ of v_i, because other routes in $\mathbf{HT}_{\succ_C}$ of v_i are dominated by p'. This also explains why we use a priority queue as the $value$ in hash table $\mathbf{HT}_{\succ_C}$. 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 $\langle v_0, \cdots, v_i \rangle$ from the $\mathbf{HT}_{\prec_C}$ of v_i, so that the next candidate route that reaches v_i 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 possible partially explored candidate routes are considered (lines 14–17 and 20–22) except for the dominated routes (line 19) which can be removed from extending according to Lemma 1. After an optimal sequenced route is found, the dominated routes that can be extended to be the next optimal sequenced route are reconsidered. Therefore, Algorithm 2 returns the correct result for a KOSR query. ■

Table III. RUNNING EXAMPLE OF ALG. 2 FOR FIG. 1

(a) Routes in the priority queue Q

Step	Routes (route(cost), x)
1	$(\langle s \rangle(0), 1)$
2	$(\langle s, a \rangle(8), 1)$
3	$(\langle s, c \rangle(10), 2), (\langle s, a, b \rangle(13), 1)$
4	$(\langle s, a, b \rangle(13), 1), (\langle s, c, b \rangle(15), 1)$
5	$(\langle s, a, e \rangle(14), 2), (\langle s, c, b \rangle(15), 1), (\langle s, a, b, d \rangle(16), 1)$
6	$(\langle s, c, b \rangle(15), 1), (\langle s, a, b, d \rangle(16), 1), (\langle s, a, e, d \rangle(17), 1)$
7	$(\langle s, a, b, d \rangle(16), 1), (\langle s, a, e, d \rangle(17), 1), (\langle s, c, e \rangle(27), 2)$
8	$(\langle s, a, e, d \rangle(17), 1), (\langle s, a, b, d, t \rangle(20), 1), (\langle s, c, e \rangle(27), 2),$ $(\langle s, a, b, f \rangle(40), 2)$
9	$(\langle s, a, b, d, t \rangle(20), 1), (\langle s, a, e, f \rangle(24), 2), (\langle s, c, e \rangle(27), 2),$ $(\langle s, a, b, f \rangle(40), 2)$
10	$(\langle s, c, b \rangle(15), -), (\langle s, a, e, d \rangle(17), -), (\langle s, a, e, f \rangle(24), 2),$ $(\langle s, c, e \rangle(27), 2), (\langle s, a, b, f \rangle(40), 2)$
11	$(\langle s, a, e, d \rangle(17), -), (\langle s, c, b, d \rangle(18), 1), (\langle s, a, e, f \rangle(24), 2),$ $(\langle s, c, e \rangle(27), 2), (\langle s, a, b, f \rangle(40), 2)$
12	$(\langle s, c, b, d \rangle(18), 1), (\langle s, a, e, d, t \rangle(21), 1), (\langle s, a, e, f \rangle(24), 2),$ $(\langle s, c, e \rangle(27), 2), (\langle s, a, b, f \rangle(40), 2)$
13	$(\langle s, a, e, d, t \rangle(21), 1), (\langle s, c, b, d, t \rangle(22), 2), (\langle s, a, e, f \rangle(24), 2),$ $(\langle s, c, e \rangle(27), 2), (\langle s, a, b, f \rangle(40), 2), (\langle s, c, b, f \rangle(42), 2)$

(b) Hash tables of vertex b with respect to Table III(a)

Step	$\mathbf{HT}_{\prec_C}$	$\mathbf{HT}_{\succ_C}$
1	$\emptyset$	$\emptyset$
4	$(3, \langle s, a, b \rangle)$	$\emptyset$
6	$(3, \langle s, a, b \rangle)$	$(3, \{\langle s, c, b \rangle(15)\})$
9	$\emptyset$	$(3, \{\})$
10	$(3, \langle s, c, b \rangle)$	$(3, \{\})$

Example 2: Consider Figure 1. Suppose the given query is $(s, t, \langle MA, RE, CI \rangle, 2)$. Table III shows the routes in the priority queue Q at each step and the hash tables of vertex b at different steps. At step 1, route $\langle s \rangle$ is added to the queue, then it is extended via a (s's nearest neighbor in category MA), and no candidate route can be generated. At step 2, $\langle s, a \rangle$ is examined, it is extended via b (a's nearest neighbor in category RE) and candidate route $\langle s, c \rangle$ is generated via s's 2nd nearest neighbor in category MA. At step 4, $\langle s, a, b \rangle$ is examined and extended at b, we insert it into the $\mathbf{HT}_{\prec_C}$ of b. Subsequently, at step 6, since $\langle s, c, b \rangle$ is dominated by $\langle s, a, b \rangle$ in the $\mathbf{HT}_{\prec_C}$ of b , $\langle s, c, b \rangle$ won't be extended at b, instead, we insert $\langle s, c, b \rangle$ into the $\mathbf{HT}_{\succ_C}$ of b, and generate candidate route $\langle s, c, e \rangle$ via c's 2nd nearest neighbor e in category RE. At step 9, the first optimal sequenced route $\langle s, a, b, d, t \rangle$ is found. Since both $\langle s, c, b \rangle$ and $\langle s, a, e, d \rangle$ in $\mathbf{HT}_{\succ_C}$ of b and d, respectively, are dominated by $\langle s, a, b \rangle$ and $\langle s, a, b, d \rangle$ in $\mathbf{HT}_{\prec_C}$ of b and d, respectively, we re-add them into the queue with x='-' and remove the corresponding dominating routes from $\mathbf{HT}_{\prec_C}$. Finally, at step 13, the second optimal sequenced route $\langle s, a, e, d, t \rangle$ is found, and we return $\{\langle s, a, b, d, t \rangle, \langle s, a, e, d, t \rangle\}$ 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 $\langle s, t, C, k \rangle$, 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 v (line 16 in Algorithm 2), and at most $|C_{i+1}| - 1$ candidate routes can be generated via v's next nearest neighbors in category C_{i+1} (line 21 in Algorithm 2).

As a result, in the worst case, the number of routes to be examined by Algorithm 2 for the first optimal sequenced route is $\sum_{0\leq i\leq |C|} |C_i| \cdot |C_{i+1}|$, in which $\sum_{0\leq i\leq |C|} |C_i|$ 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 $\sum_{0\leq i\leq |C|} |C_i| \cdot |C_{i+1}| + (k-1) \cdot \sum_{2\leq i\leq |C|+1} |C_i|$ partially explored routes need to be examined, in which $\sum_{0\leq i\leq |C|} |C_i| + (k-1) \cdot |C|$ routes are extended. Compared to KPNE, the searching space is reduced from exponential complexity ($\sum_{1\leq i\leq |C|+1} \prod_{1\leq j\leq i} |C_j|$) down to polynomial complexity ($\sum_{0\leq i\leq |C|} |C_i| \cdot |C_{i+1}| + (k-1) \cdot \sum_{2\leq i\leq |C|+1} |C_i|$). Lemma 3 shows the time complexity of Algorithm 2.

Lemma 3: Given a KOSR query (s, t, C, k), let $M = \sum_{0\leq i\leq |C|} |C_i| \cdot |C_{i+1}| + (k-1) \cdot \sum_{2\leq i\leq |C|+1} |C_i|, N = \sum_{0\leq i\leq |C|} |C_i| + (k-1) \cdot |C|$, the time complexity of Algorithm 2 is $O(M\rho + M \log N)$, where ρ 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 next 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 $O(M \log N)$. In summary, the total time complexity of Algorithm 2 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_i in category C_{i+1} is that by using Dijkstra's search. We start from v_i 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 [2], [3], [8] 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 $d_{u,v}$=$dis(u, v)$. Similarly, $L_{out}(v)$ consists of a set of label entries in the form of $(u', d_{v,u'})$, where $u' \in V$ is a vertex that can be reached by v, and $d_{v,u'}$=$dis(v, u')$. 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 $dis(s, t)$ 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 [8]. Thus, existing methods [2]–[4], [8] are all heuristic to approximate the minimal 2-hop labeling index. Alternatively, we may use an all-pairs shortest path algorithm to generate index. Although it works, it requires index size of $O(|V|^2)$, which is not acceptable for large graphs.

Table IV. A LABEL INDEX FOR FIG. 1

Vertex	$L_{in}(v)$	$L_{out}(v)$
a	$(a,0), (s,8), (t,33)$	$(a,0), (b,5), (e,6), (s,10), (t,12)$
b	$(b,0), (s,13), (t,20)$	$(b,0), (s,5), (t,7)$
c	$(c,0), (s,10), (t,15)$	$(b,5), (c,0), (d,3), (s,10), (t,7)$
d	$(b,3), (d,0), (e,3), (s,13), (t,13)$	$(d,0), (t,4)$
e	$(e,0), (s,14), (t,10)$	$(e,0), (t,7)$
f	$(e,10), (f,0), (s,24), (t,20)$	$(f,0), (t,3)$
s	$(s,0), (t,25)$	$(s,0), (t,17)$
t	$(t,0)$	$(t,0)$

Table V. THE INVERTED LABEL INDEX OF CATEGORY MA, $IL(MA)$

Inverted label $IL(v)$	**Label entries**
$IL(a)$	$(a,0)$
$IL(c)$	$(c,0)$
$IL(s)$	$(a,8), (c,10)$
$IL(t)$	$(c,15), (a,33)$

Example 3: For the directed weighted graph in Figure 1, a possible 2-hop label indexes L_{in} and L_{out} is shown in Table IV. Suppose we compute the least cost from vertex a to vertex c, i.e., $dis(a, c)$, we look up $L_{out}(a)$ and $L_{in}(c)$, and find the matching label entries $(s, 10), (t, 12)$ in $L_{out}(a)$ and $(s, 10), (t, 15)$ in $L_{in}(c)$, respectively. Since $10 + 10 = 20 < 12 + 15 = 27$, we return 20 as the result of $dis(a, c)$.

By using the label index, an easy way to find the x-th nearest neighbor of v_i in category C_{i+1} is, for each $u \in V_{C_{i+1}}$, compute $dis(v_i, u)$ by looking up $L_{out}(v_i)$ and $L_{in}(u)$. By maintaining a min heap of size x, the x-th nearest neighbor of v_i is the vertex u with the x-th least $dis(v_i, u)$ among all vertices in $V_{C_{i+1}}$. Therefore, the time complexity is $O(\Sigma_{u\in V_{C_{i+1}}}(|L_{out}(v_i)| + |L_{in}(u)|) + |C_{i+1}| \log x)$, which is inefficient for categories with many vertices in large graphs. 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 $u \in V_{C_i}$. That is, $IL(u')$ consists of a list of label entries $(u, d_{u',u})$, such that $u \in V_{C_i}$ and $(u', d_{u',u}) \in L_{in}(u)$, and all label entries in $IL(u')$ are sorted by their costs, i.e., $d_{u',*}$, in an ascending order. With the inverted label index $IL(C_i)$, for each label entry $(u', d_{v,u'}) \in L_{out}(v)$, 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.

Example 4: Table V shows the inverted label index of category MA with respect to the label index in Table IV. Let's find the nearest neighbor of s in category MA. Since $L_{out}(s) = \{(s, 0), (t, 17)\}$, we look up $IL(s)$ and $IL(t)$ in $IL(MA)$. Because label entries are sorted, only $(a, 8)$ in $IL(s)$ and $(c, 15)$ in $IL(t)$ need to be considered, then the nearest neighbor of s in MA is a with the cost of $0 + 8 = 8$.

Based on the inverted label index, the detail process of finding the x-th nearest neighbor of v_i in category C_{i+1} is described by Algorithm 3. To avoid overlapping search, 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 keep the candidate label entries $(u', d_{v',u'})$ in matching inverted label $IL(v')$ such that $(v', d_{v_i,v'}) \in L_{out}(v_i)$ that have been found so far into a priority queue NQ and all entries are sorted by $d_{v_i,v'} + d_{v',u'}$ 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 v' and $value$ is the entry position of $IL(v')$. NL, NQ and KV 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 $IL(v') \in IL(C_{i+1})$, then insert the first label entry of each $IL(v')$ into NQ and initialize KV (lines 6–10). Subsequently, we get the minimal label entry $(u, d_{v',u})$ in NQ which is the next nearest neighbor (line 11). In addition, we add the next label in $IL(v')$ 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, we find the 2nd nearest neighbor of s in category MA. Let's follow Example 4, since the 1st nearest neighbor of s is a from $IL(s)$, after finding a, for s, $NL = \{a\}$, $NQ = \{(c, 10), (c, 15)\}$ and $KV = \{\langle s, 2\rangle, \langle t, 1\rangle\}$. 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 '-'. At this point, $KV = \{\langle s, -\rangle, \langle t, 1\rangle\}$, $NL = \{a, c\}$ and $NQ = \{(c, 15)\}$. We return c as the 2nd nearest neighbor of s with the cost of $0 + 10 = 10$.

After the inverted label index is constructed offline, finding the 1st nearest neighbor of v takes $O(|L_{out}(v)| \log |L_{out}(v)|)$ time, because it scans all label entries in $L_{out}(v)$ and adds the first label entry of the matching inverted label to the priority queue, and it only takes $O(\log |L_{out}(v)|)$ time to find the next nearest neighbors, which is very efficient. Let's reconsider Lemma 3, suppose the average index size of $L_{out}(v)$ for all $v \in V$ is $|L_{out}|$, the expected complexity of Algorithm 2 will be $O(N|L_{out}| \log |L_{out}| + (M - N) \log |L_{out}| + M \log N)$.

Algorithm 3: $FindNN(v_i, C_{i+1}, x)$

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

1 $NL \leftarrow$ list of v_i's neighbors in C_{i+1} that have been found;
2 $NQ \leftarrow$ priority queue of v_i for the label entries in $IL(v) \in IL(C_{i+1})$;
3 $KV \leftarrow v_i$'s hash table structure for $IL(v) \in IL(C_{i+1})$;
4 **if** $|NL| \geq x$ **then**
5 **return** $NL[x]$;
6 **if** $|NL| = 0$ **then**
7 **for** *each label entry* $(v', d_{v_i,v'}) \in L_{out}(v_i)$ **do**
8 $(u', d_{v',u'}) \leftarrow IL(v')[1]$;
9 NQ.insert($(u', d_{v',u'})$);
10 KV.add($v', 1$);
11 $(u, d_{v',u}) \leftarrow NQ$.extractMin();
12 **do**
13 KV.add(v', KV.get(v') $+ 1$);
14 $(u', d_{v',u'}) \leftarrow IL(v')[KV$.get($v'$)$]$;
15 **while** $u' \notin NL$;
16 NQ.insert($(u', d_{v',u'})$);
17 NL.add(u);
18 **return** u;

Given a witness that we have found, to get the corresponding actual route, we need to restore the route between consecutive vertices in the witness. By adding a parent vertex in each label entry of the hop labeling, it is easy to construct the actual route between two vertices [3]. Hence, the actual route can be restored by concatenating all sub-routes between consecutive vertices in the witness.

B. Integrating A* Heuristic Estimation

Inspired by A* algorithm [18], the efficiency of KOSR can be further improved by using a destination-based strategy. To quickly find the feasible route, the partially explored candidate routes with a smaller cost but far away from the destination should be given lower priority to be examined, so that 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 $\langle s, t, C, k\rangle$, for a partially explored candidate route (witness) $p = \langle v_0 = s, v_1, \cdots, v_i\rangle$, the optimal feasible route extended from p can be represented as $p' = \langle s, v_1, \cdots, v_i, v_{i+1}, \cdots, v_{|C|}, t\rangle$, so that $w(p') = w(p) + w(\langle v_i, v_{i+1}, \cdots, v_{|C|}, t\rangle)$. That is, we need to estimate the cost of $\langle v_i, v_{i+1}, \cdots, v_{|C|}, t\rangle$ which is the optimal sequenced route starts from v_i and passes through p's

Algorithm 4: $FindNEN(v_i, C_{i+1}, x)$

Input: Vertex v_i, category C_{i+1}, integer x.
Output: Vertex v_{i+1} in C_{i+1} such that $dis(v_i, v_{i+1}) + dis(v_{i+1}, t)$ is the x-th least.

1 $ENL \leftarrow$ list of v_i's estimated neighbors in C_{i+1} that have been found;
2 $ENQ \leftarrow$ priority queue of v_i for the candidate neighbors in C_{i+1};
3 $ln \leftarrow v_i$'s last nearest neighbors that have been computed in C_{i+1};
4 **if** $|ENL| \geq x$ **then**
5 **return** $ENL[x]$;
6 **while** $(|ENL| = 0 \bigwedge |ENQ| = 0) \bigvee (ln \neq NULL \bigwedge dis(v_i, ln) < dis(v_i, v) + dis(v, t))$, *where* v *is the vertex in* ENQ *with minimal* $dis(v_i, v) + dis(v, t)$ **do**
7 **if** $ln \neq NULL$ **then**
8 ENQ.insert(ln);
9 $ln \leftarrow$ FindNN($v_i, C_{i+1}, |ENL| + |ENQ| + 1$);
10 $v_{i+1} \leftarrow ENQ$.extractMin();
11 ENL.add(v_{i+1});
12 **return** v_{i+1};

remaining categories and reaches the destination t. We say that a heuristic estimation h for a route P is *admissible* if $h(P) \leq w(P)$. 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 $dis(v_i, t) \leq w(\langle v_i, v_{i+1}, \cdots, v_{|C|}, t\rangle)$, which means $dis(v_i, t)$ is an admissible estimation of the cost of route $\langle v_i, v_{i+1}, \cdots, v_{|C|}, t\rangle$. Therefore, the estimated cost of p' is $w(p) + dis(v_i, t)$. By applying this target-directed estimation, we propose another improved method *StarKOSR*. Instead of ordering the routes, i.e., $p = \langle v_0, v_1, \cdots, v_i\rangle$, in the priority queues (Q and priority queues in $\mathbf{HT}_{\succ C}$) by their real costs, i.e., $w(p)$, in StarKOSR, we order routes by their estimated costs, i.e., $w(p) + dis(v_i, t)$, 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 v_i, we find v_i's neighbor v_{i+1} in category C_{i+1} such that $dis(v_i, v_{i+1}) + dis(v_{i+1}, t)$ is the x-th least among all vertices in C_{i+1}, we call v_{i+1} the x-th *nearest estimated neighbor* of v_i. To this end, we devise the algorithm FindNEN (Algorithm 4).

Given vertex v_i, category C_{i+1} and integer x, Algorithm 4 finds v_i's x-th nearest estimated neighbor, v_{i+1}, in C_{i+1}. To avoid computing v_i's x-th nearest estimated neighbor multiple times, we maintain an array list ENL of v_i 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 v_i for candidate neighbors that have been considered so far and sort the neighbors (v) by their estimated costs ($dis(v_i, v) + dis(v, t)$) in an ascending order. Meanwhile, we store the last nearest neighbor of v_i that have been computed into variable ln, so that we can

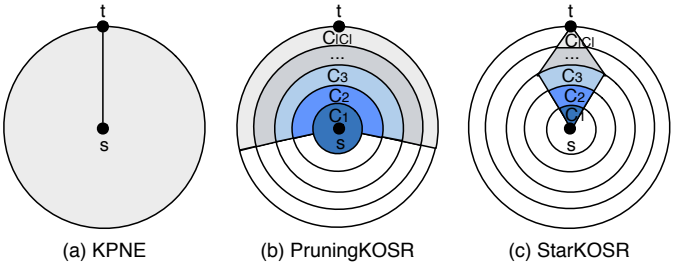


Figure 2. The searching space of different methods

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 v_i 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 C_{i+1} to find the x-th nearest estimated neighbor, we incrementally find the next nearest neighbor ln of v_i in C_{i+1} by calling FindNN (line 9), if $dis(v_i, ln)$ is greater than the minimal cost $dis(v_i, v) + dis(v, t)$ in ENQ, then v has the minimal estimated cost among all remaining vertices in C_{i+1}, because other vertices, say v', that have not been checked hold $dis(v_i, v') \geq dis(v_i, ln) \geq dis(v_i, v) + dis(v, t)$, which means that their estimated cost cannot be less than that of v. Since FindNEN 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). A concrete example of StarKOSR is included in the full version [1].

Lemma 4: Algorithm StarKOSR returns the correct result for a KOSR query $(s, t, C = \langle C_1, \cdots, C_j\rangle, k)$.

Proof: We include the proof in the full version [1] due to space limitation. ■

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: suppose we examine route $p = \langle v_0, \cdots, v_i\rangle$, 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 $dis(v_i, NN_y) < dis(v_i, v_{i+1}) + dis(v_{i+1}, t)$ for each nearest neighbor $NN_y, 1 \leq y < j$. If $w(p) + dis(v_i, v_{i+1}) + dis(v_{i+1}, t) < w(P)$, where $w(P)$ is the cost of the next optimal feasible route P, then $w(p) + dis(v_i, NN_y) < w(P)$, that is, to find P, $j - 1$ candidate routes $\langle v_0, \cdots, v_i, NN_y\rangle$ should be examined in PrunningKOSR by calling FindNN j times. In this case, both methods call FindNN the same times. On the other hand, if $w(p) + dis(v_i, v_{i+1}) + dis(v_{i+1}, t) \geq w(P)$, then $\langle v_0, \cdots, v_i, v_{i+1}\rangle$ won't be examined and subsequently, all possible candidate routes derived from $\langle v_0, \cdots, v_i, v_{i+1}\rangle$ 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 first optimal sequenced route. Since KPNE

examines all possible candidate routes that with smaller costs than the optimal sequenced 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 $|C_i|\cdot|C_{i+1}|$ 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.

Extensions. Our methods can be easily extended to solve variants of KOSR, handle dynamic updates on label index and process disk-based query answering. We include the extensions in the full version [1] due to space limitation.

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 VI gives the sizes in terms of the cardinalities of both vertex and edge sets.

Table VI. Real-World Graphs

Dataset	$\|V\|$	$\|E\|$
CAL[1]	68,345	68,990
NYC[2]	980,632	1,280,981
COL[3]	435,666	1,057,066
FLA[3]	1,070,376	2,687,902
$G+$[4]	107,614	13,673,453

In particular, CAL is a weighted, undirected graph where edge weights represent the distances of the corresponding roads. In addition, 47,298 vertices in CAL 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 [25] 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 [28], 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.

Table VII. Parameter Settings

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 [3], which achieves good performance and is easy to implement, to precompute the label index for each graph in Table VI. Based on the label index, we then construct the inverted label index for each category in the graph. Table VIII 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 (see the full version [1] for details). Alternatively, labeling compression method [11] can be applied to further reduce the index sizes.

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 III-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: StartKOSR with label indexes resident on disks. (6) KPNE-Dij, PK-Dij, SK-Dij: the KPNE, PruningKOSR, and StarKOSR algorithms by using

[1] http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm
[2] http://www.openstreetmap.org
[3] http://www.dis.uniroma1.it/challenge9/download.shtml
[4] http://snap.stanford.edu/data/index.html

Table VIII. PREPROCESSING RESULTS ON DIFFERENT GRAPHS

For label indexes				
Graph	**Time [H:M]**	**Avg.** $\lvert L_{in}(v)\rvert$	**Avg.** $\lvert L_{out}(v)\rvert$	**Index Size**
CAL	0:1	122.90	122.90	95.24MB
NYC	0:52	704.94	704.94	14.11GB
COL	0:9	1,101.06	1,101.06	5.05GB
FLA	2:42	1,495.84	1,495.84	18.25GB
$G+$	0:2	335.53	347.96	230.47MB
For inverted label indexes				
Graph	**Time [H:M]**	**Avg.** $\lvert IL(C_i)\rvert$	**Avg.** $\lvert IL(v)\rvert$	**Index Size**
CAL	0:1	9543.48	13.75	49.05MB
NYC	0:1	10863.24	12.61	120.50MB
COL	0:3	98,345.80	121.94	2.53GB
FLA	0:14	181,763.43	82.32	9.18GB
$G+$	0:1	39,621.12	91.13	113.32MB

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)∼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 $|C_i|$, 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 KPNE-Dij, PK-Dij and SK-Dij cannot return the results on $G+$.

Figure 3(b) and Figure 3(c) show the number of examined routes and NN queries, respectively, in different methods on different graphs. Clearly, the number of examined routes and NN queries in SK is much fewer than PK on all graphs, which means the searching space of SK is much smaller than that of PK. As a result, SK significantly outperforms PK. Note that the average NN queries per vertex in each examined route of SK is much greater than that of PK, for example, about 4 vs. 1 on CAL, 50 vs. 1 on FLA, and 217 vs. 1 on $G+$, because SK needs to compute more nearest neighbors to find the next nearest estimated neighbor. However, the total times of NN queries of SK is significantly less than that of PK, which also explains its excellent query run-time.Note that different index loading methods (in memory vs. disk) and NN query algorithms (FindNN vs. Dijkstra's search) do not change the process of the KOSR algorithm, hence SK and SK-DB, KPNE (or PK or SK) and KPNE-Dij (or PK-Dij or SK-Dij) have the same number of examined routes and NN queries.

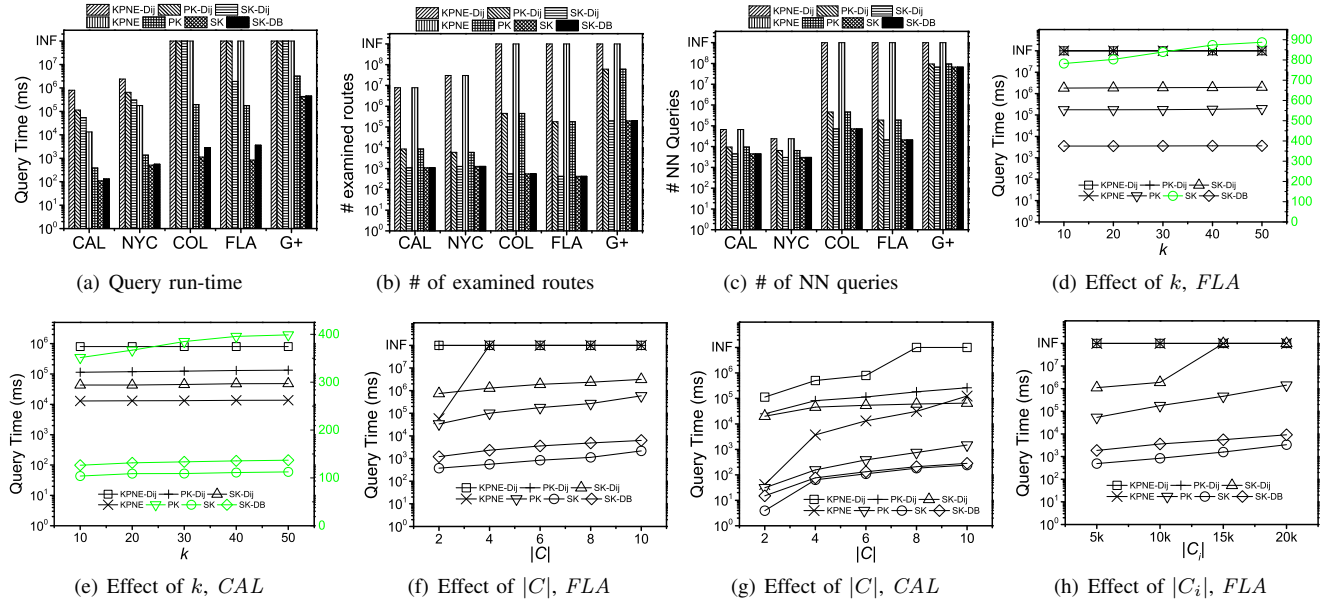
Table IX. DISTRIBUTIONS OF THE QUERY TIME (MS) ON FLA

	PK	**SK**
Overall query time	177,622.60	838.96
NN query time	177,175.84	732.87
Priority queue maintenance time	303.68	0.11
Estimation time	0	101.99
Others time	143.08	3.99

Table IX 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 4 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 4 is also consistent with the intuition shown in Figure 2(c).

Next, we show performance while varying important pa-



(a) Query run-time (b) # of examined routes (c) # of NN queries (d) Effect of k, FLA

(e) Effect of k, CAL (f) Effect of $|C|$, FLA (g) Effect of $|C|$, CAL (h) Effect of $|C_i|$, FLA

Figure 3. Performance of different methods with different parameter settings for KOSR queries

rameters. 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 ks. 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. When k is small, i.e., 2, 3, 4, 5, we include the results in the full version [1] due to space limitation.

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.

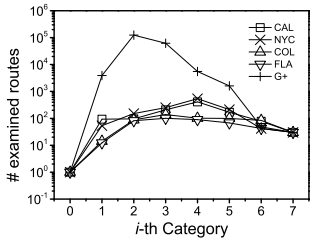


Figure 4. Searching space of SK at different categories

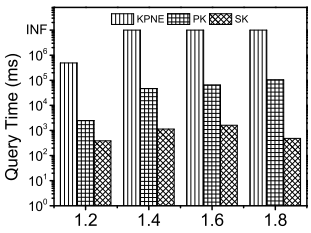


Figure 5. Performance on zipfian distribution

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 3. 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 5 illustrates the results of different f with $|C| = 6, k = 30$ on FLA. Clearly, our methods greatly outperforms 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 3), 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 6 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 runtime 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.

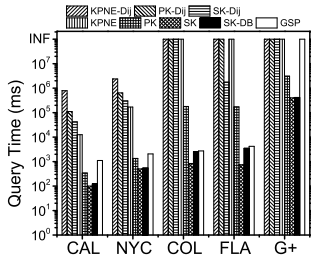


Figure 6. Performance of different methods for OSR queries

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 I to solve the KOSR querying when partial or arbitrary category orders and personal preferences for categories are allowed. It is also of interest to explore how to solve KOSR in parallel, e.g., using MapReduce [33].

Acknowledgement

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

References

[1] Full version. https://arxiv.org/pdf/1802.08014.pdf.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.

[3] 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.

[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] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *PVLDB*, 5(11):1136–1147, 2012.

[7] 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.

[8] 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.

[9] J. Dai, B. Yang, C. Guo, and Z. Ding. Personalized route recommendation using big trajectory data. In *ICDE*, pages 543–554, 2015.

[10] J. Dai, B. Yang, C. Guo, C. S. Jensen, and J. Hu. Path cost distribution estimation using trajectory data. *PVLDB*, 10(3), 2017.

[11] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *Proc. SEA*, pages 18–29, 2013.

[12] 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.

[13] Z. Ding, B. Yang, R. H. Güting, and Y. Li. Network-matched trajectory-based moving-object database: Models and applications. *IEEE Trans. Intelligent Transportation Systems*, 16(4):1918–1928, 2015.

[14] 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.

[15] C. Guo, C. S. Jensen, and B. Yang. Towards total traffic awareness. *SIGMOD Record*, 43(3):18–23, 2014.

[16] 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.

[17] C. Guo, B. Yang, J. Hu, and C. S. Jensen. Learning to route with sparse trajectory sets. In *ICDE*, page 12 pages, 2018.

[18] 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.

[19] 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.

[20] T. Hashem, T. Hashem, M. E. Ali, and L. Kulik. Group trip planning queries in spatial databases. In *SSTD*, pages 259–276, 2013.

[21] 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.

[22] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng. On trip planning queries in spatial databases. In *SSTD*, pages 273–290, 2005.

[23] J. Li, Y. D. Yang, and N. Mamoulis. Optimal route queries with arbitrary order constraints. *TKDE*, 25(5):1097–1110, 2013.

[24] H. Liu, C. Jin, B. Yang, and A. Zhou. Finding top-k shortest paths with diversity. *TKDE*, 30(3):488–502, 2018.

[25] M. N. Rice and V. J. Tsotras. Engineering generalized shortest path queries. In *ICDE*, pages 949–960, 2013.

[26] M. N. Rice and V. J. Tsotras. Parameterized algorithms for generalized traveling salesman problems in road networks. In *SIGSPATIAL*, pages 114–123, 2013.

[27] 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.

[28] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB J.*, 17(4):765–787, 2008.

[29] M. Sharifzadeh and C. Shahabi. Processing optimal sequenced route queries using voronoi diagrams. *GeoInformatica*, 12(4):411–433, 2008.

[30] B. Yang, J. Dai, C. Guo, and C. S. Jensen. Pace: A PAth-CEntric paradigm for stochastic path finding. *VLDB Journal*, online first, 2017.

[31] 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.

[32] B. Yang, C. Guo, Y. Ma, and C. S. Jensen. Toward personalized, context-aware routing. *VLDB Journal*, 24(2):297–318, 2015.

[33] B. Yang, Q. Ma, W. Qian, and A. Zhou. TRUSTER: trajectory data processing on clusters. In *DASFAA*, pages 768–771, 2009.