UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CAMILA FERREIRA COSTA

NEAREST NEIGHBORS WITH OPERATING TIME CONSTRAINTS AND OPTIMAL SEQUENCED ROUTE QUERIES IN TIME-DEPENDENT ROAD NETWORKS

FORTALEZA, CEARÁ

2014

CAMILA FERREIRA COSTA

NEAREST NEIGHBORS WITH OPERATING TIME CONSTRAINTS AND OPTIMAL SEQUENCED ROUTE QUERIES IN TIME-DEPENDENT ROAD NETWORKS

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Área de concentração: Banco de Dados

Orientador: Prof. Dr. Javam de Castro Machado

Coorientadores: Prof. Dr. José Antônio Fernandes de Macêdo
Prof. Dr. Mario A. Nascimento

FORTALEZA, CEARÁ

2014

A000z COSTA, C. F..

Nearest Neighbors with Operating Time Constraints and Optimal Sequenced Route Queries in Time-Dependent Road Networks / Camila Ferreira Costa. 2014.

75p.;il. color. enc.

Orientador: Prof. Dr. Javam de Castro Machado

Coorientadores: Prof. Dr. José Antônio Fernandes de Macêdoe Prof. Dr. Mario A. Nascimento

Dissertação (Ciência da Computação) - Universidade Federal do Ceará, Departamento de Computação, Fortaleza, 2014.

1. Processamento de consultas espaciais 2. Redes dependentes do tempo 3. Rotas ótimas 4. Tempo para serviço I. Prof. Dr. Javam de Castro Machado(Orient.) II. Universidade Federal do Ceará– Ciência da Computação(Mestrado) III. Mestre

CDD:000.0

CAMILA FERREIRA COSTA

NEAREST NEIGHBORS WITH OPERATING TIME CONSTRAINTS AND OPTIMAL SEQUENCED ROUTE QUERIES IN TIME-DEPENDENT ROAD NETWORKS

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação. Área de concentração: Banco de Dados

Aprovada em: __/__/____

BANCA EXAMINADORA

Prof. Dr. Javam de Castro Machado
Universidade Federal do Ceará - UFC
Orientador

Prof. Dr. José Antônio Fernandes de Macêdo
Universidade Federal do Ceará - UFC
Coorientador

Prof. Dr. Mario A. Nascimento
University of Alberta - UofA
Coorientador

Prof. Dr. Ângelo Roncalli Alencar Brayner
Universidade de Fortaleza - UNIFOR

À minha avó Maria.

AGRADECIMENTOS

Agradeço aos meus pais, Neila e Roberto, pela educação que me foi dada que me permitiu chegar até aqui.

À minha querida tia Neiva por sempre apoiar minhas decisões e torcer pelo meu sucesso.

Ao meu namorado Antônio pela paciência incondicional e por todo o apoio dado durante meu mestrado.

Ao Professor José Antônio pelo suporte na pesquisa e pelas oportunidades que me foram dadas.

Ao Professor Mario Nascimento pelo acolhimento na Universidade de Alberta, por me guiar durante a pesquisa e contribuir com seus ensinamentos e conhecimentos, que foram de fundamental importância para o meu crescimento como aluna e pessoa.

Ao Professor Javam Machado pelas contribuições para o bom resultado deste trabalho.

Ao Professor Ângelo Brayner pela disponibilidade em participar da banca avaliadora e contribuir com suas observações para a melhoria deste trabalho.

Por fim, agradeço aos amigos do ARIDa por todos os momentos compartilhados durante esta etapa.

“Dizem que a vida é para quem sabe viver, mas ninguém nasce pronto. A vida é para quem é corajoso o suficiente para se arriscar e humilde o bastante para aprender.”

(Clarice Lispector)

RESUMO

Nesta dissertação nós estudamos os problemas de processar uma variação de consulta de vizinhos mais próximos e de planejamento de rotas em redes viárias dependentes do tempo. Diferentemente de redes convencionais, onde o custo de deslocamento de um ponto a outro é geralmente dado pela distância física entre esses dois pontos, uma rede dependente do tempo representa de forma mais realista o custo de realizar esse deslocamento, considerando o histórico das condições de tráfego. Mais especificamente, o tempo que um objeto móvel leva para percorrer uma via em tal rede depende do tempo de partida. Por exemplo, o tempo para se deslocar de um ponto a outro em grandes centros durante os horários de pico, quando o tráfego é intenso e as ruas estão congestionadas, é muito maior do que em horários normais.

Dentro do contexto apresentado, primeiramente nós estudamos o problema de encontrar k pontos de interesse, como por exemplo, museus ou restaurantes, nos quais um usuário pode começar a ser servido o mais rápido possível. Em outras palavras, nós buscamos minimizar a soma do tempo de viagem até um ponto de interesse mais o tempo de espera até que ele abra, caso esteja fechado. Trabalhos anteriores tratam do problema de encontrar os k vizinhos mais próximos em redes dependentes do tempo, porém, eles não levam em consideração o horário de funcionamento dos pontos de interesse. Desta forma, a consulta abordada nesses trabalhos pode retornar pontos de interesse que estão mais próximos do usuário, considerando um dado tempo de partida, mas que podem demorar para abrir, fazendo com que o usuário espere por muito tempo.

Nós propomos e discutimos três soluções para essa consulta que são baseadas em um algoritmo de expansão incremental da rede previamente proposto na literatura e usam o algoritmo de busca A* equipado com funções heurísticas adequadas para cada solução. Com o uso do algoritmo A*, nós visamos reduzir o percentual da rede avaliado na busca, evitando expandir vértices que oferecem uma baixa probabilidade de alcançar nosso objetivo. Também apresentamos resultados experimentais que comparam o número de acessos ao disco exigido em cada solução em relação a alguns parâmetros diferentes e que indicam em que casos deve-se optar por cada solução.

Na segunda consulta, nós visamos encontrar a rota ótima que conecta uma dada origem a um dado destino e que passa por uma série de pontos de interesse pertencentes a categorias determinadas pelo usuário em uma certa ordem também especificada pelo usuário. Esse tipo de consulta é conhecida como OSR, do inglês, Optimal Sequenced Route, na literatura. Como exemplo, considere que alguém está indo do trabalho para casa e no seu caminho deseja passar em um banco para sacar dinheiro e depois ir a um restaurante para jantar. Embora existam vários bancos e restaurantes em uma cidade, uma consulta OSR deve procurar pelo banco e pelo restaurante que minimizam o custo da viagem do trabalho para casa. Trabalhos anteriores propuseram soluções para consultas OSR em redes com arestas de custo fixo, mas nenhum deles considerou que esse custo pode variar de acordo com o tempo de partida.

Nós propomos uma solução ótima para esse problema que, assim como as abordagens propostas para o problema anterior, expande a rede incrementalmente e usa o algoritmo A* para guiar essa expansão. Além disso, como uma consulta OSR em redes viárias tende a re-expandir um número muito grande de vértices, nós incorporamos à essa solução um esquema para reduzir o número de re-expansões. Nós também apresentamos resultados experimentais que mostram a eficiência dessa solução em comparação com uma solução de base que

foi obtida a partir da estensão de um algoritmo anteriormente proposto na literatura. Todos os experimentos foram realizados em redes sintéticas.

Palavras-chave: Processamento de consultas espaciais . Redes dependentes do tempo. Rotas ótimas. Tempo para serviço.

ABSTRACT

In this thesis we study the problems of processing a variation of nearest neighbors and of routing planning queries in time-dependent road networks, i.e., one where travel time along each edge is a function of the departure time.

We first study the problem of finding the k points of interest (POIs), for example, museums or restaurants, in which a user can start to be served in the minimum amount of time, accounting for both the travel time to the POI and the waiting time there, if it is closed. Previous works have proposed solutions to answer k-nearest neighbor queries considering the time dependency of the network but not the operating times of the points of interest. We propose and discuss three solutions to this type of query which are based on the previously proposed incremental network expansion and use the A* search algorithm equipped with suitable heuristic functions. We also present experimental results comparing the number of disk access required in each solution with respect to a few different parameters.

In the second query, we aim at finding the optimal route that connects a origin to a destination and passes through a number of POIs in a specific sequence imposed on the categories of the POIs. Previous works have addressed this problem, but they do not consider the time dependency of the network. We propose an optimal sequenced route query algorithm which performs an incremental network expansion adopting an A* search. Furthermore, as an OSR query on road network tends to re-expand an extremely large number of nodes, we propose a scheme to reduce the re-expansions. For comparison purposes, we also present a baseline solution which was obtained by extending the previously proposed progressive neighbor exploration algorithm to cope with the time-dependent problem. We performed experiments in synthetic networks comparing the proposed solutions according to the number of expanded vertices in the search and the processing time of the queries.

Keywords: Optimal Sequenced Route. Processing spatial queries. Time-Dependent Networks. Time to Service.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

1.1 Motivation

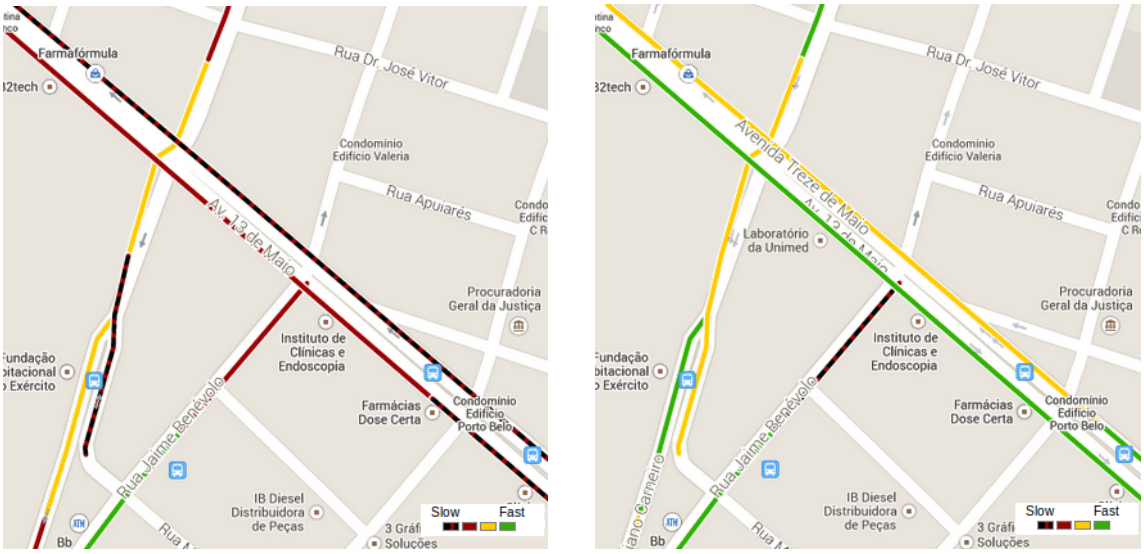
Our society faces many challenges in solving problems related with people mobility in big cities. Mobility is becoming a big issue because big cities' road network does not grow in the same rate as the number of vehicles growth. The high amount of vehicles creates congestions in the roads, which makes travel time forecasting extremely difficult. Clearly, travel time may radically change from rushing hours to normal hours. A possible solution to this problem is to use historical traffic data in order to compute more realistic travel time forecasting. With the high availability of GPS tracking devices, besides the traffic sensors positioned in road segments in several countries, it is possible to collect large amounts of trajectory data of vehicles. Thus it becomes feasible to model the dependence of traveling speed on the time of the day based on historical traffic data and to use this model to solve complex spatio-temporal queries.

Several variations of spatial queries have been investigated by the database community, such as nearest neighbors (NN) as well as range queries and their variants (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008; LI et al., 2005; CHEN et al., 2011; PAPADIAS et al., 2003; JENSEN et al., 2003; KOLAHDOUZAN; SHAHABI, 2004, 2005) and more complex and advanced queries as route planning (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008; LI et al., 2005; CHEN et al., 2011).

However, the majority of the solutions proposed to these queries fails to represent the reality in the sense that they do not consider the temporal dependence of road networks. They assume that these networks are static and the cost to traverse each edge is given by the length of this edge and, thus, it does not vary with time. This assumption is certainly not true on a real scenario, that is better represented by a time-dependent network, which takes into consideration that the time one takes to traverse a road segment, typically depends on the departure time. Figure 1.1 shows an real example of how traffic is influenced by the time of day and, consequently, the time spent to move from one point to another within a large city. It illustrates two different moments of a same avenue in Fortaleza, Brazil. At 19:30 this avenue is completely congested and, thus, the time to cross it is longer than at 20:40, when the traffic is less intense.

Some recent studies have included the temporal dependence to solve conventional spatial queries, such as k-nearest neighbors (DEMIRYUREK; BANAEI-KASHANI; SHAHABI, 2010b, 2010a; CRUZ; NASCIMENTO; MACÊDO, 2012) and shortest path (NANNICINI et al., 2012) queries. As in those works, we also consider this dependence to answer queries. More specifically, we assume that the network is modeled as a graph where the travel time along each edge is a function of the departure time. These functions give the travel time of an edge according to the time instant when one starts traversing this edge.

Processing queries in time-dependent road networks is challenging due to a number of reasons. The space required to store these networks is significantly larger than the space used to store the time-independent ones, since it is necessary to keep, for each edge, the cost to



(a) Traffic at 19:10.

(b) Traffic at 20:40.

Figura 1.1: Traffic on the avenue 13 de Maio in Fortaleza, Brazil at two different times of a day. Source: Google Maps (https://maps.google.com/).

traverse it for every time interval of a day. Furthermore, solutions for conventional queries in static networks can not be directly applied to solve the time-dependent problems. Particularly, it is complicated to apply the well-known speed-up technique of bi-directional search, that starts a search simultaneously from the source and the target, to solve the shortest path problem in a time-dependent network since the arrival time would have to be known in advance for such a procedure.

In this context, we first propose a variation of the k-nearest neighbor query, named TD-kNN-OTC, in which the operating time of facilities is taken into account. The problem of finding the k nearest points of interest (POIs), for example, museums or restaurants, in time-dependent road networks has been studied in previous works. However, they have focused only on searching for the POIs that can be reached the quickest, without taking into consideration if it will take a long time for these POIs to open from the moment they are reached. Differently from those works, we aim at minimizing the time for one to be served, which takes into account both the travel time to the POI and the waiting time, if it is closed. Even though this is discussed in details in Chapter 3, let us consider the following example for the sake of motivation. Imagine that we are looking for the closest POI from us at 19:00 and the time it takes to reach the nearest POI is 20 minutes, but it opens at 20:00. However, it takes 25 minutes to reach the second nearest POI, but it opens at 19:30. A regular k-NN query returns the first POI as the answer, since it is the closest in terms of travel-time from our location, but when we get there we have to wait 40 minutes until this POI opens. Even though it takes 5 minutes more to reach the second POI, it is a better answer, since we just need to wait 5 minutes.

Although this type of query is useful, more often a user intends to make a plan for a trip passing through several locations belonging to certain categories in a given order. For example, when heading back home from work there are often things to do on the way like refueling at a gas station and grocery shopping. Naturally, it is preferable to find the fastest route

that meets the user needs. Motivated by this fact, we also address the problem of processing Optimal Sequenced Route (OSR) queries in time-dependent road networks. This query was originally proposed for static networks by (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008). An OSR query strives to find a route of minimum length starting from an origin location and passing through a number of POIs in a specific sequence imposed on the categories of POIs, before reaching a given destination. The time-dependent variation of this query, named TD-OSR in this thesis, returns the shortest route, in terms of time, considering a certain departure time.

1.2 Objectives

Given the motivating scenario presented above, the general objective of this work is to study the TD-*k*NN-OTC and the TD-OSR problems, proposing efficient and optimal solutions to them. To achieve this objective, we established the following specific objectives:

- To propose solutions to the TD-*k*NN-OTC problem considering that mobility applications may require different update rates of the network;
- To evaluate the proposed solutions to the TD-*k*NN-OTC problem and to indicate in which types of applications each one should be used;
- To propose an efficient algorithm to solve the TD-OSR optimally as well as a baseline solution, for comparison purposes;
- To evaluate the proposed solutions to the TD-OSR problem. Furthermore, to compare the optimal solution to a greedy solution in terms of processing time and quality of the routes returned.

1.3 Contributions

We propose solutions to the TD-*k*NN-OTC and to the TD-OSR problems which perform an incremental network expansion and use the A* search algorithm to guide this expansion. This algorithm determines the order in which vertices are expanded in a search by using a cost function. This function is a sum of two other functions: the known distance from the starting vertex to the current vertex plus a heuristic function that estimates the distance from this vertex to the goal. In order to speed-up the query processing, we split these solutions in two parts. During an offline phase, called pre-processing, we compute bound values that assist the calculation of heuristics which accelerate queries during the online phase, the query processing.

Mobility applications may require different update rate of the network. For instance, in a traffic management application, the time-dependent networks must be frequently updated in order to reflect the actual state of the traffic. A solution that needs a more costly pre-processing may not be suitable for such application since an update on the network may require the information computed in this step to be also updated. Based on this fact, we present

three solutions to the TD-*k*NN-OTC problem. Each solution is equipped with a suitable heuristic function and requires a different pre-processing. The more information is computed during the off-line phase, more accurate the heuristic function and more costly is the pre-processing.

We also propose an optimal algorithm to solve the TD-OSR problem. Furthermore, as an OSR query on road network tends to expand an extremely large number of nodes, we propose a scheme to reduce the number of nodes re-expansion. For comparison purposes, we also present a baseline solution which is obtained by extending the previously proposed progressive neighbor exploration algorithm to cope with the time-dependent problem.

The following items summarize the main contributions of this thesis:

- We propose and discuss three solutions to process TD-*k*NN-OTC queries in time-dependent networks;
- We propose an optimal algorithm to solve the TD-OSR problem efficiently as well as a baseline solution based on a previously proposed algorithm;

1.3.1 Publications

The efforts during the research process for this thesis made it possible the following publications:

- COSTA, C. F.; NASCIMENTO, M. A.; MACÊDO, J. A. F.; MACHADO, J. de C. Nearest neighbor queries with service time constraints in time-dependent road networks. In: Proc. of the 2nd ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems, 2013. p. 22–29.
- COSTA, C. F.; NASCIMENTO, M. A.; MACÊDO, J. A. F.; MACHADO, J. de C. A*-based Solutions for KNN Queries with Operating Time Constraints in Time-Dependent Road Networks. In: 15th IEEE International Conference on Mobile Data Management, 2014.

1.4 Structure of the thesis

The next chapters of this thesis are structured as follows:

- Chapter 2 presents the key concepts involved in this work. It formalizes the concept of time-dependent graph and presents some definitions useful to formalize our problems. Furthermore, we discuss in details the A* search (HART; NILSSON; RAPHAEL, 1968) and Incremental Network Expansion (INE) (PAPADIAS et al., 2003) algorithms, which are bases for our solutions and a method to access information about adjacency of vertices and history traffic in time-dependent networks.

- Chapter 3 presents the TD-kNN-OTC query in details and the proposed solutions to this problem, besides an experimental evaluation;
- Chapter 4 explains and evaluates the proposed approaches to the TD-OSR query;
- Chapter 5 concludes this thesis with a summary of our findings and some suggestions for further work.

2 THEORETICAL FOUNDATION

This chapter describes the concepts and model used for the representation of time-dependent networks and some of their properties. Section 2.1 presents the graph used to model this network, called time-dependent graph, as well as some concepts necessary to the formulation of the problems studied. In Section 2.2 we discuss in details the A* search (HART; NILSSON; RAPHAEL, 1968) and the Incremental Network Expansion (INE) (PAPADIAS et al., 2003) algorithms, which are of key importance to our proposed solutions. Section 2.3 describes an efficient access method for time-dependent networks. Finally, Section 2.4 concludes this chapter.

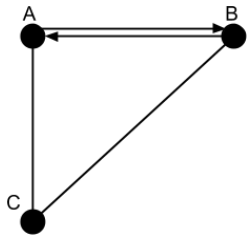
2.1 Time-Dependent Graph (TDG)

We assume that the structure of a time-dependent road network is modeled by a graph where the vertices represent starting and ending points of road segments or intersections. Those are connected by edges and the cost to traverse these edges vary with time. More formally, the network is modeled by a **time-dependent graph (TDG)** $G = (V, E, C)$, where V is a set of vertices, E is a set of edges and the cost (time in our domain of interest), represented by C, to traverse an edge is a function of the departure time. In other words, a TDG is a graph in which the costs of the edges varies with time. The concept of TDG is formally defined below.

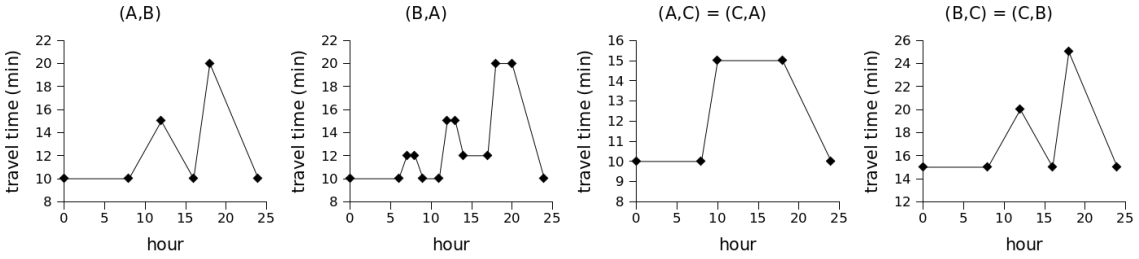
Definition 1. *A **time-dependent graph** (TDG) $G = (V, E, C)$ is a graph where: (i) $V = \{v_1, ..., v_n\}$ is a set of vertices; (ii) $E = \{(v_i, v_j) \mid v_i, v_j \in V, i \neq j\}$ is a set of edges; (iii) $C = \{c_{(v_i,v_j)}(\cdot) \mid (v_i, v_j) \in E\}$, where $c_{(v_i,v_j)} : [0,T] \rightarrow R^+$ 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.*

We assume that C is a set of piecewise-linear functions that are defined in the interval [0, T] where T is a domain-dependent time length. Particularly, in this work we assume that T is equal to 24, i.e., we model a day in terms of whole hours. For each edge (u, v), a function $c_{(u,v)}(t)$ gives the cost of traversing (u, v) at the departure time $t \in [0, T]$. We also assume that the travel times of the edges in the network follow the FIFO property, i.e., an object that starts traversing an edge first has to finish traversing this edge first as well. The general time-dependent shortest path problem in which the departure is immediate, i.e. the user departs exactly at the time t, and in which waiting is disallowed everywhere along the path through the network is NP-hard (ORDA; ROM, 1990), but it has a polynomial time solution in FIFO networks. Since the travel times satisfy the FIFO property, waiting in a intermediary vertex in a path is not beneficial.

Note that the definition given above does not require the graph to be bidirected. More specifically, the existence of an edge (u, v) does not imply in the existence of the edge (v, u). Furthermore, there may be opposing edges (u, v) and (v, u) such that $c_{(u,v)}(t) \neq c_{(v,u)}(t)$. As an example, consider the graph shown in Figure 2.1 which is a representation of a time-dependent road network. The travel times of its edges for each instant of a day are shown in



(a) A graph representing a road network.



(b) Cost functions, or travel time on edges.

Figura 2.1: A graph representing a road network and the costs of its edges for different times of a day.

the graphics in Figure 2.1b. The pairs of opposite edges (A,C) and (C,A) and (B,C) and (C,B) have the same cost. However, (A,B) and (B,A), although opposite, have distinct costs.

The time cost to traverse a path from a specific starting time, or departure time, is called *travel-time*. The *travel-time* is calculated assuming that stops are not allowed because, as discussed before, we consider that the network is FIFO waiting in a vertex do not anticipate the arrival time of a vehicle. The *travel-time* of a path is calculated considering the *arrival time* at each vertex belonging to it. These concepts are formally defined below.

Definition 2. *Given a TDG G = (V,E,C), the* ***arrival time*** *at the vertex v_j of an edge (v_i, v_j) $\in E$ at departure $t \in$ [0, T] is given by $AT(v_i, v_j, t) = t + c_{(v_i,v_j)}(t)$ mod T.*

Given that a vehicle starts a path at a vertex of the graph and this path starts at a determined departure time, the *arrival-time* calculates the time instant when the vehicle arrives at the other end of the edge. Considering the cost functions shown in 2.1b, when traversing the road represented by (B,A) at 10:00 am, a vehicle arrives at 10:10 am at A. Note that the operation of the rest (mod) exists for the calculation of the arrival-time be circular. For instance, consider that one departs from B towards A at t = 24:00, $AT(v_i, v_j$,24:00) is 0:10, since the vehicle arrives at the other end of the edge at this time.

Definition 3. *Given a TDG G = (V,E,C), a path $p = \langle v_{p_1}, ..., v_{p_k} \rangle$ 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 $t_{i+1} = AT(v_{p_i}, v_{p_{i+1}}, t_i)$.*

The definition given above shows how the cost of a path, called *travel-time*, is calculated. Given the sequence of vertices that compose a path and the time instante when one starts to traverse this path, the *travel-time* is the sum of the costs to go from one vertex to the next one

in the sequence. The cost to go from the first to the second vertex is calculated considering the departure time t. The cost to reach the next vertices depends on the *arrival time* at the previous vertex. It is important to notice that this definition does not take into consideration stops at the nodes of the graph, that is, the way to the next vertex in the sequence begins at the same moment when the previous vertex was reached. As an example, consider the path $\langle B,A,C\rangle$ in the graph shown in Figure 2.1a. At a departure time t = 10:00 am, the cost of traversing this path is given by $TT(\langle B,A,C\rangle$, 10:00) which is equal to 25 minutes, since the cost to go from B to A at 10 am is 10 minutes and the arrival time at A is 10:10 am and the cost to go from A to C at 10:10 am is 15 minutes.

For simplicity, we assume that points of interest as well as origin and destination (an input parameter of the TD-OSR query) points are located on a vertex throughout this thesis. On the original network, those points are not necessarily vertices, however, they can be transformed into new vertices of the network as shown in (CRUZ; NASCIMENTO; MACÊDO, 2012). That work proposes the IncludePOI algorithm which take as input a TDG G and a POI $p = \langle (u,v), \tau_p \rangle$, where (u,v) is the edge over which p is positioned and τ_p is a ratio which indicates how far p is from the begin of the edge (u). To illustrate how this algorithm works, let us consider that we want to include a new vertex (not necessarily a POI) represented by $p = \langle (C,A), \frac{1}{3} \rangle$ in the network shown in Figure 2.1a. The IncludePOI algorithm works as follows. It first inserts the new vertex p in the set of vertices V. Figure 2.2a shows the network with the inclusion of p. As p is a point over (C,A), (C,A) is removed from E and two new edges (C,p) and (p,A) are created. The travel time functions for (C,p) and (p,A) are $c_{(C,p)} = \frac{1}{3} c_{(C,A)}$ and $c_{(p,A)} = \frac{2}{3} c_{(C,A)}$ and the travel time function $c_{(C,A)}$ is removed from C. The graphics representing the travel times of the new edges are shown in Figure 2.2b. Next, as (A,C) is also in E, we need to repeat the same process executed for the edge (C,A). (A,C) is removed from E and the edges (A,p) and (p,C) are created. The new cost functions are $c_{(A,p)} = \frac{2}{3} c_{(A,C)}$ and $c_{(p,C)} = \frac{1}{3} c_{(A,C)}$ as shown in Figure 2.2b.

2.2 INE and A* search algorithms

In this section we discuss in details how the Incremental Network Expansion (INE) and the A* search algorithms, bases for the proposed solutions in this thesis, work.

2.2.1 Incremental Network Expansion (INE)

The problem of processing k-nearest neighbor (k-NN) queries in road networks has been investigated since the pioneering study by (PAPADIAS et al., 2003), where the Incremental Euclidean Restriction (IER) and the Incremental Network Expansion (INE) methods were proposed.

The basic idea of the IER method is to first find the k POIs from the query point q on the Euclidean distance using R-trees (GUTTMAN, 1984). Then, the network distances from q to these POIs are calculated and the distance to the farthest of these POIs is used as an upper bound. Next, all the POIs with an Euclidean distance from q less or equal to the upper bound

are investigated, that is, their network distances are calculated, because they offer a chance to be part of the k-NN result. Figure 2.3 shows how this method works when one NN is required. IER first retrieves the Euclidean nearest neighbor p_{E1} of q. Then, the network distance $d_N(q, p_{E1})$ of p_{E1} is computed. This distance is then used as an upper bound, that is, all the objects closer (to q) than p_{E1} in the network, should be within Euclidean distance at most $d_N(q, p_{E1})$, and, thus, they should lie in the shaded area of the left figure. Next, as shown in the right figure, the second Euclidean NN, p_{E2}, is found. Similarly, the network distance $d_N(q, p_{E2})$ of p_{E2} is computed. Since $d_N(q, p_{E2}) < d_N(q, p_{E1})$, p_{E2} becomes the new NN and the upper bound is updated accordingly. As the distance to the next Euclidean NN p_{E3} is greater than $d_N(q, p_{E2})$, the algorithm stops and returns p_{E2} as the nearest neighbor.

Clearly, the problem with this approach is that, generally, k-NN POIs on the Euclidean distance are not always k-NN on the road network distance, especially when time-dependent costs are considered. Thus, several false hits must be investigated. To remedy this problem, the Incremental Network Expansion (INE) algorithm was proposed. It performs network expansion and searches neighbor POIs by visiting vertices in order of their proximity from q, using Dijkstra's algorithm (DIJKSTRA, 1959), until all k nearest points of interest are located. Returning to the example shown in Figure 2.3, as p_{E2} is the NN from q considering the network distance, the INE algorithm first locates this POI without investigating p_{E1}. The blue shaded area in Figure 2.4 indicates the search area on the road network of a k-NN query with the INE approach for k = 10. As shown in this figure, the search area is enlarged from q until the k POIs have been found.

One drawback of this algorithm is that the search is not guided, i.e., the vertices are examined in order of proximity from q without any estimate for the cost of achieving the POIs. In order to guide the execution of this method, we incorporate an A* search to the INE expansion, being possible to discard the verification of paths that do not lead to the solution. We explain how this search works below.

2.2.2 A* search

The A* search is an algorithm that was originally proposed to find the shortest path from an origin to a goal node and it is similar to Dijkstra's algorithm. The main difference to this algorithm lies in the use of a potential function that guides the search towards the goal. The A* algorithm determines the order in which vertices are expanded in a search by using a cost function, $f(v)$. This function is a sum of two other functions: the known distance from the starting vertex to the current vertex, $d(q, v)$, plus a heuristic function, $h(v)$, that estimates the distance from this vertex to the goal.

As A* traverses the graph, it follows a path of the lowest expected total cost. It maintains a priority queue of nodes to be traversed and it expands first vertices that appear to be most likely to lead towards the goal. The lower $f(v)$ for a given node v, the higher its priority. At each step of the algorithm, the node with the lowest $f(v)$ value is removed from the queue. Then, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm stops when the destination vertex is removed from the queue or

when the queue is empty.

If the potential function h does not overestimate the cost to reach the goal from all $v \in V$, then A* always finds shortest paths. If h(v) is a good approximation of the cost to reach the goal, A* efficiently drives the search towards the goal, and it explores considerably fewer nodes than Dijkstra's algorithm. If $h(v) = 0 \ \forall v \in V$, A* behaves exactly like Dijkstra's algorithm, i.e., it explores the same vertices.

Unlike the solutions used in the calculation of shortest paths, in the proposed solutions in this thesis, the A* search is incorporated directly into the incremental expansion of the network, rather than being used to calculate the travel-time from the query vertex q to each candidate point of interest. Particularly, in the TD-kNN-OTC query, where we aim at finding the nearest POIs from q considering the operating time of the POIs, there are multiple and unknown goals. In the TD-OSR query, the goal is not only to reach the destination the quickest, but also to pass through a number of POIs belonging to certain categories in a given order.

2.3 Access Method for TDGs

Building strategies and algorithms for correct and efficient query processing in time-dependent networks is a challenge, since the common properties of graphs can not be satisfied in the time-dependent case (GEORGE; KIM; SHEKHAR, 2007). Particularly, these networks can not be stored in the same way than a static network, the same applies to the access to the network information. Thus, it emerges the need for storage methods that facilitate the access to the network information and that support the design of efficient algorithms for computing the frequent queries on such networks.

Some characteristics of the time-dependent networks should be considered in the development of this method. First of all, these networks require more space than the static ones to store the costs, since, for each edge, we need to keep the cost to traverse it for each time interval of constant size. Another important observation is that the cost of storing the edges of the network grows as the time granularity (number of intervals) increases. Finally, to store the costs of traversing an edge for all the time intervals together implies accessing unnecessary information when retrieving disk page(s) that contains this edge, since the access to the adjacency list is executed to get the cost of the edges for a given time.

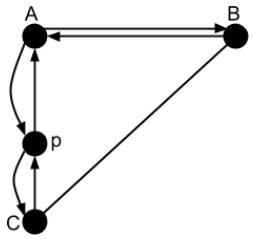
Based on these observations, in order to process our queries in a more efficient way, we resort to the access method proposed by (CRUZ; NASCIMENTO; MACÊDO, 2012). It is important to notify that we use this method without any modification or extension. It is composed by three levels, the Time-Level, the Graph-Level and the Data-Level shown in Figure 2.5. The data pages in the time-level contain pointers to index structures in the graph-level. As a TDG can be seen as a set of static graphs for each time interval, the idea behind the first level is to access first the graph corresponding to a given time interval, avoiding retrieving the edge costs for every possible departure time. The graph-level has an index structure for each time partition, such that it is possible to, given a vertex identifier (*Nid*), access its adjacency list in the graph corresponding to the departure time. 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.

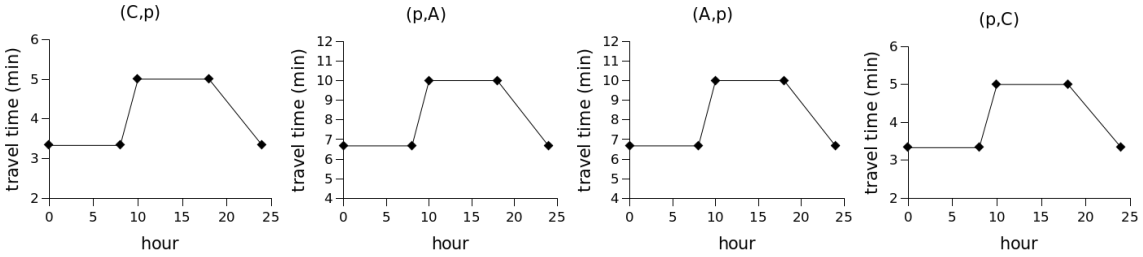
The index structures for the time and graph levels are generic in the originally proposed method. In this work, we opted to use a B^+-tree in the two index levels, which is provided by the XXL library (BERCKEN et al., 2001). The pointers to the graph and data levels are stored in the leaves and each node (including the internal nodes) is a page in disk. Thus, the number of pages accessed for each retrieved data entry is of the order of $O(\log_b |TP| + \log_b |V|)$, where b is the order of the tree, TP is the number of temporal partitions that compose the cost of an edge and V is the number of vertices.

2.4 Conclusion

In this chapter we formally defined the concept of time-dependent graph and presented some definitions useful to formalize our problems in the following chapters. Furthermore, we discussed how the IER and the INE algorithms work. We showed that the first one is not suitable to be applied in our solutions because POIs are investigated in order of their Euclidean distance. However, generally, k-NN POIs on the Euclidean distance are not always k-NN on the road network distance, especially when time-dependent costs are considered. Thus, a number of unnecessary network distance has to be computed, which is very costly. The INE algorithm was proposed to cope with this problem. It searches neighbor POIs by visiting vertices in order of their proximity from q. One drawback of this algorithm is that it performs a blind search. Thus, we proposed to incorporate an A* search directly into the INE algorithm in order to guide it. We also presented an efficient method to access adjacency lists of time-dependent graphs, which is based on some particular characteristics of time-dependent networks that differentiate them from static networks.



(a) A graph representing the network with the inclusion of the new vertex p.



(b) Cost functions of the new edges.

Figura 2.2: A new graph representing the inclusion of the vertex p over the edge (A,C) and the travel time functions of the new edges created.

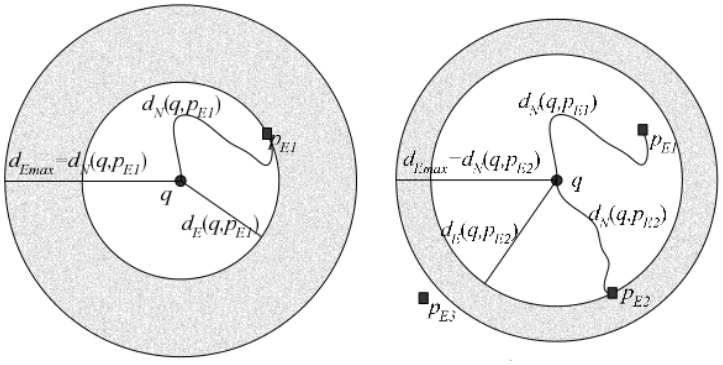


Figura 2.3: Illustration of the Incremental Euclidean Restriction (IER) (PAPADIAS et al., 2003).

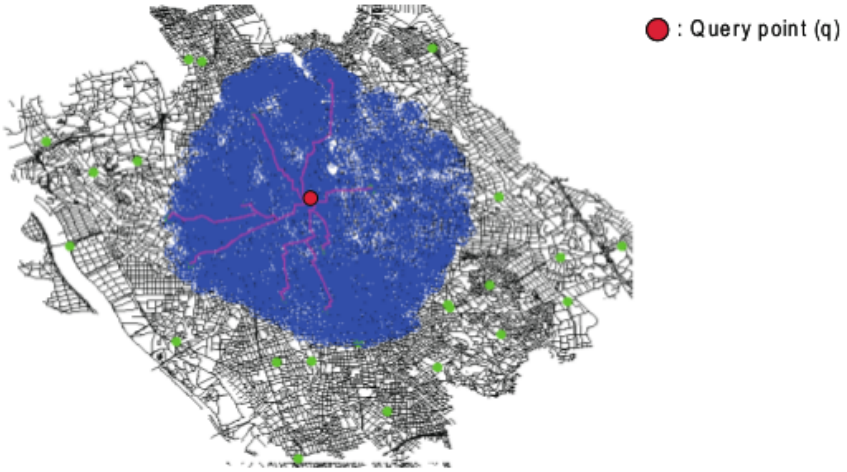


Figura 2.4: Illustration of the Incremental Network Expansion (INE) for $k = 10$. Source: (HTOO, 2013).

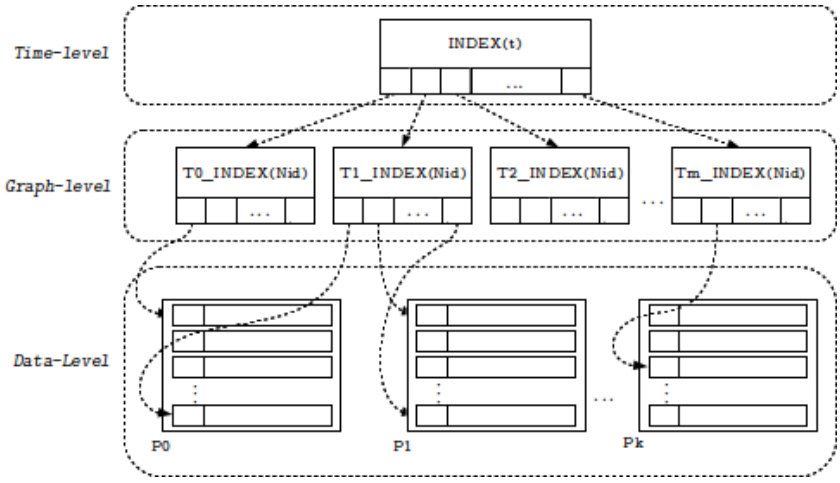


Figura 2.5: An access method for time-dependent road networks (CRUZ; NASCIMENTO; MACÊDO, 2012).

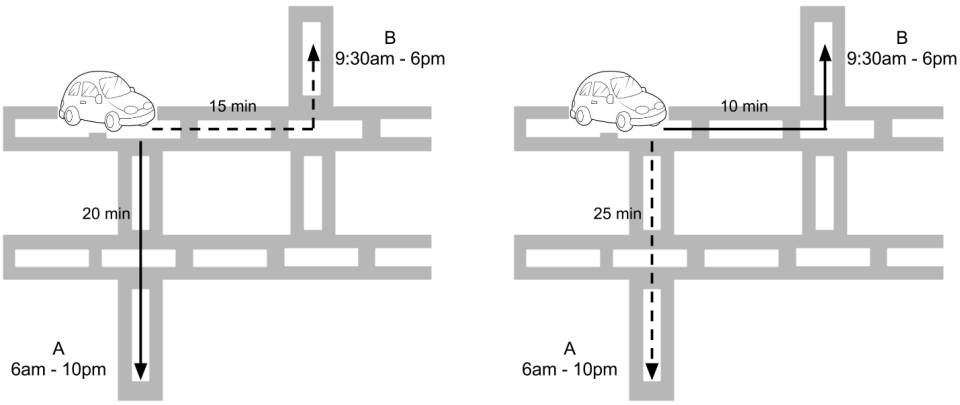
3 K-NEAREST NEIGHBOR QUERIES WITH OPERATING TIME CONSTRAINTS

3.1 Introduction

The problem of finding the k nearest neighbors in time-dependent road networks was introduced by (DEMIRYUREK; BANAEI-KASHANI; SHAHABI, 2010b) and an improved solution was proposed in (CRUZ; NASCIMENTO; MACÊDO, 2012). The query addressed in these works returns the k points of interest (POIs), for example, museums or restaurants, that are closest in terms of travel time from a query point q given a certain departure time t. As only the travel time is considered, the answer to this query can lead to POIs that are closer to q but that are not operational, i.e., are not open yet and/or will take a long time to open.

In this chapter, we study the problem of processing k nearest neighbors (kNN) queries considering the operating times of the facilities in time-dependent road networks. Differently from previous works, we aim at minimizing the time for one to be served, instead of just search for facilities that can be reached the quickest.

The following scenario illustrates the difference between our query of interest and a query oblivious to operating time constraints (for k = 1). Imagine a tourist who is interested in visiting the touristic attraction closest to him/her. Let us consider two points of interest in the city, A and B as shown in Figure 3.1a. At 8 am, a regular query returns B as the nearest neighbor, but when the tourist gets there, he/she has to wait 1 hour and 15 minutes. Even though the travel time to A is longer, it is a better answer, since he/she does not need to wait. In some cases, the answer returned by a regular NN query is coincidentally the same as the one returned by our query. For example, consider Figure 3.1b, at 9:15, B is the nearest POI both in terms of travel time and time to service. The travel time to this POI is 10 minutes and when the tourist arrives there, he/she only needs to wait 5 minutes, so the total time to service is 15 minutes, whereas for A, the time to service, which is also equal to the travel time, is 25 minutes.



(a) Travel Times at 8 am. (b) Travel Times at 9:15 am.

Figura 3.1: Travel Times from the tourist location to the POIs A and B at two different moments of a day. The operating times are shown below the POIs.

It is important to notice that, in the case where POIs are open and able to serve all the time, the solution to both types of queries is the same, thus our proposal generalizes previous solutions, i.e., nearest neighbor queries on non-time dependent networks and/or without operating times constraints.

In this chapter we propose three solutions to process k nearest neighbor queries in time-dependent networks considering that the operating times of the POIs may be limited. Each solution requires a different pre-processing step in order to calculate bounds to guide the search for POIs and to be used in a pruning process. In fact, our decision on proposing three different solutions to this problem is justified by the fact that mobility applications may require different update rate of the network. For example, in a traffic management application the time-dependent networks must be frequently updated in order to reflect the actual state of the traffic. In contrast, in a touristic application the time-dependent network is less dynamic, requiring updates only when tourist places operating time change. Thus, since pre-processing step is of key importance to our solution, we decide to provide three solutions to accommodate the static and dynamic network scenarios. The first solution uses loose bounds in exchange for a low cost pre-processing. The second one has an intermediate pre-processing cost and uses bounds which have a tightness between the ones used in the other solutions. Finally, the third solution requires a more costly pre-processing in order to calculate tighter bounds.

The remaining of this chapter is structured as follows. We introduce some important definitions for understanding our proposed solutions and formalize the problem in Section 3.2. In Section 3, we present a brief discussion of related works. In Section 4, we explain our proposed approaches. The experimental evaluation and results are shown in Section 5. Finally, Section 6 concludes this chapter.

3.2 Problem Definition

The concepts of *waiting time* and *time to service* are useful to formalize the problem addressed in this chapter, since our query aims at minimizing the *time to service* of a POI, which takes into account both the *travel time* (discussed in chapter 2) and the *waiting time* at a POI. We formally define these concepts belows.

Definition 4. *Given a POI p_i. Let the arrival time be denoted as $a = AT(q, p_i, t)$. The **waiting time** $W(p_i, a)$, for p_i is the amount of time that takes for p_i to open from the time a. If a is greater or equal to the opening time (OT_{p_i}) of p_i and is less or equal to the closing time (CT_{p_i}), $WT(p_i, a) = 0$. If $a < OT_{p_i}$, $WT(p_i, a) = OT_{p_i} - a$. Otherwise, if $a > CT_{p_i}$, $WT(p_i, a) = (T - a + OT_{p_i})$. In other words, it is necessary to wait until p_i opens in the next day.*

Let us suppose that the operating time of a POI A is from 9:00 to 17:00. If, for example, this POI is reached at 10:30, there is no waiting. If one arrives there at 8:30, the waiting time is 30 minutes. On the other hand, if it is reached at 18:00, is necessary to wait until it opens the next day and thus, the waiting time is 15 hours.

Definition 5. *Given the query point $q \in V$, the departure time $t \in [0, T]$ and a POI $p_i \in V$,*

*the **time to service** for p_i is given by $TS(q,p_i,t) = TT(q,p_i,t) + WT(p_i,AT(q,p_i,t))$, where $TT(q,p_i,t)$ is the travel time from q to p_i for the departure time t.*

To exemplify the definition presented above, let us suppose that a vehicle departs from a vertex q at t = 8:00 and the travel time to the POI A is 20 minutes. As this POI is reached at 8:20 and is opens at 9:00, the waiting time is 40 minutes and thus, the time to service of A is 60 minutes.

The problem of processing k nearest neighbors queries in road networks where the travel time is time-dependent and the operating times of the facilities are taken into consideration can now be defined as follows:

Problem Definition 1. *Let $G = (V,E,C)$ be a TDG and $P \subseteq V$ be a set of points of interest in G. Given a query point q and a departure time t, a time-dependent k nearest neighbor with operating time constraints query (TD-kNN-OTC) returns a set $R = \{v_{r_1},...,v_{r_k}\} \subseteq P$ such that $\forall v \in P \setminus R$, $TS(q,v_{r_i},t) \leq TS(q,v,t)$. In other words, a TD-kNN-OTC query returns the k points of interest where the time to service from q is smaller than the others remaining points of interest considering a departure time t.*

3.3 Related Work

The problem of processing kNN queries in road networks was first addressed in (PAPADIAS et al., 2003). Two solutions were presented in that paper: the Incremental Euclidean Restriction (IER) and the Incremental Network Expansion (INE) algorithms. IER uses the Euclidean distance between two points on the network as a lower bound, since that distance is less than the network distance. The points are retrieved according to the Euclidean distance and the network distance is used as an upper bound to avoid expanding vertices that have an Euclidean distance greater than the current network distance. The INE algorithm performs network expansion from a query point q and examines entities in order of their proximity from q until all k nearest data objects are located.

Another solution to the problem of processing kNN queries in road networks was proposed in (KOLAHDOUZAN; SHAHABI, 2004). The authors execute a pre-processing step to compute the network's voronoi polygons (NVP) (ERWIG; HAGEN, 2000). The cost of a kNN search can be reduced by using NVPs, since it is easy to retrieve the nearest neighbor of a query point. In (HU; LEE; XU, 2006) the authors developed a network reduction technique where the network topology is simplified by a set of interconnected tree-based structures (SPIE's). By doing that the number of edges is reduced while all network distances are preserved. They proposed the *nd* (nearest descendant) index on the SPIE such that a kNN query on those structures follows a predetermined tree path, avoiding unnecessary network expansion. In (LEE; LEE; ZHENG, 2009) the authors exploited the search space pruning technique. With the observation that during a search some subspaces of the network with no objects can be skipped, they organized a road network as a hierarchy of interconnected regional sub-networks (Rnets). They speed-up the search performance by incorporating shortcuts that avoid detailed traversal

and object lookup within Rnets, allowing bypass those Rnets that do not contain objects of interest.

These solutions, however, do not consider the time dependency of the network, neither the operating times of the facilities. Therefore they are not suitable to solve the query we are interested in.

The problem of kNN queries in time-dependent network was introduced in (DEMIRYUREK; BANAEI-KASHANI; SHAHABI, 2010b), where two baseline methods to solve this problem were presented. In the first approach the network is modeled as time-expanded graphs, allowing the use of previous solutions in static networks. However, it has been show in (DEMIRYUREK; BANAEI-KASHANI; SHAHABI, 2010b) that this solution is not efficient, yielding high storage overhead, slower response time and also incorrect results. In the second one, they exploited a generalization of INE algorithm (PAPADIAS et al., 2003), that was originally proposed for static road networks.

In (DEMIRYUREK; BANAEI-KASHANI; SHAHABI, 2010a) the authors proposed an algorithm that involves an off-line spatial network indexing phase, that builds two different indexing structures, the Tight Network Index (TNI) and Loose Network Index (LNI), and an on-line query processing phase. TNI and LNI 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. Using TNI one can immediately find the nearest neighbor of a query object. For $k > 1$, the next nearest neighbor is the POI generator of neighboring cells to cells of the points of interest which have been found. To decide which is the next point of interest, the network is expanded incrementally until finding a POI in one of the neighboring cells.

An improved solution was proposed by (CRUZ; NASCIMENTO; MACÊDO, 2012). In this work, the authors proposed an algorithm that is based on the INE expansion (PAPADIAS et al., 2003) and uses an A* search to guide this expansion. The vertices that offer a great chance to be in a fast path to a POI are expanded first. A pruning process was also proposed in order to avoid expanding vertices that are far to any POI.

These solutions are time-to-service oblivious, i.e., the answer to these queries can lead to POIs that are closer to a query point, but that are not open and may take a long time to open. On the other hand, our query aims at finding the k POIs in which the user can be served in the minimum amount of time, accounting for both the travel time to the facility and the waiting time. As far as we know there is no published research addressing this specific type of query.

3.4 Proposed Approaches

In this section, we present three solutions to process TD-kNN-OTC queries, namely Naive, Optimistic and Bounded. All of them are based on an algorithm that performs as incremental network expansion (INE) (PAPADIAS et al., 2003) and uses an A* search (HART; NILSSON; RAPHAEL, 1968) to guide this expansion, i.e., to determine the order in which vertices are expanded in the search tree. It uses the current distance plus a heuristic function $H(.)$,

which, in our case, is an estimate to the time to service. Each solution has a different heuristic function, which in combination with a pruning process is going to determine its efficiency in terms of the query processing. A pre-processing in the TDG is needed in order to compute lower bounds that assist the computation of the heuristic function values and upper bounds to assist the pruning of unpromising vertices.

Algorithm 1 (TD-kNN-OTC) presents the general structure of the algorithm to solve the TD-kNN-OTC problem. As the three solutions differ only in the heuristic function and in the pruning process, we use this algorithm as the baseline, indicating the differences between the solutions.

3.4.1 Naive Solution (Baseline)

This approach is basically an extension of the TD-NE-A* algorithm proposed by (CRUZ; NASCIMENTO; MACÊDO, 2012) to cope with TD-kNN-OTC problem. That work addresses the problem of processing k-NN queries in time-dependent networks. The TD-NE-A* solution performs a guided incremental expansion of the network by using the previously proposed INE (PAPADIAS et al., 2003) and A* search (HART; NILSSON; RAPHAEL, 1968) algorithms.

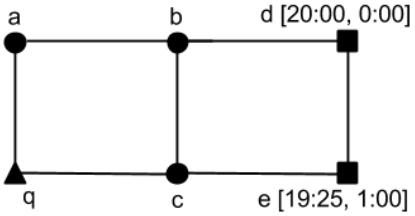
In the TD-NE-A* algorithm, the heuristic function h(.) adds to each vertex an estimate of the cost to reach any POI from it. The idea behind it is to avoid expanding nodes in a path that is fast but is far to any POI. Vertices that offer a greater chance of achieving POIs quickly are expanded first.

Although that solution does not take into consideration the operating time of POIs and does not try to minimize the time to start to be served, the heuristic used in that work can also be used as an estimate for the time to service. Thus, this heuristic is used in our first solution to the TD-kNN-OTC problem. The idea is to reach POIs the quickest. Then, after they are reached, we calculate the waiting time until they open. This is clearly a sub-standard solution, but which serves nevertheless as a baseline.

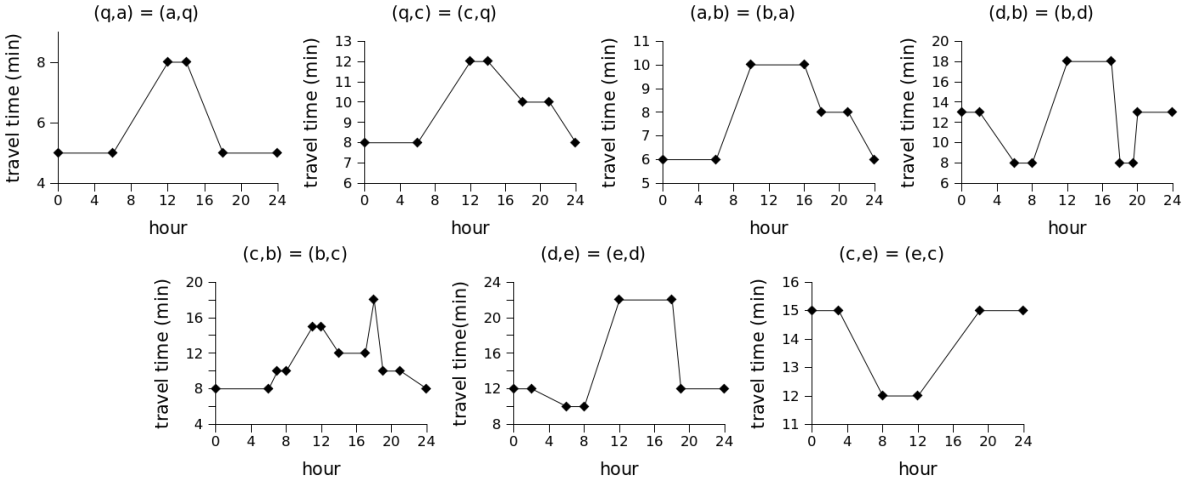
As discussed before, a pre-processing step is needed in order to calculate heuristic function values which accelerate the query processing. In this solution, these values are optimistic estimates of the cost to reach any POI from a given vertex. Additionally, in this step we also calculate upper bound values to support the pruning of unpromising vertices during the search. In order to calculate these values, we construct a lower bound graph ($\underline{G}$) and an upper bound graph ($\overline{G}$). Given a TDG G, $\underline{G}$ is a graph that has the same set of vertices as G and the cost of an edge (v_i, v_j) is given by the minimum cost possible to go from v_i to v_j for any time in G. Similarly, in $\overline{G}$, the cost of the edges are given by the maximum time possible to traverse the edge (for any time).

As an example, consider the TDG shown in Figure 3.2a. The costs of its edges for each time of a day are shown in Figure 3.2b. The POIs are represented by squares and the operating time of a POI is shown beside it. The $\underline{G}$ and the $\overline{G}$ graphs of this TDG are shown, respectively in Figures 3.3a and 3.3b. Note that $\underline{G}$ and $\overline{G}$ are static graphs, i.e., the edges

costs are constant and is given by the minimum and maximum, respectively, points of the cost functions of the edges.

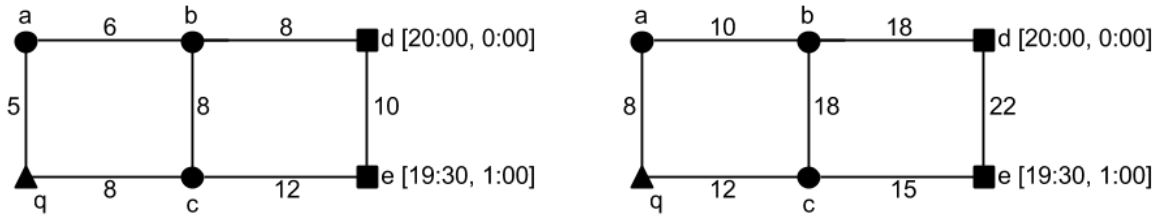


(a) A network, a query point q, represented by a triangle, points of interest d and e, represented by squares, and their operating times.



(b) Time-dependent edges cost.

Figura 3.2: A road network and its respective graph considering points of interest and their operating times.



(a) Lower bound graph.

(b) Upper bound graph.

Figura 3.3: Lower and upper bound graphs of the TDG shown in Figure 3.2a.

We define $L(v_i, v_j)$ and $U(v_i, v_j)$ as the travel time of the fastest path between v_i and v_j in $\underline{G}$ and $\overline{G}$, respectively. Note that the values of $L(v_i, v_j)$ and $U(v_i, v_j)$ are not dependent on a departure time, since the cost functions in $\underline{G}$ and $\overline{G}$ are constants. To illustrate the concept of $L(v_i, v_j)$, let us consider again the $\underline{G}$ shown in Figure 3.3a. For example, $L(a, d) = 14$ because this is the cost of the shortest path $\langle a, b, d \rangle$ between a and d in this graph. This cost indicates that the minimum cost to go from a to d, at any departure time, is 14 minutes. Similarly, the cost

of $U(a,d)$ is 28, meaning that departing from a it is possible to reach d in at most 28 minutes for any departure time.

Let p be the nearest POI from u in $\underline{G}$, the naive heuristic function value of a vertex u is then given by $H_N(u) = L(u,p)$. This function is an optimistic expectation to the travel time of a path that connects u to p. As an example, in the lower bound graph shown in Figure 3.3a, d is the nearest POI of a and the cost of the shortest path from a to d is 14 minutes and thus, $H_N(a) = 14$. This means that, a POI is reached from a in at least 14 minutes. Note that this value does not overestimate the cost to reach a POI from a for any departure time.

In order to avoid expanding unpromising vertices, we also calculate an upper bound value $UB(u,ATu)$, that depends on the time $ATu = AT(q,u,t)$ when a vertex u is reached. We denote the nearest POI from u in $\overline{G}$ by $UNN(u)$. The pessimistic expectation to the time to service of this POI, from u, is given by $UB(u,ATu) = U(u,UNN(u)) + WT(UNN(u),ATu + U(u,UNN(u)))$, meaning that it is possible to reach $UNN(u)$ from u and start to be served in at most $UB(u,ATu)$ units of time. These values are used to prune vertices that have an estimate to reach a POI greater than the time to service of a set of candidates. Note that the values of $UB(u,ATu)$ cannot be pre-computed, since they depend on the time when u is reached in order to calculate the waiting time based on the pessimistic travel time to $UNN(u)$, but $U(u,UNN(u)))$ and $UNN(u)$ can be calculated off-line.

To illustrate the definitions presented above, let us consider the upper bound graph shown in Figure 3.3b. The nearest neighbor of a in this graph is d, thus $UNN(a) = d$ and $U(a,d) = 28$. Now, let us suppose that one departs from q, shown in Figure 3.2a, at t = 19:00. The travel time from q to a at this time is 20 minutes and, thus, the arrival time there is at = 19:20. The upper bound value of the vertex a is then given by $UB(a,19{:}20) = U(a,d) + WT(d,19{:}20+U(a,d)) = 28 + WT(d,19{:}48) = 28 + 12 = 40$. This means that it is possible to start to be served at the POI d in at most 40 minutes from a. Thus, as the travel time from q to a is 20 minutes, the time to service of d from q is at most 60 minutes.

3.4.1.1 Off-line Pre-processing

The Naive solution has two pre-processing steps that are executed off-line, i.e., before processing the query. In both steps, an algorithm to find the nearest point of interest in static networks is executed.

The first step calculates the value of the heuristic function to the vertices of G. For each vertex $v \in V$, the distance between it and the nearest point of interest in $\underline{G}$ is calculated. This distance is used as an estimate to the time to reach a POI from v. The second step computes the nearest neighbor of v in $\overline{G}$, denoted by $UNN(v)$ and the distance from v to $UNN(v)$ in $\overline{G}$, denoted by $U(v,UNN(v))$. The values $UNN(v)$ and $U(v,UNN(v))$ are used to calculate upper bounds to the time to be served, allowing us to prune vertices in which the optimistic estimate to reach a POI is greater than the upper bounds found during the search.

We run Dijkstra's algorithm (DIJKSTRA, 1959) from each vertex v to find its nearest neighbor in $\underline{G}$ and in $\overline{G}$. In the worst case, this algorithm runs in time $O(|E| + |V| \log |V|)$

for each execution. As we need to find the nearest neighbor from each vertex, the total complexity in the worst case is $O(|V||E|+|V|^2\log|V|)$ for each step. The worst case happens when the whole graph is expanded and all the edges are traversed for each search. Note that this is very unlikely to occur in this specific search because as soon as the nearest POI of a vertex is found, the search stops.

It is important to observe that all the vertices in the path from a vertex v to its nearest neighbor p also have p as their nearest neighbor. Thus, we do not need to start one search for each vertex in the network, improving the time of execution.

3.4.1.2 Query Processing

Algorithm 1 (TD-kNN-OTC) presents the general structure of the algorithm to solve the TD-kNN-OTC problem. As the three solutions differs only in the heuristic function and in the pruning process, we use the same algorithm, indicating where each solution differs from the others. It takes as input the query point $q \in V$ and the departure time $t \in [0,T]$.

First, it inserts q in a priority queue Q that stores the set of candidates for expansion in the next step. An entry in queue Q is a tuple $(v_i, ATv_i, TTv_i, LBv_i)$, where $TTv_i = TT(q,v_i,t)$, $ATv_i = AT(q,v_i,t)$ and LBv_i is given by $TTv_i + H_N(v_i)$, i.e. the sum of the travel time from q to v_i plus the optimistic estimate to the travel time from v_i to its nearest POI. The priority of elements in Q is given by the increasing order of LBv_i values with the purpose of checking first the vertices that offer a greater chance to reach a POI quickly.

The vertices are de-queued from Q (line 8) and expanded (line 18). Then, some conditions are checked to determine whether its neighbors will be inserted in Q. A priority queue Q_U is maintained to store upper bound values, i.e., pessimistic estimate of the time to reach POIs and to start to be served. An entry in this queue is a tuple $(UNN(v), U_{UNN(v)})$, where $U_{UNN(v)} = TT_v + UB(v,AT_v)$, that is, the travel time from q to v plus the pessimistic expectation to reach $UNN(v)$ from v. For each POI given by $UNN(v)$, we maintain the lowest upper bound found in the expansion. This queue is ordered by increasing order of $U_{UNN(v)}$ values and is used in the pruning process. More specifically, a vertice v can be discarded (line 22) if LB_v is greater than $Q_U[k]$ (the k-th element in Q_U), meaning that we already have k candidates that have, in the worst case, a time to service less than the optimistic estimate to reach a POI (not necessarily to be served) in a path that passes through v. When a vertex v is reached, we check if $UNN(v)$ is in Q_U (line 31). If it is not, we include $UNN(v)$ and its upper bound $U_{UNN(v)}$ in Q_U. If $UNN(v,AT_v)$ is already in Q_U and the value given by $U_{UNN(v)}$ is smaller than the current upper bound to it, we update the upper bound value of $UNN(v)$ in Q_U and its position in the queue, if it is necessary.

If a vertex v has already been removed from Q and it is found in another path starting from q, v is not reinserted in Q (this is verified on line 23). We can avoid re-expand v because it has already been found in a shortest path. Due to the FIFO nature of the network, re-expand this vertex can not improve the total travel time to a POI. Moreover, the earlier a POI is achieved, the shorter its time to service (this is proved in Lemma 3.4.1 in Subsection 3.4.6). Thus, the time

Algorithm 1: TD-kNN-OTC

Data: A query point $q \in V$ and the departure time $t \in [0,T]$
Result: The nearest neighbor from q considering the departure time t

$TT_q \leftarrow 0$;
$AT_q \leftarrow t$;
$LB_q \leftarrow H(q)$;
$Q \leftarrow \emptyset$;
En-queue (q, AT_q, TT_q, LB_q) in Q;
$C_{NN} \leftarrow \emptyset$;
while $Q \neq \emptyset$ **do**
    $(u, AT_u, TT_u, LB_u) \leftarrow$ De-queue Q;
    Mark u as de-queued;
    if $LB_u > C_{NN}[k]$ **then**
        Return $C_{NN}[1...k]$;
    end
    if *u is POI* **then**
        $W_u \leftarrow WT(u, AT_u)$;
        $C_{NN} \leftarrow C_{NN} \cup (u, TT_u + W_u)$;
        Re-order C_{NN};
    end
    for $v \in adjacency(u)$ **do**
        $TT_v \leftarrow TT_u + c_{(u,v)}(AT_u)$;
        $AT_v \leftarrow (t + TT_v) \bmod TD$;
        $LB_v \leftarrow TT_v + H(v)$;
        if $LB_v < Q_U[k]$ **then**
            if *v is not in Q* **then**
                if *v was not de-queued* **then**
                    En-queue (v, AT_v, TT_v, LB_v) in Q;
                    Mark v as en-queued;
                end
            else
                Update L_v, if it is necessary;
                Re-order Q;
            end
            $U_{UNN(v)} \leftarrow TT_v + UB(v, AT_v)$;
            if $UNN(v, AT_v)$ *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 $C_{NN}[1...k]$;

to service can also not be improved, so it is useless to re-expand v. Therefore, we can conclude that a vertex is never re-expanded and thus, the maximum number of expanded vertices in this

algorithm is $|V|$.

We also maintain a queue C_{NN} to store the POIs candidates to be returned. An entry in this queue is a tuple $(p_i, TS(q, p_i, t))$. When a POI p_i is de-queued from Q, it is inserted in C_{NN}, which is ordered by increasing order of $TS(q, p_i, t) = TT(q, p_i, t) + WT(p_i, ATp_i)$, that is the time to service. The algorithm stops when the next vertex u to be expanded has $LB_u > C_{NN}[k]$ (which is verified on line 10), i.e., it is not possible to find a POI in a path that passes by u which has a travel time and, consequently, a time to service shorter than the times to service of the POIs in C_{NN}, or when Q is empty.

The space complexity of the TD-kNN-OTC algorithm is $|Q| + |C_{NN}| + |V|$, which in the worst case is equal to $3|V|$.

3.4.2 Optimistic Solution

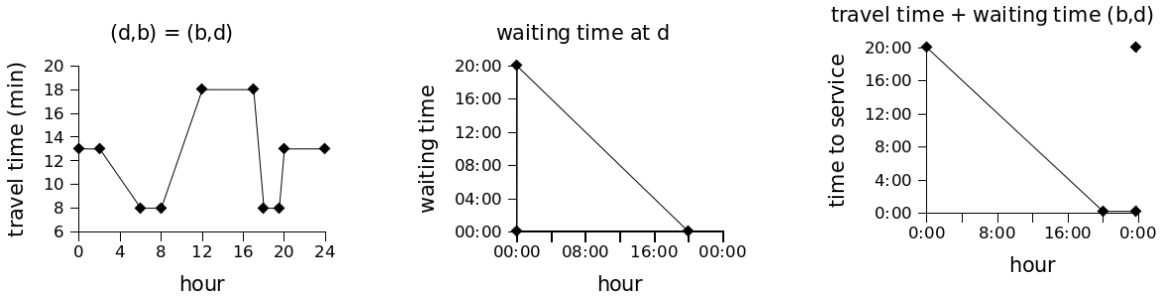
The heuristic function of the Naive solution does not take into consideration any waiting time. Its goal is to reach POIs the quickest. Then, after these POIs are reached, the waiting time until they open is calculated. This solution is not very efficient in the sense that it spends time searching for POIs that are close to q, but that may take a long time to open. Clearly, a heuristic that besides considering an estimate to reach POIs also takes into account the waiting time, can better guide the search for POIs where one can start to be served in less time.

Based on this, we propose an Optimistic solution in which the heuristic function, $H_O(u)$, is an optimistic estimate to the cost to reach a POI from u and start to be served. In addition to avoid expanding vertices that are far to any POI, this heuristic function also avoids expanding those that leads to POIs in which the optimistic time to service is longer than a set of candidates.

As in the Naive solution, the heuristic function values are calculated off-line during the pre-processing. In order to calculate these values, we construct a lower bound graph $\underline{G}_{TS}$ which is similar $\underline{G}$, but in addition to the travel time costs of each edge, that is given by the minimum possible cost to traverse this edge for any time in G, for each edge (u, p) where p is a POI, we assign a lower bound cost $L_{TS}(u, p)$ to the time to service of this edge. The value of $L_{TS}(u, p)$ is given by $L_{TS}(u, p) = min_{t \in TD}\{c_{(u,p)}(t) + WT(p, t + c_{(u,p)}(t))\}$.

To exemplify how the $L_{TS}(u, p)$ cost is calculated, let us consider again the TDG G shown in Figure 3.2a. Figure 3.4a represents the travel time costs of the edge (b, d). The graphic shown in Figure 3.4b shows the waiting time function of the POI d. For instance, if this POI is reached at 8 am, the waiting time is 12 hours. However, if it is reached from 20:00 to 0:00, there is no waiting. The graphic in Figure 3.4b is a combination of the travel time function of the edge (b, d) and the waiting time at d. More specifically, it is given by $c_{(b,d)}(t) + WT(d, t + c_{(b,d)}(t))$, where t is the departure time. For example, if one departs from b towards d at 18:00, the travel time is 8 minutes and thus, the arrival time at d is 18:08. The waiting time until d opens from 18:08 is 1:52. Therefore, the total time to service is 2 hours. Note that the sum of the travel time plus the waiting time is minimized from 20:00 to 23:47. For t = 20:00, we have $c_{(b,d)}$(20:00)

$+WT(d,20{:}00+c_{(b,d)}(20{:}00)) = 13+WT(d,20{:}13) = 13$, thus, $L_{TS}(u,p) = 13$. This is a lower bound for the time to start to be served at d when one departs from b at any time.



(a) Travel time costs of the edge (b,d).

(b) Waiting time function of the POI d.

(c) Time to service of the the POI d when one departs from b.

Figura 3.4: Graphic of time to service of the POI d when one departs from b constructed from the graphics of travel time of the edge (b,d) and of the waiting time at d.

The $\underline{G}_{TS}$ of G is shown in Figure 3.5. The cost of the edge (a,b), for example, is given by 8 as it is the minimum travel time from a to b for any time in G. On the other hand, since d is a POI, there are two associated costs to the edge (b,d), the cost to pass by d and the cost to stop at d and wait until it opens, considering optimistic estimates. Similarly, the minimum travel time from b to d is 8, but, as explained above, the minimum sum of the travel time between these two vertices plus the waiting time at d is 13 (represented by the edge (b,d')).

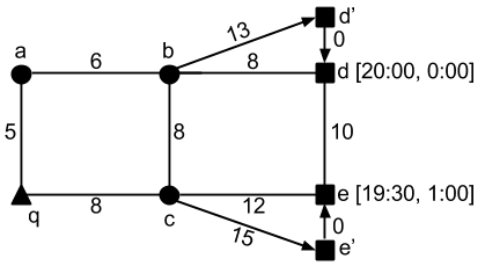


Figura 3.5: $\underline{G}_{TS}$ graph of the TDG shown in Figure 3.2a.

The optimistic heuristic function of a vertex u, $H_O(u)$, is then given by the minimum cost to reach a POI and start to be served in $\underline{G}_{TS}$. For example, the heuristic function value of the vertex a is equal to 19 (the cost of the path $\langle a,b,d'\rangle$) because this is the minimum time to service from it to its nearest neighbor, in terms of time to service, d. This cost indicates that the time it takes to reach a POI and start to be served from a is at least 19 minutes.

As in the Naive solution, we use the upper bound value given by $UB(u,ATu) = U(u,UNN(u))+WT(UNN(u),ATu+U(u,UNN(u)))$ as a pessimistic expectation for the time to service of the POI given by $UNN(u)$. It considers a pessimistic estimate for the time to reach the POI $UNN(u)$ from u to calculate an estimated arrival time at $UNN(u)$ and then, it computes a waiting time. The sum of the pessimistic travel time from u to $UNN(u)$ plus this waiting time is as an upper bound to time to service of $UNN(u)$. These bounds are used to prune vertices that lead to POIs that certainly have a time to service greater than a set of candidates.

3.4.2.1 Offline Pre-processing

The pre-processing of this solution also has two steps. In the first one we calculate the values of the heuristic function. For each vertex $v \in V$, we calculate the cost of the shortest path from v to its nearest neighbor in $\underline{G}_{TS}$. Differently from the previous solution, the heuristic function gives an optimistic estimation of the time to start to be served. In the second step, we calculate the cost of the shortest path from v to its nearest neighbor in $\overline{G}$, denoted by $UNN(v)$, that is used in the pruning of unpromising vertices.

In each one of these steps, we execute Dijkstra's algorithm (DIJKSTRA, 1959) from each vertex v to find its nearest neighbor. In the worst case, this algorithm runs in time $O(|E| + |V| \log |V|)$ for each execution. As we need to find the nearest neighbor from each vertex, the total complexity in the worst case is $O(|V||E| + |V|^2 \log |V|)$.

Note that the time of execution of the first step tends to be more costly than the second one, since the first POI found in the search is not necessarily the one that has the minimum time to service, and because of this, it is necessary to expand more vertices. Consequently, the pre-processing of the Optimistic solution is more costly than the one executed in the Naive solution.

It is also important to observe that, as in the previous solution, it is not necessary to start one search for each vertex in the network, since all the vertices in the path from a vertex v to its nearest neighbor p also have p as their nearest neighbor both in $\underline{G}_{TS}$ and $\overline{G}$, which improves the time of execution.

3.4.2.2 Query Processing

Let us use Algorithm 1 to show how the Optimistic Solution works. The aspects that differentiate this solution from the Naive one are the following:

- The queue Q is ordered by increasing order of $LBTSv_i = TTv_i + H_O(v_i)$, i.e., vertices which offer a greater chance to reach a POI and be served, considering an optimistic estimate, are expanded first;
- Regarding the pruning process, a vertice u can be discarded if $LBTS_u$ is greater than $Q_U[k]$, meaning that we already have k candidates that have, in the worst case, a time to service less than the optimistic time to service of a path that passes by u;
- The algorithm stops when the next vertex u to be expanded has $LBTS_u > C_{NN}[k]$, i.e., it is not possible to find a POI that has a time to service shorter than the candidates in C_{NN} as well as when Q is empty.

3.4.3 Bounded Solution

The heuristic function of the Optimistic solution is a very low estimate for the time to service from a vertex v since it is given by the minimum time to reach a POI from v and to

start to be served among all departure times. Particularly, it does not take into consideration the time when v is reached during the query processing in its calculation. This information in addition to an estimate time to achieve a POI from v, allows us to calculate an optimistic arrival time at this POI and a more precise waiting time, and so, better estimate the time to service from v.

Thus, we propose a Bounded solution in which the heuristic function is a tighter estimate to time to reach a POI from v *and* to start to be served. The heuristic function value of v is given by $H_B(v,ATv) = min_{p_j \in P}\{L(v,p_j) + WT(p_j, t + L(v,p_j))\}$, that is an optimistic expectation for the time to service of a path that connects v to the point of interest where the user can start to be served the quickest.

As an example, let us consider the TDG shown in Figure 3.2a. If one departs from q at t = 19:00, the travel time to the vertex b is 13 minutes, through the path $\langle q,a,b\rangle$, and the arrival time is at = 19:13. As shown in the lower bound graph $\underline{G}$ in Figure 3.3a, d is the nearest POI from b considering an optimistic travel time, and $L(b,d) = 8$. The time to service of d, departing from b is, thus, at least $L(b,d) + WT(d,\ 19{:}13{+}L(b,d)) = 8 + WT(d,19{:}21) = 47$. However, although e is the farthest POI from b in $\underline{G}$, there is a greater chance to start being served faster at this POI because $L(b,e) + WT(e,\ 19{:}13{+}L(e,d)) = 18 + WT(e,19{:}31) = 18$. Thus, the heuristic function value of b, when it is reached at 19:13, is $H_B(b,19{:}13) = 18$.

Note that it is not enough just to know the cost of the shortest path from a vertex to its nearest POI in $\underline{G}$ to calculate the heuristic function values. For instance, in the example presented above, if we only knew the cost of the shortest path from b to its nearest POI d, we could overestimate the cost to reach a POI from b and to start to be served. Thus, similarly to the pre-processing of the Naive solution, we construct the lower bound graph $\underline{G}$, but we need to compute more information than in that solution. More specifically, we pre-compute from each $v \in V$ the travel time of the fastest path from it to every $p_j \in P$ in $\underline{G}$. It is important to notice that this extra computation is justified by the fact that with these information we can calculate a tight lower bound to the time to service.

Similarly, we calculate upper bound values $UB_{TS}(v,ATv) = min_{p_j \in P}\{U(v,p_j) + WT(p_j, ATv + U(v,p_j))\}$, that unlike the heuristic function, consider the pessimistic travel time to POIs to calculate upper bounds to the time to service. We also keep the POIs corresponding to these upper bounds, denoted by $UNN_{TS}(v,AT_v)$. Thus, the upper bound of a vertex v indicates that is possible to reach the POI given by $UNN_{TS}(v,AT_v)$ and to start to be served in at most $UB_{TS}(v,ATv)$ units of time. These upper bounds are used to prune vertices that lead to POIs that have a time to service greater than a set of candidates.

Note that the values of the heuristic function $H_B(v,ATv)$, the upper bound $UB_{TS}(v,ATv)$ and $UNN_{TS}(v,AT_v)$ cannot be pre-computed, since they depend on the time when v is reached, thus, they are calculated during the query processing. However, the $L(v,p_i)$ and $U(v,p_i)$ values are calculated in the pre-processing step.

3.4.3.1 Offline Pre-processing

In the first step of this phase, we calculate lower bounds that assist the computation of heuristic function values during the query processing. For each vertex $v \in V$, the cost of the shortest path from v to every $p_j \in P$ in $\underline{G}$ is calculated. These costs are used to compute a tighter estimate to the time to service from v considering the time when v is reached. Similarly, in the second step it is calculated the cost of the shortest path from every $v \in V$ to each $p_j \in P$ in $\overline{G}$. These pessimistic travel time costs are used to compute upper bounds to the time to service that are used to prune vertices that certainly lead to POIs that have a longer time to service than a set of candidates.

Thus, our goal in the pre-processing is to calculate the cost of th shortest paths from each $v \in V$ to every $p_j \in P$ in $\underline{G}$ and $\overline{G}$. Since $|P| < |V|$, it is less costly to run Dijkstra's algorithm (DIJKSTRA, 1959) on the reverse graph of $\underline{G}$ or $\overline{G}$ from each $p_j \in P$ until reaching every $v \in V$ to calculate these costs. A reverse graph of G is a graph with the same set of vertices, but the edges are reversed, i.e., if G contains an edge (u,v) then the reverse of G contains an edge (v,u).

The total complexity of each step in the worst case is $O(|P|\,|E| + |P|\,|V|\log|V|)$. Note that, in fact, we can not expect the time of execution to be less than this complexity, since we have to calculate the cost of the shortest path from each POI for every vertex, thus expanding the whole network for each search.

3.4.3.2 Query Processing

The Bounded solution's query processing is very similar to the Optimistic solution, excepting from these two aspects:

- The lower bound value $LBTSv_i$ is given by $TTv_i + H_B(v_i, AT_{v_i})$, i.e. a sum of the travel time from q to v_i plus a tight estimate to the time to start to be served from v_i considering the time when it is reached;
- An entry in the queue Q_U is a tuple $(UNN_{TS}(v,AT_v), U_{UNN_{TS}}(v,AT_v))$, where $U_{UNN_{TS}}(v, AT_v) = TTv_i + UB_{TS}(v,AT_v)$, that is, the travel time from q to v plus the pessimistic estimate to the time to service of the POI given by $UNN_{TS}(v,AT_v)$. For each POI given by $UNN_{TS}(v,AT_v)$ we maintain the lowest upper bound value, $U_{UNN_{TS}}(v,AT_v)$, found so far in the expansion. This queue is ordered by increasing order of $U_{UNN_{TS}}(v,AT_v)$ values and is also used in the pruning process. More specifically, a vertice u can be discarded if $LBTS_u > Q_U[k]$, meaning that we already have k candidates that have, in the worst case, a time to service less than the estimated time to service of a path that passes by u

3.4.4 Updating the Network Information

Some applications require updates on the time-dependent road networks, which imply changing the pre-computed information. We consider four cases of updates: updates on

the travel time functions; deletion of a POI object; inclusion of a POI object and update on the operating time of a POI.

Update on the travel-time functions. When the value of the travel-time function changes, we have to run the pre-processing step again if the lower bound in $\underline{G}$ changes in the Naive and Bounded solutions or in $\underline{G}_{TS}$ in the Optimistic approach. Similarly, if the upper bound in $\overline{G}$ changes, we have to run the pre-processing again in all the solutions.

Deletion of a POI. When a POI is deleted from the network, we have to run the pre-processing step for all vertices that have the POI object removed as a nearest neighbor in $\underline{G}$ or $\overline{G}$ (in the Naive solution) or $\underline{G}_{TS}$ or $\overline{G}$ (in the Optimistic solution). However for the Bounded solution, we only need to delete the costs of the shortest paths from the all the vertices to this POI.

Inclusion of a POI. When a POI is included we have to run the pre-processing step again, since it is possible that the new POI becomes the nearest neighbor of any vertex in $\underline{G}$ or $\overline{G}$ (in the Naive solution) or $\underline{G}_{TS}$ or $\overline{G}$ (in the Optimistic solution). Concerning the Bounded solution, we have to compute the cost of the shortest path from every vertex to this POI, which can be done by running Dijkstra's algorithm on the reverse graph of $\underline{G}$ or $\overline{G}$ from the new POI until finding all the vertices.

Updating the operating time of a POI. Particularly in the Optimistic solution, we need to consider this case of update. If the operating time of a POI p is updated such that this implies the growth of the value of $L_{TS}(v, p)$ for any v, where $(v, p) \in E$, we have to run the pre-processing step for all vertices that have p as the nearest neighbor in $\underline{G}_{TS}$. On the other hand, if the value of $L_{TS}(v, p)$ decreases, we have to run the pre-processing step again.

3.4.5 Running Example

In order to exemplify the algorithms presented above, let us consider the graph instance in Figure 3.2a. The travel times of its edges are shown in Figure 3.2b. The input for the algorithm is the query point q and the departure time t = 19:00.

3.4.5.1 Naive Solution

The result of the pre-processing step of this solution is shown in Figure 3.6a. For example, the cost of the shortest path from a to its nearest POI in $\underline{G}$ is 14 and to its nearest POI in $\overline{G}$, d, is 28. Table 3.1 shows the entries stored in the queue Q after each expansion of a vertex in the Naive solution.

First, the vertex q is expanded and the following entries are inserted in Q, Q = ⟨(a, AT_a = 19:05, TT_a = 5, LB_a = 19), (c, AT_c = 19:10, TT_c = 10, LB_c = 22)⟩. As it is possible to reach d from a in at most 28 minutes, in the worst case d is reached at 7:33, but the waiting time at d from 7:33 is 27 minutes and the time to service is equal to 60 minutes, thus, the entry $(d, 60)$ is inserted in Q_U. Similarly, it is possible to reach e at most at 7:25 from c and as this POI is open at this time, there is no waiting, thus, the entry $(e, 25)$ is inserted in Q_U.

v	$H_N(v)$	UNN(v)	U(v,UNN(v))
a	14	d	28
b	8	d	18
c	12	e	15

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

v	$H_O(v)$
a	19
b	13
c	15

(b) Value of $H_O(v)$ function.

	d	e
a	14	24
b	8	18
c	16	12

	d	e
a	28	43
b	18	33
c	36	15

(c) Costs of the shortest paths from the vertices to every POI in $\underline{G}$ in the left table and in $\overline{G}$ in the right table.

Figura 3.6: Pre-processed information of the Naive, Optimistic and Bounded solutions, respectively.

The vertex a is expanded and since its adjacency vertex not yet en-queued is b, the new entry (b, AT_b = 19:13, TT_b = 13, LB_b = 21) is en-queued in Q. Next, the vertex b is expanded and the entry (d, AT_d = 19:21, TT_d = 21, LB_d = 21) is inserted in Q. The next vertex expanded is d and as it is a POI, the queue C_{NN} is initialized with the entry $(d,60)$, since the travel time to d is 21 and the waiting time from the moment when it is reached is 39 minutes. Its adjacency vertex not yet en-queued is e and the entry corresponding to it is (e, AT_e = 19:33, TT_e = 33, LB_e = 33). As $LB_e > Q_U[1] = 25$, this entry is discarded.

The only vertex in Q at this moment is c that is expanded generating the entry (e, AT_e = 19:25, TT_e = 25, LB_e = 25) which is inserted in Q. The vertex e is expanded and since it is a POI, the entry $(e,25)$ is en-queued in C_{NN}, since the travel time to e is 25 and there is no waiting from AT_e = 19:25. As there are no more vertices to be expanded in Q, the algorithm stops and the POI e is returned as the nearest neighbor.

step	Q
1	(a, AT_a = 19:05, TT_a = 5, LB_a = 19), (c, AT_c = 19:10, TT_c = 10, LB_c = 22)
2	(b, AT_b = 7:13, TT_b = 13, LB_b = 21), (c, AT_c = 19:10, TT_c = 10, LB_c = 22)
3	(d, AT_d = 7:21, TT_d = 21, LB_d = 21), (c, AT_c = 19:10, TT_c = 10, LB_c = 22)
4	(c, AT_c = 19:10, TT_c = 10, LB_c = 22)
5	(e, AT_e = 7:25, TT_e = 25, LB_e = 25)

Tabela 3.1: Entries stored in the queue Q in the Naive solution.

3.4.5.2 Optimistic Solution

The result of the first step of the pre-processing step, the heuristic function value $H_O(v)$, is shown in Figure 3.6b. As the second step is the same as the Naive solution, the values

of $UNN(v)$ and $U(v,UNN(v))$ are shown in Figure 3.6a. Table 3.2 shows the entries stored in the queue Q after each expansion of a vertex in this solution.

First, q is expanded generating the entries Q = $\langle$(a, AT_a = 19:05, TT_a = 5, $LBTS_a$ = 24), (c, AT_c = 19:10, TT_c = 10, $LBTS_c$ = 25)$\rangle$ which are en-queued in Q. As in the previous example, the entries $(b,25)$ and $(d,60)$ are inserted in Q_U. The next vertex expanded is a and the entry (b, AT_b = 7:13, TT_b = 13, $LBTS_b$ = 26) corresponding to its neighbor b is discarded, since $LBTS_b > Q_U[1] = 25$. The vertex c is expanded and the entry (e, AT_e = 7:25, TT_e = 25, $LBTS_e$ = 25) is en-queued in Q.

The vertex e is expanded and since it is a POI, the queue C_{NN} is initialized with the entry $(e,25)$, since the travel time to e is 25 and there is no waiting from the moment when e is reached. The adjacency vertex not yet en-queued of e is d and the entry to this vertex is (d, AT_d = 7:25, TT_d = 25, $LBTS_d$ = 25), which is inserted in Q. As $LBTS_d > C_{NN}[1] = 25$ the algorithm stops, since it is not possible to find any other POI with a shorter time to service than the ones in C_{NN}. The vertex e (the first vertex in C_{NN}) is returned as the nearest neighbor.

step	Q
1	(a, AT_a = 19:05, TT_a = 5, $LBTS_a$ = 24), (c, AT_c = 19:10, TT_c = 10, $LBTS_c$ = 25)
2	(c, AT_c = 19:10, TT_c = 10, $LBTS_c$ = 25)
3	(e, AT_e = 7:25, TT_e = 25, $LBTS_e$ = 25)
4	(d, AT_d = 7:25, TT_d = 25, $LBTS_d$ = 25)

Tabela 3.2: Entries stored in the queue Q in the Optimistic solution.

3.4.5.3 Bounded Solution

The result of the pre-processing step is shown in Figure 3.6c, note that we just show the costs that we use in the example. The left and right tables show, from each vertex $v \in V$, the shortest path to every *POI* $p \in P$ in $\underline{G}$ and in $\overline{G}$, respectively. Table 3.3 shows the entries stored in the queue Q after each expansion of a vertex in this solution.

The vertex q is expanded and the entries $\langle$(a, AT_a = 19:05, TT_a = 5, $LBTS_a$ = 29), (c, AT_c = 19:10, TT_c = 10, $LBTS_c$ = 25)$\rangle$ are en-queued in Q. Even though d has more potential to be the nearest POI from a when only the travel time is taken into account, the minimum estimated time to reach d is at 7:19 and the waiting time until this POI opens is 41 minutes, giving a time to service equals to 60 minutes. On the other hand, the POI e is reached from a at least at 7:29, but the waiting time from 7:29 at e is 0, thus $LBTS_a$ is set to 29. The entry $\langle$(e, 25)$\rangle$ is inserted in Q_U because $TT(q,c,t)+U(c,e) = 25$ and the time that takes to b opens from 7:25 is 0.

The vertex c is expanded and since its adjacency vertices not yet en-queued are b and e, the new entries are (b, AT_b = 7:20, TT_b = 20, $LBTS_b$ = 40) and (e, AT_e = 7:25, TT_e = 25, $LBTS_e$ = 25). As $LBTS_b > Q_U[1] = 25$, the entry corresponding to b is not en-queued. The vertex e is expanded and as it is a POI, the queue C_{NN} is initialized with the entry $(e,25)$, since the travel time to e is 25 and there is no waiting from the moment when e is reached. The next

vertex to be expanded is a,but as $LBTS_a > C_{NN}[1]$, the algorithm stops and e is returned as the nearest neighbor.

step	Q
1	(c, AT_c = 19:10, TT_c = 10, $LBTS_c$ = 25), (a, AT_a = 19:05, TT_a = 5, $LBTS_a$ = 29)
2	(e, AT_e = 7:25, TT_e = 25, $LBTS_e$ = 25), (a, AT_a = 19:05, TT_a = 5, $LBTS_a$ = 29)
3	(a, AT_a = 19:05, TT_a = 5, $LBTS_a$ = 29)

Tabela 3.3: Entries stored in the queue Q in the Bounded solution.

3.4.6 Correctness of the solutions

The three approaches return exactly the same k POIs where the user can be served the quickest. They only differ in the moment when these POIs are inserted in C_{NN}. Based on this, the following proof of correctness is applicable to all three solutions we proposed above.

Lemma 3.4.1. *The POIs corresponding to $C_{NN}[1...k]$ returned by TD-kNN-OTC(q, t) have a shorter time to service from q and departure time t than the other remaining POIs.*

Proof. We need to show that: (1) when a POI r is de-queued from Q and then en-queued in C_{NN} it has the value of the quickest time to service from q to r and (2) there is no POI, excluding the ones in $C_{NN}[1...k-1]$, that has a time to service shorter than $p = C_{NN}[k]$.

(1) It is enough to prove that the sooner a POI r is reached, the shorter is its time to service. As the algorithm is a shortest-path algorithm, it is trivial to note that the POIs are reached by the fastest path from q to it. Let tt be the cost of the fastest path from q to r and let us suppose by contradiction that r is reached at a time tt^* greater than tt, such that $tt^* = tt + a$, where $a > 0$, but the time to service of r at the arrival time $t + tt^*$ is the quickest one. So, $tt^* + WT(r, t + tt^*) < tt + WT(r, t + tt)$, $tt + a + WT(r, t + tt + a) < tt + WT(r, t + tt)$, $a + WT(r, t + tt + a) < WT(r, t + tt)$ (3). For the inequality (3) be true, we necessarily have $WT(r, t + tt + a) < WT(r, t + tt)$. The only case in which one arrives later at a POI and wait for a shorter amount of time is when the POI is not open yet. Thus, as for that case the waiting time is given by the opening time of the POI minus the arrival time, by (3) we have $a + OT_r - (t + tt + a) < OT_r - (t + tt)$, $a + OT_r - t - tt - a < OT_r - t - tt$ and then $OT_r < OT_r$, a contradiction. Thereby, we can conclude that the time to service of a POI when it is not reached by the fastest path is never less than when the POI is reached by the fastest path.

(2) Let us suppose by contradiction, that there is a POI p^* such that $TS(q, p^*, t) < TS(q, p, t)$. We need to consider two cases: p^* is in C_{NN} and p^* is not in C_{NN}. Regarding the first case, as C_{NN} is ordered by increasing order of time to service and as $TS(q, p^*, t) < TS(q, p, t)$, p^* has to be in a position lower than p's position in C_{NN}. As there are $k-1$ POIs that have a time to service shorter than $TS(q, p, t)$, if p^* is in C_{NN}, p should thus appear in the position $k+1$ in C_{NN}, contradicting the fact that p is the kth element in this queue. If p^* is not in C_{NN}, it means that p^* was not de-queued from Q. Let us

consider u the last vertex de-queued from Q, thus $LB_u > TS(q,p,t)$, as $LB_{p^*} \geq LB_u$, $LB_{p^*} > TS(q,p,t)$ and $TS(q,p^*,t) > TS(q,p,t)$, contradicting the initial hypothesis that $TS(q,p^*,t) < TS(q,p,t)$.

□

3.5 Experiments

We compared the three proposed solutions according to the number of I/O operations to access the disk pages at both graph-level and data-level and according to the pre-processing cost. We set the page size of the data structures to 4KB to store the B-tree nodes as well as the data relating to the data-level. In order to know the total number of disk access for each solution without any external influence, we performed the experiments without caching.

We generated synthetic time-dependent road networks with the granularity of 15 minutes during a day. A network is generated as a grid where each point corresponds to a vertex. Each vertex has a uniformly distributed number of neighbors from one to four, which are chosen between its adjacent vertices, thus, the average degree of the vertices is 2,5. The POIs are uniformly distributed over the network and each one has an uniformly distributed length of opening time with average values shown in Table 3.4. To generate the edges cost, for each edge, we chose a random speed between 30 km/h and 80 km/h for each interval of the day, so that the time cost given by the ratio between the edge length and this speed satisfies the FIFO property.

Density of POIs	0.1%, **1%**, 5%, 10%
Network Size	2000, **4000**, 8000, 15000, 30000, 50000
Length of opening time	4, **8**, 12, 24
k	1, **3**, 5, 10

Tabela 3.4: Parameters values of experiments.

We evaluated how the approaches perform with respect to four variables that are shown in Table 3.4, the network size (number of vertices), the density of POIs, the average length of the opening time of the POIs and k (the number of POIs to be found). For each experiment, we varied a parameter and set the others parameters to default values (in bold) and for each set of different parameters, we generated 10 distinct time-dependent road networks and executed 10 queries randomly selected for each network. As only the network size and the density of POIs affect the cost of the pre-processing of all solutions, we investigate how these costs are influenced by these variables.

Effect of the average length of opening time. As shown in Figure 3.7, the number of data pages accessed decreases as the average length of opening time increases for all the solutions. This happens because it is not necessary to travel for a long time until reaching a POI where the user does not have to wait for a long time, consequently fewer vertices need to be expanded, and ,thus, less data pages are accessed.

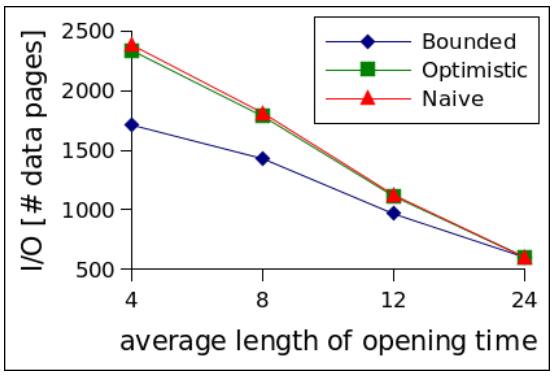


Figura 3.7: Number of I/Os when the average length of opening time increases.

The bounded solution outperforms the other ones because the waiting time, which is used in the calculation of the estimate for the time to be served, is calculated considering the estimated time to reach a POI. This estimate is closer to the real time to service than the ones used in the other two solutions. Basically, this happens because the other solutions are not able to measure how long is the waiting time. The shorter the length of opening times of the POIs, the longer the waiting time and the greater the efficiency of this solution, since its power of pruning vertices increases. The number of data pages accessed in the Naive solution is up to 40% greater than the number of pages accessed in this solution.

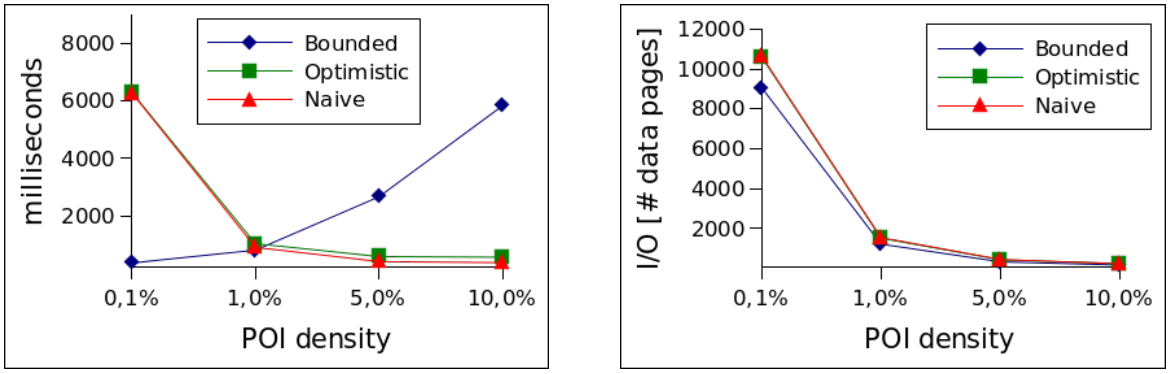
The performances of the Naive and the Optimistic solutions are virtually the same. The addition of an optimistic waiting time does not increase significantly the power of pruning vertices of the Optimistic solution compared to the Naive. This can also be noticed in the following experiments.

Note that the gap between the performances of the solutions decreases as the average length of opening time increases because the heuristic function values calculated in each solution approach each other. In the extreme case where the POIs are open for 24 hours, all the algorithms did access the same number of data pages, since as there is no waiting, the heuristic function value is the same for the three approaches.

Effect of the density of POIs. As shown in Figure 3.8b, when the POIs become denser, the number of data pages accessed decreases since it takes less effort to find the answer. In other words, the probability of finding a POI in which the user can be served quickly is higher simply because there are more choices of POIs. The number of data pages accessed in the Naive solution is from 17.4 % to 35.8% greater than the number of pages accessed in this solution.

The Bounded solution also outperforms the other ones. As the estimate of the time to be served used in this approach is closer to the real time to service, it expands first the vertices that offer a greater chance to be in the path to a POI where the user can be served the quickest. The quality of the heuristic used in this solution is not sensitive to this variable.

The performances of the Naive solution and the Optimistic solution are also virtually the same in this experiment. In the Naive solution, the quality of the heuristic function decreases and the number of false hits increases with the POI density, as more POIs can be reached quickly, but some of them are not suitable. In the Optimistic solution, some vertices will



(a) (b)

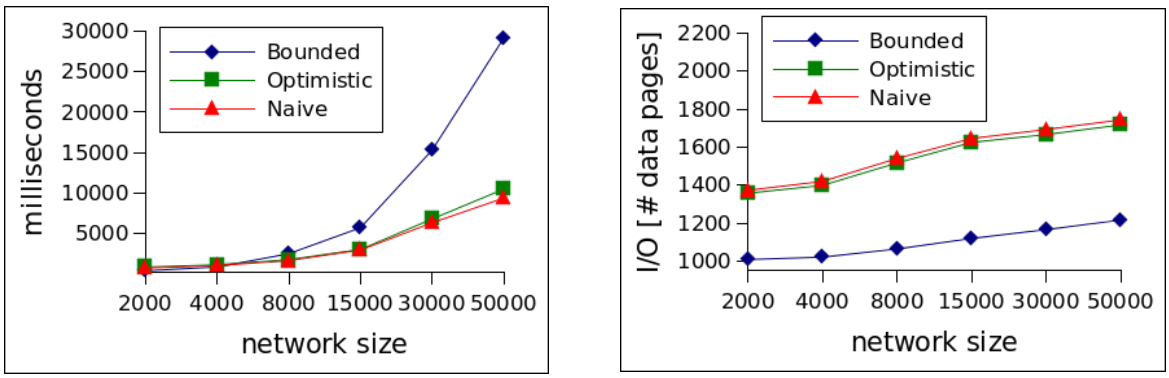
Figura 3.8: Pre-processing cost and number of I/Os when the density of POIs increases.

be expanded by offering a low value for the estimate of being in a path to a POI where the user can be served quickly, but, in fact, this estimate is far from the real time to service. Therefore, the quality of the heuristic function also decreases and the number of false hits increases with the POI density in the Optimistic solution.

As shown in Figure 3.8a, the pre-processing time for the Bounded solution increases with the POI density, while it decreases in the other solutions. This happens because, the whole network is expanded $|P|$ times in this solution, while in the other ones, at most $|V|$ NN searches are performed and the cost of them decrease with the POI density. Note that, in the case where the POI density is 0.1% (for $|V| = 4000$), the pre-processing of the Bounded solution is less costly than the other ones, because it is faster to expand 4000 vertices 4 times than finding the NNs of those vertices.

Effect of the network size. Figure 3.9b shows that the number of data pages accessed in all the solutions increases with the network size. This is reasonable since there are more paths to a POI in a larger network, thus, more paths need to be investigated until finding the shortest one.

This experiment also indicates that the Bounded solution outperforms the other two.



(a) (b)

Figura 3.9: Pre-processing cost and number of I/Os when the network size increases.

As the Naive solution is only interested in reaching a POI the quickest, it can waste time expanding vertices that lead to POIs which will take a long time to open. Thereby, as it is more difficult to find a POI in a network with more vertices, this solution expands more vertices until finding the POIs that are part of the answer and, thus, it accesses more data pages.

The same reasoning is used to explain the behavior of the Optimistic solution. As the estimate of the time to be served is optimistic, this solution expands first some vertices that, in fact, do not lead to POIs where the user can be served the quickest. Therefore, as the difficult of finding a POI increases with the number of vertices, this solution expands more vertices and, consequently, accesses more data pages.

Figure 3.9a shows that the pre-processing time of all the solutions increase with network size, however, the pre-processing cost of the Bounded solution increases faster than the others. In the Optimistic and Bounded solutions more NN searches have to be executed and the cost of such searches increases with the network size, since there are more paths to be investigated. This happens because more vertices have to be expanded as well as more edges have to be traversed in every search starting from a POI. In the Bounded solution more searches have to executed since the number of POIs increases with the network size as well as the number of vertices that have to be reached in each search.

Effect of k. Figure 3.10 shows that the number of data pages accessed increases as k increases, as expected. More vertices have to be expanded to find more points of interest.

The Naive solution expands first the vertices that offer a greater chance to reach a POI in less time, regarding any possible waiting time. The number of false hits tend to grow with k, since more suitable POIs need to be found. Thus, it expands more vertices until finding k POIs where the user can be served quickly.

The same reasoning is used to explain the behavior of the Optimistic solution . As the estimate for the time to be served in this solution is optimistic and the number of expanded vertices increases with k, this solution tends to expand more vertices in which the estimated time for being served is short, but, in fact, is far from the real time to service until finding the answer.

The graphic shows that the Bounded solution also outperforms the other two appro-

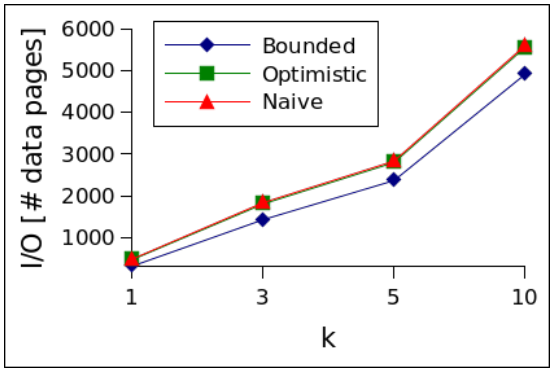


Figura 3.10: Number of I/Os when k increases.

aches. As a tighter estimate for the time to service is used, the number of false hits is smaller than in the other solutions. Thus, fewer vertices need to be expanded to find k POIs in which a user can be served the quickest. The number of data pages accessed in the Naive solution is from 14 % to 54% greater than the number of pages accessed in this solution.

3.6 Conclusion

In this chapter we discussed the problem of processing TD-kNN-OTC queries in time-dependent road networks. This query aims at finding the k points of interest in which a user can start to be served in the minimum amount of time, accounting for both the travel time to the point of interest and the waiting time, if it is closed, named . We discussed some previously proposed solutions to the problems of processing regular kNN queries in static and in time-dependent road networks, the main problems related to the problem we are addressing, and we argued why they are not suitable to solve the query we are interested in.

We proposed and proved the correctness of three approaches to solve TD-kNN-OTC queries, named Naive, Optimistic and Bounded. All of them are based on an incremental network expansion and use the A* search to guide this expansion. Each solution uses a different heuristic function, which in combination with a pruning process determines its efficiency in terms of query processing.

We presented an experimental evaluation which showed that the number of data pages accessed in the Naive solution is up to 54% greater than the number of pages accessed in the Bounded solution. Even though this solution outperforms the other two consistently, it is the most expensive in terms of pre-processing and may not be suitable if the TDG is updated frequently. In such cases one may want to use the Optimistic one.

4 OPTIMAL SEQUENCED ROUTE QUERIES

4.1 Introduction

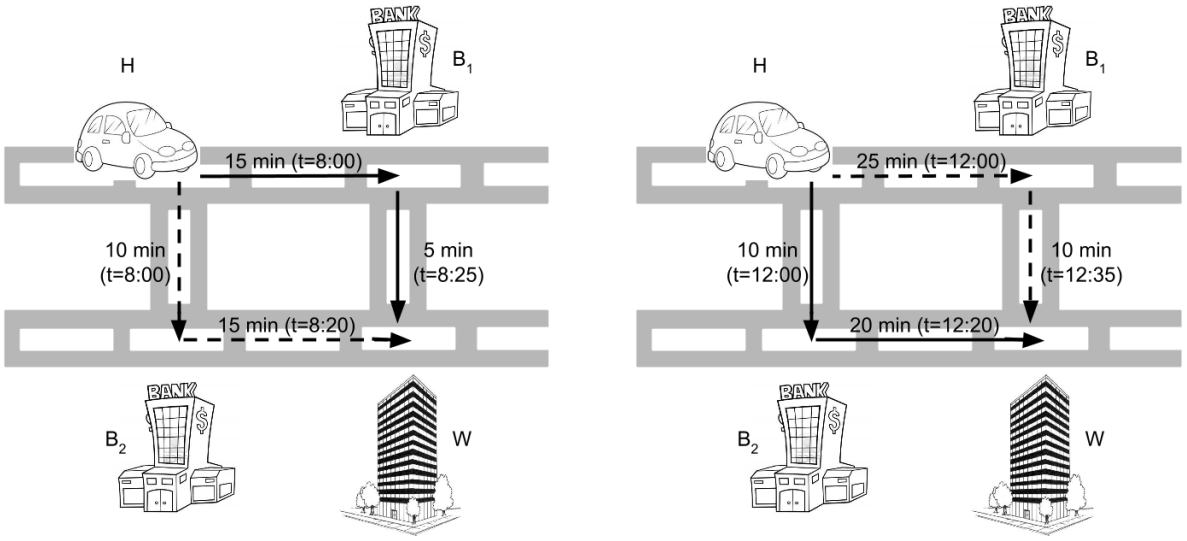
The optimal sequenced route (OSR) query was introduced by (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008). It has been used for several trip planning applications, such as location-based services (LBS) and car navigation systems. This query aims at finding the optimal route from a origin location to a destination passing through a number of points of interest (POIs) in a specific sequence imposed on the categories of the POIs. For instance, suppose that we are planning the following trip in the city: first we intend to leave home towards a bank to withdraw money and then we plan to visit a mall to buy clothes before returning home. The constraint that enforces the order in this example is that there is no reason to go to the mall without money for shopping. Although there may be many banks and malls in the city, the OSR query chooses the bank and the mall, in this order, that minimize the total cost of the trip.

Some improved solutions to the OSR query were proposed in (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008; OHSAWA et al., 2012; HTOO et al., 2012; EISNER; FUNKE, 2012). The query addressed in these works does not take into consideration that, typically, the time a moving object takes to traverse a segment depends on the departure time. Differently from previous works, in this chapter we study the problem of processing OSR queries in time-dependent road networks. In such networks, a OSR query returns the route with minimum travel-time from the query point to the destination considering a certain departure time.

As we are working with a time-dependent network, we need to know in advance how long the user expects to stay in each POI because the time he/she leaves the POI affect the result, since the travel time depends on the departure time.

In order to illustrate our query let us consider the following scenario. Suppose that one is leaving home towards a bank to withdraw money and expects to spend 10 minutes there before going to work. We need to find the location of the bank which driving toward it shortens the trip (in terms of time) from home to work considering the time that he/she leaves home. Let us consider two banks in the city, B_1 and B_2 as shown in Figure 4.1. At 8 am, the best route is $\langle H, B_1, W \rangle$. It takes 15 minutes to go from the user location (denoted as H) to the bank B_1 and to go from there to his/her work from the moment when the user leaves the bank (at 8:25) it takes 5 minutes, thus the total time of the route $\langle H, B_1, W \rangle$ is 20. On the other hand, the route $\langle H, B_2, W \rangle$ takes 25 minutes. At 12 pm, the route with minimum travel time is no longer $\langle H, B_1, W \rangle$ which has cost 35, but $\langle H, B_2, W \rangle$ with cost equal to 30 minutes. Clearly, the route with minimum travel time depends on the departure time.

A greedy approach to solve the OSR problem is to perform independent NN searches locally. More specifically, we first locate the nearest POI from the query location, p_1^i, belonging to the first category in the sequence, then we locate the nearest POI from p_1^i belonging to the second category in the sequence, and so on, until the destination is reached. As discussed in (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008; SHARIFZADEH;



(a) Travel times considering that the user leaves home at 8 am.

(b) Travel times considering that the user leaves home at 12 pm.

Figura 4.1: The OSR (in solid line) in two different moments of a day.

SHAHABI, 2008), the OSR problem can not be optimally solved by using this approach and this clearly also applies for the time-dependent problem. Returning to Figure 4.1a, at 8 am, a greedy approach would return $\langle B_2, W \rangle$ as the best route since B_2 is the closest bank from the user's home at this time, but as we saw in the example above, this is not the route with minimum travel time.

In this chapter we propose an optimal solution based on the incremental network expansion (INE) (PAPADIAS et al., 2003) and the A* search algorithms (HART; NILSSON; RAPHAEL, 1968) to process TD-OSR queries as well as a suitable heuristic function to be used in our A* search. For comparison purposes, we also present a solution which is obtained by extending the progressive neighbor exploration (PNE c.f. Section 4.2) (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008) to cope with the time-dependent problem.

This chapter is structured as follows. In Section 4.2 we present a brief discussion of related works. We formalize the TD-OSR problem resorting to the definitions given in Chapter 2 in Section 4.3. In Section S4.4, we explain our proposed approaches. The experimental evaluation and results are shown in Section 4.5. Finally, Section 4.6 concludes this chapter with a summary of our findings.

4.2 Related Work

In (BéRUBé; POTVIN; VAUCHER, 2006) it was proposed a travel planning problem which consists in, finding the best travel plan from a origin to a destination that follows a given sequence of nodes considering a transportation network with deterministic time-dependent travel times. The authors proposed a decomposition scheme in which the whole

problem is divided into sub-problems and each one of them is solved as a one-to-many shortest path problem by adding a surrogate node to the graph. That work is focused on finding a path that passes through a predetermined sequence of nodes. On the other hand, our query chooses one POI from certain categories according to the specified sequence in order to minimize the total travel-time of the route.

The Trip Planning Query (TPQ) was proposed in (LI et al., 2005). This query is similar to our query of interest, but the order in which the user wants to visit the categories of POIs is not given. As this problem is NP-hard, because of the existence of multiple possibilities of ordering the categories of POIs, the authors proposed a number of approximation algorithms. We, on the other hand, propose both optimal and heuristic solutions.

A solution to the OSR query in vector and metric spaces was first proposed in (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008) and was extended by the authors in (SHARIFZADEH; SHAHABI, 2008). For vector spaces, (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008) proposed a light threshold-based iterative algorithm named LORD, that utilizes various thresholds to filter out the locations that cannot be in the optimal route. They also proposed R-LORD, an extension of LORD which uses R-tree to examine the threshold values more efficiently. For metric spaces, they proposed an approach that progressively apply *NN* queries on different point types to construct the optimal route for the OSR query. In (SHARIFZADEH; SHAHABI, 2008) the authors proposed an approach which is applicable for both vector and metric space. They exploited the geometric properties of the solution space and theoretically proved its relation to additively weighted Voronoi diagrams which are recursively accessed to incrementally build the OSR.

More recent studies have addressed the OSR problem in road networks (OHSAWA et al., 2012; HTOO et al., 2012; EISNER; FUNKE, 2012). The difference between those works and the previous ones is that a destination is given as an input to the query which aims at finding the shortest route from the current position with stops at one of each specified POI category from the visiting sequence before reaching the final destination.

In (OHSAWA et al., 2012) the authors proposed a solution that is based on an Incremental Euclidean Restriction (IER) (PAPADIAS et al., 2003). It first finds the shortest OSR given by searches in the Euclidean space and then verifies its length in the road network and this value is used as an upper bound. All OSRs that have a length less than this value also have the potential to be the shortest route in the road network, therefore all these OSRs must be searched in the Euclidean space, and then the results must be verified in the road network. The shortest of them is returned as the result. In (HTOO et al., 2012) the algorithms USVPG and BSVPG that performs an unidirectional and a bidirectional search, respectively, were proposed. Both of them are controlled by an A* algorithm and use the Euclidean distance between a vertex and the destination as the heuristic function. The authors also proposed a visited POI graph (VPG), in order to reduce multiple node expansions.

Two speed-up techniques were presented in (EISNER; FUNKE, 2012). The first, named Iterative Doubling, is an improvement of the EDJ solution (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008). The OSR is found by performing l Dijkstra runs on the graph,

where l is the number of categories of POIs that must be found. Making use of the fact that sequenced route queries tend to be mostly local, this solution is improved by avoiding exploring facilities that are too far. The second solution is based on an extension of the contraction hierarchy (GEISBERGER et al., 2008), that was originally proposed as a pre-processing step for ordinary shortest path queries.

Another query related to the one addressed in this chapter, named multi-rule partial sequenced route (MRPSR) query, was proposed in (CHEN et al., 2011). As an example, suppose that one is planning to visit a bank, a restaurant and a gas station, but he/she imposes the following restriction: visit a bank to withdraw money before having lunch at a restaurant. Note that there is no constraint that determines when the gas station must be visited, thus, it can be visited in any order. This query generalizes the OSR and the TPQ queries. The restrictions imposed by the user are formulated into rules. When the set of partial sequence rules is empty this query is equal to the TPQ query and when the set of partial sequence rules contains one partial sequence rule specifying the same order given by the user, this problem is identical to the OSR problem.

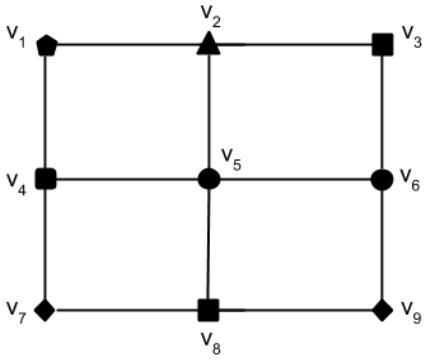
As far as we know there is no published research addressing the problem of OSR queries in time-dependent networks. Furthermore we present both optimal and heuristic solutions to the problem.

4.3 Problem Definition

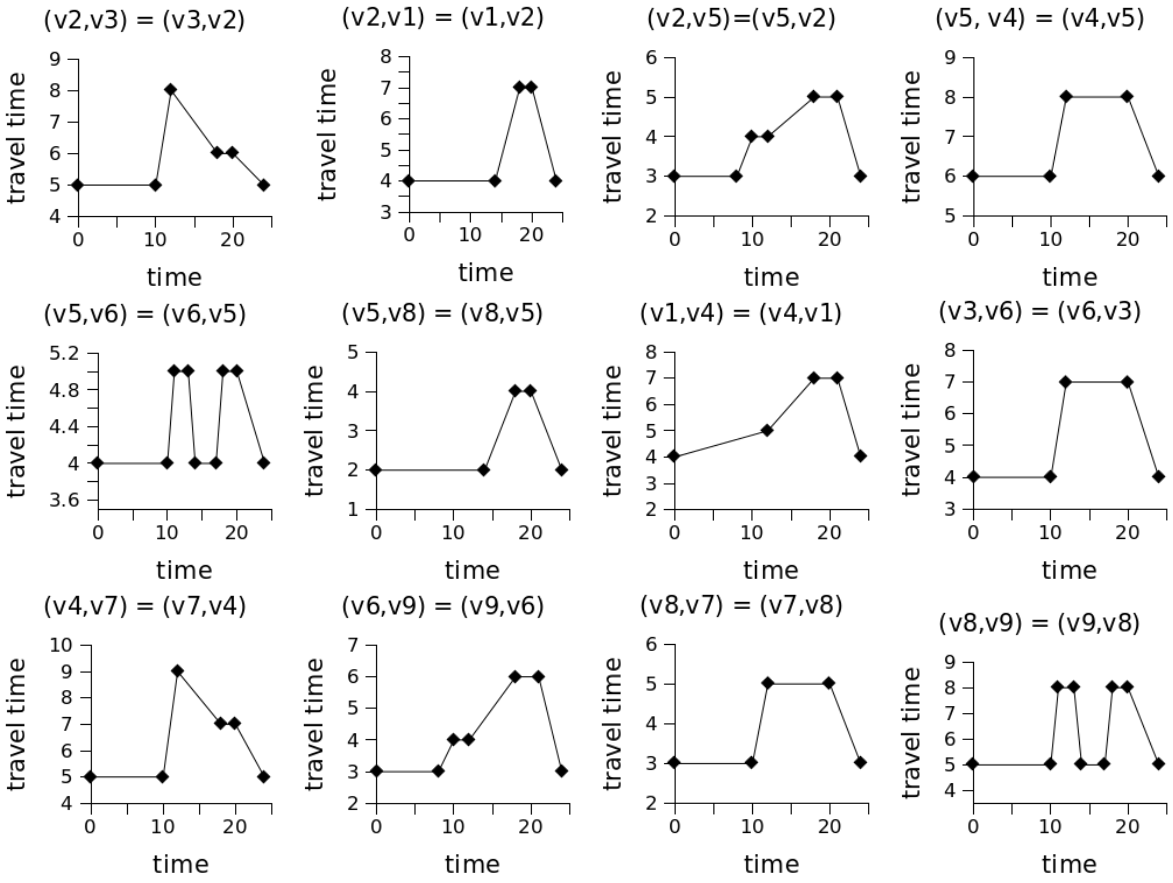
We consider the problem of processing *OSR* queries in road networks where the travel time is time-dependent, defined as follows:

Problem Definition 2. *Let G = (V,E,C) be a TDG and* $U = \{U_1, U_2, ..., U_k\}$ *a set of categories of POIs with* $U_i \subset V$. *Given a query point* $q \in V$, *a departure time t, a visiting order of POI category sets* $M = [(M_1, \tau_{M_1}), (M_2, \tau_{M_2}), ..., (M_m, \tau_{M_m})]$ *and a final trip destination* $d \in V$, *the TD-OSR (q, t, M, d) query finds the route with minimum travel time starting from q and the departure time t, selecting one POI according to the visiting sequence from each* U_{M_i}, *considering that the user stays there* τ_{M_i} *units of time, and finally arriving at d.*

In order to exemplify the definition presented above, let us consider the graph shown in Figure 4.2a, which is a representation of a time-dependent network. The graphics shown in 4.2b represent the travel time costs of each edge of the network. This network contains three sets of categories of POIs: restaurant (U_R), bank (U_B) and gas station (U_G). A sequence given as the query input contains any subset of these categories. For example, if the sequence $M = [(B, 15), (R, 60)]$ is given, it means that, on his way from q to d, the user wants to go to a bank and spend 15 minutes there and then go to a restaurant and stay 60 minutes.



(a) A network with three different categories of POIs: bank (v_3 and v_8), restaurant (v_7 and v_9) and gas station (v_1); and a query point $q = v_2$ represented by a triangle.



(b) Cost Functions

Figura 4.2: Network and travel time functions

4.4 Proposed Approaches

We propose two solutions to solve the TD-OSR problem optimally. For comparison purposes, we first present a baseline solution which is based on the progressive neighbor exploration (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008) and uses the TD-NE-A* (CRUZ; NASCIMENTO; MACÊDO, 2012) algorithm to perform the necessary time-dependent

NN local searches. Next, we propose an improved solution which performs an Incremental Network Expansion (INE) (PAPADIAS et al., 2003) and uses an A* (HART; NILSSON; RAPHAEL, 1968) search to guide this expansion. We also propose a scheme to reduce the number of node re-expansions.

Aiming to reduce the number of expanded vertices and thus, accelerate the query processing, an off-line pre-processing step is performed in both solutions in order to calculate bounds to guide the expansion of vertices. In the baseline solution, the TD-NE-A* (CRUZ; NASCIMENTO; MACÊDO, 2012) algorithm pre-calculate lower bounds to reach *any* POI from every vertex in the network to guide the expansion. As the NN searches performed in the PNE algorithm looks for a POI that belongs to a certain category, we modified the original TD-NE-A* algorithm so that, in the pre-processing step, we calculate estimates to reach every category of POI from each $v \in V$. Thus, a NN search that, for example, looks for a restaurant, uses as heuristic function the estimated cost to reach a restaurant.

In the second solution we use the potential of a vertex of being part of the optimal route to guide the expansion. Vertices that have a greater potential are expanded first. To measure the potential of a vertex v, we calculate an estimated cost to reach the categories of POIs belonging to sequence given as input and to reach the destination from v. As the cost to reach those POIs is time-dependent, we do not know in advance this cost for a certain departure time, but we can pre-calculate an underestimated value that is suitable for any departure time. Moreover, before receiving the query, we do not know which categories of POIs the user is interested in visiting. So, as in the Baseline solution, we need to pre-calculate cost estimates to reach every category of POIs from each vertex in the network. Note that these cost estimates are independent of the sequence and the departure time given as input.

In order to guide the search towards the destination, both solutions also need an estimated cost to reach it. In the baseline solution, the NN search that starts from a POI that belongs to the last category in the sequence looks for the destination. Thus, an estimated cost to reach the destination is required to guide such searches. Similarly, in the second solution, this estimate is used to calculate the potential of a vertex. The calculation of an estimate to reach the destination is a more difficult task because it can be any vertex of the network, while the POIs are fixed. To calculate the shortest path from every vertex to every vertex is impractical in terms of both space and execution time. This calculation is then postponed to after we receive the query, that is when we know the destination. We discuss how this estimate is calculated and how the pre-processing is performed below.

4.4.1 Off-line Pre-processing

We need to perform a pre-processing step in order to calculate from every $v \in V$ a cost estimate to reach the categories of POIs. As explained before, these values need to be independent of a departure time. Thus, the cost of the shortest path at any departure time from a vertex to a POI belonging to a certain category can not be overestimated. In order to calculate these costs, we construct a lower bound graph $\underline{G}$, defined in Chapter 3. Figure 4.3 shows the $\underline{G}$ graph of the TDG G shown in Figure 4.2a.

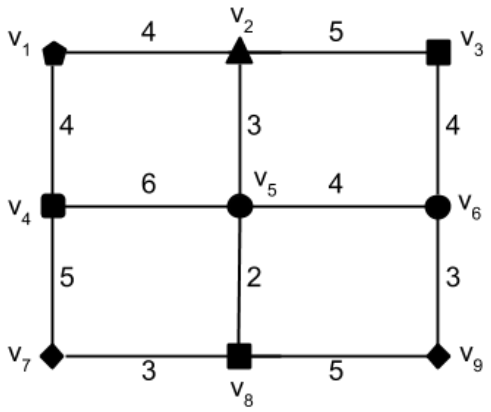


Figura 4.3: The lower bound graph of the TDG shown in 4.2a.

We denote $L(v,U_i)$ as the cost of the fastest path from v to its nearest POI belonging to the category U_i in $\underline{G}$. Once $\underline{G}$ is constructed, we pre-compute, from every $v \in V$ and for every $U_i \in U$, the cost of $L(v,U_i)$. By doing this, we have an optimistic expectation for the time to reach each category of POIs in the network from every vertex. To exemplify how this costs are computed, let us consider the $\underline{G}$ shown in Figure 4.3. For instance, the closest bank from the vertex v_5 is v_8 and the cost to reach it in $\underline{G}$ is 2. Similarly, the closest restaurant from v_5 is v_7 and the cost to reach this restaurant is 5. This indicates that, for example, the cost to reach a restaurant from v_5 is at least 5 units of time for any departure time.

The result of the pre-processing step for every v in G is shown in the Table 4.1. Regarding the time complexity to compute these cost estimates, we run Dijkstra's algorithm (DIJKSTRA, 1959) from each vertex v to find its nearest POI of each category in $\underline{G}$. In the worst case, this algorithm runs in time $O(|E|+|V|\log|V|)$ for each execution. As we start a search from each vertex, the total complexity in the worst case is $O(|V||E|+|V|^2\log|V|)$. The space required to store all the information calculated in the pre-processing step is linear to the number of vertices, more specifically, it is equal to $|U||V|$, where $|U|$ is the number of categories of POIs.

v	Bank	Restaurant	Gas Station
v_1	9	9	0
v_3	0	7	9
v_4	8	5	4
v_5	2	5	7
v_6	4	3	11
v_7	3	0	9
v_8	0	3	9
v_9	5	0	14

Tabela 4.1: Cost of the shortest path from each vertex to its nearest bank, restaurant and gas station, respectively.

4.4.2 Estimate of the cost to reach the destination

Once we know the destination d, before processing the query, we compute the shortest path from every vertex $v \in V$ to the destination by running the Dijkstra algorithm from d on the reverse graph of $\underline{G}$ until reaching all the vertices. A reverse graph of G is a graph with the same set of vertices, but the edges are reversed, i.e., if G contains an edge (u,v) then the reverse of G contains an edge (v,u). The result of this computation for the $\underline{G}$ in Figure 4.3 and $d = v_4$ is shown in Table 4.2. For instance, the cost to reach the destination v_4 from v_1 is at least 4 units of time. As just one search is required, the cost of this step is $O(|E| + |V| \log |V|)$.

v	v_1	v_3	v_4	v_5	v_6	v_7	v_8	v_9
L(v_i,d)	4	13	0	6	10	5	8	13

Tabela 4.2: Cost of the shortest path from each vertex to the destination in $\underline{G}$.

4.4.3 TD-PNE

We use an extension of the PNE (SHARIFZADEH; KOLAHDOUZAN; SHAHABI, 2008) algorithm, named TD-PNE, to address the OSR in time-dependent networks as a baseline solution. The original PNE algorithm generates the optimal route from the starting to the ending point progressively by performing local NN searches in road networks with static costs. It stores partial routes candidates to be expanded in a heap.

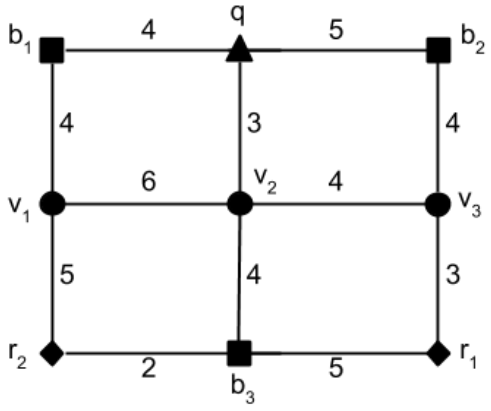


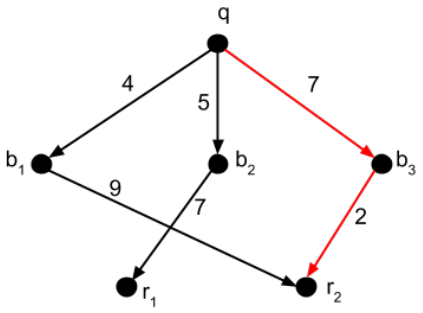
Figura 4.4: A static network with three banks, b_1, b_2 and b_3; two restaurants, r_1 and r_2; and a query point q, represented by a triangle.

To exemplify how the original PNE algorithm works, let us consider the static network shown in Figure 4.4. Let us suppose that one departs from the vertice q and wants to visit a bank and a restaurant in this order. Table 4.5a depicts the values stored in the heap in each step of the algorithm. The PNE algorithm first looks for the nearest bank from q, b_1, and stores this bank and the cost to reach it in the heap. Next, it expands the entry with minimum cost in the heap. The only entry at this moment is (b_1: 4). It looks for the second nearest bank from q and for the nearest restaurant from b_1. It finds b_1 with cost 5 and r_2 with cost 9. The

entries (b_2: 5) and (b_1, r_2: 13) are inserted in the heap. Note that a complete route $\langle b_1, r_2 \rangle$ was found. The cost of this route is then used as an upper bound, i.e, routes with cost greater than 13 are not investigated. The next entry expanded is (b_2: 5). PNE looks for the third nearest bank from q and for the nearest restaurant from b_2. The bank b_3 is found with cost 7 from q. The nearest restaurant from b_2 is r_1 and the cost to reach it is 7. Thus, the entries (b_3: 7) and (b_2, r_1: 12) are inserted in the heap. The upper bound is then updated to 12 because a shorter route has been found. Next, it looks for the fourth nearest bank from q and the nearest restaurant from b_3. As there is no more banks in the network and r_2 is the nearest restaurant from b_3, only the entry (b_3, r_2: 9) is inserted in the queue. The route $\langle b_3, r_2 \rangle$ becomes the shortest route. As the next entry in the heap has cost greater than 9, the cost of the shortest route, the algorithm stops and returns $\langle q, b_3, r_2 \rangle$ as the optimal route.

step	heap contents
1	(b_1: 4)
2	(b_2: 5), (b_1, r_2: 13)
3	(b_3: 7), (b_2, r_1: 12)
4	(b_3, r_2: 9)

(a) Entries stored in the heap during the execution of PNE algorithm.



(b) The POIs found during the execution of the PNE algorithm. The optimal route is highlighted in red.

Figura 4.5: Entries stored in the heap and the POIs found during the execution of PNE algorithm.

We modified this algorithm such that each NN search is substituted by a TD-NN search to progressively generate candidates routes. More specifically, the TD-NE-A* (CRUZ; NASCIMENTO; MACÊDO, 2012) algorithm is executed in order to find the NN from the POIs in the sequence considering the departure from there. Furthermore, we add another layer to the graph shown in Figure 4.5b. This new layer represents the destination. For instance, a TD-NN search that departs from a restaurant looks for the destination. We named the time-dependent variation of the PNE algorithm as TD-PNE.

We also modified the TD-NE-A* algorithm to make it more efficient to solve our problem of interest. As each NN search in the PNE algorithm looks for a POI belonging to a specific category U_i, we modified the heuristic function used in the original TD-NE-A* algorithm. Originally, this function was an estimate to the time to reach *any* POI from a vertex. We modified it such that now it is given by the cost of the shortest path between a vertex and its nearest POI of the category U_i in $\underline{G}$. By doing this, instead of expand first the vertices that offer a greater chance to be in a path to *any* POI, we first examine those that lead to POIs which belong to U_i.

Moreover, although each TD-NN search in the TD-PNE algorithm looks for just *one* POI, we do not know in advance how many nearest POIs from a certain origin and departure time we need to find because the PNE algorithm is progressive. For instance, in the example

presented above, when we started the search that found b_1 as the nearest bank from q, we did not know that would be required to look for 2 more nearest banks from q. On the other hand, this number is the parameter k given as input to the original TD-NE-A* algorithm. Thus, we need to modify it such that the searches are performed incrementally on demand. Note that, if, for example, we have already found the nearest POI of the category U_{M_1} from q at the departure time t and the TD-PNE algorithm asks for the second nearest POI of this category, we do not need to start another search from q at t. For each POI (or origin) from where a search was started, we maintain a queue Q which stores vertices candidates to be expanded in the next step. Thus, if the ith nearest POI of the category U_{M_1} from q at t is requested, we just continue the search that found the $i-1$ nearest POIs.

4.4.4 TD-OSR

In this section we present a solution, named TD-OSR, which is based on an incremental network expansion (INE) (PAPADIAS et al., 2003) and uses an A* search (HART; NILSSON; RAPHAEL, 1968) to guide this expansion, i.e., to determine the order in which vertices are expanded in the search tree. Vertices with more potential are examined first. To measure the potential of a vertex v, it uses the current distance from the query point q to v given by $d(q,v)$ plus a heuristic function, which in our case is an estimate to the time to reach the categories of POIs belonging to the sequence and the destination. The smaller the value given by the sum of the current distance to a vertex plus its heuristic function value, the greater its potential.

In order to calculate heuristic function values, we need cost estimates to reach the categories of POIs in the graph and the destination. The first is computed in the pre-processing step and is independent of the sequence given as input. The second is calculated on-line, after receiving the query, but before processing it.

Once we have the estimates to reach the categories of POIs and to reach the destination computed, we can calculate the heuristic function. Given the sequence $M = (M_1, M_2, ..., M_m)$ and considering that a path to a vertex v has already passed by POIs of the categories $M_1..M_i$, the heuristic function value for v is given by

$$H(v, M[i+1...m], d) = max\{L(v, M_{i+1}), ..., L(v, M_m), L(v, d)\} \tag{4.1}$$

where $L(v, M_i)$ is the cost of the shortest from v to its nearest POI of the category M_i in $\underline{G}$ and $L(v,d)$ is the estimate to reach d from v. In other words, the cost to pass through the categories of POIs in $M = (M_1, M_2, ..., M_m)$ and to reach the destination is at least the minimum cost to reach the farthest of them in $\underline{G}$. For example, according to Tables 4.1 and 4.2, $H(v_5, M[B,R], v_4) = max\{L(v,B), L(v,R), L(v,v_4)\} = max\{2,5,6\} = 6$, meaning that the cost to reach a bank, a restaurant and the destination v_4 from v_5 is at least 6 units of time. Assuming that the path of from the query point q to v_5 have already visited a bank, the heuristic function value of the vertex v_5 in this situation is given by $H(v_5, M[R], v_4) = max\{L(v,R), L(v,v_4)\} = max\{5,6\} = 6$.

Algorithm 2: TD-OSR

Data: A query point $q \in V$, the departure time $t \in [0,T]$, a sequence $M = [M_1,...,M_m]$ and a destination $d \in V$
Result: The minimum route from q to d considering the time t and selecting one POI from each category in M according to the visiting sequence

1 $\underline{G}^R \leftarrow loadReverse\underline{G}$;
2 $destination[] \leftarrow Dijkstra(\underline{G}^R, d)$;
3 $TT_q \leftarrow 0$;
4 $AT_q \leftarrow t$;
5 $LB_q \leftarrow H(q, M[1...m], d)$;
6 $R_q \leftarrow []$;
7 En-queue $(q, AT_q, TT_q, LB_q, R_q)$ in Q;
8 **while** $Q \neq \emptyset$ **do**
9 $(u, AT_u, TT_u, LB_u, R_u) \leftarrow$ De-queue Q;
10 $Removed[|R_u|] \leftarrow u$;
11 **if** $u = d$ *and* $|R_u| = |M|$ **then**
12 Return R_u;
13 **end**
14 **for** $v \in adjacency(u)$ **do**
15 $next \leftarrow |R_u| + 1$;
16 $R_v \leftarrow R_u$;
17 $spent \leftarrow 0$;
18 **if** $v \in M[next]$ **then**
19 $spent \leftarrow \tau_{M[next]}$;
20 $R_v[next] \leftarrow v$;
21 $next \leftarrow next + 1$;
22 **end**
23 $TT_v \leftarrow TT_u + c_{(u,v)}(AT_u)$;
24 $AT_v \leftarrow (t + TT_v + spent) \bmod TD$;
25 $LB_v \leftarrow TT_v + H(v, M[next...m], d)$;
26 **if** *v was not removed in a position* $\geq |R_v|$ **then**
27 **if** *v is not in Q* **then**
28 En-queue $(v, AT_v, TT_v, LB_v, R_v)$ in Q;
29 **else**
30 length $\leftarrow \left|R_{v_{inQ}}\right|$;
31 lb $\leftarrow LB_{v_{inQ}}$;
32 **if** *length* $\leq |R_v|$ *and lb* $\geq LB_v$ **then**
33 Update $(v, AT_v, TT_v, LB_v, R_v)$ in Q;
34 **end**
35 **end**
36 **end**
37 **end**
38 **end**

4.4.4.1 Query Processing

Algorithm 2 formalizes our solution to the TD-OSR problem. It takes as input the query point $q \in V$, the departure time $t \in [0,T]$, a sequence $M = [M_1,...,M_m]$, where $|M| = m$,

and the destination $d \in V$.

First, the algorithm loads the reverse graph of $\underline{G}$, $\underline{G}^R$, from the disk. Then, the Dijkstra algorithm is called to calculated the costs of the shortest paths from d to every $v \in V$ in $\underline{G}^R$. These costs, which are estimates to reach the destination, are stored in the vector $destination$. This initialization step is shown in the lines 1 and 2.

Next, the algorithm begins the expansion and inserts q in a priority queue Q (line 7) that stores the set of candidates for expansion in the next step. An entry in queue Q is a tuple $(v_i, ATv_i, TTv_i, LBv_i, Rv_i)$, where $ATv_i = AT(q, v_i, t)$, $TTv_i = TT(q, v_i, t)$, LBv_i is given by $TTv_i + H(v_i, M[i...m])$ and Rv_i stores the *POIs* that have already been reached in the path from q to v_i according to the visiting order. The priority of elements in Q is given by the increasing order of LBv_i values with the purpose of checking first the vertices that offer a greater chance to reach the *POIs* in M and the destination quickly.

The vertices are dequeued from Q (line 9) and expanded. When a vertex u is dequeued from Q it is marked as removed (line 10) in the *Removed* list corresponding to the number of POIs that have already been reached in the path to it. For example, the $Removed[1]$ list contains all the vertices that have been removed from Q and the path from q to them passes through 1 POI according to the given sequence M. As a path to a vertex might contain from 0 to m POIs, there are exactly $m+1$ *Removed* lists.

For every v neighbor of u we check if it is a POI and if it belongs to next category in the sequence. If this condition is satisfied (which is verified on line 16), v is inserted in the next position of R_v and the number of *POIs* that belong to the sequence found in the path to v is incremented. Furthermore, we add the time the user wants to spend at v to the arrival time at the neighbors of this vertex (line 22).

For each v, we check if it is in any of $Removed[|R_v|]...Removed[m]$ (line 26). If it is in any of these lists, it is not beneficial to us inserting v in Q, since it has been already found in a path that includes more POIs and has a lower LB_v, we prove that it is valid in Lemma 4.4.1. If it is not in any of these queues neither in Q, which is verified in line 27, it is inserted in Q. If it is in Q, we check if the new path to it includes more *POIs* in sequence than in the old one in Q and if LB_v (of the new entry) is less or equal to the one in Q (line 32). If these conditions are satisfied, the old entry of v is removed from Q and the new one is inserted. By doing this, we avoid re-expand vertices unnecessarily.

The algorithm stops when the next vertex expanded is the destination d and the path from q to d, R_d, includes all the categories of *POIs* in M according to the visiting sequence as shown on lines 11 and 12.

Lemma 4.4.1. *It is not beneficial to insert a vertex v in Q to be expanded if it is found by a path that includes i POIs belonging to the sequence and v is in any of* $Removed[i]...Removed[m]$.

Proof. Let us suppose that v has been found by a path A that includes i POIs belonging to categories in the sequence. First, consider that v is in $Removed[i]$. Thus, v has already been found by a path B that includes i POIs. Moreover, as the queue Q is ordered by increasing values of LBv_i, $LB_v^B < LB_v^A$. Therefore, $TT(B,t) + H(M[i+1...m]) < TT(A,t) + H(M[i+1...m])$

and thus, $TT(B,t) < TT(A,t)$, meaning that v has already been found by a shortest path B. Consequently, expanding this vertex again is not beneficial because it can not improve the cost of a route that passes by v due the FIFO nature of the network.

Now, consider that v is in any of $Removed[k]$ for $i+1 \leq k \leq m$. Without loss of generality, there is a path C from q to v that includes k POIs that follow the sequence and $LB_v^C < LB_v^A$. Thus, $TT(C,t) + H(M[k...m]) < TT(A,t) + H(M[i+1...m])$, as $H(M[k...m]) \leq H(M[i+1...m])$ for $i+1 \leq k \leq m$, implying $TT(C,t) < TT(A,t)$. This means that v has already been found by a shortest path C that includes more than i POIs and clearly is not beneficial expanding v in this situation. □

As a consequence of the Lemma 4.4.1, the following property holds.

Lemma 4.4.2. *A vertex is expanded at most $m+1$ times in the TD-OSR algorithm.*

Proof. By contradiction, let us suppose that a vertex v was expanded more than $m+1$ times. As we know, there are exactly $m+1$ *Removed* lists. Thus, if a vertex was expanded more than $m+1$ times, it was inserted more than once in at least one of the *Removed* lists. According to Lemma 4.4.1, this is a contradiction since a vertex is never reinserted in the same list. □

4.4.5 Running Example

In order to exemplify our solutions, let us consider the graph instance in Figure 4.2a. The input for the algorithm is the query point $q = v_2$, the departure time t = 18:00, the sequence $M = [(B,15),(R,60)]$ and the destination v_4. The result of the pre-processing in shown in Table 4.1 and the estimated costs to reach the destination from every vertex are shown in Table 4.2.

4.4.5.1 TD-PNE

Table 4.3 depicts the values stored in the heap in each step of the algorithm. Each entry contains the travel time to each POI as well as the arrival time there. For example, in step 1, the entry $(v_3 : 6,$ 18:06$)$ indicates that the bank v_3 has been reached with cost 6 at 18:06.

step	heap contents
1	$(v_3 : 6,$ 18:06$)$
2	$(v_8 : 9,$ 18:09$)$, $(v_3, v_9 : 19,$ 18:34$)$
3	$(v_8, v_7 : 14,$ 18:29$)$, $(v_3, v_9 : 19,$ 18:34$)$
4	$(v_8, v_9 : 17,$ 18:32$)$, $(v_3, v_9 : 19,$ 18:34$)$, $(v_8, v_7, v_4 : 21,$ 19:36$)$
5	$(v_3, v_9 : 19,$ 18:34$)$, $(v_8, v_7, v_4 : 21,$ 19:36$)$
6	$(v_8, v_7, v_4 : 21,$ 19:36$)$

Tabela 4.3: PNE for the running example

In the first step, the algorithm looks for the nearest bank from v_2 at t =18:00. The bank v_3 is found with cost 6, by expanding the vertices v_2 and v_3, and the entry $(v_3 : 6,$ 18:06$)$

is inserted in the heap. In the second step, the entry $(v_3 : 6, 18{:}06)$ is removed from the heap and the algorithm looks for the second nearest bank from v_2 at 18:00 and the nearest restaurant from v_3 at 18:21, since v_3 was reached at 18:06 and the user intends to spend 15 minutes there. Note that, it is not necessary to start a new search from v_2 to find the second nearest bank from it, we just continue the search that found v_3. The bank v_8 is found with cost 9, by expanding the vertices v_5 and v_8, and the entry $(v_8 : 9, 18{:}09)$ is inserted in the heap. The nearest restaurant from v_3, v_9, is found with cost 10 and the entry $(v_3, v_9 : 19, 18{:}34)$ is inserted in the heap.

The entry $(v_8 : 9, 18{:}09)$ is removed and the algorithm looks for the third nearest bank from v_2 at 18:00 and for the nearest restaurant from v_8 at 18:24. As there is not a third bank in the network, the remaining vertices in the network that were not expanded in the search that started from v_2 are expanded now and no bank is found. The nearest restaurant from v_8, v_7, is found with cost 5, by expanding the vertices v_8 and v_7, and the entry $(v_8, v_7 : 14, 18{:}29)$ is inserted in the heap. The entry $(v_8, v_7 : 14, 18{:}29)$ is removed. The second nearest restaurant from v_8 at 18:24, v_9, is found with cost 8 and the entry $(v_8, v_9 : 17, 18{:}32)$ is inserted in the heap. The destination v_4 is found from v_7 with cost 7 and the entry $(v_8, v_7, v_4 : 21, 19{:}36)$ is inserted in the heap. Note that now we have an upper bound to the cost of the entire route, which is equal to 21 minutes.

The entry $(v_8, v_9 : 17, 18{:}32)$ is removed and the algorithm looks for the third nearest restaurant from v_8 at 18:24 and for the destination from v_9 at 19:32. As there is not a third restaurant in the network, the remaining vertices in the network that were not expanded in the search that started from v_8 are expanded now and no restaurant is found. The destination v_4 is reached with cost 19 from v_9 and the entry $(v_8, v_9, v_4 : 36, 19{:}51)$ is discard since the total cost of this route is greater than the upper bound already found. Next, the entry $(v_3, v_9 : 19, 18{:}34)$ is removed. The cost of the path from v_3 to its second nearest restaurant at 18:21, v_7, is 22 (through the path v_3, v_2, v_5, v_8, v_7). The entry $(v_3, v_7 : 28, 18{:}43)$ is not inserted in the queue, since the cost of this route is greater than the upper bound. The cost of the shortest path from v_9 to v_4 is equal to 19 and the entry $(v_3, v_9, v_4 : 38, 19{:}53)$ is also discarded. Finally, the entry $(v_8, v_7, v_4 : 21, 19{:}36)$ is removed from the heap and is returned as the best route.

4.4.5.2 TD-OSR

Table 4.4 shows the entries stored in the queue Q after each expansion of a vertex. The algorithm first expands the vertex v_2 and the entries corresponding to its neighbors v_5, v_3 and v_1 are inserted in Q as shown in step 1. Because, for example, the cost to reach a bank, a restaurant and the destination from the vertex v_5 is at least 6, which is the heuristic function value of this vertex. The vertex v_5 is expanded, as Q is ordered by increasing value of LB_{v_i}, generating the entries of the vertices v_8, v_6 and v_4 which are inserted in v_2. The next vertex expanded is v_1, generating the entry (v_4, AT_{v_4} = 18:13, TT_{v_4} = 13, LB_{v_4} = 21, [ ]) which is discarded because v_4 has already been found by a shorter path.

The vertex v_8 is expanded and the arrival times in its neighbors are increased by 15 minutes, since this is the time that the user wants to spend at a bank. The entries corresponding to v_7 and v_9 are inserted in Q (step 4). The next vertex expanded is v_7 and the arrival time in its

step	queue contents
1	(v_5, AT_{v_5}=18:05, TT_{v_5}=5, LB_{v_5}=11, [ ]), (v_1, AT_{v_1}=18:07, TT_{v_1}=7, LB_{v_1}=16, [ ]), (v_3, AT_{v_3}=18:06, TT_{v_3}=6, LB_{v_3}=19, [v_3])
2	(v_1, AT_{v_1}=18:07, TT_{v_1}=7, LB_{v_1}=16, [ ]), (v_8, AT_{v_8}=18:09, TT_{v_8}=9, LB_{v_8}=17, [v_8]), (v_3, AT_{v_3}=18:06, TT_{v_3}=6, LB_{v_3}=19, [v_3]), (v_6, AT_{v_6}=18:10, TT_{v_6}=10, LB_{v_6}=20, [ ]), (v_4, AT_{v_4}=18:13, TT_{v_4}=13, LB_{v_4}=21, [ ])
3	(v_8, AT_{v_8}=18:09, TT_{v_8}=9, LB_{v_8}=17, [v_8]), (v_3, AT_{v_3}=18:06, TT_{v_3}=6, LB_{v_3}=19, [v_3]), (v_6, AT_{v_6}=18:10, TT_{v_6}=10, LB_{v_6}=20, [ ]), (v_4, AT_{v_4}=18:13, TT_{v_4}=13, LB_{v_4}=21, [ ])
4	(v_7, AT_{v_7}=18:29, TT_{v_7}=14, LB_{v_7}=19, [v_8, v_7]), (v_3, AT_{v_3}=18:06, TT_{v_3}=6, LB_{v_3}=19, [v_3]), (d, AT_d=18:10, TT_d=10, LB_d=20, [ ]), (v_4, AT_{v_4}=18:13, TT_{v_4}=13, LB_{v_4}=21, [ ]), (v_9, AT_{v_9}=18:32, TT_{v_9}=17, LB_{v_9}=30, [v_8, v_9])
5	(v_3, AT_{v_3}=18:06, TT_{v_3}=6, LB_{v_3}=19, [v_3]), (v_6, AT_{v_6}=18:10, TT_{v_6}=10, LB_{v_6}=20, [ ]), (v_4, AT_{v_4} = 19:36, TT_{v_4} = 21, LB_{v_4} = 21, [v_8, v_7, v_4]), (v_9, AT_{v_9}=18:32, TT_{v_9}=17, LB_{v_9}=30, [v_8, v_9])
6	(v_6, AT_{v_6}=18:10, TT_{v_6}=10, LB_{v_6}=20, [ ]), (v_4, AT_{v_4} = 19:36, TT_{v_4} = 21, LB_{v_4} = 21, [v_8, v_7, v_4]), (v_9, AT_{v_9}=18:32, TT_{v_9}=17, LB_{v_9}=30, [v_8, v_9])
7	(v_4, AT_{v_4} = 19:36, TT_{v_4} = 21, LB_{v_4} = 21, [v_8, v_7, v_4]), (v_9, AT_{v_9}=18:32, TT_{v_9}=17, LB_{v_9}=30, [v_8, v_9])

Tabela 4.4: Entries stored in the queue Q.

neighbor v_4 is increased by 60 minutes, since v_7 is a restaurant. The entry corresponding to v_4 is updated to (v_4, AT_{v_4} = 19:36, TT_{v_4} = 21, LB_{v_4} = 21, [v_8, v_7, v_4]) because even if the old entry had the same value as a lower bound, in the new one, v_4 has been found in a path which passes either by a bank and by a restaurant. The vertex v_3 is expanded and as its neighbor d is found by a longer path, TT_{v_6}=13, the entry corresponding to v_6 in Q is not updated.

The next vertex expanded is v_6, reaching the vertices v_3 and v_9. The entries (v_3, AT_{v_3} = 18:17, TT_{v_3} = 17, LB_{v_3} = 30, [v_3]) and (v_9, AT_{v_9} = 18:18, TT_{v_9} = 18, LB_{v_9} = 31, [ ]) are discarded because the vertices v_3 and v_9 have already been found by a shorter path. Finally, the vertex v_4 is expanded, as it is the destination and the path to it has passed by both a bank and restaurant, the algorithm stops and returns the route $\langle v_2, v_8, v_7, v_4 \rangle$ with cost equal to 21 minutes.

4.5 Experiments

We compared the proposed solutions according to the number of expanded vertices in the search and the processing time of the query. In order to know how longer is the travel time of a route returned by a greedy solution in relation to the optimal solution, we compared the total travel time returned by both of them.

We generated synthetic time-dependent road networks with the granularity of 60 minutes during a day. A network was generated as a grid where each point corresponds to a vertex. The POIs were uniformly distributed over the network and each category has, in average, the same number of POIs. To generate the edges cost, for each edge, we chose a random speed between 30 km/h and 80 km/h for each interval of the day, so that the time cost given by the

ratio between the edge length and this speed satisfies the FIFO property.

Network Size	25000, **50000**, 100000
Density of POIs	0.5%, **1%**, 5%
Vertex degree	2, **2.5**, 3
Number of categories	5, **10**, 20
Sequence Size	1, **3**, 10
Query Locality	5%, **15%**, 50%

Tabela 4.5: Parameters values of experiments.

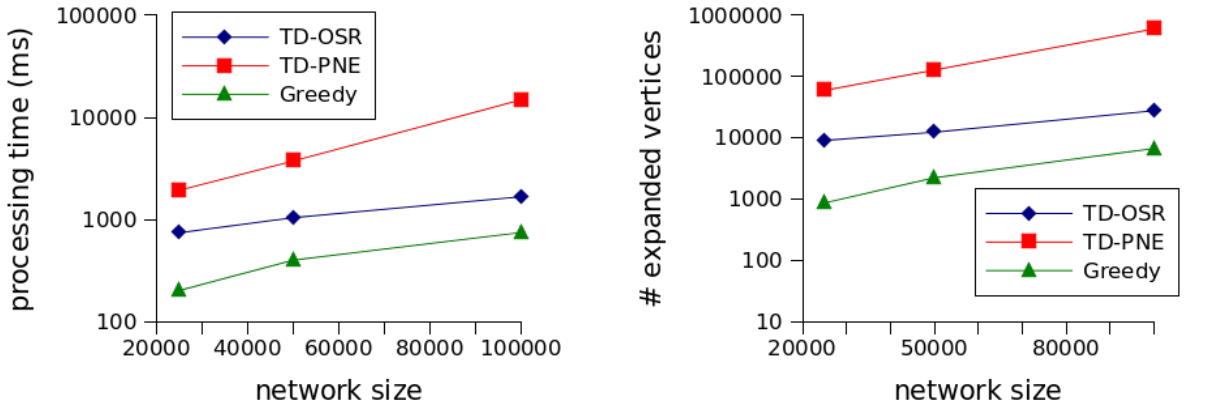
We evaluated how the approaches perform with respect to the variables shown in Table 4.5 related to the network and to the query. Regarding the network, we varied its size, the density of POIs, the vertices degree and the number of categories of POIs in the network. Furthermore, we investigated how the algorithms behave in relation to the size of the sequence given as input and the distance between the query and the destination vertices. This distance was varied according to the values shown in the line query locality, which are a percentage of the network diameter. For each experiment, we varied a parameter and set the others parameters to default values (in bold). For each set of different parameters, we generated one time-dependent road network and executed 10 queries randomly selected for each network.

Effect of the network size. The graphics in Figure 4.6, on logarithmic scale, show how the solutions behave according to the network size. Both the number of vertices expanded and the processing time increase with the network size in all solutions. This can be explained by the existence of a greater number of paths that can lead to a POI in a larger network, making it difficult to find POIs and thus a route. It is important to notice that the TD-PNE and the Greedy solutions are more affected by this variable than the TD-OSR solution. This happens because the TD-NN searches, performed in both solutions, become more costly with an increase in the number of vertices in the network.

As noticed in this experiment and also in the following ones, the greedy solution outperforms the other ones in terms of processing time and number of vertices expanded simply because it makes no effort to find the route of minimum travel-time. On the other hand, the total cost of a route returned by this solution is, in average, from 6.6% to 27.3% greater than an optimal route.

The figure also shows that, as expected and evidenced in the following experiments, the TD-OSR solution outperforms the TD-PNE. The TD-PNE solution investigates all the possible routes which has a cost within the cost of the shortest route found so far. This algorithm is not very efficient in the sense that the routes are investigated in order of proximity to the origin with no estimate for the cost of reaching the destination. Thereby, routes that actually have a high cost, are investigated for offering a false chance to be the optimal route. On the other hand, in the TD-OSR solution, vertices that offer a greater chance of being in the optimal route, which takes into account an estimated cost to reach the POIs in the sequence that have not yet been found and the destination, are expanded first.

Effect of the density of POIs. As shown in Figure 4.7, the processing time of the

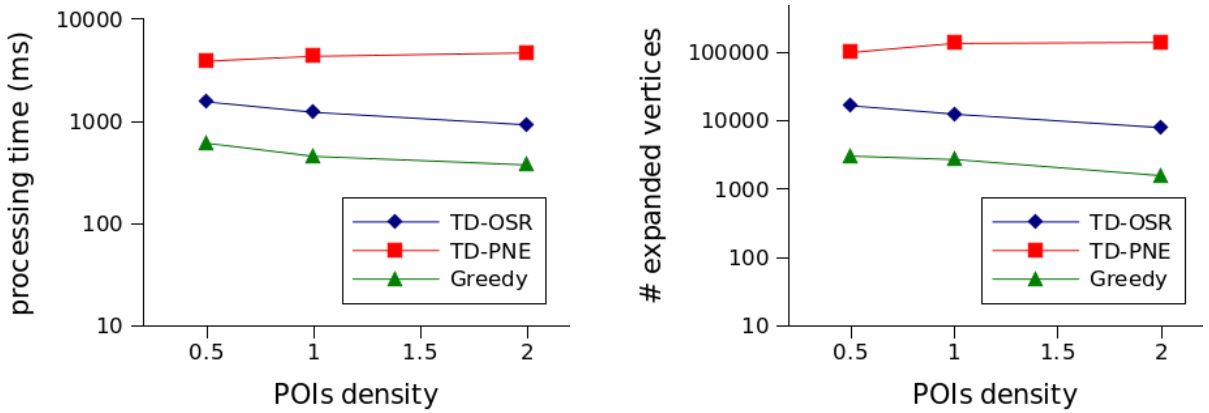


(a) (b)

Figura 4.6: Processing time of the queries and number of expanded vertices when the network size increases.

queries and the number of expanded vertices decrease with the density of POIs for the TD-OSR and Greedy solutions whereas it increases for the TD-PNE solution. When the number of POIs in the network increases, it is easier to find a POI and thus a route simply because there are more choices of POIs. This explains the behavior of the first two solutions. On the other hand, in the TD-PNE solution, even if the TD-NN searches are faster when the POIs become denser, the number of candidate routes to be investigated is greater. With an increasing number of POIs, there is a greater chance of finding partial routes that have a small cost and these routes will be investigated as they offer a chance to be the optimal route.

As in the previous experiment, the Greedy solution was faster and did expand less vertices than the other ones in exchange for a worse quality of routes. In this experiment, the cost of routes returned by this solution were, on average, 12.4% to 27.8% greater than the optimal route.

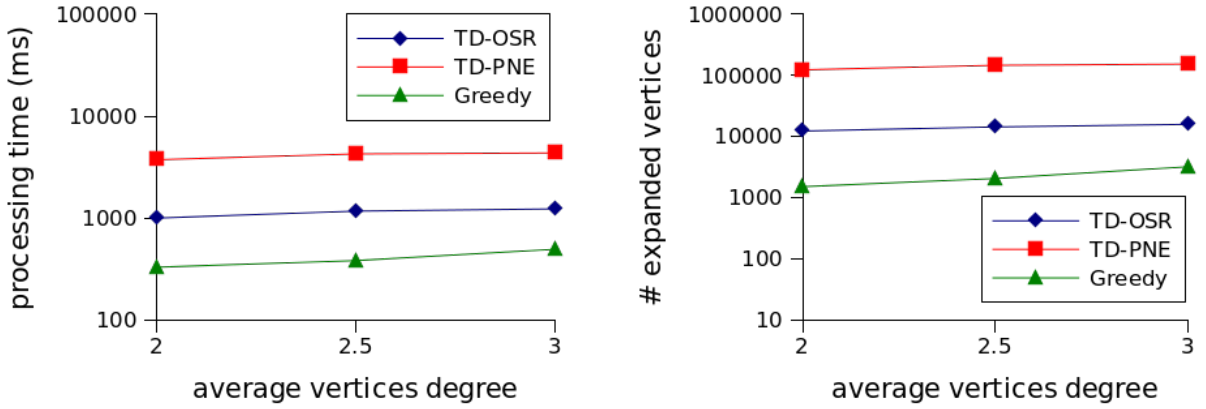


(a) (b)

Figura 4.7: Processing time of the queries and number of expanded vertices when the POI density increases.

Effect of the degree of the vertices. As expected and shown in Figure 4.8 the processing time of the queries and the number of expanded vertices increase with the degree of

the vertices in all solutions. This is reasonable since the number of paths in a network where the vertices have a higher degree is greater, making it more difficult to find POIs and consequently a route.

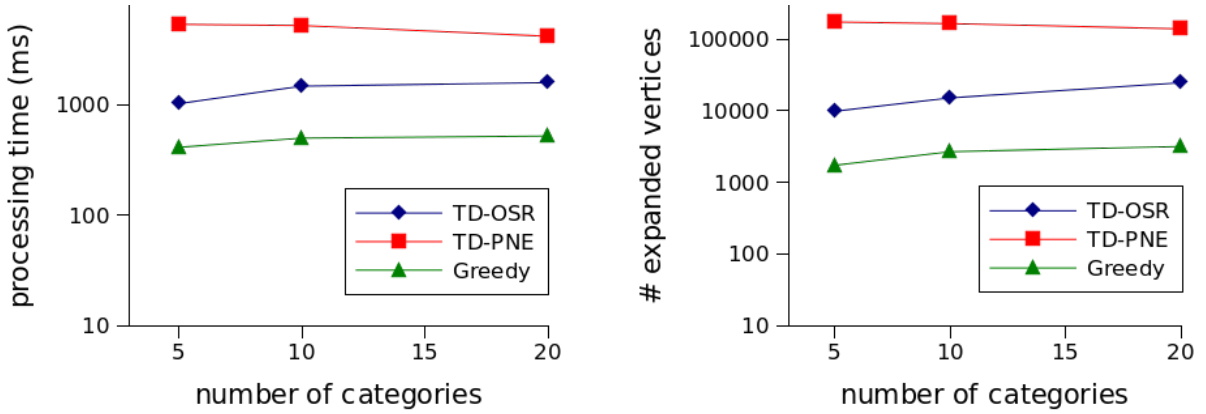


(a) (b)

Figura 4.8: Processing time of the queries and number of expanded vertices when the degree of the vertices increases.

We can notice that the Greedy solution is the most affected by this variable. This solution tends to find routes with higher cost than the optimal route. More specifically, the cost between some consecutive POIs in the sequence is higher than the cost in the optimal route. Thus, the TD-NN search between any of those POIs is costly, since they are far from each other. When the average degree of the vertices increases, those searches are likely to be even more costly because there is a greater number of paths which may be traversed when searching for a POI. Regarding the average cost of a route returned by this solution, it is from 12.2% to 37.1% greater than the optimal route.

Effect of the number of categories in the network. Figure 4.9 shows that the processing time of the queries and the number of expanded vertices increase with the number of categories of POIs in the network in the TD-OSR and the Greedy solutions while it decreases



(a) (b)

Figura 4.9: Processing time of the queries and number of expanded vertices when the number of categories of POIs in the network increases.

in the TD-PNE solution. As the POIs are uniformly distributed among the categories, when the number of categories increases, the number of POIs per category decreases. Thus, it is harder to find a POI that belongs to a certain category. This explains the behavior of the first two solutions. On the other hand, the TD-PNE solution takes advantage of this because it limits the number of candidate routes to be investigated.

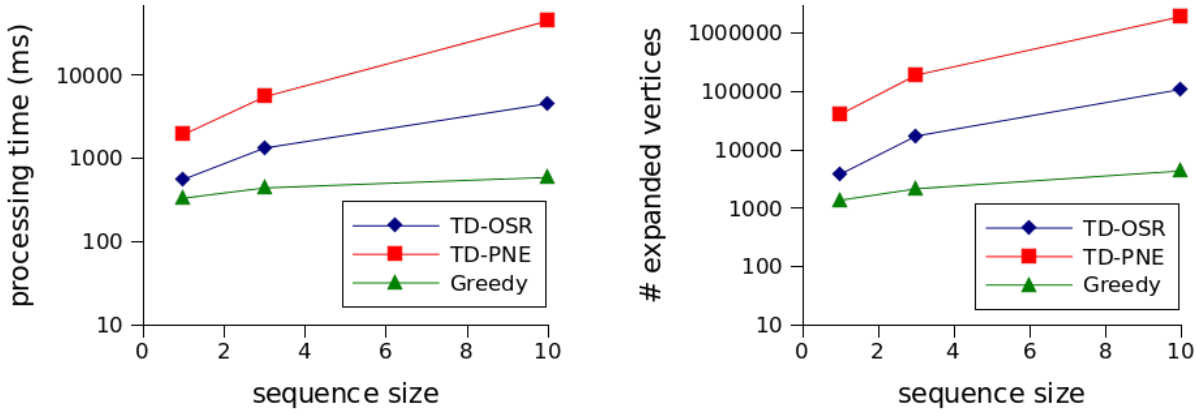
The cost of routes returned by the Greedy solution were, on average, 15.9% to 29.3% greater than the optimal route.

Effect of the sequence size. As expected and shown in Figure 4.10, the processing time of the queries and the number of expanded vertices increase with the size of the sequence given as input in all solutions. Generally speaking, this is because more vertices have to be checked to find a sequence with more POIs.

Note that this variable does not influence the cost of a single TD-NN search but more of those searches need to be performed in the TD-PNE and the Greedy solutions when the sequence size increases. Furthermore, in addition to this cost, in the TD-PNE solution there will be more candidates routes to be investigated. This explains why the sequence size affects more this solution than the Greedy.

We can also notice that this variable significantly affects the TD-OSR solution. This is reasonable since we guarantee that the maximum number of times a vertex is expanded is bounded to the size of the sequence given as input as proved in Lemma 4.4.2. The greater the size of the sequence, the greater the probability of a vertex to be re-expanded.

We emphasize that even though the Greedy solution is faster than the other ones, it does not guarantee the optimality of the route. Regarding this experiment, the average cost of a route returned by this solution is from 16.1% to 36.7% greater than the optimal route.



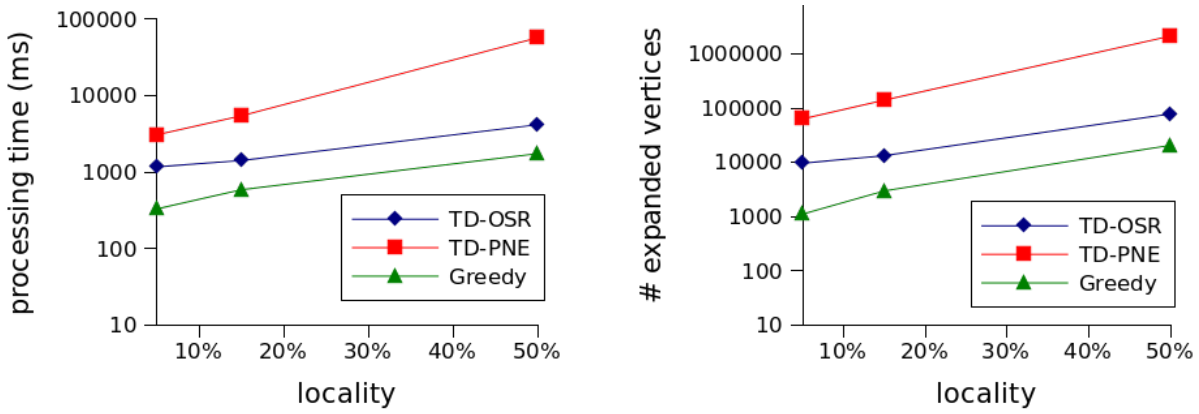
(a) (b)

Figura 4.10: Processing time of the queries and number of expanded vertices the sequence size given as input increases.

Effect of the locality of the query. Figure 4.11 shows that the cost of all solutions increase with distance between the origin and the destination. The larger the distance between the origin and the destination, the greater the number of candidate shortest paths between these two locations. This explains the behavior of the TD-OSR and TD-PNE solutions. Particularly,

more routes will be investigated in the TD-PNE solution because the upper bound found in this solution, in this case, tends to be high and many partial routes with cost within this upper bound may be found. This involves a greater number of TD-NN searches, which explains why this solution is the most affected by this variable.

The Greedy solution makes no effort to minimize the total cost of the route to the destination. Thus, the probability to follow a partial route that is far to the destination is high. This probability is even higher when the origin and the destination are farther, which implies an increase in the cost of some TD-NN searches performed in this solution. The average cost of a route returned by this solution is from 16.7% to 40.2% greater than the optimal route.



(a) (b)

Figura 4.11: Processing time of the queries and number of expanded vertices when the distance between the origin and the destination increases.

4.6 Conclusion

In this chapter we discussed the problem of processing OSR queries in time-dependent road networks. For comparison purposes, we presented a baseline solution, named TD-PNE, which was obtained by extending the previously proposed PNE algorithm, which generates the optimal route from the starting to the ending point progressively by performing local NN searches. This algorithm was originally proposed to solve OSR queries in static networks. To cope with the time-dependent problem, we substituted the NN searches by TD-NN searches. Also, we proposed an improved solution which is based on the INE algorithm and uses an A* search to guide the expansion of vertices. Furthermore, we proposed a scheme to reduce the number of nodes re-expansion. Both of the solutions return the optimal route.

We presented an experimental evaluation that compared the proposed solutions in terms of processing time and number of expanded vertices. The TD-OSR solution outperforms the TD-PNE in both the criteria. We also compared the TD-PNE and TD-OSR solutions to a greedy approach in order to know how longer is the travel time of a route returned by this solution. The results showed that the greedy solution outperforms the other ones, but a route returned by this solution is, on average, from 6.6% to 40.2% longer than the optimal route.

5 CONCLUSION AND FUTURE WORK

In this thesis we discussed the problems of processing k-NN with operating time constraints and OSR queries in time-dependent road networks.

We proposed three solutions to the first problem named Naive, Optimistic and Bounded. All of them perform an incremental expansion of the network and use an A* search to guide this expansion. The solutions are split into two phases: an off-line pre-processing and the query processing. In the first phase, bound values are calculated in order to accelerate query processing. Each solution is equipped with a suitable heuristic function and requires a different pre-processing to calculate its values. Our decision on proposing three different solutions to this problem is justified by the fact that mobility applications may require different update rate of the network. Experimental results showed that the number of data pages accessed in the Naive solution is up to 54% greater than the number of pages accessed in the Bounded solution. However, even though this solution outperforms the other two consistently, it is the most expensive in terms of pre-processing and may not be suitable if the TDG is updated frequently. In such cases one may want to use the Optimistic one.

For the second problem, we proposed two solutions. For comparison purposes, we first presented a baseline solution, named TD-PNE, which was obtained by extending the previously proposed PNE algorithm. Next, we proposed an improved solution which, similarly to the solutions proposed for the first problem, is based on the INE algorithm and uses an A* search in order to expand first vertices that offer a greater chance to be part of the optimal route.

We performed experimental evaluations for both problems. Regarding the first problem, the results showed that even though the Bounded solution outperforms the other two consistently, it is the most expensive in terms of pre-processing and may not be suitable if the TDG is updated frequently. In such cases one may want to use the Optimistic one. The experiments of the second problem indicated that the TD-OSR solution outperforms the TD-PNE in terms of both processing time and number of expanded vertices. We also conducted experiments using a greedy approach. The results showed that it did expand less vertices than the other two solutions and, consequently, the processing time was shorter, however, it is not suitable when the optimal route is required.

5.1 Future Work

Regarding the TD-OSR problem, a possible extension would be consider how long it takes to be served at a POI, i.e., the time between arrival and departure. Considering this cost instead of just the time one expects to spend at a POI represents better a real scenario. In some categories of POIs, for example, restaurants, it would be interesting to take into account both the estimated time to be served and the time that one wants to spend there. It is important to notice that the time that one spends at a POI may depend on the arrival time. Thus, as the time it takes to traverse an edge on a time-dependent network is a function of the departure time, the time spent at a POI would be a function of the arrival time.

Note that an extension of the TD-kNN-OTC problem could also include the estimated time that one spends at POI. Thus, this query would return the k POIs in which the sum of the travel time, the waiting time and the time it takes to be served is minimized.

Another possible future work is to create time-dependent road networks using trajectories of real moving objects. Then, we would like to perform the experiments using such networks. Furthermore, it would be interesting to implement the proposed approaches in a database, instead of using a file system. Finally, we would also like to propose solutions to solve others popular spatial queries but in the context of time-dependent networks.

REFERENCES

BERCKEN, J. V. den et al. Xxl - a library approach to supporting efficient implementations of advanced database queries. In: *Proc. of the 27th VLDB Conf.* [S.l.: s.n.], 2001. p. 39–48.

BéRUBé, J.-F.; POTVIN, J.-Y.; VAUCHER, J. Time-dependent shortest paths through a fixed sequence of nodes: application to a travel planning problem. *Computers & Operations Research*, p. 1838 – 1856, 2006.

CHEN, H. et al. The partial sequenced route query with traveling rules in road networks. *GeoInformatica*, p. 541–569, 2011.

CRUZ, L. A.; NASCIMENTO, M. A.; MACÊDO, J. A. F. de. K-nearest neighbors queries in time-dependent road networks. *Journal of Information and Data Management*, p. 211–226, 2012.

DEMIRYUREK, U.; BANAEI-KASHANI, F.; SHAHABI, C. Efficient k-nearest neighbor search in time-dependent spatial networks. In: *Proc. of the 21st DEXA Conf.* [S.l.: s.n.], 2010. p. 432–449.

DEMIRYUREK, U.; BANAEI-KASHANI, F.; SHAHABI, C. Towards k-nearest neighbor search in time-dependent spatial network databases. In: *Proc. of the 6th DNIS Workshop*. [S.l.: s.n.], 2010. p. 296–310.

DIJKSTRA, E. A note on two problems in connexion with graphs. *Numerische Mathematik*, Springer-Verlag, p. 269–271, 1959.

EISNER, J.; FUNKE, S. Sequenced route queries: Getting things done on the way back home. In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. [S.l.: s.n.], 2012. (SIGSPATIAL '12), p. 502–505.

ERWIG, M.; HAGEN, F. The graph voronoi diagram with applications. *Networks*, v. 36, p. 156–163, 2000.

GEISBERGER, R. et al. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: *Proceedings of the 7th International Conference on Experimental Algorithms*. [S.l.: s.n.], 2008. (WEA'08), p. 319–333.

GEORGE, B.; KIM, S.; SHEKHAR, S. Spatio-temporal network databases and routing algorithms: A summary of results. In: PAPADIAS, D.; ZHANG, D.; KOLLIOS, G. (Ed.). *Advances in Spatial and Temporal Databases*. [S.l.: s.n.], 2007, (Lecture Notes in Computer Science). p. 460–477.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, p. 47–57, 1984.

HART, P.; NILSSON, N.; RAPHAEL, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, p. 100–107, 1968.

HTOO, H. *Query Algorithms for Location Based Services in Road Network Distance*. Tese (Doutorado) — Saitama University, 2013.

HTOO, H. et al. Optimal sequenced route query algorithm using visited poi graph. In: *Web-Age Information Management*. [S.l.: s.n.], 2012, (Lecture Notes in Computer Science). p. 198–209.

HU, H.; LEE, D. L.; XU, J. Fast nearest neighbor search on road networks. In: *In Proc. of the 10th EDBT Conf.* [S.l.: s.n.], 2006. p. 186–203.

JENSEN, C. S. et al. Nearest neighbor queries in road networks. In: *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*. [S.l.: s.n.], 2003. p. 1–8.

KOLAHDOUZAN, M.; SHAHABI, C. Voronoi-based k nearest neighbor search for spatial network databases. In: *Proc. of the 13th VLDB Conf.* [S.l.: s.n.], 2004. p. 840–851.

KOLAHDOUZAN, M.; SHAHABI, C. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, p. 321–341, 2005.

LEE, K. C. K.; LEE, W.-C.; ZHENG, B. Fast object search on road networks. In: *Proc. of the 12th EDBT Conf.* [S.l.: s.n.], 2009. p. 1018–1029.

LI, F. et al. On trip planning queries in spatial databases. In: *Proceedings of the 9th international conference on Advances in Spatial and Temporal Databases*. [S.l.: s.n.], 2005. (SSTD'05), p. 273–290.

NANNICINI, G. et al. Bidirectional a* search on time-dependent road networks. *Netw.*, p. 240–251, 2012.

OHSAWA, Y. et al. Sequenced route query in road network distance based on incremental euclidean restriction. In: *Database and Expert Systems Applications*. [S.l.: s.n.], 2012, (Lecture Notes in Computer Science). p. 484–491.

ORDA, A.; ROM, R. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, v. 37, p. 607–625, 1990.

PAPADIAS, D. et al. Query processing in spatial network databases. In: *Proc. of the 29th VLDB Conf.* [S.l.: s.n.], 2003. p. 802–813.

SHARIFZADEH, M.; KOLAHDOUZAN, M. R.; SHAHABI, C. The optimal sequenced route query. *VLDB J.*, p. 765–787, 2008.

SHARIFZADEH, M.; SHAHABI, C. Processing optimal sequenced route queries using voronoi diagrams. *GeoInformatica*, p. 411–433, 2008.