Monitoring Path Nearest Neighbor in Road Networks

Zaiben Chen[†], Heng Tao Shen[†], Xiaofang Zhou[†], Jeffrey Xu Yu[‡]
[†] School of Information Technology & Electrical Engineering
The University of Queensland, QLD 4072 Australia
[‡]The Chinese University of Hong Kong, Hong Kong, China
{zaiben, shenht, zxf}@itee.uq.edu.au, yu@se.cuhk.edu.hk

ABSTRACT

This paper addresses the problem of monitoring the k nearest neighbors to a dynamically changing path in road networks. Given a destination where a user is going to, this new query returns the k-NN with respect to the shortest path connecting the destination and the user's current location, and thus provides a list of nearest candidates for reference by considering the whole coming journey. We name this query the k-Path Nearest Neighbor query (k-PNN). As the user is moving and may not always follow the shortest path, the query path keeps changing. The challenge of monitoring the k-PNN for an arbitrarily moving user is to dynamically determine the update locations and then refresh the k-PNN efficiently. We propose a three-phase *Best-first Network Expansion* (BNE) algorithm for monitoring the k-PNN and the corresponding shortest path. In the *searching phase*, the BNE finds the shortest path to the destination, during which a candidate set that guarantees to include the k-PNN is generated at the same time. Then in the *verification phase*, a heuristic algorithm runs for examining candidates' exact distances to the query path, and it achieves significant reduction in the number of visited nodes. The *monitoring phase* deals with computing update locations as well as refreshing the k-PNN in different user movements. Since determining the network distance is a costly process, an expansion tree and the candidate set are carefully maintained by the BNE algorithm, which can provide efficient update on the shortest path and the k-PNN results. Finally, we conduct extensive experiments on real road networks and show that our methods achieve satisfactory performance.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-551-2/09/06 ...$5.00.

Keywords

Path Nearest Neighbor, Road Networks, Spatial Databases

1. INTRODUCTION

Nearest Neighbor query is one of the fundamental issues in spatial database research area. It is designed to find the closest object p to a specified query point q, given a set of objects and a distance metric. This problem is well studied in the literature, and its variants include k-Nearest Neighbor search [6, 17], Continuous Nearest Neighbor search [1, 10, 21], Aggregate Nearest Neighbor queries [14, 22], etc.

While all the queries mentioned above concern only the locally optimized results, in this paper, we investigate the problem of *Path Nearest Neighbor* (PNN) query, which retrieves the nearest neighbor with respect to the whole query path. Here, 'locally optimized results' means the nearest neighbors with respect to the current query location. However, sometimes a user is moving and may want to know the best choice by considering the whole path to be traveling on, thus a globally optimal choice for the nearest neighbor to a given path is required, and that is the motivation of this work. As exemplified in Figure 1, assume that we are traveling from s to t along a path $P = \{s, n_2, n_3, t\}$ and we hope to find the nearest gas station for refueling. If we use conventional Nearest Neighbor query, gas station A is returned at the beginning. However, A is not the best choice because there is another gas station B not far away which is much closer to the path we are traveling on. So PNN query suits the applications where a user wants to consume a service when traveling towards a given destination. For such applications, neither the current nearest neighbor nor the nearest neighbor at any particular point is the best for the user; instead, the user wants to know the nearest neighbor relative to the route he/she will travel (B in this example).

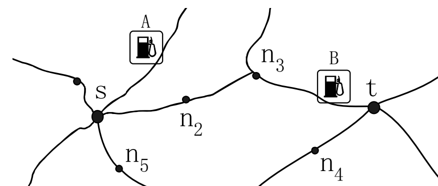

Figure 1: an example

A similar issue called In-Route Nearest Neighbor (IRNN) query is first proposed by Shekhar et al. in [20] to search a facility instance (e.g. gas station) with the minimum detour

distance from the query route on the way to the destination. Still considering Figure 1, the detour distance of A from P is greater than that of B, so obviously IRNN(P) would return B to the user. The intuition behind is that users (e.g. commuters) prefer to follow the route they are familiar with, thus they would like to choose the gas station with the smallest deviation from the route. After refueling, they will return to the previous route and continue the journey. However a drawback of IRNN is that the user has to input exactly the whole query path in advance, which is identified by all intersections along the path, while a user's driving path often cannot be precisely pre-decided. Imagine that a user is driving from Washington to New York, which is a long journey. It is impractical for a user to input hundreds of intersections before successfully making a query.

Therefore, we propose the *Path Nearest Neighbor* query, requiring users to input only the destination as well as the current location rather than the whole specified path. For each PNN query, we construct a shortest path connecting the destination and the current location and then search for the nearest facility instance to the shortest path (i.e. the facility instance with the minimum detour distance). Since a moving user may not follow the shortest path and the driving route might change over time in the coming journey, we provide continuous monitoring of the k-PNN, which always gives the user the best candidates for consideration. This raises the issue of how to dynamically query the nearest neighbors to a changing shortest path efficiently. To provide efficient monitoring of the k-PNN, we propose in this paper a *Best-first Network Expansion* (BNE) method. Specifically, the BNE consists of three main phases, including *searching phase*, *verification phase* and *monitoring phase*.

In the searching phase, the BNE algorithm incorporates a bi-directional search method for establishing the shortest path, which conducts two independent network expansions from the starting location and the destination separately, and when the two expansions meet, the shortest path is determined. The novelty of the searching phase is that, we can also derive all encountered nodes' lower bounds and upper bounds of minimum detour distance during the bi-directional search, which are further utilized in determining a candidate set for the k-PNN and examining candidates' exact detour distances. As the searching for the shortest path is inevitable for the k-PNN query (if not consider pre-computation for distance browsing), the BNE algorithm is designed to retrieve as much information as possible during the searching process, and improve the performance of monitoring by using this information.

With a list of potential candidates that guarantee to include the k-PNN results returned from the searching phase, the verification phase processes these candidates in the order of their lower bounds. Here, a heuristic verification function for examining candidates' exact minimum detour distances to the query path is devised. The heuristic function searches the minimum detour path from a candidate towards the query path directionally, instead of simply conducting a Dijkstra's network expansion. By doing so, the area of searching is reduced greatly especially when the candidate is not close to the query path.

In the monitoring phase, the main task is to figure out where an update for the k-PNN is needed, which could be an update of the order, or a re-calculation of the k-PNN. We discuss these two cases in the situation when the user follows the shortest path or deviates from the shortest path respectively. To facilitate the k-PNN updates, the BNE carefully maintains an expansion tree rooted at the destination, which stores the shortest paths (from destination) to the surrounding nodes. This expansion tree is firstly recorded during the bi-directional search in the searching phase, and it enlarges or shrinks accordingly while the user's current location is changing. Besides, the candidate set and candidates' lower bounds/upper bounds acquired previously are also updated gradually in the monitoring phase, which are utilized to accelerate the update algorithms.

To sum up, we make the following main contributions:

- We define a new type of query for searching the k nearest neighbors to a changing shortest path. It provides new features for advanced spatial-temporal information systems, and may benefit users by reporting best candidates from the global view.
- We devise the BNE algorithm which efficiently monitors the k-PNN while the user is moving arbitrarily. An expansion tree and the candidate set are utilized with lower and upper bounds on minimum detour distance for fast k-PNN update.
- We also propose the methods for determining the update locations which invoke potential updates on the k-PNN results in different user movements, as well as the algorithms for efficiently updating the k-PNN results.
- We conduct extensive experiments on real datasets to study the performance of the proposed approaches.

The remainder of the paper is organized as follows. In Section 2 we discuss the related work. In Section 3, a formal definition of the problem is given. The *searching phase* and the *verification phase* of the BNE algorithm are presented in Section 4, and the *monitoring phase* is introduced in Section 5. Finally we show our experiment results in Section 6 and draw a conclusion in Section 7.

2. RELATED WORK

Spatial queries in advanced traveler information system continue to proliferate in recent years. Nearest Neighbor (NN) query is considered as an important issue in such kind of applications. This query aims to retrieve the closest neighbor to a query point from a set of given objects. In [17] and [6] a depth-first and a best-first tree traversal approaches are proposed respectively for NN query in Euclidean space and they employ a branch-and-bound strategy.

The Nearest Neighbor query is also extended to a road network scenario by using network distance as the distance metric. Papadias et al. present in [15] the Incremental Euclidean Restriction (IER) and Incremental Network Expansion (INE) algorithms for retrieving k-NN according to network distance. IER uses the Euclidean distance as a lower bound for pruning during the search, and INE performs a network expansion similar to the Dijkstra's algorithm [3]. Jensen et al. also propose in [7] a general spatial-temporal framework for NN queries in a road network which is represented by a graph. In [19], a graph embedding technique is proposed to transform a road network to a high-dimensional Euclidean space and then the approximate k-NN can be

found. Pre-computation based methods for k-NN queries are also studied in [8] and [18], in which Voronoi diagrams and Shortest Path Quadtrees are utilized separately.

Many variants of Nearest Neighbor search are studied as well, like Aggregate k-NN monitoring [16], Trip Planning Queries [9] and Continuous Nearest Neighbor queries (CNN) [1, 10, 13, 21]. CNN queries report the k-NN results continuously while the user is moving along a path. The main challenge of this type of queries is to find the split points on the query path where an update of the k-NN is required, and thus to avoid unnecessary k-NN re-calculations. However, a limitation of CNN queries is that the query path has to be given in advance and it can not change during the user's movement. Therefore, in [11], Mouratidis et al. investigate the Continuous Nearest Neighbor monitoring problem in a road network, in which the query point moves freely and the data objects' positions are also changing dynamically. The basic idea of [11] is to carefully maintain a spanning tree originated from the query point and to grow or discard branches of the spanning tree according to the data objects and query point's movements. To some extent, the motivation of our k-PNN monitoring problem is similar to that of the CNN monitoring. However, we aim to provide monitoring of the k-NN to a dynamically changing path rather than a moving query point, and we assume all data objects (e.g. restaurants, gas stations) keep stationary.

In-Route Nearest Neighbor Queries (IRNN) in [20] is designed for users that drive along a fixed path routinely. As this kind of drivers would like to follow their preferred routes, IRNN queries are proposed for finding nearest neighbor with the minimum detour distance from the fixed route, because they make the assumption that a commuter will return to the route after going to the nearest facility (e.g. gas station) and will continue the journey along the previous route. Our problem is an extension of the IRNN query, by monitoring the k nearest neighbors to a continuously changing shortest path, and the user only needs to input the destination rather than exactly the whole query path.

3. PROBLEM DEFINITION

In this paper, a road network is modeled as a weighted undirected graph $G(V, E)$, in which V consists of all vertices (nodes) of the network, and E is the set of all edges. We assume that all facility instances (data objects) lie on the road. If a data object is not located at a road intersection, we treat the data object as a node and further divide the edge it lies on into two edges. So V is a node set comprised of all intersections and data objects and E contains all the edges between them. Each edge is associated with a non-negative weight representing the time cost of traveling or simply the road distance between the two neighboring nodes.

We define the *network distance* $D_n(n_1, n_2)$ between two nodes n_1 and n_2 as the length of the *shortest path* $SP(n_1, n_2)$ connecting n_1 and n_2. A *path* P from node s to destination t is represented by a series of nodes $P = \{n_1, n_2, \cdots, n_r\}$, in which $n_1 = s$, $n_r = t$ and the length $|P|$ is the sum of the weight of all edges on P. The *minimum detour distance* $D_d(o, P)$ of a data object o from a path P is defined as:

$$D_d(o, P) = \min_{n_i \in P} \{D_n(o, n_i)\}$$

We may also denote $D_d(o, P)$ by $D_d(o)$ alternatively when in a clear context.

Table 1: A list of notations

Notation	Description
V	The set of all nodes
E	The set of all edges
$weight(n_1, n_2)$	The weight of edge (n_1, n_2)
$P, \|P\|$	A path in a road network, and its length
(n_1, n_2)	The edge between n_1 and n_2, or the path from n_1 to n_2 if in a clear context
$SP(n_1, n_2)$	The shortest path between n_1 and n_2
$D_n(n_1, n_2)$	The network distance between n_1 and n_2
$D_d(o, P)$	The minimum detour distance of data object o from path P
$D_e(n_1, n_2)$	The Euclidean distance between n_1, n_2
$LB(o, P), UB(o, P)$	The lower bound and upper bound of minimum detour distance of o from P
$L_f(), L_r(), L_v()$	The distance labels in forward, reverse and verification searches
$Dist_p(n_i, s, t)$	The perpendicular distance from n_i to line (s,t)

Definition 1. (***k*-Path Nearest Neighbor query**)
Given a starting node s*, a destination node* t*, a road network* $G(V, E)$ *and a set of data objects* O ($O \subseteq V$)*, the k-Path Nearest Neighbor (k-PNN) query is to find the k data objects:* $O' = \{o_1, o_2, \cdots, o_k\}$ ($O' \subseteq O$)*, such that*

$$D_d(o_i, SP(s, t)) \leq D_d(o_j, SP(s, t)), \forall o_i \in O', o_j \in O - O'$$

Here $SP(s, t)$ is the shortest path from s to t. We aim to monitor the k-PNN relative to $SP(s, t)$ while s is moving in a road network. In our application scenarios, $SP(s, t)$ keeps changing and the k-PNN needs to be reported dynamically. Table 1 shows a list of notations used in this paper.

4. K–PATH NEAREST NEIGHBOR QUERY

Intuitively the k-PNN query can be solved by issuing at each node of the current shortest path a traditional k-NN search and thereafter combining all the results together. However the cost of this method is high especially in a monitoring scenario. Therefore, in this section, we propose the *Best-first Network Expansion* (BNE) algorithm for efficient monitoring of the k-PNN. The BNE is composed of three phases: the *searching phase* for finding the shortest path and potential candidates at the beginning; the *verification phase* for determining the exact k-PNN results; and the *monitoring phase* for updating the k-PNN efficiently. In the verification phase, the BNE always selects the data object which is most likely to be the closest one from the candidate set for verification, and that is why we call it *best-first.* As determining distance in a road network is a costly network expansion process, the BNE takes advantage of previous expansion results by maintaining an expansion tree and a candidate set of data objects that must contain the k-PNN results. In our approach, we estimate the minimum detour distance of a data object by a lower bound derived from the triangular inequality of shortest path, and that is the basis of our searching and verification algorithms. In a road network, the triangular inequality holds for shortest path such that

$$|SP(n_1, n_2)| + |SP(n_2, n_3)| \geq |SP(n_1, n_3)|$$

$$|SP(n_1, n_2)| - |SP(n_2, n_3)| \leq |SP(n_1, n_3)|$$

$SP(n_1, n_2)$ indicates the shortest path between nodes n_1

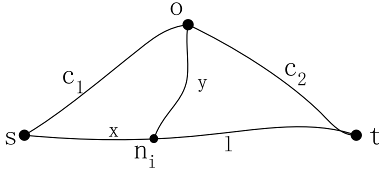

Figure 2: Lower bound

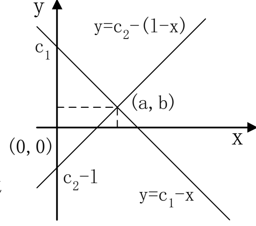

Figure 3: (a,b)

and n_2, and $|SP(n_1, n_2)|$ is the length of the path. Considering the illustration in Figure 2, there is a shortest path $SP(s,t)$ connecting the two nodes s and t with $|SP(s,t)| = l$, while o is a data object in the road network with $|SP(o,s)| = c_1$ and $|SP(o,t)| = c_2$. n_i is a node on the shortest path $SP(s,t)$. Obviously, o has an upper bound of minimum detour distance UB determined by

$$UB(o, SP(s,t)) = \min\{c_1, c_2\} \tag{1}$$

This upper bound can be further tightened during the searching phase as discussed later in this section. Now we expect to estimate the lower bound LB of the minimum detour distance for the data object o. Assume that the distance from s to n_i is x, and the distance from o to n_i is y. According to the triangular inequality theory stated above, we have:

$$\begin{cases} c_1 - x \leq y \\ c_2 - (l - x) \leq y \end{cases}$$

Therefore, the distance (y) from data object o to the shortest path $SP(s,t)$ is no shorter than LB:

$$LB = \min_{x \in [0,l]} \{\max\{c_1 - x, c_2 - (l - x)\}\} \tag{2}$$

Consequently the lower bound $LB(o, SP(s,t))$ of the minimum detour distance of o from $SP(s,t)$ is determined by figuring out the intersection point (a, b) of the two lines $y = c_1 - x$ and $y = c_2 - (l - x)$, as shown in Figure 3. We get:

$$a = \frac{l + c_1 - c_2}{2}, b = \frac{c_1 + c_2 - l}{2}$$

So the lower bound is estimated by

$$LB(o, SP(s,t)) = \frac{c_1 + c_2 - l}{2} \tag{3}$$

With l fixed, we can infer from Equation 3 that a smaller lower bound also implies a smaller value of $(c_1 + c_2)$, which means that $(c_1 + c_2)$ declines to l. This happens when the data object is closer to the shortest path connecting s and t. Therefore, a data object with smaller lower bound has higher opportunity in having a shorter minimum detour distance. Based on this observation, the BNE algorithm chooses data objects for verification in the order of their lower bounds until the current selected data object's minimum detour distance is smaller than the next object's lower bound.

Firstly, in the searching phase of our algorithm, the BNE finds the shortest path between s and t. Here, we adopt a bidirectional algorithm [12] by running the forward and reverse versions of the Dijkstra's algorithm [3] from s and t separately. The novel point is that we can also obtain the scanned nodes' lower bounds and upper bounds of the minimum detour distance during the searching for the shortest path. The forward version of the Dijkstra's algorithm expands from s and the reverse version expands from t in the road network, while each of them maintains its own set of distance labels. Once the two searches meet (a node scanned by the forward search has also been scanned by the reverse search, or vice versa), a shortest path from s to t is detected. During the search for the shortest path $SP(s,t)$, some data objects around s and t are scanned and their distances to s or t are determined as well. We can utilize these recorded distances for the verificaton of the k nearest neighbors in the following verification phase.

Another task during the bidirectional search is to get a candidate set of data objects that guarantees to include the k-PNN results. To achieve that, the bidirectional expansion may need to continue even after the shortest path is found, until we find a data object o, satisfying that the lower bound $LB(o, SP(s,t))$ is not less than at least k found data objects' upper bounds. We denote by $L_f(n_i)$ the distance label of a node n_i maintained by the forward search, and by $L_r(n_i)$ the distance label of a node n_i maintained by the reverse search, and by l the length of the shortest path $SP(s,t)$. We formalize the process as following: assume that during the bidirectional search, so far there is a set of k' data objects (O') get scanned (expanded) by either the forward search or the reverse search or both of them. Among O', each $o_i \in O'$ is assigned an upper bound $UB(o_i, SP(s,t)) = \min\{L_f(o_i), L_r(o_i)\}$ according to Equation 1, or $L_f(o_i)$ if only scanned by the forward search, or $L_r(o_i)$ if only scanned by the reverse search, while those scanned by both searches also have a lower bound $LB(o_i, SP(s,t)) = \frac{L_f(o_i)+L_r(o_i)-l}{2}$ according to Equation 3.

THEOREM 1. *During the bidirectional search, if there exists a data object $o \in O'$, and we can find at least k data objects $O = \{o_1, o_2, \cdots, o_k\}$ from O', such that*

$$LB(o, SP(s,t)) \geq \max_{o_i \in O}\{UB(o_i, SP(s,t))\}$$

Then, the k*-PNN must be included in O'.*

PROOF. For any data object o_j that is not in O', which means it has not been scanned yet, if we continue the bidirectional search till o_j gets both distance labels from the forward and the reverse searches, we have

$$L_f(o_j) \geq L_f(o_i), \forall o_i \in O'$$

$$L_r(o_j) \geq L_r(o_i), \forall o_i \in O'$$

because the search process based on the Dijkstra's algorithm always chooses the node with the smallest distance label value for expansion. $o \in O'$, then

$$\frac{L_f(o_j) + L_r(o_j) - l}{2} \geq \frac{L_f(o) + L_r(o) - l}{2}$$

$$\Rightarrow LB(o_j, SP(s,t)) \geq LB(o, SP(s,t))$$

$$\Rightarrow LB(o_j, SP(s,t)) \geq UB(o_i, SP(s,t), \forall o_i \in O$$

Therefore, any o_j must not have a minimum detour distance less than that of the k data objects in O found so far. □

Notice that the k data objects $\{o_1, o_2, \cdots, o_k\}$ are not necessarily to be the k-PNN results. We can only guarantee

that the k-PNN is within the set of data objects ($O^{'}$). The searching phase of the BNE is shown in Algorithm 1.

Algorithm 1: BNE - *searching phase*

```
   input  : Node s, t; G(V,E)
   output: SP(s,t); Candidate Set CS
1  S, T, Q_s, Q_t ← null; l ← ∞;
2  ∀p ∈ V, L_f(p), L_r(p) ← ∞; L_f(s), L_r(t) ← 0;
3  Q_s ← Q_s ∪ s; Q_t ← Q_t ∪ t;
4  Heap Lowerbounds, Upperbounds;
5  while Q_s, Q_t ≠ null do
      // Forward search
6     u ← ExtractMin(Q_s);
7     S ← S ∪ u;
8     if u ∈ T and l = ∞ then
9        l ← L_f(u) + L_r(u);
10       record SP(s,t);
11    for each node v ∈ u.adjacentNodes do
12       if L_f(v) > L_f(u) + weight(u,v) then
13          L_f(v) ← L_f(u) + weight(u,v);
14          Q_s ← Q_s ∪ v;
15          π_f(v) ← u;
16    if u is a data object then
17       Upperbounds.add(L_f(u));
18       if L_r(u) ≠ ∞ then
19          u.lowerbound ← (L_f(u)+L_r(u)−l)/2;
20          Lowerbounds.add(u.lowerbound);
21          k-minimal values ← Upperbounds.minK();
22          if Lowerbounds.min ≥ max{the k-minimal
            values} then
23             CS ← all data objects in S ∪ T;
24             return SP(s,t) & CS;
      // Reverse search
25    The same process as the forward search, with (S,
      Q_s, L_f(), π_f()) replaced by (T, Q_t, L_r(), π_r());
```

In Algorithm 1 the forward and reverse searches run alternately. During the initialization step, the sets of scanned nodes S and T are initialized to be *null*, and all nodes' distance labels except $L_f(s)$ and $L_r(t)$ are set to be ∞. The Heaps are for recording all data objects' lower bounds and upper bounds found so far (non-data object nodes' lower bounds/upper bounds are also recorded in another heaps). The search process is similar to the Dijkstra's algorithm, which always chooses the node with the minimal distance label for expansion (line 6). When a node scanned by both searches is found, the shortest path $SP(s,t)$ is recorded (line 9-10). A data object's upper bound of minimum detour distance is stored as the $\min\{L_f(u), L_r(u)\}$ (line 17), and once the object gets scanned by both forward and reverse searches, it is assigned a lower bound of the minimum detour distance (line 19). This part of the algorithm stops when Theorem 1 meets (line 22-24) and a candidate set is then returned.

Note that after the candidate set CS and the shortest path $SP(s,t)$ are returned, there could still be some data objects in CS that have not been scanned by both the forward and reverse searches and thus their lower bounds are unknown yet. Therefore, before going to the candidate verification phase, we further continue the network expansion of the bidirectional search until all data objects in CS have their lower bounds be determined. This part of the searching phase is intuitive and we omit it in Algorithm 1 for the simplicity of presentation.

During the searching phase presented above, we can also get two expansion trees T_f and T_r originated from s and t respectively (by recording parent node as $\pi_f(v)$,$\pi_r(v)$ at line 15), which can be re-used as 'pre-computed' knowledge in our monitoring phase. As illustrated in Figure 4 (we only show the expansion tree originated from t with thicker lines), if the user moves from s to another node $s^{'}$ that has already been included in T_r, then the shortest path from $s^{'}$ to t is figured out to be $SP(s^{'},t) = \{s^{'}, n_4, t\}$ by using the expansion tree easily without extra search. Besides, during the network expansion after $SP(s,t)$ is found in the searching phase, we can also **tighten the upper bounds** of some found data objects if their ancestor nodes in the expansion tree are on $SP(s,t)$. For example, the data object o in Figure 4 has an ancestor node n_3 (not necessarily the parent node) on $SP(s,t) = \{s, n_2, n_3, t\}$, then the upper bound $UB(o, SP(s,t))$ is tightened to be $|D_n(o, n_3)|$ and Algorithm 1 may return results faster since smaller upper bounds make Theorem 1 easier to be satisfied.

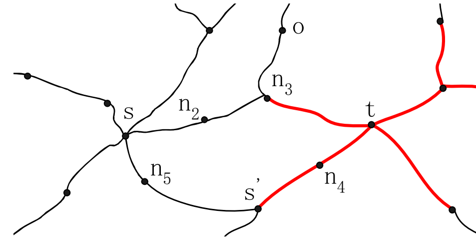

Figure 4: Expansion tree originated from t

On acquiring the candidate set CS together with lower bounds of candidates, as well as the shortest path $SP(s,t)$, the verification phase executes to verify the k-PNN candidates in CS in the sequence of their lower bounds as shown in Algorithm 2.

Algorithm 2: BNE - *verification phase*

```
   input  : Lowerbounds, SP(s,t)
   output: k-PNN
1  count ← 0; Heap kpnn; kpnn.max ← ∞;
2  while Lowerbounds ≠ null do
3     o ← Lowerbounds.popMin();
4     if kpnn.max > o.lowerbound then
5        D_d(o, SP(s,t)) ← verify(o, SP(s,t));
6        if D_d(o, SP(s,t)) < kpnn.max then
7           if count < k then
8              kpnn.add(o);
9              count++;
10          else
11             kpnn.deleteMax();
12             kpnn.add(o);
13    else
14       return kpnn;
```

The verification phase examines the exact minimum detour distance of each candidate from CS in the order of lower bound (the node with the minimal lower bound is pop out

at line 3), until a candidate's lower bound is not less than the $kpnn$'s max value (line 4-12, $kpnn$ stores the k minimal detour distances found so far). The $verify()$ function performs a network expansion from the candidate o to get its exact minimum detour distance. As this function is invoked every time an update occurs, the expansion method can affect the efficiency of monitoring significantly. Normally, the Dijkstra's expansion method can be used. Here, we propose a heuristic expansion approach that improves the efficiency greatly. The basic idea is to select the next node n with the minimum $(D_n(n, o) + n.detourEstimate)$ for expansion. $n.detourEstimate$ is the estimate of n's minimum detour distance, and it is determined by either $LB(n, SP(s, t))$, or $Dist_p(n, s, t)$ which is the perpendicular distance from n to the line (s, t). $Dist_p(n, s, t)$ uses Euclidean distance to approximate the minimum detour distance and it can be easily figured out by using the Cosine Theorem as follows. Let $c_1 = D_e(n, s)$, $c_2 = D_e(n, t)$ and $l = D_e(s, t)$ ($D_e()$ is Euclidean distance), then we have:

$$Dist_p(n, s, t) = |c_1 \times \sin(\arccos(\frac{c_1^2 + l^2 - c_2^2}{2c_1 l}))|$$

However, the Euclidean detour estimate may not be applicable when the weight of an edge is not measured by real geographic distance (e.g. time cost). In contrast $LB(n, SP(s, t))$ gives a more tightened estimate and holds for any type of edge weight. One potential drawback is that some nodes encountered during the expansion may have not been previously scanned yet and have no lower bound determined. In this case we need further expansion of T_f and T_r to get the node's lower bound. However, in our experiments on real datasets, this situation is rare and very limited number of encountered nodes haven't been scanned as most of them are covered by the expansion trees.

Basically, the search area of the $verify()$ function using the Dijkstra's expansion is a circle, while the search area is normally in a triangle shape towards $SP(s, t)$ if using the detour estimate as a heuristic. Algorithm 3 describes the details.

Algorithm 3: $verify(o, SP(s, t))$

```
1  S_v, Q_v ← null; detourDist ← ∞;
2  ∀p ∈ V, L_v(p) ← ∞; L_v(o) ← 0; Q_v ← Q_v ∪ o;
3  while Q_v ≠ null do
4      n ← ExtractMin(Q_v), such that
       L_v(n) + n.detourEstimate is minimized ;
5      if L_v(n) + n.detourEstimate ≥ detourDist then
6          return detourDist;
7      if n ∈ SP(s,t) and detourDist > L_v(n) then
8          detourDist ← L_v(n);
9      S_v ← S_v ∪ n;
10     for each node v ∈ n.adjacentNodes do
11         if L_v(v) > L_v(n) + weight(n, v) then
12             L_v(v) ← L_v(n) + weight(n, v);
13             Q_v ← Q_v ∪ v;
```

In Algorithm 3, the node with the minimum $(D_n(n, o) + n.detourEstimate)$ gets explored first (line 4). Once a node $\in SP(s, t)$ gets scanned (line 7-8), a detour path from o to $SP(s, t)$ is found and we update the current minimum detour distance $detourDist$ if a shorter one is found. Here, $L_v()$ is the distance label indicating how far a node is from o. Notice that the $verify()$ function may continue the search even after it reaches the shortest path $SP(s, t)$ since it is not necessarily that a node with smaller distance label $L_v(n)$ gets explored first, until the current $detourDist$ is not greater than the current $(L_v(n) + n.detourEstimate)$ which is a lower bound of all unscanned nodes' minimum detour distances (line 5-6). The correctness of Algorithm 3 is guaranteed as stated in the following:

LEMMA 1. *For every node n scanned by the $verify()$ function, $L_v(n)$ is equal to the length of the shortest path $SP(o, n)$, where o is the data object for verification.*

PROOF. Denote $detourEstimate$ by e. The $verify()$ function's expansion method is equal to the Dijkstra's algorithm if we replace the distance label $L_v(n)$ by $L_v(n) + n.e$. Thus we can define a new weight of an edge as:

$$weight'(n_1, n_2) = L_v(n_2) + n_2.e - (L_v(n_1) + n_1.e)$$

$$= weight(n_1, n_2) - n_1.e + n_2.e$$

$weight(n_1, n_2)$ is the original weight defined in $G(V, E)$. Straightforwardly, $weight(n_1, n_2) - n_1.e + n_2.e \geq 0$ because of the triangular inequality (proof by replacing e with Equation 3). Suppose we replace the $weight$ of each edge in $G(V, E)$ by the non-negative $weight'$. Then for any two nodes n_x, n_y, the length of any path from n_x to n_y changes by the same amount: $n_y.e - n_x.e$. Therefore, a path is the shortest path from n_x to n_y with respect to $weight$, if and only if it is also the shortest path from n_x to n_y with respect to $weight'$. □

The rationale of the expansion method in Algorithm 3 is similar to that of the $A^\star$ algorithm [5], although a different heuristic is designed, and the *detour estimate* is essentially a feasible potential function in [4]. As $L_v(n)$ is guaranteed to be the length of the shortest path from o by Lemma 1, once the minimal $detourDist$ is confirmed, it must be the minimum detour distance from o to $SP(s, t)$.

5. MONITORING K–PNN

In this section, we present the monitoring phase of the BNE algorithm and show how to update the k-PNN results when the user is moving arbitrarily. As described before, the user may deviate from the shortest path and then the current shortest path which is actually the query path may be changed from time to time, and thus an update of the k-PNN results is caused by the change of the query path. Even though the user always follows the shortest path, the path is also becoming shorter while the user is going towards the destination. Therefore, we need to deal with the shortest path update and consequently the k-PNN update.

In this part, the candidate set CS of data objects, the expansion tree T_r and T_f rooted at t and s respectively, as well as lower bounds and upper bounds of scanned nodes that acquired previously are all further utilized and carefully maintained in the monitoring phase as they provide 'pre-computed' knowledge to accelerate our update algorithm. Obviously, T_r is static because the destination does not change, by which we can figure out a node's shortest path to the destination quickly. As the user is probably moving closer gradually towards the destination, the candidate set CS probably covers the new k-PNN results. All

these information are also updated gradually in the monitoring phase, based on which we design the update algorithms.

There are basically two types of updates for the k-PNN: (1) update of the order; and (2) update of the members. In the first category, the k-PNN results are still the same but the order with respect to minimum detour distance changes, while in the second category some data objects of the k-PNN become invalid and new data objects are inserted into the k-PNN results. Now the problem is to determine where an update of the k-PNN will be needed (i.e. update location), and then only refresh the k-PNN results when necessary. In the following, we present our update algorithms for the cases when the user follows the shortest path, and deviates from the shortest path.

5.1 Following the Shortest Path

Firstly, we discuss the case that so far the user follows the shortest path found previously. Figure 5 illustrates such a 4-PNN $= \{o_2, o_5, o_4, o_3\}$ example, in which we assume the user follows $SP(s,t)$ and his/her current position is denoted by $s^{'}$. The shortest path from a data object o_i to $SP(s,t)$ intersects $SP(s,t)$ at n_i, and we call $SP(o_i, n_i)$ the *minimum detour path* of o_i, and n_i the *entrance point* of o_i's minimum detour path. For instance $SP(o_2, n_2)$ is the minimum detour path of o_2, and n_2 is o_2's entrance point.

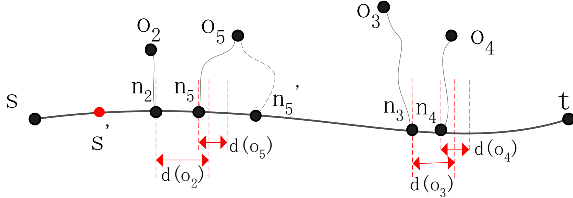

Figure 5: Update locations

It is not hard to see that before s' reaches the first entrance point of the current k-PNN (n_2 in this example), neither the order nor the members of the k-PNN needs to be updated, because when $s^{'}$ is on $SP(s,t)$, we have $SP(s^{'},t) \subseteq SP(s,t)$, which means $SP(s^{'},t)$ is the same as the part of $SP(s,t)$ from $s^{'}$ to t, and hence the minimum detour path of any o_i does not change, and there can not be any other data object closer to $SP(s^{'},t)$, otherwise the closer data object must be included in the k-PNN of $SP(s,t)$.

Once $s^{'}$ overtakes the entrance point of a data object o_i, the minimum detour distance of o_i will increase and thus it may affect the order of the k-PNN. For instance, when $s^{'}$ overtakes n_2 and keeps going forwards, the minimum detour distance of o_2 becomes larger and the order of o_2 (the 1st PNN) and o_5 (the 2nd PNN) may change when o_2's minimum detour distance rises to a certain value. If o_i is just the kth PNN, it may also become invalid and the $(k+1)^{th}$ PNN will replace it to be the kth PNN. To detect the change of the kth PNN, we actually maintain the $(k+1)$-PNN results in the algorithm, and we calculate the update locations for the k-PNN to indicate where a change of the order could happen. Normally, an update location for a data object o_i is computed every time when $s^{'}$ arrives at o_i's entrance point by:

$$d(o_i) = |SP(o_j, n_j)| - |SP(o_i, n_i)| \tag{4}$$

where $d(o_i)$ is the distance from o_i's entrance point n_i to the update location. Let o_i be the λth PNN ($\lambda \le k$), then we choose $o_j = (\lambda+1)^{th}$ PNN for calculating $d(o_i)$ in Equation 4. The idea is that an upper bound of the minimum detour distance of o_i from $SP(s^{'},t)$ is $|SP(o_i, n_i)| + |SP(n_i, s^{'})|$, and as long as this upper bound is smaller than the $(\lambda+1)^{th}$ PNN's minimum detour distance $|SP(o_j, n_j)|$, the order of the k-PNN keeps the same.

For example in Figure 5, the 4-PNN $= \{o_2, o_5, o_4, o_3\}$, when $s^{'}$ arrives at n_2, it generates an update location for o_2 determined by $d(o_2)$, which is equal to $|SP(o_5, n_5)| - |SP(o_2, n_2)|$. While the user is traveling within the range of $d(o_2)$ from n_2, it is expected that no change of the order between o_2 and o_5 is required. However, if the $(\lambda+1)^{th}$ PNN's entrance point is met before $s^{'}$ arrives at the λth PNN's update location, for example $s^{'}$ meets o_5's entrance point n_5 and it generates an update location for o_5 with $d(o_5)$ as shown in Figure 5, in this case o_5's update location is reset to be the same as o_2's update location which is closer to $s^{'}$, because we need to re-compute both o_2 and o_5's minimum detour distances at o_2's update location to determine whether the order changes, and to figure out their next update locations. However, if o_5's update location is closer to $s^{'}$, we do not need to reset o_5's update location. Similarly, if the $(\lambda-1)^{th}$ PNN's entrance point is met before $s^{'}$ arrives at the λth PNN's update location, like that n_4 is encountered before $s^{'}$ reaches o_3's update location as illustrated in Figure 5, since o_3's update location is closer to $s^{'}$, there is no need to adjust o_3's update location. Algorithm 4 shows how to determine the update location when encountering a data object's entrance point.

Algorithm 4: Encountering o_i's *entrance point*

```
/* o_i = the λ^th PNN          */
/* o_j = the (λ+1)^th PNN      */
/* o_k = the (λ−1)^th PNN      */
1 o_i.updateLoc ← pos(n_i) + d(o_i);
2 if o_k.updateLoc ≠ null and
  o_k.updateLoc < o_i.updateLoc then
3   o_i.updateLoc ← o_k.updateLoc;
4 if o_j.updateLoc ≠ null and
  o_i.updateLoc < o_j.updateLoc then
5   o_j.updateLoc ← o_i.updateLoc;
```

Here, $pos(o_i)$ is the position of o_i, and $d(o_i)$ is computed by Equation 4. The criteria is to reset a lower ranking PNN's update location (denoted by *updateLoc*) to the higher ranking one's update location if the higher one's update location is closer to $s^{'}$ (with a smaller value).

On arriving at o_i's update location, the minimum detour distance of o_i is re-examined by running Algorithm 3 and the k-PNN is refreshed accordingly. Recall Algorithm 3, note that the lower bound $LB(o_i, SP(s,t))$ determined previously at s can still be used as the detour estimate in the verification process even the current query path has changed to be $SP(s^{'},t)$, because $LB(o_i, SP(s,t)) \le LB(o_i, SP(s^{'},t))$. Let $D_n(o_i, s) = c_1$, $D_n(o_i, t) = c_2$, $D_n(o_i, s^{'}) = c_3$, we have:

$$LB(o_i, SP(s,t)) - LB(o_i, SP(s^{'},t))$$

$$= \frac{c_1 + c_2 - |SP(s,t)|}{2} - \frac{c_2 + c_3 - |SP(s^{'},t)|}{2}$$

$$= \frac{(c_1 - c_3) - (|SP(s,t)| - |SP(s',t)|)}{2}$$

$$= \frac{(c_1 - c_3) - |SP(s,s')|}{2} \leq 0$$

The update algorithm is invoked when encountering an *update location Loc* as described in Algorithm 5. Firstly it verifies all corresponding data objects' minimum detour distances, and then refreshes the order of the $(k+1)$-PNN. If the previous k^{th} PNN is not valid any longer (line 5), a recomputation of the whole $(k+1)$-PNN is executed by calling the $updateKPNN()$ function in Algorithm 6.

Algorithm 5: Encountering an *update location Loc*

1 **for** *each object* o_i *that* $o_i.updateLoc = Loc$ **do**
2 $\quad D_d(o_i) \leftarrow verify(o_i, SP(Loc, t))$;
3 $\quad$ remove $o_i.updateLoc$;
4 refresh the order of the $(k+1)$-PNN ;
5 **if** *the* k^{th} *PNN is changed* **then**
6 $\quad updateKPNN(Loc, t)$;
7 **for** *each object* o_i' *that* $o_i'.entrancePoint = Loc$ **do**
8 $\quad$ calculate $o_i'.updateLoc$ by Algorithm 4;

In some cases, o_i's minimum detour path may have a new entrance point even ahead of s' after verification, such as n_5' in Figure 5. After the update of k-PNN, a data object is assigned a new update location if its new entrance point is right at s' (line 7-8).

Algorithm 6: $updateKPNN(n, t)$

1 $S, Q_s \leftarrow null$; $\forall p \in V$, $L_f(p) \leftarrow \infty$;
2 $L_f(n) \leftarrow 0$; $Q_s \leftarrow Q_s \cup n$;
3 Heap $Lowerbounds, Upperbounds$;
4 **while** $Q_s \neq null$ **do**
5 $\quad u \leftarrow ExtractMin(Q_s)$;
6 $\quad S \leftarrow S \cup u$;
7 $\quad$ **if** $u \in T_r$ *and SP(n,t) is not determined* **then**
8 $\quad\quad$ record $SP(n,t)$;
9 $\quad$ **for** *each node* $v \in u.adjacentNodes$ **do**
10 $\quad\quad$ **if** $L_f(v) > L_f(u) + weight(u,v)$ **then**
11 $\quad\quad\quad L_f(v) \leftarrow L_f(u) + weight(u,v)$;
12 $\quad\quad\quad Q_s \leftarrow Q_s \cup v$; $\pi_f(v) \leftarrow u$;
13 $\quad$ **if** *u is a data object* **then**
14 $\quad\quad Upperbounds.add(L_f(u))$;
15 $\quad\quad$ **if** $u \notin T_r$ **then**
16 $\quad\quad\quad$ further expand T_r until $L_r(u) \neq \infty$;
17 $\quad\quad u.lowerbound \leftarrow \frac{L_f(u)+L_r(u)-|SP(n,t)|}{2}$;
18 $\quad\quad Lowerbounds.add(u.lowerbound)$;
19 $\quad\quad$ k-minimal values $\leftarrow Upperbounds$.minK();
20 $\quad\quad$ **if** $Lowerbounds.min \geq \max\{$*the k-minimal values*$\}$ **then**
21 $\quad\quad\quad T_r \leftarrow T_r - \{n_i : L_r(n_i) > L_r(u)\}$;
22 $\quad\quad\quad CS \leftarrow$ all data objects in $S \cup T_r$;
23 $\quad\quad\quad$ break;
24 continue the expansion until for each $n_i \in CS$ we have $n_i.lowerbound \neq null$;
25 run Algorithm 2 for verifying k-PNN;

In Algorithm 6, a Dijkstra's expansion from the current position n is conducted to update the candidate set CS, and all candidates' lower bounds and upper bounds of the minimum detour distance. This process is similar to the searching phase in Algorithm 1. Since the expansion tree T_r rooted at t and the distance label $L_r(u)$ are invariable, we just need a forward expansion from n to get $L_f(u)$ and subsequently the lower bound of n. All $L_r(u)$ $(u \in Tr)$ are added to the *Upperbounds* in the initialization step. If a data object scanned by the forward expansion is not included in T_r (line 15), which happens when the user deviates from the shortest path too much, T_r needs a further expansion to catch up with the forward expansion, and during the expansion of T_r the shortest path $SP(n,t)$ may also be recorded if it has not been determined yet (line 16). In fact, with the user approaching the destination, a smaller search area from n is required, and the candidate set CS and the expansion tree T_r are also updated to smaller ones (line 21-22). At the same time, all scanned nodes' lower bounds and upper bounds of minimum detour distance are also updated. Note that at line 14, we choose the $\min\{L_f(u), L_r(u)\}$ as u's upper bound. Finally, the verification function runs to acquire the exact $(k+1)$-PNN results. As the k-PNN is already known, we just need a verification for the $(k+1)^{th}$ PNN.

5.2 Deviating from the Shortest Path

In the case that the user does not follow the shortest path, as exemplified in Figure 6, and leaves the current shortest path $SP(s,t) = \{s, n_2, n_3, t\}$ for destination t through n_5, s_t and n_4, firstly we need to update the current shortest path to the destination. There will be a *split point* f on the coming edge such that the shortest path from the current position s' to t is $\{s', s, n_2, n_3, t\}$ through node s when s' is on the path (s, f), and the shortest path changes to be $\{s', s_t, n_4, t\}$ through node s_t after the user passes f.

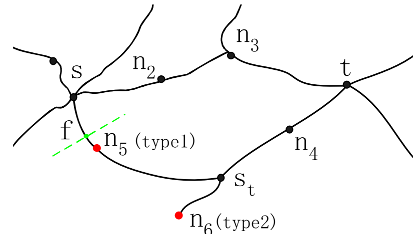

Figure 6: Split point & Object types

To find the split point *f*, first of all we search along the coming edges until encountering the first node with out degree ≥ 3 (s_t in this example), and it is easy to see that the shortest path from s' to t must go through $SP(s,t)$ or $SP(s_t,t)$ when s' is on the path (s, s_t). So the next step is to find the shortest path $SP(s_t,t)$. If s_t is already contained in the expansion tree T_r, $SP(s_t,t)$ can be constructed by tracing upwards from s_t along parent node (recorded by $\pi_r()$) until it reaches the root t. Otherwise, again a Dijkstra's network expansion from s_t is conducted, trying to touch the expansion tree T_r. As stated before, T_r covers the surrounding area of t, therefore, as long as the user does not deviate too much, s_t is close to T_r and the expansion from s_t will meet T_r very soon, after which the shortest path from s_t to t is determined just like that in the bidirectional search of the searching phase. In addition, the expansion tree T_f rooted

at s probably also includes s_t, so the branch of T_f starts from s_t can be re-used for the expansion. This is similar to the query update in [11], and other branches of T_f are then discarded. Once the shortest path $SP(s_t, t)$ is determined, the split point f is figured out by:

$$|(s, f)| = \frac{|SP(s, t)| + |SP(s_t, t)| + |(s, s_t)|}{2} - |SP(s, t)|$$

$$= \frac{|SP(s_t, t)| - |SP(s, t)| + |(s, s_t)|}{2}$$

where $|(s, f)|$ is the length of the path (s, f), and $|(s, s_t)|$ is the length of (s, s_t) (the path along which the user goes from s to s_t). Occasionally if $SP(s_t, t)$ is through s, we set the split point at node s_t, and $SP(s', t)$ is always through s when the user moves on (s, s_t).

In the following, we elaborate how to update the k-PNN when the user is moving on (s, f) only, since after the user passes the split point f, we can monitor the k-PNN as if the user follows the shortest path $SP(f, t)$ and the algorithm for that is already introduced in Subsection 5.1. During the user's movement on (s, f), however, the computation of update locations is different from the previous method in Algorithm 4. Assume the user is currently at $s' \in (s, f)$, we observe that the k-PNN of $SP(s', t)$ must be from the k-PNN results of $SP(s, t)$, or those data objects become closer enough to $SP(s', t)$ because of the movement on (s, f). Based on this observation, we develop the following lemma:

LEMMA 2. *Let kpnn be the* k-PNN *of* $SP(s, t)$, *knn be the k nearest neighbors of* s_t *and* O_{s,s_t} *be the set of all data objects located on path* (s, s_t). *When* s' *is on the path* (s, f) *between s and the* split point f, *the* k-PNN *of* $SP(s', t)$ *must be included in* $\{kpnn \cup knn \cup O_{s,s_t}\}$.

PROOF. Suppose on the contrary there exists a data object o such that o belongs to the k-PNN of $SP(s', t)$, and o is not in $\{kpnn \cup knn \cup O_{s,s_t}\}$. As stated previously, $SP(s', t)$ equals to $SP(s, t)$ plus (s, s') when s' is on (s, f). If o's entrance point is on $SP(s, t)$, straightforwardly o must be included in $kpnn$. Except that, the only way o connects to $SP(s', t)$ is through node s_t, or o is right located on the path (s, s_t). In the former case o can not have the minimum detour distance shorter than that of the k-NN of s_t, while in the later case o is a data object lies on (s, s_t). Therefore, o must be included in $\{kpnn \cup knn \cup O_{s,s_t}\}$. □

From Lemma 2, we confirm that the k-PNN must be from $\{kpnn \cup knn \cup O_{s,s_t}\}$ when the user is moving on (s, f), and hence only data objects belong to this set may have an update location. Furthermore, for data objects belong to this set, there are only two types of data objects (type1 and type2) as exemplified in Figure 6 that can trigger an update on the k-PNN. Data objects of ***type1*** are all those objects on path (s, s_t), and data objects of ***type2*** are the k-NN of s_t except those belong to type1. For a data object contained in the k-PNN of $SP(s, t)$ excluding those in type1 and type2, it's minimum detour distance does not change during the user's movement on (s,f), and thus it does not have an update location. For type1 and type2 data objects, their minimum detour distances may decrease as the user moves towards the split point, and we calculate the update locations for them when a deviation occurs.

(1) **For a data object o_i of type1**, it may become closer to the current shortest path $SP(s', t)$ since $SP(s', t)$ extends with s' moving towards o_i. If o_i is already the λth PNN ($\lambda \leq k$), its update location is then determined by:

$$o_i.updateLoc = pos(s') + |(s', o_i)| - D_d((\lambda - 1)^{th} PNN)$$

Here $pos(s')$ is the user's current position which is initially equal to $pos(s)$, and $|(s', o_i)|$ is the distance from s' to o_i along path (s, f). $o_i.updateLoc$ stands for the position where the distance from s' to o_i drops to $D_d((\lambda - 1)^{th} PNN)$ and o_i may become the $(\lambda - 1)^{th}$ PNN.

Otherwise, if o_i is not included in the current k-PNN and then we compare $|(s', o_i)|$ with the kth PNN's minimum detour distance and decide where o_i may become the kth PNN:

$$o_i.updateLoc = pos(s') + |(s', o_i)| - D_d(k^{th} PNN)$$

(2) **For a data object o_i of type2**, similarly, if o_i is the λth PNN ($\lambda \leq k$), then we have $o_i.updateLoc =$

$$pos(s') + |(s', s_t)| + D_n(o_i, s_t) - D_d((\lambda - 1)^{th} PNN)$$

In this equation, $|(s', s_t)| + D_n(o_i, s_t)$ is the distance from the user's current position s' to o_i through path (s, s_t). Otherwise if o_i does not belong to the k-PNN, we have:

$$o_i.updateLoc = pos(s') + |(s', s_t)| + D_n(o_i, s_t) - D_d(k^{th} PNN)$$

In both cases, we assign $D_d(0^{th} PNN) = D_d(1^{st} PNN)$. If $\lambda^{th} PNN.updateLoc < (\lambda - 1)^{th} PNN.updateLoc$, then the $(\lambda-1)^{th}$ PNN's update location is reset to be the same as the λth PNN's update location, as we need to get both the λth and $(\lambda - 1)^{th}$ PNNs' exact minimum detour distances when refreshing the order of k-PNN. If $o_i.updateLoc > |(s, f)|$, the update location for o_i is not a valid one because the current shortest path will change after the user crosses f. On encountering an update location during the deviation from s towards the split point f, Algorithm 5 is called for updating the k-PNN results. The steps for processing updates when a deviation happens are summarized in Algorithm 7.

Algorithm 7: Dealing with deviation

1. search ahead for the next node s_t with out degree ≥ 3;
2. get $SP(s_t, t)$ and the k-NN of s_t;
3. calculate the *split point* f, and *update location*s;
4. update the current k-PNN by using Algorithm 5;

Notice that, at line 2, the k-NN of s_t can be acquired by recording the k first found data objects during the search for $SP(s_t, t)$. If $|(f, s_t)| \geq D_d(k^{th} PNN)$, then all data objects of type 2 do not have an update location. Once the user passes the split point, the whole k-PNN is updated by calling the $updateKPNN(f, t)$ defined in Algorithm 6.

6. EXPERIMENTS

In this section, we conduct experiments on datasets of California Road Network and City of Oldenburg Road Network[1] (stored as adjacency lists), which contain $21,048$ nodes and $6,105$ nodes respectively (see Figure 7 & Figure 8). All algorithms are implemented in Java and tested on a windows platform with Intel Core2 CPU (2.13GHz) and 2GB memory. The main metric we adopt is the *CPU time* that

[1] http://www.cs.fsu.edu/ lifeifei/SpatialDataset.htm

reflects how much time the monitoring process of our algorithm costs during the user's movement from the start to the end. Data objects for monitoring are generated on the networks uniformly with different *density* from 1% to 10% which is equal to ($\frac{\text{the number of data objects}}{\text{the number of nodes in the network}}$). In both Figure 7 and Figure 8, a 5% density of distribution is illustrated. We also generate query paths with different *deviation* from 0% (no deviation, i.e. shortest path) to 40%. The *deviation* is defined as the percentage of how many times a user deviates from the shortest path to how many intersections totally on the user's route to the destination (i.e. $\frac{\text{the number of deviations}}{\text{the number of intersetions}}$). The setting of our experiments is summarized in Table 2.

Table 2: Experiment setting

Name	Value
Density of data object	1% to 10%. Default: 5%
Deviation of query path	10% to 40%. Default: 10%
Length of query path	100 to 500 in California Road Network, and 30 to 100 in Oldenburg Road Network. Default: 200 and 80
Number of k	1 to 15. Default:10

By default, we set the density of data object distribution to be 5%, as according to our analysis of the California Road Network and Points of Interest, 71.4% categories of data objects have a distribution density less than 5%, and the deviation is set to be 10% which means the possibility that a user does not follow the shortest path at an intersection is 1/10, and the length of query path is configured as 200 when using the California Road Network, and 80 when using the Oldenburg Road Network (200 and 80 are close to the diameters of the networks).

For the purpose of comparison, an algorithm based on the In-Route Nearest Neighbor query [20] is also implemented, and we mention it as *Monitoring based on IRNN* (MIRNN). In this algorithm, the way to figure out update locations and split points is still the same as that in the BNE algorithm, but we replace the parts of searching and updating k-PNN with the *SDJ* algorithm in [20], which implements IRNN query for a given path (pseudo code can be found in [23]). The *SDJ* algorithm utilizes closest pair query (distance join) [2] for determining the order of k-PNN verification and an R-tree index of data objects is adopted. In MIRNN, every time an update of the whole k-PNN is needed, the SDJ algorithm is invoked, and the current shortest path is also established by a bi-directional Dijkstra's search.

6.1 Effect of Data Object Density

First of all, we study the effect of data object density on the performance of *10*-PNN, with the deviation of query path fixed at 10%, and path length fixed at 200 and 80 in the California and Oldenburg Road Networks respectively. Intuitively, the denser the data object distribution is, the smaller search area is required and thus the results are responded more quickly. As we can see in Figure 9(a) and Figure 10(a), the *CPU time* decreases for both the BNE and MIRNN while the density rises. However, the density has very limited influence on the BNE whose performance is quite stable with *CPU time* always lower than 5 seconds while the MIRNN generates very heavy load when data objects are not densely distributed. In fact, with a sparse distribution, the k-PNN results do not change frequently if the actual query path does not deviate from the shortest path too much. While the BNE can always re-use the previous expansion tree and candidate set for monitoring, the MIRNN has to perform distance join operations for each k-PNN update. When the distribution is dense enough, the performance of both algorithms tend to be similar.

Figure 7: California Road Network

Figure 8: City of Oldenburg Road Network

Actually, when data objects are densely distributed, it is likely that the k-PNN results all lie on the path with minimum detour distance equals to 0, therefore once the shortest path to the destination is found, the k data objects on the path are also discovered. So we mainly design the Path Nearest Neighbor query for data objects that are not very densely distributed (e.g. gas station), although our algorithm can also cope with high density efficiently.

6.2 Effect of Path Length

The query path is the route on which the user travels to the destination. It is expected that the monitoring cost is in linear to the length of the path since a longer path causes more updates of the k-PNN. In this experiment, we test the performance of *10*-PNN with the query path length varies from 100 to 500, and 30 to 100, in the California and Oldenburg Road Networks respectively. The density is set to be 5% and deviation is set to be 10%. In the California Road Network, as shown in Figure 9(b), when the path length is between 100 and 300, the cost of the BNE algorithm increases gradually from less than 1 second to nearly 5 seconds, while the cost of the MIRNN rises dramatically to nearly 17 seconds. After that, the query path becomes

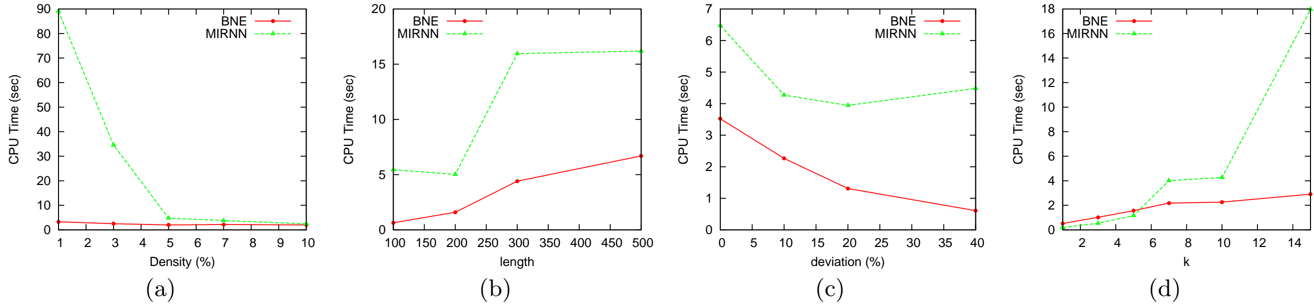

Figure 9: Performance in California Road Network

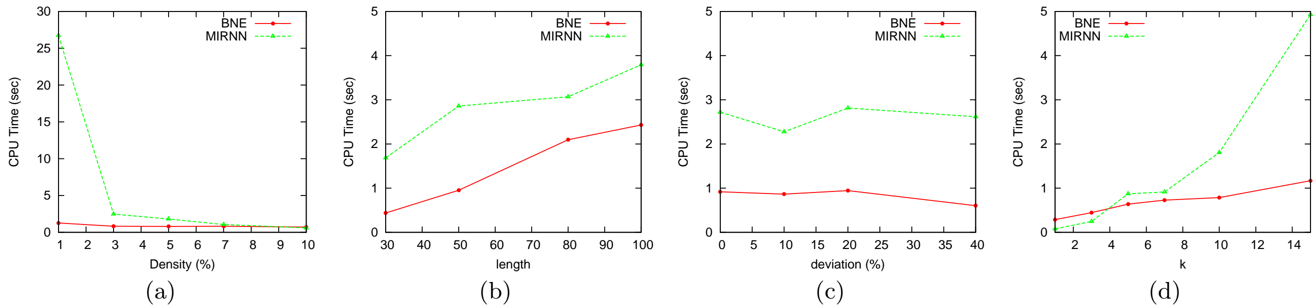

Figure 10: Performance in City of Oldenburg Road Network

more and more 'zigzag' as the path grows longer in a fixed-size network, and the performance of both methods tend to be stable as the search space does not increase in proportional to the length. In the Oldenburg Road Network, the size of which is a smaller, the curves are similar, while the BNE always outperforms the MIRNN method (Figure 10(b)).

6.3 Effect of Deviation

The deviation of a query path reflects how 'zigzag' the path is. A query path with deviation equals to 0% is actually the shortest path connecting the start and end nodes. High deviation implies that the user usually does not choose the shortest path and an update of the whole k-PNN is required each time the user deviates. Straightforwardly, higher deviation causes more updates and thus higher cost. However, interestingly, the cost of the BNE algorithm in both datasets drops increasingly with the deviation goes from 0% to 40%, as shown in Figure 9(c) and Figure 10(c) (with default settings for other paramters). The reason is that for a fixed length query path, a smaller search area can cover the whole path, compared with the search area for a not so 'zigzag' query path. Therefore, a smaller expansion tree is maintained by the BNE algorithm to cover all the data object candidates. As a consequence, the performance of the BNE is improved.

In contrast, for the MIRNN method, as illustrated in Figure 9(c) and Figure 10(c), the performance is improved as the deviation increases from 0% to 10% at the beginning, after which the cost rises or fluctuates because the MIRNN highly depends on the efficiency of the closest pair query which may involve expensive self-join.

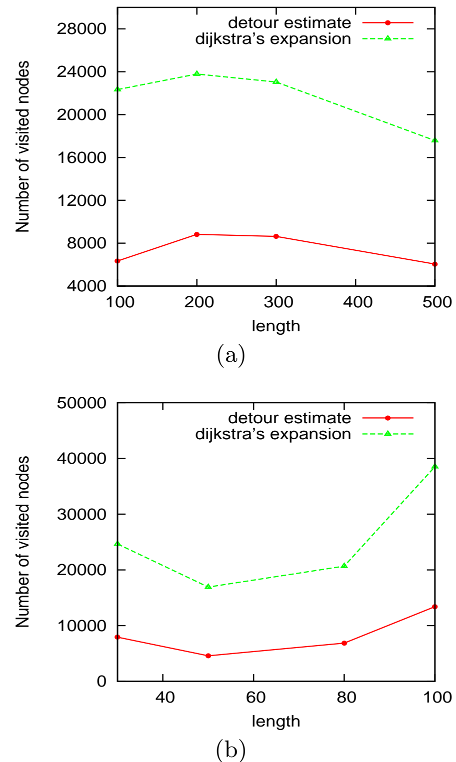

Figure 11: Performance of verification

6.4 Effect of k

The number of k is another critical parameter affecting the performance. Nevertheless, as we can see in Figure 9(d) and Figure 10(d), the cost of the BNE rises slowly with k goes up from 0 to 15, which implies that the search cost of determining the candidate set as well as the $(k+1)$-PNN only differs slightly for different $k \leq 15$. In comparison with the BNE, the performance of the MIRNN degrades drastically with k increases because of a dramatic rise in the number of join operations. Note that when k is small (e.g. ≤ 3), the performance of the MIRNN is close to that of the BNE or even better slightly. So we may say the BNE is much more efficient in handling large number of k.

6.5 The Verification Function

As stated in section 4, the $verifiy$ function of the BNE algorithm determines how efficiently the minimum detour distance of a data object candidate can be figured out. We compare the performance of the $verify$ function that using the lower bound as detour estimate, with that of simple Dijkstra's expansion, by how many nodes totally they visit during the monitoring process. Under the default settings, the detour estimate based approach reduces the number of visited nodes by approximately 66% in the California Road Network (Figure 11(a)), and by about 50% in the Oldenburg Road Network (Figure 11(b)) compared with the Dijkstra's expansion method.

7. CONCLUSION

In this paper we propose a new query for monitoring k-PNN which retrieves nearest neighbors by considering the whole coming journey of the user, and we present a three-phase BNE algorithm for efficient searching and monitoring of the k-PNN while the user is moving arbitrarily. The BNE utilizes a bi-directional search scheme to acquire the current shortest path to the destination and data object candidates as well, and a heuristic verification function is designed for examining each candidate's exact minimum detour distance efficiently. The monitoring part mainly involves the calculation of update locations of the k-PNN. In all these phases, information from previous searching are well maintained to minimize the new computation effort. Finally, the BNE algorithm is tested using real datasets with different settings. As we can see, the BNE provides efficient monitoring under different data object density, and performs well when the length/deviation of query path increases.

8. REFERENCES

[1] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cnn queries in a road network. In *Proceedings of VLDB*, pages 865–876, 2005.

[2] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of SIGMOD*, pages 189–200, 2000.

[3] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Math*, 1:269–271, 1959.

[4] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of SODA*, pages 156–165, 2005.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[7] C. S. Jensen, J. Kolářvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proceedings of GIS*, pages 1–8, 2003.

[8] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of VLDB*, pages 840–851, 2004.

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

[10] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proceedings of SIGMOD*, pages 634–645, 2005.

[11] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proceedings of VLDB*, pages 43–54, 2006.

[12] T. A. J. Nicholson. Finding the shortest route between two points in a network. *Computer Journal*, 9(3):275–280, 1966.

[13] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *Proc. VLDB Endow.*, 1(1):1095–1106, 2008.

[14] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proceedings of ICDE*, page 301, 2004.

[15] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proceedings of VLDB*, pages 802–813, 2003.

[16] L. Qin, J. X. Yu, B. Ding, and Y. Ishikawa. Monitoring aggregate k-nn objects in road networks. In *Proceedings of SSDBM*, pages 168–186, 2008.

[17] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of SIGMOD*, pages 71–79, 1995.

[18] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of SIGMOD*, pages 43–54, 2008.

[19] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *Proceedings of GIS*, pages 94–100, 2002.

[20] S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *Proceedings of GIS*, pages 9–16, 2003.

[21] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proceedings of VLDB*, pages 287–298, 2002.

[22] M. L. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):820–833, 2005.

[23] J. S. Yoo and S. Shekhar. In-route nearest neighbor queries. *GeoInformatica*, 9(2):117–137, 2005.