2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops

kNN Search Utilizing Index of the Minimum Road Travel Time in Time-Dependent Road Networks

Yuka Komai, Nguyen Duong Hong, Takahiro Hara and Shojiro Nishio
Department of Multimedia Engineering, Graduate School of Information Science and Technology
Osaka University
Yamadaoka 1-5, Suita-shi, Osaka, Japan
Email: {komai.yuka, hara, nishio}@ist.osaka-u.ac.jp, nguyen.duong@ise.eng.osaka-u.ac.jp

***Abstract*—Recently, there has been an increasing interest in search in time-dependent road networks where the travel time on roads depends on the time. In such a time-dependent network, the result of k Nearest Neighbor (kNN) queries, which search the k nearest neighbors (kNNs) from the specified location, depends on the query-issuing time. Therefore, existing approaches in static networks are not directly applied for kNN query in time-dependent road networks. In this paper, we propose a kNN search method to achieve a small number of visited vertexes and small response time in time-dependent road networks. In our proposed method, an index structure is constructed based on the minimum travel time on roads in the preprocessing phase. In query processing, a network is expanded by A* algorithm with referring the minimum travel time in the index until kNNs are found. An experimental result shows that our proposed method reduces the number of visited vertexes and the response time compared with an existing method.**

***Keywords*-time-dependent road network; kNN search; A* algorithm; index;**

I. Introduction

Location-based service (LBS) is a typical application for road networks. In an LBS, it is common for a user to issue k Nearest Neighbor (kNN) queries, which search the information on the k nearest neighbors (kNNs) from the specified location (query point). For example, a user on a car can effectively acquire the information or coupons on the 10 nearest restaurants by using a kNN query.

There have been many existing methods for processing queries which are efficient for static road networks where the travel time on each road is constant. On the other hand, recently, there has been an increasing interest in search in time-dependent road networks where the travel time on roads depends on the time (i.e., it is a function of the time). Figure 1 shows an example of the average travel time on a road in Osaka city, Japan. Since actual travel time on roads is time-dependent, i.e., it depends on traffic on roads, the road network in a real environment is a time-dependent network rather than a static network.

KNN search in such a time-dependent road network has some deferences from that in a static road network. In Figure 2, a user searches the nearest store at 8 AM, and in Figure 3, another user searches the nearest store at 2 PM. In Figure 2, the nearest store from the user is store A because the travel time to store A is the smallest (3 min.). On the other hand, in Figure 3, it is store B at 2 P.M.. Thus, the kNN result in time-dependent road networks is not fixed (i.e., the kNN result depends on the query-issuing time) since the travel time on roads is not static. Therefore, existing approaches are not directly applied for kNN search in time-dependent road networks.

In [3], to search kNNs in time-dependent networks, the authors proposed the Time Dependent Incremental Network Expansion (TD-INE) algorithm which is based on Dijkstra's algorithm [5]. In this method, vertexes in a network are visited in order of travel time from the query point until

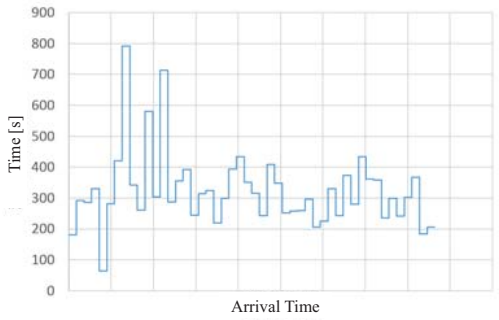


Figure 1. The average travel time on a road in Osaka city, Japan

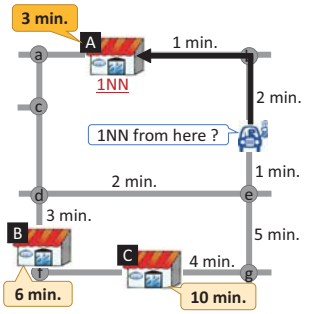


Figure 2. Query processing at 8 AM

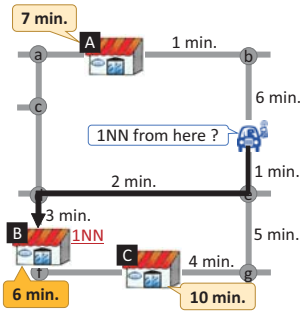


Figure 3. Query processing at 2 PM

978-1-4799-7361-3/14 $31.00 © 2014 IEEE
DOI 10.1109/SRDSW.2014.17

k data objects are found. This method is an on-demand search which can work when the travel time on roads is both static and time-dependent. However, this method expands the network in every direction to search all the vertexes to which the travel time from the query point is smaller than that to the k-th nearest data object. Therefore, the number of visited vertexes increases, and thus, the response time for a query becomes large.

Compared with Dijkstra's algorithm, A* algorithm is efficient for the shortest path search on road networks. In A* algorithm, when searching from a vertex s (start point) to a vertex d (destination), vertexes are visited in order of cost which is the travel time from s to a visited vertex plus the estimated travel time from the visited vertex to d which is given by a heuristic function. A* algorithm achieves a small number of visited vertexes since a network is expanded only toward d.

A* algorithm is also effective for kNN search to achieve a small number of visited vertexes. It is expected that A* algorithm can reduce the number of visited vertexes and the response time for a query than that in TD-INE which is based on Dijkstra's algorithm. Here, the performance of A* algorithm depends on the heuristic function, but it is difficult to appropriately estimate the travel time to d since the travel time on roads is time-dependent. Moreover, while A* algorithm requires to specify the destination in advance, the destinations are unknown in kNN search, i.e., the destinations are the search target.

In this paper, we propose a kNN search method to achieve a small number of visited vertexes and also small response time in time-dependent road networks. In our method, an index structure is constructed for each time segment based on the minimum travel time in the time segment in the preprocessing phase. The entries of the indexes are the CNNs (C is a constant) from each vertex and the travel time through the fastest path to each of CNNs. In query processing, A* algorithm is executed with the indexes for the a time segment including the query-issuing time. Starting from the query point, a vertex is visited toward a data object, which is indexed on the vertex and not already found, in order of cost (i.e., expected travel time) until k data objects are found. Here, the expected travel time is calculated by using the travel time in the index. Our proposed method can reduce the number of visited vertexes in search because the search space is expanded only toward data objects which are potential kNNs.

The remainder of this paper is organized as follows. In Section 2, we define kNN query in time-dependent road networks. In Section 3, we introduce related work. In Section 4, we present our proposed kNN search method. In Section 5, we discuss the results of simulation experiments, and in Section 6, we summarize the paper.

II. Problem Definition

In this section, we define kNN query in time-dependent road networks. In this paper, we assume that there are static data objects on the road network, and a user acquires the k nearest data objects by issuing a query. For simplicity, the query point specified by the user is the location of the query-issuing node. We model a time-dependent road network into a weighting graph where the weight of each edge is time-dependent and is statically defined in advance. In this paper, points of data objects (e.g., restaurants and shops) are on vertexes. The time-dependent road graph, $G_t(V, E)$, is defined as follow.

Definition 2.1: Time-dependent road graph, $G_t(V, E)$.

In $G_t(V, E)$, $V = \{v_1, v_2, ..., v_m\}$ is a set of vertexes and E ($E \subseteq V \times V$) is a set of edges. An edge e is described by $e(v_i, v_j)$. G_t is an undirected graph, i.e., $e(v_i, v_j) = e(v_j, v_i)$. Each edge $e(v_i, v_j)$ has a cost, $c_{v_i,v_j}(t)$, which is the travel time from v_i to v_j at time t. Here, the minimum travel time of $e(v_i, v_j)$ between t and t' is described as $min(c_{v_i,v_j}[t, t'])$.

Definition 2.2: Travel time through paths.

A path is described as $\{s = v_{a_1}, v_{a_2}, ..., v_{a_j} = d\}$, where $e(v_{a_i}, v_{a_{i+1}}) \in E$, $i = 1, ..., j-1$. Here, s is the vertex of start point and d is the vertex of destination. The travel time through a path from s to d on G_t at time t_s is defined by the following equation:

$$TT(s \longrightarrow d, t_s) = \sum_{i=1}^{j-1} c_{v_{a_i}, v_{a_{i+1}}}(t_i) \quad (1)$$

Here, $t_1 = t_s, t_{i+1} = t_i + c_{v_{a_i}, v_{a_{i+1}}}(t_i), i = 1, ..., j-1$.

Definition 2.3: Fastest path in a time-dependent road network.

In G_t, $TDFP(s, d, t_s)$ is defined as the fastest path from s to d at time t_s. For example, when t_s is 5, $TDFP$ from v_1 to v_5 in Figure 4(a) is $TDFP(v_1, v_5, 5) = \{v_1, v_2, v_3, v_5\}$. Moreover, the travel time through $TDFP(s, d, t_s)$, $TDFT(s, d, t_s)$, is defined by the following equation:

$$TDFT(s \longrightarrow d, t_s) = TT(s \longrightarrow d, t_s) \quad (2)$$

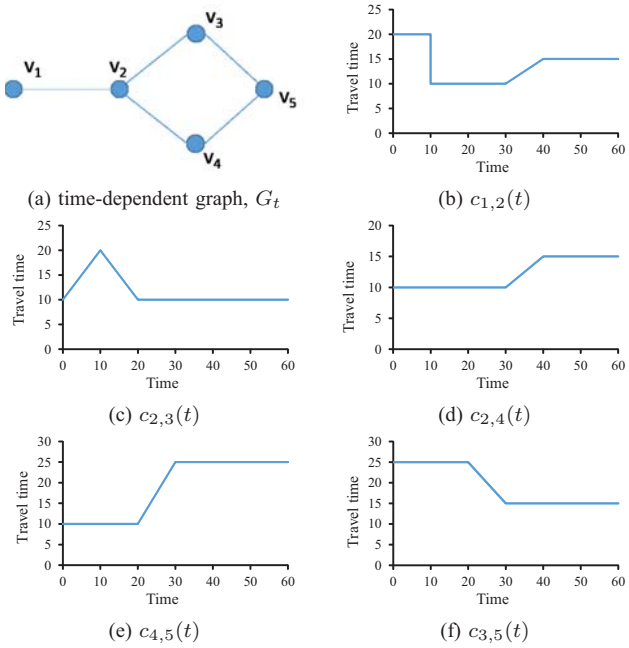
Definition 2.4: kNNs in a time-dependent road network.

A kNN query in a time-dependent road network acquires k data objects with the minimum cost (i.e., the travel time from the query point).

III. Related Work

A. The fastest path search in time-dependent graphs

Since kNNs on road networks is decided based on the travel time of the fastest (shortest) path from the query point to each data object, fastest path search is related to kNN search. In [6], the authors probed that Dijkstra's algorithm can be applied for the fastest path search in time-dependent



(a) time-dependent graph, G_t (b) $c_{1,2}(t)$ (c) $c_{2,3}(t)$ (d) $c_{2,4}(t)$ (e) $c_{4,5}(t)$ (f) $c_{3,5}(t)$

Figure 4. time-dependent graph

graphs. On the other hand, in [4], the authors proposed a method using bidirectional A* algorithm, which is more efficient than Dijkstra's algorithm in fastest path search. These methods can acquire the fastest path to the destination, but they are not directly applied for kNN search because the destinations are multiple and unknown (i.e., search targets themselves) in kNN search.

B. kNN query processing in time-dependent road networks

[3] is the first study that addressed kNN search in time-dependent road networks, and proposed TD-INE. This method is based on Dijkstra's algorithm like INE [9]. In this method, vertexes are visited in order of distance from the query point to each vertex until k data objects are found.

In [2], the authors proposed a kNN search method using Tight Network Index (TNI) and Loose Network Index (LNI). Using TNI, a query immediately finds the nearest data object from any query point, while using LNI, the query selects the candidates of kNN data objects because it can know data objects closer to the nearest data object from the query point. Therefore, the fastest path to the candidates of kNN data objects is calculated, and then, kNNs are decided. However, in this method, many iterations of the fastest path calculation are required, which causes an increase of the response time for a query. Moreover, when the density of data objects is high, the search time increases since the number of candidates is large.

In [1], the authors proposed an A* algorithm based method for kNN search in time-dependent road networks. This method achieves smaller response time than that by Dijkstra's algorithm. In this method, a heuristic function in A* algorithm is the minimum travel time during all time on each edge. However, in a network where there is a big difference between the minimum and maximum travel time on each road, e.g., between the midnight and rush hours, visited vertexes are not efficiently pruned because the heuristic value is far from the actual travel time.

IV. kNN SEARCH METHOD

In this section, first, we describe the design policy of our method, and explain the heuristic function used in our method. Then, we explain our proposed method using the TD-FTT index.

A. Design Policy

In kNN search in a time-dependent road network, the result of kNN queries depends on the query-issuing time since the travel time on each edge depends on the arrival time at the edge. Calculating the travel time of the fastest path from every vertex to each object in advance and storing it as an index entry is unrealistic since it is too costly in terms of computation and storage space. Therefore, in our method, kNN queries are processed by network expansion, and we adopt A* algorithm to reduce the search for unnecessary vertexes during the network expansion.

A* algorithm finds the fastest path from the start point to the destination. When searching via vertex n, the travel time through the fastest path from the start point to the destination via n, $f(n)$, is calculated as $f(n) = g(n)+h(n)$. Here, $g(n)$ is the travel time through the fastest path from the start point to n, and $h(n)$ is the travel time through the fastest path from n to the destination. If $g(n)$ and $h(n)$ are already known, $f(n)$ is easily calculated.

However, $h(n)$ is basically unknown in advance although $g(n)$ is known during searching. Therefore, in A* algorithm, $f(n)$ is altered to $f^*(n) = g(n) + h^*(n)$. Here, $h^*(n)$ is called a heuristic function, i.e., the estimation value of the travel time from n to the destination. In A* algorithm, the search result when using $h^*(n)$ is guaranteed as correct when $\forall n, 0 \leq h^*(n) \leq h(n)$. Moreover, for $\forall n, h_1^*(n) < h_2^*(n) \leq h(n)$, h_2^* is a better function than h_1^*, i.e., unnecessary visits of vertexes are more suppressed. Therefore, it is better to adopt a value closer to the actual travel time as the heuristic function.

B. Heuristic function in our method

Definition 4.1: A road network with the minimum travel time during time segment $[t, t']$.

We define a graph $G_t(V, E)$ with the minimum travel time on each edge during time segment $[t, t']$, as $G_{FTT}[t, t'](V, E)$. In $G_{FTT}[t, t'](V, E)$, sets of V and E are the same as in $G_t(V, E)$, and the cost of each edge, $e(v_i, v_j)$, equals $min(c_{v_i,v_j}[t, t'])$. Here, $FP[t, t'](s, d)$ is the fastest path from s to d on $G_{FTT}[t, t'](V, E)$, and described as

$FP[t,t'](s,d) = \{s = v_{a_1}, v_{a_2}, ..., v_{a_j} = d\}$. The travel time from s to d in $[t,t']$ is defined by the following equation:

$$FTT[t,t'](s \longrightarrow d) = \sum_{i=1}^{j-1} min(c_{v_{a_i},v_{a_{i+1}}}[t,t']) \quad (3)$$

Here, $t_s \in \{t,t'\}$, $FTT[t,t'](s \longrightarrow d) \leq TDFT(s \longrightarrow d, t_s)$.

Definition 4.2: Heuristic function based on the minimum travel time during time segment $[t,t']$.

First, we assume that the query-issuing time is $t_s (t_s \in [t,t'])$ and the arrival times at all kNN data objects (starting from t_s) are earlier than t'. Here, the set of all data objects, D, is defined as $\{d_1, d_2, ..., d_x\}$. After a query is issued at t_s, a network is expanded from the query point, q, and let us assume that vertex n is currently visited. Here, a set of objects in the query result which are already found, NN, is defined as $\{NN_1, NN_2, ..., NN_l\}$ $(l < k)$. In the network expansion, the destination from n is defined as the nearest data object from n, $d(n)$ $(d(n) \in D \setminus NN)$. Therefore, $h^*(n) = FTT[t,t'](n \longrightarrow d(n))$.

C. kNN search method with index

In our proposed method, the kNN result is acquired with a small number of visits of vertexes owing to A* algorithm with the heuristic function described in subsection IV-B. However, if the value of the heuristic function (which we call heuristic value) is calculated during query processing, the response time becomes large. Therefore, the expected travel time from each vertex to a data object is calculated in advance and stored as an entry of indexes, which we call the Time Dependent Fast Travel Time (TD-FTT) index.

Preprocessing: First, the entire time $[0, t_b]$ is divided into some time segments $[0, t_1]$, $[t_1, t_2]$,..., $[t_{b-1}, t_b]$. After $G_{FTT}[t_i, t_{i+1}](V,E)$ is constructed, for each vertex, CNNs from itself on $G_{FTT}[t_i, t_{i+1}](V,E)$ are found by Dijkstra's algorithm. At the same time, the travel times to CNNs are calculated. Then, the C nearest data objects and their travel times are stored in an index. Algorithm 1 shows the pseudo code of a TD-FTT Index construction and it is processed for each time segment. In our proposed method, to effectively search kNNs, the heuristic function is based on the minimum travel time in a time segment which is more closer to the actual travel time than the minimum travel time during the entire time.

Figure 5 shows an example of constructing the TD-FTT index in $[t,t']$ on vertex b where data objects are stores. Here, the travel times of $(b,$ store A$)$, (b,e), (d,e), (e,g), $(d,$ store B$)$, and $(g,$ store C$)$ are respectively 5 min, 1 min, 2 min, 2 min, 3 min, and 1 min. When $C = 2$, the travel times from b to Store C and Store A are stored as heuristic values in the index on b.

Algorithm 1 TD-FTT Index($G_{FTT}[t,t'](V,E)$, C)

// m:the number of vertexes in a network
// C:the number of data points held by an index of each vertex
// $vertex$: a vertex in a road network
// $INE(vertex[i], c)$: a function for kNN search algorithm based on Dijkstra's algorithm[9]
// $INE(vertex[i], k).data$: ID of the k-th nearest data objects from $vertex[i]$
// $INE(vertex[i], k).travel\_time$: the travel time of $vertex[i]$ is k-th nearest data object from $vertex[i]$
// $index[i]$: a TD-FTT index on a vertex i at $[t_i, t_{i+1}]$
// $index[i].rank[j]$: the value is j when j-th nearest data object in $index[i]$
// $index[i].data[j]$: for a vertex i, j-th nearest data object in $G_{FTT}[t,t'](V,E)$
// $index[i].heuristic\_value[j]$: for a vertex i, the travel time from a vertex i to the j-th nearest object in $G_{FTT}[t,t'](V,E)$
// $vertex[i].pointer$: the pointer for an index on a vertex i

1: **for** $i = 1$ to m **do**
2: **for** $j = 1$ to c **do**
3: $index[i].rank[j] = j$;
4: $index[i].data[j] = INE(vertex[i], j).data$;
5: $index[i].heuristic\_value[j] = INE(vertex[i], j).travel\_time$;
6: **end for**
7: $vertex[i].pointer = 0$;
8: **end for**
9: **return** $index$

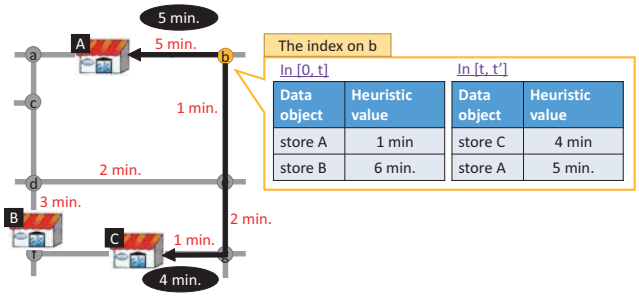


Figure 5. Example of a TD-FTT index

Processing kNN search: In our proposed method, kNN queries are processed by A* algorithm referring to heuristic values in indexes. Since an index is constructed for each time segment, the index for $[t,t']$ $(t \in [t,t'])$ is referred when a query is issued at t. Starting from the query point, a vertex is successively visited toward a data object, which is indexed on the vertex and not already found, in order of cost (i.e., expected travel time) until k data objects are found. Here, the expected travel time is calculated by the heuristic value $h^*(n)$ of the data object in the index. Algorithm 2 shows the procedures of the proposed kNN search algorithm.

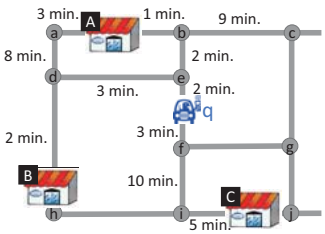
Figure 6 shows an example of 2NN search in a time-dependent road network at time T $(t < T < t')$. The query first visits vertexes e and f because these are neighboring vertexes of the query point q, and the two vertexes are enqueued. Then, the expected travel time for e is calculated by the travel time of (q, e) and the minimum heuristic value (for store A) in the index on e at $[t,t']$. Thus, the cost on e is 5 min (2 min + 3 min). In the same way, the expected travel time for f is calculated by the travel time of (q, f) and the minimum heuristic value (about store C) in the index on f at $[t,t']$, and thus, the cost on f is 9 min (3 min + 6 min). Because the cost on e is the smallest in the queue,

Algorithm 2 kNN(q, $start\_time$, $index$,k)

// q:the query point
// $start\_time$: the query issuing time
// $come\_time$: arrival time on the edge
// $index$:FTT index
// k: the number of requested number of kNNs
// u: the reference node
// $vertex$: a set of nodes
// $vertex[i]$: vertex i
// $vertex[i].travel\_time$: the travel time from the query point to vertex i
// $vertex[i].label = 0$: vertex i is not inserted to queue
// $vertex[i].label = 1$: vertex i is already inserted to queue
// $vertex[i].cost$: $vertex[i].travel\_time + FTT(n \longrightarrow d(n))$
// $vertex[i].pointer$: pointer to heuristic value in the index on vertex i
// $vertex[i].adjacent$: a set of neighboring vertexes of vertex i
// $queue$: the priority queue
// $sort(queue)$: sort vertexes in queue in a $cost$ ascending order
// $queue.insert(vertex[i])$: enqueue vertex i
// $TD(road\_id, come\_time)$: a function calculating the travel time from $road\_id$ and an arrival time $come\_time$
// $Datapoint$: a set of data objects
// $Result$: a set of the k nearest data objects
// $Result.insert(u)$: insert u to $Result$

1: **for** $i = 0$ to $b - 1$ **do**
2: **if** $start\_time \in [t_i, t_{i+1}]$ **then**
3: **for** $i = 1$ to n **do**
4: $vertex[i].travel\_time = \infty$;
5: $vertex[i].label = 0$;
6: **end for**
7: $q.label = 1$;
8: $q.travel\_time = 0$;
9: $q.cost = index[q].travel\_time[0]$;
10: $queue.insert(q)$;
11: **while** $queue$ is not empty and $Result.size() < k$ **do**
12: $sort(queue)$;
13: $u \leftarrow dequeue$;
14: $come\_time = u.travel\_time + start\_time$;
15: **for** $j = 1$ to $u.adjacent.size()$ **do**
16: $travel\_time = TD(e(u, u.adjacent[j]), come\_time) + u.travel\_time$;
17: **if** $travel\_time < u.adjacent[j].travel\_time$ **then**
18: $u.adjacent[j].travel\_time = travel\_time$;
19: **if** $u.adjacent[j].pointer.rank < c$ **then**
20: **while** $u.adjacent[j].pointer.data \in Result$ **do**
21: $u.adjacent[j].pointer = u.adjacent[j].pointer + 1$;
22: **end while**
23: **end if**
24: $u.adjacent[j].cost = travel\_time + u.adjacent[j].pointer.heuristic\_value$;
25: **if** $u.adjacent[j].label \neq 1$ **then**
26: $u.adjacent[j].label = 1$;
27: $queue.insert(u.adjacent[j])$;
28: **end if**
29: **end if**
30: **end for**
31: **if** $u \in Datapoint$ **then**
32: $Result.insert(u)$;
33: **end if**
34: **end while**
35: **end if**
36: **end for**
37: **return** $Result$

e is dequeued, and then, the vertexes b and d which are neighboring vertexes of e are enqueued. In the next step, the travel time of (q, d) is calculated as 5 min because the travel time of (e, d) is 3 min. Thus, the cost of d is 7 min (5 min + 2 min) since the heuristic value equals to 2 min because Store B is the nearest object from d. After the calculations, b is dequeued because b has the minimum cost in the queue, and then, the vertexes a and c are enqueued, which are the neighboring vertexes of b. As a result, store A is found as the nearest data object. During the processing, the index about store A is not referred at all after store A is found as the result. The vertexes are visited in order, and finally, the search is finished when store B is found as the 2nd nearest data object.



(a)Road network

<1st step>

Index in [t, t'] on e

Data object	Heuristic value
store A	3 min.
store B	5 min.

Index in [t, t'] on f

Data object	Heuristic value
store C	6 min.
store B	7 min.

<2nd step>

Index in [t, t'] on b

Data object	Heuristic value
store A	1 min.
store B	7 min.

Index in [t, t'] on d

Data object	Heuristic value
store B	2 min.
store A	6 min.

<3rd step>

Index in [t, t'] on a

Data object	Heuristic value
store A	2 min.
store B	6 min.

Index in [t, t'] on c

Data object	Heuristic value
store C	6 min.
store A	9 min.

<4th step>

Index in [t, t'] on h

Data object	Heuristic value
store B	1 min.
store C	5 min.

(b) References of indexes

<1st step>

Queue	Calculation	Cost
e	2 min. + 3 min.	5 min.
f	3 min. + 6 min.	9 min.

<2nd step>

Queue	Calculation	Cost
b	4 min. + 1 min.	5 min.
d	5 min. + 2 min.	7 min.
f	3 min. + 6 min.	9 min.

<3rd step>

Queue	Calculation	Cost
d	5 min. + 2 min.	7 min.
f	3 min. + 6 min.	9 min.
a	8 min. + 6 min. (A)	14 min.
c	13 min. + 6 min.	19 min.

<4th step>

Queue	Calculation	Cost
f	3 min. + 6 min.	9 min.
h	8 min. + 5 min. (B)	13 min.
a	8 min. + 6 min.	14 min.
c	13 min. + 6 min.	19 min.

(c) Priority queue

Figure 6. Example of 2NN search

V. Simulation Experiments

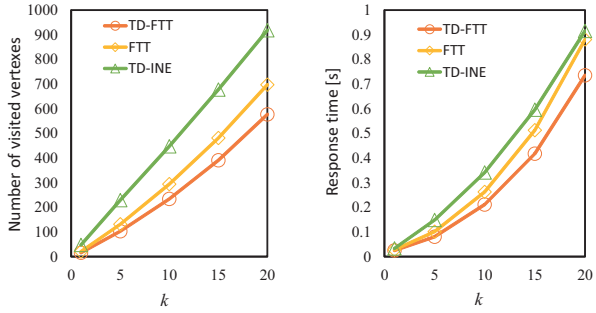
A. Simulation Model

We conducted experiments on the computer with Core i7 (3.4GHz) and 8GB main memory and used C++ to implement our method. We used Boost Graph Library 1.54.0 [1] for maintaining the network graph. To evaluate the performance with a real data set, we used the Osaka city map provided from the Japan Digital Road Map Association, and the car trace data provided from the Honda Motor Co., Ltd.[2]. The area size of the map is about 10km × 20km, and the number of intersections (vertexes) and roads (edges) are respectively 14,027 and 21,189. The points of data objects are randomly chosen from all the vertexes. We calculated the travel time on each edge based on the result of a map matched car trace data. For each edge, the travel time is defined as the average of the travel times of map matched cars par hour between 9 AM and 6 PM. Here, the travel times of edges on which no car or only a car passed from 9 AM to 6 PM, is applied for the cost model of the edge on which the largest number of cars passed.

[1] $http://www.boost.org/doc/libs/1\_54\_0/libs/graph/doc/index.html$
[2] $http://www.honda.co.jp/$

Table I
PARAMETER CONFIGURATION

Parameters	Values (Range)
The number of data objects	300 (100∼500)
k	10 (1∼20)



(a) Number of visited vertexes (b) Response time

Figure 7. Impact of k

We compare the performance of our proposed method with that of two different methods. The first method is TD-INE [3], and the other is our method which uses the minimum travel time in 9 hours (entire time) as the heuristic function (denoted by "FTT" in the graphs). In our proposed method (denoted by "TD-FTT" in the graphs), we divided 9 hours into 3 time segments (9 AM - 0 PM, 0 PM - 3 PM, and 3 PM - 6 PM), i.e., 3 indexes were constructed.

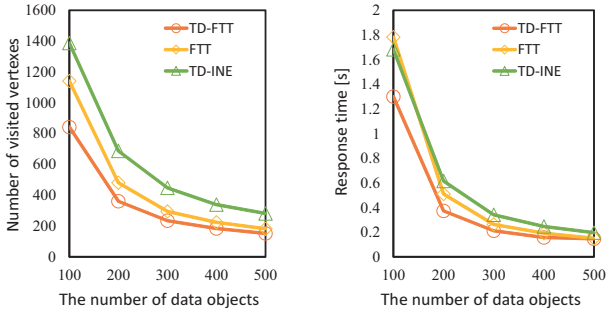
Table I shows the parameters used in the simulation experiments. Each parameter was set by default at the constant value to the left of its parenthetical range, and varied over this range in the simulations. The number of indexed data objects in each index, C, is 20.

The query point is randomly chosen among all the vertexes and a kNN query from the query point is processed. We repeated this process 100 times (i.e., 100 queries) and evaluated the following two criteria.

- Number of visited vertexes: We examine the number of vertexes visited until receiving the kNN result. We define "number of visited vertexes" as the average number of visited vertexes for all queries issued.
- Response time: We examine the time from issuing a query until receiving the kNN result. We define "response time" as the average time for all queries issued.

B. Impact of k

Figure 7 shows the simulation results with varying k. From Figure 7(a), in TD-FTT and FTT, the number of visited vertexes is smaller than that in TD-INE. This is because many unnecessary visits of vertexes are pruned by A* algorithm which does not expand the search space in all directions. Moreover, the number of visited vertexes in TD-FTT is smaller than that in FTT because the heuristic function in TD-FTT effectively works, i.e., it is closer to the actual travel time.



(a) the number of searched vertexes (b) the search time

Figure 8. Impact of the number of data objects

From Figure7(b), when k is 1, the differences of response time among all the methods are trivial. This is because the nearest data object is very close to the query point, and thus, the fastest path to the nearest data object can be immediately calculated in all methods. When k is large, the response time in TD-FTT is smaller than that in TD-INE because the number of visited vertexes is smaller. However, the differences of response time between TD-FTT and TD-INE is smaller than that of the number of visited vertexes (shown in Figure 7(a)). This is because in TD-FTT (and FTT), the response time includes the time for referring indexes.

C. Impact of the number of data objects

Figure 8 shows the simulation results with varying the number of data objects. From Figure 8(a), the number of visited vertexes in TD-FTT and FTT is smaller than that in TD-INE. This is because TD-INE adopts Dijkstra's algorithm, which makes much more unnecessary visits of vertexes than A* algorithm, especially when the number of data objects is small (i.e., the density of data objects is low).

From Figure 8(b), when the number of data objects is large, the response time in all methods is small because k nearest objects exists in a small area due to the high density of data objects. The response time in TD-FTT is smaller than that in TD-INE, but the differences are smaller than in Figure 8(a). This is due to the same reason described in subsection V-B. When the number of data objects is 100, the response time in FTT is larger than that in TD-INE. This is because in FTT, the time for referring indexes increases as the number of visited vertexes increases since the heuristic value is farther from the actual travel time. On the other hand, TD-FTT can reduce the response time because the number of visited vertexes is small by using A* algorithm with finer heuristic functions. This shows that indexing heuristic values for finer time segments is efficient because these values are closer to the actual travel times.

VI. CONCLUSION

In this paper, we proposed a kNN search method in time-dependent road networks to reduce the number of visited vertexes and response time. In our proposed method, a kNN

search is conducted with TD-FTT index and A* algorithm. In the preprocessing phase, an index is constructed for each time segment based on the minimum travel time in the time segment. The entries of indexes are CNNs from each vertex and the travel times through the fastest paths to the CNNs. In query processing, A* algorithm is executed with the indexes for the time segment including the query-issuing time. Starting from the query point, a vertex is visited toward a data object, which is indexed on the vertex and not already found, in order of cost (i.e., expected travel time) until k data objects are found. Here, the expected travel time is calculated by using the heuristic value of the data object in the index. The results of our experiments show that our proposed method reduces the number of visited vertexes and the response time compared with the existing method. Moreover, the results show that indexing heuristic values for finer time segments is efficient. As part of our future work, we plan to extend our method to handle continuous kNN search.

ACKNOWLEDGMENT

This research is partially supported by the Grant-in-Aid for Scientific Research A(2620013) of MEXT, Japan.

REFERENCES

[1] L.A. Cruz, M.A. Nascimento, and J.A.F. de Macedo, "k-nearest neighbor queries in time-dependent road networks," *Journal of Information and Data Management*, vol.3, no.3, pp.211-226, 2012.

[2] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Efficient K-nearest neighbor search in time-dependent spatial networks," *Proc. Int'l Conf. on Database and Expert Systems Applications*, vol.1, pp.432-449, 2010.

[3] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Towards K-nearest neighbor search in time-dependent spatial network databases," *Databases in Networked Information Systems*, vol.5999, pp.296-310, 2010.

[4] U. Demiryurek, F. Banaei-Kashani, C. Shahabi, and A. Ranganathan, "Online computation of fastest path in time-dependent spatial networks," *Proc. Int'l Conf. on Advances in Spatial and Temporal Databases*, pp.92-111, 2011.

[5] E. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol.1, no.1, pp.269-271, 1959.

[6] S.E. Dreyfus, "An appraisal of some shortest-path algorithms," *Operations Research*, vol.17, no.3, pp.395-412, 1969.

[7] J. Halpern, "Shortest route with time dependent length of edges and limited delay possibilities in nodes," *Zeitschrift fur Operations Research*, vol.21, no.3, pp.117-124, 1977.

[8] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. on Systems Science and Cybernetics*, vol.4, no.2, pp.100-107, 1968.

[9] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," *Proc. Int'l Conf. on Very Large Data Bases*, vol.29, pp.802-813, 2003.