Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2021,1
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Visit Places on YourWay: A Skyline Approach in Time-Dependent Networks

Saeed Taghizadeh, Abel Elekes, Martin Schäler

2021

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Visit Places on Your Way: A Skyline Approach in Time-Dependent Networks

Saeed Taghizadeh
Karlsruhe Institute of Technology
saeed.taghizadeh@kit.edu

Abel Elekes
Karlsruhe Institute of Technology
abel.elekes@kit.edu

Martin Schäler
Karlsruhe Institute of Technology
martin.schaeler@kit.edu

ABSTRACT

Many people take the same path every day, such as taking a specific autobahn to get home from work. However, one needs to frequently divert from this path, e.g., to visit a Point of Interest (POI) from a category like the category of restaurants or ATMs. Usually, people want to minimize not only their overall travel cost but also their detour cost, i.e., one wants to return to the known path as fast as possible. Finding such a POI minimizing both costs efficiently is highly challenging in case one considers time-dependent road networks which are the case in real-world scenarios. For such road networks time decency means the time a user needs to traverse a road, heavily depends on the user's arrival time on that road. Prior works have several limitations, such as assuming that travel costs are coming from a metric space and do not change over time. Both assumptions hardly match real-world requirements: Just think of traffic jams at the rush hour. To overcome these limitations, we study how to solve this problem considering time-dependent road networks relying on linear skylines. Our main contribution is an efficient algorithm called STACY to find all non-dominated paths. A large-scale empirical evaluation on real-world data reveals that STACY is accurate, efficient and effective in real-world settings.

1 INTRODUCTION

Problem Definition. Many movement patterns in our lives exhibit some regularity. Probably the most obvious one is that many people take a fixed path from home to work. Other examples for such regular, preferred paths P^* are visiting friends or family members or pursuing regular hobbies. The optimal route planning for such paths is a well-studied problem. However, when following such a regular pattern, it frequently happens that one needs to additionally visit points of interest (POIs) like a shopping center or an ATM machine not lying on the preferred path P^*. Next, it is often not necessary to visit a specific POI, but one from a category called the category of interest (COI), such as the category of ATMs [1]. This offers flexibility when suggesting a route optimal according to the *personal* preferences of the user. In this context, some people want to minimize their *overall travel cost* (TC)[1], while others want to minimize their *detour cost* (DC), (i.e., one wants to return to P^* as fast as possible) or a combination of these two costs. This means that, given the same start and end location and a COI, there may be different optimal routes for different people. We illustrate this with Example 1.1 in the following.

Example 1.1. Assume that Alice and Bob, by chance, have the same P^* and want to visit the same COI. Further suppose that there are two possible routes R_1 and R_2. R_1 has minimal overall travel cost (TC), but high detour cost (DC), while R_2 has higher TC, but minimal DC. This means route R_1 may be optimal for Alice who likes exploring new parts of the city but has little time. In contrast, route R_2 may be optimal for Bob who wants to follow his daily routine as strictly as possible.

Various prior work shows that finding a path minimizing only TC is not always best for the user since users have different preferences [7, 12, 22]. Prior work further reveals that one has to additionally consider DC, and that it is best if the user can select between options, i.e., from the skyline [3] of all non-dominated paths considering TC and DC [12].

Regarding an approach that finds the skyline of all paths, we can derive the following requirements.

- **I – Comprehensiveness and correctness of results** One wants to find the skyline of all non-dominated paths (according to TC and DC) which connect the start and end location and visit one POI of the desired category. While this requirement might sound trivial, work targeting at related problems has issues finding *all* paths, as we explain later.
- **II – Generality** Road costs should not be constrained to metric spaces as others have often done in the literature so far[1, 9]. This is because users are not only interested in the length of the path they are traversing, but also in the time they spend on it. Since time is not metric, the triangle inequality does not hold in this case [5]. As an example, highway routes might be longer, but since one is allowed to drive faster it may take less time to follow them.
- **III – Flexibility** Route planning must adapt to the current traffic situation and consider potential better routes with respect to traffic conditions. The time a user needs to traverse a road heavily depends on the time of day. For instance, it may take one hour to traverse a road in the evening when everybody is returning home. At night time, this may take only 10 minutes. So we must consider *time-dependent networks* to handle real-world scenarios.
- **IV – Near Real-time Query Performance** We aim at near real-time query response times on real-world data (By near real-time, we mean a few seconds). This is because our solution is intended to be the foundation of a (mobile) service where users expect immediate feedback, i.e., query responses in a few seconds even on large road networks.

The first three requirements specify the functional properties of the query type. The last requirement asks for efficiency. We will see in Section 2 that none of the existing approaches addresses all of these requirements. Moreover, *Generality* and *Flexibility* are not addressed at all in prior work. Since there is no work addressing all the properties, this is a novel query type, which we name vi**S**i**T**ing pl**AC**es on the wa**Y** (STACY) query.

[1]Without loss of generality, costs refer to time in the remainder of this paper.

In this paper, we study how to evaluate STACY queries (Requirements I-III) efficiently (Requirement IV).

Challenges. As we will discuss in detail later, the basic approach to solve STACY queries suffers from severe time complexity issues. This is mainly because of the fact that the basic approach needs to look into the whole search space in order to find non-dominated paths. This turns out to be computationally infeasible especially when the network is large and the number of POIs is high.

The following additional challenges arise which we address in this paper: It is unclear how to reduce the search space to reduce query response time, ultimately targeting instant query answering. In particular, considering non-metric spaces means that we cannot use known pruning strategies from the literature to shrink the search space. In addition, the *Flexibility* requirement further increases the computational effort, calling for even more efficient pruning methods.

Contributions. As our first contribution, we propose two strong pruning strategies called local and global pruning strategies. Local pruning helps in shrinking the search space for paths that share the same exit and entrance nodes on P^*. Global pruning is employed to stop searching for paths when a certain condition holds. These two pruning strategies shrink the search space significantly.

The main contribution of this paper is an efficient algorithm to evaluate STACY queries. This algorithm employs the aforementioned pruning strategies to discard the partial paths that have no chance to be a part of the query answer set in early stages of their expansion. This reduces the network expansion factor significantly. With this, our proposed algorithm is much more scalable than the basic approach. In addition, our proposed algorithm is based on mathematical proofs ensuring the correctness of the applied pruning strategies, and to this end of the query result itself.

Since there is no prior work addressing STACY queries, and approaches in metric spaces are not directly applicable to time-dependent networks, we compare our approach to a baseline approach that solves STACY queries. Our results show that our proposed approach is several magnitudes faster than the baseline, with being qualitatively just as good. In addition, we show that our approach is not only faster in query processing than the baseline approach, but faster than present-day state-of-the-art approaches for non-time-dependent networks.

Paper Outline: In Chapter 2 we review related work. In Chapter 3 we discuss preliminaries and define the problem. In Chapters 4 and 5 we present our proposed solution and the experiments. Chapter 6 concludes our findings.

2 RELATED WORK

Since the introduction of skylines [3], it has become a key approach in the field of route planning [2, 11, 13, 15, 18, 21, 22].

The skyline concept is used in other areas such as finding Regions of Interest (ROIs) [14] and finding routes maximizing the probability of visiting POIs [10]. However, our settings are different than theirs in the sense that we focus on visiting POIs under certainty assumption.

In the following, we review work on finding optimal routes to visit a POI close to a given path considering travel or detour cost, i.e., approaches related to our problem. An inspection of respective approaches (See Table 1) reveals that none of them addresses all four requirements presented in the introduction. In the following, we discuss the approaches listed in the table.

Table 1: Comparison with previous works

	Requirements			
Works	I	II	III	IV
IRNN [20]	✗	✗	✗	✓
BPD [16]	✗	✗	✗	✓
kPNN [4]	✓	✗	✗	✓
ISR [9]	✓	✗	✗	✓
BCIRNN[1]	✓	✗	✗	✓
TDOSR [5]	✗	✓	✓	✓
STACY	✓	✓	✓	✓

In-Route Nearest Neighbor (IRNN) queries have been studied in [20]. An IRNN query returns a path including a POI with the minimum detour distance from the regular path. However, since it only minimizes the detour distance, it does not address *Generality* and *Flexibility*. It also does not cover the case where the user is interested in other costs such as total travel distance (*Comprehensiveness and correctness*).

[16] proposes best point detour (BPD) queries. Given a preferred path, a set of POIs and a detour distance threshold, BPD finds a POI which minimizes the detour distance from the preferred path, with the condition that the detour distance is less than a threshold. Like IRNN queries, this work only minimizes the detour distance and hence does not support the *Correctness and comprehensiveness* requirement. They also work on weight functions of metric spaces, so *Generality* and *Flexibility* are not covered.

While the queries have a preferred path, the work in [4] assumes that the shortest distance connecting the source with the destination specifies the path of the user and finds K points of interest with minimum detour distance. This query is called k-Path Nearest Neighbor (kPNN) query. Like IRNN and BPD, it does not cover *Generality* and *Flexibility*.

Unlike the previous approaches, [9] studied In-Route Skyline (ISR) queries in which not only the detour distance but the total travel distance as well is important for the user. The result of such a query is a set of POIs (not entire paths) which are neither dominated in terms of travel distance nor detour distance. This work addresses the *Comprehensiveness and correctness* requirement, but it does not address the case where the weights are non-metric (*Generality*) and time-varying (*Flexibility*).

Best Compromise In Route Nearest Neighbor (BCIRNN) is the most similar concept to our work. BCIRNN queries find the paths which minimize the detour distance and travel distance in a skyline manner. As with ISR queries, the restrictive assumption is that the weights come from metric spaces, and therefore the triangle inequality holds (i.e., no *Generality*), and the weights are invariant (no *Flexibility*). This gives way to strong pruning methods in the first phase of BCIRNN, eliminating a large share of candidate POIs. In contrast, time is not metric in the STACY case, hence any existing POI could be a potential candidate to be visited (i.e., one cannot prune the candidate POIs even if they are far away in terms of

distance). Moreover, in contrast to STACY, it is not straightforward to extend BCIRNN to answer queries where multiple categories are of interest.

Regarding time-dependent networks, [5] proposes TDOSR queries. Such a query gets a sequence of COIs as input and returns a path visiting at least one POI of each COI, minimizing the travel cost of the path, but not the detour cost. STACY, on the other hand, aims at minimizing the combination of the detour and travel cost. In addition, as we will explain later, TDOSR has a correctness issue, which has been verified by the original authors (*Correctness and comprehensiveness*).

In contrast to the previous works listed here, STACY addresses all requirements presented in the introduction.

3 PRELIMINARIES AND PROBLEM DEFINITION

In this section, we introduce our notation and discuss preliminaries. To improve readability, all definitions are intended to use the same or at least a similar notation as in related work. As a reference, we refer to [1].

3.1 Road Network Structure

In contrast to work relying on Euclidean networks (i.e., networks where the triangle inequality holds), our underlying road network is a *Time-Dependent Network*. We now formally introduce this kind of network and explain the difference with the Euclidean network.

Definition 3.1. Time-Dependent Networks: A time-dependent network $G^t(V, E, W(t))$ is a set of nodes V (junctions) and edges E (roads). $W(t)$ is a set of functions returning the time needed to traverse each edge $(v_i, v_j) \in E$ at time instance t.

$W(t)$ is also called the weight function. The lowest travel cost $time_{Euc}(v_i, v_j)$ between two nodes v_i and v_j is the time needed to traverse the (Euclidean) distance between the nodes with maximal speed (v_{max}). Since we work with real-world networks there exists a speed limit on each road. We assume that the maximal speed of the user never exceeds the speed limit of the corresponding road.The reason that we need new lower bounds (in comparison to metric spaces) is that, due to the above definition, the weights do not need to come from a metric space. This means, known pruning strategies [1] exploit metricity, i.e., are based on the triangle inequality and hence are not applicable here.

Regarding the road network G^t, we assume that the user who starts traversing an edge first will finish traversal first as well. This common and realistic assumption is called first-in-first-out (FIFO) property [5, 6]. This means that it never pays off to wait at some node, hoping that the travel cost drops. In other words, if two partial paths meet at a shared node, the one that has visited fewer COIs and has a later arrival time on that node can safely be pruned.

In addition, we refer to the *arrival time* (AT) at node v_j^i on path P^i departing at time t as $AT(v_j^i, t)$ [2]. Note that the time served in each POI (for example spending half an hour in a restaurant) also affects the detour cost DC and travel cost TC of the corresponding path. This is because when a user leaves the POI, he enters the next road at a different point in time. Hence, we assign a serving time to each POI, which we call 'spent time'.

A *path* $P^i = \langle v_1^i, v_2^i, \ldots, v_n^i \rangle$ is a cycle-free sequence in G^t [3] and any two consecutive nodes v_j^i, v_{j+1}^i are neighbors in G^t. Similarly, a *detour path* $P^i(ds, dd)$ is defined as a cycle-free path in G^t with only detour source (ds) and detour destination (dd) nodes located on the preferred path P^*.We refer to a path that has visited at least one POI and ends in the destination as *full path*. Finally, in a preferred path $P^* = \langle v_1^*, v_2^*, \ldots, v_n^* \rangle$, the first and last node are the source and destination, respectively.

3.2 Costs

Definition 3.2. Travel Cost (TC): given a path $P^i = \langle v_1^i, v_2^i, \ldots, v_n^i \rangle$ and a departure time t, the travel cost is as follows:

$$TC(P^i, t) = \sum_{j=1}^{n-1} w(v_j^i, v_{j+1}^i, AT(v_j^i, t))$$

The travel cost is the sum of the individual traversals of edges (road weights). Since road weights change over time, $AT(v_j^i, t)$ specifies the time instance.

Definition 3.3. Detour Cost (DC): Given a preferred path P^*, a path $P^i = \langle v_1^i, v_2^i, \ldots, v_n^i \rangle$ and and a departure time t, the detour cost is as follows:

$$DC(P^i, P^*, t) = \sum w(v_j^i, v_{j+1}^i, AT(v_j^i, t))$$

where the sum goes for all edges $(v_j^i, v_{j+1}^i) \in P^i$ which are not a part of P^*.

As we can see, the detour cost is the sum of the travel costs on the edges that do not belong to preferred path P^*. Note, that by following a detour, the user can make a shortcut to the destination, i.e., ds is not necessarily equal to dd.The travel cost and detour cost are functions of the departure time t of the user.

3.3 Linear Skyline Operator

Previous work has shown that the skyline operator is meaningful, but the query result can be large, which may be confusing [1]. Hence, we use a variant of the operator called *linear skyline operator* [19]. In the following, we introduce the conventional skyline and linear skyline operators, then we discuss their differences.

We define conventional dominance as follows:

Definition 3.4. Conventional Dominance Let P be the set of all possible paths. Each path P^i is assigned a cost vector $p^i = (p_1^i, p_2^i)$ representing its travel cost and detour cost (i.e., $(p_1^i, p_2^i) = (TC(P^i), DC(P^i))$). $P^i \in P$ dominates $P^j \in P$, written as $P^i \prec P^j$, when the following holds:

$$(p_1^i < p_1^j \wedge p_2^i \leq p_2^j) \vee (p_1^i \leq p_1^j \wedge p_2^i < p_2^j)$$

Conventional dominance guarantees that P^i is better than P^j in at least one parameter and not worse in the other parameters. Thus, the set of non-dominated paths in P is $\left\{P^i \in P | \nexists P^j, P^j \prec P^i\right\}$.

[2] For presentation purposes, we drop the departure time t from the equations whenever needed assuming that the user departed at time t

[3] Except in detours where a path potentially can visit a node more than one time; this happens when the user reaches the POI through a route and comes back to P^* through the same route

In contrast, linear skyline maximizes the set of full paths which minimize the parameter $F = \delta_1 \cdot p_1^i + \delta_2 \cdot p_2^i$ for all possible combinations of weight vectors $\delta = (\delta_1, \delta_2)$:

Definition 3.5. Linear Dominance A Path P^i is δ-dominated by path P^j iff $\delta^T \cdot p^i < \delta^T \cdot p^j$ where $\delta \in R^2 > (0,0)$ is a weight vector, δ^T is the transpose of vector δ, R represents the real numbers, and R^2 is a point in the XY plane.

Using this definition, the linear skyline [19] is as follows:

Definition 3.6. Linear Skyline Let P be the set of all possible paths. Set $P' \subseteq P$ dominates path $P^j \in P$, denoted as $P' \prec_L P^j$, when the following holds:

$$(\exists P^i \in P' | P^i \prec P^j) \vee (\forall \delta \in R^2 > (0,0), \exists P^i \in P' | \delta^T p^i < \delta^T p^j)$$

The maximal set of linearly non-dominated paths is called linear skyline (LS) set.

It is computationally expensive to check the predicates above for any possible δ vector. The authors of [19] have shown that the linear skyline has a graphical explanation, simplifying the computation of linear dominance. A path P^i is dominated by the paths in P' if P^i is located somewhere above the straight line with a negative slope connecting any two paths from P'. This is illustrated in Figure 1b, where P^3 is dominated by $P' = \{P^1, P^2\}$.

Suppose that $\{P^i, P^j\} \subseteq P'$, and $p_1^i > p_1^j$ and $p_2^i < p_2^j$ holds. Let $u_1 = \min(p_1^i, p_1^j)$ and $u_2 = \min(p_2^i, p_2^j)$. We define $u := (u_1, u_2)$. Next, let n be the normal vector of the straight line connecting P^i and P^j in the two dimensional vector space so that $n^T p^i = n^T p^j$. Then a path P^k is linearly dominated by $\{P^i, P^j\} \subseteq P'$ if and only if the following condition holds:

$$(u \prec p^k) \wedge (n^T p^k > n^T p^i = n^T p^j).$$

Figure 1 illustrates the differences between conventional skyline and the linear skyline operator. The paths $\{P_1, P_2, P_3, P_4\}$ have different DC and TC, as shown in Figure 1. Figure 1a graphs the share of the plane which is conventionally dominated by each path. As one can see, only P_4 is dominated by P_3. So the result of the conventional skyline is $\{P_1, P_2, P_3\}$. On the other hand, Figure 1b graphs the share of the plane which is linearly dominated, using an arrow (i.e. the highlighted area pointed by arrow is linearly dominated area). Paths P_3 and P_4 are in the dominated area and are not in the result, which is $\{P_1, P_2\}$.

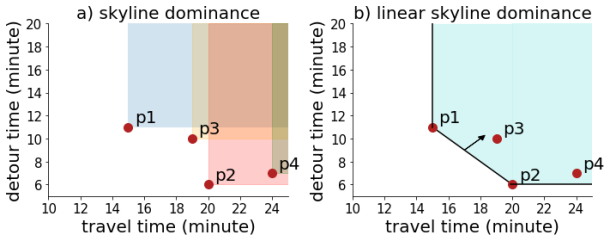

Figure 1: Skyline dominance: conventional (left), linear (right)

3.4 Problem Definition

In the following, we define the problem formally:

Definition 3.7. STACY Query $STACY(P^*, t, G^t, COI)$ takes as input a departure time t, a preferred path P^* within a time-dependent network G^t, and a COI. It returns all paths visiting at least one point of interest from the COI which are not linearly dominated by any other path in terms of detour cost DC and travel cost TC.

The result of a STACY query is a list $LS = \{P^1, P^2, \ldots, P^k\}$ containing paths which are not linearly dominated by any other combination of paths in the list. Note that the result of a STACY query depends on the departure time.

3.5 A^* Search Algorithm

Similar to all approaches presented in the related work section, our approach relies on ideas directly originating from the A^* Search Algorithm. Thus, we briefly introduce the A^* Search Algorithm helping to introduce our STACY algorithm in the remainder.

Basically our approach expands partial paths in an A^* manner. Since it is the most commonly used search algorithm in route planning, this work relies on the notion of partial paths used in A^* aiming at improving the readability of the paper.

The A^* or best-first search is an algorithm used frequently in path-finding problems in weighted graphs works as follows: Starting from node s, its goal is to find a path to the destination node d having the smallest cost. This is done by keeping a tree of partial paths originating from start node s and expanding those partial paths one edge at a time. The idea is presented in Algorithm 1.

Algorithm 1: A^* algorithm

Input: Starting node s, Destination node d, Departure time t,Time-Dependent graph G^t
Output: Path P with minimum travel time from s to d

```
1 function A* Algorithm
2     Initialize priority queue PQ with start node s
3     while PQ ≠ ∅ do
4         P ← PQ.pop()
5         if P is a path ending in d then
6             return P
7         Expand partial path P with the neighbors of its
8         last visited node and add the expanded paths to PQ
```

At first step, the algorithm initializes the priority queue (PQ) with the start node s (Line 2). At each iteration of the algorithm, the path P with minimum cost is selected from PQ (Line 4). If path P meets the destination node d, the algorithm has found the path and terminates immediately (Lines 5-6). Otherwise, path P is expanded with all neighbors of its last visited node (Lines 7-8).

As one can see in the algorithm, A^* needs to select a partial path for expansion (Line 4). This selection is based on the actual travel cost of the partial path and also an estimated travel cost to reach the destination node d. In other words, it selects the path which minimizes the following function:

$$OTC(v) = TC(v) + h(v)$$

where OTC presents optimistic travel cost on node v, TC shows the travel cost of partial path P (expanded partially to node v) and $h(.)$ (also called heuristic function) estimates the travel cost to reach destination d.

It has been proven that if the function $h(.)$ underestimates the remaining travel cost of the path, the algorithm always finds the optimal path.

4 PROPOSED APPROACH

This section features our approach to evaluate STACY queries. This section is organized as follows. In the first subsection, we present a naïve STACY approach and motivate why extensions are needed to improve it. In Section 4.2 we introduce the preliminary ideas which we will use in the STACY algorithm. Then, we present our proposed algorithm to solve STACY queries in Section 4.3.

4.1 Naïve STACY Approach To Motivate Our Extensions

For didactic reasons, we first introduce a straight-forward approach to compute STACY query results. It illustrates the general idea including how we use (a modified) variant of the A^* algorithm. As we outline the problem of this solution, is its computational complexity being in opposition to our near real-time query response time requirement. To this end, based on this naïve approach, we motivate our main contributions within the STACY algorithm aiming at significantly reducing the computational complexity.

4.1.1 Naïve STACY. Recall from Section 3, that a STACY query aims at finding the set of paths which are not linearly dominated by any other paths. Therefore, any path connecting the desired start location s and destination d also visiting a POI of the desired category has to leave the preferred path P^* in some node named detour source (ds) and return to P^* in some node named detour destination (dd). As a result, given that path P^* has n nodes and there are m POIs belonging to the desired COI, a naive approach to compute STACY queries is the following.

The naïve algorithm for STACY queries first runs a modified A^* algorithm for each possible combination of detour start and detour destination (ds,dt) and each POI m finding the shortest path connecting them. This results in $m * \binom{n}{2}$ A^* algorithm invocations. Then, we compute the linear skyline on the result set. Note that running the modified A^* algorithm that often to find the paths is unavoidable while the network is time-dependent.

4.1.2 Required Extensions. In the context of the naïve algorithm for STACY, modified A^* refers to an A^* algorithm using a modified heuristic function $h(.)$ being a lower bound for the real cost. We develop this bound, such that it also holds for time-dependent networks.

The extensive computation of invoking the modified A^* algorithm $m * \binom{n}{2}$ times hardly leads to the desired near real-time query performance (*Requirement IV*). So we need to shrink the search space. The general idea of how to shrink the search space is to find provable upper bounds for costs in order to prevent expansion of paths that cannot be part of the linear skyline. This is done by the two pruning strategies, namely local and global pruning. The purpose of the local pruning strategy is to limit the A^* invocations per detour start and detour destination (ds,dt), such that we do not need to consider all m POIs. Subject of the global pruning strategy is to reduce the number of detour start and detour destination (ds,dt) pairs, we need to consider. Since this is done by finding a global upper bound for costs, we refer to it as global pruning strategy.

4.1.3 Outline for Introducing the STACY algorithm. In Section 4.2, we introduce the heuristic function for the modified A^* algorithms as well as the two pruning strategies. Then in Section 4.3, we introduce the STACY algorithm in detail illustrating, for instance, how the pruning strategies are used when expanding partial paths and how additional properties relevant in time-dependent networks, such as the FIFO property, are ensured.

4.2 Search Heuristic and Pruning Strategies

In this section, first we discuss the heuristic function used by A^* and then we show how we can speed up its calculation by preprocessing the road networks. Then, we present the pruning strategies used in the STACY algorithm and prove their correctness.

4.2.1 A^* Search Heuristic Function. Generally, the total expected travel costs of a path P that has been expanded until some node v_i is the sum of the cost so far $TC(v_i)$ and an estimation of the remaining costs. The remaining costs include reaching the destination and visiting a POI (if not done until v_i). For estimating the remaining costs, we need a search heuristic function $h(.)$.

As we mentioned earlier, the A^* search heuristic function $h(.)$ should underestimate the travel cost of the remaining path, i.e., be a lower bound. For a path P that has been expanded to node v_i, we compute the heuristic function as follows:

$$h(v_i) = \begin{cases} \max(LB(v_i), time_{Euc}(v_i, d)), \text{if } P \text{ has not visited a POI} \\ time_{Euc}(v_i, d), \text{otherwise.} \end{cases}$$

Where $LB(v)$ is a lower bound of the cost to the nearest POI of the desired category starting from v_i.

The rational behind this heuristic is: If path P has not visited a desired POI during its expansion already, its remaining travel cost is at least the maximum of two possible costs: (1) Either the cost of visiting the closest POI ($LB(v_i)$), or (2) the cost of heading to destination d through a direct line (ignoring the road network). In case P already visited a POI during its expansion, it only remains to continue its travel towards the destination.

Calculating $LB(v)$. In order to calculate A^* search heuristic function, one needs to calculate $LB(v)$. However, calculating $LB(v)$ has two main difficulties: First, the travel costs of roads are changing over time. Second, calculating $LB(v)$ is a time-consuming task and this is in contrast with the last requirement (*IV–near real-time Query Performance*).

To overcome the first difficulty, we propose to obtain a time-invariant version of graph G^t. We refer to time-invariant version as $\underline{G^t}$. $\underline{G^t}$ has the same nodes V and edges E as G^t, but the edge weights are the minimum travel cost over the entire day. Hence graph $\underline{G^t}$ is an optimistic time-invariant version of graph G^t.

While $\underline{G^t}$ has fixed travel costs for its edges, for each node v in V, one can calculate and store cost to the closest POI from COI. In other words, the values of $LB(v)$ can be computed for every node v in graph $\underline{G^t}$ in a preprocessing step. Then, when answering STACY

queries, one does not need to compute $LB(v)$: it suffices only to look up the stored values and take the corresponding value for v.

Having $\underline{G^t}$, one needs to run Dijkstra's algorithm starting from each node v to find the nearest POI from specified COI. Time complexity of Dijkstra's algorithm in worst-case scenario is $O(|E|\log(|V|))$ and since we run it $|V|$ times, the total complexity of this preprocessing is $O(|V||E|\log(|V|))$. However, note that this preprocessing is only done once for the entire road network. Also, the space complexity for storing lower bounds is $|V|$, while for each node $v \in V$ we calculate and store the cost to the nearest POI from specified COI.

4.2.2 Local Pruning Strategy. The following lemma allows the algorithm to discard paths sharing the same detour start and detour destination (*ds*,*dd*) using local pruning.

Lemma 4.1. *Let P and P' be two full paths which share the same detour start and detour destination nodes on P^*. If $DC(P,t) < DC(P',t)$, then P' cannot be in the query result.*

Proof. The proof is in appendix A. □

As result of this lemma, having found a full path, we can stop expanding (i.e., search for better POI for this (ds, dd) pair) in case the DC exceeds the DC of the currently best found full path regardless of whether the paths are partial or full paths. Example 4.2 presents the local pruning strategy.

Example 4.2. Consider paths P and P' in Figure 2 which share (v_i^*, v_j^*) as their detour source and destination respectively. Also, assume that path P has the minimum DC among all other paths connecting v_i^* to v_j^* visiting at least a POI. Then, according to the local pruning strategy, we do not need to be concerned about paths like P', while the above-mentioned lemma guarantees that such paths will not be part of the query answer set.

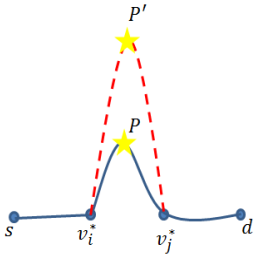

Figure 2: Detour paths

As one can see, the local pruning strategy assures us that for any (ds, dd) pair on P^*, we only need to look for the path with minimum DC. Therefore, for any (ds, dd) pair, we need to run A^* only once. Hence, the total number of times that one needs to run A^* is reduced significantly from $m * \binom{n}{2}$ to $\binom{n}{2}$.

4.2.3 Global Pruning Strategy. Using local pruning, we already reduced the required invocations of our modified A^* algorithm from $m * \binom{n}{2}$ to $\binom{n}{2}$, i.e., to one invocation per detour start and detour destination (*ds*,*dd*). Now global pruning aims at reducing the number of such considered pairs since time complexity is still quadratic regarding the number of nodes on the preferred path P^*.

The global pruning strategy is by far more complex than the local one. To this end, we first introduce the general idea behind the global pruning strategy which is based on finding the travel cost of the fastest path $Fastest^{TravelCost}$. Then, we prove the existence of $Fastest^{TravelCost}$ path and show that it is in the STACY query result set LS. Finally, we give details on how to obtain $Fastest^{TravelCost}$ in an efficient way.

The general idea of the global pruning strategy. The general idea behind our global pruning strategy is to find an upper bound for detour cost DC *before* starting to generate the partial paths at all.

By the definition of skyline queries, the path with overall minimal travel costs TC departing from source s, visiting a POI, and arriving at destination node d, provides an upper bound for the detour cost DC. That is, it dominates all paths having larger DC, and we can safely prune them (cf. $P1$ in Figure 1). Note this also includes partial paths whose already known DC or a respective lower bound of the DC, meaning we can stop expanding such paths.

We call the fastest path $Fastest^{TravelCost}$ and the corresponding upper bound for the detour cost $DetourCost^{Upper}$. The formal rationale for correctness of the bound is that, since we are looking for skylines, any path P^i with $DC(P^i) > DetourCost^{Upper}$ is dominated by $Fastest^{TravelCost}$. This is because $Fastest^{TravelCost}$ incurs minimal travel cost by definition, and hence $TC(P^i) > TC(Fastest^{TravelCost})$.

Proving that $Fastest^{TravelCost}$ path exists and is in the LS set. The argument above shows that $Fastest^{TravelCost}$ cannot be pruned by any other path using the skyline operator, i.e., the bound is correct. However, to be able to use $DetourCost^{Upper}$ as the upper bound for detour cost, two problems arise that we have to address: First, since the network is FIFO, we need to prove that $Fastest^{TravelCost}$ cannot be discarded by other partial paths due to FIFO property of the network. Second, we need to obtain $Fastest^{TravelCost}$ efficiently. In the following, we discuss the existence of $Fastest^{TravelCost}$ in the linear skyline set and then show how one can obtain it.

Due to the FIFO property, we need to make sure that there is no path arriving at any node being part of $Fastest^{TravelCost}$ earlier than $Fastest^{TravelCost}$ itself. Otherwise, the path with earlier arrival time will be expanded and $Fastest^{TravelCost}$ will be discarded, i.e., is not part of the skyline. The following lemma proves that $Fastest^{TravelCost}$ will not be pruned and will be part of the skyline answer set.

Lemma 4.3. *Having query $STACY(P^*, t, G^t, COI)$, consider $Fastest^{TravelCost}$ to be the fastest path connecting source $s \in P^*$ to destination $d \in P^*$ visiting at least one POI $\in COI$ at departure time t. Then $Fastest^{TravelCost}$ is a part of the query result.*

Proof. The proof is in appendix B. □

Since $Fastest^{TravelCost}$ will be in the result, we can use $DetourCost^{Upper} = DC(Fastest^{TravelCost})$ to shrink the search space.

Obtaining $Fastest^{TravelCost}$ with TDOSR. We now discuss how to calculate $Fastest^{TravelCost}$. For this purpose we use a *modified*

version of the TDOSR algorithm [5]. The TDOSR algorithm in its original version addresses the optimal sequenced route query [17] in time-dependent networks. However, a detailed inspection of the algorithm reveals that the TDOSR algorithm does not produce optimal results in some cases. We corrected this issue in consent with the original authors being another minor contribution of this paper.

TDOSR algorithm retains the partial paths in a priority queue PQ ordered by their optimistic travel costs (OTCs). At each iteration of the algorithm, the path P with the lowest OTC is popped from the queue and is expanded with the nearest neighbors of its last vertex P_{last}. The new partial paths will be inserted into the priority queue if they do not violate the FIFO property of the network. For this purpose, TDOSR employs a validation procedure as follows: If paths P^i and P^j are two different paths from s to v and $OTC(P^i) < OTC(P^j)$ and P^i contains more POIs, then P^j could not be in the result set. However, we argue that this could not be true in some cases. This is simply because of the fact that having $OTC(P^i) < OTC(P^j)$, one cannot make sure that P^i will arrive at destination d earlier than P^j (which violates the FIFO property of the network).

We have corrected the pruning strategy of TDOSR so that it yields the optimal results. Personal communication with the inventors of TDOSR has resulted in the consent that our modification solves the problem. We call this modified version of TDOSR tailored to our problem time-dependent Optimal Route (TDOR). TDOR shares the same heuristic with TDOSR. However, its FIFO validating strategy is changed as we explain next.

The correct strategy would be to compare the travel costs (not optimistic travel costs) of paths P^i and P^j. The following lemma states this:

Lemma 4.4. *For two paths P^i and P^j, if $TC(P^i) < TC(P^j)$ and P^i contains more POIs, then P^j could not be in the result set.*

Proof. The proof is in Appendix C. □

4.3 The STACY Algorithm

In this section, we first discuss the general idea behind our proposed algorithm to solve STACY queries. Then we present details of the different phases of our proposed algorithm. As illustrated in Algorithm 2, the core of the Stacy Algorithm is how it generates and prunes paths, which is explained in Section 4.3.1 (Line 3). Path generation and pruning contains several sub-steps including network expansion and updating the result set LS explained in Sections 4.3.2 to 4.3.4. Finally, we need to check whether the FIFO property for all found paths holds (cf. Section 4.3.5) (Line 4).

4.3.1 Generating and Pruning Partial Paths. In this section, we discuss the process of generating partial paths and pruning them based on the pruning strategies we introduced before. Similar to the A^* algorithm, we use a priority queue of partial paths aiming at expanding promising paths first, which also allows early termination of the algorithm if the whole result is found. However, as we consider two costs (travel and detour costs), return a set of paths building the linear skyline, and use as input a preferred path P^*, the semantics of the queue as well as the whole procedure of generating and pruning paths is quite different as we outline below.

Algorithm 2: STACY algorithm

Input: Starting node s, Destination node d, Departure time t,Time-Dependent graph G^t, $P^* = \langle v_1^*, v_2^*, \ldots, v_n^* \rangle$ and a COI

Output: Linear sklyline set VLS containing paths that have visited at least one POI from COI

1 **function** STACY()
2 $VLS \leftarrow \emptyset$
3 $LS \leftarrow$ *Generate and prune paths, Update LS* ▷ (Section 4.3.1, Algorithm 3)
4 $VLS \leftarrow$ *Validate FIFO property of LS* ▷ (Section 4.3.5)
5 *return VLS*

Priority Queue Initialization and Termination. To fully benefit from both pruning strategies, we associate each pair (ds, dd) with a lower bound of the respective detour costs DC named optimistic detour cost (ODC). ODC is computed using the same heuristic that we have presented earlier in Section 4.2.1, with the only difference that its source and destination are ds and dd respectively. This is because we want to generate the paths based on increasing the lower bound of the DC, such that we can prune them based on local and global pruning strategies. Specifically, this means, we terminate considering the next (ds, dd) pair having found the first pair whose ODC is worse than the DC of the path having minimum travel cost TC (apply global pruning).

This is valid since ODC is a lower bound for DC and paths are generated based on increasing DC, the search for new paths can be terminated when the first path P is found with $ODC(P) > DetourCost^{Upper}$. This is guaranteed by the following lemma and the respective proof.

Lemma 4.5. *The search for new paths can be terminated once the first partial detour path P^i is found such that $ODC(P^i) > DetourCost^{Upper}$.*

Proof. The proof is in appendix D. □

Path Generation. Having proven the termination condition of our search, we propose the path generation and pruning algorithm which also updates the linear skyline. The outline of the algorithm is presented in Algorithm 3.

In the algorithm, first we order the (ds, dd) pairs using a priority queue (PQ) based on increasing ODC. To initialize PQ, we generate all possible (ds, dd) pairs using nodes on P^* and put them in PQ (Lines 3-4). At each iteration the partial detour path P with minimum ODC is dequeued from PQ (Line 6). If P is a full detour path (i.e. a path that has visited at least one POI from COI and has met its corresponding dd), it is added to the linear skyline (LS) set and LS is updated to preserve the linear skyline (Lines 7-9). If P is a full path with (ds, dd) pair, according to local pruning strategy (Section 4.2.2), other paths that share the same (ds, dd) pair cannot be part of the linear skyline set LS and therefore we remove them immediately from PQ (Line 10). According to the global pruning strategy (Section 4.2.3), if the detour cost of path P is greater than $DetourCost^{Upper}$, we can stop our path generation algorithm since the remaining partial paths cannot be a part of LS anymore (Lines 11-12). If none of the above conditions holds, we expand the path P

Algorithm 3: Generating and pruning paths, updating LS

```
1  function Generate and Prune Paths (DetourCost^Upper)
2      LS ← ∅
3      Initialize priority queue PQ with all possible detour
4      source and detour destination pairs
5      while PQ ≠ ∅ do
6          P ← PQ.pop()
7          if P is a full path with (ds, dd) pair then
8              LS.add(P) ▷ (Section 4.3.3)
9              Update(LS) ▷ (Section 4.3.4)
10             Local pruning with (ds, dd) ▷ (Section 4.2.2)
11         if Global pruning holds ▷ (Section 4.2.3) then
12             Stop path generation
13         Expand partial path P with the Network Expansion
14         Algorithm ▷ (Section 4.3.2, Algorithm 4)
15     return LS
```

Algorithm 4: Network expansion

```
1  function Network Expansion()
2      forall v ∈ G^t.neighbors(u) do
3          if |P.pois| = 1 then
4              spent = spent + POI_Visit_Time
5          TCV ← P.TC + TC_(P_last,v)(P.AT)
6          ATV ← (t + spent + TCV + P.TC_BeforeDetour
7          if |pois| = 1 then
8              ODC ← TCV + time_Euc(v, P.dd)
9          ODC ← TCV + max(time_Euc(v, P.dd), LB(v))
10         if v¬isVisited then
11             P_new ← expand P by adding v
12             if PQ does not contain a path which has same
13             values on ds , dd and |pois| as P_new then
14                 PQ.add(P_new)
15             else
16                 P_inq ← PQ.getPath(P_new.ds, P_new.dd)
17                 if P_new.TC ≤ P_inq.TC &|P_new.POIs|| ≥
                    |P_inq.POIs|| then
18                     PQ.remove(P_inq)
19                     PQ.add(P_new)
```

with the neighbors of its last visited node (Lines 13-14) and continue with the next (*ds*,*dd*) pair.

The detailed explanation of adding a path to the *LS* set, updating the *LS* set and expanding path *P* are presented next in Sections 4.3.3, 4.3.4, and 4.3.2, respectively.

4.3.2 Network Expansion. In this section, we briefly discuss how the partial paths are expanded during network expansion. The general outline is presented in Algorithm 4.

The algorithm expands the path P adding to the currently last vertex v_{last} all (new) neighboring vertex. This way, we create multiple new paths. For each of the new paths, we do the following. If the new partial path has visited a POI during its expansion, we add the spent time (e.g., expected serving time within a restaurant) on the POI (Lines 3-4). Then the algorithm calculates the corresponding travel cost and arrival time namely TCV and ATV for the new path to v (i.e. the neighbor of v_{last}) and updates corresponding optimistic detour cost namely ODC (Lines 5-9). If v has not been visited before via another partial path with more number of visited POIs and less arrival time, we can expand the path P with this vertex and FIFO property still has not been invalidated. This, in turn, means, there is a possibility that adding this new path P_{new} to the priority queue PQ causes some of the previously expanded paths to v to be pruned out due to the FIFO property of the network. Hence, the algorithm checks possible partial paths in the PQ and discards all paths which have visited the same POIs with greater travel cost TC (later), (Lines 10-19) because they cannot visit the remaining POIs and reach to destination d faster. Note, this does not entirely ensure the FIFO property holds for all full paths found, as PQ contains only partial paths. There may be full paths already added to the result. This explains why we need to validate the FIFO property after having computed the skyline.

4.3.3 Adding a New Full Paths to the Linear Skyline. When a full path is generated it is not necessarily part of the skyline, i.e., it may be dominated by some other path. Since checking for linear dominance is expensive, we first examine whether the path is conventionally dominated.

Remember that *LS* keeps all linearly non-dominated full paths ordered based on the detour costs of the paths. On the other hand, the expansion algorithm itself generates the paths based on the ascending order of their respective detour costs. As a result, we already know that the newly generated path's detour cost is greater than the last element of *LS* namely P^k, i.e., $DC(P^i) > DC(P^k)$. Therefore, in order to add P^i to the *LS*, one needs to make sure that $TC(P^i) < TC(P^k)$. Otherwise P^i is conventionally dominated by P^k. In case $TC(P^i) < TC(P^k)$, it means that it is not conventionally dominated by the last path in *LS*. So in order to add it to the *LS*, we need to make sure that it is not linearly dominated by the set of paths in *LS*. However, this examination of all paths in *LS* is not necessary. It has been shown in [19] that we can safely add full path P^i to *LS* if it is not conventionally dominated by the path with the highest detour cost in *LS*.

4.3.4 Delete Old Dominated Paths. By adding a new P^i to *LS*, there is a chance that some of the existing paths in *LS* are now linearly dominated by the newly added path (this is a problem since we are looking for linearly non-dominated paths). We need to find all those paths and remove them from *LS*. To do so, the authors of [19] have shown that, it is not necessary to check all possible paths in *LS*. They have proven that a path is linearly dominated by *LS* if and only if it has been linearly dominated only by its neighbors. Therefore after adding P^i, for P^k, it should be checked that $\left\{P^{k-1}, P^i\right\} \prec_L P^k$ holds or not. If it holds, P^k will be removed. Then P^i and P^{k-2} will be the neighbors of P^{k-1} and therefore $\left\{P^{k-2}, P^i\right\} \prec_L P^{k-1}$ will

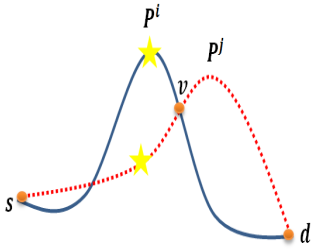

Figure 3: FIFO problem after obtaining *LS*

be checked. This process will continue until there are only two elements left in *LS* or there is a path that is not linearly dominated by its immediate neighbors.

4.3.5 Validating the FIFO Property. After obtaining all non-dominated paths *LS*, one challenge remains. The paths with different (ds, dd) pairs are expanded independently from each other. Therefore, there is the possibility that a path like P^i could be discarded by some other path P^j in a node v due to the FIFO property of the network. The example below illustrates the existence of such paths in *LS* that invalidate the FIFO property of the network.

Example 4.6. Assume Paths P^i and P^j shown in Figure 3 are part of *LS* and they meet in node v after visiting their respective POIs (shown with star signs in the figure). Consider $TC(P^i) = 30$ and $TC(P^j) = 20$. If path P^i meets node v earlier than P^j, path P^j cannot continue its expansion due to FIFO property of road network. This is because if P^i meets node v earlier, it should arrive at d earlier, however as we see $TC(P^i) > TC(P^j)$. Therefore, path P^j should be removed from *LS* in order to preserve FIFO property.

In order to solve this problem, we propose Algorithm5.

Algorithm 5: Validating FIFO

```
1  function Validate FIFO()
2      forall P^i ∈ LS do
3          forall P^j ∈ LS do
4              forall v^i ∈ P^i do
5                  forall v^j ∈ P^j do
6                      if v^i equals v^j then
7                          if both P^i and P^j have visited
8                          POI before or after v^i then
9                              if AT(P^i) < AT(P^j) &
10                             TC(P^i) > TC(P^j) then
11                                 remove P^i from LS
12                             if AT(P^j) < AT(P^i) &
13                             TC(P^i) > TC(P^j) then
14                                 remove P^j from LS
```

For any pair of full paths P^i and P^j (Lines 2-3), in any node $v = v^i = v^j$ which they share (Lines 4-6), we examine the corresponding arrival time on v and the number of visited POIs before visiting v (Lines 7-8). In case a path has smaller arrival time to v and finishes its trip later, the other path will be pruned out since the path which has arrived at the shared node v earlier, should finish its trip faster (Lines 9-14).

5 EVALUATION

In this section, we discuss how we want to evaluate our previously proposed approach in terms of the requirements(I-IV) presented in the introduction. While the requirements are of two types (functional requirements vs. efficiency requirements), we discuss their evaluation in two different settings. The organization of this section is as follows: First, we discuss the functional requirements (requirements I-III). Then we evaluate the efficiency requirement (requirement IV).

5.1 Theoretical Evaluation of Functional Requirements

In the following, we argue that our proposed approach addresses all aforementioned functional requirements.

I–Correctness and Comprehensiveness. In Sections 4.2.2 and 4.2.3, we prove the correctness of the local and global pruning strategies in our proposed algorithm. Therefore, STACY provides accurate query results. On the other hand, it does not consider only a single criterion for optimality, rather it takes TC and DC into account and it minimizes both costs simultaneously.

II–Generality. Our approach satisfies generality by considering time as the path cost which is not constrained to metric spaces.

III–Flexibility. In order to satisfy this criterion, one needs to deal with time-dependent networks, which STACY does.

5.2 Experimental Evaluation of Efficiency Requirement

In this section, we evaluate the efficiency of our proposed approach (requirement ***IV–Real Time Query Performance***).

Organization. This section is organized as follows: first, we introduce the baseline approach which we use to compare our proposed method introduced in Section 4 to. Next, we present the datasets used, the details of the evaluation procedure and the hardware specification. Finally, we study the effects of different parameters on query processing time (i.e., the length of the preferred path, the density of POIs in the road network and the effects of the linear skyline operator on the result set size).

5.2.1 Baseline Approach. Due to the lack of any prior work to solve STACY queries, we propose the following approach as a baseline to solve STACY queries allowing to evaluate the efficiency of our pruning strategies. The idea is using the modified version of *TDOR* algorithm discussed earlier iteratively in order to generate all possible detour paths.

Since in our problem we need to get along with two criteria namely *TC* and *DC*, the *TDOR* algorithm alone could not provide a solution to our problem directly. However, by having different combinations of nodes v_i^* and v_j^* in P^* as source and destination of detour path (i.e. the path which starts from v_i^* in P^* and after visiting a POI returns back to v_j^* in P^* with $j \geq i$) we are able to find all possible detour paths. To be fair in comparison with the

proposed method, we use lemma 4.1 which guarantees that between any pairs, only the minimum detour path could be in the result.

Once all paths are found, they are ordered by detour cost. We add them to the linear skyline set in case they are not conventionally dominated by the last element in *LS*. When a new path is added to *LS* we make sure that there is not any previously existing path that is linearly dominated by the new one. Such linearly dominated paths will be removed during the process. Finally, the algorithm examines the consistency of the remaining paths in *LS* with respect to the FIFO property of the network. We also argue that the time complexity of the baseline algorithm deteriorates for long paths P^* with length n while it needs to execute *TDOR* algorithm $O(n^2)$ times.

5.2.2 Datasets. We work with the two real-world datasets obtained from OSM[4]. These datasets represent the real road networks of Berlin and Oslo. Table 2 shows statistics of the datasets. As points of interest, we consider having a set of restaurants and cafes in each city.

Table 2: Statistics of the datasets

Dataset	# Nodes	# Edges	#POIs
Berlin	27795	72434	6504
Oslo	8969	20305	813

5.2.3 Experimental Setup. In line with previous work in time-dependent networks [5], for each road, we obtain its corresponding travel costs during the daytime as follows: First, we obtain the minimum of the maximum speeds for all roads, i.e. edges. Then, to create the cost function per edge, we use a normal random variable $X \sim \mathcal{N}(\mu, \sigma^2)$ with its mean μ as the average of the value obtained above and its own maximum speed. The standard deviation for this normal random variable is set to one-fourth of its average (i.e. $\sigma = \frac{\mu}{4}$). Then we sample a random number from the random variable X and assign it as the speed of vehicles in that road. Then we calculate the travel cost of the road, simply by dividing its length to its speed.

To simulate the traffic jam conditions in different hours of the day, we calculate a base cost for travel time of each road and then multiply this cost by different factors to scale up the time needed to traverse the road during busy times. We consider the travel cost between 12 am and 8 am as the base cost. For time intervals 8am-10am, 10am-16pm, 16pm-19pm, 19pm-23pm and 23pm-24pm this cost was multiplied by 1.7, 1.4, 1.9, 1.3 and 1.1 respectively. This is in line with related work [5].

We evaluate the scalability of STACY with respect to different parameters. To quantify the effects of the different parameters separately, we use a fixed default setting of the parameter values, and for each parameter, we evaluate how the processing time changes when only this parameter changes. The parameters and their values are shown in Table 3.

As shown in the table with bold fonts, the default value for n is set to 10. This value is chosen with respect to the average trip length in a city. For example, in Berlin, the average trip length is about 5km [8]. This is equal to the sum of 10 average-length streets (i.e., $500m$).

[4]https://www.openstreetmap.org

Table 3: Parameter settings

Parameter	values
Length of preferred path (n)	5, **10**, 15, 20
Density of POIs (D_p)	1%, **2%**, 5%

Regarding POIs, each of them is mapped to the closest node in the graph. In line with previous work [1], for parameter D_p, we use only a small portion of POIs in order to make the settings independent of the dataset used. In this setting, we only choose POIs randomly, in a way that it only covers a small number of nodes in the graph. We try this with a very small number of POIs (1% of the nodes in the graph) and incrementally evaluate until 5% of the nodes are covered by POIs. This makes the problem more challenging since the number of POIs is much less than the real number of POIs, one needs much more processing time to obtain query answers.

For generality and statistical soundness, we have evaluated our proposed approach with ten thousand queries for each sub-experiment.

System Specification. Our code is in Java and Python. We use a modern machine with 16 CPU cores (2.4GHz) and 132 GB RAM.

5.2.4 Effect of Trip Length n. As one can see in Figure 4, for both datasets the proposed approach is several magnitudes faster than the baseline approach in terms of query processing time. The green horizontal line shows the threshold of near real-time query processing time which we set it to two seconds for all our experiments.. As shown in the figure, for longer preferred paths with $n = 20$, the proposed approach provides the results in 10 seconds, but the baseline approach needs 58 seconds which is approximately six times slower than our proposed approach.

Also, as the length of preferred path n increases, the baseline approach slows down with a much higher rate than the proposed approach. This is because the baseline approach needs to call the TDOR algorithm $O(n^2)$ times while the proposed approach with the A* search heuristics, directly goes for finding the paths.

On the other hand, as the network size increases, the query processing time tends to be higher. This is because the branching factor of A^* algorithm increases as the network size grows and this causes the search space to be much bigger.

5.2.5 Effect of Density of POIs D_p. The results of D_p parameter are shown in Figure 5. In both datasets, as the ratio of POIs grows, query processing time decreases, however, the proposed approach is several magnitudes faster than the baseline approach. For example, when the number of nodes containing POIs is 1% in the Berlin dataset, the proposed approach needs only 1.3 seconds to answer the queries, but the baseline approach provides the results in 12.2 seconds which is approximately 9 times slower. For the Oslo dataset, this is 1.7 seconds for the proposed approach while 4 seconds for the baseline approach. This difference is because as the frequency of POIs grows, it gets much easier for both algorithms to find full paths and hence the paths can be generated faster.

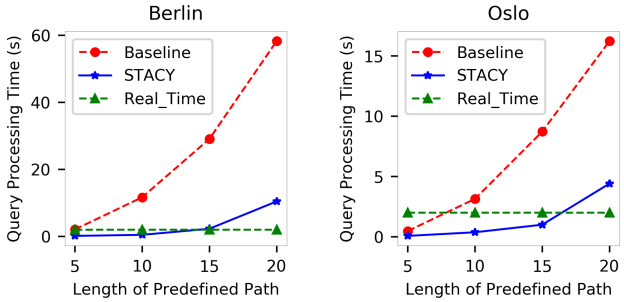

Figure 4: Effects of parameter n

Also as one can see, when the network size grows the query processing time increases. This is because of the branching factor of the A^* algorithm which mentioned earlier.

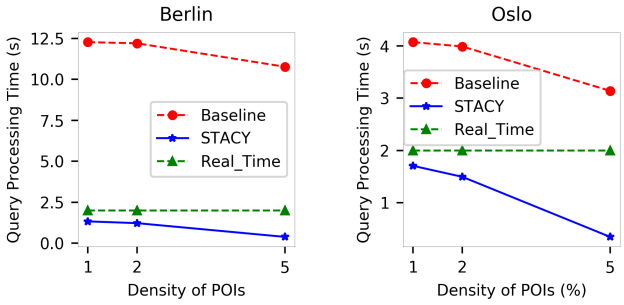

Figure 5: Effects of Parameter D_p

5.2.6 Effect of Linear Skyline Operator. As mentioned earlier, the linear skyline operator is used in order to get smaller number of items in the query result set, so that the user can select the best option without getting puzzled. Therefore, we evaluate the effect of linear skyline operator as follows: we calculate the ratio of the paths in linear skyline set to the number of paths generated by the algorithm using default parameter settings for both datasets. These numbers are 7% and 13% for Berlin and Oslo dataset respectively. As one can see, the linear skyline operator filters out 93% of irrelevant paths in Berlin dataset and 87% in Oslo dataset. Also, note that the maximum size of the query result set in Berlin and Oslo datasets were 6 and 4 respectively. So, the user can select the one she prefers among the small number of items in the query result set.

6 CONCLUSIONS AND FUTURE WORKS

Even though the users of road networks tend to catch up with their favorite routes during their daily lives, but not all the places they need to visit are on their favorite route. Therefore, it happens a lot that they need to deviate from their regular route to visit the places. Since their regular route is the most familiar to them, they would like most to be on their favorite routes. However, this may cause them to spend more travel cost to visit their target places and come back to their favorite route. In fact, this is a two-variable optimization problem in which the users tend to minimize their travel costs as well as their detour costs. On the other hand, time plays an essential role when we are studying road networks while the travel cost of each road depends heavily on the time of day.

Having time as the measure of the travel cost, our first contribution is proposing an efficient algorithm named STACY which solves the aforementioned optimization problem by means of travel cost and detour cost. In other words, having a time-dependent road network, a user with a favorite path and a set of POIs which the user is interested in visiting at least one of them, we find all appropriate paths which the user might be interested in. To this end, we have proposed a solution with guaranteed upper bound for detour cost which shrinks the search space efficiently. As a result, the processing time of this query type is improved significantly.

Our second contribution is to evaluate the goodness of STACY with respect to different parameter settings. In order to do this, we compare the processing time of queries with a baseline approach. We show that STACY is several magnitudes better than the baseline approach in terms of processing time.

REFERENCES

[1] Elham Ahmadi, Camila F Costa, and Mario A Nascimento. 2017. Best-compromise in-route nearest neighbor queries. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 41.

[2] Saad Aljubayrin, Zhen He, and Rui Zhang. 2015. Skyline trips of multiple POIs categories. In *International Conference on Database Systems for Advanced Applications*. Springer, 189–206.

[3] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proceedings of the 17th International Conference on Data Engineering*. IEEE, 421–430.

[4] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. 2009. Monitoring path nearest neighbor in road networks. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 591–602.

[5] Camila F Costa, Mario A Nascimento, José AF Macêdo, Yannis Theodoridis, Nikos Pelekis, and Javam Machado. 2015. Optimal time-dependent sequenced route queries in road networks. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 56.

[6] Lívia A Cruz, Mario A Nascimento, and José AF de Macêdo. 2012. K-nearest neighbors queries in time-dependent road networks. *Journal of Information and Data Management* 3, 3 (2012), 211.

[7] Jian Dai, Bin Yang, Chenjuan Guo, and Zhiming Ding. 2015. Personalized route recommendation using big trajectory data. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 543–554.

[8] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. 2009. BerlinMOD: a benchmark for moving object databases. *The VLDB Journal* 18, 6 (2009), 1335–1368.

[9] Xuegang Huang and Christian S Jensen. 2004. In-route skyline querying for location-based services. In *International Workshop on Web and Wireless Geographical Information Systems*. Springer, 120–135.

[10] Arzoo Katiyar et al. 2014. Efficient and effective route planning in road networks with probabilistic data using skyline paths. In *IKDD*. ACM.

[11] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route skyline queries: A multi-preference path planning approach. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 261–272.

[12] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. 2018. Finding top-k shortest paths with diversity. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (2018), 488–502.

[13] Ernesto Queiros Vieira Martins. 1984. On a multicriteria shortest path problem. *European Journal of Operational Research* 16, 2 (1984), 236–245.

[14] Shiladitya Pande et al. 2017. SkyGraph: retrieving regions of interest using skyline subgraph queries. *VLDB* (2017).

[15] Yuya Sasaki, Yoshiharu Ishikawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2018. Sequenced Route Query with Semantic Hierarchy. In *Proceedings of EDBT 2018, Vienna, Austria, March 26-29, 2018*. 37–48. https://doi.org/10.5441/002/edbt.2018.05

[16] Shuo Shang, Ke Deng, and Kexin Xie. 2010. Best point detour query in road networks. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 71–80.

[17] Mehdi Sharifzadeh, Mohammad Kolahdouzan, and Cyrus Shahabi. 2008. The optimal sequenced route query. *VLDB Journal* 17, 4 (2008), 765–787.

[18] Michael Shekelyan, Gregor Jossé, and Matthias Schubert. 2015. Linear path skylines in multicriteria networks. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 459–470.
[19] Michael Shekelyan, Gregor Jossé, Matthias Schubert, and Hans-Peter Kriegel. 2014. Linear path skyline computation in bicriteria networks. In *International Conference on Database Systems for Advanced Applications*. Springer, 173–187.
[20] Shashi Shekhar and Jin Soung Yoo. 2003. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *Proceedings of the 11th ACM international symposium on Advances in geographic information systems*. ACM, 9–16.
[21] Yuan Tian, Ken CK Lee, and Wang-Chien Lee. 2009. Finding skyline paths in road networks. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 444–447.
[22] Bin Yang, Chenjuan Guo, Yu Ma, and Christian S Jensen. 2015. Toward personalized, context-aware routing. *The VLDB Journal—The International Journal on Very Large Data Bases* 24, 2 (2015), 297–318.

A PROOF OF LEMMA 4.1

Proof. Consider two paths P and P' sharing same exit and entrance nodes v_i^* and v_j^* respectively. Consider sub-path P^*_{s,v_i} with corresponding travel cost $TC(P^*_{s,v_i}, t)$. On the other hand, according to lemma conditions we have $DC(P,t) < DC(P',t)$. This means that travel cost of the detour path $P_{v_i^*,v_j^*}$ has greater travel cost than $P'_{v_i^*,v_j^*}$. Therefore we have $TC(P_{s,v_j^*}, t) < TC(P'_{s,v_j^*}, t)$ and therefore P arrives at v_j^* earlier than P'. Since this is a FIFO network, we have $TC(P_{v_j^*,d}, t) < TC(P'_{v_j^*,d}, t)$ and therefore $TC(P_{s,d}, t) < TC(P'_{s,d}, t)$. Having $DC(P,t) < DC(P',t)$ alongside the obtained inequality shows that path P dominates path P' using skyline operator and hence P' could not be part of the skyline set. □

B PROOF OF LEMMA 4.3

Proof. We prove this lemma by contradiction. Assume $Fastest^{TravelCost} = \langle s = v_1, \ldots, v_i, \ldots, d = v_n \rangle$ and $P' = \langle s = v'_1, \ldots, v'_j, \ldots, d = v'_m \rangle$ having met each other at node $v_i = v'_j$ during their traversal and travel cost of the partial path $P'_{1,j} = \langle s = v'_1, \ldots, v'_j \rangle$ is less than travel cost of partial path $P^{TC}_{1,i} = \langle s = v_1, \ldots, v_i \rangle$. We also assume that they have visited same number of POIs before reaching to node $v_i = v'_j$ (i.e. they have either visited a POI or they have not visited any POI). Since the network is FIFO, then $P^{TC}_{1,i}$ cannot reach to destination d earlier than $P'_{1,j}$. Therefore, we have $TC(Fastest^{TravelCost}) > TC(P')$, and this is in contradiction with our main assumption that $Fastest^{TravelCost}$ is the fastest path. So no path P' can discard $Fastest^{TravelCost}$, and it will be part of the query result. □

C PROOF OF LEMMA 4.4

Proof. Let P^i and P^j be two paths from s to v and assume that $TC(P^i) < TC(P^j)$ and that P^i contains more POIs than P^j. Consider, for example, that P^j has passed by POIs belonging to the categories $COI_1, \ldots, COI_k$ and P^i passed by the categories $COI_1, \ldots, COI_{k+1}$. P^i needs to be extended with a path from v that passes by $COI_{k+2}, \ldots, COI_m$ and finally goes to d. Let P^{min} be the path with minimum travel cost satisfying these conditions. Similarly, P^j needs to be extended with a path from v that passes by $COI_{k+1}, \ldots, COI_m$ and finally goes to d. Let P^b be the path with minimum cost satisfying these conditions. Note that since P^b needs to visit more POIs than P^a (and they both are paths from v), then $TC(P^a) < TC(P^b)$. Since by assumption $TC(P^i) < TC(P^j)$, then $TC(P^i) + TC(P^a) < TC(P^j) + TC(P^b)$ also holds. Therefore, it is not worth expanding P^j, since P^i leads to better paths (in terms of travel cost). □

D PROOF OF LEMMA 4.5

Proof. We differentiate between two cases that may occur:

- If P^i is a full detour path with (ds, dd) we define $P^j = P^*_{s,ds} \| P^i \| P^*_{dd,d}$ as a full path containing P^i. Since the detour path P^i is a sub-path of P^j, we have $DC(P^i) = DC(P^j) = ODC(P^i)$. By definition we have $ODC(P^i) > DetourCost^{Upper}$, thus we can infer $DC(P^j) > DetourCost^{Upper} = DC(Fastest^{TravelCost})$. On the other hand, $Fastest^{TravelCost}$ is defined as the fastest path, so we have $TC(Fastest^{TravelCost}) < TC(P^j)$. Therefore P^j is worse than $Fastest^{TravelCost}$ in terms of travel cost and detour cost and thus we can prune P^i immediately.
- If P^i is a partial detour path with (ds, dd) we take P^k with (ds, dd) to be a full detour path with minimum detour cost obtained by extending partial path P^z in the queue. Since P^i is dequeued first we have $ODC(P^z) > ODC(P^i)$. Again according to definition $ODC(P^i) > DetourCost^{Upper}$. Thus we can conclude $ODC(P^z) > DetourCost^{Upper}$. Then we define $P^j = P^*_{s,ds} \| P^z \| P^*_{dd,d}$ as a full path containing P^z. Since the detour path P^z is a sub-path of P^j, we have $DC(P^z) = DC(P^j) = OTD(P^z)$. By definition we have $ODC(P^z) > DetourCost^{Upper}$, thus we can infer $DC(P^j) > DetourCost^{Upper} = DC(Fastest^{TravelCost})$. On the other hand, $Fastest^{TravelCost}$ is defined as the fastest path, so we have $TC(Fastest^{TravelCost}) < TC(P^j)$. Therefore P^j is worse than $Fastest^{TravelCost}$ in terms of travel cost and detour cost and thus we can prune P^i immediately.

Once we have proven the network expansion for detour paths with same (ds, dd) pair won't result in non-dominated paths, it is trivial to extend it to the paths with different (ds, dd) than that of P^i. □