Efficient K-Nearest Neighbor Search in Time-Dependent Spatial Networks

3 authors, including:

Farnoush Banaei-Kashani
University of Colorado
116 PUBLICATIONS   **1,419** CITATIONS

Cyrus Shahabi
University of Southern California
575 PUBLICATIONS   **14,827** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   Spatial Crowdsourcing: A Survey and Taxonomy of a Growing Field View project

Project   Sensor stream and hand gesture recognition View project

Efficient K-Nearest Neighbor Search in Time-Dependent Spatial Networks *

Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi

University of Southern California
Department of Computer Science
Los Angeles, CA 90089-0781
`[demiryur, banaeika, shahabi]@usc.edu`

Abstract. The class of k Nearest Neighbor (kNN) queries in spatial networks has been widely studied in the literature. All existing approaches for kNN search in spatial networks assume that the weight (e.g., travel-time) of each edge in the spatial network is constant. However, in real-world, edge-weights are time-dependent and vary significantly in short durations, hence invalidating the existing solutions. In this paper, we study the problem of kNN search in time-dependent spatial networks where the weight of each edge is a function of time. We propose two novel indexing schemes, namely Tight Network Index (TNI) and Loose Network Index (LNI) to minimize the number of candidate nearest neighbor objects and, hence, reduce the invocation of the expensive fastest-path computation in time-dependent spatial networks. We demonstrate the efficiency of our proposed solution via experimental evaluations with real-world data-sets, including a variety of large spatial networks with real traffic-data.

1   Introduction

Recent advances in online map services and their wide deployment in hand-held devices and car-navigation systems have led to extensive use of location-based services. The most popular class of such services is k-nearest neighbor (kNN) queries where users search for geographical points of interests (e.g., restaurants, hospitals) and the corresponding directions and travel-times to these locations. Accordingly, numerous algorithms have been developed (e.g., [20, 15, 19, 2, 13, 16, 22]) to efficiently compute the distance and route between objects in large road networks.

The majority of these studies and existing commercial services makes the simplifying assumption that the cost of traveling each edge of the road network is constant (e.g., corresponding to the length of the edge) and rely on pre-computation of distances in the network. However, the actual travel-time on road networks heavily depends on the traffic congestion on the edges and hence is a function of the time of the day, i.e.,

travel-time is *time-dependent*. For example, Figure 1 shows the real-world travel-time pattern on a segment of I-10 freeway in Los Angeles between 6AM and 8PM on a weekday. Two main observations can be made from this figure. First, the arrival-time to the segment entry determines the travel-time on that segment. Second, the change in travel-time is significant and continuous (not abrupt), for example from 8:30AM to 9:00AM, the travel-time of this segment changes from 30 minutes to 18 minutes (40% decrease). These observations have major computation implications: the fastest path from a source to a destination may vary significantly depending on the departure-time from the source, and hence, the result of spatial queries (including kNN) on such dynamic network heavily depends on the time at which the query is issued.

Figure 2 shows an example of time-dependent kNN search where an ambulance is looking for the nearest hospital (with least travel-time) at 8:30AM and 2PM on the same day on a particular road network. The time-dependent travel-time (in minutes) and the arrival time for each edge are shown on the edges. Note that the travel-times on an edge changes depending on the arrival time to the edge in Figures 2(a) and 2(b). Hence, the query issued by the ambulance at 8:30AM and 2PM would return different results.
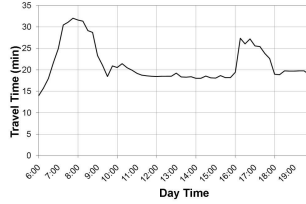

(a) 1-NN Query at 8:30 AM   (b) 1-NN Query at 2:00 PM

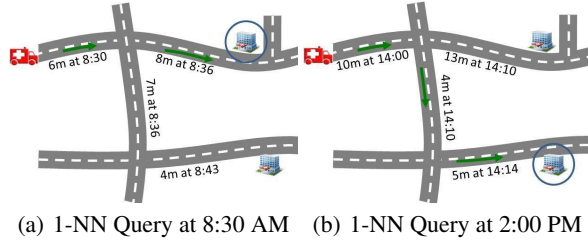**Fig. 1.** Real-world travel-time   **Fig. 2.** Time-dependent 1-NN search

Meanwhile, an increasing number of navigation companies have started releasing their time-dependent travel-time information for road networks. For example, Navteq [17] and TeleAtlas [21], the leading providers of navigation services, offer traffic flow services that provide time-dependent travel-time (at the temporal granularity of as low as five minutes) of road network edges up to one year. The time-dependent travel-times are usually extracted from the historical traffic data and local information like weather, school schedules, and events. Based on Navteq's analysis, the time-dependent weight information improves the travel-time accuracy by an average of 41% when compared with typical speeds (time-independent) on freeways and surface streets. Considering the availability of time-dependent travel-time information for road networks on the one hand and the importance of time-dependency for accurate and realistic route planning on the other hand, it is essential to extend existing literature on spatial query processing and planning (such as kNN queries) in road networks to a new family of time-dependent query processing solutions.

Unfortunately, once we consider time-dependent edge weights in road networks, all the proposed kNN solutions assuming constant edge-weights and/or relying on distance precomputation would fail. However, one can think of several new baseline solutions. Firstly, Dreyfus [7] has studied the relevant problem of time-dependent shortest path

planning and showed that this problem can be solved by a trivially-modified variant of any label-setting (e.g., Dijkstra) static shortest path algorithm. Consequently, we can develop a primitive solution for the time-dependent kNN problem based on the incremental network expansion (INE [19]) approach where Dreyfus's modified Dijkstra algorithm is used for time-dependent distance calculation. With this approach, starting from a query object q all network nodes reachable from q are visited in order of their time-dependent travel-time proximity to q until all k nearest objects are located (i.e., blind network expansion). However, considering the prohibitively high overhead of executing blind network expansion particularly in large networks with a sparse (but perhaps large) set of data objects, this approach is far too slow to scale for real-time kNN query processing. Secondly, we can use time-expanded graphs [9] to model the time-dependent networks. With time-expanded graphs the time domain is discretized and at each discrete time instant a snapshot of the network is used to represent the network. With this model, the time-dependent kNN problem is reduced to the problem of computing the minimum-weight paths through a series of static networks. Although this approach allows for exploiting the existing algorithms for kNN computation on static networks, it often fails to provide the correct results because the model misses the state of the network between any two discrete time instants. Finally, with a third approach we can precompute time-dependent shortest paths between all possible sources and destinations in the network. However, shortest path precomputation on time-dependent road networks is challenging. Because, the shortest path on time-dependent networks (i.e., a network where edge weights are function of time) depends on the departure time from the source, and therefore, one needs to precompute all possible shortest paths for *all possible departure-times*. Obviously, this is not a viable solution because the storage requirements for the precomputed paths would quickly exceed reasonable space limitations. With our prior work [4], for the first time we introduced the problem of *Time-Dependent k Nearest Neighbor* (TD-kNN) search to find the kNN of a query object that is moving on a *time-dependent network*. With this work, we also investigated the first two baseline approaches discussed above (the third approach is obviously inapplicable) by extensive experiments to rigorously characterize the inefficiency and inaccuracy of the two baseline solutions, respectively.

In this paper, we address the disadvantages of both baseline approaches by developing a novel technique that efficiently and accurately finds kNN of a query object in time-dependent road networks. A comprehensive solution for TD-kNN query should a) efficiently answer the queries in (near) real-time in order to support moving object kNN search on road networks, b) be independent of density and distribution of the data objects, and c) effectively handle the database updates where nodes, links, and data objects are added or removed. We address these challenges by developing two types of complementary index structures. The main idea behind these index structures is to localize the search space and minimize the costly time-dependent shortest path computation between the objects hence incurring low computation costs. With our first index termed *Tight Network Index (TNI)*, we can find the nearest objects without performing any shortest path computation. Our experiments show that in 70% of the cases the nearest neighbor can be found with this index. For those cases that the nearest objects cannot be identified by TNI, our second index termed *Loose Network Index (LNI)* allows us to fil-

ter in only a small number of objects that are potential candidates (and filter out the rest of the objects). Subsequently, we only need to perform the shortest path computation only for these candidates. Our TD-kNN algorithm consists of two phases. During the first phase (off-line), we partition the spatial network into subnetworks (cells) around the data objects by creating two cells for each data object called *Tight Cell (TC)* and *Loose Cell (LC)* and generate TNI and LNI on these cells, respectively. In the second phase (online), we use TNI and LNI structures to immediately find the first nearest neighbor and then expand the search area to find the remaining *k-1* neighbors.

The remainder of this paper is organized as follows. In Section 2, we review the related work on both kNN and time-dependent shortest path studies. In Section 3, we formally define the TD-kNN query in spatial networks. In Section 4, we establish the theoretical foundation of our algorithms and explain our query processing technique. In Section 5, we present experimental results on variety of networks with actual time-dependent travel-times generated from real-world traffic data (collected for past 1.5 years). In Section 6, we conclude and discuss our future work.

2   Related Work

In this section we review previous studies on kNN query processing in road networks as well as time-dependent shortest path computation.

2.1 kNN Queries in Spatial Networks

In [19], Papadias et al. introduced Incremental Network Expansion (INE) and Incremental Euclidean Restriction (IER) methods to support kNN queries in spatial networks. While INE is an adaption of the Dijkstra algorithm, IER exploits the Euclidean restriction principle in which the results are first computed in Euclidean space and then refined by using the network distance. In [15], Kolahdouzan and Shahabi proposed first degree *network Voronoi diagrams* to partition the spatial network to network Voronoi polygons (NVP), one for each data object. They indexed the NVPs with a spatial access method to reduce the problem to a point location problem in Euclidean space. Cho et al. [2] presented a system UNICONS where the main idea is to integrate the precomputed kNNs into the Dijkstra algorithm. Hu et al. [12] proposed a distance signature approach that precomputes the network distance between each data object and network vertex. The distance signatures are used to find a set of candidate results and Dijkstra is employed to compute their exact network distance. Huang et al. addressed the kNN problem using *Island* approach [13] where each vertex is associated to all the data points that are in radius r (so called islands) covering the vertex. With their approach, they utilized a restricted network expansion from the query point while using the precomputed islands. Recently Samet et al. [20] proposed a method where they associate a label to each edge that represents all nodes to which a shortest path starts with this particular edge. The labels are used to traverse *shortest path quadtrees* that enables geometric pruning to find the network distance. With all these studies, the edge weight functions are assumed to be constant and hence the shortest path computations and precomputations are no longer valid with time-varying edge weights. Unlike the

previous approaches, we make a fundamentally different assumption that the weight of the network edges are time-dependent rather than fixed.

2.2 Time-dependent Shortest Path Studies

Cooke and Halsey [3] introduced the first time-dependent shortest path (TDSP) solution where dynamic programming is used over a discretized network. In [1], Chabini proposed a discrete time TDSP algorithm that allows waiting at network nodes. In [9], George and Shekhar proposed a time-aggregated graph where they aggregate the travel-times of each edge over the time instants into a time series. All these studies assume the edge weight functions are defined over a finite discrete sequence of time steps $t \in t_0, t_1, .., t_n$. However, discrete-time algorithms have numerous shortcomings. First, since the entire network is replicated for every specified time step, the discrete-time methods require an extensive amount of storage space for real-world scenarios where the spatial network is large. Second, these approaches can only provide approximate results since the computations are done on discrete-times rather than in continuous time. In [7], Dreyfus proposed a generalization of Dijkstra algorithm, but his algorithm is showed (by Halpren [11]) to be true only in FIFO networks. If the FIFO property does not hold in a time-dependent network, then the problem is NP-Hard as shown in [18]. Orda and Rom [18] proposed a Bellman-Ford based solution where edge weights are piece-wise linear functions. In [6], Ding et al. used a variation of label-setting algorithm which decouples the path-selection and time-refinement by scanning a sequence of time steps of which the size depends on the values of the arrival time functions. In [14], Kanoulas et al. introduced allFP algorithm in which they, instead of sorting the priority queue by scalar values, maintain a priority queue of all the paths to be expanded. Therefore, they enumerate all the paths from a source to a destination which yields exponential run-time in the worst case.

3 Problem Definition

In this section, we formally define the problem of time-dependent kNN search in spatial networks. We assume a road network containing a set of data objects (i.e., points of interest such as restaurants, hospitals) as well as query objects searching for their kNN. We model the road network as a *time-dependent weighted graph* where the non-negative weights are time-dependent travel-times (i.e., positive piece-wise linear functions of time) between the nodes. We assume both data and query objects lie on the network edges and all relevant information about the objects is maintained by a central server.

Definition 1. *A Time-dependent Graph (G_T) is defined as $G_T(V, E)$ where V and E represent set of nodes and edges, respectively. For every edge $e(v_i, v_j)$, there is a cost function $c_{(v_i, v_j)}(t)$ which specifies the cost of traveling from v_i to v_j at time t.* □

Figure 3 depicts a road network modeled as a time-dependent graph $G_T(V, E)$. While Figure 3(a) shows the graph structure, Figures 3(b), 3(c), 3(d), 3(e), and 3(f) illustrate the time-dependent edge costs as piece-wise linear functions for the corresponding edges. For each edge, we define upper-bound ($max(c_{v_i, v_j})$) and lower-bound ($min(c_{v_i, v_j})$) time-independent costs. For example, in Figure 3(b), $min(c_{v_1, v_2})$ and $max(c_{v_1, v_2})$ of edge $e(v_1, v_2)$ are 10 and 20, respectively.
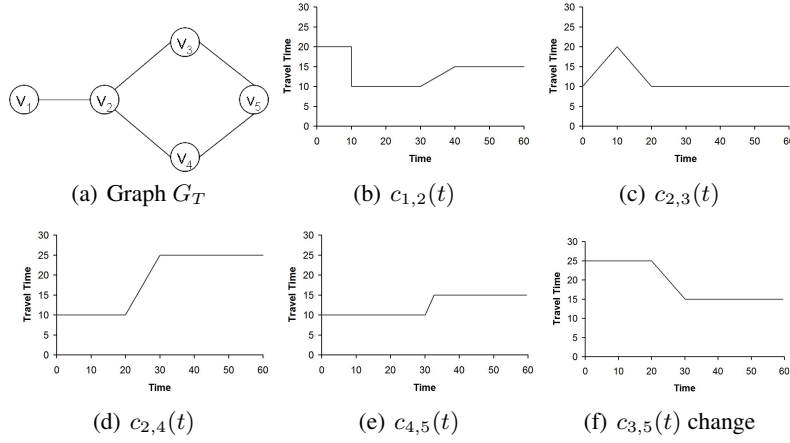
(a) Graph G_T (b) $c_{1,2}(t)$ (c) $c_{2,3}(t)$

(d) $c_{2,4}(t)$ (e) $c_{4,5}(t)$ (f) $c_{3,5}(t)$ change

Fig. 3. A Time-dependent Graph $G_T(V, E)$

Definition 2. *Let $\{s = v_1, v_2, ..., v_k = d\}$ represent a path which contains a sequence of nodes where $e(v_i, v_{i+1}) \in E$ and $i = 1, ..., k - 1$. Given a G_T, a path $(s \rightsquigarrow d)$ from source s to destination d, and a departure-time at the source t_s, the time-dependent travel time $TT(s \rightsquigarrow d, t_s)$ is the time it takes to travel along the path. Since the travel-time of an edge varies depending on the arrival-time to that edge (i.e., arrival dependency), the travel time is computed as follows:*

$$TT(s \rightsquigarrow d, t_s) = \sum_{i=1}^{k-1} c_{(v_i, v_{i+1})}(t_i) \text{ where } t_1 = t_s, t_{i+1} = t_i + c_{(v_i, v_{i+1})}(t_i), i = 1, .., k.$$

The upper-bound travel-time $UTT(s \rightsquigarrow d)$ and the lower-bound travel time $LTT(s \rightsquigarrow d)$ are defined as the maximum and minimum possible times to travel along the path, respectively. The upper and lower bound travel time are computed as follows,

$$UTT(s \rightsquigarrow d) = \sum_{i=1}^{k-1} max(c_{v_i, v_{i+1}}), LTT(s \rightsquigarrow d) = \sum_{i=1}^{k-1} min(c_{v_i, v_{i+1}}), i = 1, .., k.$$

To illustrate the above definitions in Figure 3, consider $t_s = 5$ and path (v_1, v_2, v_3, v_5) where $TT(v_1 \rightsquigarrow v_5, 5) = 45$, $UTT(v_1 \rightsquigarrow v_5) = 65$, and $LTT(v_1 \rightsquigarrow v_5) = 35$.

Note that we do not need to consider arrival-dependency when computing UTT and LTT hence; t is not included in their definitions. Given the definitions of TT, UTT and LTT, the following property holds for any path in G_T: $LTT(s \rightsquigarrow d) \leq TT(s \rightsquigarrow d, t_s) \leq UTT(s \rightsquigarrow d)$. We will use this property in subsequent sections to establish some properties of our algorithm.

Definition 3. *Given a G_T, s, d, and t_s, the time-dependent shortest path $TDSP(s, d, t_s)$ is a path with the minimum travel-time among all paths from s to d. Since we consider the travel-time between nodes as the distance measure, we refer to $TDSP(s, d, t_s)$ as time-dependent fastest path $TDFP(s, d, t_s)$ and use them interchangeably in the rest of the paper.* □

In a G_T, the fastest path from s to d is based on the departure-time from s. For instance, in Figure 3, suppose a query looking for the fastest path from v_1 to v_5 at $t_s = 5$. Then, $TDFP(v_1, v_5, 5) = \{v_1, v_2, v_3, v_5\}$. However, the same query at $t_s = 10$

returns $TDFP(v_1, v_5, 10) = \{v_1, v_2, v_4, v_5\}$. Obviously, with constant edge weights (i.e., time-independent), the query would always return the same path as a result.

Definition 4. *A time-dependent k nearest neighbor query (TD-kNN) is defined as a query that finds the k nearest neighbors of a query object which is moving on a time-dependent network G_T. Considering a set of n data objects $P = \{p_1, p_2, ..., p_n\}$, the TD-kNN query with respect to a query point q finds a subset $P' \subseteq P$ of k objects with minimum time-dependent travel-time to q, i.e., for any object $p' \in P'$ and $p \in P - P'$, $TDFP(q, p', t) \leq TDFP(q, p, t)$.* □

In the rest of this paper, we assume that G_T satisfies the First-In-First-Out (FIFO) property. This property suggests that moving objects exit from an edge in the same order they entered the edge. In practice many networks, particularly transportation networks, exhibit FIFO property. We also assume that objects do not wait at a node, because, in most real-world applications, waiting at a node is not realistic as it requires the moving object to exit from the route and find a place to park and wait.

4 TD-KNN

In this section, we explain our proposed TD-kNN algorithm. TD-kNN involves two phases: an off-line spatial network indexing phase and an on-line query processing phase. During the off-line phase, the spatial network is partitioned into *Tight Cells (TC)* and *Loose Cells (LC)* for each data object p and two complementary indexing schemes *Tight Network Index (TNI)* and *Loose Network Index (LNI)* are constructed. The main idea behind partitioning the network to TCs and LCs is to localize the kNN search and minimize the costly time-dependent shortest path computation. These index structures enable us to efficiently find the data object (i.e., generator of a tight or loose cell) that is in shortest time-dependent distance to the query object q. During the on-line phase, TD-kNN finds the first nearest neighbor of q by utilizing the TNI and LNI constructed in the off-line phase. Once the first nearest neighbor is found, TD-kNN expands the search area by including the neighbors of the nearest neighbor to find the remaining *k-1* data objects. In the following sections, we first introduce our proposed index structures and then describe online query processing algorithm that utilizes these index structures.

4.1 Indexing Time-Dependent Network (Off-line)

In this section, we explain the main idea behind tight and loose cells as well as the construction of tight and loose network index structures.

Tight Network Index (TNI) The tight cell $TC(p_i)$ is a sub-network around p_i in which any query object is guaranteed to have p_i as its nearest neighbor in a time-dependent network. We compute tight cell of a data object by using parallel Dijkstra algorithm that grows shortest path trees from each data object. Specifically, we expand from p_i (i.e., the generator of the tight cell) assuming maximum travel-time between the nodes of the network (i.e., UTT), while in parallel we expand from each and every other data object

7

assuming minimum travel-time between the nodes (i.e., LTT). We stop the expansions when the shortest path trees meet. The main rationale is that if the upper bound travel-time between a query object q and a particular data object p_i is less than the lower bound travel-times from q to any other data object, then obviously p_i is the nearest neighbor of q in a time-dependent network. We repeat the same process for each data object to compute its tight cell. Figure 4 depicts the network expansion from the data objects during the tight cell construction for p_1. For the sake of clarity, we represent the tight cell of each data object with a polygon as shown in Figure 5. We generate the edges of the polygons by connecting the adjacent border nodes (i.e., nodes where the shortest path trees meet) of a generator to each other. Lemma 1 proves the property of TC:
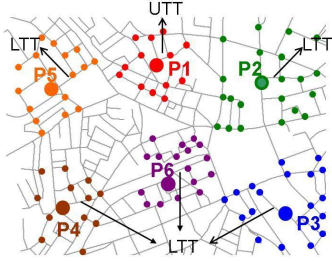

Fig. 4. Tight cell construction for P_1 **Fig. 5.** Tight Cells

Lemma 1 *Let P be a set of data objects $P = \{p_1, p_2, ..., p_n\}$ in G_T and $TC(p_i)$ be the tight cell of a data object p_i. For any query point $q \in TC(p_i)$, the nearest neighbor of q is p_i, i.e., $\{\forall q \in TC(p_i), \forall p_j \in P, p_j \neq p_i, TDFP(q, p_i, t) < TDFP(q, p_j, t)\}$.*

Proof. We prove the lemma by contradiction. Assume that p_i is not the nearest neighbor of the query object q. Then there exists a data object p_j ($p_i \neq p_j$) which is closer to q; i.e., $TDFP(q, p_j, t) < TDFP(q, p_i, t)$. Let us now consider a point b (where the shortest path trees of p_i and p_j meet) on the boundary of the tight cell $TC(p_i)$. We denote *shortest upper-bound path* from p_i to b (i.e., the shortest path among all $UTT(p_i \rightsquigarrow b)$ paths) as $D_{UTT}(p_i, b)$, and similarly, we denote *shortest lower-bound path* from p_j to b (i.e., the shortest path among all $LTT(p_j \rightsquigarrow b)$ paths) as $D_{LTT}(p_j, b)$. Then, we have $TDFP(q, p_i, t) < D_{UTT}(p_i, b) = D_{LTT}(p_j, b) < TDFP(q, p_j, t)$. This is a contradiction; hence, $TDFP(q, p_i, t) < TDFP(q, p_j, t)$. □

As we describe in Section 4.2, if a query point q is inside a specific TC, one can immediately identify the generator of that TC as the nearest neighbor for q. This stage can be expedited by using a spatial index structure generated on the TCs. Although TCs are constructed based on the network distance metric, each TC is actually a polygon in Euclidean space. Therefore, TCs can be indexed using spatial index structures (e.g., R-tree [10]). This way a function (i.e., $contain(q)$) invoked on the spatial index structure would efficiently return the TC whose generator has the minimum time-dependent network distance to q. We formally define Tight Network Index as follows.

Definition 5. *Let P be the set of data objects $P = \{p_1, p_2, ..., p_n\}$, the Tight Network Index is a spatial index structure generated on $\{TC(p_1), TC(p_2), ..., TC(p_n)\}$.* □

As illustrated in Figure 5, the set of tight cells often does not cover the entire network. For the cases where q is located in an area which is not covered by any tight cell, we utilize the Loose Network Index (LNI) to identify the candidate nearest data objects. Next, we describe LNI.

Loose Network Index (LNI) The loose cell $LC(p_i)$ is a sub-network around p_i outside which any point is guaranteed *not* to have p_i as its nearest neighbor. In other words, data object p_i is guaranteed *not* to be the nearest neighbor of q if q is outside of the loose cell of p_i. Similar to the construction process for $TC(p_i)$, we use the parallel shortest path tree expansion to construct $LC(p_i)$. However, this time, we use minimum travel-time between the nodes of the network (i.e., LTT) to expand from p_i (i.e., the generator of the loose cell) and maximum travel-time (i.e., UTT) to expand from every other data object. Lemma2 proves the property of LC:

Lemma 2 *Let P be a set of data objects $P = \{p_1, p_2, ..., p_n\}$ in G_T and $LC(p_i)$ be the loose cell of a data object p_i. If q is outside of $LC(p_i)$, p_i is guaranteed not to be the nearest neighbor of q, i.e., $\{\forall q \notin LC(p_i), \exists p_j \in P, p_j \neq p_i, TDFP(q, p_i, t) > TDFP(q, p_j, t)\}$.*

Proof. We prove by contradiction. Assume that p_i is the nearest neighbor of a q, even though the q is outside of $LC(p_i)$; i.e., $TDFP(q, p_i, t) < TDFP(q, p_j, t)$. Suppose there exists a data object p_j whose loose cell $LC(p_j)$ covers q (such a data object must exist, because as we will next prove by Lemma 3, the set of loose cells cover the entire network). Let b be a point on the boundary of $LC(p_i)$. Then, we have, $TDFP(q, p_j, t) < D_{UTT}(p_j, b) = D_{LTT}(p_i, b) < TDFP(q, p_i, t)$. This is a contradiction; hence, p_i cannot be the nearest neighbor of q. $\square$

As illustrated in Figure 6, loose cells, unlike TCs, collectively cover the entire network and have some overlapping regions among each other.
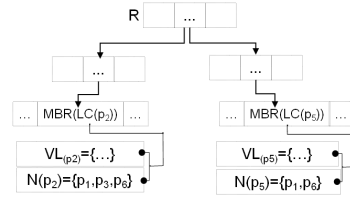

Fig. 6. Loose Cells

Fig. 7. LN R-tree

Lemma 3 *Loose cells may overlap, and they collectively cover the network.*

Proof. As we mentioned, during loose cell construction, LTT is used for expansion from the generator of the loose cell. Since the parallel Dijkstra algorithm traverses every node until the priority queue is empty as described in [8], every node in the network is

9

visited; hence, the network is covered. Since the process of expansion with LTT is repeated for each data object, in the overall process some nodes are visited more than once; hence, the overlapping areas. Therefore, loose cells cover the entire network and may have overlapping areas. Note that if the edge weights are constant, the LCs would not overlap, and TCs cells and LCs would be the same.

Based on the properties of tight and loose cells, we know that loose cells and tight cells have common edges (i.e., all the tight cell edges are also the edges of loose cells). We refer to data objects that share common edges as *direct neighbors* and remark that loose cells of the direct neighbors always overlap. For example, consider Figure 6 where the direct neighbors of p_2 are p_1, p_3, and p_6. This property is especially useful for processing *k-1* neighbors (see Section 4.2) after finding the first nearest neighbor. We determine the direct neighbors during the generation of the loose cells and store the neighborhood information in a data component. Therefore, finding the neighboring cells does not require any complex operation.

Similar to TNI, we can use spatial index structures to access loose cells efficiently. We formally define the Loose Network Index (LNI) as follows.

Definition 6. *Let P be the set of data objects $P = \{p_1, p_2, ..., p_n\}$, the Loose Network Index is a spatial index structure generated on $\{LC(p_1), LC(p_2), ..., LC(p_n)\}$.* □

Note that LNI and TNI are complementary index structures. Specifically, if a q cannot be located with TNI (i.e., q falls outside of any TC), then we use LNI to identify the LCs that contain q; based on Lemma 2, the generators of such LCs are the only NN candidates for q.

Data Structures and Updates With our approach, we use R-Tree [10] like data structure to implement TNI and LNI, termed TN R-tree and LN R-tree, respectively. Figure 7 depicts LN R-tree (TN R-tree is a similar data structure without extra pointers at the leaf nodes, hence not discussed). As shown, LN R-tree has the basic structure of an R-tree generated on minimum bounding rectangles of loose cells. The difference is that we modify R-tree by linking its leaf nodes to the the pointers of additional components that facilitate TD-kNN query processing. These components are the direct neighbors ($N(p_i)$) of p_i and the list of nodes (VL_{p_i}) that are inside $LC(p_i)$. While $N(p_i)$ is used to filter the set of candidate nearest neighbors where $k > 1$, we use VL_{p_i} to prune the search space during TDSP computation (see Section 4.2).

Our proposed index structures need to be updated when the set of data objects and/or the travel-time profiles change. Fortunately, due to local precomputation nature of TD-kNN, the affect of the updates with both cases are *local*, hence requiring minimal change in tight and loose cell index structures. Below, we explain each update type.

Data Object Updates: We consider two types of object update; insertion and deletion (object relocation is performed by a deletion following by insertion at the new location). With a location update of a data object p_i, only the tight and loose cells of p_i's neighbors are updated locally. In particular, when a new p_i is inserted, first we find the loose cell(s) $LC(p_j)$ containing p_i. Clearly, we need to shrink $LC(p_j)$ and since the loose cells and tight cells share common edges, the region that contains $LC(p_j)$ and

$LC(p_j)$'s direct neighbors needs to be adjusted. Towards that end, we find the neighbors of $LC(p_j)$; the tight and loose cells of these direct neighbors are the only ones affected by the insertion. Finally, we compute the new TCs and LCs for p_i, p_j and p_j's direct neighbors by updating our index structures. Deletion of a p_i is similar and hence not discussed.

Edge Travel-time Updates: With travel-time updates, we do not need to update our index structures. This is because the tight and loose cells are generated based on the minimum (LTT) and maximum (UTT) travel-times of the edges in the network that are time-independent. The only case we need to update our index structures is when minimum and/or maximum travel-time of an edge changes, which is not that frequent. Moreover, similar to the data object updates, the affect of the travel-time profile update is local. When the maximum and/or minimum travel-time of an edge e_i changes in the network, we first find the loose cell(s) $LC(p_j)$ that overlaps with e_i and thereafter recompute the tight and loose cells of $LC(p_j)$ and its direct neighbors.

4.2 TD-*k*NN Query Processing (Online)

So far, we have defined the properties of TNI and LNI. We now explain how we use these index structures to process kNN queries in G_T. Below, we first describe our algorithm to find the nearest neighbor (i.e., k=1), and then we extend it to address the kNN case (i.e., $k \geq 1$).

Nearest Neighbor Query We use TNI or LNI to identify the nearest neighbor of a query object q. Given the location of q, first we carry out a depth-first search from the TNI root to the node that contains q (Line 5 of Algorithm 1). If a tight cell that contains q is located, we return the generator of that tight cell as the first NN. Our experiments show that, in most cases (7 out of 10), we can find q with TNI search (see Section 5.2). If we cannot locate q in TNI (i.e., when q falls outside all tight cells), we proceed to search LNI (Line 7). At this step, we may find one or more loose cells that contain q. Based on Lemma 2, the generators of these loose cells are the only possible candidates to be the NN for q. Therefore, we compute TDFP to find the distance between q and each candidate in order to determine the first NN (Line 8-12). We store the candidates in a minimum heap based on their travel-time to q (Line 10) and retrieve the nearest neighbor from the heap in Line 12.

kNN Query Our proposed algorithm for finding the remaining *k-1* NNs is based on the direct neighbor property discussed in Section 4.1. We argue that the second NN must be among the direct neighbors of the first NN. Once we identify the second NN, we continue by including the neighbors of the second NN to find the third NN and so on. This search algorithm is based on the following Lemma which is derived from the properties of TNI and LNI.

Lemma 4 *The i-th nearest neighbor of q is always among the neighbors of the i-1 nearest neighbors of q.*

Proof. We prove this lemma by induction. We prove the base case (i.e., the second NN is a direct neighbor of the first NN of q) by contradiction. Consider Figure 9 where

11

| **Algorithm 1** NN-Algorithm(q,TNI,LNI)	**Algorithm 2** kNN-Algorithm(TNI,LNI,q,k)
1: // q: location of the query object,	1: // N: an array of NN set
2: // S: an array containing the candidate set	2: // H: a minimum heap, p: any NN, k: number of NN
3: // H: a minimum heap, p: the first NN	3: Initialize H, N
4: Initialize S and H;	4: $p \leftarrow NN\text{-}Algorithm(q, TNI, LNI)$;
5: $p \leftarrow contain_{TNI}(q)$;	5: add p to N
6: **if** p is null **then**	6: **while** $N.size \leq k$ **do**
7: $S \leftarrow contain_{LNI}(q)$;	7: **for** each neighbor s_i of N **do**
8: **for** each data object s_i in S **do**	8: $computeTDFP(q, s_i, t)$
9: $computeTDFP(q, s_i, t)$;	9: add s_i to H;
10: insert s_i to H;	10: **end for**
11: **end for**	11: $p \leftarrow deHeap\ H$; // find next NN
12: $p \leftarrow deHeap\ H$;	12: add p to N
13: **end if**	13: **end while**
14: $return\ p$;	14: $return\ N$ // return kNN
(a) Algorithm 1	(b) Algorithm 2

Fig. 8. kNN query algorithm in time-dependent road networks

p_2 is the first NN of q. Assume that p_5 (which is not a direct neighbor of p_2) is the second NN of q. Since p_2 and p_5 are not direct neighbors, a point w on the time-dependent shortest path between q and p_5 can be found that is outside both $LC(p_2)$ and $LC(p_5)$. However, p_5 cannot be a candidate NN for w, because w is not in $LC(p_5)$. Thus, there exists another object such as p_1 which is closer to w as compared to p_5. Therefore, $TDFP(w, p_5, t) > TDFP(w, p_1, t)$. However, as shown in Figure 9, we have $TDFP(q, p_5, t) = TDFP(q, w, t) + TDFP(w, p_5, t) > TDFP(q, w, t) + TDFP(w, p_1, t) = TDFP(q, p_1, t)$. Thus, p_5 is farther from q than both p_2 and p_1, which contradicts the assumption that p_5 is the second NN of q. The proof of inductive step is straight forward and similar to the above proof by contradiction; hence, due to lack of space, we omit the details. □

The complete TD-kNN query answering process is given in Algorithm 2. Algorithm 2 calls Algorithm 1 to find the first NN and add it to N, which maintains the current set of nearest neighbors (Lines 4-5). To find the remaining $k - 1$ NNs, we expand the search area by including the neighboring loose cells of the first NN. We compute the TDSP for each candidate and add each candidate to a minimum heap (Lines 9) based on its time-dependent travel-time to q. Thereafter, we select the one with minimum distance as the second NN (Line 11). Once we identify the second NN, we continue by investigating the neighbor loose cells of the second NN to find the third NN and so on. Our experiments show that the average number of neighbors for a data object is a relatively small number less than 9 (see Section 5.2).
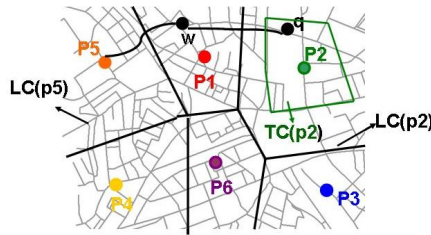
Fig. 9. Second NN Example
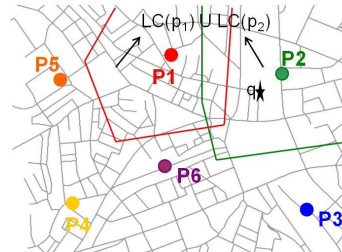
Fig. 10. TDSP localization

12

Time-dependent Fastest Path Computation As we explained, once the nearest neighbor of q is found and the candidate set is determined, the time-dependent fastest path from q to all candidates must be computed in order to find the next NN. Before we explain our TDFP computation, we note a very useful property of loose cells. That is, given p_i is the nearest neighbor of q, the time-dependent shortest path from q to p_i is guaranteed to be in $LC(p_i)$ (see Lemma 5). This property indicates that we only need to consider the edges contained in the loose cell of p_i when computing TDFP from q to p_i. Obviously, this property allows us to localize the time-dependent shortest path search by extensively pruning the search space. Since the localized area of a loose cell is substantially smaller as compared to the complete graph, the computation cost of TDFP is significantly reduced. Note that the subnetwork bounded by a loose cell is on average $1/n$ of the original network where n is the total number of sites.

Lemma 5 *If p_i is the nearest neighbor of q, then the time-dependent shortest path from q to p_i is guaranteed to be inside the loose cell of p_i*

Proof. We prove by contradiction. Assume that p_i is the NN of q but a portion of TDFP from q to p_i passes outside of $LC(p_i)$. Suppose a point l on that outside portion of the path. Since l is outside $LC(p_i)$, then $\exists p_j \in P$, $p_j \neq p_i$ that satisfies $D_{LTT}(p_i, l) > D_{UTT}(p_j, l)$ and hence $TDFP(p_i, l, t) > TDFP(p_j, l, t)$. Then, $TDFP(p_i, q, t) = TDFP(p_i, l, t) + TDFP(l, q, t) > TDFP(p_j, l, t) + TDFP(l, q, t) = TDFP(p_j, q, t)$, which contradicts the fact that p_i is the NN of q. $\square$

We note that for TD-kNN with $k > 1$, the TDFP from q to the kth nearest neighbor will lie in the combined area of neighboring cells. Figure 10 shows an example query with $k > 1$ where p_2 is assumed to be the nearest neighbor (and the candidate neighbors of p_2 are, p_1, p_6 and p_3). To compute the TDFP from q to data object p_1, we only need to consider the edges contained in $LC(p_1) \cup LC(p_2)$. Below, we explain how we compute the TDFP from q to each candidate.

As initially showed by Dreyfus [7], the TDFP problem (in FIFO networks) can be solved by modifying any label-setting or label-correcting static shortest path algorithm. The asymptotic running times of these modified algorithms are same as those of their static counterparts. With our approach, we implement a time-dependent A* search (a label-setting algorithm) to compute TDFP between q and the candidate set. The main idea with A* algorithm is to employ a heuristic function $h(v)$ (i.e., lower-bound estimator between the intermediate node v_i and the target t) that directs the search towards the target and significantly reduces the number of nodes that have to be traversed. With static road networks where the length of an edge is considered as the cost, the Euclidean distance between v_i and t is the lower-bound estimator. However, with time-dependent road networks, we need to come up with an estimator that never overestimates the travel-time between v_i and t for *all possible departure-times* (from v_i). One simple lower-bound is $d_{euc}(v_i, t)/max(speed)$, i.e., the Euclidean distance between v_i and t divided by the maximum speed among the edges in the entire network. Although this estimator is guaranteed to be a lower-bound between v_i and t, it is a very loose bound, hence yields insignificant pruning. Fortunately, our approach can use Lemma 5 to obtain a much tighter lower-bound. Since the shortest path from q to p_i is guaranteed to

13

be inside $LC(p_i)$, we can use the maximum speed in $LC(p_i)$ to compute the lower-bound. We outline our time-dependent A* algorithm in Algorithm 3 where essential modifications (as compared to [7]) are in Lines 3, 10 and 14. As mentioned, to compute TDFP from q to candidate p_i, we only consider the nodes in the loose cell that contains q and $LC(p_i)$ (Line 3). To compute the labels for each node, we use arrival time and the estimator (i.e., $cost(v_i) + h_{LC}(v_i)$ where $h_{LC}(v_i)$ is the lower-bound estimator calculated based on the maximum speed in the loose cell) to each node that form the basis of the greedy algorithm (Line 10). In Lines 10 and 14, $TT(v_i, v_j, t_{v_i})$ finds the time-dependent travel-time from v_i to v_j (see Section 3).

Algorithm 3 TDFP($G_T(V, E), q, d, t_s$)
1: // q:source, d:target, t_v:departure-time from node v,
2: //$cost(v)$:cost from s to v, $pre(v)$:previous node in optimal path
3: $Q \leftarrow$ set of nodes in $LC(q)$ and $LC(d)$
4: $\forall v \in Q \; cost(v) = \infty, cost(q) = 0$
5: **while** Q *is not emtpy* **do**
6:     $v_i \leftarrow$ node in Q with smallest cost
7:     *remove* v_i *from* Q
8:     IF $v_i = d$ THEN return path
9:     *for each* neighbor v_j of v_i
10:     $l(v_j) = cost(v_i) + h_{LC}(v_i) + TT(v_i, v_j, t_{v_i})$
11:     IF $l(v_j) < cost(v_j)$ THEN
12:     $cost(v_j) = l(v_j)$
13:     $pre(v_j) = v_i$
14:     $t_{v_j} = d_t(v_i) + TT(v_i, v_j, t_{v_i})$
15: **end while**

Fig. 11. TDSP Algorithm

5 Experimental Evaluation

5.1 Experimental Setup

We conducted several experiments with different spatial networks and various parameters (see Figure 12) to evaluate the performance of TD-kNN. We run our experiments on a workstation with 2.7 GHz Pentium Duo Processor and 12GB RAM memory. We continuously monitored each query for 100 timestamps. For each set of experiments, we only vary one parameter and fix the remaining to the default values in Figure 12. With our experiments, we measured the tight cell hit ratio and the impact of k, data and query object cardinality as well as the distribution. As our dataset, we used Los Angeles (LA) and San Joaquin (SJ) road networks with 304,162 and 24,123 segments, respectively.

We evaluate our proposed techniques using a database of actual time-dependent travel-times gathered from real-world traffic sensor data. For the past 1.5 year, we have been collecting and archiving speed, occupancy, volume sensor data from a collection of approximately 7000 sensors located on the road network of LA. The sampling rate of the data is 1 reading/sensor/min. Currently, our database consists of about 900 million sensor reading representing traffic patterns on the road network segments of LA. In order to create the time-dependent edge weights of SJ, we developed a system [5] that synthetically generates time-dependent edge weights for SJ.

5.2 Results

Impact of Tight Cell Hit Ratio and Direct Neighbors As we explained, if a q is located in a certain tight cell $TC(p_i)$, our algorithm immediately reports p_i as the first NN. Therefore, it is essential to asses the coverage area of the tight cells over the entire network. Figure 13(a) illustrates the coverage ratio of the tight cells with varying data object cardinality (ranging from 1K to 20K) on two data sets. As shown, the average tight cell coverage is about $\%68$ of the entire network for both LA and SJ. This implies that the first NN of a query can be answered immediately with a ratio of $7/10$ with no further computation. Another important parameter affecting the TD-kNN algorithm is the average number of direct neighbors for each data object. Figure 13(b) depicts the average number of neighbor cells with varying data object cardinality. As shown, the average number of neighbors is less than 9 for both LA and SJ.
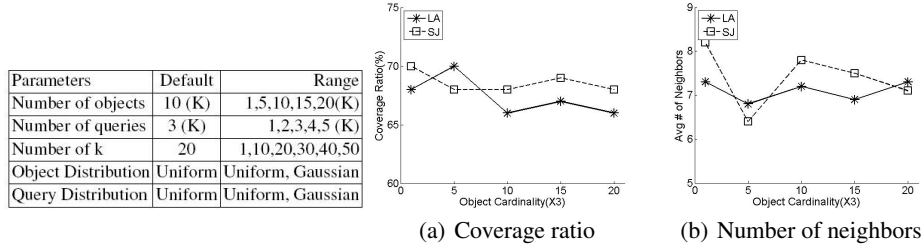
Parameters	Default	Range
Number of objects	10 (K)	1,5,10,15,20(K)
Number of queries	3 (K)	1,2,3,4,5 (K)
Number of k	20	1,10,20,30,40,50
Object Distribution	Uniform	Uniform, Gaussian
Query Distribution	Uniform	Uniform, Gaussian

(a) Coverage ratio (b) Number of neighbors

Fig. 12. Experimental Parameters **Fig. 13.** Time-dependent fastest path localization

As mentioned, in [7] we developed an incremental network expansion algorithm (based on [7]) to evaluate kNN queries in time-dependent networks. Below we compare our results with this naive approach. For the rest of the experiments, since the experimental results with both LA and SJ networks differ insignificantly and due to space limitations, we only present the results from LA dataset.

Impact of k In this experiment, we compare the performance of both algorithms by varying the value of k. Figure 14(a) plots the average response time versus k ranging from 1 to 50 while using default settings in Figure 12 for other parameters. The results show that TD-kNN outperforms naive approach for all values of k and scales better with the large values of k. As illustrated, when k=1, TD-kNN generates the result set almost instantly. This is because a simple $contain()$ function is enough to find the first NN. As the value of k increases, the response time of TD-kNN increases at linear rate. Because, TD-kNN, rather than expanding the search blindly, benefits from localized computation. In addition, we compared the average number of network node access with both algorithms. As shown in Figure 14(b), the number of nodes accessed by TD-kNN is less than the naive approach for all values of k.

Impact of Object and Query Cardinality Next, we compare the algorithms with respect to cardinality of the data objects (P). Figure 15(a) shows the impact of P on
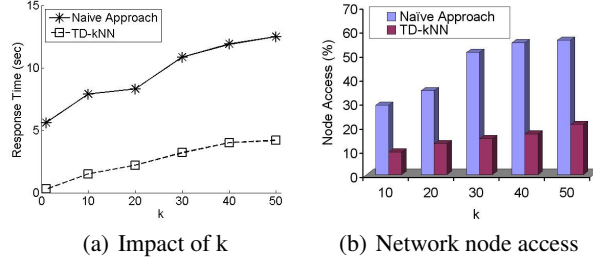
(a) Impact of k (b) Network node access

Fig. 14. Response time and node access versus k

response time. The response time linearly increases with the number of data objects in both methods where TD-kNN outperforms the naive approach for all cases. From P=1K to 5K, the performance gap is more significant. This is because, for lower densities where data objects are possibly distributed sparsely, naive approach requires larger portion of the network to be retrieved. Figure 15(b) shows the impact of the query cardinality (Q) ranging from 1K to 5K on response time. As shown, TD-kNN scales better with larger Q and the performance gap between the approaches increases as Q grows.
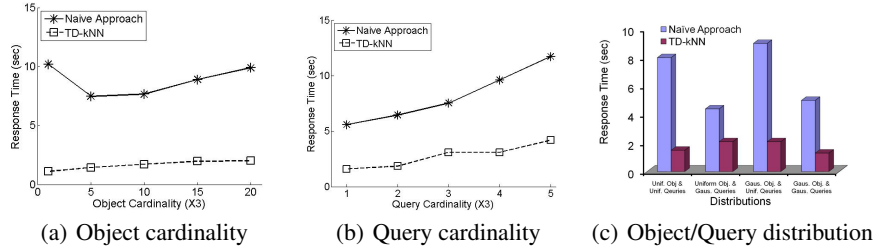

(a) Object cardinality (b) Query cardinality (c) Object/Query distribution

Fig. 15. Impact of k

Impact of Object/Query Distribution Finally, we study the impact of object, query distribution. Figure 15(c) shows the response time of both algorithms where the objects and queries follow either uniform or Gaussian distributions. TD-kNN outperforms the naive approach significantly in all cases. TD-kNN yields better performance for queries with Gaussian distribution. This is because as queries with Gaussian distribution are clustered in the network, their nearest neighbors would overlap hence allowing TD-kNN to reuse the path computations.

6 Conclusion and Future Work

In this paper, we proposed a time-dependent k nearest neighbor search algorithm (TD-kNN) for spatial networks. With TD-kNN, unlike the existing studies, we assume the edge weights of the network are time varying rather than fixed. In real-world, time-varying edge utilization is inherit in almost all networks (e.g., transportation, internet, social networks). Hence, we believe that our approach yields a much more realistic scenario and is applicable to kNN applications in other domains. We intend to pursue this

study in two directions. First, we plan to investigate new data models for effective representation of time-dependent spatial networks. This is critical in supporting development of efficient and accurate time-dependent algorithms, while minimizing the storage and cost of the computation. Second, we intend to study a variety of other spatial queries (including continuous kNN, range and skyline queries) in time-dependent networks.

References

1. I. Chabini. The discrete-time dynamic shortest path problem: Complexity, algorithms, and implementations. In *Journal of Transportation Research Record, 1645*, 1999.
2. H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cnn queries in a road network. In *Proceedings of VLDB*, 2005.
3. L. Cooke and E. Halsey. The shortest route through a network with timedependent internodal transit times. In *Journal of Mathematical Analysis and Applications*, 1966.
4. U. Demiryurek, F. B. Kashani, and C. Shahabi. Towards k-nearest neighbor search in time-dependent spatial network databases. In *Proceedings of DNIS*, 2010.
5. U. Demiryurek, B. Pan, F. B. Kashani, and C. Shahabi. Towards modeling the traffic data on road networks. In *Proceedings of SIGSPATIAL-IWCTS*, 2009.
6. B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over graphs. In *Proceedings of EDBT*, 2008.
7. P. Dreyfus. An appraisal of some shortest path algorithms. In *Journal of Operation Research 17*, NY, USA, 1969.
8. M. Erwig and F. Hagen. The graph voronoi diagram with applications. *Journal of Networks*, 36, 2000.
9. B. George, S. Kim, and S. Shekhar. Spatio-temporal network databases and routing algorithms: A summary of results. In *Proceedings of SSTD*, 2007.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, 1984.
11. J. Halpern. Shortest route with time dependent length of edges and limited delay possibilities in nodes. In *Journal of Mathematical Methods of Operations Research,21*, 1969.
12. H. Hu, D. L. Lee, and J. Xu. Fast nearest neighbor search on road networks. In *Proceedings of EDBT*, 2006.
13. X. Huang, C. S. Jensen, and S. Saltenis. The island approach to nearest neighbor querying in spatial networks. In *Proceedings of SSTD*, 2005.
14. E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *Proceedings of ICDE*, 2006.
15. M. Kolahdouzan and C. Shahabi. Voronoi-based k nn search in spatial networks. In *Proceedings of VLDB*, 2004.
16. U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation and Mobilitat*, 2004.
17. Navteq. http://www.navteq.com. Last visited January 2, 2010.
18. A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM,37*, 1990.
19. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial networks. In *Proceedings of VLDB*, 2003.
20. H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of SIGMOD*, 2008.
21. TeleAtlas. http://www.teleatlas.com. Last visited January 2, 2010.
22. D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. *Proceedings of Algorithms-ESA*, 2003.