LNCS 3428, pp120-135, April 2005.
(URL: http://www.springerlink.com/link.asp?id=hqy70fvd46yb2taw)

In-Route Skyline Querying for Location-Based Services

Xuegang Huang and Christian S. Jensen

Department of Computer Science, Aalborg University, Denmark
{xghuang,csj}@cs.aau.dk

Abstract. With the emergence of an infrastructure for location-aware mobile services, the processing of advanced, location-based queries that are expected to underlie such services is gaining in relevance. While much work has assumed that users move in Euclidean space, this paper assumes that movement is constrained to a road network and that points of interest can be reached via the network. More specifically, the paper assumes that the queries are issued by users moving along routes towards destinations. The paper defines in-route nearest-neighbor skyline queries in this setting and considers their efficient computation. The queries take into account several spatial preferences, and they intuitively return a set of most interesting results for each result returned by the corresponding non-skyline queries. The paper also covers a performance study of the proposed techniques based on real point-of-interest and road network data.

1 Introduction

Location-based services (LBSs) utilize consumer electronics, mobile communications, positioning technology, and traditional map information to provide mobile users with new kinds of on-line services. Examples include location-sensitive information services that identify points of interest that are in some sense nearest and of interest to their users and that offer travel directions to their users. Data management is a core aspect of the provisioning of LBSs, and advanced services pose new challenges to data modeling as well as update and query processing.

Using the moving users' freedom of movement, three scenarios for LBSs can be distinguished. First, unconstrained movement is characterized by the mobile users being able to move freely in physical space. Next, constrained movement occurs when movement is constrained by obstacles, e.g., buildings and restricted areas. The third scenario is that of network-constrained movement, which is this paper's focus. Here, user movement is restricted to a transportation network, and Euclidean distances are generally of little use. Rather, the notion of travel distance takes center stage, and query processing techniques must use this distance notion.

This paper considers so-called in-route queries. These assume that the user's destination is known, in addition to the user's current location; and they assume that an anticipated route towards the destination is known. This setting is motivated by the observation that few mobile users move about aimlessly, but rather travel towards a known destination along a known route. Such routes can be obtained from navigation systems or past behavior [3].

Next, users are likely to take several spatially-related criteria, with varying weights, into account when deciding on points of interest to visit. As examples, a user looking

Y.-J. Kwon, A. Bouju, and C. Claramunt (Eds.): W2GIS 2004, LNCS 3428, pp. 120–135, 2005.

for a gas station may prefer to minimize detour rather than distance, while a user searching for an emergency room is likely to be interested in minimizing the distance and is insensitive to the detour. We propose to use the mechanism inherent in the skyline operator [2] to balance several criteria, and we generalize the skyline mechanism to enable queries that return larger result sets from which the user can then choose. The skyline mechanism returns a result if no other result exists that is better with respect to all the criteria considered. This is useful when a total ordering cannot be defined on the space of all criteria.

The paper offers algorithms for in-route kth order skyline queries, and it covers performance studies with real point-of-interest and road network data. Although focus is on spatial preferences, non-spatial preferences can be integrated into the contribution.

Inspired by recent work by Speičys et al. [17] and Hage et al. [8], we use generic data structures for representing a road network and points of interest within the network. These structures separate the network topology from the points of interest, which is important for maintainability, and are kept simple, so that the paper's contributions are broadly applicable.

Query processing for network-constrained moving objects has recently received attention in the research literature. Data models and query processing frameworks [8, 15, 17] as well as indexing methods [6, 13] have been proposed for this scenario. Nearest neighbor querying has attracted the most attention [9, 18, 20]. The paper builds on these advances, and it considers in detail the approaches for in-route nearest neighbor query proposed by Shekhar and Yoo [20].

Next, the skyline operator was recently introduced into the context of databases [2]. Several skyline algorithms have since appeared [4, 11, 12, 14, 21]. Most algorithms assume that the points to be queried are stored in an R-tree like structure [7]. This paper generalizes the skyline operator and applies it in a new setting; and it does not assume the presence of an index, but rather uses different means of pruning the search space.

Three contributions may be identified. First, the paper advances the idea of in-route movement with associated spatial preferences and consequent kth order skyline queries. Second, techniques for computing such queries for different kinds of in-route movement are presented. Third, a performance evaluation using real point-of-interest and road network data is covered.

Section 2 introduces the data model, the road network representation, and basic algorithms. The next section discusses the movement of in-route users and the corresponding nearest-neighbor based preferences used in queries. Then Section 4 proposes the query algorithms, and Section 5 covers the performance experiments. The last section summarizes and offers research directions.

2 Data Model and Basic Algorithms

2.1 Problem Statement

As discussed, we assume that the users of LBSs are road-network constrained; for example, users may drive by car or may be bus passengers. Next, a number of facilities or so-called points of interest, e.g., gas stations and supermarkets, are located within the road network. We assume that users query the points of interest en route towards

a destination. Specifically, the users issue queries with the purpose of finding a point of interest to visit while moving along a pre-defined route towards a given destination. Having found a point of interest, the user visits this point and then continues towards the destination. Section 3 discuses distance-related preferences.

We term users *query points* and the points of interest *data points*. We proceed to model the problem scenario, including the road network, data points, and the current location, route, and destination associated with query points.

2.2 Data Model

A *road network* is a labeled graph $RN = (V, E)$, where V is a set of vertices and E is a set of edges. A vertex models an intersection or the start or end of a road. An edge e models the road in-between two vertices and is a three-tuple $e = (u, v, w)$, where $u, v \in V$ are vertices and w is the length (or, weight) of e. We assume that an edge may be traversed in both of its directions.

Next, a *data point* dp is a two-tuple $dp = (e, pos_u)$, where e is the edge on which dp is located and pos_u represents the distances from vertex u of e to dp. The distance from v of e to dp is then $w - pos_u$. Note that adding and removing data points does not affect the graph itself, which is important for maintainability in practice. A data point found by an algorithm proposed in this paper is a *candidate point*, $cp = (dp, x_1, \ldots, x_m)$, where the x_i are attributes generated by the particular algorithm. A *query point* qp has the same format as a data point, and pos_u denotes the current distance from vertex u of e to the (moving) point.

Finally, a *route* is given by a sequence of neighboring vertices $\langle r_0, r_1, \ldots, r_l \rangle$, where $r_i \in V, i = 0, \ldots, l$. We assume the current location c of the query point is on the edge between r_0 and r_1, as edges of a route that have already been traversed are of little interest in our context. It is also assumed that the query point cannot make a "u-turn" between its current location c and r_1. The destination associated with a route is the last vertex, r_l. A desired destination that is not represented by a vertex may be handled by introduction of a temporary vertex into the road network, or by running the algorithms twice, once for each neighbor vertex of a destination on an edge, followed by refinement of the results. Figure 1 illustrates the concepts defined above.

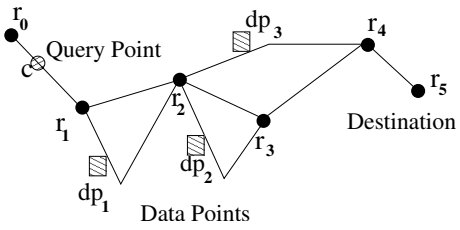


Fig. 1. Data Model Concepts.

2.3 Disk-Based Road Network Representation

A road network is stored as two paginated adjacency lists: one for vertices and one for edges. To obtain locality in operations, vertices are organized in pages according to their Hilbert values.

Each element in the adjacency list of a vertex v corresponds to an adjacent vertex and contains 3 entries: a pointer to the page that contains the adjacent vertex, a pointer to the page in the adjacency list of edges that contains the corresponding edge, and the length of this edge.

The adjacency list for edges records the relations between edges and data points. The adjacency lists for two edges e_1 and e_2 are put in the same page if their vertices are in the same page. Each element in an edge's adjacency list contains information about a data point located on this edge.

2.4 Basic Operations and Algorithms

Skyline Operation. The skyline operation retrieves those points in an argument data set that are not dominated by any other points in the set. More specifically, consider a set of points in l-dimensional space. Point p_1 dominates point p_2 if p_1 is at least as good as p_2 in all dimensions and better than p_2 in at least one dimension [2]. Here, we define "better" as "smaller than." For example, assume we have points $p_1 = (2, 4)$, $p_2 = (3, 5)$, $p_3 = (1, 6)$ in two-dimensional space. Point p_1 dominates p_2 as it is better than p_2 in all dimensions. But p_1 and p_3 do not dominate each other. So p_1 and p_3 are the skyline points in set $\{p_1, p_2, p_3\}$.

We are interested in finding not only those points that are not dominated by any other point, but also the points that are dominated by less than k other points, termed the kth order skyline points. This generalization is similar in spirit to the generalization of the nearest neighbor query to the k nearest neighbors query.

Many existing skyline algorithms assume that the argument points are pre-processed into a data structure such as the R-tree. In contrast, we aim to prune unnecessary search of candidate points before the skyline operation, and we assume instead that the candidate points are organized in a simple main-memory list structure.

For a point set $P = \{p_1, \ldots, p_n\}$ and $p_i = (dp, x_1, \ldots, x_m)$, we denote the kth order skyline operation by $SKYLINE(k, P, \{d_1, \ldots, d_l\})$, where $\{d_1, \ldots, d_l\}$ are the dimensions to be considered. To compute the result, each point is compared to all other points to determine its skyline order. The set of points with skyline order less than k is the result.

Distance Function. We define the distance between two vertices v_i, v_j, denoted by $D(v_i, v_j)$ as the sum of lengths of the edges along the shortest path between these vertices. We use Dijkstra's single-source shortest path algorithm [5] for computing such distances. Since edges may be traversed in both directions, $D(v_i, v_j) = D(v_j, v_i)$.

In addition, for two vertices v_i, v_j along a route R, the road distance from v_i to v_j along the route is denoted by $\mathbf{D}(v_i, v_j, R)$ and is defined as the sum of edge lengths between each two neighboring vertices in the vertex sequence of route R. If v_i is after v_j along the route, the edge lengths are counted as negative so that $\mathbf{D}(v_i, v_j, R) < 0$. Note that $D(v_i, v_j) \leq |\mathbf{D}(v_i, v_j, R)|$. We also use $\mathbf{D}(c, r, R)$ to denote the distance from the query point's current location c to a vertex r along route R. This distance is the sum of the distance from c to r_1 and $\mathbf{D}(r_1, r, R)$.

Spatial Range and Nearest Neighbor Queries. We use $RQ(v, d)$, where v is a vertex and d is a real-valued range, to denote the range query that returns all data points that are within distance d of vertex v. We use a traditional best-first search algorithm [10] extended by the reading of data points from edges for computing this query.

We use $NNQ(k, v)$, where k is a positive integer and v is a vertex, for denoting the set of (up to) k data points that satisfy the condition that no other data points are nearer to vertex v than any point in the argument data set. A number of algorithms exist that compute this query [9, 15, 18, 20], and we use and algorithm that is based on ideas from these. Briefly, it is a traditional best-first search, extended incrementally to read data points from edges and to retrieve the k nearest ones.

We also use an algorithm that, for a vertex v, finds the (up to) k nearest neighbor data points that are also within distance d of v. We denote this query by $RNNQ(k, v, d)$. This algorithm combines the range and nearest neighbor algorithms.

3 Classification of In-Route Nearest Neighbor Queries

We first discuss the possible distance-related preferences for in-route skyline queries. Then we propose a query classification based on the movement of the query point.

3.1 Distance-Related Preferences

Two distance-related preferences are of particular interest in relation to a query point, a data point, and a destination.

Total distance difference: The pre-defined movement is from the current location to the destination along the route. To visit a data point, the query point will change its movement, to go from the current location to the data point and then to the destination. The total distance difference is the larger or smaller distance, compared to that of the pre-defined movement, that the query point must travel to visit the data point and then go to the destination.

Distance to the data point: The distance to the data point is the distance the query point needs to travel to reach the data point.

Optimizing for one preference may adversely affect the other. So if short total distance *as well as* short distance to the data point are of interest, combining these in a skyline query is natural. For brevity, we will in the sequel denote the total distance difference as the "detour," while the distance to the data point is denoted as the "distance."

To find the skyline points, we need to first associate distance and detour values with the data points. Our focus will be on the process of searching for candidate points in the skyline operation. We proceed to classify the possible movements of a query point and consider the distance and detour values for each classification. The resulting classification may be used when pruning the search.

3.2 Classification

Although a query point may move unpredictably, it will always be leaving for a data point at a vertex along its route, and its return to the route can also be characterized by

a vertex along the route. We consider next the general case and then two special cases that are likely to be of interest in specific real-world uses. Recall that without loss of generality, the query point's current location c is between r_0 and r_1 along the route and the destination r_l is the last route vertex. All three cases are shown in Figure 2.

General Case. We first discuss the general case of a query point's movement. A query point issues a query en route towards its destination. The user selects a data point from the result, visits that point, and then proceeds to the destination.

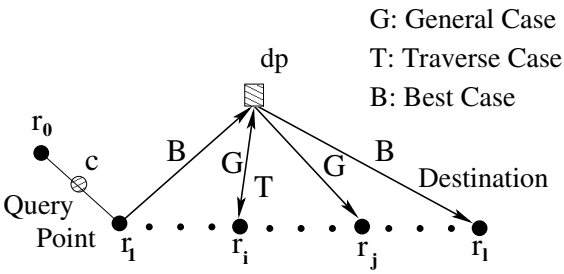


Fig. 2. Classification.

In the general case, the query point leaves its route for data point dp at vertex r_i and returns to the route at r_j—see the sub-paths labeled "G" in Figure 2. Two vertices are special: the "leaving" vertex, r_i, at which the query point leaves the route, and the "returning" vertex, r_j, at which the query point returns to the route. Vertices r_i and r_j can be any vertices in the pre-defined route and r_j may possibly be the destination. In Figure 2, if the route is R, the distance from the query point's current location to data point dp is $\mathbf{D}(c, r_i, R) + D(r_i, dp)$, and the detour is $D(r_i, dp) + D(dp, r_j) - \mathbf{D}(r_i, r_j, R)$.

Traverse Case. In the special case considered here (see label "T" in Figure 2), the query point leaves the route at r_i for data point dp and returns to the route at r_i. In this case, the "leaving" and "returning" vertices are the same vertex. The distance to the data point is the same as the general case, while the detour value is $D(r_i, dp) + D(dp, r_i) = 2D(r_i, dp)$ so that unless the data point is on the route, the detour is bigger than zero. This case applies to users who are faithful to their route, which may be the case for, e.g., tourists following a scenic route.

Best Case. In this case (see label "B" in Figure 2), the query point leaves the route for data point dp at r_1 and then goes directly towards the destination in the route. The distance and detour for this case are obtained by replacing i, j by 1 and l in the formulas given for the general case. Observe that in this case, the route carries little significance to the user.

In addition to the traverse and best cases, other special types of movement are possible. We simply categorize these as belonging to the general case. So in the next section, we provide algorithms for the traverse case and the best case; and based on the observations from these algorithms, we present an algorithm for the general case of the in-route nearest neighbor skyline query.

4 Algorithms for In-Route Skyline Queries

One basic approach to compute the $SKYLINE$ query is to scan all data points, determining for each point whether it is a skyline point. Assuming a large number of data points, this is too costly. A strategy should be employed that prunes unnecessary search.

4.1 Algorithm for the Traverse Case

In the traverse case, the query point leaves the route and returns at the same vertex. Shekhar and Yoo [20] have previously considered this case, and our algorithm for this case is similar to theirs, the main differences being that we use road distance and any k, while they use Euclidean distance and assume $k = 1$.

Given an order k, a query point's current location c, and a route $R = \langle r_0, r_1, \ldots, r_l \rangle$, the algorithm for the traverse case, $TraverseSQ$, is seen below. Two auxiliary queues, T and P, are used to store the result data points of nearest neighbor queries and the candidate points for the skyline query.

```
(1)  procedure TraverseSQ(k, c, R)
(2)      P ← ∅
(3)      T ← NNQ(k, r_1)
(4)      for each t ∈ T
(5)          dis ← D(c, r_1, R) + D(r_1, t)
(6)          det ← 2D(r_1, t)
(7)          P ← P ∪ {(t, dis, det)}
(8)      d ← distance from r_1 to its kth neighbor
(9)      for each r_i, i = 2, ..., l
(10)         T ← RNNQ(k, r_i, d)
(11)         if T not empty
(12)             for each t ∈ T
(13)                 dis ← D(c, r_i, R) + D(r_i, t)
(14)                 det ← 2D(r_i, t)
(15)                 P ← P ∪ {(t, dis, det)}
(16)             d_1 ← distance from r_i to its kth neighbor
(17)             if d_1 < d
(18)                 d ← d_1
(19)     return (SKYLINE(k, P, {dis, det}))
```

The algorithm is based on the following observations (Figure 3). Assume r_1 and r_2 are route vertices. Let data point dp_1 be the kth nearest neighbor of r_1 and let data point dp_2 be the $k+1$st nearest neighbor of r_1. Then $D(r_1, dp_2) > D(r_1, dp_1)$. Since the query point's current location c is before r_1, it is obvious that the distance from the query point's current location to dp_1, via r_1, is smaller than the distance to dp_2. Also in this case, the detour of dp_1 is $2D(r_1, dp_1)$ and the detour of dp_2 is $2D(r_1, dp_2)$. So dp_1 dominates dp_2. Since dp_1 is the kth nearest neighbor of r_1, dp_2 is at most in the $k+1$st order skyline. As we perform nearest neighbor search incrementally, there is no need to continue after the kth nearest neighbor of r_1 is found.

Let $d_1 = D(r_1, dp_1)$. A k range nearest neighbor query is issued at r_2 with range d_1 (dashed polygon in Figure 3). If the kth nearest neighbor to r_2, e.g., dp_3 is found, it will be collected as a candidate point. And its distance to r_2, i.e., $d_2 = D(r_2, dp_3)$, will be compared to d_1 to possibly obtain a smaller search range at the next route vertex.

For any data point outside the range of d_1, e.g., dp_4, it is obvious that the detour of dp_4 is bigger than the detour of dp_1. The distance from the user's current location to dp_1 is $\mathbf{D}(c, r_1, R) + d_1$, and the distance to dp_4 is $\mathbf{D}(c, r_1, R) + \mathbf{D}(r_1, r_2, R) + D(r_2, dp_4)$. So any data point like dp_4 will be dominated by dp_1, which means that such data points will at most be in the $k + 1$st order skyline. So there is no need to expand the search beyond d_1 at r_2.

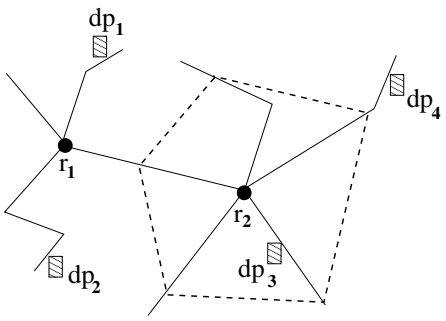


Fig. 3. Observations for the Traverse Case.

Note that if the query point can leave for the data point at only a few specified vertices along the route, algorithm $TraverseSQ$ can easily be adapted to handle this special case more efficiently.

The complexity of the algorithm is dominated by the two basic operations, NNQ in line 3 and $RNNQ$ in line 10, the inner loop in lines 12–15, and the skyline operation. The two primitive operations can be seen as network expansion processes that use Dijkstra's single source shortest path algorithm to search for data points along the edges. Let $|E|$ be the number of edges and $|V|$ the number of vertices in the road network. The complexity of this operation is then $O(|E| + |V|log|V|)$, if an F-heap is employed [5]. Taking into account that (up to) k data points are to be retrieved, each of the two operations has a complexity of $O(|E| + |V|log|V| + k)$. We assume that the data points on an edge can be accessed in the order of their distances from the beginning (and ending) vertex of the edge. The inner loop runs at most k times at each route vertex. The complexity is $O(k|R|)$, where $|R|$ is the number of route vertices. The skyline operation, as described in Section 2.4, compares each argument point with all other argument points to determine its skyline order. Since up to k data points may be found at each route vertex, the complexity of the skyline operation is then $O(k^2|R|^2)$. Thus, the complexity of the $TraverseSQ$ algorithm can be given as $O(|R|(|E|+|V|log|V|+k^2|R|))$. Section 5 provides a much more detailed empirical study of the performance of the algorithm.

4.2 Algorithm for the Best Case

The skyline algorithm for the best case only takes the current location c, vertex r_1 and destination r_l on the route into consideration. For any skyline point dp found by this algorithm, the movement of the query point is from c to dp via vertex r_1 and then from dp to r_l.

The algorithm is based on observations illustrated in Figure 4 and explained next. Let r_i and r_j be vertices on route R and let the query point's movement be from r_i to the data point and then to r_j. Then, to find all the candidate points, two range queries are issued at r_i and r_j. Let dp_1 be the kth nearest neighbor of r_i and let $d_1 = 2D(r_i, dp_1) +$

$\mathbf{D}(r_i, r_j, R)$ and $d_2 = \mathbf{D}(r_i, r_j, R) + D(r_i, dp_1)$. Then d_1 is the distance from r_i to the kth neighbor data point dp_1 and the back, plus the distance from r_i to r_j. So, that all the k nearest neighbor data points from route vertex r_i should be found within a range around r_i of size d_1. Next, d_2 is the distance from dp_1 to r_i plus the distance from r_i to r_j along the route. Consideration of a range of size d_2 around vertex r_j is sufficient to find all the k nearest neighbor data points of r_i. We proceed to discuss this observation in detail.

With d_1 and d_2, the two range queries are then $RQ(r_i, d_1, RN)$ (dotted polygon in Figure 4) and $RQ(r_j, d_2, RN)$ (dashed polygon in Figure 4). Let all the data points in the road network be in set S, let the data points found by the range query using r_i be in set S_i, and let the data points found by range query using r_j be in set S_j.

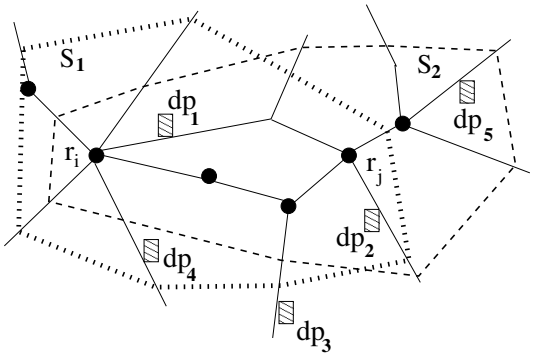


Fig. 4. Observations for the Best Case.

It is clear that dp_1 belongs to $S_i \cap S_j$: $dp_1 \in S_i$ since $D(r_i, dp_1) \leq d_1$; and $D(dp_1, r_j) \leq D(dp_1, r_i) + \mathbf{D}(r_i, r_j, R)$, so $dp_1 \in S_j$. Also the detour of dp_1 is less than or equal to $2D(r_i, dp_1)$ since the length of the shortest path from r_i via dp_1 to r_j is no longer than $2D(r_i, dp_1) + \mathbf{D}(r_i, r_j, R)$. It should be noted that $dp_2 \in S_j$, but dp_2 may not be in $S_i \cap S_j$.

We proceed to explain why data points in $S_i \cap S_j$ dominate other data points. Without loss of generality, we compare dp_1 with dp_3, dp_4, and dp_5 from Figure 4. It can be observed that dp_3 is inside neither S_i nor S_j. So $D(r_i, dp_3) > D(r_i, dp_1)$ and $D(dp_3, r_j) > D(dp_1, r_j)$. So the distance from r_i to dp_1 is smaller than that to dp_3. Also, since the whole road length from r_i via dp_1 to r_j is smaller than from r_1 via dp_3 to r_j, the detour of dp_1 is smaller than that of dp_3. So any data point such as dp_3 will be dominated by dp_1.

For dp_4, since dp_1 is the nearest neighbor to r_i, $D(r_i, dp_4) \geq D(r_i, dp_1)$, and as dp_4 does not belong to S_j, $D(dp_4, r_j) > D(dp_1, r_j)$. So dp_4 is also dominated by dp_1.

For dp_5, we have $D(r_i, dp_5) > 2D(r_i, dp_1) + \mathbf{D}(r_i, r_j, R)$, so the distance from r_i to dp_5 is bigger than the distance to dp_1. And the detour from r_i to r_j via dp_5 is $D(r_i, dp_5) + D(dp_5, r_j) - \mathbf{D}(r_i, r_j, R) > 2D(r_i, dp_1)$. As the detour of dp_1 is less than or equal to $2D(r_i, dp_1)$, dp_5 is dominated by dp_1.

Since dp_1 is the kth nearest neighbor of r_i, all the other $k-1$st nearest neighbors whose distances to r_i are smaller than that of dp_1 are also in $S_i \cap S_j$. So there are at least k data points in $S_i \cap S_j$. And based on the discussion above, data points not in $S_i \cap S_j$ are dominated by the points inside this region, making them at most $k+1$st order skyline points. Thus, there is no need to check data points not in $S_i \cap S_j$.

The algorithm that implements the above-described search procedure is given next. It takes an order k, a query point's current location c, two route vertices r_i and r_j, and

a route R as arguments. Two auxiliary queues, T_1 and T_2, store the data points found in the two range queries at r_i and r_j. Queue P stores the result set of candidate points.

(1) **procedure** $DNNQ(k, c, r_i, r_j, R)$
(2) $T_1, T_2 \leftarrow \emptyset$; $P \leftarrow \emptyset$
(3) $d \leftarrow$ distance from r_i to its kth neighbor
// compute by this by expansion from r_1 and pause when the kth neighbor is found
(4) $d_1 \leftarrow 2d + \mathbf{D}(r_i, r_j, R)$
(5) $d_2 \leftarrow \mathbf{D}(r_i, r_j, R) + d$
(6) $T_1 \leftarrow RQ(r_i, d_1)$ // continue the paused expansion
(7) $T_2 \leftarrow RQ(r_j, d_2)$
(8) **for each** $t \in T_1 \cap T_2$
(9) $dis \leftarrow \mathbf{D}(c, r_i, R) + D(r_i, t)$
(10) $det \leftarrow D(r_i, t) + D(t, r_j) - \mathbf{D}(r_i, r_j, R)$
(11) $P \leftarrow P \cup \{(t, dis, det)\}$
(12) **return** P

With the input arguments k, c, and route $R = \langle r_0, r_1, \ldots, r_l \rangle$, the skyline query algorithm for the best case, $BestSQ$, follows.

(1) **procedure** $BestSQ(k, c, R)$
(2) $P \leftarrow DNNQ(k, c, r_1, r_l, R)$
(3) $P \leftarrow SKYLINE(k, P, \{dis, det\})$
(4) **return** P

Algorithm $DNNQ$ suggests that to find candidate points for any type of movement, one needs to perform two range queries, at the "leaving" and "returning" vertices, and then collect data points in the intersection of the two ranges. We proceed to present a general skyline algorithm that is applicable to any type of movement of the query point.

The complexity of the $BestSQ$ algorithm is dominated by the two range queries in lines 6 and 7 and the iteration in lines 8–11 in algorithm $DNNQ$, as well as the skyline operation. The range queries can be treated as network expansion processes. In the worst case, the whole network and all data points DP are read, yielding a complexity of $O(|E| + |V|log|V| + |DP|)$. The iteration has complexity at most $O(|DP|^2)$. So, taking also into account the complexity of the skyline operation, as described in the previous section, the complexity of algorithm $BestSQ$ is $O(|E| + |V|log|V| + |DP|^2)$.

4.3 Algorithm for the General Case

While the two special cases may be prevalent, other cases exist. We thus provide an algorithm that works independently of the kind of movement of the user. The algorithm is based on the observation from $DNNQ$ that since there always needs to be range queries at all the route vertices, we can issue range queries at each route vertex with the biggest range once and for all and then check the data points in the intersections of each pair of ranges.

Two auxiliary structures are used in the algorithm. First, a set of queues $T = \{T_1, \ldots, T_l\}$ store result data points found from the range queries at route vertices. To

retrieve the data points in T_i within a distance range d, an auxiliary function $Retr(T_i, d)$ is used. Next, a float array D stores the distance from each route vertex to its kth nearest neighbor.

The general in-route nearest neighbor skyline algorithm, $GeneralSQ$, uses the same arguments as does $TraverseSQ$ and is given below.

```
(1) procedure GeneralSQ(k, c, R)
(2)     P ← ∅; T_i ← ∅; D_i ← 0, (i = 1, ..., l)
(3)     for each r_i ∈ R
(4)         d ← maximum range for r_i
                // computed by comparing the range size for each pair of route vertices
(5)         T_i ← RQ(r_i, d)
(6)         D_i ← distance from r_i to its kth neighbor
(7)     for each pair r_i, r_j ∈ ⟨r_1, ..., r_l⟩, i ≠ l
(8)         d_1 ← 2D_i + D(r_i, r_j, R)
(9)         d_2 ← D(r_i, r_j, R) + D_i
(10)        for each t ∈ {Retr(T_i, d_1) ∩ Retr(T_j, d_2)}
(11)            dis ← D(c, r_i, R) + D(r_i, t)
(12)            det ← D(r_i, t) + D(t, r_j) − D(r_i, r_j, R)
(13)            P ← P ∪ {(t, dis, det)}
(14)    P ← SKYLINE(k, P, {dis, det})
(15)    return P
```

The algorithm first issues range queries at each route vertex with the biggest range to obtain all possible data points. Then, for each pair of route vertices r_i and r_j, the algorithm finds candidate data points assuming that the query point's movement is from r_i to the data point and then to r_j. The results are then filtered by the $SKYLINE$ algorithm. Finally, all skyline points are collected. It is possible that one data point is collected more than once because of different kinds of movement.

Considering the algorithm's complexity, we observe that the RQ operation in line 5 is issued for each route vertex. This yields a complexity of $O(|R|(|E| + |V|log|V| + |DP|))$, as discussed earlier. The iteration in line 7 executes $|R|(|R| - 1)/2$ times. At each iteration, the nested iteration starting in line 10 checks all data points found for a pair of route vertices, which is all data points in the worst case. This yields a complexity of the entire iteration of $O(|DP|^2|R|^2)$. The skyline operation has complexity $O(|DP|^2|R|^4)$. Thus, the complexity of algorithm $GeneralSQ$ is $O(|R|(|E| + |V|log|V|) + |R|^4|DP|^2)$.

It can be seen from the complexity analysis that if there are many route vertices, it will be relatively costly to gather the candidate points for each intersection of ranges, since there will be many pairs of vertices. We observe that $GeneralSQ$ may work well if there are only a few possible "leaving" and "returning" vertices along the route. This corresponds to a scenario where a traveler would like to leave the route for a point of interest and return at some "familiar" or "well-known" locations along the route.

We proceed to offer a more detailed study of the performance of this and the previous two algorithms covered in this section.

5 Experimental Evaluation

The experiments described here use a real-world representation of the road network covering the municipality of Aalborg, Denmark. The road network contains $11,300$ vertices, $13,375$ bi-directional edges, and 279 data points that can be accessed via the network. So the data density, the number of data points over the number of edges of a road network, is 2%.

We define the size of a route as the number of vertices it contains. The page size is set to 4k bytes, and an LRU buffer is employed for simulation. A total of 136 pages contain adjacency lists of vertices, and 3 pages contain adjacency lists of edge. For the $GeneralSQ$ algorithm, we use 10% of the number of route vertices as the possible "leaving" and "returning" vertices and assume that the "leaving" vertices are before the "returning" vertices.

To evaluate the effect of route size, skyline order (k), and buffer size, three experiments are conducted that measure query performance in terms of CPU time, I/O cost, and number of candidate points. The CPU-time is the actual running time for these algorithms. The I/O cost is the amount of pages read into the LRU buffer. The number of candidate points is the count of the candidate points that are found in these algorithms before the skyline operation. Since an in-memory skyline algorithm is used in the algorithms, the skyline computation is only evaluated in terms of CPU-time.

In the experiments, random routes are generated. The current location and destination are assumed to be the first and last vertices in the route. For each algorithm, we execute a workload of 100 queries and report the average performance. The experiments were performed on a Pentium IV 1.3 GHZ processor with 512 MB of main memory and running Windows 2000. The C++ programming language was used. To determine the variations in results due to external factors beyond our control (e.g., operation system tasks), we executed the same workloads multiple time. The results obtained exhibit only insignificant variations across repeated executions.

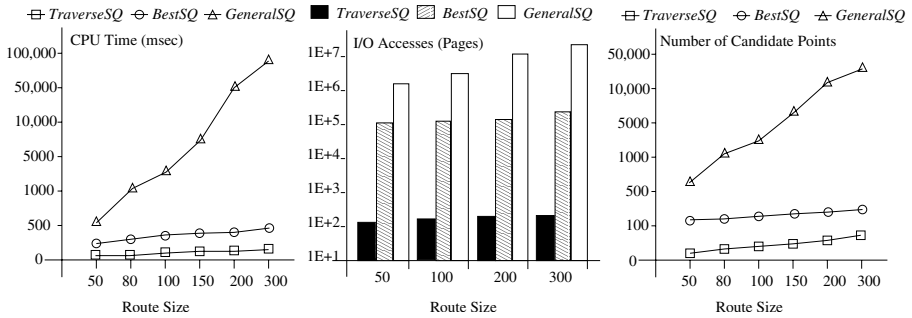
It should also be noted that the I/O costs for all the algorithms are measured using simulation. These costs are thus independent of the hardware and operating system used. Also, for the algorithms we have discussed, the I/O cost is more significant than the CPU time.

5.1 Experiment on the Effect of Route Length

In this experiment, the skyline order k is set to 5, and the buffer size is 10% of the road network size. The route size is varied from 50 to 300 to check the CPU time and I/O cost of the three algorithms. The results are shown in Figure 5.

It is clear that the $TraverseSQ$ algorithm has the best performance and that $BestSQ$ is in turn better than $GeneralSQ$. We proceed to discuss the findings for each algorithm.

It can be observed that the cost of $TraverseSQ$ grows slightly with an increase in route size. Since the data density of the road network is 2%, when the number of route vertices is small, the data points are far from the route vertices. CPU and I/O costs grow slightly as the route size increases because a search process is required for each route vertex. But when the amount of route vertices grows, chances of finding a k-nearest neighbor close to a route vertex are higher, so that the search range for subsequent



(a) CPU Time (b) I/O Accesses (c) Candidate Points

Fig. 5. Performance of Algorithms Versus Route Size.

vertices is smaller. Also, it can be seen from Figure 5(c) that the numbers of candidate points are so small that the skyline operation has no impact on the overall performance of $TraverseSQ$.

The overall cost of $BestSQ$ grows slightly with an increase in route size. This is because the sizes of the two range queries in the $BestSQ$ algorithm depend both on the route size and the distance from the first route vertex to its kth nearest neighbor data point. When the route size is small, the distance from the first route vertex to its k-nearest data point has the biggest effect on the cost. As the route size grows, its effect on the performance of the algorithm increases little by little. Note that in this case, the maximum number of candidate points can be found is 279. Because of this, the skyline operation has only minor impact on the overall performance.

It can be seen from Figure 5 that the cost of $GeneralSQ$ increase drastically with the growth of route length. When the route size grows, more range queries are issued. The size of these range queries also grows since the route is longer. Also, since the number of candidate points exceeds $5,000$ when the route size is 150, secondary-memory skyline processing is required.

5.2 Experiment on the Effect of k

In this experiment, the route size is set to 100, and the buffer is 10% of the size of the road network. The skyline order k is varied from 1 to 50. The results are shown in Figure 6. The general performances of the three algorithms follow the same trends as in the experiment on the effect of route size. However, when $k = 50$, $TraverseSQ$ and $BestSQ$ are quite close. This is because when $k = 50$, all the data points have been scanned by both algorithms. We proceed to discuss each individual algorithm.

It can be observed that the cost of the $TraverseSQ$ algorithm increases with an increase in k. When k is increased from 10 to 50, the increase is quite significant. This is because the performance of $TraverseSQ$ depends mostly on the k nearest neighbor search at the first several vertices in the route, so that when the k is increased, the search range at these beginning vertices is enlarged. When $k = 50$, scanning of all the data points is unavoidable. Note that since the candidate points found is always less than 100, the skyline operation does not have a substantial impact on the performance.

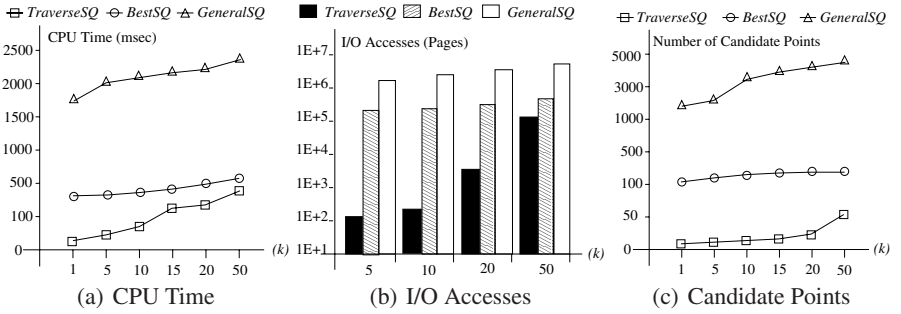


Fig. 6. Performance of Algorithms Versus k.

As k is increased, the cost of the $BestSQ$ algorithm increases slightly. This is because when k increases, the cost of searching for the kth nearest neighbor increases. But the cost of the two range queries does not increase much since the route size is the major factor in determining the range size. Also note that since the maximum number of candidate points for $BestSQ$ is 279, when k is bigger than 15, the amount of candidate points comes to be this maximum value.

The cost of the $GeneralSQ$ algorithm also increases with an increase in k. Since the route size is fixed at 100 in this experiment, the number of range queries is constant. The bigger k is, the larger the size of these range queries. The slight increase in cost as k increases indicates that route size has the biggest effect on performance of $GeneralSQ$. Also, note that as the number of candidate points is always below $5,000$, the skyline operation does not have great impact on the overall cost.

5.3 Experiment on the Effect of Buffer Size

In this experiment, k is set to 5 and the route size is set to 100. The buffer size is varied from 10% to 50%. Since the buffer size has little influence on the CPU time and number of candidate points, the experiment only considers I/O cost. It can be seen in Figure 7 that the number pages accessed decreases greatly with as the buffer size increases. This is because when the buffer grows, more road network data reside in the buffer, thus decreasing the chances of reading data from outside the buffer. For the $TraverseSQ$ algorithm, when the buffer size increases to 20% of the road network, the I/O access do not change since the size of the pages accessed by the $TraverseSQ$ algorithm is smaller than the size of the buffer.

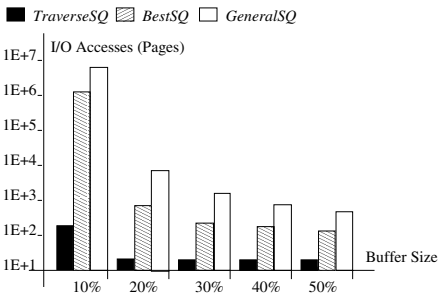


Fig. 7. I/O Accesses Versus Buffer Size.

6 Summary and Future Work

This paper introduces a novel location-based query in a road network based usage scenario, namely the in-route nearest neighbor skyline query. Two interesting special case of the general usage scenario considered are identified. The paper then proceeds to reuse and extend existing query processing techniques to apply to the new type of query. Specifically, it provides algorithms for the two special cases as well as the general case. Finally, the paper reports on experimental performance studies with the three algorithms. Real point of interest and road network data are used.

Providing efficient support for in-route location-based queries for moving users is an interesting and important topic in traveler information system. Since movement is normally restricted to a transportation network, traditional spatial-temporal queries, e.g., nearest neighbor, spatial range, closest pair, distance join, need to be re-considered because Euclidean distance becomes inappropriate. The algorithms proposed in this paper may be extended to make use of indexing and pre-computation techniques. Several research directions may be identified, including the following two:

- It is required in this paper that the pre-defined route is a sequence of neighboring vertices, while in the real world a pre-defined route may also consist simply of several specified locations in the road network. Algorithms in the context of such routes is an interesting direction for future work.
- In this paper, all the data points are organized into one group, and the in-route query finds k candidate points from this group. More complex query preferences may occur naturally in real applications. For example, a moving user may want to visit a bank as well as a supermarket before arriving at the destination. Processing of location-based queries under such complex settings is also an interesting direction for future work.

Acknowledgments

This work was supported in part by grant 216 from the Danish National Center for IT Research. In addition to his primary affiliation, the second author is an adjunct professor at Agder University College, Norway.

References

1. R. Benetis, C. S. Jensen, G. Karciauskas, S. Šaltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proc. IDEAS,* pp. 44–53, 2002.
2. S. Borzsonyi, D. Kossmann, K. Stocker. The Skyline Operator. In *Proc. ICDE,* pp. 421–430, 2001.
3. A. Brilingaitė, C. S. Jensen, N. Zokaitė. Enabling Routes as Context in Mobile Services. In *Proc. ACM GIS*, pp. 127–136, 2004.
4. J. Chomicki, P. Godfrey, J. Gryz, D. Liang. Skyline with Presorting. In *Proc. ICDE,* pp. 717–816, 2003.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms: Second Edition. The MIT Press, 2001.

6. E. Frentzos. Indexing Objects Moving on Fixed Networks. In *Proc. SSTD,* pp. 289–305, 2003.
7. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. SIGMOD,* pp. 47–57, 1984.
8. C. Hage, C. S. Jensen, T. B. Pedersen, L. Speičys, and I. Timko. Integrated Data Management for Mobile Services in the Real World. In *Proc. VLDB,* pp. 1019–1030, 2003.
9. C. S. Jensen, J. Kolář, T. B. Pedersen, I. Timko. Nearest Neighbor Queries in Road Networks. In *Proc. ACMGIS,* pp. 1–8, 2003.
10. D. E. Knuth. Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley Pub Co., 1998.
11. D. Kossmann, F. Ramsak, S. Rost. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. In *Proc. VLDB,* pp. 275–286, 2002.
12. H. X. Lu,Y. Luo, X. Lin. An Optimal Divide-Conquer Algorithm for 2D Skyline Queries. In *Proc. ADBIS,* pp. 46-60, 2003.
13. D. Pfoser, C. S. Jensen. Indexing of Network Constrained Moving Objects. In *Proc. ACMGIS,* pp. 25–32, 2003.
14. D. Papadias, Y. Tao, G. Fu, B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *Proc. SIGMOD Conf.,* pp. 467–478, 2003.
15. D. Papadias, J. Zhang, N. Mamoulis, Y. Tao. Query Processing in Spatial Network Databases. In *Proc. VLDB,* pp. 802–813, 2003.
16. N. Roussopoulos, S. Kelley, F. Vincent. Nearest Neighbor Queries. In *Proc. SIGMOD,* pp. 71–79, 1995.
17. L. Speičys, C. S. Jensen, A. Kligys. Computational Data Modeling for Network Constrained Moving Objects. In *Proc. ACMGIS,* pp. 118–125, 2003.
18. C. Shahabi, M. R. Kolahdouzan, M. Sharifzadeh. A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases. In *GeoInformatica* 7(3), pp. 255–273, 2003.
19. Z. Song, N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proc. SSTD,* pp. 79–96, 2001.
20. S. Shekhar, J. S. Yoo. Processing In-Route Nearest Neighbor Queries: A Comparison of Alternative Approaches. In *Proc. ACMGIS,* pp. 9–16, 2003.
21. K. L. Tan, P. K. Eng, B. C. Ooi. Efficient Progressive Skyline Computation. In *Proc. VLDB,* pp. 301–310, 2001.
22. Y. Tao, D. Papadias, Q. Shen. Continuous Nearest Neighbor Search. In *Proc. VLDB,* pp. 287–298, 2002.