Best Point Detour Query in Road Networks

Shuo Shang, Ke Deng and Kexin Xie
School of Information Technology and Electrical Engineering
The University of Queensland, Australia
{shangs, dengke, kexin}@itee.uq.edu.au

ABSTRACT

A *point detour* is a temporary deviation from a user preferred path P (not necessarily a shortest network path) for visiting a data point such as a supermarket or McDonald's. The goodness of a point detour can be measured by the additional traveling introduced, called *point detour cost* or simply *detour cost*. Given a preferred path to be traveling on, *Best Point Detour* (BPD) query aims to identify the point detour with the minimum detour cost. This problem can be frequently found in our daily life but is less studied. In this work, the efficient processing of BPD query is investigated with support of devised optimization techniques. Furthermore, we investigate continuous-BPD query with target at the scenario where the path to be traveling on continuously changes when a user is moving to the destination along the preferred path. The challenge of continuous-BPD query lies in finding a set of update locations which split P into partitions. In the same partition, the user has the same BPD. We process continuous-BPD query by running BPD queries in a deliberately planned strategy. The efficiency study reveals that the number of BPD queries executed is optimal. The efficiency of BPD query and continuous-BPD query processing has been verified by extensive experiments.

Categories and Subject Descriptors

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

General Terms

Algorithms

Keywords

Best point detour query, Road networks, Spatial databases

1. INTRODUCTION

When a user is moving along a preferred path P (not necessarily a network shortest path) from a source location to a destination location in road networks, a *point detour* is a temporary deviation from P for visiting a point such as a supermarket or McDonald's [1]. The detour causes extra traveling. The *detour distance* is the network shortest path distance from the exit off P, called *out-exit*, to the data point and then to the exit back to P, called *in-exit*. A natural problem is to find the detour which introduces the minimum extra traveling and this problem is motivated particularly by the proliferation of GPS-enabled location based services.

In the detour planning of real scenarios, using different exits is allowed. However, the most relevant works only studied a special case of this problem which forces users to use the same out- and in-exit in a detour. The special case of the problem is known as *Path Nearest Neighbor* query (PNN) [1] and *In-route Nearest Neighbor* query (IRNN) [16, 18]. In [16, 18, 1], it is pragmatically assumed that users (e.g. commuters) prefer to follow the route they are familiar with, thus they would like to choose the data point with the minimum deviation from the path, after visiting, they will return to the previous route and continue the journey. However, the utility of IRNN and PNN query is limited since users have to use the same exit for a detour. The essential objective of IRNN and PNN is to find the data point which is the closest to P. In most situations, on the other hand, what users really concern is the extra traveling introduced by the detour but not how far away the data point is from the path. The goodness of a point detour should be measured by the net increase of the traveling introduced, called *point detour cost* or simply *detour cost*. In this sense, the data point closest to P may not be a good choice. The reason is that a detour using different exits may avoid a part of traveling on P; thus offset the extra travelling for detour.

See an example in figure 1, s, t are source and destination locations. The preferred path P from s to t is indicated by the bold line; e_1, e_2, e_3 and e_4 are exits in P; o_1, o_2, o_3 and o_4 are data points. A user is currently at location c. IRNN and PNN will return the point detour $e_4 \rightarrow o_3 \rightarrow e_4$ relevant to the path to be traveling on, i.e. the path from c to t along P. The extra traveling $sd(e_4, o_3) + sd(o_3, e_4)$ is 30 where $sd(x, y)$ denotes the network shortest path distance between two locations x and y. Nonetheless, if different exits are allowed, the point detour $e_1 \rightarrow o_2 \rightarrow e_3$ will be returned and the extra traveling $sd(e_1, o_2)+sd(o_2, e_3)-d(e_1, e_3)$ is 7 where $d(x, y)$ denotes the path distance between two locations x and y along P.

[1]In this paper, the *point detour* is also called *detour*

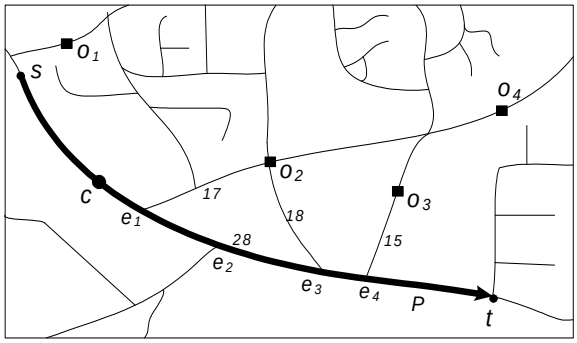

Figure 1: An example of point detour

In road networks, given a data point set O, the path to be traveling on is called *query path*, i.e. the path from user's current location to destination along the preferred path. A detour is relevant to a query path if both out- and in-exit are in the query path. The *Best Point Detour* (BPD) query searches all possible point detours relevant to the query path for the one with the minimum detour cost. The focus of this work is on the efficiency issue of BPD query processing. A number of pruning techniques are devised and an efficiency study is provided to optimize the network search. (This work reports the best point detour search but the proposed techniques are applicable to k best point detour query with direct extension.) The solution of a BPD query is valid if user's location does not change in P. However, when user is moving, the query path keeps changing and consequently the solution of a BPD query may change at some points. It is impractical to execute BPD query repeatedly to monitor the update of BPD without a premeditated approach.

This motivates the study of *Continuous Best Point Detour* (continuous-BPD) query in this work. The challenge of continuous-BPD query lies in finding a set of update locations which split P into partitions. In the same partition, the user has the same BPD. The continuous-BPD query processing is accelerated by performing BPD queries in a deliberately planned strategy. The efficiency study reveals that the number of BPD queries executed is optimal. The efficiency has been verified by extensive experiments. As a further step, this work also investigates a practical situation where the destination is too far away, but the interest of a user may be the BPD available within a certain distance δ, such as "*Find the best detour to a MacDonald's from the way* $50km$ *ahead along* P".

So far, the user is supposed to go back to the preferred path after visiting a data point and continues the journey to the destination. Alternatively, the user may chose not to go back and would like to find a new path from the current location to the destination. Both of them are common in practice. The proposed BPD query is capable to handle both situations by introducing a parameter τ, called *detour distance threshold*, which indicates the tolerance of user on the traveling not on the preferred path. In specific, the traveling off the preferred path cannot be greater than τ. If the latter situation is the choice of users, $\tau = \infty$ is specified and BPD query returns a best point detour which takes the shortest path to the destination after visiting a data point. For the former situation, τ can be specified in different ways such as a percentage of the length of the preferred path, or a factor of the deviation of IRNN (or PNN) to the preferred path. No matter what the setting of τ is, the point detour with the minimum detour cost is always returned by the proposed techniques. In this work, we suppose τ has been specified by user and our focus is on the BPD query processing under the constraint of τ.

The rest of the paper is organized as follows. Section 2 presents related work and section 3 introduces the network distance index used in this paper as well as problem definitions. After that, the BPD query processing is described in section 4. This is followed by continuous-BPD query processing in section 5. This paper is concluded in section 7 after discussion on experiments results in section 6.

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 kind of query aims to retrieve the closest neighbor to a query point from a set of given objects. Based on different constraint conditions, NN query processing can be classified into three categories, such that in Euclidean spaces (e.g. [12, 6]), in spatial networks (e.g. [9, 8, 10, 15]), and in higher dimensional spaces (e.g. [7, 3]).

As a variant of NN queries, Continuous Nearest Neighbor queries (CNN) [2, 11, 17, 14] report the kNN results continuously while the user is moving along a path. This type of queries aims to find the split points on the query path where an update of the kNN is required, and thus to avoid unnecessary re-computation. In [11], Mouratidis et al. investigate the CNN 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 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.

In-Route Nearest Neighbor Queries (IRNN) in [16, 18] 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. Recently, Path Nearest Neighbor (PNN) query proposed by Chen et al. [1] is an extension of the IRNN query to monitor the change of solution when user is moving along the predefined path and when a user gets off the predefined path and decides not to return back. To some extent, the motivation of PNN monitoring problem is similar to that of the CNN monitoring. However, PNN provide monitoring of the NN to a dynamically changing path rather than a moving query point. This is also the difference of continuous-BPD query studied in this work and CNN query.

3. PRELIMINARY

3.1 Road network

In this work, road networks are modeled by connected and undirected planar graphs $G(V, E)$, where V is the set of vertices and E is the set of edges. A weight can be assigned to each edge to represent length or application specific factor such as traveling time obtained from mining the historic traffic data [5]. Given two locations a, b in road networks, the network distance between them is the length of their shortest network path, i.e. a sequence of edges linking a and b where the accumulated weight is minimal. When weight associates factor such as traveling time, the lower bound of network distance is not necessary the corresponding Euclidean distance; thus the spatial indexes such as R-tree are not effective.

We assume the network paths between all pairs have been pre-computed and then make use of an encoding to reduce storage cost from $O(|V|^3)$ to $O(|V|^{1.5})$ [13]. This encoding takes advantage of the fact that the shortest paths from a vertex u to all of the remaining vertices can be decomposed into subsets based on the first edges on the shortest paths to them from u. The simplest way of representing the shortest path information is to maintain an array A of size $|V| \times |V|$. The $A[u, v]$ contains the first vertex on the shortest path from u to v. Using A, the construction of shortest path between u and v is performed by repeatedly visiting $A[u', v]$, $u' = u$ for the first time and $u' = A[u', v]$ in subsequence, the time is proportional to the length to the path. The precomputed network paths between all pairs are suitable to provide the network distance for two given vertices. But they are not suitable to identify the vertices which are in a certain distance to a given vertex, say u. In this situation, we apply the method to expand a wavefront starting at u as the Dijkstra's algorithm.

The data points are embedded in networks and they may be located in edges. If the network distances to the two end vertices of an edge are known, it is straightforward to derive network distance to any point in this edge. Thus, we assume that all data points are in vertices for the sake of clear description.

3.2 Problem Definition

Given any two locations a, b in road networks, the shortest network path between them is denoted as $SP(a, b)$ and the length of $SP(a, b)$ is denoted as $sd(a, b)$. Given any two locations u, v in the preferred path P (not necessarily a short network path), the path between them along P is denoted as $P(u, v)$ and the length of $P(u, v)$ is denoted as $d(u, v)$. O is a set of data points and $< e_o, o, e_i >$ is a point detour to $o \in O$ where e_o is the out-exit and e_i is the in-exit. We use a non-negative real number τ to represent *detour distance threshold*.

Definition: Point Detour Given a preferred path P and a data point set O in road networks, a point detour $< e_o, o, e_i >$ is the shortest network path between two exits $e_o, e_i \in P$ which passes the data point $o \in O$. The detour distance of $< e_o, o, e_i >$ is $sd(e_o, o) + sd(o, e_i)$. □

Definition: Point Detour Cost Given a preferred path P and a data point set O in road networks, the detour cost of a detour $< e_o, o, e_i >, e_{o,i} \in P, o \in O$, is $sd(e_o, o) + sd(o, e_i) - d(e_o, e_i)$, denoted as $< e_o, o, e_i > .dc$. □

In this work, *point detour cost* is also called *detour cost* for simplicity.

Definition: Best Point Detour Query (BPD) Given a preferred path P, a data point set O and a detour distance threshold τ, BPD(O, P, τ) returns a detour $< e_o, o, e_i >$, $e_{o,i} \in P, o \in O$, such that $< e_o, o, e_i > .dc$ is less than the detour cost of any other point detour $< e'_o, o', e'_i >$, $e'_{o,i} \in P, o' \in O$, where the detour distances of $< e_o, o, e_i >$ and $< e'_o, o', e'_i >$ are not greater than τ. □

In BPD(O, P, τ), P is the *query path* and the detour returned by BPD(O, P, τ) is the *best point detour* (BPD) relevant to P.

Definition: Continuous Best Point Detour Query (Continuous-BPD) Given a preferred path P from s to t, O and τ, continuous-BPD query returns a set of update locations in P which divide P into partitions, each partition ξ together with a detour $\xi.detour$, such that at any location c in a partition ξ, the BPD returned by BPD(O, P_{ct}, τ) is $\xi.detour$, where $P_{ct} = P(c, t)$ is the query path. □

4. BEST POINT DETOUR QUERY

Given a preferred path P from c to t, O and τ, BPD query processing takes four steps.

1. Identify candidates from O whose network distances to P are not greater than 0.5τ (section 4.1);
2. Compute detour cost lower bound for each candidate, and the candidate with the minimum detour cost lower bound is selected each time for next step. (section 4.2);
3. For each selected candidate from step 2, search the local best point detour (the best point detour to this candidate). (section 4.3).
4. Among local best point detours, the global best point detour is returned and BPD query terminates. (section 4.3).

4.1 Identify Candidate

The candidate criteria is whether a data point o has the shortest network distance to P not greater than 0.5τ, i.e. $min_{\forall e \in P}(sd(o, e)) \leq 0.5\tau$.

To find candidates, a set of vertices in P are selected as centers such that P is divided into a set of segments. If the network distances from o to the two ends e_h, e_j of a path segment $P(e_h, e_j)$ are known, the candidature of o can be determined. As the schematic example shown in figure 2, the path segment $P(e_h, e_j)$ is illustrated. Our objective is to find all data points which have network distances to $P(e_h, e_j)$ less than 0.5τ. From e_h, a browsing wavefront is expanded in road networks as Dijkstra's algorithm [4] and so does e_j. In concept, the browsed region is round as shown in figure 2 where the radius is the network distance from the center to the browsing wavefront, denoted as r. Once

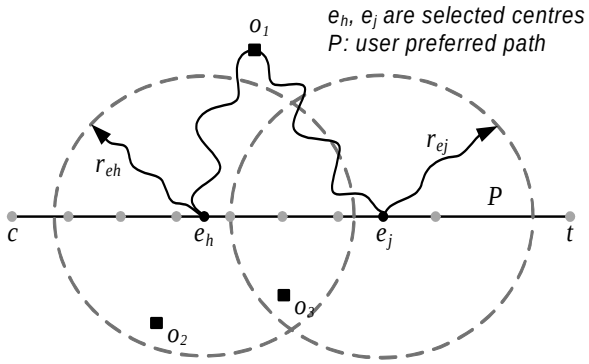

Figure 2: Candidate Criteria.

$r_{e_h} + r_{e_j} - d(e_h, e_j)$ is greater than τ, the expansion of browsing wavefronts stops. It can be proved that the data points outside the browsed region cannot have network distance to $P(e_h, e_j)$ less than τ, thus they can be pruned safely such as o_1. For the data points inside the browsed region such as $o_{2,3}$, they are candidates to the path segment $P(e_h, e_j)$.

Lemma 1: Given a set O of data points, the detour distance threshold τ and a preferred path P which is partitioned into n segments by centers, i.e. a set of selected vertices in P, the candidates to the k^{th} segment $P(e_h, e_j)$ is

$$C_k = \{o | sd(o, e_h) + sd(o, e_j) - d(e_h, e_j) \leq \tau,\ o \in O,\ e_{h,j} \in P\}. \quad (1)$$

Proof : As shown in figure 2, suppose there is a data point o with $sd(o, e_h) + sd(o, e_j) - d(e_h, e_j) > \tau$. Let e_n be the closest exit in $P(e_h, e_j)$ to o. According to triangle inequality, we have $sd(o, e_n) \geq sd(e_h, o) - sd(e_h, e_n)$ and $sd(o, e_n) \geq sd(e_j, o) - sd(e_j, e_n)$, thus $2sd(o, e_n) \geq sd(o, e_h) + sd(o, e_j) - sd(e_h, e_j)$. Since $sd(e_h, e_j)$ is network shortest path distance, $sd(e_h, e_j) \leq d(e_h, e_j)$. Thus, the relation $2sd(o, e_n) \geq sd(o, e_h) + sd(o, e_j) - d(e_h, e_j)$ is still true. From $sd(o, e_h) + sd(o, e_j) - d(e_h, e_j) > \tau$, we have $2sd(o, e_n) > \tau$. Since it cannot satisfy the constraint on detour distance, o is not a candidate to $P(e_h, e_j)$. □

For some candidates in C_k, the network distances to both e_h or e_j are known and for other candidates only the network distance to either e_h or e_j is known when the browsing wavefronts stop expanding. For example o_2 in figure 2, only the network distance to e_h is known. If network distances to $e_{h,j}$ are known, o_2 may not be a candidate according to Lemma 1. Due to the use of pre-computed network path information as discussed in section 3.1, the cost is trivial to compute network distances and examine all candidates.

To identify complete candidate set to the whole path of P, the candidates to individual segments are combined. Since one data point may be candidate to more than one path segments, such duplicate candidates are merged and purged. The complete candidates to P is

$$C = \{o | o \in \bigcup_{i=1}^{n} C_i\}. \quad (2)$$

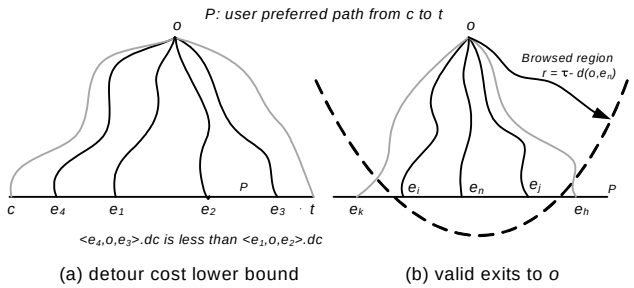

Figure 3: Detour cost lower bound and valid exits.

The selection of centers impacts the performance of BPD query processing. This issue is discussed later in section 4.4.

4.2 Detour Cost Lower Bound

We now have a set of candidates. To avoid refining all of them, their detour cost lower bounds are calculated at trivial cost. A candidate with less detour cost lower bound is more likely to be the solution, thus should be refined first. See figure 3(a), suppose o is a candidate and its network distances to four exits e_1, e_2, e_3 and e_4 in P are known. We can prove that $< e_4, o, e_3 > .dc$ is the lower bound of $< e_1, o, e_2 > .dc$. We use a lemma to describe this relationship.

Lemma 2: Given a candidate $o \in O$ and a preferred path P, $< e_1, o, e_2 > .dc$ must be greater than $< e_4, o, e_3 > .dc$ if $e_{1,2}$ are in between $e_{3,4}$ in P.

Proof : As shown in figure 3(a), $< e_4, o, e_3 > .dc = sd(e_4, o) + sd(o, e_3) - d(e_4, e_3)$ and $< e_1, o, e_2 > .dc = sd(e_1, o) + sd(o, e_2) - d(e_1, e_2)$. $< e_4, o, e_3 > .dc - < e_1, o, e_2 > .dc = (sd(e_4, o) - sd(e_1, o) - d(e_4, e_1)) + (sd(e_3, o) - sd(e_2, o) - d(e_2, e_3))$.

According to triangle inequality, it is true that $sd(e_4, o) - sd(e_1, o) - sd(e_4, e_1) \leq 0$ and $sd(e_3, o) - sd(e_2, o) - sd(e_2, e_3) \leq 0$. Since $sd(e_4, e_1)$ is network shortest path distance, it must be no more than $d(e_4, e_1)$ and thus $sd(e_4, o) - sd(e_1, o) - d(e_4, e_1) \leq 0$ is held. For the same reason, we have $sd(e_3, o) - sd(e_2, o) - d(e_2, e_3) \leq 0$. As a consequence, $< e_4, o, e_3 > .dc - < e_1, o, e_2 > .dc \leq 0$, i.e. $< e_4, o, e_3 > .dc$ is the lower bound of $< e_1, o, e_2 > .dc$. □

Suppose o in figure 3(a) is a candidate to path segment $P(e_1, e_2)$, but not to $P(c, e_1)$ and $P(e_2, t)$. We have $sd(e_1, o) + sd(o, e_2) - d(e_1, e_2) \leq \tau$, $sd(c, o) + sd(o, e_1) - d(c, e_1) > \tau$ and $sd(e_2, o) + sd(o, t) - d(e_2, t) > \tau$. Given a detour $< e_o, o, e_i >$, $e_{o,i}$ must be in $P(c, e_2)$ or in $P(e_1, t)$ in order to satisfy the constraint of τ. According to Lemma 2, if $e_{o,i} \in P(c, e_2)$, the detour cost lower bound of $< e_o, o, e_i >$ is $< c, o, e_2 > .dc$; if $e_{o,i} \in P(e_1, t)$, the detour cost lower bound of $< e_o, o, e_i >$ is $< e_1, o, t > .dc$. Since the network distances from o to $e_{1,2}$ are already known, we only need to compute the network distances from o to c, t using pre-computed network distance information. In summary, we have the following lemma:

Lemma 3: Given a preferred path P from c to t, $o \in O$ is a candidate to a path segment $P(e_1, e_2)$ where e_2 is closer to

t. The detour cost lower bound of o, denoted as $o.dclb$, can be derived:

$$o.dclb = min(< c, o, e_2 > .dc, < e_1, o, t > .dc). \quad (3)$$

Proof : As shown in figure 3(a), we assume that o is a candidate to $P(e_1, e_2)$ and not a candidate to $P(c, e_1)$ and to $P(e_2, t)$ using Lemma 1. That is, $sd(e_1, o) + sd(o, e_2) - d(e_1, e_2) \leq \tau$, $sd(c, o) + sd(o, e_1) - d(c, e_1) > \tau$ and $sd(o, e_2) + sd(o, t) - d(e_2, t) > \tau$. Given a detour $< e_o, o, e_i >$, if one of $e_{o,i}$ is in $P(c, e_1)$ and the other is in $P(e_2, t)$, $sd(e_o, o)$ and $sd(o, e_i)$ are greater than 0.5τ and thus the detour distance of $< e_o, o, e_i >$ must be greater than τ. Thus, we only consider the situation that both $e_{o,i}$ are in $P(c, e_2)$ or in $P(e_1, t)$. In case of $P(c, e_2)$ (or $P(e_1, t)$), $< c, o, e_2 > .dc$ (or $< e_1, o, t > .dc$) is the detour cost lower bound of $< e_o, o, e_i > .dc$ according to Lemma 2. Thus, the minimum of $< c, o, e_2 > .dc$ and $< e_1, o, t > .dc$ must the lower bound of $< e_o, o, e_i > .dc$. □

For all candidates in C, their detour cost lower bounds are computed and the one with the minimum value is selected each time for next step.

4.3 Search Best Point Detour

For each candidate o from last step, the local best detour to o is searched. Then among all found local best detours, the global best detour is returned and the BPD query processing terminates.

4.3.1 Valid Exits

Given a candidate o and a detour distance threshold τ, an exit $e \in P$ is valid to o only if at least one detour using e to o has detour distance no greater than τ. The task is to find all valid exits to o (at this point, only the network distances from o to few centers are known). To minimize the network scan for this purpose, we need to identify the closest exit $e_n \in P$ to o first, that is, $sd(o, e_n) < sd(o, e'), e' \in P - e_n$. Suppose e_n is used in a detour to o, the other half of the detour from o back to P (or from P to o) cannot have network distance greater than $\tau - sd(o, e_n)$ in order to satisfy constraint of τ. In specific, any exit whose network distance to o is greater than $\tau - sd(o, e_n)$ is invalid.

Lemma 4: Given a preferred path P, the valid exits to a data point $o \in O$, denoted as $o.VE$, is:

$$o.VE = \{e | sd(o, e) \leq \tau - min_{\forall e_i \in P}(sd(o, e_i)),\ o \in Q,\ e \in P\}. \quad (4)$$

Proof : We assume that $sd(o, e_n) = min_{\forall e_i \in P}(sd(o, e_i))$, $o \in O$, $e_n \in P$. For any exit $e \in P - e_n$, if $sd(o, e) > \tau - sd(o, e_n)$, we have $sd(o, e) + sd(o, e_n) > \tau$. Thus, e is not valid to o. □

By expanding a browsing wavefront from o as Dijkstra's algorithm, e_n is the first vertex in P which has the minimum value among all vertices in the wavefront, i.e. wavefront will expand next from e_n to e_n adjacent vertices. After $sd(o, e_n)$ is computed, the radius of the browsed region is no greater than $\tau - sd(o, e_n)$ according to Lemma 4. An example is shown in figure 3(b). $e_{i,j}$ are valid exits and $e_{k,h}$ are invalid exits to o.

4.3.2 Test Valid Exit Combinations

Among the valid exits of o, the pair resulting in the detour with the minimum detour cost is searched. It is costly to test all possible combinations due to complexity $|o.VE|^2$. We propose the following lemma to reduce combinations to be tested.

Lemma 5: Given a preferred path P and a data point $o \in O$, suppose e_n is the closest exit in P to o. The valid exits in P are separated by e_n into two groups $o.VE_b$ and $o.VE_a$ where $o.VE_b$ includes e_n and those exits visited earlier than e_n when user is moving from c to x along P, and $o.VE_a$ includes e_n and those exits visited later than e_n. The best point detour $< e_o, o, e_i >$ must be formed by e_o and e_i from different groups of $o.VE_b$ and $o.VE_a$.

Proof : If e_o and e_i are both in $o.VE_b$, the detour cost of $< e_o, o, e_i >$ is $sd(e_o, o) + sd(o, e_i) - d(e_o, e_i)$ which is greater than $sd(e_o, o) + sd(o, e_n) - d(e_o, e_n)$ according to Lemma 2. So $< e_o, o, e_i >$ can not be the best point detour of o. It is same if e_o and e_i are both in $o.VE_a$. Therefore, e_o and e_i must be from different groups of $o.VE_b$ and $o.VE_a$. □

From Lemma 5. the local best detour to o can be found by testing all pairs of exits, one from $o.VE_a$ and the other from $o.VE_b$. In our method, such all pair test is prevented. We test exits in $o.VE_b$ one by one according to the distance to e_n. The farther one is tested earlier. For each exit e_o in $o.VE_b$, it forms pairs with the exits in $o.VE_a$ in the order from the one far from e_n to the one close to e_n and this process stops once the first exit pair satisfies the constraint of τ; this exit pair forms the best detour to o using e_o, denoted as $e_o.BPD$; the correctness can be proved by applying Lemma 2. Among all $e \in o.VE_b$, the $e.BPD$ with minimum detour cost is the local best detour to o we are searching for.

So far, we calculated local best detour to one candidate. Any candidates in C can be pruned if they have detour cost lower bounds greater than the detour cost of this local best detour. From the remaining candidates in C, the one with the minimum lower bound is refined in this step. The termination condition of BPD query is when no candidate is left in C. Before termination, the one with the minimum detour cost among all computed local best detours is the global best detour and is returned. In addition, when the BPD with the second minimum detour cost is required, the above method can be easily adapted.

4.4 Efficiency Issues

Suppose data points are uniformly distributed in the networks and the exits in P are uniformly distributed as well. We now analyze BPD complexity by estimating the cost in each step. In the first step, the network browsing is performed by centers to identify candidates. In a given network $G(V, E)$, the complexity is $O(n(Vlg(V) + E))$ where n is the number of centers. The second step calculates the detour cost lower bound for each candidate. The cost is $O(y * m)$ where y is the number of candidates and m is the number of vertices in network path from each candidate to c and t. In the third step, the network is browsed from selected candidate to identify the valid exits in P and local best detour is refined by testing combinations of valid exits. The cost is

$O(x*(Vlg(V)+E)+x*(\tau/\rho)^2)$ where the constant ρ is the density of exits in query path P and x is the number of candidates to be refined. By considering above all, BPD query processing algorithm is in time complexity $O(n\cdot C_1+y\cdot C_2)$ where $C_1=Vlg(V)+E$ and $C_2=m+Vlg(V)+E+(\tau/\rho)^2$. Clearly, the number of centers and the number of candidates determine the overall cost of BPD query processing.

In one extreme case, every vertex in P is a center. The candidate set is minimized but the number of centers is maximized. Since both candidates and centers need to browse the networks and these operations dominate the BPD query processing, the overall performance maybe ruined, in particular in case of sparse data points. In the other extreme case, only two ends of P, i.e. source and destination, are selected as centers (as does in [1]). While the number of centers is minimized the candidate set may be very large. The optimal selection of centers can be estimated using linear programming. Suppose $\{e_1=c, e_2.., e_{n-1}, e_n=t\}$ are exits in path P in the order from c to x. Let A be a $n\times n$ matrix where the i^{th} column corresponds to e_i, so does the i^{th} row. In A, the entry $a_{ij}=\{1,0\}$; $a_{ij}=1$ if i^{th}, j^{th} exits are adjacent centers, i.e. no exits in between them are centers, $a_{ij}=0$ otherwise. Our goal is to minimize the objective function

$$\omega=\sum(a_{ij}\frac{1}{4}\pi(d_{ij}+\tau)^2\lambda)+\sum a_{ij}. \quad (5)$$

subject to $i<j$, $\sum_{j=1}^{n} a_{0j}=1$, $\sum_{i=1}^{n} a_{i0}=1$, $\sum_{j=1}^{n} a_{ij}\leq 1$, $\sum_{i=1}^{n} a_{ij}\leq 1$, and $\sum_{j=1}^{n} a_{ij}=\sum_{k=1}^{n} a_{ki}$. λ is a constant to indicate the average number of data points per unit area in space. We use $\sum(a_{ij}\frac{1}{4}\pi(d_{ij}+\tau)^2\lambda)$ to estimate the number of candidates and $\sum a_{ij}$ to estimate the number of centers. Considering the online processing, the time cost to find optimal selection of centers is not practical by solving the objective function. Thus, we simplify the objective function (5) by assuming the gaps between between adjacent centers are equal and the vertices in P are uniformly distributed. Our aim is changed to find the optimal number of centers. Then, we have

$$\omega(n)=n(\frac{1}{4}\pi(\frac{P.l}{n}+\tau)^2\lambda)+n. \quad (6)$$

where $P.l$ is the length of P and n is the number of centers. $P.l$ is a constant and n is the only independent variable of the objective function. The n resulting in the minimum ω can be found using the derivative of the function (6)

$$\omega(n)'=\frac{\partial\omega}{\partial n}=0.\Rightarrow 2n^3-\pi\lambda\tau nP.l-\pi\lambda P.l^2=0. \quad (7)$$

This cubic function can be directly solved by applying general formula of roots. Then, n uniformly distributed locations (including the source and destination) are selected in P as the centers. If a selected location is not on an exit, an auxiliary point is inserted as a virtual exit.

5. CONTINUOUS-BPD QUERY

The solution of a BPD query is relevant to the query path which is from user's current location to the destination along the preferred path. If user's location does not change, the solution is valid. However, when user is moving, the query path keeps changing and consequently the solution of a BPD query may change at some point. It is impractical to execute BPD query repeatedly to monitor the update of BPD without a deliberate scheme. This motivates the continuous-BPD Query.

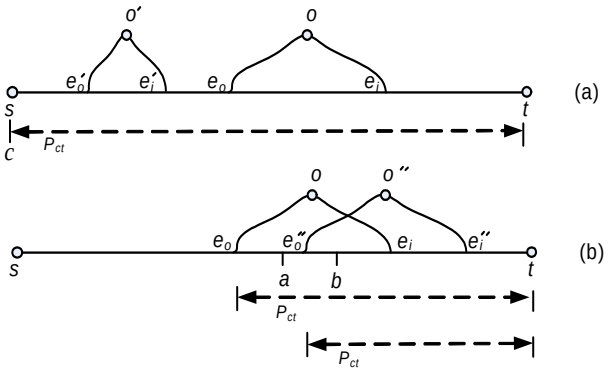

Figure 4: In the case of query path P_{ct}.

Continuous-BPD query is processed by running a BPD query first where $P_{ct}=P(s,t)$. Suppose $<e_o,o,e_i>$ is the BPD returned by BPD query. Any other detour using out-exit before e_o is impossible to replace $<e_o,o,e_i>$ as a BPD when a user is moving along P from s to t. Thus, such detours can be pruned safely. To prove this, let $<e_o',o',e_i'>$ be a detour to o' (equal to o or not) where e_o' is an exit before e_o, and e_i' can be any exit. An example is shown in figure 4(a). Since $<e_o,o,e_i>$ is the best detour, $<e_o',o',e_i'>.dc$ must be greater than $<e_o,o,e_i>.dc$. When a user is moving from s to e_o', $<e_o',o',e_i'>.dc$ remains the same. When the user passes e_o' before reaching e_o, $<e_o',o',e_i'>.dc$ increases since the user has to turn way back in order to use detour $<e_o',o',e_i'>$. At the same time, $<e_o,o,e_i>.dc$ remains the same. Thus, $<e_o,o,e_i>$ is still the best point detour. That is, no update location exists from s to e_o and the detours with out-exit before e_o can be pruned.

Then, the best detour relevant to P_{ct} starting at e_o (i.e. $P_{ct}=P(e_o,t)$) is searched by executing a BPD query, see the example in figure 4(b). Let $<e_o'',o'',e_i''>$ be such best detour. Our aim is to find the update location by passing which $<e_o'',o'',e_i''>$ replaces $<e_o,o,e_i>$ to be BPD. As discussed above, $<e_o,o,e_i>.dc$ increases when the user passes e_o and moves forward. Once $<e_o,o,e_i>.dc$ is greater than $<e_o'',o'',e_i''>.dc$, the current location of user is the update location, i.e. the location $0.5(<e_o'',o'',e_i''>.dc-<e_o,o,e_i>.dc)$ after e_o. However, the update location found by this way is not always valid.

See an example in figure 4(b), if the update location is in between e_o and e_o'' such as a, it is valid; if the update location is after e_o'' such as b, it is invalid. Let us see why b is invalid. As discussed above, $0.5(<e_o'',o'',e_i''>.dc-<e_o,o,e_i>.dc)$ is the difference from e_o to b, i.e. $d(e_o,b)$. Since $d(e_o,e_o'')\leq d(e_o,b)$ and $d(e_o,b)=0.5(<e_o'',o'',e_i''>.dc-<e_o,o,e_i>.dc)$, we have $2d(e_o,e_o'')+<e_o,o,e_i>.dc$ is no greater than $<e_o'',o'',e_i''>.dc$, in specific, $<e_o'',o'',e_i''>.dc$ is greater than $<e_o,o,e_i>.dc$ when user is at location e_o''. This is also true when user passes e_o'' and moves forward to b since the detour costs for both increases by $d(e_o'',b)$. Thus, when user passes b, $<e_o'',o'',e_i''>$ cannot replace $<e_o,o,e_i>$ to be BPD. So, b is an invalid update location. If the update

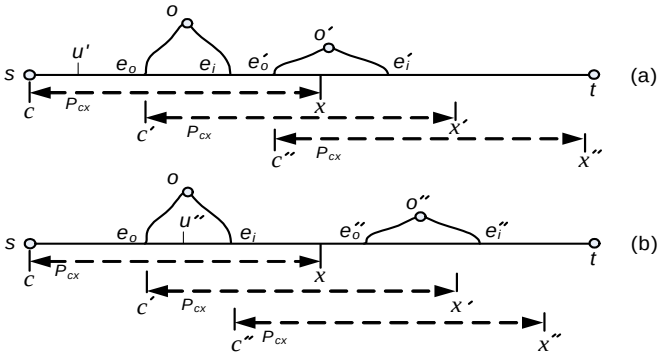

Figure 5: In the case of query path P_{cx}.

location such as b is invalid, the detour $< e''_o, o'', e''_i >$ is discarded and we examine the second best detour relevant to current P_{ct} (i.e. $P(e_o, t)$).

If the update location such as a is valid, $< e''_o, o'', e''_i >$ is the BPD after a. It is interesting to note that no other update location exists in between e_o and a. This can be proved by contradiction. Suppose a detour $< e*_o, o*, e*_i >$ relevant to $P_{ct} = P(e_o, t)$ has a valid update location $a*$ which is before a. That is, when the user passes $a*$, the best detour is $< e*_o, o*, e*_i >$. This happens only if $< e''_o, o'', e''_i > .dc$ is greater than $< e*_o, o*, e*_i > .dc$. This conflicts with the situation that $< e''_o, o'', e''_i > .dc$ is BPD relevant to $P_{ct} = P(e_o, t)$. In summary, once we determine the update location such as a of $< e''_o, o'', e''_i >$ is valid, it is the first update location. This update location is recorded in the solution together with $< e''_o, o'', e''_i >$. Then, we go to identify next update location in the same way as though the source is at the newly identified update location such as a, and since the BPD is already known such as $< e''_o, o'', e''_i >$, the next query path to be processed is $P(e''_o, t)$, see figure 4(b).

5.1 A Practical Situation

In this section, we study the continuous-BPD query in the situation where the destination is too far away, but the interest of a user may be the BPD available within a certain distance δ. This situation asks for techniques additional to the typical continuous-BPD query processing. Since the query path P_{ct} is always from user's current location c to destination t, the best point detour $< e_o, o, e_i >$ relevant to P must be the best point detour relevant to P_{ct} until user passes e_o. This is not true to the query path P_{cx} which is from user's current location c to x, the location δkm ahead along P, and $d(c, t) > \delta$. That is, only the detours not far from user's current location are considered.

When user's current location is s, a BPD query is executed for the BPD relevant to P_{cx}. As shown in the example in figure 5(a), $< e_o, o, e_i >$ is the BPD. When user arrives at e_o, P_{cx} changes from $P(c, x)$ to $P(c', x')$ and a BPD query is executed for query path $P(c', x')$. Different from the case of P_{ct}, update location may exist in between s and e_o. This is because some detours such as $< e'_o, o', e'_i >$ initially irrelevant to $P_{cx} = P(c, x)$ becomes relevant when P_{cx} changes to $P(c', x')$. Suppose $< e'_o, o', e'_i > .dc$ is less than $< e_o, o, e_i > .dc$. For $< e'_o, o', e'_i >$, the update location is the location u' which is $d(e'_i, x')km$ before e_o. When user passes u', $< e'_o, o', e'_i >$ turns to be relevant to the query path and replaces $< e_o, o, e_i >$ to be the BPD. If several detours have update locations in between s to e_o, they all need to be identified by the same BPD query and the one with the earliest update location (i.e. the update location visited first by user when moving from s to t along P) is recorded in solution together with the associated update location. Suppose the update location u' of $< e'_o, o', e'_i >$ is the earliest. Then the next update location is searched as if u' is the source, and since the BPD $< e'_o, o', e'_i >$ is already known, the next query path to be processed is $P(c''(= e'_o), x'')$, see figure 5(a).

If no update location can be found in between s and e_o, we execute a BPD query for query path $P(c''(= e_i), x'')$ as shown in figure 5(b), and let the BPD be $< e''_o, o'', e''_i >$. Two situations need to be considered. In the first situation, $< e''_o, o'', e''_i > .dc$ is less than $< e_o, o, e_i > .dc$. For $< e''_o, o'', e''_i >$, the update location u'' is $d(e'', x'')km$ before e_i. In the second situation, $< e''_o, o'', e''_i > .dc$ is greater than $< e_o, o, e_i > .dc$. For $< e''_o, o'', e''_i >$, the update location u'' is $d(e'', x'')km$ before e_i (after this location, $< e''_o, o'', e''_i >$ becomes relevant to query path), or $0.5(< e''_o, o'', e''_i > .dc - < e_o, o, e_i > .dc)km$ after e_o (This is the situation that user already passed e_o and need turn way back to e_o in order to use $< e_o, o, e_i >$, thus $< e_o, o, e_i > .dc$ increases. If user reaches location $0.5(< e''_o, o'', e''_i > .dc - < e_o, o, e_i > .dc)km$ after e_o, $< e''_o, o'', e''_i > .dc$ will be less than $< e_o, o, e_i > .dc$). From these two locations, the later one (the location closer to destination) is u''.

Now, we discuss the situation that there is no detour satisfying τ constraint for a query path. Suppose the first such query path P_{cx} is $P(s, x)$. In this situation, we will execute the BPD query for next query path which is starting at x. This is repeated until the first detour(s) satisfying τ constraints. Let $< e_o, o, e_i >$ be the BPD. As discussed above, a BPD query is executed for the next query path P_{cx} starting at e_o and e_i (as in figure 5(b)). If no detour satisfying τ constraint except $< e_o, o, e_i >$, we may need to find the location, say s', after e_i. If P_{cx} starts at s', the detour distance of $< e_o, o, e_i >$ would fail to satisfy the τ constraint. This is because that the detour distance begins counting when user leaves the query path and stops until user returns back, i.e. the detour cost of $< e_o, o, e_i >$ is $d(s', e_o) + sd(e_o, o) + sd(o, e_i) + d(e_i, s')$. s' is an update location. Before s', the best point detour is $< e_o, o, e_i >$, and after s', we start to search a new best point detour for query path starting at s'.

When the current location c is close to the destination and $d(c, t) \leq \delta$, the query path changes to P_{ct} and the techniques discussed in section 5 are applied.

5.2 Efficiency Issues

Theorem 1: The continuous-BPD query is answered by searching BPDs relevant to a number of query paths; for each query path, a BPD query is executed. The number of the query paths to be processed (same as the number BPD queries to be executed) is optimal, i.e. errors may be introduced if the BPD relevant to any such query path is

missed.

Proof : First, we prove the number of query paths to be processed in the case of P_{ct} is optimal. The BPD queries are executed for query path starting at every out-exit of each BPD in solution and the source location, see figure 4. Suppose the query path $P(e_o, t)$ is not processed. Thus, the relevant BPD $< e''_o, o'', e''_i >$ is unknown. The possibility exists that the out-exit of the relevant BPD is extremely close to e_o and the update location such as a is also extremely close to e_o. Thus, if the query path $P(e_o, t)$ is not processed, this situation can not be surely avoided. Second, we can prove the number of query paths to be processed in the practical situation discussed in section 5.1 is optimal in the similar way. □

Thus, the cost of continuous-BPD query processing is based on the efficiency of BPD query processing. In the above discussion, BPD query algorithm is invoked and executed from scratch whenever it is necessary. A drawback is that the same data points and centers are potentially processed repeatedly. An improvement can be achieved by executing a single BPD query and store the intermediate results for subsequent processing. To do that, continuous-BPD query first runs a BPD query for the preferred path P from s to t. All candidates as well as the information after the refinement are recorded. The situation in case of P_{ct} is straightforward. Whenever the best detour is required, the stored information is used for quick response. The stored information about the detour is discarded once it is with invalid update location or with out-exit before the out-exit of the current best detour.

In the practical situation as discussed in section 5.1, the continuous-BPD query processing is started by processing a BPD query where the query path is the preferred path P from s to t. Instead of searching the BPD relevant to P, the execution pauses after the first step as described in section 4 and a set of candidates are stored. Each time when the BPD relevant to P_{cx} is requested, we only examine the candidate data points whose network distances to P_{cx} less than 0.5τ using the method similar to Lemma 1. Among the qualified candidates, the one with the minimum detour cost lower bound is selected and processed in the third step. Given a candidate o in the third step, the valid exits are searched from P_{cx}; then the o's local best detour is found and eventually the global BPD relevant to P_{cx} is identified.

6. EXPERIMENTS

In this section, we conduct experiments on data sets of California Road Network (CRN) and City of Oldenburg Road Network (ORN)[2], which contain 21,048 vertices and 6,105 vertices respectively and are stored as adjacency lists. The length of each road segment is derived using the positions of the end vertices specified in geographic coordinate. All algorithms are implemented in Java and tested on a windows platform with Intel Core2 CPU (2.13GHz) and 2GB memory. In real scenarios, the number of data points such as a supermarket and MacDonald's are much less than vertices of the underlying networks. Thus, 10% of the vertices from road network are randomly selected to constitute the set O of data points. For two reasons, the networks reside in memory in our experiments for running Dijkstra's algorithm. First, the storage occupied by CRN/ORN in memory is less than 1MB which is a quite small even for most handhold devices today; second, the main purpose of experiments is to compare the trend of the performance between ours and the simple algorithm; such comparison is independent of network residence (i.e. memory-based or disk-based). We also construct all-pair shortest path index as [13]. The experiment results shown are average of 20 independent testes with different query paths. The main metric we adopt is CPU time. The parameter settings are listed in table 1.

[2]www.cs.fsu.edu/ lifeifei/SpatialDataset.htm

Table 1: Parameter setting

Name	CRN (km)	ORN (km)
detour distance threshold τ	4-40/default 12	2-20/default 10
query path for BPD P_{ct}	100-600/default 250	40-100/default 80
preferred path for continuous-BPD P	250	80
query path length for continuous-BPD δ	50-250/default ∞	20-80/default ∞

6.1 Best Point Detour Query

An algorithm based on simple extension of PNN [1] is also implemented, called EPNN. EPNN takes four steps. First, all candidates satisfying the detour distance threshold constraint are identified using bi-direction network expansion method (i.e. the special case of the method discussed in section 4.1 where only source and destination are centers). To be fair, EPNN does not scan the entire networks as in [1]. Second, EPNN calculates the network distance from each candidate o to P by Dijkstra's algorithm to find the closest exit e_n to o. Third, at both side of e_n in P, we make a range query to find all exits whose network distances to e_n is less than τ. The exits before e_n form a set $o.VE_b$ and the exits after e_n form a set $o.VE_a$. EPNN only examines the detours which are formed by out-exit from $o.VE_b$ and in-exit from $o.VE_a$. The local BPD detour to o is identified. When all candidates have been refined, the global BPD is returned by EPNN.

6.1.1 *Effect of detour distance threshold τ*

In figure 6(a) 6(b), the experiment results demonstrate the effect of τ to the detour cost of BPD and PNN-detour (recall the PNN-detour not allowing different exit while Best point detour (BPD) allowing different exits). The detour cost of PNN-detour is constant at varying settings of τ since it always is $2d(O, P)$ (i.e. $2min_{\forall o \in O, \forall e \in P}(sd(o, e))$); in contrast, BPD has decreasing detour cost. In figure 6(a), the detour cost of PNN-detour is $8km$. When $\tau = 12km$, the detour cost of BPD is $4km$, 50% improvement to that of PNN-detour. When $\tau = 32km$, the detour cost of BPD is $1km$, eight times improvement to that of PNN-detour. The similar trend appears in figure 6(b) on ORN. Figure 6(c) 6(d) show the CPU time used by BPD and EPNN. The longer τ means more candidates since more data points can satisfy the τ constraint. Thus, the CPU time increases when τ increases.

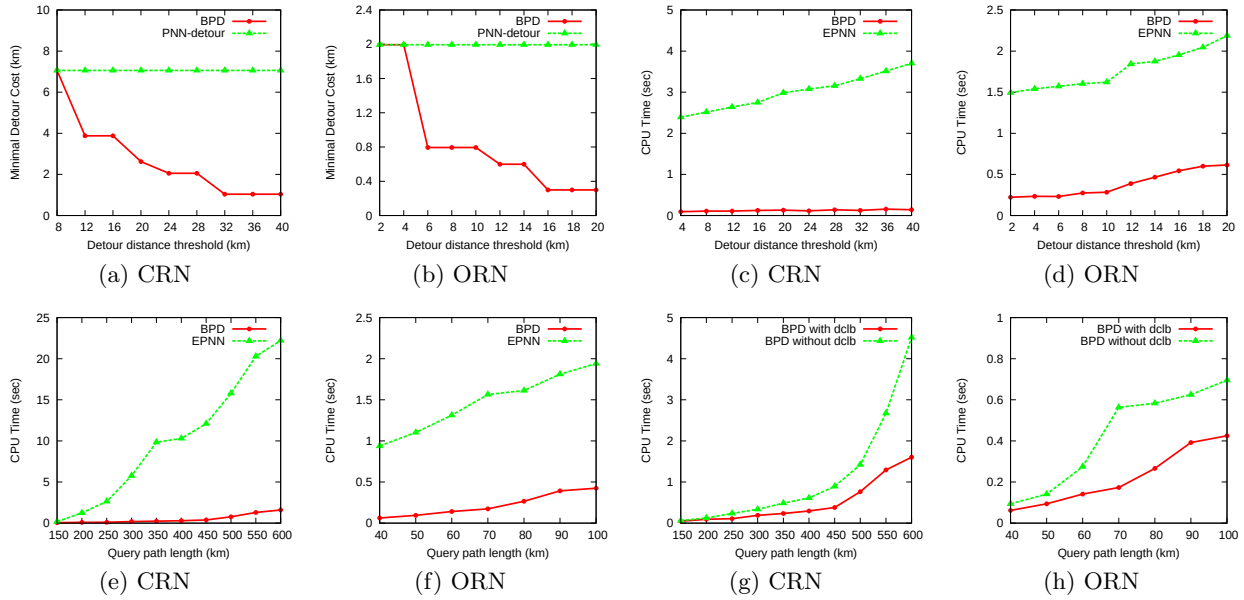

Figure 6: Experiment results of BPD query processing.

6.1.2 *Effect of query path length*

In figure 6(e) 6(f), the performance of BPD and EPNN is tested when the length of query path varies. Since the longer query path causes more data points to be processed, thus the CPU time used is expected to be longer for both BPD and EPNN. This point has been verified in the experiment results. At a given setting of query path length, the CPU time used by EPNN is much longer than BPD for two reasons. The first is due to the much larger number of candidates by using bi-direction network expansion method (as discussed in section 4.4), and the second is that EPNN has to refine all candidates since no detour cost lower bound is used. As a consequence, the performance of BPD beats that of EPNN by almost one order of magnitude. To a great extent, this result demonstrates the importance of smart selection of centers and the necessariness of detour cost lower bound.

6.1.3 *Effect of detour cost lower bound*

This experiment tests the effect of detour cost lower bound (dclb) to the performance. The dclb is used to prune candidates and to ensure the candidates are refined in proper order. We run BPD with and without support of dclb and the results are shown in figure 6(g) 6(h). We can see the performance is accelerated by 2-4 times by using dclb.

6.2 Continuous-BPD Query

The continuous-BPD can be processed by executing BPD query in two ways as discussed in section 5. In the first way, BPD query is executed only once and the intermediate results are stored for subsequent processing and in the second way, BPD query is invoked and executed from scratch whenever it is necessary. In figure 7(a) 7(b) 7(c) 7(d), the former is denoted as continuous-BPD-I and the latter is denoted as continuous-BPD-II. As exhibited in figure 7(a) 7(b), the continuous-BPD-I performs always better than continuous-BPD-II by around three times. When the query path length

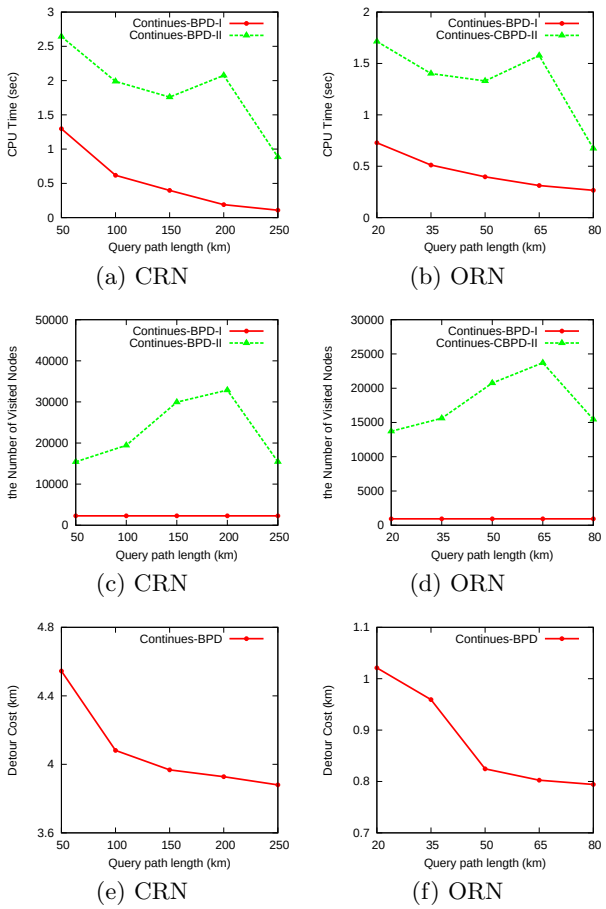

Figure 7: Experiment results of continuous-BPD query processing.

increases, the performance is getting better since the number of BPD queries to be processed tends to decrease (i.e. we need search BPD for less query paths). One interesting phenomenon is that the performance of continuous-BPD-II has a hump. The reason is that longer query path causes more data points to be processed in each BPD query. So, even though the number of BPD queries to be executed decreases, the overall effect is a worse performance at some settings of query path length.

Figure 7(c) 7(d) demonstrate the total number of network vertices accessed during the continuous-BPD query processing. The continuous-BPD-I has constant low network access since it runs the first step of BPD once only and thus access the network once. In contrast, the continuous-BPD-II accesses network whenever it is necessary such that same region may be touched repeatedly. In addition, continuous-BPD-II has a hump in the network access. The reason is same as the hump in performance.

We also test the detour cost of BPD when query path length changes. Since more choices are available in case of a longer query path, the detour cost should tend to decrease. This is verified by the experiment results in figure 7(e) 7(f). The memory cost for storing intermediate result in continuous-BPD-I is only in level of kilobytes in our experiments and thus can be ignored.

7. CONCLUSION

A point detour is a temporary deviation from a user preferred path P (not necessarily a shortest network path) for visiting a data point. In most real scenarios, different exits are allowed and what users concern is the extra traveling distance introduced. This work aims to efficiently find the best detour point (i.e. the one introducing the minimum extra traveling) on the path to be traveling on along P. Efficient techniques have developed to identify BPD and further divide P into partitions by a set of update locations. In the same partition, user has the same BPD. The efficiency studies and experiments have verified the superiority of the proposed techniques.

8. REFERENCES

[1] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring path nearest neighbor in road networks. In *Proceedings of SIGMOD*, pages 591–602, 2009.

[2] 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.

[3] K. Deng, X. Zhou, H. T. Shen, K. Xu, and X. Lin. Surface k-nn query processing. In *Proceedings of ICDE*, page 78, 2006.

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

[5] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. Sondag. Adaptive fastest path computation on a road network: A traffic mining approach. In *Proceedings of VLDB*, pages 794–805, 2007.

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

[7] H. Jagadish, B. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbour search. *TODS*, 30(2):364–397, 2005.

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

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

[10] 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.

[11] 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.

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

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

[14] 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 ACM GIS*, pages 94–100, 2002.

[15] M. Sharifzadeh, M. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. In *VLDB Journal*, pages 765–787, 2008.

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

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

[18] J. S. Yoo and S. Shekhar. In-route nearest neighbor queries. In *GeoInformatica*, volume 9, pages 117–137, 2005.