The VLDB Journal
DOI 10.1007/s00778-006-0038-6

REGULAR PAPER

The optimal sequenced route query

Mehdi Sharifzadeh · Mohammad Kolahdouzan · Cyrus Shahabi

Received: 8 October 2005 / Revised: 21 April 2006 / Accepted: 18 May 2006
© Springer-Verlag 2006

Abstract Real-world road-planning applications often result in the formulation of new variations of the nearest neighbor (NN) problem requiring new solutions. In this paper, we study an unexplored form of NN queries named optimal sequenced route (OSR) query in both vector and metric spaces. OSR strives to find a route of minimum length starting from a given source location and passing through a number of *typed* locations in a particular order imposed on the types of the locations. We first transform the OSR problem into a shortest path problem on a large planar graph. We show that a classic shortest path algorithm such as Dijkstra's is impractical for most real-world scenarios. Therefore, we propose LORD, a light threshold-based iterative algorithm, which utilizes various thresholds to prune the locations that cannot belong to the optimal route. Then we propose R-LORD, an extension of LORD which uses R-tree to examine the threshold values more efficiently. Finally, for applications that cannot tolerate the Euclidean distance as estimation and require exact distance measures in metric spaces (e.g., road networks) we propose PNE that progressively issues NN queries on different point types to construct the optimal route for the OSR query. Our extensive experiments on both real-world and synthetic datasets verify that our algorithms significantly outperform a disk-based variation of the Dijkstra approach in terms of processing time (up to two orders of magnitude) and required workspace (up to 90% reduction on average).

M. Sharifzadeh (✉) · M. Kolahdouzan · C. Shahabi
Computer Science Department,
University of Southern California,
Los Angeles, CA 90089-0781, USA
e-mail: sharifza@usc.edu

M. Kolahdouzan
e-mail: mohammad-ysm@yahoo.com

C. Shahabi
e-mail: shahabi@usc.edu

Keywords Spatial databases · Nearest neighbor search · Trip planning queries

1 Introduction

The objective of the nearest neighbor query is to find the object(s) with the shortest distance(s) to a given query point. Although this type of query is useful, more often what the user is really after is to plan a trip to *several* (and possibly different types of) locations in some *sequence*, and is interested in finding the optimal route that minimizes her total traveling distance (or time). This type of query is important to both commercial applications such as in-car navigation systems or for online map services as well as non-commercial applications such as in crisis management, emergency response and defense/intelligence systems. Similar types of "planning queries" have also been studied by the database community in other application domains such as in air traffic flow and supply chain management [5,10,14]. In this paper, we formally introduce and address this specific type of trip planning query in the context of spatial database systems.

1.1 Motivation

Suppose we are planning a Saturday trip around the town as follows: first we intend to visit a shopping center in the afternoon to check the season's new arrivals,

then we plan to dine in an Italian restaurant in early evening, and finally, we would like to watch a specific movie at late night. Naturally, we intend to drive the minimum overall distance to these destinations. That is, we need to find the locations of the shopping center s_i, the Italian restaurant r_j, and the theater t_k that shows our movie, where traveling between these locations in the given order would result in the shortest travel distance (or time). Note that in this example, a time constraint enforces the order in which these destinations should be visited; we usually do not have dinner in the afternoon, or go for shopping at late night. This type of query is also essential in other application domains such as *crisis management*, *air traffic flow management*, *supply chain management*, and *video surveillance*. In crisis management, suppose that an ambulance needs to repeatedly visit one of the several attacked points a_i and hospitals h_j, respectively. The ambulance should visit as many of the attacked points as possible in the shortest time. The constraint that enforces the order in this example is that there is no reason for the ambulance to go to a hospital if it has not yet picked up an injured person. Note that in this example, although there are only two different types of points (i.e., attacked points and hospitals), the size of the sequence can become arbitrary large [e.g., $(a_1, h_1, a_2, h_2, \ldots, a_i, h_i)$]. With the video surveillance application, consider searching for a sequence of relevant images of different scene types to discover an event. For instance, among time-stamped images showing individuals approaching a building, waiting in the lobby or leaving the building, we look for the minimal three images with approach/wait/leave sequence. Here, the distance could be defined according to high-dimensional feature space of images including their time-stamps.

We call this type of queries where the order of points to be visited is given and fixed, the optimal sequenced route queries or *OSR* for short. Using Fig. 1, we show that the OSR query cannot be optimally answered by simply performing a series of independent nearest neighbor searches from different locations. We use the first example described above as our running example throughout the paper. The figure shows a network of equally sized connected square cells, three different types of *point sets* shown by white, black and gray circles representing shopping centers, Italian restaurants, and theaters, respectively, and a starting point p (shown by $\triangle$).

A greedy approach to solve OSR is to first locate the closest shopping center to p, s_2, then find the closest restaurant to s_2, r_2, and finally find the closest theater to r_2, t_2. Assuming the length of each edge of a cell is 1 unit, the total length of the route found by this greedy approach,

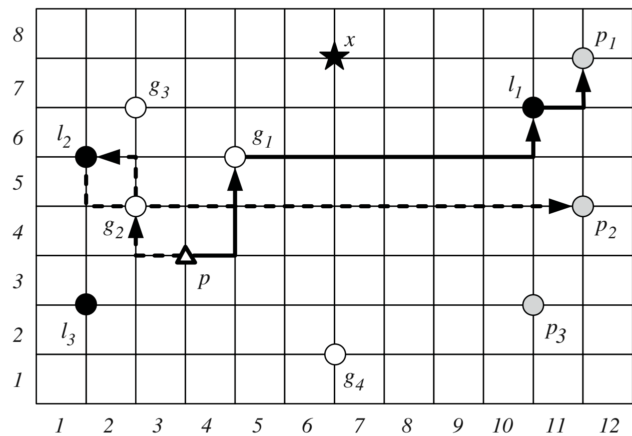

Fig. 1 A network with three different types of point sets

(p, s_2, r_2, t_2), shown by dotted lines in the figure, is 15 units. However, the route (p, s_1, r_1, t_1) (shown with solid lines in the figure) with the length of 12 units is the optimum answer to our query. Note that s_1 is *not* the closest shopping center to p and r_1 is actually the *farthest* restaurant to s_1. Hence, the optimum route for an OSR query can be significantly different from the one found by the greedy approach.

1.2 Uniqueness

Even though different variations of the nearest neighbor query have been extensively studied by the database community, to the best of our knowledge, no other study investigates the optimal sequenced route (OSR) query. This problem is closely related to the traveling salesman problem (TSP). TSP looks for the minimum cost round-trip path from a point passing through a given set of points. As a classic problem in graph theory, TSP is the search for the minimum weight *Hamiltonian cycle* in a weighted graph. With TSP, all the points are of the same type and they must all participate in the route in a given sequence. In contrast, OSR enforces a sequence of points each of which is of a specific type (i.e., from a different point set).

The most similar TSP-related problem to OSR is sequential ordering problem (SOP) in which a Hamiltonian path with a precedence constraint on the nodes is required. Similar to all the TSP variations, the solution path to SOP must still pass through *all* the given points. Conversely, the main challenge with OSR is to efficiently select a sequence of points, where each of which can be any member of a given point set. The commercial online Yellow Pages such as those of Yahoo! and MapQuest can only search for the k-nearest neighbors in *one* specific category (or point set) to a given query

location and cannot find the optimal sequenced route from the query to a *group* of point sets.

Recently and in parallel to our work, the spatial database community has paid attention to queries similar to our OSR query [5,10,14]. Li et al. [10] propose heuristic approximate solutions to trip planning queries (TPQ) which relaxes OSR queries by eliminating the sequence constraint. The *k-stop* problem studied by Terrovitis et al. [14] is also a specialization of OSR for which they provide a solution only in vector spaces. Hadjieleftheriou et al. [5] study the problem of identifying the temporal order patterns within trajectories stored in a moving object database. Our study is among the first attempts to investigate the complex route queries in spatial databases.

1.3 Contributions

In this paper, we introduce and formally define the problem of OSR query in spatial databases. Our general definition of OSR query also covers a variation where the query must end in a specific given point (e.g., the user's starting point). We propose alternative solutions to the OSR queries for both vector and metric spaces.

We start by studying the brute-force search solution to the OSR queries in vector/metric spaces. We show that the exponential size of the search space renders this solution impractical. Hence, for vector spaces, we propose a naive solution that first generates a weighted directed graph from the input point sets, and then uses Dijkstra's algorithm to find the distances from a starting point to all possible end points on the generated graph. This solution becomes impractical when the generated graph is large, which is the case for most of the real-world problems. Therefore, we propose a second solution, LORD, which utilizes some threshold values to filter out the points that cannot possibly be on the optimal route, and then builds the optimal route in reverse sequence (i.e., from ending to the starting point). We then propose R-LORD, which improves LORD by reformulating its thresholds into a single range query and subsequently performing the range query utilizing an R-tree index structure. Finally, we propose PNE to solve OSR in metric spaces (e.g., road networks). PNE is based on progressively finding the nearest neighbors to different point sets in order to construct the optimal route from the starting to the ending point. We also show how LORD, R-LORD, and PNE can be adopted to address variations of OSR (e.g., when the first *k* optimal routes are needed).

We analytically prove the correctness of all of our algorithms. We also theoretically calculate their time, space, and I/O complexities. Finally, through extensive experiments with both real-world and synthetic datasets, we show that LORD, R-LORD, and PNE can efficiently answer OSR queries; R-LORD and PNE scale to large datasets; and R-LORD performs independently from the distribution and density of the data.

The remainder of this paper is organized as follows. We first formally define the problem of OSR queries and the terms we use throughout the paper in Sect. 2. In Sect. 3, we discuss our alternative solutions for OSR queries in vector and metric spaces. We theoretically explore the average case complexity of these solutions in terms of time, memory, and I/O in Sect. 4. We address two variations of OSR queries in Sect. 5. The performance evaluation of our proposed algorithms is presented in Sect. 6. The related work is discussed in Sect. 7. Finally, we conclude the paper and discuss our future work in Sect. 8.

2 Formal problem definition

In this section, we describe the terms and notations that we use throughout the paper, formally define the OSR query, and discuss the unique properties of OSR that we utilize in our solutions. Table 1 summarizes our set of notations.

2.1 Problem definition

Let $U_1, U_2, \ldots, U_n$ be n sets, each containing points in a d-dimensional space $\mathbb{R}^d$, and $D(\cdot,\cdot)$ be a distance metric defined in $\mathbb{R}^d$ where $D(\cdot,\cdot)$ obeys the triangular inequality. To illustrate, in the example of Fig. 1, U_1, U_2, and U_3 are the sets of black, white, and gray points, representing

Table 1 Summary of notations

Symbol	Meaning
U_i	A *point set* in $\mathbb{R}^d$
$\|U_i\|$	Cardinality of the set U_i
n	Number of point sets U_i
$D(\cdot,\cdot)$	Distance function in $\mathbb{R}^d$
M	A sequence, $= (M_1, \ldots, M_m)$
$\|M\|$	m, Size of sequence M = number of items in M
M_i	i-th item of M
R	Route $(P_1, P_2, \ldots, P_r)$, where P_i is a point
$\|R\|$	r, Number of points in R
P_i	i-th point in R
$L(R)$	Length of R
$p \oplus R$	Route $R_p = (p, P_1, \ldots, P_r)$ where $R = (P_1, \ldots, P_r)$
$L(p,R)$	Length of the route $p \oplus R$
$Pfx(M,n)$	The sequence $(M_1, \ldots, M_n)$ where $1 \leq n \leq \|M\|$
$Pfx(R,n)$	The route $(P_1, \ldots, P_n)$ where $1 \leq n \leq \|R\|$

restaurants, shopping centers and theaters, respectively. We first define the following four terms.

Definition 1 *Given n, the number of point sets U_i, we say the m-tuple $M = (M_1, M_2, \ldots, M_m)$ is a* sequence *if and only if $1 \leq M_i \leq n$ for $1 \leq i \leq m$. That is, given the point sets U_i, a user's OSR query is valid only if she asks for existing location types. For the example of Fig. 1 where $n = 3$, $(2, 1, 2)$ is a sequence (specifying a shopping center, a restaurant, and a shopping center), while $(3, 4, 1)$ is not a sequence because 4 is not referring to an existing point set. We use m and $|M|$ to denote the size of the sequence M.*

Definition 2 *We say $R = (P_1, P_2, \ldots, P_r)$ is a* route *if and only if $P_i \in \mathbb{R}^d$ for each $1 \leq i \leq r$. We use $p \oplus R = (p, P_1, \ldots, P_r)$ to denote a new route that starts from point p and goes sequentially through P_1 to P_r. The route $p \oplus R$ is the result of adding p to the head of route R.*

Definition 3 *We define the* length *of a route $R = (P_1, P_2, \ldots, P_r)$ as*

$$L(R) = \sum_{i=1}^{r-1} D(P_i, P_{i+1}) \tag{1}$$

For example, the length of the route (s_2, r_2, s_3) in Fig. 1 is 4 units where D is the Manhattan distance. Note that $L(R) = 0$ for $r = 1$.

Definition 4 *Let $M = (M_1, M_2, \ldots, M_m)$ be a sequence. We refer to the route $R = (P_1, P_2, \ldots, P_m)$ as a* sequenced route *that follows sequence M if and only if $P_i \in U_{M_i}$ where $1 \leq i \leq m$. In Fig. 1, (s_2, r_2, s_3) is a sequenced route that follows $(2, 1, 2)$ which means that the route passes only through a white, then a black and finally a white point.*

We now formally define the OSR query.

Definition 5 *Assume that we are given a sequence $M = (M_1, M_2, \ldots, M_m)$. For a given* starting *point p in $\mathbb{R}^d$ and the sequence M, the OSR query, $Q(p, M)$, is defined as finding a sequenced route $R = (P_1, \ldots, P_m)$ that follows M where the value of the following function L is minimum over all the sequenced routes that follow M:*

$$L(p, R) = D(p, P_1) + L(R) \tag{2}$$

Note that $L(p, R)$ is in fact the length of route $R_p = p \oplus R$. Throughout the paper, we use $Q(p, M) = (P_1, P_2, \ldots, P_m)$ to denote the *optimal SR*, the answer to the OSR query Q. Without loss of generality, we assume that this optimal route is unique for given p and M.[1] For the example in Sect. 1.1 where $(U_1, U_2, U_3) = (black, white, gray)$, $M = (2, 1, 3)$, and D is the Manhattan distance, the answer to the OSR query is $Q(p, M) = (s_1, r_1, t_1)$. We use *candidate* SR to refer to all sequenced routes that follow sequence M. The definition of one of these routes whose length is used as a threshold value in our algorithm follows.

Definition 6 *Given a starting point p, a sequence $M = (M_1, \ldots, M_m)$, and point sets $\{U_1, \ldots, U_n\}$, we refer to $R_g(p, M) = (P_1, \ldots, P_m)$ as the* greedy sequenced route *that follows M from point p if and only if it satisfies the followings*:

1. *P_1 is the closest point to p in U_{M_1}, and*
2. *For $1 \leq i < m$, P_{i+1} is the closest point to P_i in $U_{M_{i+1}}$.*

Without loss of generality, we assume that the closest points of p in U_{M_1} and P_i in $U_{M_{i+1}}$ are unique. Therefore, $R_g(p, M)$ is unique for a given point p, a sequence M, and the sets U_i. Moreover, by definition, the optimal sequenced route R is never longer than the greedy sequenced route for the given sequence M, i.e., $L(p, R) \leq L(p, R_g(p, M))$. As this holds for all *candidate routes that follow M, our proposed algorithms are correct in the general case where the closest points are not unique.*

2.2 Properties

Before describing our algorithms for OSR queries, we present the following three properties which are exploited by our algorithms.

Property 1 For a route $R = (P_1, \ldots, P_i, P_{i+1}, \ldots, P_r)$ and a given point p, we have

$$L(p \oplus R) \geq D(p, P_i) + L((P_i, \ldots, P_r)) \tag{3}$$

Proof The triangular inequality implies that $D(p, P_1) + \sum_{j=1}^{i-1} D(P_j, P_{j+1}) \geq D(p, P_i)$. Adding $\sum_{j=i}^{r-1} D(P_j, P_{j+1}) = L((P_i, \ldots, P_r))$ to both sides of the inequality and considering the definition of the function $L()$ in Eq. 2, yields Eq. 3. □

As we will show in Sect. 3.2.1, we utilize property 1 to narrow down the candidate sequenced routes for $Q(p, M)$ by filtering out the points whose distance to p is greater than a threshold, and hence cannot possibly be on the optimal route. Note that this property is applicable to *all* routes in the space.

[1] Notice that similar assumptions made throughout the paper are intended for simplifying our algorithms and do not enforce any requirement.

The answer to the OSR query $Q(p, M)$ demonstrates the following two unique properties. We utilize these properties to improve the exhaustive search among all potential routes of a given sequence.

Property 2 If $Q(p, M) = R = (P_1, \ldots, P_{m-1}, P_m)$, then P_m is the closest point to P_{m-1} in U_{M_m}.

Proof The proof of this property is by contradiction. Assume that the closest point to P_{m-1} in U_{M_m} is $p_x \neq P_m$. Therefore, we have $D(P_{m-1}, p_x) < D(P_{m-1}, P_m)$ and hence $L(p, (P_1, \ldots, P_{m-1}, p_x)) < L(p, (P_1, \ldots, P_{m-1}, P_m))$. This contradicts our initial assumption that R is the answer to $Q(p, M)$. □

Property 2 states that given that $P_1, \ldots, P_{m-1}$ are subsequently on the optimal route, it is only required to find the first nearest neighbor of P_{m-1} to complete the route and subsequent nearest neighbors cannot possibly be on the optimal route and hence, will not be examined. Note that this property does not prove that the greedy route is always optimal. Instead, it implies that *only* the last point of the optimal sequenced route R (i.e., P_m) is the nearest point of its previous point in the route (i.e., P_{m-1}).

Property 3 If $Q(p, M) = (P_1, \ldots, P_i, P_{i+1}, \ldots, P_m)$ for the sequence $M = (M_1, \ldots, M_i, M_{i+1}, \ldots, M_m)$, then for any point P_i and $M' = (M_{i+1}, \ldots, M_m)$, we have $Q(P_i, M') = (P_{i+1}, \ldots, P_m)$.

Proof The proof of this property is by contradiction. Assume that $Q(P_i, M') = R' = (P'_1, \ldots, P'_{m-i})$. Obviously $(P_{i+1}, \ldots, P_m)$ follows sequence M', therefore we have $L(P_i, R') < L(P_i, (P_{i+1}, \ldots, P_m))$. We add $L(p, (P_1, \ldots, P_i))$ to the both sides of this inequality to get:

$$L(p, (P_1, \ldots, P_i, P'_1, \ldots, P'_{m-i})) < L(p, (P_1, \ldots, P_m))$$

The above inequality shows that the answer to $Q(p, M)$ must be $(P_1, \ldots, P_i, P'_1, \ldots, P'_{m-i})$ which clearly follows sequence M. This contradicts our assumption that $Q(p, M) = R$. □

3 OSR solutions

In this section, we first study the brute-force solution to OSR queries and describe its issues. Subsequently, we propose alternative solutions for OSR queries in vector and metric spaces. We start by discussing a naive solution based on the Dijkstra's shortest path algorithm. We then propose LORD, an approach that employs some threshold values to efficiently prune non-candidate routes. Next we discuss R-LORD, that is an optimization of LORD by utilizing an R-tree index structure. Finally, we discuss PNE, a solution that progressively performs nearest neighbor queries on different point sets to find the optimal route for metric spaces.

Suppose we have an OSR query with a starting point p, a sequence M, and point sets $\{U_{M_1}, \ldots, U_{M_m}\}$. The brute-force search algorithm for finding the optimal solution to OSR query must compute all sequenced routes that follow M. Considering the definition of OSR query, the number of these routes is $\prod_{i=1}^{m} |U_{M_i}|$. Calculating the length of each of these routes consists of $|M|$ distance computation operations. Consequently, the brute-force search algorithm to find the route with minimum length requires $|M| \prod_{i=1}^{m} |U_i|$ distance computations. The computation and storage costs of this algorithm grows exponentially with the sequence size and polynomially with the cardinality of the database which renders it impractical for answering OSR queries in the context of typically large databases.

3.1 The Dijkstra-based solution

This section studies a different naive approach which slightly improves the brute-force algorithm. We are given an OSR query with a starting point p, a sequence M, and point sets $\{U_{M_1}, \ldots, U_{M_m}\}$. We construct a weighted directed graph G where the set $V = \bigcup_{i=1}^{m} U_{M_i} \cup \{p\}$ are the vertices of G and its edges are generated as follows. The vertex corresponding to p is connected to all the vertices in point set U_{M_1}. Subsequently, each vertex corresponding to a point x in U_{M_i} is connected to all the vertices corresponding to the points in $U_{M_{i+1}}$, where $1 \leq i < m - 1$. Figure 2 illustrates an example of such graph. As shown in the figure, the graph G is a *k-bipartite* graph where $k = m + 1$. The weight assigned to each edge of G is the distance between the two points corresponding to its two end-vertices. This graph is in fact showing all possible candidate sequenced routes (candidate SRs) for the given M and the set of U_{M_i}'s. To be precise, it shows all the routes $R_p = p \oplus R$ where R is a candidate SR. By definition, the optimal route for

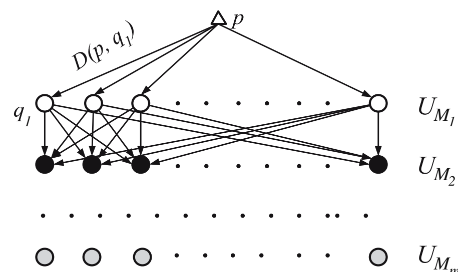

Fig. 2 Weighted directed graph G for sequence M

the given OSR query is the candidate SR, R, for which R_p has the minimum length. Considering graph G, we notice that the OSR problem can be simply considered as finding the shortest paths (i.e., with minimum weight) from p to each of the vertices that correspond to the points in U_{M_m} (i.e., the last level of points in Fig. 2), and then returning the path with the shortest length as the optimal route. This can be achieved by performing the Dijkstra's algorithm on graph G.

There are two drawbacks with this solution. First, the graph G has $|E|=|U_{M_1}|+\sum_{i=1}^{m-1}|U_{M_i}|\times|U_{M_{i+1}}|$ directed edges which is a large number considering the usually large cardinality of the sets U_i. For instance, for a real-world dataset with 40,000 points and $|M| = 3$, G has 124 million edges (see Sect. 6). The time complexity of the Dijkstra's classic algorithm to find the shortest path between two nodes in graph G is $O(|E|\log|V|)$. Hence, the complexity of this naive algorithm is $O(|U_{M_m}|\,|E|\log|V|)$. Second, this huge graph must be built and kept in main memory. Although there exist versions of the Dijkstra's algorithm that are adjusted to use external memory [7], but they result in so much of overhead which makes them hard to employ for OSR queries (see Sect. 7 for the complete discussion). This renders the classic Dijkstra's algorithm to answer OSR queries in real-time impractical.

In order to improve the performance of this naive Dijkstra-based solution, we can issue a range query around the starting point p and only select the points that are closer to p than $L(p, R_g(p,M))$. This is because the length of any route R which includes a point outside this range is greater than that of the greedy route $R_g(p,M)$. Therefore, we build the graph G using only the points within the range instead of all the points. In Sect. 6, we show that even this enhanced version of the Dijkstra's algorithm (EDJ) is not as efficient as our approaches.

3.2 OSR in vector space

In this section, we assume that the distance function $D(\cdot,\cdot)$ is the Euclidean distance between the points in $\mathbb{R}^d$. We provide two solutions for OSR problem in this vector space.

3.2.1 Light optimal route discoverer

This section describes our light optimal route discoverer (LORD) for addressing OSR queries. LORD has the same flavor as Dijkstra's algorithm but as a threshold-based algorithm it functions in the context of the OSR problem considering its unique properties described in Sect. 2.2. The LORD is *light* in terms of memory usage as we will show that LORD's workspace is less than the workspace required when the Dijkstra-based approach is applied to the OSR problem.

Given an OSR query $Q(p,M)$, LORD iteratively builds and maintains a set of partial sequenced routes (partial SR) in the reverse sequence, i.e., from the end points (points in U_{M_m}) toward p. During each iteration i of LORD, points from the point set $U_{M_{(m-i+1)}}$ are added to the head of each of these partial SRs to make them closer to a candidate SR and finally, to the solution (i.e., optimal SR). To make the solution space smaller, LORD only considers those points in $U_{M_{(m-i+1)}}$ that adding them to the partial SRs will not generate routes which are longer than a *variable* threshold value T_v. LORD further examines the partial SRs by calculating their lengths after adding p, and discards the routes whose corresponding length is more than a *constant* threshold value T_c, where T_c is the length of the greedy route.

We now describe LORD in more details using the example shown in Fig. 3. Figure 3a depicts a starting point p and three different sets of points U_1, U_2, and U_3, shown as black (b_i), white (w_i) and gray (g_i) points, respectively. Without loss of generality, we assume that the distance between each two points in the space is their Euclidean distance. Given the starting point p (shown as $\triangle$ in the figure), we want to find the route R with the minimum $L(p,R)$ from a white, to a black and then a gray point. Therefore, the required OSR query is formulated as $Q(p,(2,1,3))$.

Figure 4 shows the pseudo-code of LORD. The algorithm generates a set, S, for partial candidate routes and initializes it to the empty set. The first step in LORD is to issue $m(=3)$ consecutive nearest neighbor queries to find the greedy route that follows $(2,1,3)$ from p. To be specific, the algorithm first finds the closest w_i to p (i.e., w_2), then the closest b_i to w_2 (i.e., b_2), and finally the closest g_i to b_2 (i.e., g_2). Figure 3b renders the greedy route $R_g(p,(2,1,3))$ as (w_2,b_2,g_2). LORD initiates both threshold values T_v and T_c to the length of $p\oplus R_g(p,M)$ (i.e., $L(p,(w_2,b_2,g_2))$). Note that the value of T_c remains the same while the value of T_v reduces after each iteration. Subsequently, LORD discards all the points whose distances to p are more than T_v, i.e., the points that are outside the circle shown in Fig. 3c (i.e., w_1, w_4, and g_1). This is because any route (e.g., R) that contains a point that is outside this circle will lead to $L(p,R) > L(p,R_g(p,M))$ and hence, by definition, cannot be the optimal route. At this point, LORD inserts the gray nodes (i.e., points in U_{M_m}) which are inside the circle in Fig. 3c, in to S, i.e., $S=\{(g_2),(g_3),(g_4),(g_5),(g_6)\}$. Note that at this stage, the length of the partial routes in S is zero.

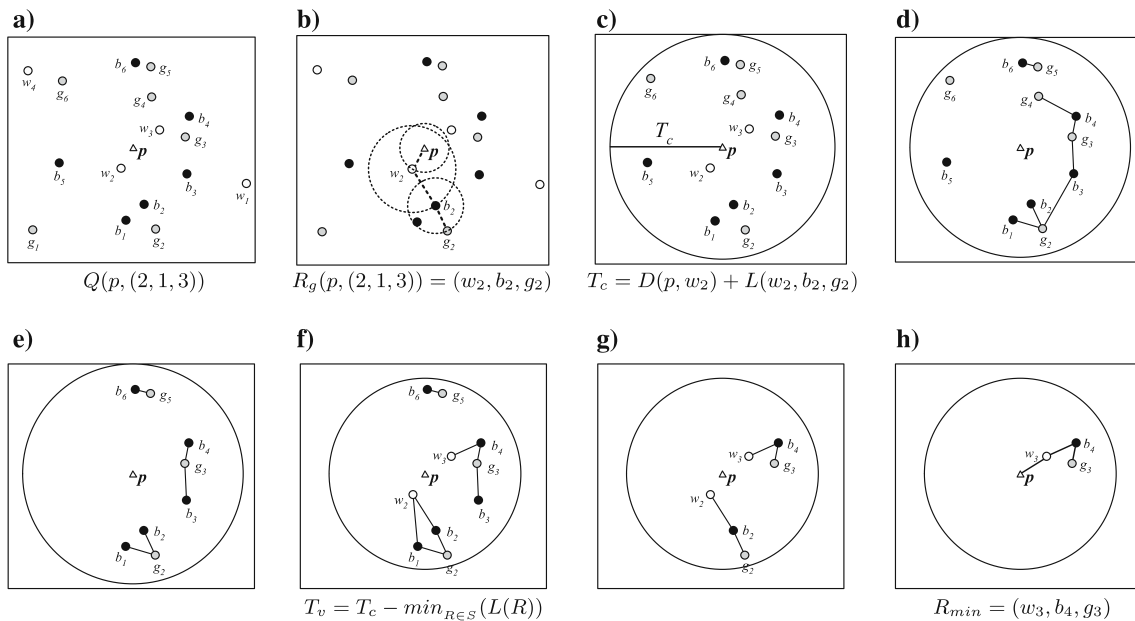

Fig. 3 Iterations of LORD

In the first iteration of LORD, each point $x \in U_{M_{m-1}}$ (i.e., b_i's) is added to the head of each partial SR, $PSR = (P_1) \in S$, if: (a) x is inside the circle T_v, and (b) $D(p,x) + D(x,P_1) + L(PSR) \leq T_c$. The rational behind the second condition is property 1; if the inequality does not hold, then $L(p,(x,P_1,\ldots,P_i))$ will be greater than T_c and hence, $(x,P_1,\ldots,P_i)$ cannot be part of the optimal route. For instance, in Fig. 3d, point b_4 is added to (g_3) and (g_4) resulting in new partial SRs $\{(b_4,g_3),(b_4,g_4)\}$, but cannot be added to (g_2), (g_5) and (g_6). Moreover, between partial SRs that have the same first point (e.g., (b_4,g_3) and (b_4,g_4)), only the one with the shortest length will be kept in S (i.e., property 2). In addition, any $PSR \in S$ that no x can be added to it will be discarded. For example, in Fig. 3d, (g_6) will be discarded because if any b_i is added to it, at least one of the above two conditions will not be met. Hence, at the end of the first step, the set of the partial SRs will become $\{(b_6,g_5),(b_4,g_3),(b_3,g_3),(b_2,g_2),(b_1,g_2)\}$ (Fig. 3e).

At the end of each iteration, the value of variable threshold T_v is decreased as follows. Suppose that $Q(p,M) = (q_1,\ldots,q_i,\ldots,q_m)$ and we are examining iteration $(m-i+1)$ (i.e., the partial SRs in S are in the form of $(p_{i+1},\ldots,p_m)$). The definition of the greedy route implies that

$$L(p,(q_1,\ldots,q_m)) \leq L(p,R_g(p,M)) = T_c$$

and by considering Property 1, we have

$$D(p,q_i)+L((q_{i+1},\ldots,q_m)) < D(p,q_i)+L((q_i,\ldots,q_m)) \leq T_c$$

which can be rewritten as

$$D(p,q_i) \leq T_c - L((q_{i+1},\ldots,q_m)) \qquad (4)$$

Note that inequality 4 must hold for all points q_i that are to be examined at iteration $(m-i+1)$. Hence, by replacing $L((q_{i+1},\ldots,q_m))$ with its minimum value, we obtain the maximum value for $D(p,q_i)$ for any q_i. Therefore, for any point q_i that is examined in iteration $(m-i+1)$, we must have

$$D(p,q_i) \leq T_v = T_c - min_{PSR \in S}(L(PSR))$$

Note that at each iteration, the lengths of the partial SRs in S, and hence the value of $min_{PSR \in S}(L(PSR))$ is increasing. This yields to smaller values for T_v after each iteration. This is also shown in Fig. 3; the radius of the circle in Fig. 3f is smaller than the radius of the circle in Fig. 3c.

The subsequent $(m-2)$ iterations of LORD are performed similarly and the partial routes in S will become complete routes (i.e., candidate SRs that follow M) after the last iteration is completed (Fig. 3g). Finally, LORD examines the distance from p to the first point in each complete route in S (i.e., $\{(w_2,b_2,g_2), (w_3,b_4,g_3)\}$) and selects the one that generates the minimum total distance, i.e., the route with the minimum value for $L()$ function, as the result of $Q(p,(2,1,3))$ (route (w_3,b_4,g_3) in Fig. 3h).

Now that we described the details of our LORD algorithm, we explain why it builds the candidate sequenced routes in reverse sequence (i.e, from the points in U_{M_m}

to the points in U_{M_1}). First, all iterations of LORD use a common circular range around p to select the points to be included in the candidate routes. The radius of this circle is decreasing from one iteration to the next one which incrementally increases the number of points to be pruned. If we start from p (and the points in U_{M_1}) to generate candidate routes, this range either remains the same as its initial size, or varies for *each* candidate route during the iterations. Second, LORD continues to generate the partial route in reverse sequence to be able to employ the properties described in Sect. 2.2 and prune more candidate points. This is not feasible if LORD builds these routes in the original sequence. Finally, as we show in Sect. 3.2.2 adding points in to the middle of the candidate routes (between p and the head of any partial route) enables LORD to utilize the locus of these candidate points as an ellipse. This helps LORD's iterations to prune more points.

```
     Algorithm LORD(point p, sequence M)
01.    set S = {};
02.    T_v = T_c = L(p, R_g(p, M));
03.    for q in U_{M_m} {
04.       if (D(p, q) ≤ T_v)
05.          S = S ∪ {(q)};
       }
06.    for i = m − 1 downto 1 {
07.       set S' = {};
08.       for q in U_{M_i} {
09.          if (D(p, q) ≤ T_v)
10.             S'' = {};
11.             for R = (P_1, ..., P_{m−i}) in S {
12.                if (D(p, q) + D(q, P_1) + L(R) ≤ T_c)
13.                   S'' = S'' ∪ {q ⊕ R};
                }
14.             S' = S' ∪ {argmin_{R''∈S''}(L(R''))};
          }
15.       S = S';
16.       T_v = T_c − min_{R∈S}(L(R));
       }
17.    route R_min = argmin_{R∈S}(L(p, R));
18.    return R_min;
```

Fig. 4 Pseudo-code of the LORD algorithm

3.2.2 R-LORD: R-tree-based LORD

We described LORD in Sect. 3.2.1 without any assumption on the structure of the points in each U_i. We now discuss the situation that the points in U_i's are stored in an R-tree index structure. We utilize the features of the index structure to develop an R-tree-friendly version of LORD. The core idea behind this solution is to use the points' neighborhood information implicitly stored in R-tree MBR's to more efficiently prune the candidate points at each iteration of LORD. Towards this goal, we transform the LORD's point selection criterium to the range queries applicable on an R-tree. Then, we show that the point selection can be performed using a *single* range query. Finally, we describe our algorithm which uses this range to find the solution for an OSR query utilizing an R-tree.

3.2.2.1 Point selection criterium in LORD As we discussed in Sect. 3.2.1, at each iteration i, LORD prunes the points in U_{M_i} in two steps. First, it ignores any point of the set U_{M_i} that is farther than the value of the variable threshold T_v from the starting point p. This is a simple range query **Q1** given the range *Range*(**Q1**) as a circle with a known radius T_v centered at p. Second, any point x resulting from query *Range*(**Q1**) is checked against all partial SRs $PSR \in S$. If for each $PSR = (P_1, \ldots, P_{|PSR|}) \in S$, the value of $D(p,x) + D(x,P_1) + L(PSR)$ is greater than the constant threshold T_c (i.e., the length of the greedy route), then point x is not added to the beginning of that PSR. Otherwise, a new partial SR, $(x, P_1, \ldots, P_{|PSR|})$, is generated. This clearly shows that the second query **Q2** uses a more complicated range to prune the results of **Q1**.

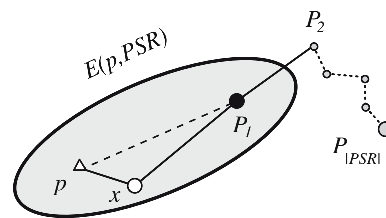

Fig. 5 The locus of the points x for LORD

To identify *Range*(**Q2**), we first find the locus of the points x which can possibly be added to a $PSR = (P_1, \ldots, P_{|PSR|}) \in S$. For such a point x, we must have $D(x,p) + D(x,P_1) \le T_c - L(PSR)$ (line 12 in Fig. 4). As $L(PSR)$ and T_c are constant values for a given PSR and query $Q(p,M)$, the sum of x's distances from two fixed points p and P_1 cannot be larger than a constant. Hence, x must be on or inside an ellipse defined by the foci p and P_1 and the constant $T_c - L(PSR)$. Figure 5 shows the locus of the points x for a given route PSR as inside and on an ellipse $E(p, PSR)$.

Query **Q2** is defined in terms of the set of partial SRs stored in S during the current iteration. For each PSR, we showed that LORD appends points inside ellipse $E(p, PSR)$ to the head of the PSR in order to build a new partial candidate route. All such ellipses, each corresponding to a partial SR in S, are intersecting as they all share the common focus point p. The union of these ellipses contains all the points x (of the appropriate set), where for each, there is exactly one route starting with x built at the end of the current iteration. In other words, this union should be the range used in query **Q2**. Figure 6 illustrates an example for the current set S during an iteration of LORD. The set includes three partial SRs of the same length each starting with a black point.

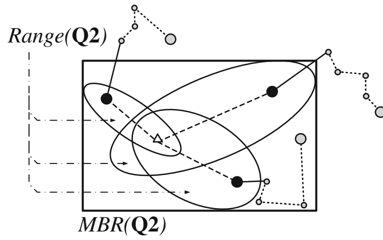

Fig. 6 Range query **Q2** and its MBR for partial routes in LORD

```
Algorithm RangeQuery(point p, range R)
01. list L = empty; set R = {};
02. insert R-tree root into L;
03. while L is not empty {
04.     remove first node N from L;
05.     if N is a data point q and q ∈ R then
06.         R = R ∪ {q};
07.     else // N is an intermediate node
08.         for each child node N' of N {
09.             if MBR(N)' intersects with R then
10.                 add N' to L;
            }
    }
11. return R;
```

Fig. 7 General range query using an R-tree

The sequence M of the query $Q(p,M)$ dictates the type of the point which must be added to the head of each partial SR. Any point outside the union of these three ellipses is ignored by LORD.

Up to this point, we have identified the range of the two main queries **Q1** and **Q2** used in LORD. In the following, we show that any ellipse for the range **Q2** is entirely inside the circle for range **Q1** and hence, the range of **Q2** is completely inside that of **Q1**.

Lemma 1 *During each iteration of LORD for $Q(p,M)$, given a partial sequenced route $PSR \in S$, any point x inside or on the ellipse $E(p,PSR)$ has a distance less than current value of the variable threshold T_v from point p (i.e., $D(x,p) < T_v$).*

Proof As point x is inside or on ellipse $E(p,PSR)$ corresponding to the route PSR, we have

$$\begin{aligned} D(x,p) + D(x,P_1) &\leq T_c - L(PSR) \\ &\leq T_c - min_{PSR\in S}(L(PSR)) \end{aligned} \tag{5}$$

The right side of the above inequality has the same value as that of the current value of T_v. It directly yields that $D(x,p) \leq T_v - D(x,P_1)$ and subsequently, we have $D(x,p) < T_v$. □

Lemma 1 shows that any ellipse $E(p,PSR)$ is completely inside the circular range of **Q1**. Now, as *Range* (**Q2**) is the union of all ellipses $E(p,PSR)$ corresponding to all the partial SRs in S, it can be concluded that it is entirely inside *Range*(**Q1**).

Note that at each iteration, LORD builds a new route using only the points in the intersection of *Range*(**Q1**) and *Range*(**Q2**). Given Lemma 1, this intersection is the same as *Range*(**Q2**). Hence, the algorithm must only consider the points which are within the range of **Q2** from p, to be added to the partial SRs in S.

3.2.2.2 R-tree friendly LORD Recall that our goal is to transform the threshold values utilized by LORD to a single range query that can be performed efficiently using an R-tree index structure. Then, to retrieve the points in the specific range, we need to traverse the R-tree from its root down to the leaves, visit the intermediate nodes, extract the child nodes of only those that intersect with the given range and report the points that are inside that range. Figure 7 shows the pseudo-code of a general range query processing algorithm on R-tree that we employ.

In Sect. 3.2.2, we showed that the two range queries **Q1** and **Q2** employed by LORD can be reduced to only one query as **Q2** is entirely inside **Q1**. However, as Fig. 6 illustrates, the range specified by **Q2** (union of the ellipses) is a complex parameterized curved shape. This range cannot be efficiently handled by the R-tree range query algorithm of Fig. 7 as computing its intersection with a rectangular node requires a complicated expensive operation. To make this range simpler (i.e., rectangular), we employ its minimum bounding box (*MBR*(**Q2**)) as shown in Fig. 6. However, *MBR*(**Q2**) is no longer inside the range of **Q1**. Therefore, our R-tree version of LORD must use the intersection of *MBR*(**Q2**) and *Range*(**Q1**) in its single range query to examine the points in U_{M_i}'s.

Now that we have identified the range queries used to select the points in LORD and studied how they can be evaluated together using an R-tree, we propose R-LORD, the R-tree version of LORD. Figure 8 shows the pseudo-code of R-LORD. The only difference between R-LORD and LORD is that R-LORD incorporates a single R-tree implementation of the two range queries of LORD in its iterations. First, it initializes *Range*(**Q1**) to a circle with the radius $T_v = L(p, R_g(p,M))$ centered at p. The range *MBR*(**Q2**) is also initialized to the minimum bounding box of the entire space of points.

During each iteration, R-LORD traverses the entire R-tree starting from the root down to the data points. It should extract only those nodes that are inside the intersection of *Range*(**Q1**) and *MBR*(**Q2**) and prune the others. Therefore, for each visited node N, we first check whether N (i.e., $MBR(N)$) intersects with *Range*(**Q1**). As this range is a circle centered at p, we utilize the

```
Algorithm R-LORD(point p, sequence M)
01. set S = {};
02. box MBR(Q2) = entire space;
03. T_c = T_v = L(p, R_g(p, M));
04. for i = m downto 1 {
05.     set S' = {};
06.     box B = empty;
07.     insert R-tree root into list L;
08.     while L is not empty {
09.         remove first node N from L;
10.         if ((N is a data point q) and
               (q ∈ U_{M_i}) and (D(p, q) ≤ T_v) and
               (q inside MBR(Q2))) then
11.             if (i = m) then // first iteration
12.                 S = S ∪ {(q)};
13.             else
14.                 set S'' = {};
15.                 for R = (P_1, ..., P_{m-i}) in S {
16.                     if (D(p, q) + D(q, P_1) + L(R) ≤ T_c)
17.                         S'' = S'' ∪ {q ⊕ R};
                    }
18.                 route R_min = argmin_{R''∈S''}(L(R''));
19.                 S' = S' ∪ {R_min};
20.                 B = MBR(B union MBR(E(p, R_min));
21.         else // N is an intermediate node
22.             for each child node N' of N {
23.                 if ((mindist(N', p) ≤ T_v) and
                       (N' intersects with MBR(Q2))) then
24.                     add N' to L;
                }
        }
25.     S = S';
26.     MBR(Q2) = B;
27.     T_v = T_c − min_{R∈S}(L(R));
    }
28. route R_min = argmin_{R∈S}(L(p, R));
29. return R_min;
```

Fig. 8 Pseudo-code of the R-LORD algorithm

function *mindist*(N,p) which gives a lower bound on the smallest distance between the point p and any point in the subtree of node N. Any R-tree node N with *mindist* (N,p) greater than radius of *Range*(**Q1**) (i.e., threshold T_v) does not intersect with *Range*(**Q1**) and should be pruned.

For any node N intersecting with *Range*(**Q1**), we check whether N intersects with *MBR*(**Q2**) too. This is an easy intersection check against two rectangles. Now, if an intermediate or leaf node passes both checks, its children are added to the list of nodes/points to be visited. Otherwise, the node is discarded which prunes all the nodes/points in its corresponding subtree. We treat the visited data points similarly. The only difference is that any point that passes the above checks (i.e., is in both ranges) creates a new *PSR* (R_{min}). The minimum bounding box of its corresponding ellipse $E(p, R_{min})$ also updates *MBR*(**Q2**) for the next iteration (i.e., B in Fig. 8). At the end of each iteration, we update *MBR*(**Q2**) to B, the minimum bounding box of the current partial sequenced routes. This is simply the union of the MBRs of all ellipses corresponding to the newly built partial routes.

3.2.3 Correctness

We show that LORD and R-LORD both correctly answer OSR queries. As R-LORD is only a specialization of LORD for R-tree, we only prove the correctness of LORD which also validates R-LORD.

Lemma 2 *For a given starting point p and sequence M, LORD accurately finds the optimal sequence route* $Q(p,M)$.

Proof We show that LORD examines all the routes eligible to be the optimal route. First, for the optimal route $L(p, Q(p,M))$ is not greater than $L(p, R_g(p,M))$. The initialization step of LORD guarantees that LORD starts by examining only the routes R that follow M and their $L(p,R)$ are not greater than that of the greedy route (line 3 in Fig. 4). Second, during its iterations, LORD uses Property 1 to prune the routes R with $L(p,R)$ greater than $L(p, R_g(p,M))$ (line 12). Third, LORD utilizes the unique property of the optimal sequenced route Property 3 to examine only those routes that exhibit this property (line 14). Finally, LORD returns the examined route R with minimum $L(p,R)$ which is the optimal route by definition. □

3.3 OSR in metric space

The previous proposed solutions for OSR queries (discussed in Sects. 3.2.1 and 3.2.2), although efficient in vector spaces, are impractical or inefficient for a sequence M in a *metric* space (road networks). Even though both EDJ and LORD can be applied to both vector and metric spaces, their extensive usage of the $D(\cdot,\cdot)$ function renders them inefficient for metric spaces where the distance metric is usually a computationally complex function. When applied on road networks, both EDJ and LORD require significant number of distance computations, each of them corresponds to finding a shortest path in the road network. This makes EDJ and LORD infeasible for road networks. Likewise, R-LORD can only be applied to vector spaces since it is based on utilizing R-tree index structure.

Road networks (or general spatial networks) can be modeled as weighted graphs where the intersections are represented by nodes of the graph and roads are represented by the edges connecting the nodes. The weights can be the distances of the nodes or they can be the time it takes to travel between the nodes (representing shortest times). The distance between any two points on the nodes or edges of the graph is the length of the shortest path connecting them via the graph edges. Therefore, the triangle inequality obviously holds for this distance function. Although the triangle inequality is the only

```
Function NN(point p, dataset U_i)
    returns the closest point to p in U_i;

Function NextNN(point p, point n, dataset U_i)
    returns q ≠ n, the next closest point to p in U_i s.t.
            D(q,p) ≥ D(n,p);

Algorithm PNE(point p, sequence M)
01. MinHeap H = {};
02. q = NN(p, U_{M_1});
03. add ((q), D(p,q)) to H;
04. do {
05.     remove route PSR from H; // shortest route
06.     k = |PSR|;
07.     if (k = m) then
08.         return PSR;
09.     else
10.         P_{k+1} = NN(P_k, U_{M_{k+1}});
11.         PSR' = (P_1, ..., P_k, P_{k+1});
12.         add (PSR', L(p, PSR')) to H;
13.         if (k > 1) then
14.             P'_k = NextNN(P_{k-1}, P_k, U_{M_k});
15.             PSR' = (P_1, ..., P_{k-1}, P'_k);
16.         else
17.             P'_k = NextNN(p, P_1, U_{M_1});
18.             PSR' = (P'_k);
19.         add (PSR', L(p, PSR')) to H;
20. } while |PSR| < m;
```

Fig. 9 Pseudo-code of the PNE algorithm for a metric space

requirement of the model used by our proposed algorithm, throughout the paper we assume that the graph model of the road network is *undirected* so the distance function is symmetric.

In this section, we describe our proposed algorithm, progressive neighbor exploration (PNE), to address OSR queries in metric spaces for arbitrary values of M. PNE uses efficient fast nearest neighbor algorithms such as INE [11] or VN^3 [9] utilized for road network databases to replace the extensive use of distance computation operations in LORD. It utilizes the progressiveness of these algorithms to efficiently build candidate sequenced routes and refine them. Similar to EDJ and LORD, PNE addresses OSR in both vector and metric spaces. However, it is suitable for the spaces where the computation of the distance metric is very expensive. Notice that PNE uses the same road network model specified by its underlying nearest neighbor algorithm.

Figure 9 shows the pseudo-code of the PNE algorithm. Unlike LORD, the idea behind PNE is to incrementally create the set of candidate routes for $Q(p,M)$ in the same sequence as M, i.e., from p toward U_{M_m}. This is achieved through an iterative process in which we start by examining the nearest neighbor to p in U_{M_1}, generating partial SR from p to this neighbor, and storing the candidate route in a heap based on its length. At each subsequent iteration of PNE, a partial SR (*e.g.*, $PSR = (P_1, P_2, \ldots, P_{|PSR|})$) from top of the heap is fetched and examined as follows.

1. If $|PSR| = m$, meaning that the number of nodes in the partial SR is equal to the number of items in M and hence PSR is a candidate SR that follows M, the PSR is selected as the optimal route for $Q(p,M)$ since it also has the shortest length.
2. If $|PSR| < m$:
   (a) First the last point in PSR, $P_{|PSR|}$, (which belongs to $U_{M_{|PSR|}}$) is extracted and its next nearest neighbor in $U_{M_{|PSR|+1}}$, $P_{|PSR|+1}$, is found.[2] This will guarantee that (a) the sequence of the points in PSR always follows sequence specified in M, and (b) the points that are closer to $P_{|PSR|}$ and hence may potentially generate smaller routes are examined first. The fetched PSR is then updated to include $P_{|PSR|+1}$ and is put back in to the heap.
   (b) We then find the next nearest neighbor in $U_{M_{|PSR|}}$ to $P_{|PSR|-1}$, $P'_{|PSR|}$, generate a new partial SR, $PSR' = (P_1, P_2, \ldots, P_{|PSR|-1}, P'_{|PSR|})$, and place the new route in to the heap. This is because once the point $P_{|PSR|}$, which we can assume is the k-th nearest point in $U_{M_{|PSR|}}$ to $P_{|PSR|-1}$, is chosen in step (a) above, the $(k+1)$-st nearest point in $U_{M_{|PSR|}}$ to $P_{|PSR|-1}$ (e.g., $P'_{|PSR|}$) is the only next point that may generate a shorter route and hence, must be examined. If $|PSR| = 1$, we find the next nearest point in U_{M_1} to p.

We describe PNE in more details using the example of Sect. 1.1. Recall that our OSR query was to drive toward a shopping center, a restaurant, and then a theater (i.e., $M = (2,1,3)$ and $|M| = m = 3$). Figure 2 depicts the values stored in the heap in each step of the algorithm. In step 1, the first nearest s_i to p, s_2, is found and the first partial SR along with its distance, $(s_2 : 2)$, is generated and placed in to the heap. In step 2, first $(s_2 : 2)$ is fetched from the heap. Since for this route $|PSR| < 3$, the above steps 2(a) and 2(b) are performed. More specifically, first the next nearest r_i to s_2, r_2, is found; the partial SR is updated by adding r_2 to it; and is placed back into the heap. Second, the next nearest s_i to p, s_1, is found and is placed in to the heap. Similarly, this process is repeated until the route on top of the heap follows the sequence M (i.e., (s_1, r_1, t_1) in step 13). Note that we only keep one candidate SR (i.e., route with m points) in the heap. That

[2] Assume the points in U_i can be ranked based on the increasing distance from a point p and ties are broken for the points with the same distance (e.g., we use the direction of points considering p to break the tie). Although we assume that the underlying NN algorithm can track the nearest neighbors of any query point per user's request, one can easily modify an incremental NN algorithm such as VN^3 [9] to support this extension.

Table 2 PNE for the example of Fig. 1

Step	Heap contents (partial candidate route R : $L(p,R)$)
1	$(s_2 : 2)$
2	$(s_1 : 3), (s_2, r_2 : 4)$
3	$(s_2, r_2 : 4), (s_3 : 4), (s_1, r_2 : 6)$
4	$(s_3 : 4), (s_2, r_3 : 5), (s_1, r_2 : 6), (s_2, r_2, t_2 : 15)$
5	$(s_2, r_3 : 5), (s_4 : 5), (s_1, r_2 : 6), (s_3, r_2 : 6)$ $(s_2, r_2, t_2 : 15)$
6	$(s_4 : 5), (s_1, r_2 : 6), (s_3, r_2 : 6), (s_2, r_1 : 12)$ $(s_2, r_3, t_3 : 14)$, ~~$(s_2, r_2, t_2 : 15)$~~
7	$(s_1, r_2 : 6), (s_3, r_2 : 6), (s_4, r_3 : 11), (s_2, r_1 : 12)$ $(s_2, r_3, t_3 : 14)$
8	$(s_3, r_2 : 6), (s_1, r_3 : 9), (s_4, r_3 : 11), (s_2, r_1 : 12)$ $(s_2, r_3, t_3 : 14)$, ~~$(s_1, r_2, t_2 : 17)$~~
9	$(s_1, r_3 : 9), (s_3, r_3 : 9), (s_4, r_3 : 11), (s_2, r_1 : 12)$ $(s_2, r_3, t_3 : 14)$, ~~$(s_3, r_2, t_2 : 17)$~~
10	$(s_3, r_3 : 9), (s_1, r_1 : 10), (s_4, r_3 : 11), (s_2, r_1 : 12)$ $(s_2, r_3, t_3 : 14)$, ~~$(s_1, r_3, t_3 : 18)$~~
11	$(s_1, r_1 : 10), (s_4, r_3 : 11), (s_2, r_1 : 12), (s_3, r_1 : 12)$ $(s_2, r_3, t_3 : 14)$, ~~$(s_3, r_3, t_3 : 18)$~~
12	$(s_4, r_3 : 11), (s_2, r_1 : 12), (s_3, r_1 : 12), (s_1, r_1, t_1 : 12)$ ~~$(s_2, r_3, t_3 : 14)$~~
13	$(s_2, r_1 : 12), (s_3, r_1 : 12), (s_1, r_1, t_1 : 12)$ ~~$(s_4, r_3, t_3 : 20)$~~

is, if during step 2(a) a route with m points is generated, it is only added to the heap if there is no other candidate SR with a shorter length in the heap. Moreover, after a candidate SR is added to the heap, any other SR with longer length will be discarded. For example, in step 6, adding the route (s_2, r_3, t_3) with the length of 14 to the heap will result in discarding the route (s_2, r_2, t_2) with the length of 15 from the heap (crossed out in the figure). However, by keeping k routes in the heap and continuing the algorithm until k routes are fetched from the heap, we can easily address a variation of OSR where k routes with the minimum total distances are requested.

The only requirement for PNE is a nearest neighbor approach that can progressively generate the neighbors (i.e., a distance browsing algorithm [6]). Hence, by employing an approach similar to INE [11] or our VN^3 [9], which are explicitly designed for metric spaces, PNE can address OSR queries in metric spaces. In theory, PNE can work for vector spaces in a similar way; however, it is inefficient for these spaces where distance computation is not expensive. The reason is that PNE explores the candidate routes from the starting point which may result in an exhaustive search. Instead, R-LORD optimizes this search by building the routes in the reverse sequence utilizing the R-tree index structure.

3.3.1 Correctness

We prove that PNE correctly answers OSR queries in metric spaces. Throughout the proof, we use $Pfx(M,n) = (M_1, \ldots, M_n)$ where $1 \le n \le |M|$ to denote the prefix sequence of sequence M with size n. We also use $Pfx(R,n) = (P_1, \ldots, P_n)$ for route $R = (P_1, \ldots, P_r)$ to refer to the prefix route of R with size n. Given a query $Q(p,M)$, we use partial candidate route (PC route) to refer to any route R that follows a prefix sequence of M and for which we have $L(p,R) \le L(p, Q(p,M))$.

PNE starts generating and examining all candidate routes for the given query from p towards the last point in U_{M_m}. While generating route $R = (P_1, \ldots, P_m)$, PNE stops generating R when for one of R's prefix routes $Pfx(R,n)$ we have $L(p, Pfx(R,n)) > L(p, R_{min})$ where R_{min} is the best candidate route built so far. The reason is that any candidate route built by appending more points to this prefix route cannot be the optimal route. The following lemma shows that PNE examines all other routes for which the above inequality does not hold.

Lemma 3 *For a given OSR query $Q(p,M)$, PNE examines all Q's partial candidate routes.*

Proof The proof is by induction on n, the size of the partial candidate routes. First, for $n = 1$ we show that PNE examines all PC routes R that follow the sequence $Pfx(M,1) = (M_1)$. That is, it generates all routes (q) for which $q \in U_{M_1}$ and $D(p,q) \le L(p, Q(p,M))$. The initialization step of PNE (line 3 in Fig. 9) generates the route corresponding to the first closest point to p in U_{M_1}. Whenever PNE removes the route corresponding to the k-th closest point to p in U_{M_1} from the heap H, it generates the route corresponding to p's $k+1$-th closest point and adds it to H (lines 16–18). Notice that there is always an iteration during which the former route is at the top of H as long as it is not longer than any candidate route which is the assumption of the lemma. Therefore, all PC routes of size one that follow $Pfx(M,1)$ are examined by PNE.

Now, assume that PNE examines all PC routes that follow $Pfx(M,n)$. Assume that the PC route $R' = (P_1, \ldots, P_{n+1})$ follows $Pfx(M, n+1)$. We show that PNE examines R'. It is clear that the route $R'' = Pfx(R',n)$ is a PC route that follows $Pfx(M,n)$ and hence is examined by PNE. Assume that P_{n+1} is the k-th closest point to P_n in $U_{M_{n+1}}$. When PNE removes $R'' = (P_1, \ldots, P_n)$ from the heap H, it generates the route $(P_1, \ldots, P_n, P')$ in which P' is the closest point to p in U_{M_1} (lines 9–11). During subsequent iterations, this route is replaced with the routes $(P_1, \ldots, P_n, P_{n+1})$ in which P_{n+1} is the k-th closest point to P_n. Therefore, PNE examines the PC route R'. □

Lemma 3 shows that PNE progressively searches the entire space of candidate sequenced routes, prunes those that cannot be optimal as soon as it identifies them, and

Table 3 Notations used in the complexity analysis

Symbol	Meaning
N	Total number of points in all U_{M_i}'s where $1 \le i \le \|M\|$
δ	Expected distance between any two points
C_x	Expected computation cost of algorithm x
M_x	Expected workspace requirement of algorithm x (i.e., number of partial routes maintained)
$NA(i)$	Expected number of R-tree nodes accessed in the ith iteration of R-LORD
P_{NA_i}	Probability of accessing an R-tree node at level i
$N_{pne}(i)$	The number of partial routes that PNE generates for a sequence of size i
C_{NN}	Expected computation cost of a single first (or next) nearest neighbor query

finally returns the optimal route. This proves the correctness of PNE.

Lemma 4 *For a given starting point p and sequence M, PNE accurately finds the optimal sequence route $Q(p,M)$.*

4 Complexity analysis

In this section we study the complexity of our proposed algorithms. For each algorithm we compute the expected number of distance computations and amount of workspace needed to answer an OSR query.

Without loss of generality, we assume that the points in U_{M_i}'s are uniformly distributed in a unit square universe. The number of these points is $N = \sum_{i=1}^{m} |U_{M_i}|$ where $m = |M|$. We also assume that the cardinalities of the U_{M_i}'s are equal (i.e., $|U_{M_i}| = \frac{N}{m}$). Therefore, we have

1. The expected distance between any pair of points each from a different U_{M_i} set is $\delta = 1/\sqrt{\frac{2N}{m}}$.
2. The expected number of points in U_{M_i} which are closer to the starting point p than threshold value T_v is $\pi T_v^2 \frac{N}{m}$ [13].
3. The expected $L(p, R_g)$ of the greedy route R_g is $m\delta$.

Considering the above values, the number of distance computations of the brute-first search algorithm is $C_{bfs} = m \prod_{i=1}^{m} \frac{N}{m} = O\left(\frac{N^m}{m^{m-1}}\right)$. Similarly, in the Dijkstra-based solution of Sect. 3.1, the graph G has $|V| = N+1$ vertices and $|E| = O\left(\frac{N^2}{m}\right)$ edges. Therefore, the complexity of this solution is $C_{edj} = O\left(\frac{N^3}{m^2}\log(N)\right)$.

4.1 LORD

We study the time complexity of LORD by enumerating the distance computation operations in its pseudo-code given in Fig. 4. We assume that LORD stores the length of all partial SRs built during its iterations. Therefore, it does not need any distance computation to find the length of those routes (i.e., $L(p,R)$ in line 12 in Fig. 4). The initialization step of LORD uses the threshold value T_v equal to the length of greedy route (i.e., $m\delta$) to select the points from U_{M_m} (lines 3–4). This step requires $|U_{M_m}| = \frac{N}{m}$ distance computations as the entire set U_{M_m} should be examined. Hence, the expected number of partial routes at the end of this step is $\mathrm{E}[|S|] = \pi m^2 \delta^2 \frac{N}{m}$ as we have $\mathrm{E}[T_v] = m\delta$. Table 4 lists a summary of the total cost of initialization step of LORD and those of its iterations.

The first iteration computes $\frac{N}{m}$ distances between the points of $U_{M_{m-1}}$ and p considering the current threshold (lines 8–9). The expected number of points which pass this check is $\pi \mathrm{E}[T_v]^2 \frac{N}{m}$. It also performs distance computations between the points resulted from the first check and the head of PSRs (lines 11–12). The expected number of these computations is $\mathrm{E}[|S|]\pi \mathrm{E}[T_v]^2 \frac{N}{m}$ where $\mathrm{E}[|S|]$ is the expected number of current PSRs. We assume that all of the points pass the second check in line 12 so the expected number of PSRs is updated to $\pi \mathrm{E}[T_v]^2 \frac{N}{m}$ at the end of this iteration. Meanwhile, the threshold value is decreased to $(m-1)\delta$ (line 16). Summarizing all above, the cost of the first iteration will be $\frac{N}{m} + \pi^2 m^4 \delta^4 \frac{N^2}{m^2}$. The cost of other iterations is computed similarly (see Table 4). Finally, the expected computation cost of LORD is calculated as

$$C_{lord} = N + \frac{\pi^2}{4}\left(m^4 + \sum_{i=2}^{m-1} i^2 (i+1)^2\right) = O(N + m^5) \tag{6}$$

Now, we calculate the expected amount of workspace required by LORD as the expected number of partial routes maintained in the set S during LORD's iterations. This value is the average of $\mathrm{E}[|S|]$ values in Table 4

$$M_{lord} = \frac{\pi}{2}\left(m + \sum_{i=2}^{m} \frac{i^2}{m}\right) = O(m^2) \tag{7}$$

4.2 R-LORD

The workspace requirement of R-LORD is the same as that of LORD given in Eq. 7 (i.e., $M_{r\text{-}lord} = M_{lord}$). The reason is that both algorithms use the same points to build the partial routes during their iterations. However, as each algorithm uses a different approach to select these points from their corresponding sets, their time complexities are different. LORD first scans the entire set U_{M_m} to select the points which are closer to p than T_v (lines 3–4 in Fig. 4). The complexity of this scan is $m \times \frac{N}{m}$

Table 4 Cost analysis of LORD

	$\mathrm{E}[T_v]$	Cost of lines 3–4		(workspace) $\mathrm{E}[\lvert S\rvert] = \pi \mathrm{E}[T_v]^2 \frac{N}{m}$	Total cost
Initialization	$m\delta$	$\frac{N}{m}$		$\pi m^2\delta^2 \frac{N}{m}$	$\frac{N}{m}$
Iteration	$\mathrm{E}[T_v]$	Cost of lines 8–9	Cost of lines 11–12	$\mathrm{E}[\lvert S\rvert] = \pi \mathrm{E}[T_v]^2 \frac{N}{m}$	Total cost
$i = m-1$	$m\delta$	$\frac{N}{m}$	$\pi^2 m^4 \delta^4 \frac{N}{m}$	$\pi m^2 \delta^2 \frac{N}{m}$	$\frac{N}{m} + \pi^2 m^4 \delta^4 \frac{N^2}{m^2}$
$i = m-2$	$(m-1)\delta$	$\frac{N}{m}$	$\pi^2 m^2 (m-1)^2 \delta^4 \frac{N}{m}$	$\pi (m-1)^2 \delta^2 \frac{N}{m}$	$\frac{N}{m} + \pi^2 m^2 (m-1)^2 \delta^4 \frac{N^2}{m^2}$
–	–	–	–	–	–
$i = m-j-1$	$(m-j)\delta$	$\frac{N}{m}$	$\pi^2 (m-j+1)^2 (m-j)^2 \delta^4 \frac{N}{m}$	$\pi (m-j)^2 \delta^2 \frac{N}{m}$	$\frac{N}{m} + \pi^2 (m-j+1)^2 (m-j)^2 \delta^4 \frac{N^2}{m^2}$
–	–	–	–	–	–
$i = 1$	δ	$\frac{N}{m}$	$\pi^2 3^2 2^2 \delta^4 \frac{N}{m}$	$\pi 2^2 \delta^2 \frac{N}{m}$	$\frac{N}{m} + \pi^2 2^2 3^2 \delta^4 \frac{N^2}{m^2}$
LORD				$\frac{\pi}{2}(m + \sum_{i=2}^{m} \frac{i^2}{m})$	$N + \frac{\pi^2}{4}(m^4 + \sum_{i=2}^{m-1} i^2 (i+1)^2)$

(= N in Eq. 6). However, R-LORD utilizes its single range query to perform this selection.[3] We compute the complexity of this range query.

R-LORD traverses R-tree m times from root node down to the leaves. For each accessed node, it performs an $O(1)$ *mindist* computation (line 23 in Fig. 8). Therefore, the complexity of each R-tree traversal is the same as the number of node accesses during this traversal. Following the cost mode of Tao et al. [13], the expected number of node accesses is given as

$$NA = \sum_{i=0}^{h-1} (n_i \cdot P_{NA_i}) \tag{8}$$

where h is the height of the R-tree, P_{NA_i} is the probability of accessing a node at level i, and n_i is the total number of nodes at level i. Tao et al. give estimations of h and n_i based on the total number of points N, the maximum node capacity and the average fan-out of each node. To estimate P_{NA_i} for each traversal, we only need to identify the *search region* of our traversal procedures. The search regions of R-LORD is the union of the ranges used in each traversal. We assume that the range used by an R-LORD's iteration is approximately the same as *Range*(**Q1**). Therefore, each R-LORD's iteration uses range $D(p,x) \leq \mathrm{E}[T_v]$ to select the point x. During LORD's analysis in Sect. 4.1, we calculated the value of $\mathrm{E}[T_v]$ for each LORD's iteration which is the same for the corresponding iteration (see Table 4). The first iteration uses $\mathrm{E}[T_v] = m\delta$ and all other subsequent iterations $1 < i \leq m$ use $\mathrm{E}[T_v] = (m-i+2)\delta$ in R-LORD. Consequently, the search region of each R-tree traversal is clearly identified. Once the search region is found, one can straightforwardly derive the probability P_{NA_i} from the cost model given in [13], hence the details are omitted.

We use $NA(i)$ to denote the expected number of node accesses during R-LORD's iteration i. Therefore, the expected total number of R-LORD's node accesses which is also R-LORD's expected number of I/Os is calculated as

$$IO_{r\text{-}lord} = \sum_{i=1}^{m} NA(i) \tag{9}$$

While the points within the range of each iteration are selected, both LORD and R-LORD perform the same set of distance computations for each selected point (lines 11–12 in Fig. 4 and lines 15–16 in Fig. 8). Therefore, we include the same number of computation operations calculated during the analysis of LORD in the time complexity of R-LORD (i.e., $C_{lord} - N = O(m^5)$). Hence, the expected time complexity of R-LORD follows:

$$\begin{aligned} C_{r\text{-}lord} &= \sum_{i=1}^{m} NA(i) + \frac{\pi^2}{4}\left(m^4 + \sum_{i=2}^{m-1} i^2 (i+1)^2\right) \\ &= \sum_{i=1}^{m} NA(i) + O(m^5) \end{aligned} \tag{10}$$

where $NA(i)$, the expected number of nodes accessed in iteration i, is easily derived from the cost model of [13].

4.3 PNE

PNE completely relies on the efficiency of the nearest neighbor algorithm in the metric space. Therefore,

[3] We assume that both algorithms use the same range. Note that this is a conservative assumption for the general case where R-LORD uses a combination of the circular range *Range*(**Q1**) and the rectangular range *MBR*(**Q2**).

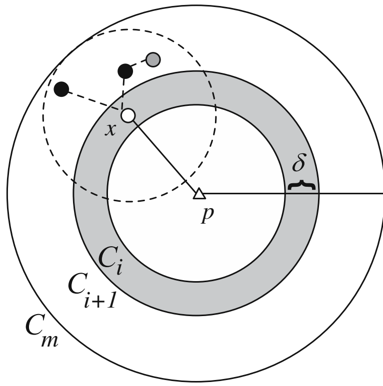

Fig. 10 The locus of all points $x \in U_{M_1}$ with which a PC route can start

it extensively performs NN queries on the database.[4] Hence, the time complexity of PNE can be computed as the number of NN queries issued by PNE. We assume that PNE only generates the PC routes. That is, it generates only the routes R that follow a prefix sequence $Pfx(M,n)$ for which we have $L(p,R) \leq L(p,Q(p,M))$. Now, as PNE issues a single NN query to generate each PC route, we count the expected number of PC routes added in to PNE's heap to calculate PNE's time complexity. Note that in general case PNE may generate non-PC routes as well.

Assume that $N_{pne}(m)$ is the number of PC routes that PNE generates for sequence M ($m = |M|$). Note that in the average case, $N_{pne}(m)$ only depends on the size of the sequence M (i.e., m). Also, assume that the $E[L(p,Q(p,M))]$, the expected value of $L(p,Q(p,M))$, is $m\delta$. Therefore, we need to count all the PC routes R for which we have $L(p,R) \leq m\delta$. Figure 10 illustrates the starting point p and the circle C_m centered at p with radius $m\delta$. Corresponding to each point $x \in U_{M_1}$ which is inside or on the circle C_m (i.e., $D(p,x) \leq m\delta$), PNE generates PC route (x). Therefore, for sequence M, the number of PC routes with size one generated by PNE is $\pi m^2 \delta^2 \frac{N}{m}$ (i.e., $\frac{\pi}{2} m^2$).

Now, we need to count the PC routes with more than one point. To follow the sequence M or any of its prefix sequences, each of these routes start with a point $x \in U_{M_1}$ inside or on circle C_m. For each point x, we recursively calculate the number of PC routes of query $Q(p,M)$ which start with x. Figure 10 divides the points such as x into m groups using circles C_i. Each circle C_i shown in the figure is centered at p with radius $i\delta$. The circle C_0 is an empty circle. Each group i includes all the points x between the two circles C_{i-1} and C_i ($(i-1)\delta \leq D(p,x) < i\delta$). We can assume that all the points of group i are almost within the same distance $i\delta$ from p. For each point x of group i, the number of PC routes of query $Q(p,M)$ which start with x is the same as the number of PC routes of query $Q(x,M')$ where $|M'| = m - i$ (see Fig. 10). Therefore, the number of PC routes which start with x is $N_{pne}(m-i)$. Furthermore, no PC route with size more than one starts with the points in group m. The reason is that p's distance to any point x in this group is already $m\delta$ which makes any route that starts with x longer that $m\delta$.

The expected number of points in group i is $A(i)\frac{N}{m}$ where $A(i) = \pi(2i-1)\delta^2$ is the area of the space including group i (i.e., the space between C_{i-1} and C_i). Hence, the number of PC routes which start with the points in group i is calculated as $\pi(2i-1)\delta^2 \frac{N}{m} N_{pne}(m-i) = \frac{\pi}{2}(2i-1)N_{pne}(m-i)$. Summing up the number of PC routes of size one and those generated by each group i, we get the following:

$$N_{pne}(m) = \frac{\pi}{2}m^2 + \sum_{i=1}^{m-1} \frac{\pi}{2}(2i-1)N_{pne}(m-i) \tag{11}$$

Equation 11 shows the expected number of all PC routes generated and examined by PNE for a given query $Q(p,M)$. Thus, the expected time complexity of PNE for the OSR query $Q(p,M)$ will be

$$C_{pne} = C_{NN} \times N_{pne}(|M|) \tag{12}$$

where C_{NN} is the cost of performing a single first (or next) nearest neighbor query. This cost depends on the incremental nearest neighbor approach used by PNE. The expected amount of workspace required by PNE is also equal to the expected number of PC routes generated. Therefore, this leads to $M_{pne} = N_{pne}(|M|)$.

5 Variations of OSR queries

In this section, we address two variations of OSR queries. The first variation is when a destination point also exists, and the second variation is when k optimal routes are requested.

5.1 Directed-OSR

Assume that the user asks for an optimal sequenced route that follows the given sequence which starts from a source and ends in a given destination. A special case of this query is where the source and destination points are the same, i.e., the user intends to return to her

[4] The main objective of PNE is to reduce the number of partial routes maintained during its execution. Other efficiency issues are handled by the NN approach.

starting location. We start by formally defining this type of query as:

Definition 7 *Given source point p, destination point q and a sequence M, the* Directed-OSR *query is defined as finding* $R = (P_1, \ldots, P_m)$*, a sequenced route that follows M, where the following function G is minimum over all sequence routes that follow M*:

$$G(p, R, q) = D(p, P_1) + L(R) + D(P_m, q) \tag{13}$$

The above equation is similar to $L(p, R) + D(P_m, q)$. We show that this new form of OSR can easily be reduced to the general form of OSR.

We define a new set $U_{n+1} = \{q\}$. Including this new set in the set of U_i's makes $M' = (M_1, \ldots, M_m, n+1)$ a valid sequence in the new setting of the problem. Now if we assume that $Q(p, M') = R' = (P'_1, \ldots, P'_{m+1})$, we know that P'_{m+1} will be q as q is the only member of U_{n+1}. Moreover, $L(p, R')$ is minimum over all candidate routes that follow M'. Recall that the length of the route $R'_p = p \oplus R'$ (i.e., $L(p, R')$) is equal to $D(p, P'_1) + L(R')$. We define the route R as $(P'_1, \ldots, P'_m)$ by excluding q from R'. It is clear that $L(p, R')$ is the same as $D(p, P_1) + L(R) + D(P_m, q)$. By comparing the latter expression with $G(p, R, q)$ of Eq. 13, we conclude that R is the answer to the directed-OSR query given the source p, destination q, and sequence M.

Since we showed that directed-OSR can be reduced to a general OSR problem, we are able to use our LORD (or R-LORD) algorithm to answer this query. Specifically, the answer to directed-OSR given the source p, destination q, and sequence M is the same as the answer to OSR query $Q(p, M')$ excluding the point q, where $U_{n+1} = \{q\}$ and $M' = (M_1, \ldots, M_m, n+1)$. Although R-LORD can similarly solve directed-OSR, we can further optimize it for directed-OSR. This is achieved by neglecting the first range query **Q1**. The reason is that the only point in this range is q. Therefore, the set S can be directly initialized to $\{(q)\}$.

5.2 k-OSR

The second variation of OSR is when the user asks for the k routes with the minimum total distances to its location. We define this as *k-OSR* query. We can easily address this type of query using both our R-LORD and PNE approaches.

Recall that in PNE, we maintain a heap of the partially completed sequenced routes and only keep one candidate sequenced route (or in other words, a route that follows M), that is the one that has the minimum total length. By modifying this policy to maintain k candidate SRs in the heap and continuing the iterations until k candidate SRs are fetched from the heap, PNE can also address k-OSR queries.

6 Performance evaluation

We conducted several experiments to evaluate the performance of our proposed approaches. First, we compared the query response time of R-LORD with those of LORD and the Dijkstra-based solution. Moreover, we evaluated R-LORD with respect to: (1) disk I/O accesses incurred by its underlying R-tree index structure, (2) effectiveness of its range query, and (3) its overall query response time. Finally, we evaluated the performance of PNE with respect to: (1) its heap size (i.e., number of route in the heap), (2) the time required to maintain the heap, (3) the number of NN queries issued for each point set, and (4) its overall query response time using a real road network. We evaluated all our proposed approaches by investigating the effect of the following parameters on their performances: (1) size of sequence M in $Q(p, M)$ (i.e., number of points in the optimal route), (2) cardinality of the datasets (i.e., $\sum_{i=1}^{n} |U_i|$), and (3) density and distribution of the datasets.

We used one real and two synthetic datasets for our experiments. The real data is obtained from the U.S. Geological Survey (USGS)[5] and consists of the locations of different businesses (e.g., schools) in the entire country. The synthetic datasets consist of randomly generated set of points with uniform and Zipf distributions. Table 5 shows the characteristics of the datasets. The real dataset has a total of 950,000 points. However, in our experiments, we randomly selected sets of 40, 70, 250 and 500 K points from this dataset. The cardinality of each synthetic dataset is 480,000. Each dataset is indexed by an R*-tree index with the page size of 1 K bytes and the maximum of 50 entries in each node (capacity of the node). The experiments were performed on a DELL Precision 470 with Xeon 3.2 GHz processor and 3 GB of RAM. We ran 1,000 OSR queries initiated from randomly selected starting points and reported the average of the results.

In the first set of experiments, we compared the performance of R-LORD, LORD and the Dijkstra-based solution. Note that the weighted directed graph G (see Sect. 3.1) for even a small dataset is a substantially large graph. For example, for a real dataset with 40,000 points and $|M| = 3$, G has 22,400 nodes and 124 million edges. This will result in substantially large query response times for the naive Dijkstra-based solution

[5] http://www.geonames.usgs.gov/.

Table 5 Datasets used in our experiments

USGS		Synthetic	
Points	Size	Points	Size
Hospital	5,314	P1 (uniform)	32,000
Building	15,127	P2 (uniform)	64,000
Summit	69,498	P3 (uniform)	128,000
Cemetery	109,557	P4 (uniform)	256,000
Church	127,949	P5 (Zipf)	32,000
School	139,523	P6 (Zipf)	64,000
Populated place	167,203	P7 (Zipf)	128,000
Institution	319,751	P8 (Zipf)	256,000

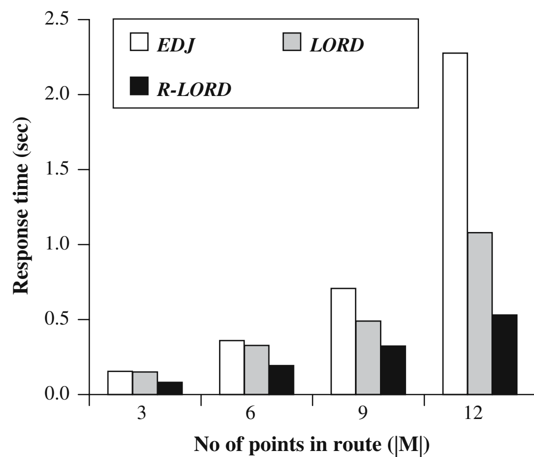

Fig. 11 Query response time versus sequence size $|M|$

(e.g., 40 s for the 40 K example). Therefore, we do not report the query and workspace costs of this expensive approach. Instead, we compare R-LORD and LORD's performances with that of enhanced Dijsktra-based approach in which the length of the greedy route is used to reduce the number of candidate points (see Sect. 3.1).

Figure 11 shows the query response time for R-LORD, LORD and the enhanced Dijkstra-based approach (EDJ) when the number of points in optimal route (i.e., $|M|$) varies from 3 to 12. While the figure depicts the results from the experiment on 250 K USGS dataset, the trend is the same with those of all of our datasets with different cardinalities and distributions. As shown in the figure, both EDJ and LORD answer an OSR query very quickly for small values of $|M|$ (less than 100 ms for $|M| = 3$). However, R-LORD outperforms both approaches for all values of $|M|$. The figure also shows that as the value of $|M|$ increases, the response time of the EDJ increases with a rate that is substantially more than that of LORD and R-LORD, confirming the impracticality of the Dijkstra-based solution for OSR on large graphs and the efficiency of LORD. Meanwhile, it depicts the performance gain due to utilizing R-tree in R-LORD. Due to the superiority of R-LORD over LORD, for the rest of the experiments, we only report the results for R-LORD.

In the second set of experiments, we varied the size of M and measured the performance of R-LORD. Figure 12a, b, c depict the performance of R-LORD on a randomly selected real dataset with 250 K points when the size of M varies from 3 to 12. For this dataset, 7,291 nodes are generated in R*-tree. Figure 12a illustrates the percentage of R*-tree nodes that were accessed by R-LORD. As shown in the figure, between 1% (for small values of $|M|$) to 11% (for large values of $|M|$) were accessed by R-LORD. The figure also shows that the rate in which the number of accessed nodes increases is slightly more than linear. That is, while the percentage of accessed nodes increases from 1 to 2% (i.e., 2 times) when $|M|$ increases from 3 to 6, it increases from 2 to 11% (i.e., 5.5 times) when M increases from 6 to 12. This is because for larger values of $|M|$, more nodes are examined against **Q2** and *mindist*() function. Figure 12b shows the total query response time of R-LORD for the same dataset. As shown in the figure, even for a large value of 12 for $|M|$, R-LORD can answer the query in less than 0.8 s. Moreover, it shows that the rate of increase in the processing time closely follows the rate of increase in accessed nodes, indicating that traversing R*-tree is the major factor in R-LORD.

Figure 12c shows the performance of the range queries of R-LORD. The bars in the figure indicate the required workspace of R-LORD (WS) as the maximum number of points that were stored in the partial SRs of S (see Sect. 3.2.2). As shown in the figure, the number of points filtered in by the range queries are substantially less than the cardinality of the points (e.g., for $|M| = 6$, only 110 points out of 250,000 are selected). This shows that the range queries of R-LORD are extremely effective. The figure also compares the effectiveness of the range queries of R-LORD. It shows the percentage of reduction in the number of selected points as compared to the Dijkstra-based approach. In the later approach the only filter is one simple range query with a range based on the length of the greedy sequenced route ($L(p, R_g(p, M))$). This is shown as vertical lines in the figure, where each line indicates the maximum, minimum, and average value of this reduction for a given M. The figure confirms that our range queries provide a filter with better selectivity as compared to the simple range query. For example, for $|M| = 6$, the decrease in the size of the candidate points is between 48 and 97.7% with an average of 77.4%. Figure 13a, b, c shows the result of the same set of experiments for our first set of synthetic data (i.e., with uniform distribution). This dataset has 250,000 points and generate

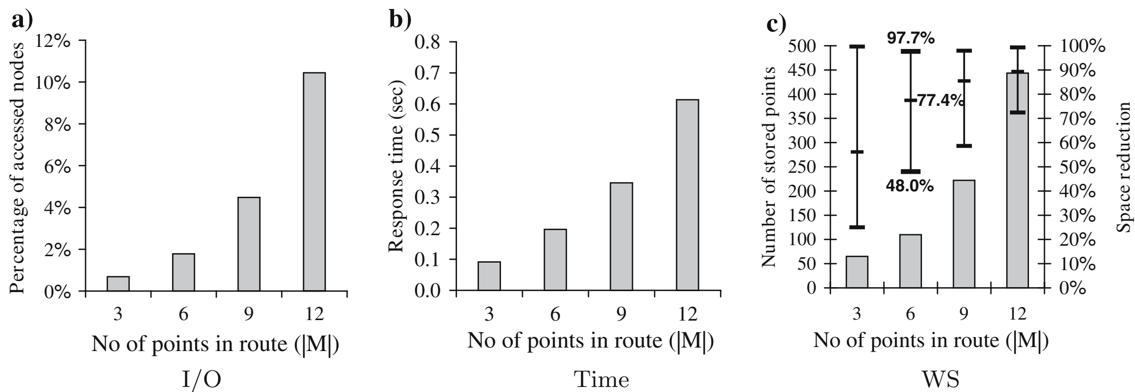

Fig. 12 Query cost versus sequence size $|M|$ (250 K USGS)

6,804 nodes in the R*-tree. The figures show identical behavior for the synthetic data as compared to the real dataset. It also shows that the range queries can filter out up to 99% of the points as compared to the simple range query with the range equal to $L(p, R_g(p, M))$.

Figure 14a, b, c shows the results of our third set of experiments, where we investigated the impact of the cardinality of the points on the efficiency of R-LORD. We varied the cardinality of our real dataset from 40 to 500 K and ran OSR queries of sequence size $|M| = 6$. Figure 14a shows the percentage of accessed nodes of R*-tree for different cardinalities of the dataset. As shown in the figure, the percentage of accessed nodes in R*-tree decreases as the cardinality of the data increases, indicating that R-LORD can efficiently scale to large datasets. Moreover, Fig. 14b shows that the processing time of R-LORD increases slightly as the cardinality of the data increases. For example, the query response time is 0.09 s for 40,000 points, and it only increases to 0.32 s (i.e., a factor of 3.5) when the number of points increases by a factor of 12. This verifies the scalability of R-LORD. Figure 14c shows the performance of the range queries for different cardinalities of the dataset. The figure shows that for a dataset with 70,000 points, only 100 (0.142%) of them are selected as the result of the range queries. The figure also indicates that this percentage decreases for larger cardinalities of data. For example, in the dataset with 500,000 points, only 110 (0.022%) are selected. Our experiments for the synthetic data show similar trends but due to lack of space, we omit the graphs for the synthetic data.

Our next set of experiments were aimed to evaluate the performance of R-LORD when the densities of the datasets U_{M_i}'s specified by the query sequence M are different. We used R-LORD to answer five different categories of queries $Q(p, M)$, each with a different *pattern of change* in the density of the datasets. The categories are

1. **LL:** The density of points is significantly decreasing from U_{M_1} to U_{M_m}. For example, a query for an optimal route to an institution (i.e., 319,751 points), then to a church (i.e., 127,949 points) and finally to a hospital (i.e., 5,314 points) in USGS dataset falls in this category.
2. **LU:** There is an $1 < i < |M|$ (usually $i = |M|/2$) where the density is decreasing from U_{M_1} to U_{M_i} and increasing from U_{M_i} to U_{M_m} (e.g., (church, hospital, school)).
3. **MM:** The densities of all U_{M_i}'s are almost the same (e.g., (school, church, school)).
4. **UL:** There is an $1 < i < |M|$ where the density is increasing from U_{M_1} to U_{M_i} and decreasing from U_{M_i} to U_{M_m} (e.g., (church, school, hospital)).
5. **UU:** The density is significantly increasing from U_{M_1} to U_{M_m} (e.g., (hospital, church, institution)).

Figure 15a, b, c illustrates the results of our experiments where M follows the above density distribution categories. In these experiments, $|M| = 6$ and the data consists of 250,000 points selected from USGS dataset. Figure 15a shows that although the percentage of the accessed nodes varies for different density categories, they are still in the range of 1 to 2%. Moreover, the query response times shown in Fig. 15b indicate that regardless of the density of the points, R-LORD answers OSR queries with almost identical response times. Figure 15c depicts that although the range queries perform similarly for different density categories, the selectivity of the range queries for LU and UU is slightly less than that of LL, MM and UL. The reason for this is that the last set of points in M (i.e., U_{M_m}), which are selected first by **Q1** (recall that R-LORD constructs the partial roads from U_{M_m} toward U_{M_1}), are denser and hence, **Q1** selects more number of points to be included in the PRCs in S. Our experiments for the synthetic data show

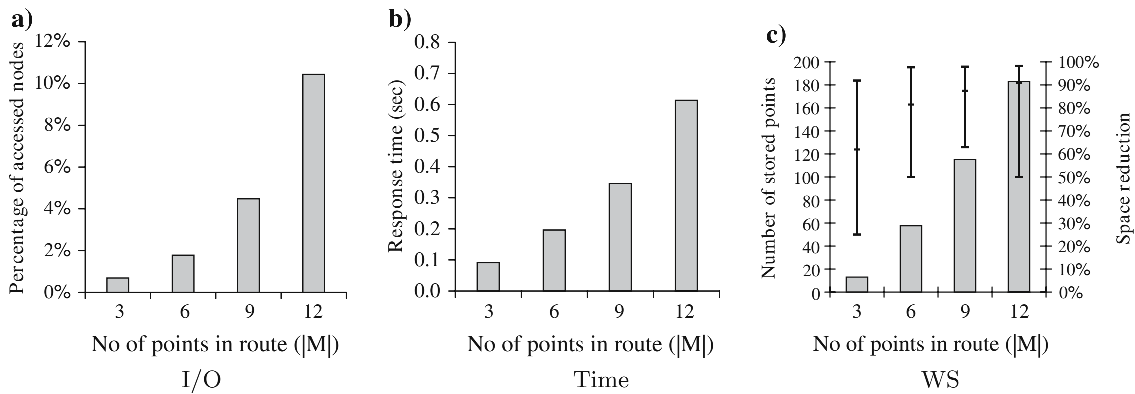

Fig. 13 Query cost versus sequence size $|M|$ (250 K uniform)

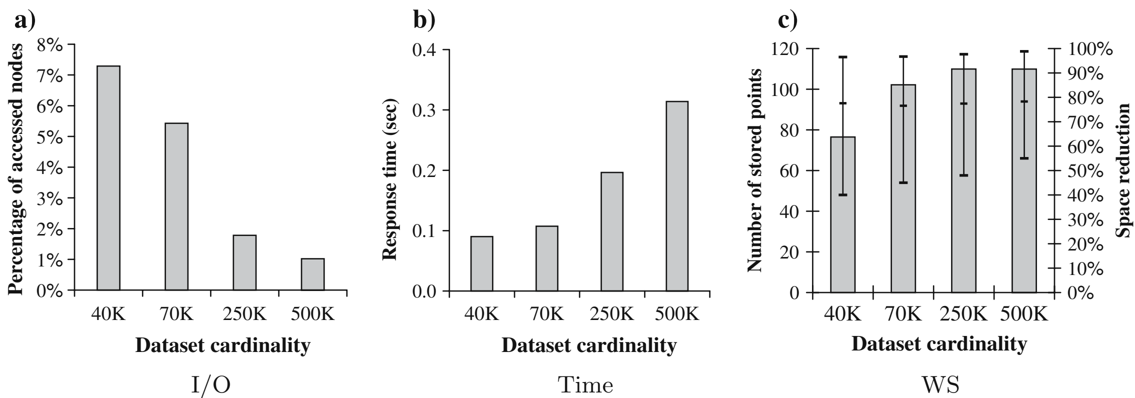

Fig. 14 Query cost versus cardinality, USGS data, ($|M| = 6$)

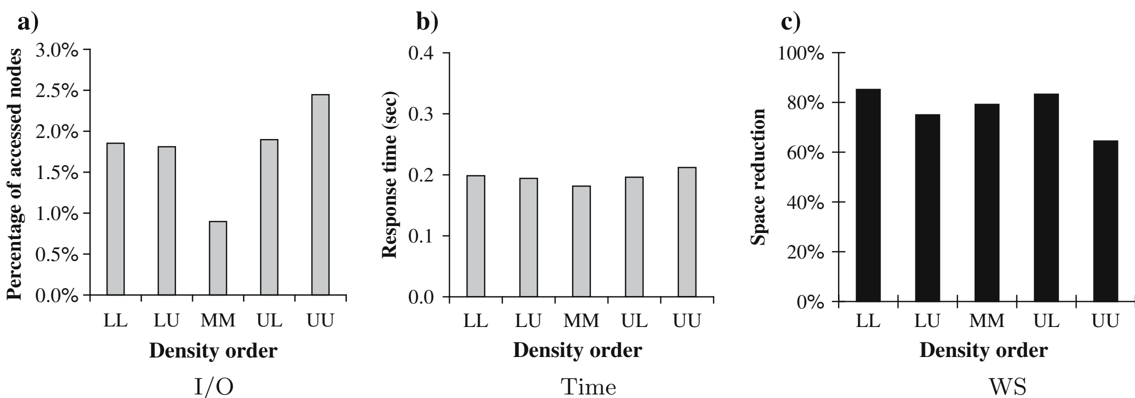

Fig. 15 Query cost versus density, 250 K USGS data, ($|M| = 6$)

similar trends but due to lack of space, we omit the graphs for the synthetic data.

We also studied the impact of the distributions of the datasets on R-LORD by performing R-LORD for different values of $|M|$ on the two synthetic datasets. The results showed that R-LORD performs independently from the distribution of the datasets; that is, the R-Tree nodes accessed, the processing time, and the space reduction were almost identical for the two datasets.

We used the Euclidean distance function together defined in the vector spaces of our previously described datasets for PNE experiments measuring heap size, its maintenance time and number of NN queries. The reason is that PNE's performance in terms of these factors is independent from the space/distance. In the next set of experiments, we evaluated the performance of PNE by investigating the effect of $|M|$ and the density of the datasets. Figure 16 depicts the average size of the

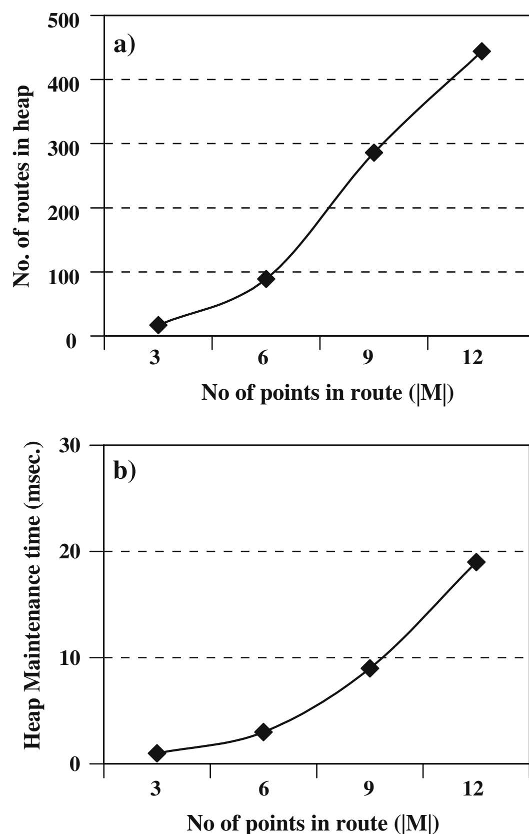

Fig. 16 Evaluation of heap in PNE

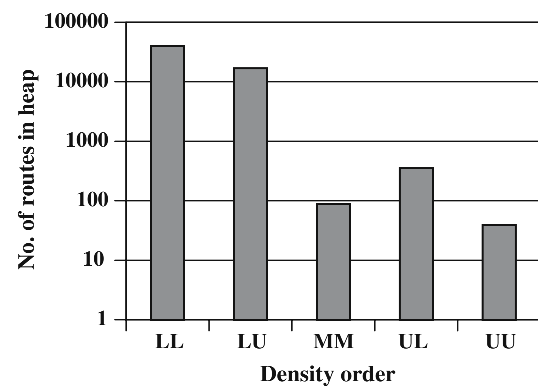

Fig. 17 Number of routes in heap versus density ($|M| = 6$)

Table 6 Number of NN queries issued by PNE ($|M| = 6$)

Density order	Number of NN queries					
	U_{M_1}	U_{M_2}	U_{M_3}	U_{M_4}	U_{M_5}	U_{M_6}
LL	437	26,403	33,173	12,882	6,048	371
LU	351	16,454	16,473	374	8	3
MM	17	59	58	24	14	5
UL	16	40	109	282	227	30
UU	11	19	15	15	14	4

heap (i.e., number of routes in the heap) and the time required by PNE to maintain the heap when the densities of all U_{M_i}'s are similar (i.e., MM category) and $|M|$ varies from 3 to 12. Figure 16a shows that the size of the heap is (almost) a linear function of the size of the sequence and hence, suggests that PNE scales for larger values of $|M|$. Figure 16b shows that although the time required to maintain the heap increases with $|M|$ (since the size of heap becomes larger), this time is still less than 20 ms. Our experiments also showed that this time is always less than 0.1% of the total processing time and hence, is negligible as compared to the time required to perform the NN queries.

In the next set of experiments, we studied the effect of the changes in the densities of the datasets of M on PNE. Figure 17 shows the heap size when $|M| = 6$ and M follows the previously defined distribution categories. The figure suggests that when the densities of the U_{M_i}'s are either similar or increasing from U_{M_1} to U_{M_m} (i.e., MM and UU, subsequently), PNE requires a very small heap (and consequently, has very small heap maintenance time). However, decrease in the density of the U's from U_{M_1} to U_{M_m} results in to larger heap sizes, particularly if the decrease is in the first portion of M (i.e., categories LL and LU). This indicates that as opposed to R-LORD, PNE's performance is sensitive to the distribution of the densities in M. The intuition here is that for categories LL and LU, where the first group of datasets in M (i.e., U_{M_1}, U_{M_2},...) are dense as compared to ($U_{M_i}, U_{M_{i+1}}, \ldots$ where $i > 1$), their distances to each other are smaller than the distances between ($U_{M_i}, U_{M_{i+1}}, \ldots$) and hence, PNE must perform exhaustive search on $\{U_{M_1}, U_{M_2}, \ldots\}$ before examining $\{U_{M_m}, U_{M_{m-1}}, \ldots\}$.

Table 6 shows the number of NN queries issued by PNE on each dataset when $|M| = 6$. Note that for the category LU shown in the table, the density decreases from U_{M_1} to U_{M_3} and then increases from U_{M_4} to U_{M_6}. The table shows that very large number of NN queries are performed when the density of the U's decreases (i.e., LL and first portion of LU) and further verifies the intuition described above.

Finally, the last set of experiments focuses on evaluating the performance of PNE in a real-world road network space. Our road network dataset is obtained from NAVTEQ Inc.,[6] used for navigation systems with GPS devices installed in cars, and represents a network of approximately 110,000 links and 79,800 nodes of the road system in the downtown Los Angeles. The point dataset is a subset of USGS dataset including different

[6] http://www.navteq.com/.

businesses in the same area. We implemented PNE using our VN^3 nearest neighbor approach for road network spaces [9]. Our experiments show that PNE's average query response time for OSR queries on our road network dataset is approximately 7.8 s. This shows that PNE can efficiently answer a typical OSR query on road network databases. However, as the number of PNE's NN queries is very sensitive to the density of the points in U_{M_i}'s, PNE may require hundreds of seconds for processing an OSR query on very sparse point datasets. This performance is still acceptable as asking for only sparse locations in an OSR query generally happens in planing for long trips for which the user can tolerate a reasonably longer time to find the optimal route .

7 Related work

In this section, we first review the related work in the area of graph theory. We then provide an overview of the related studies on variations of the nearest neighbor queries in spatial databases.

The only similarity between traveling salesman problem (TSP) and OSR is that both search for a route of minimum cost in a graph. The general form of TSP was first studied in the 1930s by Karl Menger in Harvard [2]. The most similar instance of TSP to OSR is sequential ordering problem (SOP) which sets some precedence constraints on the route [1]. Each constraint requires that a node of the graph be visited before some other node. Although the number of nodes is small in SOP and general TSP, the unknown traveling sequence makes them NP-hard. However, OSR dictates a given strict sequence order of point *types* where each point must be selected from a large set per type.

The OSR problem is also related to the problem of finding shortest path (SP) in directed weighted graphs. Two classic algorithms for solving SP in main memory are Dijkstra's and Bellman-Ford algorithms [4]. However, for addressing SP on the huge graph G of Sect. 3.1, an external memory algorithm is required. Hutchinson et al. [7] propose a tree data structure for answering SP queries on a planar graph stored in external memory. Chan at al. [3] describe a disk-based algorithm to find SP on large network systems. They partition the original large graph and search for the shortest path by locally searching in its smaller pieces. While these approaches eliminate the overheads of loading the huge graph in main memory, they are not applicable for OSR queries. The reason is that OSR graph's topology is dependent on the user's query $Q(p, M)$. Since point p and sequence M are not known in advance, this graph must be built on demand as described in Sect. 3.1. Therefore, if we intend to use an external memory SP approach, we need to store the graph on disk blocks before processing it. This makes the approach expensive and therefore impractical.

Numerous algorithms for k-nearest neighbor queries in spatial databases have been proposed. A majority of these algorithms are based on utilizing spatial index structures such as R-tree and usually perform in two filter and refinement steps. Roussopoulos et al. [12] present a branch-and-bound R-tree traversal algorithm that uses two *mindist minmaxdist* metrics. Hjaltason et al. [6] propose an incremental nearest neighbor algorithm that is based on utilizing an index structure and a priority queue. Jensen et al. in [8] discuss data models and graph representations for NN queries in road networks and provide alternative solutions for it. Papadias et al. [11] propose a solution for NN queries in network databases by generating and expanding a search region around a query point. In our previous work [9], we proposed a solution for NN queries in road network databases that is based on utilizing network Voronoi diagrams. Other variations of k nearest neighbor queries have also been studied and their solutions are usually motivated by the solutions of their regular k nearest neighbor queries.

In an independent research, Li et al. [10] studied trip planning queries (TPQ), a class of queries similar to our OSR query. With a TPQ, the user specifies a subset (not a sequence) of location types R and asks for the optimal route from her starting location to a specified destination which passes through at least one database point of each type in R. In particular, TPQ eliminates the sequence order of OSR to define a new query. Consequently, finding accurate solutions to TPQ becomes NP-hard as the size of candidate space significantly grows. The paper proposes several approximation methods to provide near-optimal answers to TPQs. However, our proposed solutions to OSR efficiently compute the exact optimal route. In theory, OSR algorithms are able to address address TPQ queries. This can be done by running any OSR algorithm $|R|!$ times, each time using a permutation of point types in R as the sequence M and returning the optimal route among all the resulted routes. This exponentially complex solution is inefficient in practice. The approximation algorithms of [10] are designed to handle the exponential growth of the problem's search space.

In parallel with our study, Terrovitis et al. [14] addressed k-stops shortest path problem for spatial databases. The problem seeks for the optimal path from a starting location to a given destination which passes through exactly k intermediate points of the location database. The k-stops problem is a specialized case of OSR queries. To be specific, OSR query $Q(p, M)$ reduces

to a k-stops problem where p is the starting location, $M = (1, \ldots, 1)$, $|M| = k$, and U_1 is the single given point set representing the location database. The destination is considered by the variation *directed-OSR* of the above OSR query. A major advantage of our OSR approach over k-stops is that we address solutions to OSR queries in both metric and vector spaces while [14] only considers Euclidean space. Our PNE approach can answer K best optimal routes of OSR in a metric space.

8 Conclusions and future work

We studied the new problem of optimal sequenced route query in both vector and metric spaces. To tackle the problem, we first proposed a Dijkstra-based approach and showed that it is not efficient for large point sets and/or routes with large number of points. Hence, we proposed a novel threshold-based algorithm, LORD, for vector spaces. Subsequently, we proposed R-LORD which transforms the LORD's thresholds into a range query and then utilizes an R-tree index structure to support OSR queries more efficiently. We conducted an extensive set of experiments to evaluate R-LORD and our main observations are

- R-LORD is *light* in terms of required workspace; as compared to the Dijkstra-based approach, it always reduces the required workspace by a factor of (on average) 55–90%. The maximum of this space reduction reaches 99.6% for some instances of our experiments.
- R-LORD is *efficient* in terms of query response time; it answers an OSR query in a time which increases with almost a linear rate as the sequence size $|M|$ increases. Moreover, R-LORD substantially outperforms the Dijkstra's classic algorithm.
- R-LORD is *I/O efficient*; it accesses at most 10.5% of the R-tree nodes.

We also proposed PNE, a progressive algorithm for metric spaces that generates the optimal route from the starting to the ending point. We showed that PNE performs efficiently, particularly when the distributions of the datasets of M are either similar, or increasing from U_{M_1} to U_{M_i}. Moreover, we showed that the overhead of PNE is always negligible as compared to the overhead of the nearest neighbor approach that it employs.

This paper is our first attempt to tackle the core problem of OSR and evaluate its base solutions. As part of our future work, we intend to study other variations of the OSR query as well as improving our base solutions. In particular, we plan to extend our definition of OSR query to include more general precedence constraints on the points of the optimal route.

Acknowledgements This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (PECASE), IIS-0324955 (ITR), and unrestricted cash gifts from Google and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

References

1. Ascheuer, N., Jünger, M., Reinelt, G.: A branch and cut algorithm for the asymmetric traveling salesman problem with precedence constraints. Comput. Optim. Appl. **17**(1), 61–84 (2000)
2. Biggs, N., Lloyd, E.K., Wilson, R.J.: Graph theory, pp. 1736–1936. Clarendon Press, Oxford (1986)
3. Chan, E.P.F., Zhang, N.: Finding shortest paths in large network systems. In: Proceedings of the 9th ACM international symposium on Advances in geographic information systems, pp. 160–166. ACM Press (2001). http://www.doi.acm.org/10.1145/512161.512197
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to algorithms. MIT Press/McGraw-Hill, New York (1990)
5. Hadjieleftheriou, M., Kollios, G., Bakalov, P., Tsotras, V.J.: Complex spatio-temporal pattern queries. In: Böhm, K., Jensen, C.S., Haas, L.M., Kersten, M.L., Larson, P.Å., Ooi, B.C. (eds.) Proceedings of the 31st International Conference on Very Large Data Bases: VLDB'05, pp. 877–888 (2005)
6. Hjaltason, G.R., Samet, H.: Distance browsing in spatial databases. ACM Trans. Database Syst. **24**(2), 265–318 (1999). http://www.doi.acm.org/10.1145/320248.320255
7. Hutchinson, D., Maheshwari, A., Zeh, N.: An external memory data structure for shortest path queries. Discret. Appl. Math. **126**(1), 55–82 (2003). http://www.dx.doi.org/10.1016/S0166-218X(02)00217-2
8. Jensen, C.S., Kolář, J., Pedersen, T.B., Timko, I.: Nearest neighbor queries in road networks. In: Proceedings of the 11th ACM international symposium on Advances in geographic information systems, pp. 1–8. ACM Press (2003). http://www.doi.acm.org/10.1145/956676.956677
9. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based K nearest neighbor search for spatial network databases. In: Nascimento,M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefer, K.B. (eds.) Proceedings of the 30th International Conference on Very Large Data Bases: VLDB'04, pp. 840–851 (2004)
10. Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., Teng, S.H.: On trip planning queries in spatial databases. In: Medeiros, C.B., Egenhofer, M., Bertino E. (eds.) Proceedings of the 9th International Symposium on Spatial and Temporal Databases: SSTD'05. Lecture Notes in Computer Science, vol. 3633, pp. 273–290. Springer, Berlin Heidelberg New York (2005)
11. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: Freytag, J.C., Lockemann, P.C., Abiteboul, S., Carey, M.J., Selinger, P.G.,

Heuer, A. (eds.) Proceedings of the 29th International Conference on Very Large Data Bases: VLDB'03, pp. 802–813 (2003)
12. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995, pp. 71–79. ACM Press (1995)
13. Tao, Y., Zhang, J., Papadias, D., Mamoulis, N.: An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. IEEE Trans. Knowl. Data Eng. **16**(10), 1169–1184 (2004). http://www.dx.doi.org/10.1109/TKDE.2004.48
14. Terrovitis, M., Bakiras, S., Papadias, D., Mouratidis, K.: Constrained shortest path computation. In: Medeiros, C.B., Egenhofer, M., Bertino E. (eds.) Proceedings of the 9th International Symposium on Spatial and Temporal Databases: SSTD'05. Lecture Notes in Computer Science, vol. 3633, pp. 181–199. Springer, Berlin Heidelberg New York (2005)