

k -Nearest Neighbors Queries in Time-Dependent Networks

Lívia A. Cruz, Mario A. Nascimento and José A. F. de Macêdo

¹ Federal University of Ceará, Brazil
{liviaac, jose.macedo}@lia.ufc.br
² University of Alberta, Canada
{mario.nascimento}@ualberta.ca

Abstract. In this paper, we study the problem of processing K -nearest neighbors (KNN) queries in road networks considering traffic conditions, in particular the case where the road speed is time-dependent. For instance, given that the user is at a given location, the query returns the K points of interest (e.g., gas stations) that could be reached in the minimum amount of time. Previous works have proposed solutions to answer KNN queries in road networks where the speed in each road is constant. Obviously, these solutions cannot be simply applied to the problem we are interested in. Our approach extends the well-known A* search algorithm by applying incremental network expansion and pruning undesirable vertices. We discuss the design and correctness of our algorithm and present experimental results that show the efficiency and effectiveness of our solution.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.2.4 [**Database Management**]: Systems—*Query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

Keywords: K -Nearest Neighbors Queries, Query Processing, Spatial Pruning, Time-Dependent Networks

1. INTRODUCTION

Travel time on road networks heavily depends on the traffic and typically the time it takes to traverse a segment depends on departure time. To exemplify, consider Figure 1(a) where a partial network is seen. The structure of a network can be modeled by a graph where the vertices represent the network junctions, starting and ending points of a road segment (e.g. a street, an avenue) and, depending on the application, additional points can represent a change in curvature or maximum speed of a segment; the edges connect vertices. The travel time is modeled by a time-dependent graph, where a function of time models the cost to traverse an edge at a specific departure time. Figure 1(b) shows us a graph representing the network in Figure 1(a). The travel time is given by functions, shown in Figure 1(c). Each edge in the graph has its respective travel time function.

Figure 1 exemplifies time-dependent travel time. There are two paths to go from vertex b to c . One can take a path $\langle b, c \rangle$, that goes from b to c directly, or $\langle b, a, c \rangle$, that pass by a . The fastest path from b to c depends on departure time t_s . Let $t_s = 10h$, path $\langle b, c \rangle$ takes $30min$. In path $\langle b, a, c \rangle$ travel from b to a , takes $20min$ and the arrival time in a is $10h20min$. The time to traverse $\langle a, c \rangle$ at $10h20min$ is $25min$, then a path $\langle b, a, c \rangle$ takes $45min$, for departure time $10h$. Similarly, if $t_s = 16h$, a path $\langle b, c \rangle$ takes $50min$ and a path $\langle b, a, c \rangle$ takes $30min$. It is fastest go directly to c in the morning, but at $16h$ the best choice is go through a and then to c .

Unfortunately, previous solutions for shortest paths, k -nearest neighbors and others common queries

Copyright©2012 Permission to copy without fee all or part of the material printed in JIDM is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

in static networks no longer work when the costs (travel time) itself depends on time. Speed-up techniques for original shortest path problem have been developed, see [Wagner and Willhalm 2007] for a survey, but they cannot be applied directly in time-dependent networks because their correctness depends on the fact of network edges costs be constant.

This paper is organized as follows. In Section 2, we introduce some important definitions, explain the road network model used in our approach and formalize the problem of processing KNN queries in time-dependent networks. In Section 3, a brief description of related works is presented. In Section 4, we explain our approach and show the correctness of our algorithm. The experimental evaluation and results are showed in Section 5. Finally, Section 6 concludes this paper.

2. PRELIMINARIES

In this section, we formalize the concept of time-dependent graph and explain how we model the points of interest on a network. We give others basic definitions as travel-time, fastest path and time-dependent distance, useful for defining the problem.

Definition 2.1. A Time-Dependent Graph (TDG) $G = (V, E, C)$ is a graph where: (i) $V = \{v_1, \dots, v_n\}$ is a set of vertices; (ii) $E = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$ is a set of edges; (iii) For all $(v_i, v_j) \in E$, exists $c_{(v_i, v_j)}(t) \in C$ where $c_{(v_i, v_j)} : \Pi(G) \rightarrow R^+$ and is a function which attributes a positive weight for (v_i, v_j) depending on a time instant $t \in \Pi(G)$.

Basically, a TDG is a graph where the edge costs varying with the time. For each edge (u, v) , a function $c_{(u,v)}(t)$ gives the cost of traverse (u, v) at departure time t . We call $\Pi(G)$ a interval time when $c_{(u,v)}(t) \in C$ is defined, that represents the system time. For example, we can suppose $\Pi(G) = [0h, 24h)$, that means $c_{(u,v)}(t)$ is defined for each instant of a day.

Note that our definition allows different edge costs to bidirectional segments, such that given a edge (u, v) and its opposite (v, u) , it is possible that $c_{(u,v)}(t) \neq c_{(v,u)}(t)$. Furthermore, we assume that C

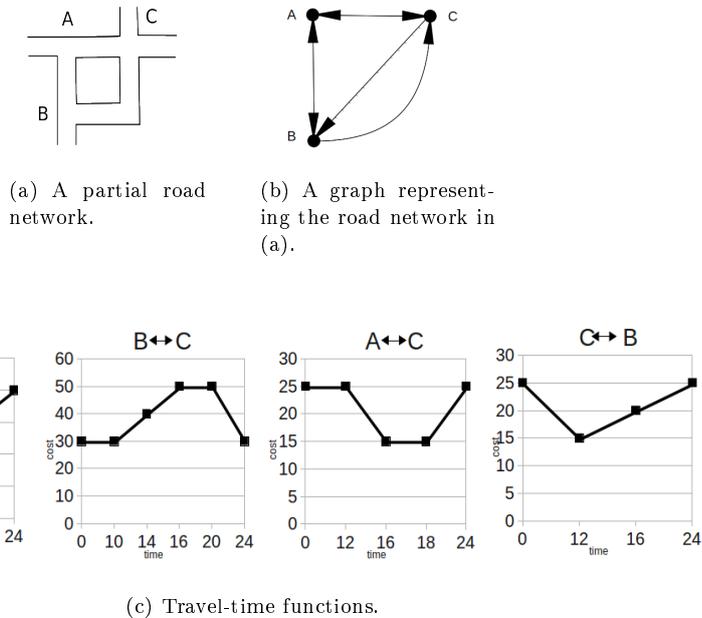


Fig. 1. A road network example, its equivalent graph and travel-time functions for their edges.

Algorithm 1: IncludePOI

Input: A TDG $G = (V, E, C)$, a POI $p = \langle (u, v), \tau_p \rangle$

```

Include  $v_p$  in  $V$ ;
Remove  $(u, v)$  from  $E$ ;
Include  $(u, v_p)$  and  $(v_p, v)$  in  $E$ ;
 $c_{(u, v_p)}(t) \leftarrow \tau_p \times c_{(u, v)}(t)$ ;
 $c_{(v_p, v)}(t) \leftarrow (1 - \tau_p) \times c_{(u, v)}(t)$ ;
Remove  $c_{(u, v)}(t)$  from  $C$ ;
Include  $c_{(u, v_p)}(t)$  and  $c_{(v_p, v)}(t)$  in  $C$ ;
if  $(v, u) \in E$  then
    Remove  $(v, u)$  from  $E$ ;
     $c_{(v, v_p)}(t) \leftarrow (1 - \tau_p) \times c_{(v, u)}(t)$ ;
     $c_{(v_p, u)}(t) \leftarrow \tau_p \times c_{(v, u)}(t)$ ;
    Remove  $c_{(v, u)}(t)$  from  $C$ ;
    Include  $c_{(v, v_p)}(t)$  and  $c_{(v_p, u)}(t)$  in  $C$ ;
end

```

is a set of piecewise-linear functions that satisfy the FIFO property. Basically, FIFO property states that if an object A starts traverse an edge before an object B, then A have to finish traverse that edge before B. The time-dependent shortest path problem has a polynomial solution in FIFO networks, however it is NP-hard in non FIFO-networks [Orda and Rom 1990].

We represent any location, like points of interest or query points, in a TDG is by a pair $p = \langle (u, v), \tau_{(u, v)} \rangle$, such that (u, v) is an edge in G where p is on and $\tau_{(u, v)} = \frac{d(u, p)}{d(u, v)}$, $d(p_i, p_j)$ is the euclidean distance between p_i and p_j , if p_i, p_j are on the same edge, and undefined, otherwise. If the edge (u, v) has an opposite edge (v, u) , the same point could have two different representations, e.g. $p = \langle (u, v), \tau_p \rangle$ and $p = \langle (v, u), 1 - \tau_p \rangle$, one for each direction. To simplifies, we consider that only one of them is given and another can be directly obtained from it.

We assume a process that generates a TDG from a road network. Given a TDG $G = (V, E, C)$ generated by this process and a set S of points of interest (POIs), we include each point of interest in G as a vertex. Algorithm 1 is a procedure that receives as input a TDG G and a point of interest $p = \langle (u, v), \tau_p \rangle$, generates a vertex in G equivalent to p and calculates the cost function for new edges. Figure 2 presents an example of how to include a point of interest as a vertex in the TDG in Figure 1(b). Figure 1(c) presents how the travel-time functions of the same TDG change according by a new vertex included. In this example, a restaurant $R = \langle (B, C), \frac{1}{3} \rangle$ is a point of interest of application.

First, we have to include R as a vertex in V . As R is a point on (B, C) , (B, C) and is removed from E to originate two new edges (B, R) and (R, C) . The travel time functions for (B, R) and (R, C) are $c_{(B, R)} = \frac{1}{3}c_{(B, C)}$ and $c_{(R, C)} = \frac{2}{3}c_{(B, C)}$, respectively, and the old travel time function $c_{(B, C)}$ is removed from C . Now, we have to check if (C, B) is an edge in E . In that case, similarly, we have to remove (C, B) from E and $c_{(C, B)}$ from C and edges in the other direction, (C, R) and (R, B) , and calculate the function cost for another direction. To calculate the new functions, we consider the complement of $1 - p_R$ and $c_{(C, B)}$ instead of p_R and $c_{(B, C)}$. The new cost functions are $c_{(C, R)} = \frac{2}{3}c_{(C, B)}$ and $c_{(R, B)} = \frac{1}{3}c_{(C, B)}$. Note that this process assumes there is no velocity variations when an object travel an edge. This assumption is acceptable since we suppose only have information about the travel time to go through a entire edge.

The temporal cost to execute a path at a specific departure time in a TDG, called travel time, and the arrival time of that path are calculated supposing that stops are not permitted. They are formally defined as follows.

Definition 2.2. Arrival-time (AT): Given a TDG $G = (V, E, C)$, the arrival-time when an edge

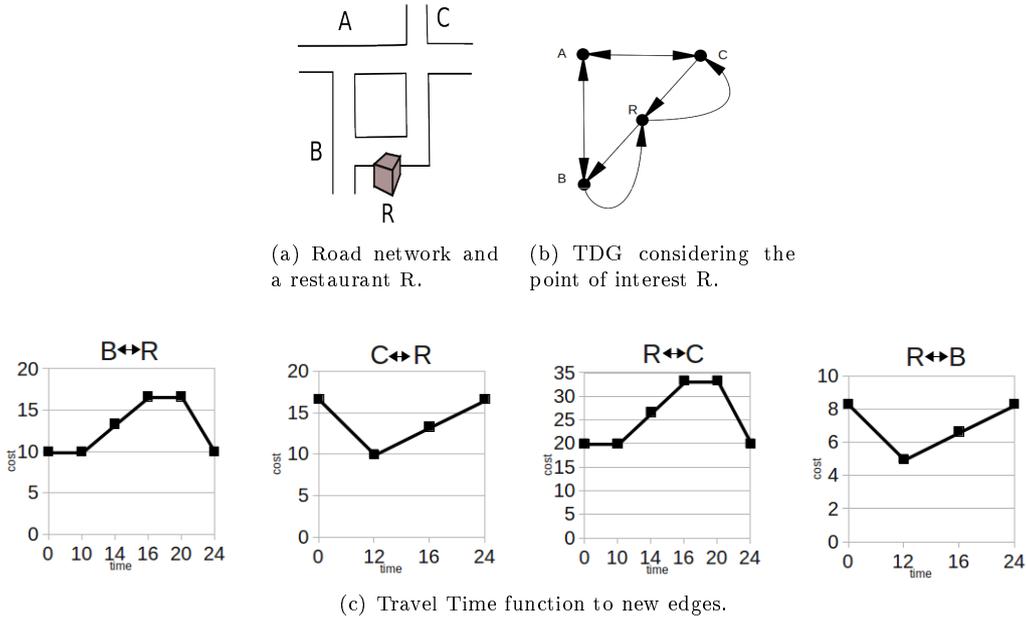


Fig. 2. Time-dependent graph representing a network and points of interest.

$(v_i, v_j) \in E$ is crossed at departure $t \in \Pi(G)$ is given by $AT(v_i, v_j, t) = t + c_{(v_i, v_j)}(t) \bmod \Pi(G)$.

Definition 2.3. Travel-time (TT): Given a TDG $G = (V, E, C)$, a path in G , $p = \langle v_{p_1}, \dots, v_{p_i}, v_{p_{i+1}}, \dots, v_{p_k} \rangle$ and a departure time $t \in \Pi(G)$, the travel-time of p is a time-dependent cost to execute this path, given by $TT(p, t) = \sum_{i=1}^{k-1} c_{(v_{p_i}, v_{p_{i+1}})}(t_i)$, where $t_1 = t$ and $t_{i+1} = AT(v_{p_i}, v_{p_{i+1}}, t_i)$.

Using TT as the cost of a path, new versions of shortest path and distance are given. A Time-Dependent Fastest Path $TDFP(u, v, t)$ is the path between u and v starting at t with minimum TT. In this case, the travel time is called Time-Dependent Distance (TDD).

2.1 Problem Statement

We consider the problem of processing KNN queries in road networks where the speed is time-dependent. For instance, given that the user is at a given location, at a specific time instant, the query returns the K points of interest (e.g. gas station) that could be reached in the minimum amount of time.

Definition 2.4. Let $G = (V, E, C)$ a TDG and $POI \subseteq V$ a set of points of interest in G . Given a query point q and a departure time t , a Time-Dependent KNN query returns a set $R = \{v_{r_1}, \dots, v_{r_k}\} \subseteq POI$ such that $\forall v \in POI \setminus R, TDD(q, v_{r_i}, t) \leq TDD(q, v, t), 1 \leq i \leq k$. In other words, a TD-KNN query returns a k of points of interest that are closest from q than the others points considering a departure time t .

Previous works have proposed solutions for answer KNN queries in road networks where the speed in each road is constant. Obviously, these solutions cannot be simply applied to the problem we are interested in.

3. RELATED WORK

3.1 Time-Dependent Shortest Path

The more usual solution to shortest path problem in static graphs is Dijkstra's algorithm [Dijkstra 1959]. [Wagner and Willhalm 2007] presents many others ideas that have been proposed to find point-to-point shortest paths. Unfortunately, these ideas would fail when time-dependent networks are considered. Much less work have been proposed to this case. The first algorithm that considers time-dependent variant of shortest paths is addressed by [Cooke and Halsey 1966], it extends Dijkstra's algorithm to dynamic case, based on assumption that the FIFO property holds in the network. Basically, FIFO property states if an object A starts cross a edge before an object B, then A have to finish crossed that edge before B. The time-dependent shortest path problem has a polynomial solution in FIFO networks, however it is NP-hard in non FIFO-networks [Orda and Rom 1990]. [Nannicini et al. 2008] proposed an algorithm that applies bidirectional search on a time-dependent network to calculate the shortest path between two vertices, their method is based on A* search with landmarks.

3.2 KNN Queries in Spatial Networks

The problem of KNN queries in spatial networks was introduced by [Papadias et al. 2003]. In that paper, the authors present two different solutions to this problem, the Incremental Euclidean Restriction (IER) and the Incremental Network Expansion algorithms (INE). IER uses the assumption that the euclidean restriction between two points on network is less than the network distance. This assumption allows to use their network distance of euclidean KNN points as a upper bound. INE is an adaptation of Dijkstra's algorithm. [Kolahdouzan and Shahabi 2004] presented an approach based on pre-computing the network voronoi polygons (NVP) [Erwig and Hagen 2000], indexed by a spatial access method. Using NVPs one can immediately find the first nearest neighbor of a query object and reduce the on-line cost in a KNN search. All these approaches cannot be directly applied to solve TD-KNN queries.

3.3 Time-dependent KNN (TD-KNN) Queries

The problem of KNN queries in time-dependent networks was introduced by [Demiryurek et al. 2010], where the authors compares two different baseline methods to solve this problem. The first approach use time-expanded graphs to model the network. Time-expanded graphs allows us exploit previous solutions in static networks to solve TD-KNN queries. However, this solution has numerous shortcomings, as high storage overhead, slower response time and correctness of results, as showed by [Demiryurek et al. 2010]. The second approach is an adaptation of INE algorithm [Papadias et al. 2003] that does a blind search expanding while expanded the network. [Demiryurek et al. 2011] proposed a pre-computation process that builds two different indexes structures, the Tight Network Index (TNI) and Loose Network Index (LNI). Both are composed for cells that reference the points of interest such that, if a query point is in a tight cell of a point P, P is its nearest neighbor, and if a q is out of a loose cell of P, P is not its nearest neighbor. As in NVP method, using TNI one can immediately find the first nearest neighbor of a query object. However, it is not clear how this process could work well when travel time functions in edges with opposite directions can be different. That is a important aspect in time-dependent networks, since the cost of a path can be determined by its orientation.

4. A*-TIME DEPENDENT INCREMENTAL NETWORK EXPANSION (A*-TD-NE)

In this section, we describe our approach to process TD-KNN queries in time-dependent networks. Our algorithm is based on incremental network expansion (INE), that was originally proposed for static network in [Papadias et al. 2003]. With INE, starting from the query object *q*, all network

vertices reachable from q are visited in order of their proximity until all k nearest neighbors objects are located.

We incorporate a A^* search directly in an INE expansion. To solve the shortest path problem, A^* search works similar to Dijkstra's algorithm. It uses a distance $d(v_i, v_k)$ plus a heuristic function $H(v_k)$ to determine the order in vertices are expanded in the search tree. The current distance plus a heuristic function on a vertex v_k is an estimate of the cost of a path between v_i and v_j that pass by v_k . The heuristic function H must be an admissible heuristic; that is, it must not overestimate the distance to the goal. If H satisfies an additional condition $H(x) \leq d(x, y) + H(y)$ for every edge x, y of the graph, then H is called monotone, or consistent. In such a case, A^* can be implemented more efficiently because that approaches the solution in an incremental way without taking any step back. A incremental expansion algorithm maintains the candidate vertices to expansion in order by their distance from q .

Instead of to use an A^* search to calculate each time-dependent distance from a query q to each point of interest in a set of candidates, we incorporate a A^* search directly in an incremental network expansion. An incremental strategy avoids re-compute costs previously calculated. In our algorithm, the heuristic function adds to each vertex an estimate of potential to it takes part of the fastest path that leads to a nearest point of interest. The idea behind our search is avoid to continue the expand nodes in a path that is fastest but is far from any point of interest in the network. We are motivated by the fact that a vertex u be the closest node from q does not imply finding a next nearest neighbor when expanding u . To explain how our method works, we need introduce some definitions.

Definition 4.1. Lower Bound Graph \underline{G} and Upper Bound Graph \overline{G} : The Lower Bound Graph of a TDG $G = (V, E, C)$ is a graph $\underline{G} = (V, E, \underline{C})$ where V and E are the same set of vertices and edges in G and \underline{C} is a set of costs $\underline{c}_{v_i v_j} = \min_{t \in \Pi} \{c_{v_i v_j}(t)\}$, for all $c_{v_i v_j} \in C$. Similarly, an Upper Bound Graph $\overline{G} = (V, E, \overline{C})$ has the same set of vertices and edges in G and \overline{C} is a set of edges costs $\overline{c}_{v_i v_j} = \max_{t \in \Pi} \{c_{v_i v_j}(t)\}$, for all $c_{v_i v_j} \in C$.

We define the $LTDD(v_i, v_j)$ and $UTDD(v_i, v_j)$ as the travel time of the fastest path between v_i and v_j in \underline{G} and \overline{G} , respectively. Note that, as the cost functions in \underline{G} and \overline{G} are constants, $LTDD(v_i, v_j)$ and $UTDD(v_i, v_j)$ are not dependent of a departure time. We set our heuristic function $H(u)$ to be equals the travel time from u to its nearest neighbor in \underline{G} . The two followed lemmas prove that $H(u)$ is admissible and consistent, thus it is feasible to be used in the A^* search.

LEMMA 4.2. H is an admissible heuristic.

PROOF. To show that H is admissible consider a query point $q = \langle (u, v), \tau_q \rangle$ and let nn_q be the nearest neighbor of q at departure time t in G . Let u be a vertex visited in the search and nn_u the nearest neighbor from u in \underline{G} . Suppose that $H(u)$ is not admissible, so we have $TDD(q, u, t) + H(u) = TDD(q, u, t) + LTDD(u, nn_u) > TDD(q, u, t) + TDD(u, nn_q, AT(q, u, t))$. The inequality $LTDD(u, nn_u) > TDD(u, nn_q, AT(q, u, t))$ is a contradiction because $LTDD(u, nn_u)$ is a lower bound to all possible travel times from u to any point of interest. We can conclude that $H(u)$ is admissible, thus does not overestimate the distance to next nearest neighbor object. \square

LEMMA 4.3. H is a consistent heuristic.

PROOF. Given a TDG $G = (V, E, C)$, and a set of points of interest S to show that H is consistent for any departure time t w.r.t S , we need show that for any edge $(u, v) \in E$, $H(u) \leq c_{(u,v)}(t) + H(v)$. Let nn_u be the nearest neighbor of u in \underline{G} . Suppose that $H(u) > c_{(u,v)}(t) + H(v)$. Hence, $LTDD(u, nn_u) = H(u) > c_{(u,v)}(t) + H(v) > \underline{c}_{(u,v)} + H(v) = LTDD(u, p)$, for some vertex $p \in S$. In

that case, p is a point of interest nearest than nn_u from u , a contradiction against the hypothesis that nn_u would be the nearest neighbor of u . \square

We consider two preprocessing steps in our approach. In both of them an algorithm to find the nearest point of interest in a network that does not varies with time is used. The first one calculates the heuristic function value for each vertex in G . For each vertex v , the distance between v and its nearest neighbor (NN) in G is calculated. This distance is attributed to $H(v)$ to be used as the heuristic function. Another step computes the NN of v in \overline{G} , denoted by $UNN(v)$, and its distance from v in \overline{G} , denoted by $UTDD(v, UNN(v))$. These values are used to prune vertices that lead to points of interest furthest that a set of candidates.

The algorithm takes three parameters as input, the query object $q = \langle (u, v), \tau_q \rangle$, the number of nearest neighbors k and the departure time t . It works similar to previous network expansion algorithm, but includes two strategies to guide the search and prune undesirable vertices. First, it gets the edge (u, v) that represents the road covering q . Then it calculates the travel time between q and each vertex of (u, v) , at departure time t , $TT(q, u, t)$ and $TT(q, v, t)$. The vertices u and v are inserted in a priority queue Q , that stores the set of candidates to expand in next step.

For each vertex v_i in Q , the travel time from q to v_i , $TT_{v_i} = TT(q, v_i, t)$, arrival time $AT_{v_i} = AT(q, v_i, t)$ and the lower bound of travel time from it to its nearest neighbor $L_{v_i} = TT_{v_i} + H(v_i)$ are calculated. An entry in Q queue is like $(v_i, AT_{v_i}, TT_{v_i}, L_{v_i})$ and the elements of Q are ordered by L_{v_i} values.

Another priority queue Q_U is maintained to store upper bounds values. More precisely, if a vertex v is expanded, we check if $NN(v)$ is in Q_U . If it is not, we include $NN(v)$ in Q_U , Q_U is ordered by increasing order of $UTDD(v, NN(v))$. If $NN(v)$ is already included in Q_U , we check if $UTDD(v, NN(v))$ is less than the old upper bound to it, we update the position and upper bound value of $NN(v)$ in Q_U . Q_U is used to pruning process, more specifically if L_{v_i} is greater than the k th upper bound in Q_U , it can be discarded.

4.1 Running Example

4.1.1 Offline Pre-processing. A result of our preprocessing step is showed in figure 3(b). For each vertex, a not time-dependent NN search is executed in graphs G and \overline{G} . For example, consider vertex f in figure 3(a). The distance between f and b in G is $LTDD(f, b) = 6$ and between f and d is $LTDD(f, d) = 8$, thus $H(f) = 6$. Furthermore, the distance between f and b in \overline{G} is $UTDD(f, b) = 8$ and between f and d is $UTDD(f, d) = 9$, thus $UNN(f) = b$ and $UTDD(f, NN(f)) = 8$.

4.1.2 Query processing. As an example, consider the graph in Figure 3. The inputs for the algorithm are a query point $Q = \langle (f, e), \frac{1}{4} \rangle$, $k = 1$, and the departure time $t = 0$. First, the algorithm calculates the travel times T_e and T_f from q to f and e , the the labels L_f and L_e , and the arrival times AT_f and AT_e . Then it initializes the queues $Q = \langle (f, AT_f = 1, T_f = 1, L_f = 7), (e, AT_e = 3, T_e = 3, L_e = 7) \rangle$ and $Q_U = \langle (d, 8), (b, 9) \rangle$, because $TT(q, e, t) + UTDD(e, d) = 8$ and d is the nearest neighbor from e in \overline{G} and $TT(q, f, t) + UTDD(f, b) = 9$ and b is the nearest neighbor from f in \overline{G} . Now, each iteration remove a vertex from Q to be expanded.

The first vertex expanded is f , its adjacency vertices not yet en-queued are a and g , the new entries are $(a, AT_a = 4.5, TT_a = 4.5, L_a = 7.5)$ and $(g, AT_g = 3.5, TT_g = 3.5, L_g = 9.5)$. As $L_g > Q_U(k)$, the entry correspondent to g is not en-queued. As $UNN(a) = b$ and $TT_a + UTDD(a, b) = 8.5$, upper bound less than 9.5 is found, so we have to update to $Q_u = \langle (d, 8), (b, 8.5) \rangle$.

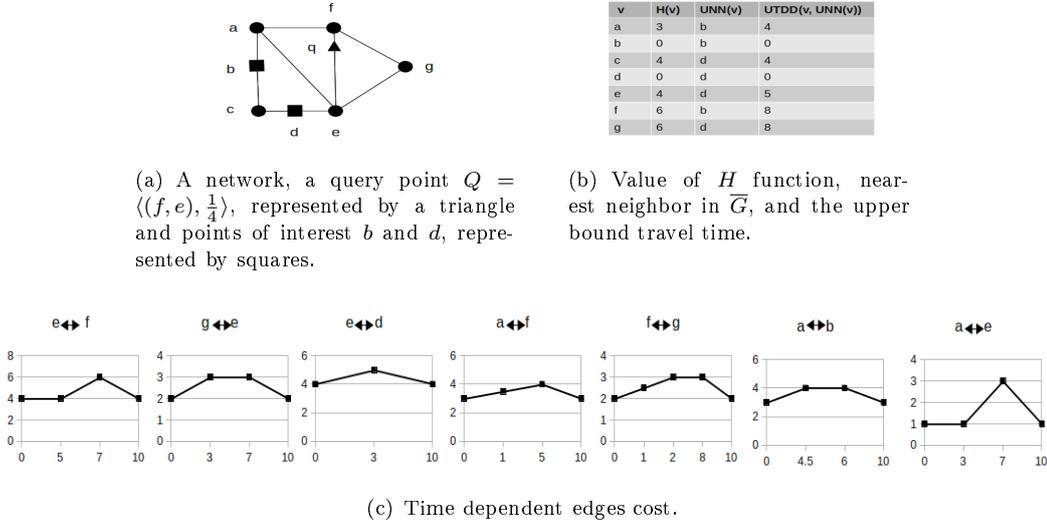
The second vertex expanded is e , the entries to its adjacency vertices are $(g, AT_g = 6, TT_g = 6, L_g = 12)$, $(d, AT_d = 8, TT_d = 8, L_d = 8)$ and $(a, AT_a = 4, TT_a = 4, L_a = 7)$. Vertex g can be discarded because $L_g > Q_U(k)$ and a has to be updated because it was found by a fastest path than before. Fur-

Algorithm 2: TD-NE- A^*

Input: A query point $q = ((u, v), \tau_q)$, an integer value k , a departure time t
Output: The set of k nearest neighbors of q

$TT_v \leftarrow \tau_q \times c_{(u,v)}(t);$
 $AT_v \leftarrow (t + TT_v) \bmod Pi;$
 $L_v \leftarrow T_v + H(v);$
En-queue (v, AT_v, TT_v, L_v) in $Q;$
if $(v, u) \in E$ **then**
 $TT_u \leftarrow (1 - \tau_q) \times c_{(v,u)}(t);$
 $AT_u \leftarrow (t + TT_u) \bmod Pi;$
 $L_u \leftarrow T_u + H(u);$
 En-queue (u, AT_u, TT_u, L_u) in $Q;$
end
 $S_{NN} \leftarrow \emptyset;$
while $Q \neq \emptyset \wedge |S_{NN}| < k$ **do**
 $(u, AT_u, TT_u, L_u) \leftarrow$ De-queue $Q;$
 Mark u as de-queued;
 if $TT_u = H(u)$ **then**
 // u is a point of interest;
 $S_{NN} \leftarrow S_{NN} \cup \{u\};$
 end
 for $v \in \text{adjacency}(u)$ **do**
 $TT_v \leftarrow TT_u + c_{(u,v)}(AT_u);$
 $AT_v \leftarrow (t + TT_v) \bmod Pi;$
 $L_v \leftarrow TT_v + H(v);$
 if $L_v \leq Q_U(k)$ **then**
 if v is not in Q **then**
 En-queued (v, AT_v, TT_v, L_v) in $Q;$
 Mark v as en-queued;
 else
 Update Q if according by $TT_v;$
 Re-order $Q;$
 end
 if $NN(v)$ is not in Q_U **then**
 $U_{NN(v)} \leftarrow TT_v + UTDD(v, NN(v));$
 En-queued $(NN(v), UTDD(v, U_{NN(v)}))$ in $Q_U;$
 else
 $U_{NN(v)} \leftarrow TT_v + UTDD(v, NN(v));$
 Update Q_U if according by $(NN(v), U_{NN(v)});$
 Re-order $Q_U;$
 end
 end
 end
 Mark v as visited;
end
Return $S_{NN};$

thermore, Q_U is updated, its new configuration is $Q_u = \langle (d, 8), (b, 8) \rangle$. At this point, the state of queue Q is $Q = \langle (a, AT_a = 4, TT_a = 4, L_a = 7), (d, TT_d = 8, TT_d = 8, L_d = 8) \rangle$. The next vertex expanded is a . Although its adjacency vertex b is a point of interest, it is discarded because $L_b = 8.5 > Q_U(k)$, that means b can be found in a fastest path. After this we have $Q = \langle (d, AT_d = 8, TT_d = 8, L_d = 8) \rangle$, d is expanded and included into S_{NN} . As $k = 1$ the algorithm finishes.



(a) A network, a query point $Q = ((f, e), \frac{1}{4})$, represented by a triangle and points of interest b and d , represented by squares.

(b) Value of H function, nearest neighbor in \overline{G} , and the upper bound travel time.

Fig. 3. A road network and its respective graphs considering points of interest.

THEOREM 4.4. Let $S_{NN} = \{v_{NN_1}, \dots, v_{NN_k}\}$ the set of points of interest returned by $TD-NE-A^*(q, k, t)$. S_{NN} is a set of the k nearest-neighbors from $q = (a, b, \tau_q)$ at departure time t .

PROOF. To show that, it is enough show that when a point of interest r is removed from Q : (i) its label T_r has the same value of travel time of the time-dependent fastest path from q to it and (ii) there is no point of interest that has a fastest path from q faster than $TDD(q, r, t)$.

- (1) We will prove it by induction in the number of dequeued vertices. The case base is the first vertex removed from Q , this case is trivial. Now, suppose that the i -th removed vertex the statement is true, for all $1 \leq i \leq l - 1$. Let be v the l -th dequeued vertex. Suppose that $T_v = TDD(q, v, t)$. Note that, there is a last vertex dequeued such that is in the fastest path from q to v . Let z be this vertex. The next vertex in the fastest path from q to v is a neighbor of z . Let w be this vertex. We have $L_w = TDD(q, z, t) + c_{(z,w)}(AT(q, z, t)) + H(w) = TDD(q, w, t) + H(w) < TDD(q, w, t) + TDD(w, v, AT(q, v, t)) + H(v) < TDD(q, u, t) + TDD(u, v, AT(q, u, t)) + H(v) = L_v$. In this case, L_w was removed before L_v . A contradiction against the fact that z was a last removed node in the fastest path. Thus, v was found by the fastest path and $T_v = TDD(q, v, t)$.
- (2) Now, suppose by contradiction that a point of interest r was removed and there is another point of interest r^* such that $TDD(q, r^*, t) < TDD(q, r, t)$ holds. Let v the last node enqueued in the fastest path between q and r^* . According by (1) $T_r = TDD(q, r, t)$. As r was removed before v , $L_r = TDD(q, r, t) \leq T_v + H(v) = TDD(q, u, t) + TDD(u, v, AT(q, u, t)) + H(v)$, where u is the node removed before v is enqueued. Hence, $L_r \leq TDD(q, u, t) + c_{(u,v)}(AT(q, u, t)) + H(v) = TDD(q, v, t) + H(v) \leq TDD(q, v, t) + TDD(v, r^*, AT(q, v, t)) = TDD(q, r^*, t)$. Thus $L_r \leq TDD(q, r^*, t)$, an absurd against hypothesis that the path between q and r^* is more fastest than the path between q and r . Thus $TDD(q, r, t) = TDD(q, r^*, t)$ and (2) holds.

□

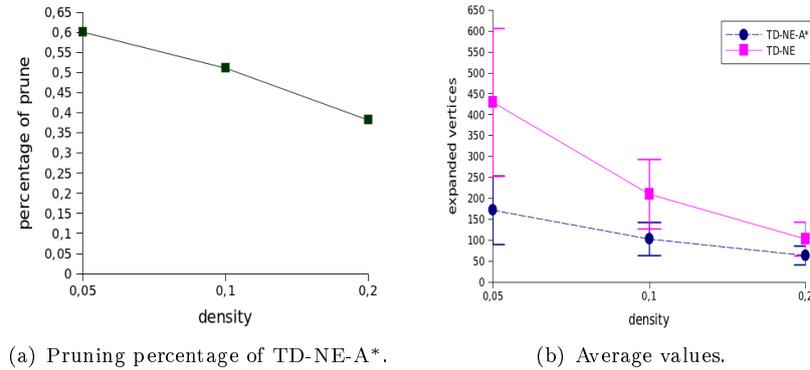


Fig. 4. Comparison when density of pois increases.

5. EXPERIMENTAL EVALUATION

5.1 Setup

We implemented an experimental system in C++ to compare our approach (TD-NE-A*) with the approach proposed by [Demiryurek et al. 2010] (TD-NE). We conducted our experiments on Unix station with xxx CPU and 4 GB main memory. We generated synthetic time-dependent road networks with temporal resolution of 96 points in time, a point at every 15 minutes of a day and average degree 4. We evaluated how our approach works according by number of vertices v , query size k and the density of pois (i.e., the ratio of pois cardinality to vertex cardinality).

5.2 Prune Power

In the first experiments, we are interested to evaluate the prune power of our approach. We compare the number of expanded nodes in the network with respect to the density of pois and query size. We believe there is a strong correlation between the number of expanded vertices and the number of accessed disk pages, when we have a disk-based implementation. In all of these experiments, for each parameters configuration we generated 10 distinct time-dependent road network and executed 10 queries randomly selected for each network, a total of 100 queries.

Effect of density First we examine the influence of density in the number of expanded vertices. We set the density of networks to be 5%, 10% and 20% of points of interest uniformly distributed. For each density, we generate 10 distinct time-dependent networks with 2000 vertices and executed 10 randomly selected queries with $k=20$ on each one. In our experiments TD-NE-A* pruned from 38% to 60% of nodes expanded by TD-NE. Figures 4(a) illustrates how density interferes the percentage of vertices pruned. We can observe that the pruning power of TD-NE-A* decreases as the network becomes denser. The reason for that the number of candidate pois to be nearest that increases with the density. Thus the quality of heuristic function decreases and the number of false hits increases. In the same way, in both algorithms is necessary to expand less vertices when the network becomes denser. This is reasonable since the number of points of interest grows for each evaluated subnetwork. Figures 4(b) illustrates the difference between the average of expanded vertices in both algorithms. The errors bars illustrate upper and lower 95% confidence limits, assuming the data to be normally distributed. That indicate our heuristic becomes less trusted when the density grows.

Effect of query size In order to evaluate the effect of query size, we generated 10 distinct networks with 2000 vertices and 10% of points of interest. To each one of them, we executed 10 queries randomly selected with $k = \{1, 10, 20, 30\}$. Figure 5(a) shows the average percentage of expanded vertices by TD-NE-A*. For all values of k , TD-NE-A* outperforms TD-NE in number of expanded vertices in more

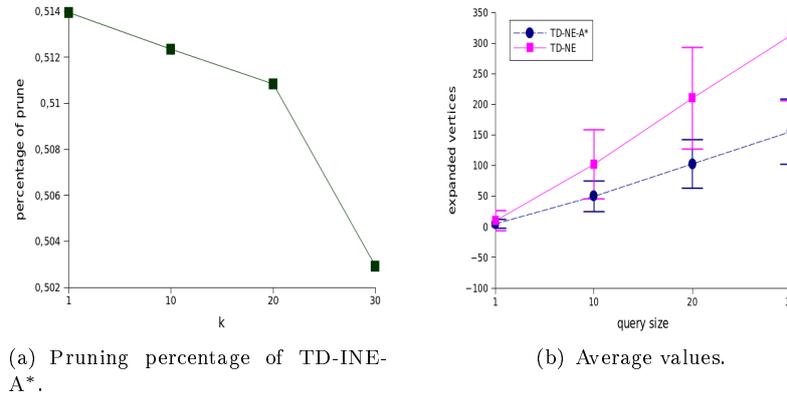


Fig. 5. Comparison when query size increases.

than 50%. Furthermore, when *k* grows the gap between TD-NE-A* and TD-NE also increases. Figure 5(b) compares the average number of expanded vertices when *k* increases. The number of expanded vertices increases with *k* because more vertices have to be checked to find more pois. The errors bars illustrate upper and lower 95% confidence limits, assuming the data to be normally distributed. That indicate our heuristic becomes more trusted when the *k* grows.

Effect of network size In this experiment, we generate 10 time-dependent networks with 1000, 2000 and 4000 vertices. Each one with 10% of points of interest. We executed 10 randomly selected queries with *k*=20 on each network. Figure 6(a) shows that the prune percentage of TD-NE-A* grows with the network size. Figure 5.2 illustrates the average behavior of both algorithms when the network size increases according by the number of expanded vertices. This experiment indicate that the size of network does not affect the average number of expanded vertices. This occurs because we use a uniform distribution of points of interest. Thus the number of expanded vertices in a bigger network is the same of a small one since they have a similar structure. However, the CPU time increases with network size. This increasing in CPU time is explained because when the network size grows we have larger data structures to be managed in both of algorithms. The errors bars illustrate upper and lower 95% confidence limits, assuming the data to be normally distributed. That indicate our heuristic becomes more trusted when the network size grows.

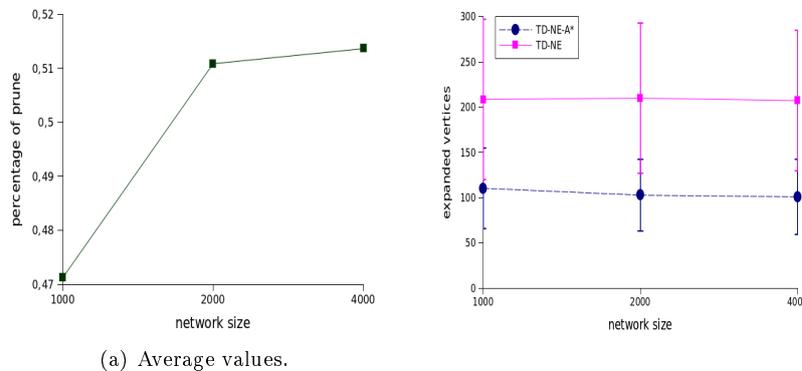


Fig. 6. Comparison when network size increases.

6. CONCLUSION AND FUTURE WORKS

In this paper we propose incorporate an A* search and prune in an incremental expansion to processing KNN queries in time-dependent road networks. The idea behind our approach is discard vertices that are nearest from the query but far from any point of interest. Our approach adds to each vertex an heuristic function that gives an expectation to find a point of interest quickly in a path that pass by this vertex. Furthermore, we use the upper bound values to prune unwanted paths. Our experiments show that TD-NE-A* can expanded 50% less vertices than TD-NE. We believe that this is a considerable gain in the I/O cost in a disk-based implementation.

As future work, we will extend our method to a disk based implementation and develop and scheme disk storage to time-dependent networks. It also would be interest for future research develop approaches to solve others popular queries that do not have solutions in time-dependent networks.

REFERENCES

- COOKE, K. AND HALSEY, E. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications* 14 (3): 493–498, 1966.
- DEMIRYUREK, U., BANAEI-KASHANI, F., AND SHAHABI, C. Towards k-nearest neighbor search in time-dependent spatial network databases. *Databases in Networked Information Systems*, 2010.
- DEMIRYUREK, U., BANAEI-KASHANI, F., AND SHAHABI, C. Efficient k-nearest neighbor search in time-dependent spatial networks. In *Database and Expert Systems Applications*. Springer, pp. 432–449, 2011.
- DIJKSTRA, E. A note on two problems in connexion with graphs. *Numerische mathematik* 1 (1): 269–271, 1959.
- ERWIG, M. AND HAGEN, F. The graph voronoi diagram with applications. *Networks* vol. 36, pp. 156–163, 2000.
- KOLAHDOUZAN, M. AND SHAHABI, C. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, pp. 840–851, 2004.
- NANNICINI, G., DELLING, D., LIBERTI, L., AND SCHULTES, D. Bidirectional A* search for time-dependent fast paths. In *Proceedings of the 7th international conference on Experimental algorithms*. WEA'08. Springer-Verlag, Berlin, Heidelberg, pp. 334–346, 2008.
- ORDA, A. AND ROM, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *J. ACM* vol. 37, pp. 607–625, July, 1990.
- PAPADIAS, D., ZHANG, J., MAMOULIS, N., AND TAO, Y. Query processing in spatial network databases. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, pp. 802–813, 2003.
- WAGNER, D. AND WILLHALM, T. Speed-up techniques for shortest-path computations. *STACS 2007*, 2007.