# Path-based Queries on Trajectory Data

Benjamin Krogh
Dept. of Computer Science
Aalborg University, Denmark
bkrogh@cs.aau.dk

Nikos Pelekis
Dept. of Statistics and
Insurance Science
University of Piraeus, Greece
npelekis@unipi.gr

Yannis Theodoridis
Dept. of Informatics
University of Piraeus, Greece
ytheod@unipi.gr

Kristian Torp
Dept. of Computer Science
Aalborg University, Denmark
torp@cs.aau.dk

## ABSTRACT

In traffic research, management, and planning a number of path-based analyses are heavily used, e.g., for computing turn-times, evaluating green waves, or studying traffic flow. These analyses require retrieving the trajectories that follow the full path being analyzed. Existing path queries cannot sufficiently support such path-based analyses because they retrieve all trajectories that touch *any* edge in the path. In this paper, we define and formalize the *strict path query*. This is a novel query type tailored to support path-based analysis, where trajectories must follow *all* edges in the path. To efficiently support strict path queries, we present a novel NETwork-constrained TRAjectory index (NETTRA). This index enables very efficient retrieval of trajectories that follow a specific path, i.e., strict path queries. NETTRA uses a new path encoding scheme that can determine if a trajectory follows a specific path by only retrieving data from the first and last edge in the path. To correctly answer strict path queries existing network-constrained trajectory indexes must retrieve data from *all* edges in the path. An extensive performance study of NETTRA using a very large real-world trajectory data set, consisting of 1.7 million trajectories (941 million GPS records) and a road network with 1.3 million edges, shows a speed-up of two orders of magnitude compared to state-of-the-art trajectory indexes.

## Categories and Subject Descriptors

H.2.8 [**Information Systems**]: Database Applications—*Spatial databases and GIS*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms, Network problems*

## General Terms

Design, Performance, Algorithms, Experimentation

## 1. INTRODUCTION

Increasingly large quantities of high frequency GPS data are being collected from vehicles. Such data makes it possible to reconstruct the exact *path* followed, e.g., each turn made and each road segment passed. Taking the path of trajectories from vehicles into consideration, enables a plethora of important use cases such as: traffic anomaly detection [14], studies of multi-edge effects, e.g., turn-time costs, studies of intersection coordination, e.g., "green-waves" [9], investigative analysis, e.g., identifying a vehicle that followed a specific path (within a specific time interval), and for determining the most frequently used path in a set of paths to improve route planning.

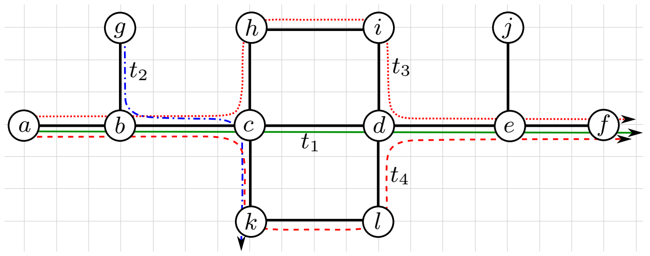

**Figure 1: Network and trajectories**

These use cases require finding the set of trajectories that follow a specific path through a network. As an example, consider computing the average travel time of the path from vertex $a$ to vertex $f$ following the edges $e_{ab}$, $e_{bc}$, $e_{cd}$, $e_{de}$, and $e_{ef}$ in Figure 1 (edges are directed and labeled using the notation $e_{from-vertex\ to-vertex}$). Intuitively, this travel time can be computed by retrieving the trajectories that follow the entire path, and taking the time difference between entering the path (at vertex $a$) and leaving it again (at vertex $f$). In Figure 1 this is only trajectory $t_1$.

The path queries discussed in [5, 16] are *range queries*. These queries retrieve all trajectories that *intersect* a given path. Thus, the path query retrieves the trajectories that touch *any* edge on the path, i.e., retrieves all four trajectories in Figure 1. This is obviously wrong, because the trajectory $t_2$ touch only a single edge ($e_{bc}$) on the path and is therefore not usable for computing the average travel time of the path from vertex $a$ to vertex $f$. Similarly, the trajectories $t_3$ and $t_4$ are not usable because they contain detours.

The path query discussed in this paper differs significantly from the path query discussed in related works. In this paper the trajectories returned follow the entire path from its beginning to its end without any detours. To distinguish between path queries in the related work and the path queries proposed here, we name our variation the *Strict Path Query* (SPQ).

We propose the NETwork-constrained TRAjectory index (NETTRA) for very efficient retrieval of trajectories that follow a specific path. The basic idea behind NETTRA, is to represent each trajectory as a sequence of touched network edges. For each edge touched, an entry is stored that contains the edge id, enter and leave time, and a novel encoding of the entire path up to and including the edge itself. Using only the novel path encoding for the first and last edge on a query path, it is possible to determine whether the trajectory followed a specific path between these edges.

NETTRA is implemented in standard SQL and is portable between DBMSs. We perform extensive experiments using a real-world trajectory data set containing 941 million GPS records, 1.7 million trajectories, and the transportation network for Denmark (1.3 million edges). Using NETTRA, we demonstrate speed-ups over state-of-the-art of up to two orders of magnitude for SPQs. To the best of our knowledge, NETTRA is the first solution that efficiently supports path-based trajectory analysis using very large trajectory data sets, on large country-sized networks.

The main contributions of this paper are the following.

- The novel strict path query (SPQ) type is introduced and formalized. This query type is essential to the traffic domain.
- An exact solution for answering SPQs. This solution only needs to retrieve trajectory data for a few edges in the SPQ.
- A practical-exact solution answering SPQ. This solution only retrieves trajectory data for the first and last edges in the SPQ.
- A thorough performance study of NETTRA using a large real-world trajectory data set.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 introduces the context and formalizes the queries. Section 4 introduces the NETTRA index. Section 5 discusses implementation details. Section 6 evaluates NETTRA experimentally. Finally, Section 7 concludes the paper.

## 2. RELATED WORK

The related work can be divided into network constrained indexing techniques where movement is constrained by an underlying network and string matching techniques. String matching is included because a SPQ can be considered as finding exact matches of a pattern (the SPQ) in a large text corpus (the paths of all trajectories).

### 2.1 Network Constrained Indexing

From a network constrained perspective, [4, 5, 7, 15, 16] give a thorough treatment on indexing and querying moving objects in a network. However, the path query previously considered does not efficiently support the retrieval of trajectories that follow a specific path through a network. Because of this, many path-based analyses are very difficult and inefficient to perform.

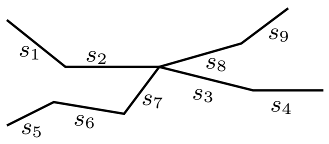

**Figure 2: Transportation network abstractions.**

In the related work, the transportation network is represented using either the segment, the edge, or the route network abstraction. A *segment* is a straight line between two spatial locations, an *edge* is a polyline between two intersections, and a *route* is a complete road, which may cross many intersections. The small transportation network in Figure 2 illustrates this distinction. The network consists of the segments, $s_1, \ldots, s_9$, or the edges $e_1 = (s_1, s_2)$, $e_2 = (s_3, s_4)$, $e_3 = (s_5, s_6, s_7)$, and $e_4 = (s_8, s_9)$, or the routes $r_1 = (s_1, s_2, s_3, s_4)$ and $r_2 = (s_5, s_6, s_7, s_8, s_9)$.

The MON-tree, presented in [5], supports segment, edge, and route abstractions. A top level 2D R-tree is used to index the transportation network, with a forest of leaf-level 2D R-trees indexing the movement along the polylines. Additionally, a top level hash structure is used to map polylines to the corresponding leaf-level 2D R-tree to speed-up insertion. When evaluating a SPQ, the hash structure can be used to directly map each edge in the query to the corresponding leaf-level 2D R-tree. All trajectories that follow the entire path can be retrieved by examining the R-tree for each edge on the path queried. The shortcomings of the MON-tree are the following: the actual path of each candidate trajectory needs to be retrieved in order to remove trajectories with detours. A route-based MON-tree may simplify some SPQs, e.g., when a SPQ is contained within a route. This is rarely the case, however, for instance when a SPQ includes turns, or when roads merge/diverge.

The PARINET approach [16] is to the best of our knowledge state-of-the-art in network-constrained trajectory indexing. PARINET supports the segment, edge, and route abstractions, with practically identical implementations. For this reason, we only describe it using the edge abstraction. Each edge is assigned a weight corresponding to the number of trajectories that touch that edge. A graph partitioning algorithm then partitions the network (and data) such that each partition has a similar accumulated weight (and size). The resulting index is thereby tuned to the specific data set. The on-disk format for a partition is shown Table 1. There is a B$^+$tree on the $t_{enter}$ column. By grouping multiple edges into one on-disk structure, PARINET reduces the number of random I/Os for range and path queries. It is straightforward to insert a trajectory $t$ into PARINET: for each edge touched an entry is inserted into the relevant partition.

| Column | Description |
|---|---|
| $t_{id}$ | Trajectory identifier |
| $e_{id}$ | Edge identifier |
| $t_{enter}$ | Time then trajectory $t_{id}$ entered edge $e_{id}$ |
| $t_{leave}$ | Time then trajectory $t_{id}$ left edge $e_{id}$ |

**Table 1: The on-disk format of PARINET**

Executing a SPQ using PARINET is similar to using a MON-tree. First, we retrieve the trajectories that touch each edge in the SPQ. Next, we filter out trajectories with detours by retrieving the full path of each candidate trajectory or query the edges adjacent to the SPQ. Because multiple edges are stored in the same partition on disk, fewer partitions are usually queried than the number of edges in the SPQ.

## 2.2 String Matching Techniques

String matching techniques can be used to execute SPQs, by first converting path, $\pi = [e_1, e_2, \ldots, e_n]$, of each trajectory $t$ into a textual representation, $s_t$. Next, the strings of all trajectories are concatenated into $S$, and separated using a special marker, e.g., $S = s_{t_1} + '\$' + s_{t_2} + \ldots$, where the + operator concatenates two strings. A SPQ is then executed by converting the SPQ into its textual representation, $s$, and retrieving all exact matches of $s$ within $S$.

Suffix trees or the more space-efficient suffix arrays [10, 12] have optimal search time complexity for finding exact matches of a search string $s$ in a large text corpus $S$. Due to the similarity, we use suffix trees to refer to either suffix trees or suffix arrays in the rest of the paper. Suffix trees require only $O(|s| + k)$ time for finding all instances of s, where $|s|$ is the number of characters in the search string and $k$ is the number of occurrences of $s$ in $S$.

Given that suffix trees have optimal search time complexity, they may appear optimal for evaluating SPQs. There are, however, a number of significant drawbacks to suffix trees: (1) they have significant main-memory requirements, (2) they do not support temporal filtering, and (3) they are difficult to update, and (4) they do not support concurrent update and read operations. In general we do not assume that the entire trajectory data-set can be loaded into main-memory, and we therefore consider the very high memory requirements of suffix trees prohibitive.

Recent approaches such as [11], reduce the main-memory requirements of suffix tree construction for texts up to 4 GB. Unfortunately, these algorithms require preprocessing the full text corpus $S$ prior to constructing the suffix array, and cannot be updated. Because trajectory data-sets are frequently updated with more trajectories, the lack of update support makes suffix trees ill-suited as a trajectory index. In addition, temporal filtering is not possible using a suffix tree. This lacking support for temporal filtering is a major drawback within the traffic domain because it is necessary to, e.g., distinguish between peak and non-peak hours.

## 3. PRELIMINARIES

This section first describes the data model used for representing the network and the trajectory data. Next, we proceed to formalize both the plain path query and the SPQ.

## 3.1 Data Model

NETTRA is designed for use with the segment and edge network abstractions. NETTRA is implemented identically independent of network abstractions, and for this reason we only describe NETTRA with the edge abstraction.

The network consists of edges and is represented as a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is a set of vertices and $\mathbf{E}$ is a set of directed edges, $\mathbf{E} = \mathbf{V} \times \mathbf{V}$.

A Moving Object (MO) reports its position periodically. Each location update is a tuple $loc = (moid, ts, pos)$, where $moid$ is an identifier, $ts$ is a time-stamp, and $pos$ is the spatial position of the MO at time $ts$.

Any network constrained indexing technique requires that the location updates of the MO is map-matched to the network, i.e., that the route of the MO is identified. Map-matching transform the input sequence of spatial location updates into network constrained location updates. Each of these updates is a tuple, $loc_{mm} = (t_{id}, e_{id}, ts_{enter}, ts_{leave})$, where $t_{id}$ is a trajectory identifier, $e_{id}$ is an edge identifier, $ts_{enter}$ and $ts_{leave}$ is the time at which $t_{id}$ entered and left edge $e_{id}$, respectively.

A *trajectory*, $t = [loc_{mm_1}, loc_{mm_2}, \ldots, loc_{mm_n}]$, is a logical sequence of network constrained location updates during the course of one trip. For a convenient notation, the function $path(t)$ returns the path followed by trajectory $t$, and the functions $entertimes(t)$ and $leavetimes(t)$ return the ordered list with enter and leave times of each edge, respectively. Finally, the set of trajectories, $\mathbf{T} = \{t_1, t_2, \ldots, t_n\}$, contains all trajectories from all moving objects.

Trajectories are usually not updated after being inserted. However, additional trajectories are frequently added to the database, e.g., every night, and the index must therefore be able to handle inserts efficiently.

## 3.2 Problem Definition

The *Strict Path Query* (SPQ) is formalized in Equation 1. It returns all trajectories that strictly follow the path, $\pi$, through the transportation network, within a temporal interval defined by $start$ and $end$. The function $SubList(path(t), i, j)$ returns the sub-list of $path(t)$, $t \in \mathbf{T}$, between index $i$ (inclusive) and index $j$ (exclusive), $i < j$. Thus the result of a SPQ consists of each trajectory from $\mathbf{T}$, that has $\pi$ as its sub-path, enters $\pi$ at a time instance equal to or later than $start$, and leaves $\pi$ at a time instance equal to or later than $end$.

$$\begin{aligned} SPQ(\pi, start, end) \;&=\; \\ &\{t \mid t \in \mathbf{T}, \exists i, j : \pi = SubList(path(t), i, j+1) \\ &\wedge\; start \;\leq\; entertimes(t)[i] \\ &\wedge\; leavetimes(t)[j] \;\leq\; end\} \end{aligned} \tag{1}$$

A SPQ is very different from the plain path query defined in [16]. To clarify the differences, we have formalized the plain path query in Equation 2. Similarly to a SPQ, a plain path query takes as arguments a path $\pi$, and a temporal interval denoted by $start$ and $end$. For a trajectory to satisfy the constraints of a plain path query, the trajectory only needs to visit an edge in the query path $\pi$, within the temporal interval.

$$\begin{aligned} PQ(\pi, start, end) \;&=\; \\ &\{t \mid t \in \mathbf{T}, \exists (e, ts_{enter}, ts_{leave}) \in t : e \in \pi \\ &\wedge (start \leq ts_{enter} \leq end \\ &\vee start \leq ts_{leave} \leq end)\} \end{aligned} \tag{2}$$

Because a SPQ is more strict in the evaluation of both the spatial and temporal components it always holds that $SPQ(\pi, start, end) \subseteq PQ(\pi, start, end)$. This is also evident when considering the path $\pi = [e_{ab}, e_{bc}, e_{cd}, e_{de}, e_{ef}]$ in Figure 1. For this path a plain path query returns all four trajectories, whereas a SPQ returns only the trajectory $t_1$. Note that sub-trajectories are used, e.g., the SPQ on path $\pi = [e_{ab}, e_{bc}]$ returns the trajectories $t_1$, $t_3$, and $t_4$.

## 4. THE NETTRA INDEX

In this section, we present the NETTRA index. The basic idea is to apply an encoding to the path of each trajectory and to use this encoding to determine whether a trajectory follows a specific path through the network.

We present three solutions for evaluating SPQs that each uses the same path encoding. Two solutions are exact and one solution is practically exact. The latter solution may include false positives in the result set (but never false negatives). However, the probability of false positives is extremely low, see Section 6.

For a SPQ on the path, $\pi = e_{ab}$, $e_{bc}$, $e_{ch}$, $e_{hi}$, $e_{id}$, $e_{de}$, and $e_{ef}$ in Figure 1, the first exact solution retrieves trajectory data of all edges in the SPQ, i.e., from edges $e_{ab}$, $e_{bc}$, $e_{ch}$, $e_{hi}$, $e_{id}$, $e_{de}$, and $e_{ef}$. The second exact solution usually only needs to retrieve data for a few edges in the SPQ, e.g., edges $e_{ab}$, $e_{id}$, and $e_{ef}$. The practically exact solution only retrieves trajectory data for the first and the last edge of the SPQ, i.e., edges $e_{ab}$ and $e_{ef}$.

### 4.1 Trajectory Representation and Encoding

Similar to [5, 7] and [16], the movement of objects is represented in terms of a network. Here the enter and leave times for each edge touched by a trajectory is stored as shown in the first four columns in Table 2, for the four trajectories in Figure 1. In addition, a *hash* value is computed for each touched edge and stored in the *hash* column in Table 2. Concretely, a weight is assigned to each edge, and the hash value is the prefix-sum of all touched edges of the trajectory. Specifically, the $i^{\text{th}}$ hash value is computed using the function defined in Equation 3. The equation uses a list of edges, $\pi = [e_1, e_2, \ldots, e_n]$ to model the path of a trajectory. The edge at position 1 is the first edge on the path, the edge at position 2 is the second edge on the path, and so on.

$$hash(\pi, i) = \sum_{j=1}^{i} \pi[j].weight \tag{3}$$

Table 2 shows the hash values for the four trajectories in Figure 1. The edge length is used as a weight and each square is assumed to have length one. In this case, $hash(t_1, 1)$ = 3, $hash(t_1, 2) = 7$, and $hash(t_1, 3) = 11$.

In general, we assume that the weight assigned to each edge is a non-zero positive value. For now, the weight is the length of the edge; a different weight configuration is discussed in Section 4.4.

Except for the *hash* column, the data model used in this paper is very similar to the data model used in [5, 7] and [16]. However, we will in Section 6 show that this addition has a huge positive impact on the performance.

#### 4.1.1 Querying

Given a candidate set of trajectories that touch the first and last edge of a path, we can use the values in the *hash* column in a filtering step. Concretely, we check that the difference in the hash column between entering and leaving the path, match the change determined by the path.

For each candidate trajectory we evaluate the expression in Equation 4. $hash_{start}$ and $hash_{end}$ are the hash values of a trajectory on the first and last edge of $\pi$, respectively. The *deltahash* function, in Equation 5, returns the change in the hash column for a given path $\pi$. In the equation, $n$, is the number of edges in $\pi$. If the expression in Equation 4 is false, the candidate trajectory necessarily followed some other path than $\pi$.

| $t_{id}$ | $e_{id}$ | $ts_{enter}$ | $ts_{leave}$ | $hash$ |
|---|---|---|---|---|
| $t_1$ | $e_{ab}$ | 9 | 11 | 3 |
| $t_1$ | $e_{bc}$ | 11 | 13 | 7 |
| $t_1$ | $e_{cd}$ | 13 | 17 | 11 |
| $t_1$ | $e_{de}$ | 17 | 21 | 15 |
| $t_1$ | $e_{ef}$ | 21 | 23 | 18 |
| $t_2$ | $e_{gb}$ | 14 | 16 | 3 |
| $t_2$ | $e_{bc}$ | 16 | 19 | 7 |
| $t_2$ | $e_{ck}$ | 19 | 21 | 10 |
| $t_3$ | $e_{ab}$ | 14 | 16 | 3 |
| $t_3$ | $e_{bc}$ | 11 | 13 | 7 |
| $t_3$ | $e_{ch}$ | 13 | 15 | 10 |
| $t_3$ | $e_{hi}$ | 15 | 18 | 14 |
| $t_3$ | $e_{id}$ | 18 | 21 | 17 |
| $t_3$ | $e_{de}$ | 21 | 24 | 21 |
| $t_3$ | $e_{ef}$ | 24 | 27 | 24 |
| $t_4$ | $e_{ab}$ | 14 | 16 | 3 |
| $t_4$ | $e_{bc}$ | 16 | 20 | 7 |
| $t_4$ | $e_{ck}$ | 20 | 22 | 10 |
| $t_4$ | $e_{kl}$ | 22 | 25 | 14 |
| $t_4$ | $e_{ld}$ | 25 | 28 | 17 |
| $t_4$ | $e_{de}$ | 28 | 30 | 21 |
| $t_4$ | $e_{ef}$ | 30 | 33 | 24 |

**Table 2: The `visitedsegments` table**

$$hash_{end} - hash_{start} = deltahash(\pi) \tag{4}$$

$$deltahash(\pi) = \sum_{j=2}^{n} \pi[j].weight \tag{5}$$

To demonstrate the filtering step, we show how to evaluate a SPQ, using the `visitedsegments` table shown in Table 2. Specifically, we consider the SPQ over the path $\pi = [e_{ab}, e_{bc}, e_{cd}, e_{de}, e_{ef}]$, in Figure 1 (we ignore the temporal constraint for now). Only the trajectory $t_1$ should be returned by this query, because the other trajectories have detours or do not follow the entire path. We first compute the candidate set of trajectories that touch both the first and last edges of $\pi$, i.e., $e_{ab}$ and $e_{ef}$. Using Table 2, we find that this candidate set contains the trajectories $t_1$, $t_3$, and $t_4$. The trajectory $t_1$ has $hash_{start} = 3$ and $hash_{end} = 18$. Both trajectories $t_3$ and $t_4$ have $hash_{start} = 3$ and $hash_{end} = 24$.

For each trajectory in the candidate set, we then evaluate the expression in Equation 4, i.e., $hash_{end} - hash_{start} = deltahash([e_{ab}, e_{bc}, e_{cd}, e_{de}, e_{ef}])$. Concretely, this is true for $t_1$, i.e., $18 - 3 = 4 + 4 + 4 + 3$, but false for both $t_3$ and $t_4$, i.e., $24 - 3 \neq 4 + 4 + 4 + 3$. After this step, it is clear that the trajectories $t_3$ and $t_4$ have followed some other path, and they are consequently removed from the candidate set.

Finding a candidate trajectory set and applying this filtering is easy to define in SQL, as shown in Listing 1. The functions $first$ and $last$ return the first and last edge of a path, respectively.

```
select  $e_{start}$.tid
from    visitedsegments as $e_{start}$
join    visitedsegments as $e_{end}$ using (tid)
where   $e_{start}$.eid  = first($\pi$)
and     $e_{end}$.eid    = last($\pi$)
and     $e_{end}$.hash - $e_{start}$.hash = deltahash($\pi$)
```

**Listing 1: Query to find a candidate trajectory set.**

Note that the query in Listing 1 can be answered by only retrieving trajectory data for the first and last edges of the path. In most cases, the query in Listing 1 removes all false positives from the candidate set (see Section 6.2). However, trajectories that follow a different path $\pi'$, from the first edge of $\pi$ to the last edge of $\pi$, and have $deltahash(\pi) = deltahash(\pi')$ cannot be removed by this approach. As an example, using the length as the weights on the edges we cannot distinguish between the path of trajectory $t_3$ and the path of trajectory $t_4$ in Figure 1, even though they clearly follow different paths. We will next show how to resolve this situation.

## 4.2 Exact SPQ Evaluation

We now describe how to refine the candidate set of trajectories such that it only contains true positives. First, we define a naïve approach that guarantees the correct result, but with a high cost in terms of I/Os compared to the query in Listing 1. This approach incurs I/Os on the order of a plain path query from the related work [16].

For a path, $\pi$, that consists of exactly two edges, $deltahash(\pi)$ uniquely identifies $\pi$ between all paths between $first(\pi)$ and $last(\pi)$. Therefore, for any path that consists of two edges the result of the query in Listing 1 is correct and does not need to be refined further. This is formalized in Proposition 1.

Proposition 1. *For any path consisting of two edges, $\pi = [e_{start}, e_{end}]$, the query in Listing 1 retrieves exactly the trajectories that strictly follow $\pi$.*

Proof. To satisfy the query in Listing 1, a trajectory $t$ must follow a path, $\pi_t$, between the first and last edges of $\pi$, such that $deltahash(\pi_t) = deltahash(\pi)$ (Equation 4). Assume, that $\pi_t = [e_{start}, e_2, \dots, e_{n-1}, e_{end}]$. $deltahash(\pi_t) = e_2.weight + \dots + e_{n-1}.weight + e_{end}.weight$. Because, $deltahash([e_{start}, e_{end}]) = e_{end}.weight$, it follows that $deltahash(\pi_t) > deltahash(\pi)$. Because $deltahash(\pi_t) > deltahash(\pi)$, t cannot be returned by the query in Listing 1. □

Proposition 1 can be used to prevent false positives for any path $\pi$, by proving that candidate trajectories follow all sub-paths of $\pi$. Concretely, we convert a path $\pi$ over $n$ edges, into the $n-1$ sub-paths that each consists of exactly two edges. For instance, the path $\pi = [e_1, e_2, e_3, e_4]$ is converted into $\pi_1 = [e_1, e_2]$, $\pi_2 = [e_2, e_3]$, and $\pi_3 = [e_3, e_4]$. First, we use the query in Listing 1 to load a candidate set of trajectories that follow the entire path $\pi$. Next, trajectories that do not follow each of the sub-paths $\pi_1$ to $\pi_{n-1}$, $n = size(\pi)$, are pruned from this candidate set. If a trajectory, $t$, has the hash values $h_1, h_2, h_3, h_4$ on edges $e_1, e_2, e_3, e_4$, respectively, and $h_2 - h_1 = deltahash(\pi_1)$, then $t$ must have followed the sub-path $\pi_1$ (guaranteed by Proposition 1). If not, we prune $t$ from the candidate trajectory set. Similarly, if $h_{i+1} - h_i = deltahash(\pi_i)$, we know that $t$ follows the sub-path $\pi_i$. As such, we can prove that $t$ follows the full path $\pi$.

Extending the query in Listing 1 with this incremental pruning approach results in the query in Listing 2. This query has a self-join for each sub-path of the SPQ. Each join adds overhead, in that for each $\pi_i$ of the $n-1$ sub-paths, the trajectories that touch the edge $last(\pi_i)$ will be retrieved.

```
select  $e_{start}$.tid
from    visitedsegments as $e_{start}$
join    visitedsegments as $e_{end}$ using (tid)
join    visitedsegments as $e_2$ using (tid)
...
join    visitedsegments as $e_{n-1}$  using (tid)
where   $e_{start}$.eid = first($\pi$)
and     $e_{end}$.eid = last($\pi$)
and     $e_2$.eid = last($\pi_1$)
 ...
and     $e_{n-1}$.eid = last($\pi_{n-1}$)
and     $e_{end}$.hash = $e_{start}$.hash + deltahash($\pi$)
and     $e_2$.hash = $e_{start}$.hash + deltahash($\pi_1$)
and     $e_3$.hash = $e_2$.hash + deltahash($\pi_2$)
 ...
and     $e_{end}$.hash = $e_{n-1}$.hash + deltahash($\pi_{n-1}$)
```

**Listing 2: Exact strict path query**

The plain path query considered in [16] retrieves all trajectories that touch each edge of $\pi$. As such, the approach in Listing 2 requires approximately the same number of I/Os as the plain path query in the related work. However, this approach does not consider the fact that the network constrains the movement of moving objects. In the next section we describe how we, by considering the network, can reduce the number of sub-paths and I/Os significantly.

## 4.3 Optimized Exact SPQ Evaluation

We now show that for some paths the query in Listing 1 is guaranteed to return exactly the trajectories that strictly follow a specific path $\pi$, independently of the number of edges in $\pi$. This implies that all additional self-joins can be omitted, without compromising correctness. Note that the query in Listing 1 returns false positives *only* when a different path, $\pi'$, exists such that $deltahash(\pi) = deltahash(\pi')$, $first(\pi) = first(\pi')$, and $last(\pi) = last(\pi')$. If such a path $\pi'$ does *not* exist, we say that $\pi$ is *unique*. When $\pi$ is unique, the query in Listing 1 is guaranteed to retrieve only the trajectories that strictly follow $\pi$. This is very valuable because the query in Listing 1 retrieves trajectory data from only the two edges $first(\pi)$ and $last(\pi)$, independently of the number of edges in $\pi$.

It is *always* the case that $\pi$ is unique when $\pi$ match exactly the shortest path between $first(\pi)$ and $last(\pi)$, and there is exactly one shortest path between $first(\pi)$ and $last(\pi)$. In such cases, any other path, $\pi'$, will always be longer than $\pi$, i.e., $deltahash(\pi) < deltahash(\pi')$. Note that, existing algorithms such as Hub Labeling (HL) [2], can very efficiently determine if $\pi$ is a shortest path and if there are more than one shortest path. As such, HL can be used to determine whether $\pi$ is a unique path. The problem we are left with is when $\pi$ is not a unique shortest path. The query listed in Listing 2 could be used to answer the query. However, this query uses $n - 1$ self-joins, where $size(\pi) = n$, and retrieves trajectory data for each edge in $\pi$. This has significant overhead, and the query in Listing 2 is therefore orders of magnitude slower than the query in Listing 1, see Section 6.

Instead of using the query in Listing 2, we split $\pi$ into $k$ sub-paths, $\pi_1, \ldots, \pi_k$, such that each $\pi_i$ is unique, $first(\pi_1) = first(\pi)$, $last(\pi_k) = last(\pi)$, and $last(\pi_i) = first(\pi_{i+1})$. In the best case, $k = 1$, i.e., $\pi$ is unique. In the worst case, $k = n - 1$, where $n = size(\pi)$. This is because Proposition 1 states that any path consisting of exactly two-edges is unique.

By proving that each candidate trajectories follows all unique sub-paths, in turn, all false positives can be pruned from the candidate set. Concretely, we first retrieve a candidate trajectory set using the query in Listing 1. If a trajectory, $t$, has the hash values $h_1, \ldots, h_k, h_{k+1}$, on the edges $first(\pi_1)$, ... , $first(\pi_k)$, $last(\pi_k)$, and $h_2 - h_1 = deltahash(\pi_1)$, then $t$ must follow $\pi_1$. Similarly, if $h_{i+1} - h_i = deltahash(\pi_i)$, it follows that $t$ follows $\pi_i$. By following this strategy for each unique sub-path, we prove that each candidate trajectory follow the entire path $\pi$

We will shortly present a very efficient algorithm for splitting $\pi$ into a number of sub-paths, such that each sub-path $\pi_i$ is unique. We will show in Section 6 that the number of sub-paths required, $k$, on average is less than $size(\pi)/30$. Therefore, the average case is much closer to the best case shown in Listing 1 than the worst case shown in Listing 2.

**Algorithm 1** Find the unique shortest paths of a SPQ.

```
Input: A path $\pi$
Output: The set of unique sub-paths of $\pi$
 1: function UniqueSubPaths($\pi$)
 2:     if $size(\pi) < 3$ then
 3:         return $\pi$
 4:     end if
 5:     $vi \leftarrow 2$
 6:     for $i = 3 \rightarrow size(\pi)$ do
 7:         $sppath \leftarrow Shortest(\pi[1], \pi[i])$
 8:         if $sppath \neq SubList(\pi, 2, i)$ then
 9:             break
10:         end if
11:         $vi \leftarrow i$
12:     end for
13:     if $vi = size(\pi)$ then
14:         return $\pi$
15:     else
16:         $first \leftarrow SubList(\pi, 1, vi + 1)$
17:         $\pi' \leftarrow SubList(\pi, vi, size(\pi) + 1)$
18:         $subpaths \leftarrow$ UniqueSubPaths($\pi'$)
19:         return $\{first\} \cup subpaths$
20:     end if
21: end function
```

The function UniqueSubPaths, shown in Algorithm 1, transforms a path into its unique sub-paths. As input the function takes the path $\pi$. As output it returns the unique sub-paths of $\pi$. In Line 2 to 4 we check if the input path $\pi$ contains less than three edges. In such case, Proposition 1 guarantees that $\pi$ is unique. The function $SubList(\pi, i, j)$ returns the sub-list of $\pi$ between index $i$ (inclusive) and index $j$ (exclusive). $vi$ (Line 5) is an index into $\pi$, where it has been verified that the path $SubList(\pi, 1, vi + 1)$ is unique. $vi$ is initialized to 2 because the corresponding path, $\pi_{vi} = SubList(\pi, 1, 2 + 1)$, consists of exactly two edges, which is guaranteed to be unique by Proposition 1. $vi$ is updated in each iteration on Line 11, after a longer sub-path of $\pi$ has been verified. Line 6 to 12 iterate through $\pi$, and check whether the path from $\pi[1]$ to $\pi[i]$ is a unique shortest path. The function $Shortest(e_{start}, e_{end})$ returns the shortest path from the end vertex of edge $e_{start}$ to the start vertex of $e_{end}$. $Shortest$ returns an empty list if the shortest path is not unique. After verifying that $Shortest(\pi[1], \pi[i]) = SubList(\pi, 2, i)$, $vi$ is updated accordingly.

If the entire path $\pi$ is a unique shortest path, $\pi$ is simply returned. In this case, the query in Listing 1 returns the correct trajectory set without any further checks. If not, the unique part of $\pi$ is extracted and stored in the $first$ variable on Line 16. The remaining part of $\pi$, $\pi'$, is extracted on Line 17, and UniqueSubPaths recursively calls itself on $\pi'$ and returns the sub-paths of it.

Algorithm 1 performs one shortest path computation for each edge in the path $\pi$. We use the state-of-the-art Hub Labeling (HL) [2] shortest path algorithm for the step on Line 7. HL has a time complexity of $O(\log |V|)$ for shortest path computations, where $|V|$ is the number of vertices in the network. As such the time complexity of Algorithm 1 is $O(n \log |V|)$, where $n$ is the number of edges in the $\pi$. Note that the comparison on Line 8 can be implemented as a $O(1)$ operation, by simply comparing the distance of $sppath$ with the distance of $SubList(\pi, 2, i)$.

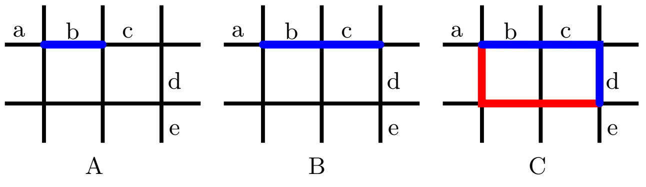

**Figure 3: Processing of path $abcde$**

To illustrate Algorithm 1, consider applying the UniqueSubPaths algorithm to the path $abcde$ on the network in Figure 3. In the first iteration of the loop in Line 6 to 12, the path $abc$ is verified to be unique. That is, the path $b$ that connects edges $a$ and $c$ is a shortest path, and there is exactly one shortest path between edges $a$ and $c$ (shown by a blue line in Figure 3A). Next, the path $abcd$ is verified to be unique path. Again, the path $bc$ that connects $a$ and $d$ is a shortest path, and there is exactly one shortest path between edges $a$ and $d$ (shown by a blue line in Figure 3B). Finally, $abcde$ is found not to be a unique path. The path $bcd$ is a shortest path between $a$ and $e$, but there are multiple shortest paths between edges $a$ and $e$ (these are shown by a red line and a blue line in Figure 3C). The longest unique sub-path along $abcde$ is therefore $abcd$. UniqueSubPaths is then invoked recursively on the remaining sub-path $de$. Because $de$ consists of two edges, it is unique (Proposition 1) and further processing is not required. Finally, the two sub-paths $abcd$ and $de$ are returned.

## 4.4 Practical Exact SPQ Evaluation

The query in Listing 1 only requires retrieving data for the first and last edges in a SPQ. However, with the current weight configuration the probability of false positives depends on the probability of two paths having equal length.

In Section 6, we show that multiple shortest paths are frequent in real-world transportation networks, even when measuring the length of edges with a millimeter granularity.

We therefore design a new weight configuration such that the probability of false positives is very low. To distinguish between the different weight configurations, we refer to the weight configuration used up to this point as *length-weight*, and the weight configuration described below as *prime-weight*.

The fundamental theorem of arithmetic [8] states that the product of a unique set of primes is unique. As such, we can assign a globally unique prime to each edge in the network, and use the product of these primes as our hash function. This hash function guarantees uniqueness for all paths without cycles, because a different path will result in a different set of primes and in turn a different product. For paths with cycles this hash function cannot guarantee the order of visiting each edge in the path, only that each edge is visited the correct number of times. A path containing cycles can be supported by transforming the path into a set of a-cyclic sub-paths or paths consisting of two edges.

However, the product of a large set of primes is an extremely large number and it is not feasible to represent such values using a fixed-width data type. Instead, we compute the logarithm to the product of primes, and store the transformed value with as many digits as possible given a specific data type. Due to the arithmetic rule for logarithms, i.e., $\log(xy) = (\log x) + (\log y)$, the *e.weight* field of each edge can be assigned the value of $\log(e.prime)$, where *e.prime* is the prime number assigned to edge $e$. By setting $e.weight = \log(e.prime)$, the *prime-weight* configuration can be used without further changes.

The logarithm transformation does not affect uniqueness, but the loss of precision due to rounding error may affect uniqueness. As such, estimating the precision of this number gives a rough estimate of the probability of two arbitrary paths having $deltahash(\pi_1) = deltahash(\pi_2)$.

In the following, we assume that an unsigned 64 bit fixed-point data type is used for the *hash* column, a network of 50 million edges, and that a trajectory touches at most 1 million edges. The largest possible hash value under these assumptions is generated by a trajectory that passes the edge assigned the largest prime number one million times. The 50 million$^{\text{th}}$ prime is just below $10^9$, and the largest possible hash value is therefore at most: $\log_{10}(10^9) \cdot 10^6 = 9 \cdot 10^6$. A value of this magnitude requires $\lceil \log_2(9 \cdot 10^6) \rceil = 24$ bits to represent the integer part, which leaves 40 bits to represent the fractional part. The number of significant decimal places, *digits*, is therefore $\lfloor log_{10}(2^{40}) \rfloor = 12$. In other words, false positives can only occur due to rounding error, and this rounding error is bound by $10^{-12}$. For two arbitrary paths, $\pi_1$ and $\pi_2$, between edges $e_1$ and $e_2$ an estimate of the probability of $deltahash(\pi_1) = deltahash(\pi_2)$ is $10^{-12}$. Obviously, a wider data type increases the number of decimal places, and reduces the probability.

In order to have a false positive in the result set for the path $\pi_1$, three conditions must be true: a path $\pi_2$ have the same first and last edge as $\pi_1$, $\pi_2$ is followed by at least one trajectory, and $deltahash(\pi_1) = deltahash(\pi_2)$. Intuitively, the risk of false positives increases with the number of followed paths between $first(\pi_1)$ and $last(\pi_1)$. One way to estimate the probability of false positives is to assume that the 12 digits used to represent the decimals are random. Under this assumption, the probability can be computed using the following formula.

$$p_n = 1 - \left(\frac{\mathcal{B} - 1}{\mathcal{B}}\right)^n$$

Where $p_n$ is the probability false positives, $n$ is the number of distinct traveled paths between $first(\pi_1)$ and $last(\pi_1)$, and $\mathcal{B}$ is the number of possible values for the fractional part, i.e., $10^{12}$. We observe that $n$ is usually low, i.e., vehicles tend to follow similar routes between locations. In our data set, $n$ is always well below 1000. Using these numbers, we arrive at a probability of approximately $10^{-9}$ for false positives when using the *prime-weight* configuration.

In Section 6, we experimentally evaluate the probability of false positives for both the *prime-weight* and the *length-weight* approach.

## 5. SPQ PROCESSING

The recursive function ProcessSPQ, described in Algorithm 2, is used to evaluate SPQs. ProcessSPQ takes as arguments the path $\pi$, a Boolean value *exact*, and the temporal predicate $(start, end)$. ProcessSPQ returns the set of trajectories, $C$, that strictly follow the path $\pi$, and satisfy the temporal predicate $(start, end)$. The Boolean parameter *exact* controls whether false positives are acceptable in the result set. On Line 2, a candidate trajectory set is retrieved using the *LoadTrajectories* function. *LoadTrajectories* is similar to the query shown in Listing 1. In addition to retrieving the *tid*, the *hash* value is retrieved for the first and last edge of the path $\pi$, i.e., $hash_{start}$ and $hash_{end}$, respectively. Further, both $e_{start}.ts_{enter}$ and $e_{end}.ts_{leave}$ are constrained to be within the temporal interval $(start, end)$.

**Algorithm 2** Processing a Strict Path Query

```
Input: A path $\pi$, Boolean $exact$, and temporal predicate
    $(start, end)$
Output: The set of trajectories that strictly follow the path
    $\pi$, and satisfy the temporal predicate $(start, end)$
 1: function ProcessSPQ($\pi$, $exact$, $(start, end)$)
 2:     $C \leftarrow LoadTrajectories(first(\pi), last(\pi),$
                              $deltahash(\pi), (start, end))$
 3:     if $exact$ or $\pi$ has cycles then
 4:         $Subpaths \leftarrow$ UniqueSubPaths($\pi$)
 5:         for all $\pi_i \in Subpaths$ do
 6:             $C' \leftarrow$ ProcessSPQ($\pi_i$, false, $(start, end)$)
 7:             if $i = 1$ then
 8:                 $C \leftarrow filter\_init(C, C')$
 9:             else
10:                 $C \leftarrow filter(C, C')$
11:             end if
12:         end for
13:     end if
14:     return $C$
15: end function
```

On Line 4, $\pi$ is transformed into the set of sub-paths that guarantee no false positives. This step is performed when the *exact* parameter is set to true, or when $\pi$ has a cycle. When $\pi$ has a cycle, the hash column cannot determine the order that edges are touched in. It is therefore necessary to transform $\pi$ into a set of acyclic sub-paths using the UniqueSubPaths algorithm.

Line 7 to 11 verifies that each sub-path is followed, using the two similar functions *filter_init* and *filter*. As the name implies, *filter_init* is only used for the first unique sub-path of $\pi$. Concretely, *filter_init* selects the trajectories from $C'$ where each *tid* from $C'$ is in $C$, and the $hash_{start}$ of *tid* in $C$ match $hash_{start}$ of *tid* in $C'$. The *filter* function selects the trajectories from $C'$ where each *tid* from $C'$ is in $C$, and the $hash_{end}$ of *tid* in $C$ match $hash_{start}$ of *tid* in $C'$.

After each sub-path has been checked, the trajectories in $C$ are guaranteed to follow exactly the path $\pi$. Obviously, ProcessSPQ is only recursive when *exact* is set to `true`, or when $\pi$ has a cycle. For all recursive invocations, *exact* is set to `false` and $\pi_i$ is acyclic. Hence Algorithm 2 is recursive to at most depth two.

The implementation of *LoadTrajectories* uses the `visit-edsegments` table shown in Table 2 with a covering B$^+$tree index on the columns: `eid`, `tenter`, `tid`, `tleave` and `hash` (in this sequence). *LoadTrajectories* is implemented using a slightly extended version of the query shown in Listing 1 as described in Section 5. Because of the order of columns in the B$^+$tree index, it can be used for both spatial and temporal filtering when evaluating the query.

## 6. EXPERIMENTAL STUDY

In this section, we experimentally study and compare the NETTRA index to state-of-the-art in network constrained trajectory indexing, i.e., PARINET.

We evaluate our index w.r.t. frequency of false positives of the practically exact solution, response times, and sensitivity to density, i.e., the number of trajectories that touch the edges in the SPQ.

The experiments are conducted on a Dell Server, with 32 GB RAM, 2 AMD Quad-Core Opteron 2376 CPUs, and 3 Dell Cheetah 15k RPM drives in a RAID 5 configuration. NETTRA is implemented in PostgreSQL 9.3.

Our experiments use a very large real-world trajectory data set[1] recorded from more than 13 thousand vehicles over several years. In total, the data set contains 941 million GPS records and 1.7 million trajectories. This results in 134 million entries such as those in Table 2. We used the map-matching algorithm described in [13], to transform the GPS records into the format in Table 2. The transportation network used is from OpenStreetMap [1] and consists of 1.3 million directed edges.

In order to experimentally study NETTRA, a realistic set of SPQs is required. We use as queries the path of each trajectory in our data set. In total, 1.7 million SPQs are created. A SPQ over 10 edges is between 90 m and 30 km long; a SPQ over 240 edges is between 9 km and 300 km long. The temporal constraint is the entire lifetime of the database unless explicitly stated otherwise.

### 6.1 Number of Sub-paths

Figure 4 shows the average number of unique sub-paths that a path is transformed into. Recall, that for each sub-path, $\pi_i$, it is necessary to retrieve additional trajectory information. As such, a reduction in the number of sub-paths translates directly into a lower number of I/Os. The *All* curve shows the number of sub-paths required by the naïve approach described in Section 4.2. The *length-weight* and *prime-weight* curves show the number of unique sub-paths for each weight configuration. Compared to the *All* curve, the *length-weight* configuration reduces the number of sub-paths by a factor of approx. 30, on average. Using the *prime-weight* configuration, the reduction is a factor of 20 on average compared to *All*. It is expected that more sub-paths are required with the *prime-weight* compared to the *length-weight* configuration because trajectories (and therefore the SPQs used) tend to follow the shortest or the fastest path through a network. A trajectory will generally not follow the path that minimizes an arbitrary cost such as *prime-weight*.

[1]The data set is owned by third parties, and therefore cannot be made publicly available.

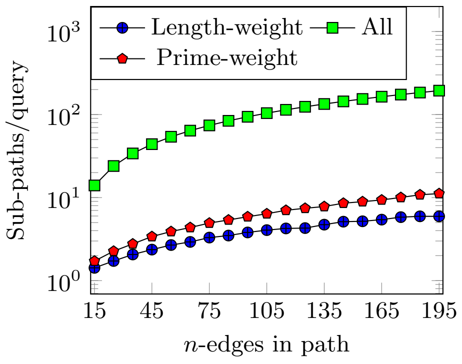

**Figure 4: The number of sub-paths of a SPQ.**

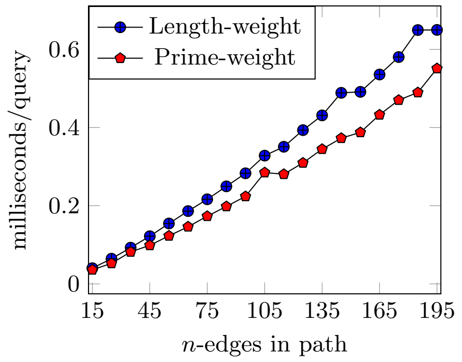

**Figure 5: Process time of Algorithm 1.**

Finally, we study the delay added by Algorithm 1. All 1.7 million paths are processed, and the average processing time w.r.t. the number of edges is measured. Figure 5 shows the average delay. The delay is well below one millisecond, and negligible compared to the time required by a random I/O. Note that the delay is nearly linear to the number of edges in the SPQ.

### 6.2 Frequency of False Positives

To study the frequency of false positives for the practically exact solution, all 1.7 million SPQs are executed. Each SPQ is executed three times using the *length-weight* configuration, using the *prime-weight*, and once using the optimized exact solution.

If the result set of a practically exact query differs from the result set of the exact query, a false positive is reported for the corresponding weight configuration. *length-weight* is configured with a granularity of one millimeter. *prime-weight* is configured as described in Section 4.4.

Figure 6 shows the frequency of false positives. In general we find false positives for each $n$ number of edges using the *length-weight* configuration. The false positives per query appears to decrease for longer paths. We believe this de-

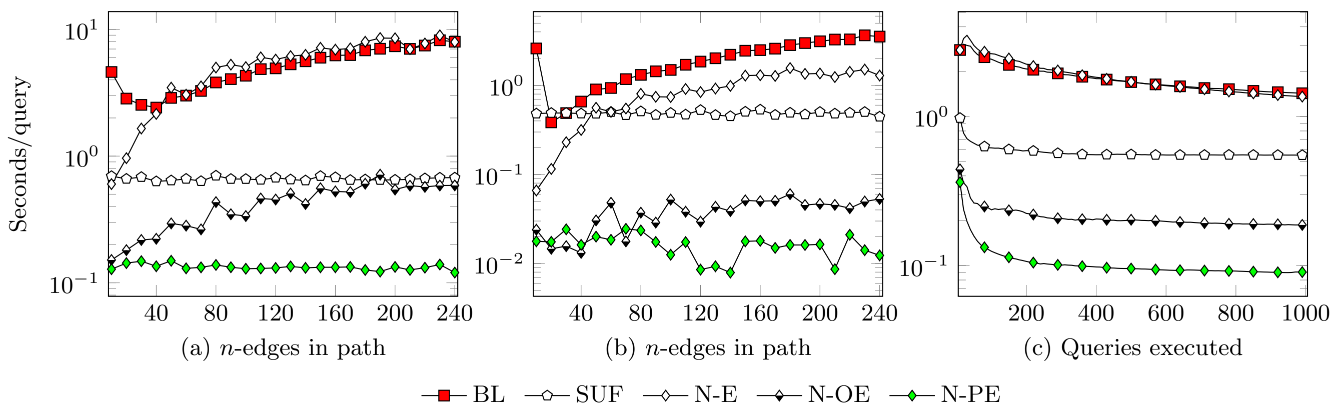

**Figure 7: Query response time. (a) and (b), SPQs with cold and warm cache, respectively. (c) Cache warm-up behavior. Note that the labels and the unit on y-axis are the same for all three graphs.**

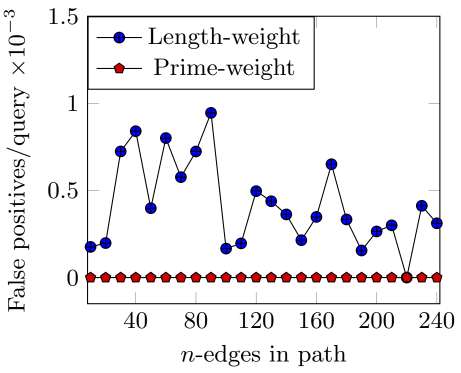

**Figure 6: Frequency of False positives per SPQ.**

crease is due to the fact that fewer trajectories touch both endpoints of a long path compared to a short path. *length-weight* works reasonably well as a filter step, and removes all false positives in 99.9% of all SPQs evaluated.

Using *prime-weight*, we did not identify any false positives. This supports our claim in Section 4.4 that the probability of false positives with this weight configuration is extremely low. Having executed 1.7 million SPQs without finding any false positives, we consider this approach virtually exact and acceptable for most real-world applications.

## 6.3 Query Response Time

We now compare the query response times of different implementations for evaluating SPQs. The first implementation is a baseline, and demonstrates the performance of SPQs using an implementation of PARINET [16]. Two methods can be used to answer a SPQ using PARINET. For both methods the first step is finding a candidate trajectory set by retrieving trajectory data for each edge in the SPQ and finding the trajectories that touch all edges in the path. The candidate set is then refined by retrieving the full path of each trajectory in the candidate set, or by querying the edges adjacent to the SPQ. Both approaches have been implemented and tested. The approach where the edges adjacent to the path are queried is consistently slower than the other approach (by up to one order of magnitude). Because querying adjacent edges is consistently slower, we have omitted it from the graphs. The baseline is referred to as BL in the following.

The second implementation is a suffix tree. As described in Section 2.2, executing a SPQ can be reduced to the problem of finding exact sub-string matches in a text database. First, the path of each trajectory is converted into a string and all strings are concatenated and separated with a special marker '\$'. Next, we instantiate a compressed suffix array on the concatenated string, using the compressed suffix array implementation from PostBio [3]. The compressed suffix array implementation is described in [6] and is to the best of our knowledge a state-of-the-art implementation of compressed suffix arrays. We use the shorthand SUF to refer to this index. We also used the ERA [11] suffix tree implementation, which is the state-of-the-art in external memory suffix tree construction. ERA alleviates some of the memory constraints of suffix trees. The source code of ERA is provided by the authors of [11]. However, the compressed suffix array implementation from [6] consistently outperforms ERA and ERA is therefore omitted from all the graphs.

Three variations of NETTRA have been implemented. N-E uses the naïve exact approach described in Section 4.2, N-OE uses the optimized exact approach described in Section 4.3, and N-PE uses the practically exact approach described in Section 4.4. The optimized exact implementation uses the *length-weight* configuration because it reduces the number of sub-paths the most, see Figure 4. The practically exact implementation use the *prime-weight* configuration to avoid false positives, see Figure 6.

The query response time w.r.t. the number of edges in the SPQ is analyzed in Figure 7 (a) and (b). For each data point, a query set of 30 randomly chosen SPQs over $n$-edges are executed (the same query set across all implementations). Before executing a query set, the file-system cache is flushed and PostgreSQL is restarted. Figure 7, (a) and (b), illustrate the query response time with cold and hot cache, respectively. The average query response times of BL, and N-E, in (a) and (b), are similar and increase with the number of edges in the path as predicted. SUF exhibits constant query time w.r.t. the number of edges in the SPQ. With cold cache in (a), the query performance of the optimized exact version, N-OE, is very competitive compared to SUF, except for very long SPQs, where the two indexes have similar query performance. With a hot cache in (b), N-OE significantly outperform SUF for any number of edges in the query. Finally, N-PE consistently outperforms the other variations. In particular, this solution is more than two orders of magnitude faster than BL.

As can be seen by comparing Figure 7 (a) and (b), the effect of the file-system cache is significant. To further study the effects of the cache, we selected 1000 paths, each consisting of 50 edges, and divided them into 100 query sets with 10 queries in each. Then we flush the file system cache, restarted PostgreSQL, and incrementally executed these 100 query sets. After executing a query set the elapsed time is recorded. Figure 7 (c) shows the execution time per query of this experiment. As can be seen, most implementations benefit considerably from the cache. N-OE and N-PE are consistently faster than both BL and SUF with up to one order of magnitude.

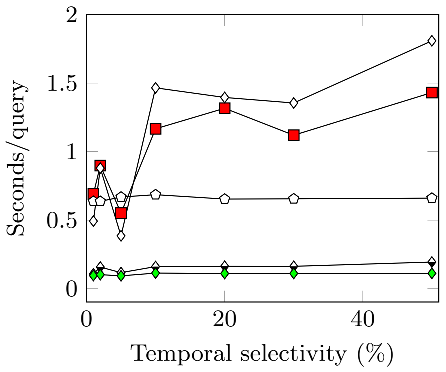

**Figure 8: Effects of temporal selectivity.**

Finally, we study the query response time w.r.t. the selectivity of the temporal constraint in Figure 8. The graph is generated by executing 30 randomly chosen SPQs that each consists of 50 edges with varying temporal selectivity. The effects of the temporal predicate appear small, but a 1% temporal constraint is generally evaluated 50% faster than queries with a 50% temporal constraint. The notable exception here is the suffix tree that does not support temporal filtering. Therefore, its query time is independent of the temporal constraint. To perform temporal filtering, suffix trees require a secondary index to look up temporal information given a network edge and trajectory identifier.

## 7. CONCLUSION

We have presented and formalized the novel strict path query (SPQ) and shown that this new query type has a large number of applications in particular within the traffic domain.

State-of-the-art in network constrained trajectory indexing [16] does not support SPQs efficiently. We have therefore presented the NETTRA index that uses a novel path-encoding scheme to significantly reduce the number of I/Os compared to other trajectory index structures. Two algorithms are used in NETTRA for executing SPQs: an exact and a practical-exact. The former is an order of magnitude faster than existing work. The latter is two orders of magnitude faster but can with an extremely low probability include false positives.

We test and validate the NETTRA index using a very large set of real-world trajectories and queries. For the 1.7 million different queries executed, the practical-exact algorithm has not returned a single false positive and we therefore conclude that this implementation is virtually exact for all practical purposes.

A SPQ can be converted to an exact string-matching problem where suffix trees have optimal search time complexity. We show that in practice NETTRA is a better approach for SPQs, because: (1) the query response time is an order of magnitude faster, (2) the memory requirements are lower, and (3) updates are possible and efficiently supported.

## Acknowledgments

Benjamin Krogh and Kristian Torp are funded by the EU FP7 REDUCTION project (`www.reduction-project.eu`). Nikos Pelekis and Yannis Theodoridis are funded by EU FP7 DATASIM project (`www.datasim-fp7.eu`) and Greek GSRT project Live Roads (code: 36NEWE2009).

## 8. REFERENCES

[1] OpenStreetMap. `http://www.openstreetmap.org/`. Retrieved June 30., 2013.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *SEA*, pages 230–241, 2011.

[3] L. Carvalho. Postbio. http://postbio.projects.pgfoundry.org. Retrieved on November 1st, 2013.

[4] J.-W. Chang, J.-H. Um, and W.-C. LeeP. A new trajectory indexing scheme for moving objects on road networks. *Flexible and Efficient Information Handling*, pages 291–294, 2006.

[5] V. T. de Almeida and R. H. Güting. Indexing the trajectories of moving objects in networks. *GeoInformatica*, 9(1):33–60, 2005.

[6] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[7] E. Frentzos. Indexing objects moving on fixed networks. *SSTD*, pages 289–305, 2003.

[8] G. Hardy and E. Wright. *An introduction to the theory of numbers*. Oxford University Press, USA, 1980.

[9] B. Krogh, O. Andersen, and K. Torp. Trajectories for Novel and Detailed Traffic Analysis. In *ACM-GIS IWGS*, 2012.

[10] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[11] E. Mansour, A. Allam, S. Skiadopoulos, and P. Kalnis. Era: Efficient serial and parallel suffix tree construction for very long strings. *PVLDB*, pages 49–60, 2011.

[12] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[13] P. Newson and J. Krumm. Hidden markov map matching through noise and sparseness. In *ACM-GIS*, pages 336–343, 2009.

[14] B. Pan, Y. Zheng, D. Wilkie, and C. Shahabi. Crowd sensing of traffic anomalies based on human mobility and social media. *ACM-GIS*, 2013.

[15] D. Pfoser and C. S. Jensen. Indexing of network constrained moving objects. In *ACM-GIS*, pages 25–32. ACM, 2003.

[16] I. S. Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial. Indexing in-network trajectory flows. *VLDB J.*, 20(5):643–669, 2011.