k-Nearest Neighbors Queries in Time-Dependent Road Networks

Lívia A. Cruz[1], Mario A. Nascimento[2] and José A. F. de Macêdo[1]

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

Abstract. In this article, we study the problem of processing k-nearest neighbors (kNN) queries in road networks considering traffic conditions, in particular the case where the speed of moving objects is time-dependent. For instance, given that the user is at a given location at certain time, the query returns the k points of interest (e.g., gas stations) that can be reached in the minimum amount of time. Previous works have proposed solutions to answer kNN queries in road networks where the moving speed in each road is constant. Obviously, these solutions cannot be simply applied to the problem we are interested in. Our approach uses the well-known A* search algorithm by applying incremental network expansion and pruning unpromising 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

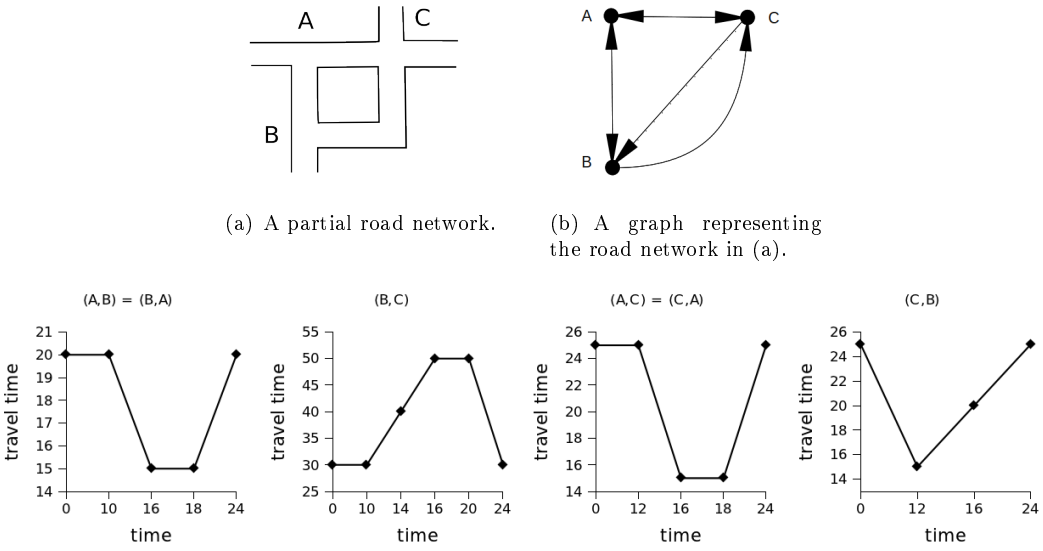
1. INTRODUCTION

The assessment and consideration of traffic conditions is key for leveraging intelligent transportation systems. With the availability of inexpensive tracking devices, such as GPS-enabled devices, it is possible to collect large amounts of trajectory data of vehicles. Using such data along with the underlying road network information allows creating an accurate picture of the traffic conditions in time and space. One possible use of this information is to compute travel time from an origin to a destination, which is a major issue for the intelligent transportation domain. The main motivation for this work is to use previously computed and time-dependent travel time in road networks in order to answer time-dependent k-nearest neighbors queries.

Travel time on road networks heavily depends on the traffic and, typically, the time a moving object takes to traverse a segment depends on departure time. 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 or an avenue) and the edges connect vertices (depending on the application, additional points can represent a change in curvature or in maximum speed of a segment). In this work, the travel time is modeled by a time-dependent graph, where the cost (time) to traverse an edge is a function of the departure time.

For example, consider Figure 1(a) where a partial network is shown. Figure 1(b) shows a graph

Research performed while the first author, supported by a CAPES scholarship, was visiting the Univ. of Alberta (under the auspices of DFAIT's Emerging Leaders of America Program, ELAP). The second author was partially supported by NSERC, Canada.



(a) A partial road network.

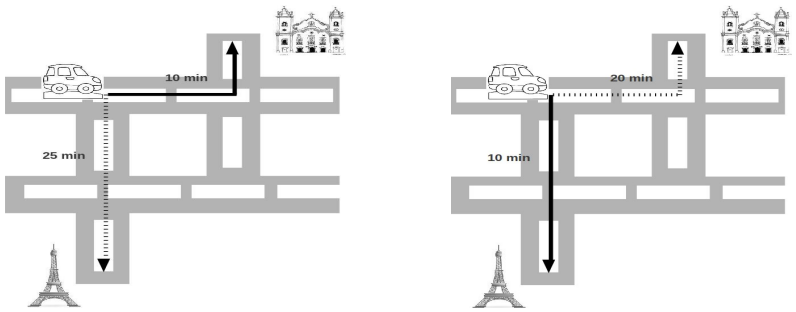
(b) A graph representing the road network in (a).

(c) Travel time functions.

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

representing the network in Figure 1(a). The travel time is given by piece-wise linear functions, shown in Figure 1(c). Each edge in the graph has its respective travel time function. There are two paths to go from vertex b to c. One can take path $\langle b, c\rangle$, that goes from b to c directly, or $\langle b, a, c\rangle$, that passes by a. The fastest path from b to c depends on the departure time t_s. For $t_s = 10h$, path $\langle b, c\rangle$ takes $30min$. Using path $\langle b, a, c\rangle$ traveling 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 path $\langle b, a, c\rangle$ takes $45min$, for departure time $10h$. Similarly, if $t_s = 16h$, path $\langle b, c\rangle$ takes $50min$ and path $\langle b, a, c\rangle$ takes $30min$. It is faster to go directly to c at $10h$, but at $16h$ the best choice is to go through a and then to c.

In this work, we consider time-dependency to answer kNN queries. A kNN query retrieve the k points of interest that are closest. In time-dependent networks, a kNN query returns the k points of interest with minimum travel-time from the query point. As a more concrete example, consider the following scenario. Imagine a tourist in Paris who is interested to visit the touristic attraction closest from him/her. Let us consider two points of interest in the city, the Eiffel Tower and the Cathedral of Notre Dame. He/she asks a query asking for the touristic attraction whose the path leading up to



(a) The nearest neighbor at $10h$.

(b) The nearest neighbor at $22h$.

Fig. 2. An example of kNN query. The fastest path from the query point to the nearest neighbor is in solid line. The fastest path from the query point to the other point of interest is in dashed line.

it is the fastest at that time, the answer depends on the departure time. For example, at $10h$ it takes 10 minutes to go to the Cathedral. It is the nearest attraction. Although, if he/she asks the same query at $22h$, in the same spatial point, the nearest attraction is the Eiffel Tower.

Previous solutions for shortest paths, k-nearest neighbors and others common queries in static networks no longer work when the edge costs (travel time) themselves depend on time. Design of efficient and correct query processing strategies and algorithms is a challenge since some of the commonly assumed graph-properties may not hold for time-dependent networks [George et al. 2007]. In this context, the main contributions of this article are:

(1) We propose an algorithm based in the well-known A* search and incremental network expansion algorithms [Papadias et al. 2003] to process kNN queries in time-dependent networks;
(2) We propose an heuristic function to be used in our A* search that works well in time-dependent networks;
(3) We propose an algorithm to include points of interest in a time-dependent network given the travel-time functions in the original network;
(4) We prove the correctness of our solution;
(5) We perform an experimental evaluation that shows our solution outperforms the approach in [Demiryurek et al. 2010b] by up to 50% in terms of I/O operations.

This article is organized as follows. In Section 2, we introduce some important definitions for understanding our proposed solution, 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 solution. The experimental evaluation and results are shown in Section 6. Finally, Section 7 concludes this article.

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 also give other basic definitions as travel-time, time-dependent fastest path and time-dependent distance, useful for stating the problem we address.

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) $E = \{(v_i, v_j) \mid v_i, v_j \in V, i \neq j\}$ is a set of edges; (iii) $C = \{c_{(v_i,v_j)}(\cdot) \mid (v_i, v_j) \in E\}$, where $c_{(v_i,v_j)} : [0, T] \rightarrow R^+$ is a function which attributes a positive weight for (v_i, v_j) depending on a time instant $t \in [0, T]$ and where T is a domain-dependent time length.

A TDG is a graph where the edge costs vary with time. For each edge (u, v), a function $c_{(u,v)}(t)$ gives the cost of traversing (u, v) at departure time t and $[0, T]$ is the domain of the functions in C. For example, $T = 24h$ means that $c_{(u,v)}(t)$ is defined for a full day.

Note that our definition allows different edge costs to bidirectional segments, in the sense that an edge (u, v) and its opposite (v, u) may be such that $c_{(u,v)}(t) \neq c_{(v,u)}(t)$. Furthermore, we assume that C is a set of piecewise-linear functions that satisfy the FIFO property. Basically, the FIFO property states that if an object A starts traversing an edge before an object B, then A has to finish traversing that edge before B. The time-dependent shortest path problem has a polynomial time 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 (POIs) or query points, by a pair $p = \langle (u, v), \tau_{(u,v)} \rangle$, such that (u, v) is an edge in G where p is on, $\tau_{(u,v)} = \frac{d(u,p)}{d(u,v)}$ is the proportional distance from u to p with respect to entire segment and $d(p_i, p_j)$ is the Euclidean distance between p_i and p_j, if p_i, p_j are

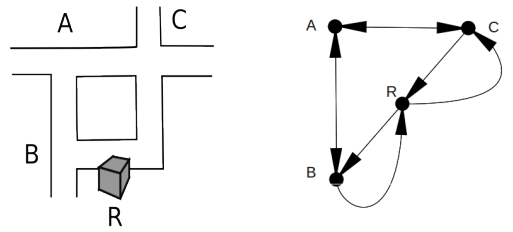
Algorithm 1: IncludePOI

Input: A TDG $G = (V, E, C)$, a POI $p = \langle (u, v), \tau_p \rangle$
Output: $G = (V^{'} = V \cup \{p\}, E^{'}, C^{'})$

$V^{'} \leftarrow V \cup \{p\}$;
$E^{'} \leftarrow E \setminus \{(u, v)\} \cup \{(u, v_p), (v_p, v)\}$;
$c_{(u,p)}(t) \leftarrow \tau_p \times c_{(u,v)}(t)$;
$c_{(p,v)}(t) \leftarrow (1 - \tau_p) \times c_{(u,v)}(t)$;
$C^{'} \leftarrow C \setminus \{c_{(u,v)}(t)\} \cup \{c_{(u,p)}(t), c_{(p,v)}(t)\}$;
if $(v, u) \in E$ **then**
    $E^{'} \leftarrow E^{'} \setminus \{(v, u)\}$;
    $c_{(v,p)}(t) \leftarrow (1 - \tau_p) \times c_{(v,u)}(t)$;
    $c_{(p,u)}(t) \leftarrow \tau_p \times c_{(v,u)}(t)$;
    $C^{'} \leftarrow C^{'} \setminus \{c_{(v,u)}(t)\} \cup \{c_{(v,p)}(t), c_{(p,u)}(t)\}$;
end
Return $G = (V^{'}, E^{'}, C^{'})$;

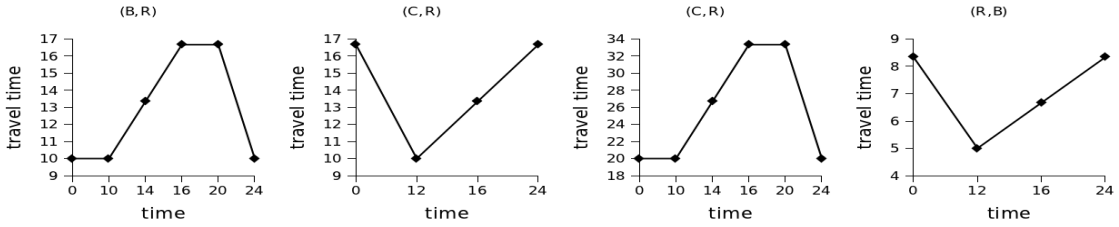
on the same edge, and undefined, otherwise. Note that to use the Euclidean distance, we suppose each curvature change in the road is mapped to a vertex in the graph. This assumption implies potential increase in the number of vertices needed to model the network. However, this is not a bottleneck in our approach, since we propose a method to efficiently retrieve the vertex information. 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. We consider that only one of them is given and the other one is directly obtained from it.

We assume a process that generates a TDG from a road network. In this process, G can be generated such that each POI from a set of POIs S is included as a vertex in G. However, a new POI may arise and we would not like have to rebuild the entire network again. The followed process show how to include a POI as a vertex in G. Given a TDG $G = (V, E, C)$ and a set S of POIs, we include each POI in G as a vertex. Algorithm 1 is a procedure that receives as input a TDG G and a



(a) Road network and a restaurant R.

(b) TDG considering the point of interest R.



(c) Travel time function to new edges.

Fig. 3. Time-dependent graph representing a network and points of interest.

POI $p = \langle (u, v), \tau_p \rangle$, generates a vertex in G equivalent to p and calculates the cost function for new edges. Figure 3 presents an example of how to include a POI as a vertex in the TDG in Figure 1(b). Figure 3(c) presents how the travel-time functions of the same TDG change according to a new vertex included. In this example, $R = \langle (B, C), \frac{1}{3} \rangle$ is a point of interest in the application domain. Note that the travel time functions of edges (A, B) and (C, B), and their opposites edges, defined in Figure 1(c) still apply.

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 create 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, we have to remove (C, B) from E, $c_{(C,B)}$ from C, the edges in the other direction, ((C, R) and (R, B)) and calculate the cost functions for the other direction. To calculate the new functions, we consider $(1 - p_R)$ 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 $c_{(R,B)} = \frac{1}{3} c_{(C,B)}$. This process assumes constant speed for objects travelling through an edge. This assumption is acceptable since we assume to only have information about the travel time to go through an entire edge.

The temporal cost to traverse a path from a specific departure time in a TDG is called travel time. The arrival time of a path is calculated assuming that stops are not permitted. They are formally defined as follows.

Definition 2.2. Given a TDG $G = (V, E, C)$, the **arrival time** to traverse an edge $(v_i, v_j) \in E$ at departure $t \in [0, T]$ is given by $AT(v_i, v_j, t) = t + c_{(v_i, v_j)}(t) \bmod T$.

Definition 2.3. Given a TDG $G = (V, E, C)$, a path $p = \langle v_{p_1}, \ldots, v_{p_i}, v_{p_{i+1}}, \ldots, v_{p_k} \rangle$ in G and a departure time $t \in [0, T]$, the **travel time** of p is the time-dependent cost to traverse 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 $t_{i+1} = AT(v_{p_i}, v_{p_{i+1}}, t_i)$.

Definition 2.4. Given a TDG $G = (V, E, C)$, $u, v \in V$ and a departure time t, let $P(v, u)$ be a set of paths from u to v in G. A **time-dependent fastest path** from u to v, denoted by $TDFP(u, v, t)$, is a path $p \in P(v, u)$ such that $TT(p, t) \leq TT(p', t)$, for all $p' \in P(v, u)$.

Definition 2.5. Given a TDG $G = (V, E, C)$, $u, v \in V$ and a departure time t, a **time-dependent distance** from u to v at departure time t is given by $TDD(u, v, t) = TT(p, t)$, where p is the TDFP from u to v.

2.1 Problem Statement

We consider the problem of processing kNN queries in road networks where the travel time is time-dependent, defined as follows:

Definition 2.6. Let $G = (V, E, C)$ be a TDG and $POI \subseteq V$ be a set of points of interest in G. Given a query point q, a departure time t and an integer $k > 0$, a Time-Dependent kNN (TD-kNN) query returns a set $R = \{v_{r_1}, \ldots v_{r_k}\} \subseteq POI$ such that $\forall v \in POI \setminus R, TDD(q, v_{r_i}, t) \leq TDD(q, v, t)$. In other words, a TD-$k$NN query returns a set of k points of interest that are closest from q than the others remaining points considering a departure time t.

Previous works have proposed solutions for answering 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. Next we discuss works that consider problems that do have similar assumptions and constraints as ours.

3. RELATED WORK

3.1 Time-Dependent Shortest Path

This problem has an intrinsically relation with the time-dependent kNN problem, since the nearest POIs from the query depend on the cost (travel time) of the shortest (fastest) path to each POI object. The usual solution to shortest path problem in static graphs is Dijkstra's algorithm [Dijkstra 1959]. [Wagner and Willhalm 2007] presents a survey with 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 has been proposed to the time-dependent case. The first algorithm that considers a time-dependent variant of shortest paths is addressed in [Cooke and Halsey 1966]. This algorithm is a modified form of Bellman's iteration scheme for finding the shortest route between any two vertices in a network. Another approach was proposed by [Nannicini et al. 2008], it applies bidirectional A* search with landmarks on a time-dependent network to calculate the fastest path between two vertices.

3.2 kNN Queries in Road Networks

The problem of kNN queries in road networks was introduced in [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 (INE) algorithms. IER uses the assumption that the Euclidean distance between two points on the network is less than the network distance. This assumption is used to retrieve kNN points, according to the Euclidean distance, and use their network distance as an upper bound for the distance of kNN points according to the network distance. INE is an adaptation of Dijkstra's algorithm. With this algorithm, starting from the query object q all network nodes reachable from q in every direction are visited in order of their proximity to q until all k nearest data objects are located. [Kolahdouzan and Shahabi 2004] presented an approach based on pre-computing the network's 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. These approaches cannot be directly applied to solve TD-kNN queries.

3.3 Time-dependent kNN Queries

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

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

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 networks in [Papadias et al. 2003]. Starting from the query object q, INE visits all network vertices reachable from q in order of their proximity until all k nearest neighbors objects are located.

We incorporate an A* search directly into INE's expansion (hence the name TD-NE-A*). To solve the shortest path problem, the A* search works similar to Dijkstra's algorithm [Goldberg and Harrelson 2005]. It uses a distance $d(v_i, v_k)$ plus a heuristic function $H(v_k)$ to determine the order in which 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 the goal vertex that pass by v_k. The heuristic function $H(v_k)$ must be an admissible heuristic; that is, it must not overestimate the distance from v_k to the goal. If $H(v_k)$ satisfies an additional condition $H(v_k) \leq d(v_k, v_j) + H(v_j)$ for every edge (v_k, v_j) of the graph, then $H(v_k)$ is called monotone, or consistent. In such a case, A* can be implemented more efficiently because it approaches the solution in an incremental way without taking any step back.

Instead of using 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 an A* search directly in an incremental network expansion. Thus our search has multiple and unknown targets. An incremental strategy avoids re-computing costs previously calculated and visits only necessary edges. In our algorithm, the heuristic function adds to each vertex an estimate of the potential for it to take part of the fastest path that leads to a nearest point of interest. The idea behind our search is avoid continuing to expand nodes in a path that is fast but is far from any point of interest in the network. We are motivated by the fact that a vertex u may be the closest node to q, but it does not imply that we would find a next nearest POI when expanding u. To explain how our method works, we need to introduce some definitions.

Definition 4.1. 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 the set of costs $\underline{c}_{(v_i,v_j)} = min_{t\in[0,T]}\{c_{(v_i,v_j)}(t)\}$, 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 the set of edge costs $\overline{c}_{(v_i,v_j)} = max_{t\in[0,T]}\{c_{(v_i,v_j)}(t)\}$, 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 on a departure time. We set our heuristic function $H(u)$ to be equal to the travel time from u to its nearest neighbor in $\underline{G}$. This is an optimistic value, since is the best travel time of a path that leads u to some point of interest. Furthermore, this is an admissible and consistent heuristic, as shown by the two followed lemmas, thus it is feasible to be used in the A* search.

LEMMA 4.2. *$H(\cdot)$ is an admissible heuristic.*

PROOF. To show that $H(\cdot)$ 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 already visited in the search and nn_u the nearest neighbor from u in $\underline{G}$. Assume that $H(u)$ is not admissible, i.e., 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 $H(u)$ does not overestimate the distance to next nearest neighbor object. Thus, it is admissible. □

LEMMA 4.3. *$H(\cdot)$ 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 to 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. Hence, it is admissible. □

4.1 Offline Preprocessing

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 vary with time is used. The first one calculates the heuristic function value for vertices in G. For each vertex v, the distance between v and its nearest neighbor in $\underline{G}$ is calculated. This distance is attributed to $H(v)$ to be used as the heuristic function. The second step computes the nearest neighbor 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 farther than a set of candidates.

Our preprocessing steps have the important advantage of using familiar solutions that run in polynomial time on non time-dependent road networks (i.e., static networks). To execute them, we use the algorithm proposed in [Papadias et al. 2003]. Each preprocessing step calls a 1NN search to find the nearest neighbor for each vertex. In the worst case, this algorithm runs in time $O(|V||E|+|V|^2 \log |V|)$. However, a kNN search started in a vertex find the 1NN of all vertices in the path from the source to its nearest neighbor. Thus, we do not need to start one search for each vertex in the network, improving the time of execution.

In some case of updates, the precomputed information has to be changed. We consider three cases of updates: updates in the travel time functions; deletion of a POI object; inclusion of a POI object.

Update in the travel-time functions. When the value of the travel-time function change, a preprocessing step in the $\underline{G}$ or $\overline{G}$ has to be run again only if the lower bound or the upper bound, respectively, changes.

Deletion of a POI. When a POI is deleted from the network we have to run the preprocessing step for all vertices that have the POI object removed as a nearest neighbor in $\underline{G}$ or $\overline{G}$.

Inclusion of a POI. When a POI is included we have to run the preprocessing step again.

4.2 Query Processing

Algorithm 2 presents the pseudo-code of TD-NE-A* and works as follows. It takes three parameters as input, the query point $q = \langle (u, v), \tau_q \rangle$, the number of nearest neighbors k and the departure time t. It works similarly to previous network expansion algorithms, but includes two strategies to guide the search and pruning of unpromising vertices. First, it gets the edge (u, v) that represents the road segment covering q. The vertices u and v are inserted in a priority queue Q that stores the set of candidates for expansion in the next step. An entry in Q queue is a tuple $(v_i, AT_{v_i}, TT_{v_i}, L_{v_i})$, where $TT_{v_i} = TT(q, v_i, t)$, $AT_{v_i} = AT(q, v_i, t)$ and L_{v_i} is given by $TT_{v_i} + H(v_i)$, i.e. a sum of the travel time from q to v_i in the path in which v_i was found plus the minimum travel time to find a nearest neighbor from v_i. The L_{v_i} value represents an optimistic expectation for the travel time of a path that leads v_i to its nearest neighbor. The priority of elements in Q is given by the increasing order of L_{v_i} values. The motivation behind this is to check first the vertices that offer a greater chance to find a point of interest quickly.

Another priority queue Q_U is maintained to store upper bound values. For each expanded vertex, the value of $UTDD(v, UNN(v))$ is a pessimistic expectation for the travel time of a path that leads v to its nearest neighbor, that represents an upper bound. For each distinct $UNN(v)$ we maintain the

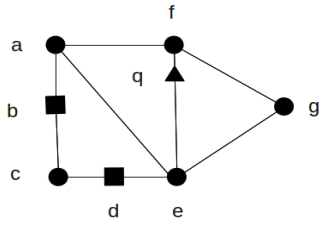
Algorithm 2: TD-NE-A*

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

$TT_v \leftarrow \tau_q \times c_{(u,v)}(t)$;
$AT_v \leftarrow (t + TT_v) \bmod T$;
$L_v \leftarrow T_v + H(v)$;
En-queue (v, AT_v, TT_v, L_v) in Q;
if $(v, u) \in E$ **then**
    $TT_u \leftarrow (1 - \tau_q) \times c_{(v,u)}(t)$;
    $AT_u \leftarrow (t + TT_u) \bmod T$;
    $L_u \leftarrow T_u + H(u)$;
    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**
        $S_{NN} \leftarrow S_{NN} \cup \{u\}$;
    end
    for $v \in adjacency(u)$ **do**
        $TT_v \leftarrow TT_u + c_{(u,v)}(AT_u))$;
        $AT_v \leftarrow (t + TT_v) \bmod T$;
        $L_v \leftarrow TT_v + H(v)$;
        if $L_v \leq Q_U(k)$ **then**
            if v *is not in* Q **then**
                En-queue (v, AT_v, TT_v, L_v) in Q;
                Mark v as en-queued;
            else
                Update L_v, if it is necessary;
                Re-order Q;
            end
            $U_{UNN(v)} \leftarrow TT_v + UTDD(v, UNN(v))$;
            if $UNN(v)$ *is not in* Q_U **then**
                En-queue $(UNN(v), U_{UNN(v)})$ in Q_U;
            else
                Update $U_{UNN(v)}$, if it is necessary;
                Re-order Q_U;
            end
        end
    end
end
Return S_{NN};

lowest upper bound found in the search. Q_U is ordered by increasing order of $UTDD(v, UNN(v))$ values and is used to pruning process. More specifically, if L_v is greater than the kth upper bound in Q_U, that means that we already have k candidates that have, in the worst case, a travel time less than the best possible travel time in a path that passes by v. Thus v can be discarded. When a vertex v is expanded, we check if $UNN(v)$ is in Q_U. If it is not, we include $UNN(v)$ and its upper bound $U_{UNN(v)}$ in Q_U. If $UNN(v)$ was already included in Q_U, we check if $UTDD(v, UNN(v))$ is smaller than the current upper bound to it. In that case, we update the position and upper bound value of $UNN(v)$ in Q_U.

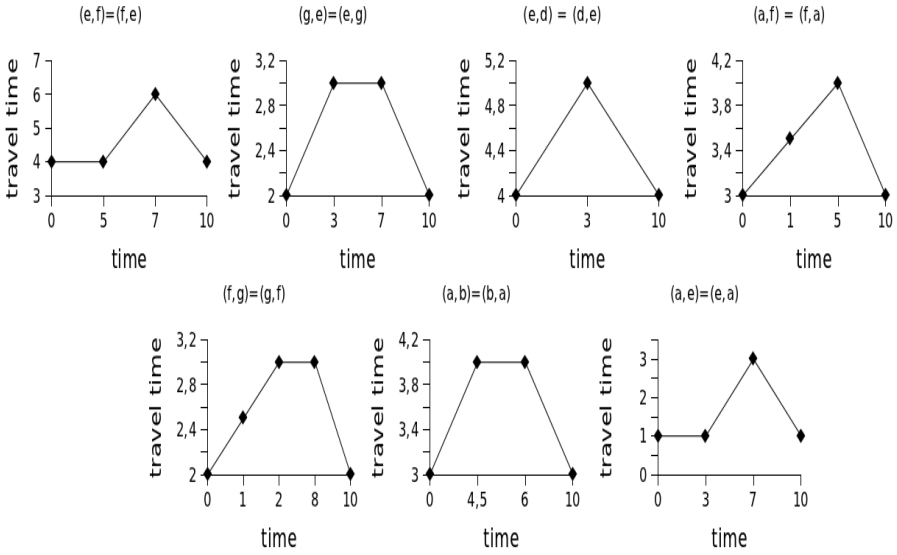
The time required for our query processing is mainly given by the I/O operations, executed to



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

v	H(v)	UNN(v)	UTDD(v, UNN(v))
a	3	b	4
b	0	b	0
c	4	d	4
d	0	d	0
e	4	d	5
f	6	b	8
g	6	d	8

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



(c) Time dependent edges cost.

Fig. 4. A road network and its respective graphs considering points of interest.

retrieve the information about the adjacency lists of the vertices. The cost to maintain the queues is given by the operations for removing, inserting and reordering entries in the queue. of entries in the queue. Each insertion or removal on Q requires a time of $O(\log |V|)$, in the worst case. Furthermore, each update requires that the priority queue be reordering, that spends a $O(|V|)$ time, in the worst case. In the same way, an insertion operation in Q_U requires $O(\log |POI|)$ and a reordering operation requires $O(|POI|)$ time. Since the operations executed to maintain the queues are done in main memory, and in polynomial time, they are not a bottleneck in our approach.

4.3 Running Example

As an example, consider the graph in Figure 4, the travel time of its edges are shown in Figure 4(c). Note that the travel times for edges (b, c) and (c, d) are not shown only because they are not used in this example.

4.3.1 *Offline Preprocessing.* The result of our preprocessing step is shown in Figure 4(b). For each vertex, a non-time-dependent (i.e. static) NN search is executed in graphs $\underline{G}$ and $\overline{G}$. For example, consider vertex f in figure 4(a). The distance between f and b in $\underline{G}$ 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 $UTDD(f, UNN(f)) = 8$.

4.3.2 *Query Processing.* The input 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 to e, 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 $TT(q,e,t) + UTDD(e,d) = 8$, d is the nearest neighbor from e in $\overline{G}$, $TT(q,f,t) + UTDD(f,b) = 9$ and b is the nearest neighbor from f in $\overline{G}$. Now, each iteration removes a vertex from Q to be expanded.

The first vertex expanded is f, since 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 $TT_a + UTDD(a,b) = 8.5$, an upper bound smaller 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. Furthermore, 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 $L_b = 8.5 > Q_U(k)$, meaning b can be found in a fastest path. After this, we have $Q = \langle (d, AT_d = 8, TT_d = 8, L_d = 8) \rangle$ and d is expanded and included into S_{NN}. As $k = 1$ the algorithm finishes.

Next, we the correctness of our solution.

THEOREM 4.4. *Let $S_{NN} = \{v_{NN_1}, \ldots, v_{NN_k}\}$ be 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 prove that, it is enough to show that when a point of interest r is removed from Q: (1) its label T_r has the same value of travel time of the time-dependent fastest path from q to it and (2) there is no point of interest that has a path from q faster than $TDD(q,r,t)$.

(1) We will prove it by induction in the number of dequeued vertices. The base case 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 v be the l-th dequeued vertex. Suppose that $T_v < TDD(q,v,t)$. Let z be the last vertex dequeued such that is in the fastest path from q to v. Let w be the next vertex in the fastest path from q to v. 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. This is 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) Let us, suppose by contradiction that a point of interest r was removed and there is another point of interested 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 to (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 $L_r \leq TDD(q,r^*,t)$, violating the hypothesis that the path between q and r^* is faster than the path between q and r. Thus $TDD(q,r,t) = TDD(q,r^*,t)$ and (2) holds.

□

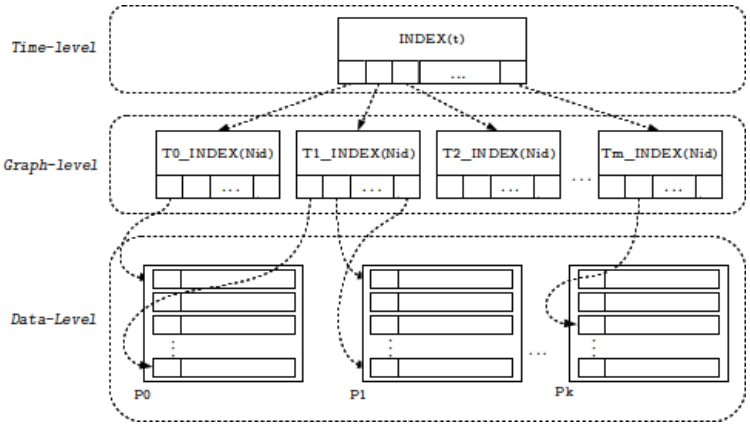


Fig. 5. An access method for time-dependent road networks.

5. ACCESS METHOD

We also propose an access method that facilitates the processing of TD-NE-A* in secondary storage, for cases of larger networks. Before explaining it, we would like to draw attention to some points that do not allow us to store a time-dependent network in the same way as static networks are stored. The first one is that as the edge costs are time-dependent, we need much more space to storage them. Actually, the storage space required to store the edge costs increases as the time granularity increases. Therefore, we can not store every edge costs of the same adjacency list together without having to partition the adjacency list. Another fact is that, in our algorithm, since a vertex is accessed only for a given departure time, it will not accessed again. Thus, if we store all costs to the same edge together, we could retrieve lots of unnecessary information when accessing the cost of an edge at a specific departure time. Finally, after we check a specific departure time, we most likely need to check a departure times close to it. Thus, we would like to maintain the information of departure times that are closer to each other at the same disk page, if possible.

Based on these observations, we propose an access method that is scalable according to both time granularity and the number of edges of the network. It is composed by three levels, the Time-Level, the Graph-Level and the Data-Level. In the first one, we use an index to access only the necessary information of a specific time and avoids retrieving the edge costs for every possible departure time. The data pages in the time-level contain pointers to index structures in the graph-level. The graph-level has an index structure for each time partition. Each index structure in the graph-level allows accessing the adjacency list, at a specific departure time, according to the vertice identifiers ($Nids$). The data pages of the structures in the graph-level contain pointers to a disk page in the data-level that stores the adjacency list of a vertex. The adjacency lists are clustered by time, such that one page has adjacency lists with costs of departure times that are closer.

Note that in our access method, the index structures are generic. We can use, for example, a B^+ tree in both levels. In that case, the number of index pages accessed for each data page retrieved is $O(\log_F |T| + \log_F |V|)$, where F is the fan out of the trees, T is the number of linear functions to compose an edge cost and $|V|$ is the cardinality of vertices in the network. Note that, when the time granularity is more coarse, the time-level index is small enough to stay in main memory. In that case, there is no I/O to access the time-level.

6. EXPERIMENTAL EVALUATION

We compare our approach (TD-NE-A*) to the approach proposed by [Demiryurek et al. 2010b] (TD-NE). We conducted our experiments on linux-based PC with 3.00GHz CPU and 4 GB main memory

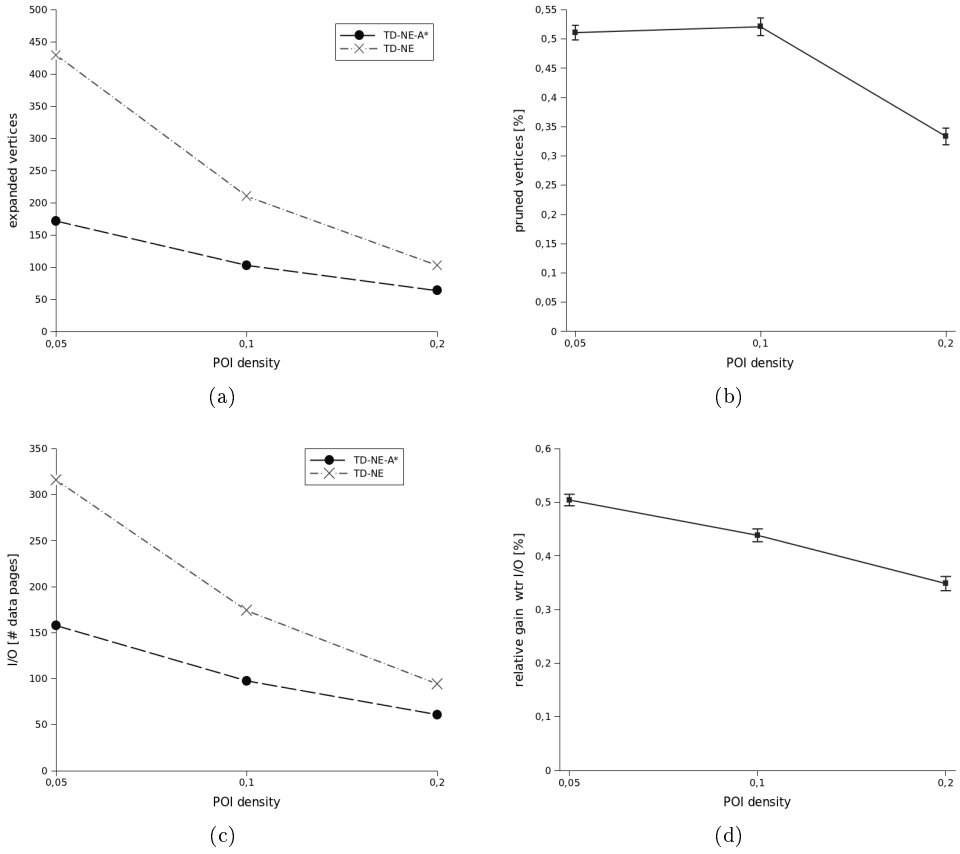


Fig. 6. Performance when density of POIs increases.

and used C++ to implementing all approaches. We generated synthetic time-dependent road networks with temporal resolution of 96 points in time, e.g., a point at every 15 minutes of a day. Each vertex has an uniformly distributed degree with an average value of 4. We evaluated how our approach works according to the network size (i.e., number of vertices), query size k and the density of POIs (i.e., the ratio of POIs cardinality to vertex cardinality). These parameters are shown in Table I. For each experiment we vary a parameter and fix the others parameters to default values (in bold). In all of these experiments, for each configuration of these parameters we generated 10 distinct time-dependent road networks and executed 10 queries randomly selected for each network, for a total of 100 queries. We calculated upper and lower 95% confidence limits for the relative gain, assuming the data to be normally distributed.

We compare the algorithms according to the number of expanded vertices and the number of I/O operations to access the disk pages in data-level, both with respect to the density of POIs, query size and network size. We simulated the I/O operations in data pages (data-level), using LRU as cache politics and 5% of the number of disk page in data-level as the cache size. We can verify, as expected, that there is a correlation between the number of expanded vertices and the number of accessed disk pages.

Table I. Parameters values of experiments.

Density of POIs	5% ,**10%**, 20%
Query size	1 , 10, **20**, 30
Network size	1000, **2000**, 4000, 10000

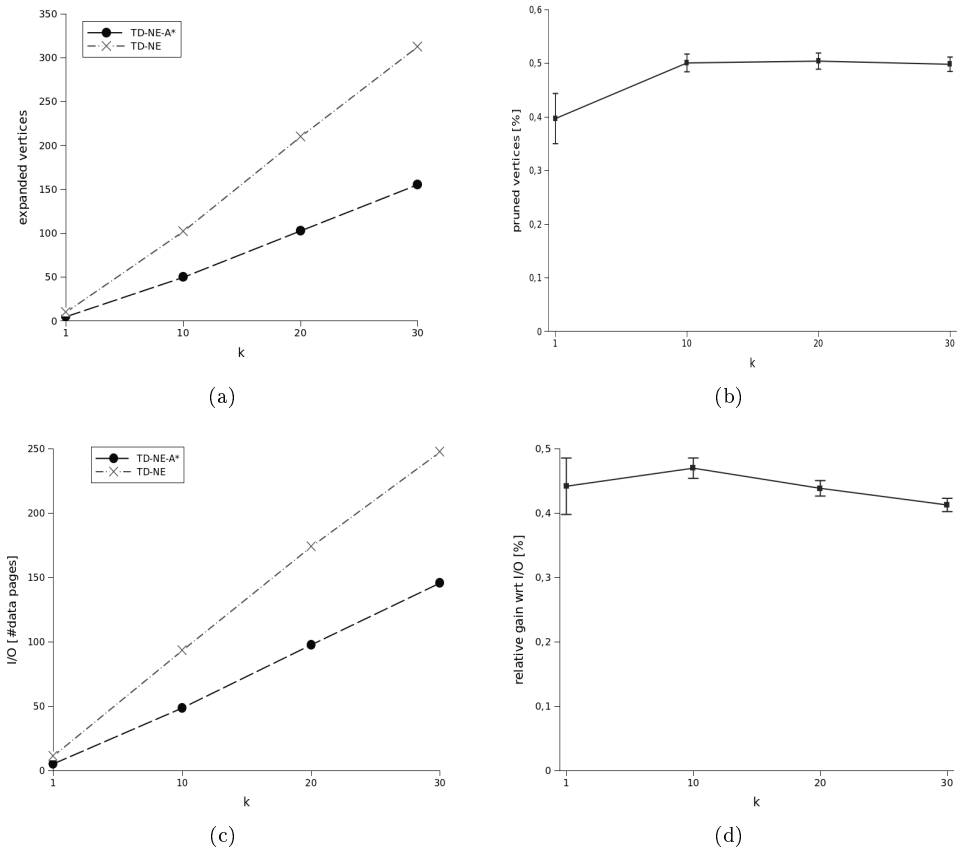


Fig. 7. Performance when query size increases.

Effect of POIs density. We set the density of POIs to be 5%, 10% and 20% of the number of points of interest (uniformly distributed). For each density, we generated 10 distinct time-dependent networks with 2000 vertices and executed 10 randomly selected queries with k=20 on each one. Figure 6(a) illustrates the difference between the average of expanded vertices in both algorithms and Figure 6(c) presents the difference according to I/O operations. We can observe that in both algorithms it is necessary to expand less vertices and access less data pages when the network becomes denser. This is reasonable since the number of POIs grows for each evaluated subnetwork; it takes less effort to find the answer. We can also observe that the gap between the algorithms increases when the network is sparse. In these experiments TD-NE-A* pruned on average from 33,33% to 51% of the nodes expanded by TD-NE. Figures 6(b) 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 this is that the number of candidate POIs increases with the density. Thus the quality of the heuristic function decreases and the number of false hits increases. Figure 6(d) show that the relative gain of our approach with respect to I/O operations was more than 34% on average, for all values of POIs density. This gain can be up to 50%, on average, when the network is sparse. The error bars shows confidence intervals for our estimates.

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. For each one of them, we executed 10 queries randomly selected with k = {1, 10, 20, 30}. Figure 7(a) 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 points of interest. In the same way, as shown in Figure 7(c),

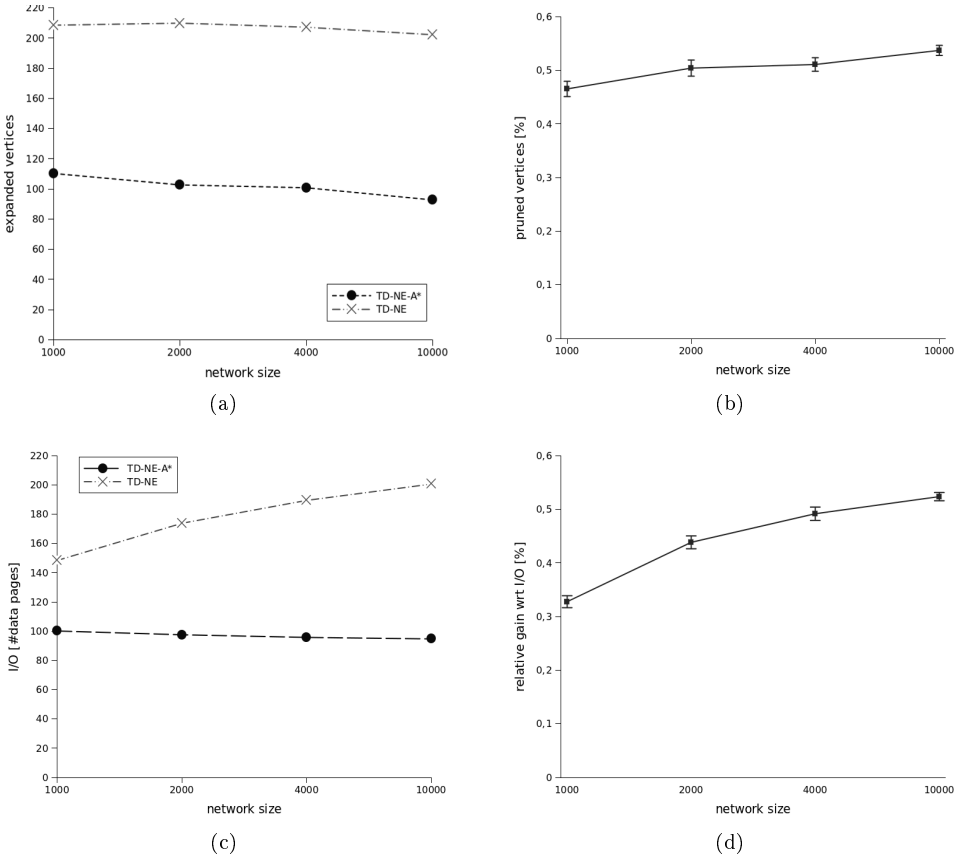


Fig. 8. Performance when network size increases.

the number of data pages accessed also increases when k increases. Furthermore, when k grows the gap between the performance of TD-NE-A* and TD-NE also increases.

Figure 7(b) 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 by more than 40%, on average. Figure 7(c) shows that this gain is reflected when we consider the I/O cost. Observe that, although the difference between the algorithms seems to be small when $k = 1$, the reason for this is that the number of expanded vertices and data pages accessed themselves are small and the absolute value of this difference is not relevant. However, the relative gain when $k = 1$ is more than 40%, as shown in Figures 7(b) and 7(c). The errors bars 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, 4000 and 10000 vertices. Each one with 10% of points of interest. We executed 10 randomly selected queries with k=20 on each network. Figure 8(a) illustrates the average behavior of both algorithms when the network size increases. This experiment indicates that the size of network does not affect significantly the average number of expanded vertices, although it affects the number of I/O operations. As we use an uniform distribution of points of interest and the networks have a similar structure, the number of POIs in a small network is, approximately, the same of a subnetwork with the same size. Thus the small network can be seen as subnetwork and the number of expanded vertices in a bigger network is, on average, the same in a small network. However, when the network size increases, the vertices expanded are more scattered throughout the disk pages that potentially implies in an increasing in the number of I/O operations. Figure 8(c) shows the number of I/O operations remains constant for TD-NE-A* and increases linearly for TD-NE.

Figure 8(b) shows that the pruning power of TD-NE-A* grows with the network size. Figure 8(d) shows the gain with respect to I/O operations grows with the network size and it varies from 32% to 52%. The error bars shows confidence intervals for our estimates.

7. CONCLUSION AND FUTURE WORK

In this paper we proposed to incorporate an A* search and pruning in an incremental expansion to processing kNN queries in time-dependent road networks. To do that, we proposed an heuristic function that works well in time-dependent networks. The idea behind our approach is to discard vertices that are near from the query but far from any point of interest. Our approach adds to each vertex an expectation to find a point of interest quickly in a path that pass by this vertex. Furthermore, we use upper bound values of travel time to prune unpromising paths. Our experiments show that TD-NE-A* require up 50% less I/O operations than TD-NE. We also propose a generic and scalable access method to support our approach on a disk-based environment.

As future work, our preprocessing step as well as handling network updates efficiently will be done. It also would be interesting for future research to develop similar approaches to solve other popular queries that do not have solutions in time-dependent networks.

REFERENCES

Cooke, K. L. 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., Kashani, F. B., and Shahabi, C. Efficient k-nearest neighbor search in time-dependent spatial networks. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part I*. Berlin, Heidelberg, pp. 432–449, 2010a.

Demiryurek, U., Kashani, F. B., and Shahabi, C. Towards k-nearest neighbor search in time-dependent spatial network databases. In *Databases in Networked Information Systems*. Berlin, Heidelberg, pp. 296–310, 2010b.

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* 36 (3): 156–163, 2000.

George, B., Kim, S., and Shekhar, S. Spatio-temporal network databases and routing algorithms: a summary of results. In *Proceedings of the 10th International Conference on Advances in Spatial and Temporal Databases*. Berlin, Heidelberg, pp. 460–477, 2007.

Goldberg, A. V. and Harrelson, C. Computing the shortest path: A* search meets graph theory. In *Proceedings of the Sixteenth Snnual ACM-SIAM Symposium on Discrete algorithms*. Philadelphia, PA, USA, pp. 156–165, 2005.

Kolahdouzan, M. and Shahabi, C. Voronoi-based k nearest neighbor search for spatial network databases. In *Proceedings of the International Conference on Very Large Data Bases*. Toronto, Canada, pp. 840–851, 2004.

Nannicini, G., Delling, D., LeoLiberti, and Schultes, D. Bidirectional A* search for time-dependent fast paths. In *Proceedings of the 7th International Conference on Experimental Algorithms*. Berlin, Heidelberg, pp. 334–346, 2008.

Orda, A. and Rom, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM* 37 (3): 607–625, 1990.

Papadias, D., Zhang, J., Mamoulis, N., and Tao, Y. Query processing in spatial network databases. In *Proceedings of the International Conference on Very Large Data Bases*. Berlin, Germany, pp. 802–813, 2003.

Wagner, D. and Willhalm, T. Speed-up techniques for shortest-path computations. In *Proceedings of the 24th Annual Conference on Theoretical Aspects of Computer Science*. Berlin, Heidelberg, pp. 23–36, 2007.