PATHFINDER: Storage and Indexing of Massive Trajectory Sets

Stefan Funke
University of Stuttgart

Tobias Rupp
University of Stuttgart

André Nusser
Max Planck Institute for Informatics

Sabine Storandt
University of Konstanz

ABSTRACT

We consider the problem of indexing massive trajectory data in an underlying road network. Our PATHFINDER index structure is based on a state-of-the-art speed-up technique for shortest path planning and allows to both compress and access huge amounts of trajectory data. In a continent-sized network with more than 400 million nodes and almost a billion edges, PATHFINDER allows to retrieve all trajectories within a given space-time cube in a few *microseconds* per reported trajectory. The applicability of PATHFINDER is shown using both synthetic and real-world trajectory sets.

CCS CONCEPTS

• **Information systems → Data management systems**; **Information retrieval**; **Query representation**.

KEYWORDS

trajectories, road networks, queries

ACM Reference Format:
Stefan Funke, Tobias Rupp, André Nusser, and Sabine Storandt. 2019. PATHFINDER: Storage and Indexing of Massive Trajectory Sets. In *16th International Symposium on Spatial and Temporal Databases (SSTD '19), August 19–21, 2019, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340964.3340978

1 INTRODUCTION

With the ubiquity of mobile devices that are capable of tracking positions (be it via GPS or via Wifi/mobile network localization) there is a continuous stream of *location data* being generated every second. Not all of this data is stored permanently, but platforms like Strava, GPSies, or OpenstreetMap allow the users to collect and share their location data with the community. Mobile network providers or companies like Google or Apple also have access to the location data of their customers and use it to improve their services, e.g., to measure traffic flows or detect special events.

In all of these cases, location measurements are typically not considered individually but rather as sequences, each of which reflects the movement of one person or vehicle. In this work we assume that such sequences of location measurements have already been mapped to paths in an underlying transportation network using appropriate methods. Throughout the paper, we use the term *trajectories* to refer to such already map-matched movement sequences.

The main contribution of this paper is the development of a data structure allowing for efficient compression and storage of as well as access to a huge number of such trajectories. Here, by 'huge' we mean tens of millions of trajectories in a country- or continent-sized network, or – in the long run – even billions of trajectories. Taking the trajectories of vehicles into consideration, this enables a plethora of important use cases such as: (1) traffic anomaly detection by monitoring the travel time of vehicles traversing a certain region at a certain time interval, (2) determining most frequently used paths within a region to facilitate historical trajectory based route planning, or (3) quick access to relevant trajectories while mapping the world (the OpenStreetMap project currently hosts more than a million such trajectories but without an efficient retrieval system).

1.1 Problem Description

An underlying road network is given as a directed weighted graph $G(V, E, c)$ with V embedded in $\mathbb{R}^2$. The trajectory data is provided as a collection $\mathcal{T}$ of paths, where each path $t \in \mathcal{T}$ is a sequence of nodes $\pi = v_0 v_1 \dots v_k$ in G annotated with timestamps $\tau_0, \tau_1, \dots, \tau_k$.

Our goal is to construct an index for $\mathcal{T}$ which allows to efficiently answer queries of the form

$$[x_l, x_u] \times [y_l, y_u] \times [\tau_l, \tau_u]$$

that aim to identify all trajectories which in the time interval $[\tau_l, \tau_u]$ traverse the rectangular region $[x_l, x_u] \times [y_l, y_u]$. In the literature this kind of query is often named *window*-query [18] or *range*-query [10], where the formal definitions may differ in detail, also see [19]. In addition, we want to answer queries which specify periodic time events, e.g., a query which asks for all trajectories on weekends that intersect a query rectangle $[x_l, x_u] \times [y_l, y_u]$.

Two straightforward approaches for answering window-queries are the *linear scan* where every edge of every trajectory $t \in \mathcal{T}$ is explicitly checked for intersection with the query rectangle/space-time cube. There is also the idea of an *inverted index* where a network edge is associated with all trajectories using that edge. Then at query time, one determines the set of network edges intersecting the query rectangle and checks the time constraints for all respective associated trajectories. However, both approaches are too slow and space consuming to handle large networks and huge sets of trajectories in practice.

1.2 Related Work

Storage and retrieval of trajectory data is an established field of research. There are basically two different flavours of the problem.

SSTD '19, August 19–21, 2019, Vienna, Austria

ACM ISBN 978-1-4503-6280-1/19/08…$15.00
https://doi.org/10.1145/3340964.3340978

The more geometric and continuous variant considers the trajectory data as mere sequences of points in $\mathbb{R}^2$ (or even $\mathbb{R}^3$) that are freely located in ambient space, see, e.g., [17]. In the other, more discrete variant, as, e.g., proposed in [8], trajectories are paths in an underlying network, which can be exploited for storage as well as indexing. In our work we focus on the latter variant and assume that the 'raw' trajectories, which are typically given as a sequence of GPS coordinates, have been matched to respective paths in G using some *map matching* technique, e.g. as described in [2, 12, 13, 15, 19].

Many other approaches also focus on already map-matched trajectories. Examples are SPNET [10], TED [18], PARINET [14], NETTRA [11], and PRESS [16]. For the kind of queries we are interested in, SPNET represents the state of the art (outperforming other approaches, see [10]), and proposes both a compression scheme (trajectories are segmented and represented by several unique shortest paths) as well as a spatio-temporal index, which is essentially based on a flat spatial decomposition.

Almost all approaches in the literature accept some loss of information to compress the trajectories, especially in the temporal component. SPNET goes as far as only saving start and end time of a trajectory. Our compression is lossless concerning spatial information while keeping a moderate amount of temporal information. Since we do not drop spatial information in the compression, we are able to answer purely spatial queries with a 100% accuracy. For queries including a time interval, our approach works conservatively like SPNET, always containing the exact result as a subset.

There is also a distinction between whether the index is designed to work in a disk or in a RAM context. The state-of-the-art disk based systems are PARINET [14] and NETTRA [11]. They can also be adapted to work in RAM but were then shown to be inferior to SPNET which was explicitly designed to work in RAM. As our approach is also designed to work in RAM, we consider SPNET [10] as our strongest competitor and focus on comparing PATHFINDER to SPNET in our empirical studies. Nevertheless we also report on some experiments externalizing our PATHFINDER structure.

Note that current approaches for efficient retrieval of trajectory data make use of different dedicated data structures for the two main tasks, compression and indexing. In contrast, our approach elegantly uses an augmented version of the so called contraction hierarchy (CH) data structure [7] (see also Section 2.1) for both of these tasks. CH is typically used to accelerate route planning queries, but has also proved successful in other settings like continuous map simplification [5]. This saves space and makes our algorithms relatively simple without the need of too many auxiliary data structures. Only this slenderness allows for scalability to continent-sized road networks and huge trajectory sets.

1.3 Contribution Summary and Outline

In this paper we present a novel index structure that allows to answer window-queries on network-constrained trajectory sets on an unprecedented scale. While current indexing schemes work on small network sizes (e.g., PRESS [16]: network of Singapore, PARINET [14]: cities of Stockton and Oldenburg, TED [18]: cities of Singapore and Bejing) or moderate sizes (e.g., SPNET [10]: network of Denmark with 800k vertices), our approach efficiently deals with country- (Germany with 57 million vertices) or even continent-sized networks (Europe with 437 million vertices). For example, for the network of Europe and 10 million trajectories, we can answer a window-query within few *microseconds* per reported trajectory in the output. The scalability of our approach is mostly due to the fact that our index structure is a very lean augmentation of a constructed *contraction hierarchy* [7], which might be available anyway, if routing queries are to be answered for the network. It inherits the hierarchical structure from the contraction hierarchies and is hence equally suitable for small and large query windows.

After introducing some basic concepts in Section 2, we develop our spatial indexing scheme in Section 3. The extension to cater for temporal information is described in Section 4, followed by an extensive experimental evaluation in Section 5.

2 PRELIMINARIES

As we assume that two nodes are connected by at most one edge, a path can be uniquely represented by its nodes or by its edges. Thus, depending on what is more convenient, we either use a representation via nodes $\pi = v_0 v_1 \dots v_k$ or via edges $\pi = e_0 e_1 \dots e_{k-1}$.

For our approach, we have to briefly introduce the main construction that it relies on, namely the contraction hierarchy (CH).

2.1 Contraction Hierarchies

Our algorithms heavily rely on the *contraction hierarchy* (CH) [7] data structure, which was originally developed to speed up shortest path queries. A nice property of CH is that as a by-product it also constructs compressed representations of shortest paths.

The CH augments a given graph $G(V, E, c)$ with *shortcuts* and *node levels*. The elementary operation to construct *shortcuts* is the so-called *node contraction*, which removes a node v and all of its adjacent edges from the graph. To maintain shortest path distances in the graph, a shortcut $s = (u, w)$ is created between two adjacent nodes u, w of v if the only shortest path from u to w is the path uvw. We define the cost of the shortcut to simply be the sum of the costs of the replaced edges, i.e. $c(s) = c(uv) + c(vw)$. The construction of the CH is the successive contraction of all $v \in V$ in some order; this order defines the *level* $l(v)$ of a node v. The order in which nodes are contracted strongly influences the resulting speed-up for shortest path queries and hence many ordering heuristics exist. In our work we choose the probably most popular heuristic: nodes with low *edge difference* [7], which is the difference between number of added shortcut edges and the number of removed edges when contracting a node, are contracted first. We also allow the simultaneous contraction of non-adjacent nodes. As a result, the maximum level of even a continent-sized road network like the one of Europe never exceeds a few hundred in practice. The final CH data structure is defined as $G(V, E^+, c, l)$ where E^+ is the union of E and all shortcuts created.

We also define the nesting depth $nd(e)$ of an edge $e = (v, w)$. If e is an original edge, then $nd(e) = 0$. Otherwise, e is a shortcut replacing edges e_1, e_2, and we define its nesting depth $nd(e) := \max\{nd(e_1), nd(e_2)\} + 1$. Clearly, the nesting depth is upper bounded by the maximum level of a node in the network.

Algorithm 1 Converting π into its CH-representation. Note that it directly works on π so the indices change while replacing edges.

```
1:  procedure toCHPath(π = e_0 ... e_{k-1})
2:      i ← 0
3:      while i + 1 < length(π) do
4:          if e_i and e_{i+1} form a shortcut then
5:              s ← getShortcut(e_i, e_{i+1})
6:              replace e_i and e_{i+1} by s in π
7:              i ← max{i − 1, 0}
8:          else
9:              i ← i + 1
10:     return π
```

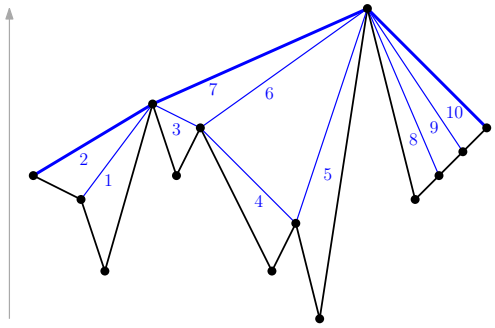

Figure 1: Original path (black, 13 edges) and derivation of its CH-representation (bold blue, 3 edges) via repeated shortcut substitution (in order according to the numbers). y-coordinate corresponds to CH level.

3 SPATIAL PATHFINDER

This section contains the main contribution of our work: our compression scheme as well as the spatial part of the Pathfinder data structure and query algorithm.

3.1 Compression

Given a precomputed CH graph, we construct a CH-representation for each trajectory $t \in \mathcal{T}$, that is, we transform the path $\pi = e_0 e_1 \dots e_{k-1}$ with $e_i \in E$ in the original graph into a path $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'-1}$ with $e'_i \in E^+$ in the CH graph.

Our algorithm to compute a CH-representation is quite simple: We repeatedly check if there is a shortcut bridging two neighboring edges e_i and e_{i+1}. If so, we substitute them with the shortcut. We do this until there are no more such shortcuts. See Algorithm 1 for the pseudocode and Figure 1 for an example. Note that uniqueness of the CH-representation can be proven and therefore it does not matter in which order neighboring edges are replaced by shortcuts. The running time of that algorithm is linear in the number of edges:

Theorem 3.1. *The CH-representation of a trajectory $\pi = e_0 \dots e_{k-1}$ can be computed in $O(k)$.*

Proof. During *CH* construction we can store all neighbors of a node v in a hash map, allowing for constant access time, see [4]. Given two edges $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_3)$, we can then check for and retrieve a shortcut (v_1, v_3) in constant time. Thus, the body of the while-loop in Algorithm 1 takes $O(1)$ time. However, the while-loop can be entered at most $2k$ times since in every round either i is increased or the length of π decreases by one. □

For the resulting CH-representation $\pi' = e'_0 e'_1 e'_2 \dots e'_{k'-1}$, the repeated substitution of an $e'_i \in E^+ \setminus E$ with the two edges it represents (also called *child edges*) yields the original path π. Thus, we have a lossless compression scheme with respect to the spatial information of t. Note that by switching to the CH-representation, we can achieve a considerable compression rate in case the trajectory is composed of few shortest paths (as shortest paths usually have a very concise CH-representation, see e.g. [6]).

3.2 Retrieval Overview

At a high level, the idea of our retrieval process is to associate a trajectory with all edges of its *compressed representation* in E^+. Only due to that compression, it becomes feasible to store a huge number of trajectories within the index. Answering a spatial query then boils down to finding all edges of the CH for which a corresponding path in the original graph intersects the query rectangle. Typically, an additional query data structure would be used for that purpose. Yet, we show how to utilize the CH itself as a geometric query data structure. Details are given in Section 3.3 and improvements in Section 3.4. After this step, however, some of the edges we retrieve represent edges or paths which do not actually intersect the query rectangle. This overestimation is due to checking the intersection with the bounding box and not the actual underlying path. Therefore, these "pseudo-intersecting" edges need to be filtered out by closer inspection. The details are explained in Section 3.5. Finally, we only have to return all trajectories which are associated with any of the edges remaining after the filtering step. High-level pseudocode is provided in Algorithm 2.

Algorithm 2 Spatial Pathfinder Algorithm

```
1: procedure PathfinderQuery(Q)
2:     E_O ← FindEdgeCandidates(Q)
3:     E_r ← RefineEdgeCandidates(Q, E_O)
4:     return GetAssociatedTrajectories(E_r)
```

3.3 Finding Edge Candidates

Let us now explain the details of the function called in Line 2 of Algorithm 2. This requires two central definitions:

- With $PB(e)$ we denote the *path box* of an edge e. It is defined as the bounding box for the path that e represents in the original graph G in case $e \in E^+$ is a shortcut, or simply the bounding box for the edge e if $e \in E$.
- We define the *downgraph box* $DB(v)$ of a node v as the bounding box of all nodes that are reachable from v on a down-path (only visiting nodes of decreasing CH-level), ignoring the orientation of the edges. In Figure 2, the downgraph boxes of the green/red/blue nodes are depicted in light green/blue/red.

Both $PB(e)$ and $DB(v)$ can be computed for all nodes and edges in linear time via a bottom-up traversal of the CH in a preprocessing step and *independently* of the trajectory set to be indexed.

For a spatial-only window-query with query rectangle Q, we start traversing the CH level-by-level in a top-down fashion, first

Algorithm 3 The algorithm to find edge candidates given a query rectangle Q.

```
1:  procedure FindEdgeCandidates(Q)
2:      V_T ← FetchTopNodes(Q)
3:      E_O ← ∅
4:      for v ∈ V_T do
5:          E_O ← E_O ∪ FindCandidatesForNode(v, Q)
6:      return E_O

7:  procedure FindCandidatesForNode(v, Q)
8:      C ← ∅
9:      for e ∈ down edges of v do
10:         if PB(e) ∩ Q ≠ ∅ then
11:             C ← C ∪ {e}
12:         v_l ← lower node of e
13:         if DB(v_l) ∩ Q ≠ ∅ then
14:             C ← C ∪ FindCandidatesForNode(v_l, Q)
15:     return C
```

inspecting all nodes which do not have a higher-level neighbor (note that there can be several of them in case the graph is not a single connected component). We can check in constant time for intersection of the query rectangle and the downgraph box of a node, only continuing with children of nodes with non-empty intersection. We call the set of nodes with non-empty intersection V_I. The set of candidate edges E_O are then all edges adjacent to a node in V_I. See Algorithm 3 for the pseudocode.

Let us first prove the following lemma, which we then use to prove correctness of our query routine.

Lemma 3.2. *For every edge $e = (v, w)$ or $e = (w, v)$ with $l(v) < l(w)$, $DB(w)$ contains the path represented by e.*

Proof. We prove the lemma by induction over the nesting depth of e. Additionally, assume that $e = (v, w)$ as the proof for $e = (w, v)$ is equivalent. Clearly, if $nd(e) = 0$, then e is an original (non-shortcut) edge, the lemma follows trivially since $DB(w)$ by definition contains v and w. Now consider the case when $nd(e) > 0$. Then e is a shortcut bridging edges $e_1 = (v, u), e_2 = (u, w)$ with smaller nesting depths. By the induction hypothesis, $DB(v)$ contains the path represented by e_1 since $l(v) > l(u)$, and $DB(w)$ contains the path represented by e_2 by the same argument. As $l(v) < l(w)$ and thus v is reachable on a down-path from w, $DB(w)$ contains the path represented by e. □

A simple application of this lemma shows that every edge that has to be reported is found by our query routine.

Theorem 3.3. *Every edge e which represents a path π intersecting Q is adjacent to a node in V_Q.*

Proof. Consider an edge $e = (v, w)$ whose represented path intersects Q, w.l.o.g. $l(v) < l(w)$. By the previous lemma, we know that $DB(w)$ contains the path represented by e. As all the ancestors of w also contain $DB(w)$, the search from the root will indeed reach w and thus $w \in V_Q$. Finally, note that e is adjacent to w. □

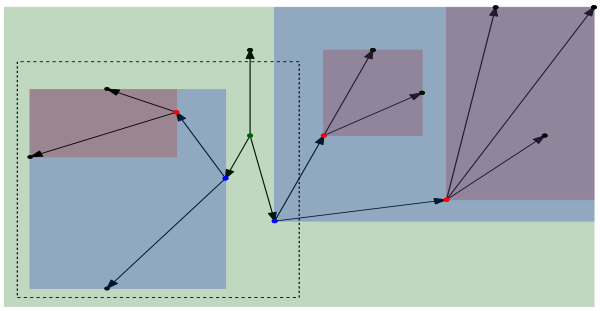

Figure 2: Dashed query rectangle: The left blue *downgraph box* is fully contained in the query rectangle, therefore we know that the inner red one is contained too.

3.4 Improvements

In the following, we explain several significant improvements to the just described edge retrieval data structure. The first improvement addresses how to quickly retrieve the top nodes. The following three improvements are related to pruning the search on the CH graph. Finally, we explain how to parallelize the search.

R-Tree for Top Nodes. In Line 2 of Algorithm 3 we fetch the relevant top nodes. Recall that top nodes are all the nodes $v \in V$ which have no edge to a higher level node. Continental road networks are often not connected and hence several top nodes might exist. By organizing the top nodes with their downgraph boxes in an *R-tree* [9] we can quickly identify the top nodes $v \in V$ for which $DB(v) \cap Q \neq \emptyset$ and continue with them as in Algorithm 3.

Downgraph Box Contained in Query Rectangle. If we notice during the CH-traversal that a *downgraph box* is completely contained in the query rectangle, we do not need to check the spatial constraint for all of its child *downgraph boxes* anymore, see Figure 2.

Obsolete Edges. Edges not associated with any trajectory and whose lower-leveled end node can be reached via other non-obsolete edges are marked obsolete and can therefore be omitted in the search. Note that the marking of obsolete edges is not uniquely determined. In this work, we greedily mark the obsolete edges. After having marked the obsolete edges, we sort our edge lists accordingly, such that we can directly access the non-obsolete edges without having to check for the obsoleteness status at query time.

Tree Edges. During the CH-traversal, it suffices to visit a node once. To speed up the CH-traversal, we reduce the DAG to a tree in the preprocessing by greedily "deleting" edges. We store the tree by marking the tree edges, which is very similar to marking obsolete edges. The tree edges are a subset of the non-obsolete edges.

Parallelization. By imposing a tree structure on the CH graph, parallelization of the CH graph also becomes straightforward via a breadth-first search where in each round we have a set of nodes (on the same CH level) which needs to be processed. This set is split up among the threads which process their part and then return a set of nodes to be processed in the following round. All the returned sets of the threads are merged (eliminating duplicates) and become the set of nodes which needs to be processed in the next round.

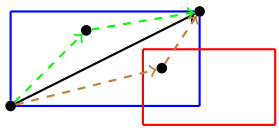

Figure 3: Unclear intersection of edge and query rectangle (red): unpacking could result in a path which does intersect the red query rectangle (brown) or not (green).

3.5 Refining Edge Candidates and Retrieving Associated Trajectories

Having retrieved the candidate edges, we now have to filter out edges e for which only the path box $PB(e)$ intersects but not the path represented by e. For simplicity, let us first focus on a single edge $e \in E_O$. If e is a non-shortcut edge, we can easily decide whether e intersects Q. However, if $e = (v, w)$ is a shortcut, more effort might be necessary. Clearly, if $PB(e) \cap Q = \emptyset$, then e must not be reported, but if $v \in Q$ or $w \in Q$, e must be reported. The interesting case is thus when $PB(e) \cap Q \neq \emptyset$ but $v, w \notin Q$. This setting is shown in Figure 3. In this case we need to recursively unpack e to decide whether e (and the trajectories associated with e) has to be reported. As soon as one child edge reports a non-empty intersection in the recursion, the search can stop and e must be reported. We call the set of edges that results from this step E_r. Note that in practice, it is almost never necessary to completely unpack an edge for a definitive decision.

Our final query result is all the trajectories which are referenced at least once by the retrieved edges, i.e. those $t \in \bigcup_{e \in E_r} \mathcal{T}_e$.

3.6 Discussion

In some sense our index structure could be interpreted as a much improved inverted index, where (a) we do not only use original edges but also CH shortcut edges to represent a trajectory if possible and (b) instead of scanning all edges we instrument the constructed contraction hierarchy as a spatial index. Improvement (a) not only dramatically decreases the space consumption of the index structure compared to a naive inverted index but also limits the number of edge-trajectory associations to collect. Additionally, (b) drastically cuts down on the edges whose associations one needs to consider at all. It is also noteworthy that a considerable part of the index construction does *not depend on the actual trajectory set to be indexed.* In particular, the construction of the CH itself as well as the path and downgraph boxes are just based on the structure of the network itself. Only the tagging of obsolete and tree edges actually depends on the trajectory set. This is in stark contrast to, e.g., SPNET [10] where the spatial index is a partition guided by the trajectory set.

4 ADDING TEMPORAL INFORMATION

4.1 Timestamps

Timestamps of a trajectory t are annotations to its nodes. In the CH-representation of t, we omit nodes via shortcuts hence losing some temporal information. Yet, PATHFINDER will always answer queries conservatively, i.e., returning a superset of the exact result set. It has been observed in [10] that "fine-grained temporal information on trajectories has limited value for spatio-temporal filtering", so we do not expect this to be a real issue in practice, but will verify this conjecture experimentally as well.

4.2 Time Intervals

Like the spatial bounding boxes $PB(e)$, we store time intervals to keep track of the earliest and latest trajectory passing over an edge. Similar to $DB(v)$ we compute minimal time intervals containing all time intervals associated with edges on a down-path from v This allows us to efficiently answer queries which specify a time interval $[\tau_l, \tau_u]$. Like the spatial bounding boxes, we use these time intervals to prune tree branches when they do not intersect the time interval of the query.

An edge is associated with a set of trajectories, each of which we could check for the time when the respective trajectory traverses the edge. It is more efficient to store for all trajectories traversing an edge their time intervals in a so-called *interval tree* [1]. By that we can efficiently retrieve the associated trajectories matching the time interval constraint of the query for a given edge. An interval tree storing ℓ intervals has space complexity $O(\ell)$, can be constructed in $O(\ell \log \ell)$, and can retrieve all intervals intersecting a given query interval in time $O(\log \ell + o)$ where o is the output size.

4.3 Time Slices

We first define *time slices* more formally. Let p be the period length and k the number of slices, we define $S_{all} = \{0, \ldots, k-1\}$ to be the set of slices and $S_q \subseteq S_{all}$ a query set of slices. For example, when the period is a week and the slices are the days of the week, we have $k = 7$ and $S_q = \{5, 6\}$ for Saturday and Sunday. Formally, the set of times of a time slice query is given by

$$\bigcup_{i \in \mathbb{N}, j \in S_q} \left[\left(i + \frac{j}{k} \right) \cdot p, \left(i + \frac{j+1}{k} \right) \cdot p \right],$$

where we assume that the zero time stamp marks the beginning of a new period due to simplicity. Otherwise, we have to adjust by a constant offset.

This enables queries for periodic time intervals. In our implementation, we set p to be one week and split it into $k = 64$ time slices. To get the corresponding time slices of a time interval, we need to compute the time slices of its borders. For a given border τ, we first compute its timestamp within the period $\tau_p = (\tau \mod p)$. Then we calculate the slice with $j = \left\lfloor \frac{\tau_p}{k} \right\rfloor$. As we know start and end slice, we can easily determine the set S_{slices} of slices $[\tau_l, \tau_u]$ falls into. We store S_{slices} as bitset with size k in which all bits for $j \in S_{\text{slices}}$ are set, whereas all $j \in S_{\text{all}} \setminus S_{\text{slices}}$ are not set. Unification and intersection of time slice sets are simple bitwise "AND" and "OR" operations in this implementation. With unification and intersection, we build analogues of $PB(e)$ and $DB(v)$ to speed up queries which specify S_q.

At this point our PATHFINDER structure can also answer queries like "return all trajectories intersecting $[x_l, x_u] \times [y_l, y_u]$ on Monday and Friday afternoons in March 2017".

5 EXPERIMENTS

We implemented the described algorithms in C++. Experiments were conducted on two machines:

Table 1: Characteristics of considered networks ($M = 10^6$).

	Germany	Europe
# nodes	$57.4M$	$437.4M$
# edges (original)	$121.7M$	$902.1M$
# edges (CH)	$248.4M$	$1694.2M$
CH construction time (min)	16	125

Table 2: Characteristics of trajectories in $\mathcal{T}_{\text{ger,real}}$.

	∅	σ	max.
# shortest paths	11	16.6	1118
length (km)	14.78	34.6	1928
original length (#edges)	347	654	46112
compressed length (#edges)	37	56	3726

(1) AMD Ryzen Threadripper 1950X (16-Core), 128 GB RAM and a 512GB Toshiba OCZ RD400 NVMe SSD (2.6 GB/s)
(2) Intel(R) Xeon(R) CPU E5-2650 v4 (24-Core), 768 GB RAM

Note that the 768GB of RAM of the second machine are humble in comparison to [10] which used 2TB of RAM for far smaller graphs (800k nodes for the Danish road network) and less trajectories.

5.1 Graph Data

We build our graphs from OpenStreetMap (OSM) data.[1] From the available networks, we chose to use the German and the European graph and constructed the respective CH-graphs G_{ger} and G_{eu}. In both cases, all types of path segments – from hiking trails to motorways – are included, and the construction of the CH roughly doubles the number of edges. G_{ger} has over 57 million nodes and almost 250 million edges, G_{eu} 437 million nodes and 1.7 billion edges; see Table 1 for the details. The maximum level of the CH never exceeded 560, which also bounds the depth of our search procedure. For our experiments with real trajectory data on G_{ger}, we used our medium sized Threadripper machine. The Xeon machine was only necessary for G_{eu} and very large synthesized data sets.

5.2 Real-World Trajectory Data

For real world data, we considered all traces within Germany in the bundled public collection of GPS traces from OSM[2], only dropping low quality traces (e.g., due to extreme outliers, non-monotonous timestamps, ...). As a result, we obtain 350 million GPS measurements that are matched to G_{ger} with the map matcher from [15] to get a dataset with 372,534 trajectories which we call $\mathcal{T}_{\text{ger,real}}$.

We consider the number of shortest paths a trajectory consists of, its length in kilometer, and its length in edges given the original (non-CH) representation as insightful quantities, which can be found in Table 2 with average, standard deviation and maximum. Note that on average, a trajectory can be represented by 11 shortest paths. Since the OSM data set is highly heterogenous, as users can upload all sorts of trajectories from short hiking trips to long road trips, the maximum values are far from the average.

[1] https://download.geofabrik.de/
[2] https://planet.openstreetmap.org/gps/gpx-planet-2013-04-09.tar.xz

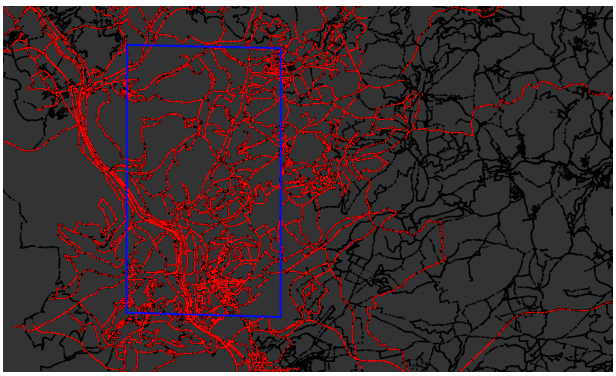

Figure 4: Visualization in the German region Saarland. The blue rectangle is the query, the red lines are the returned trajectories of $\mathcal{T}_{\text{ger,real}}$. Edges which are not contained in any retrieved trajectory are drawn black.

5.3 Compression

The original edge representation of $\mathcal{T}_{\text{ger,real}}$ consists of 121.8 million edges (992 MB on disk), whereas the CH-representation only requires 13.8 million edges (112MB on disk). The actual compression for the 372k trajectories took 42 seconds, that is, around 0.1ms per trajectory. No further compression technique was employed.

As the relationship between the number of edges in the original representation and the CH-representation is of great interest for the quality of the latter as a compression scheme, we compiled various characteristic aspects in Table 2. To no surprise, the CH-representation is significantly more compact.

5.4 Synthesized Trajectory Data

To demonstrate that PATHFINDER scales well, we additionally generate trajectory data ourselves by randomly choosing a source node v_s and a target node v_t. If the great-circle distance between v_s and v_t is below a given parameter d, we include the shortest path in our test data $\mathcal{T}_{\text{synth}}$. By varying d we can investigate the influence of the length of the trajectories on our data structure.

Compared to a real world trajectory which consisted on average of 11 shortest paths, t is only one shortest path by construction and has therefore a far conciser CH-representation. To make our experiments more realistic, we did not compute the CH-representation for the whole trajectory t but we represent it by the concatenated CH-representations of a number of shortest path segments instead. We sampled the number of such segments uniformly between 8 and 14 to get an expected average of 11. This may look like a negligible tweak, but it worsened the query time in our experiments by up to a factor of 3. To generate time data for t, we set τ_0 to a random date from 2008 onwards. For $i \in \{1, \dots k\}$ (with k being the uncompressed path length) we set $\tau_{i+1} = \tau_i + \tau_{\text{step}}$, where τ_{step} is sampled uniformly between 1 s and 9 s to get an expected average of 5 s which is the median edge time of $\mathcal{T}_{\text{ger,real}}$.

5.5 Index Structure

In Table 3 we state the setup time and space requirement of our index structure for the real-world trajectory set on the network

Table 3: Setting up the auxiliary data structures (CH construction and compression excluded). $|\mathcal{T}_{\mathbf{ger,real}}| = 372,534$ and $|\mathcal{T}_{\mathbf{eu,synth}}| = 10^7$, $d = 10^5$ km.

	$\mathcal{T}_{\text{ger,real}}$	$\mathcal{T}_{\text{eu,synth}}$
setup time	349s	5040s
total size	126GB	485GB

Table 4: Timings in seconds for $|\mathcal{T}_{\mathbf{ger,real}}| = 372,534$ for different variants with spatial only queries, single thread.

	1/2	1/4	1/8	1/16	1/32
linear scan	74.53	79.65	82.37	83.33	83.90
inverted index CH	87.26	69.78	60.05	56.45	55.37
PATHFINDER	0.89	0.39	0.14	0.04	0.01

of Germany and the synthetic trajectory set on the network of Europe. Note that currently the construction does not make use of parallelism at all when building the index structure. A considerable speed-up by parallelization can be expected there. For additional insights into the two data sets, we give the histograms of how many trajectories are associated with each edge, see Figure 5.

5.6 Query Answering

Since we expect practical applications to query for different sized rectangles, we tested our algorithms in all experiments with rectangle sizes of different orders of magnitude. For the magnitude parameter r within the range $\{1, \dots, 5\}$, we created a query rectangle $R_r = [x_l, x_u] \times [y_l, y_u]$ as follows: First we compute the bounding box R_G of the graph. Then, a random node of our graph is sampled and taken as the lower left corner of the query rectangle. Finally, we set the width and height of R_r to 2^{-r} of the width or height of R_G respectively. A visual depiction of an answered query can be seen in Figure 4.

For evaluating performance for time interval constraints, we also generate intervals of the form $[\tau_l, \tau_u]$, where τ_l lies in the time frame of the given trajectory data and the difference between τ_l, τ_u is chosen to be one week. For the time slices we constructed queries which ask for all trajectories in a region which happened on a certain day of the week.

The measurements are obtained by averaging the time for 100 queries with the same configuration. For each cell in one column, we used the same 100 queries.

Comparison to Baselines. In order to demonstrate the efficiency of PATHFINDER, we first compare it to the *linear scan* algorithm and to the *inverted index* approach described in Section 1.1. However, we do not use the most naive variant of the inverted index here but already the tuned variant where we index the trajectories in the CH graph instead of the original graph. Note that the naive variant does not allow for any compression and exhibits worse query times. Thus, using the inverted CH index already provides some improvements. However, the full power of PATHFINDER is only achieved by augmenting the CH with a spatial index structure.

Table 5: Timings in seconds for different sized sets $\mathcal{T}_{\mathbf{ger,synth}}$, spatial only queries, single thread.

set size	100, 000	$1M$	$10M$
linear scan	156.237	1568.022	15884.508
inverted index CH	69.577	68.403	71.550
PATHFINDER	0.029	0.107	0.415

Table 6: Timings *in milliseconds* for $|\mathcal{T}_{\mathbf{ger,real}}| = 372,534$ with 16 threads for different constraints.

	1/2	1/4	1/8	1/16	1/32
pure spatial	108.7	48.5	15.9	7.2	3.2
intervals	12.1	6.2	3.6	2.3	1.9
slices	44.0	20.5	8.5	3.6	2.4
intervals + slices	10.0	5.8	3.2	2.0	1.5

Table 4 shows the measured query times for all three approaches for different sizes of the query rectangle. We notice that PATHFINDER is faster than the naive approaches by several orders of magnitude, especially for small rectangles. At first glance, it is surprising that the approach using the inverted index is sometimes slower than the linear scan. This can be explained by our relatively sparse trajectory dataset. To confirm the asymptotic benefit of the inverted index, we ran tests with larger synthesized data sets for which the results can be found in Table 5. There we can see that the time complexity for the linear scan is unsurprisingly linear in $|\mathcal{T}_{\text{synth}}|$. As the time complexity for the inverted index approach is not, we can also see that it outperforms the linear scan as the number of trajectories increases. PATHFINDER is even faster by several orders of magnitude than the inverted index approach.

In Table 4, the 12 ms taken by PATHFINDER for the 1/32 sized query allows a good comparison to TED: In [18], it is reported in Figure 17f that TED requires more than 40 ms for its *window* queries in a similar setting (500k trajectories) with the exception of geographical graph size. Their graph is only Singapore (20k vertices compared to 57.4 million in our case). As time complexity with respect to the number of trajectories is linear for TED while it is not for PATHFINDER, this difference only grows when going to larger scales.

Parallelization. In Table 6 we show the times for different variants using parallelization. We can see that for smaller rectangles, a query can be answered within a few milliseconds. Adding a time slice constraint significantly reduces the required time because the output set $\mathcal{T}_{\text{out}}$ becomes smaller and parts of the traversed tree can be pruned. Specifying a time interval constraint leads to even lower response times because, on one hand $\mathcal{T}_{\text{out}}$ is even smaller, and on the other hand we used interval trees to speed up such queries.

As already mentioned, we synthesized big datasets to test the scaling behaviour of PATHFINDER on the large graph G_{eu}. The resulting times for pure spatial queries using 24-fold parallelization can be seen in Table 7(a). Additionally, timings for the most essential steps are shown in 7(b), 7(c), and 7(d). For example, having generated 10^7 random trajectories where source and target have an Euclidean distance of at most 400km, querying with a randomly placed rectangle of 1/16th the width and 1/16th the height of the

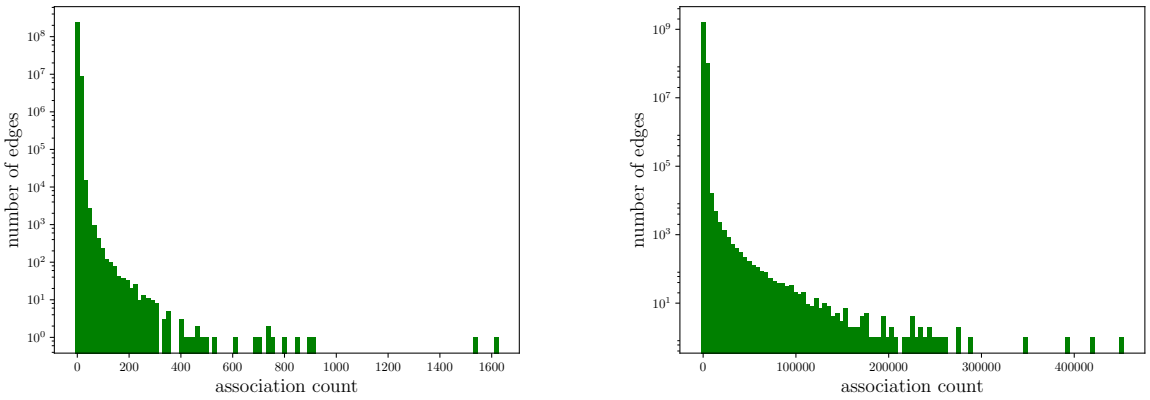

Figure 5: The histograms of edge-trajectory associations for $|\mathcal{T}_{\text{ger, real}}| = 372,534$ (left) and $|\mathcal{T}_{\text{eu, synth}}| = 10^7$, $d = 10^5$ km (right).

Table 7: Measurements for $|\mathcal{T}_{\text{eu, synth}}| = 10^7$ with 24 threads for spatial only queries.

(a) Overall querytime in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.817	0.628	0.270	0.093	0.041
100km	5.630	4.381	1.866	0.609	0.201
400km	6.177	4.959	2.124	0.685	0.222
100,000km	6.889	4.773	1.882	0.608	0.211

(b) FindEdgeCandidates in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.254	0.191	0.088	0.037	0.021
100km	1.219	0.886	0.372	0.140	0.050
400km	1.191	0.893	0.377	0.140	0.052
100,000km	1.194	0.815	0.315	0.106	0.043

(c) RefineEdgeCandidates in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.280	0.207	0.086	0.025	0.007
100km	1.576	1.158	0.474	0.144	0.042
400km	1.504	1.152	0.458	0.135	0.043
100,000km	1.430	0.931	0.344	0.104	0.028

(d) GetAssociatedTrajectories in *seconds*

d	1/2	1/4	1/8	1/16	1/32
25km	0.269	0.221	0.090	0.028	0.009
100km	2.754	2.269	0.992	0.314	0.103
400km	3.379	2.827	1.246	0.391	0.118
100,000km	4.090	2.879	1.139	0.360	0.119

(e) Result size $|\mathcal{T}_{out}|$

d	1/2	1/4	1/8	1/16	1/32
25km	168,570	134,451	57,107	18,468	6,595
100km	2,288,640	1,948,294	903,384	314,285	122,174
400km	2,824,558	2,511,392	1,359,735	568,822	256,962
100,000km	4,781,852	4,174,645	2,577,375	1,334,826	745,257

bounding box of Europe takes 0.685 seconds, reporting around 568k trajectories in the output (see Table 7(e)). Note that such a query rectangle still has width and height of several hundred kilometers. In case of such a rather large result set, most of the time is spent collecting the associated trajectories (0.391 seconds), while the other steps take only 0.140 (FindEdgeCandidates) and 0.135 (RefineEdgeCandidates) seconds. In general, though, the times for these steps are in the same order of magnitude, meaning there is no single bottleneck, which implies that all of our various optimizations are necessary as otherwise one of the steps would dominate the runtime and thereby become the bottleneck. Notably, the measured times make a big jump between $d = 25$ km and $d = 100$ km. The reason seems to be the size of the result set $\mathcal{T}_{\text{out}}$, which is larger for $d = 100$ km than for $d = 25$ km by at least a factor of 15 for all parametrizations of the query size, as we can see in 7(e). Additionally, on average a trajectory in $\mathcal{T}_{\text{out}}$ has 4 times more original graph edges for $d = 100$ km than for $d = 25$ km by construction. Luckily, our CH-representation cushions this data growth. Of highest significance is that we can handle the smaller sized rectangle queries with times far below one second. We want to emphasize again that even the smaller rectangles still have a significant size as their size is chosen relative to the bounding box of Europe. Obviously, if the set of trajectories $\mathcal{T}_{\text{out}}$ to be reported is large – e.g., querying with 1/4th of the bounding box width and height of Europe, we have $|\mathcal{T}_{\text{out}}| > 4$ million – the query times must be higher. Therefore, it is of interest to consider the query time *per reported trajectory* as

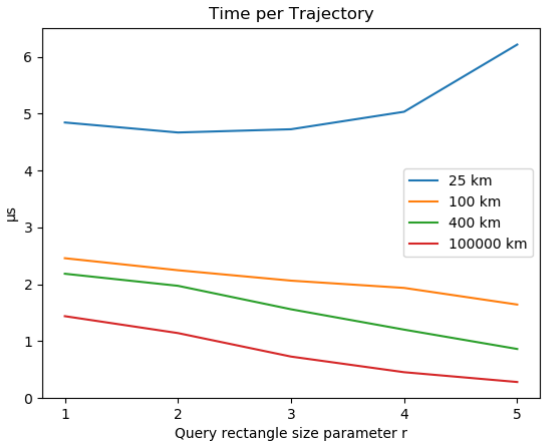

Figure 6: Time per trajectory in the output for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with 24 threads for spatial only query.

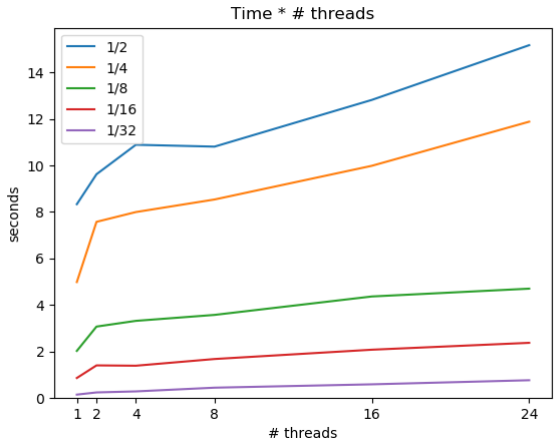

Figure 7: Table 9 as a plot with reported times normalized by multiplication with the number of used threads.

Table 8: Times in seconds for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with $d = 100\,\text{km}$ and 32 threads for different constraints.

	1/2	1/4	1/8	1/16	1/32
pure spatial	5.630	4.381	1.866	0.609	0.201
intervals	1.439	1.095	0.532	0.174	0.076
slices	4.273	3.412	1.449	0.461	0.157
intervals + slices	1.447	1.059	0.520	0.168	0.068

Table 9: Times in seconds for $|\mathcal{T}_{\text{eu,synth}}| = 10^7$ with $d = 25\,\text{km}$ and different number of threads for spatial only query.

threads	1/2	1/4	1/8	1/16	1/32
1	8.334	4.987	2.029	0.864	0.150
2	4.811	3.788	1.537	0.703	0.122
4	2.722	1.999	0.830	0.348	0.072
8	1.351	1.067	0.447	0.210	0.056
16	0.801	0.624	0.273	0.130	0.037
24	0.632	0.495	0.196	0.099	0.032

we have done in Figure 6 – essentially dividing the query times of Table 7(a) by the result set sizes of Table 7(e). We see that the time per reported trajectory is always in the *microseconds* range.

In Table 8 we have similar measurements as in Table 6 but on larger scale. In comparison, the additional time constraints do not speed up the query times by the same amount, but the general trend stays the same and proves the scalability of PATHFINDER.

In Table 9 we ran only spatial queries with different numbers of threads. In Figure 7 these times are normalized and plotted. We can see that our algorithm has only a small speedup from 1 to 2 threads because $|\mathcal{T}_{out}|$ does not need to be merged in the single-threaded case. For larger rectangles in particular we see an almost linear speedup when using more threads.

Table 10: On disk and RAM results for step GetAssociatedTrajectories and $|\mathcal{T}_{out}|$, single thread.

	1/2	1/4	1/8	1/16	1/32
On disk (s)	36.33	22.60	9.35	3.86	1.53
RAM (s)	19.20	10.67	3.56	1.11	0.26
$\lvert\mathcal{T}_{out}\rvert$ (M)	16.49	12.46	6.80	3.46	1.42

Index on Disk. Although we designed PATHFINDER as an in-memory index we also considered the case that RAM is limited and therefore implemented a variant which stores the trajectory data on disk by using the STXXL-library [3]. We generated a huge dataset of $|\mathcal{T}_{\text{ger,synth}}| = 75 \cdot 10^6$ with $d = 100\,\text{km}$ which uses 358 GB on the SSD of our threadripper machine and ran spatial queries there and also completely in the RAM of our Xeon machine for comparison. The results with respect to the step GetAssociatedTrajectories are shown in Table 10. We see that fetching from a (very fast) NVMe SSD only incurs an overhead of at most a factor of 6. Note that in both cases we assume the search structure (essentially the augmented CH) to be resident in RAM, yet the number of trajectories can be increased almost arbitrarily according to the capacity of the SSD.

Precision With Respect to Temporal Queries. In [10] it was argued (but not experimentally verified) that fine-grained temporal information has only limited value for spatio-temporal filtering. We verify this conjecture by analyzing the *interval* queries from Table 6, in particular how many more trajectories are reported due to non-exact temporal information stored in the CH representation, see Table 11. We see for example that for random queries of size $\frac{1}{8} \times \frac{1}{8}$ with a random slot out of 64 slots within a week, the average number of trajectories reported is 98.498 compared to a ground truth of 98.234. This is due to the fact that if a trajectory is not fully contained in the spatial query rectangle, our temporal compression might report trajectories which should not be reported (this is the

Table 11: Size of result set $\mathcal{T}_{\text{out}}$: quality loss due to decrease of temporal resolution (averaged over 2000 queries).

	1/2	1/4	1/8	1/16	1/32
uncompressed	959.8600	313.8005	98.2340	28.5760	9.6105
compressed	960.3985	314.2435	98.4980	28.7365	9.6890
$\frac{\text{uncompressed}}{\text{compressed}}$	0.999	0.998	0.997	0.994	0.991

only possible reason). We see, though, that this happens very rarely. We can also see that bigger query rectangles have a higher precision because their boundary-to-area ratio is smaller.

Comparison to SPNET. To make a comparison with SPNET, we have a closer look at the result for one thread and rectangle size 1/32 in Table 9 which is 150 ms. For SPNET, such queries were benchmarked in [10] with better hardware, on a smaller graph and with less trajectories which were also shorter on average. Note that the input for SPNET's range query is not a rectangle but the set of edges in a rectangle. Therefore, their query is strictly easier than ours since we have to compute the edges in the rectangle first, which is exactly what FindEdgeCandidates does. For the rectangle used in our query with size 1/32, we counted the edges within the rectangle which were on average 5.18×10^6. In Figure 5 (d) of the SPNET paper [10], we compare with the result for $\mathbf{R}_{19}$ which corresponds to rectangles containing up to 2^{19+1} edges. Consequently, we consider more edges since $2^{19+1} < 1.05 \cdot 10^6 < 5.18 \cdot 10^6$. Figure 5 (d) states that for $\mathbf{R}_{19}$, SPNET requires on average more than 10^3 ms, whereas Pathfinder requires 150 ms under harder conditions.

6 CONCLUSION AND FUTURE WORK

We built a novel framework which delivers high compression rates for heterogeneous network constrained trajectory data. However, the main achievement of our framework is to speed up spatio-temporal range queries tremendously. An advantage of Pathfinder is that it is built upon a data structure which can also be used to speed up routing related queries. We demonstrated that Pathfinder is highly parallelizable with almost linear speedup and able to deal with much larger data sets than previous work.

We also investigated storage of trajectories in external memory on an SSD. This is possible since only the CH has to reside in RAM, and the memory consumption of the CH is *independent* of the size of the trajectory set. Since the latter can then be stored on an SSD, we are only limited by the SSD capacity for the trajectory set and the RAM for the CH representation of the network. The runtime penalty for external storage of the trajectory set is moderate in case of a fast NVMe PCIe SSD.

Pathfinder drops part of the temporal information. Even though we experimentally validated the conjecture by [10] that this does not significantly affect the query results, one can simply replace some long shortcuts by shorter shortcuts or even the original edges they represent if higher precision is required at certain places. This leads to more accurate temporal resolution at the cost of higher space consumption and query time. It might be interesting to see whether more sophisticated approaches can increase the temporal resolution without affecting query times and space consumption too much.

Also, we have not discussed how to deal with dynamic updates (in particular insertion) of trajectories. Fortunately, the main components of our index structure are trajectory oblivious in a sense that they do not depend on the trajectory set to be indexed. Only the marking of obsolete and tree edges actually depends on the trajectory set. So with little effort, we are be able to cater for insertions and deletions of trajectories. Changes to the underlying road network are unfortunately more difficult to handle, and we leave this as possible future work.

Acknowledgements This work was in part supported by the Deutsche Forschungsgemeinschaft (DFG) within the priority program 1894: Volunteered Geographic Information: Interpretation, Visualization and Social Computing.

REFERENCES

[1] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, 3rd ed. edition, 2008.
[2] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On Map-matching Vehicle Tracking Data. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 853–864, Trondheim, Norway, 2005. VLDB Endowment.
[3] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: standard template library for xxl data sets. *Software: Practice and Experience*, 38(6):589–637, 2008.
[4] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, June 1984.
[5] S. Funke, N. Schnelle, and S. Storandt. Uran: A unified data structure for rendering and navigation. In *Web and Wireless Geographical Information Systems*, pages 66–82, Cham, 2017. Springer International Publishing.
[6] S. Funke and S. Storandt. Provable efficiency of contraction hierarchies with randomized preprocessing. In *ISAAC*, volume 9472 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2015.
[7] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transport. Science*, 46(3):388–404, 2012.
[8] R. H. Güting, V. T. de Almeida, and Z. Ding. Modeling and querying moving objects in networks. *The VLDB Journal*, 15(2):165–190, Jun 2006.
[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57. ACM, 1984.
[10] B. Krogh, C. S. Jensen, and K. Torp. Efficient in-memory indexing of network-constrained trajectories. In *Proc. 24th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 17:1–17:10. ACM, 2016.
[11] B. Krogh, N. Pelekis, Y. Theodoridis, and K. Torp. Path-based queries on trajectory data. In *Proc. 22nd ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 341–350. ACM, 2014.
[12] Y. Li, Q. Huang, M. Kerber, L. Zhang, and L. Guibas. Large-scale Joint Map Matching of GPS Traces. In *Proc. 21st ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 214–223. ACM, 2013.
[13] M. A. Quddus, W. Y. Ochieng, and R. B. Noland. Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transportation Research Part C: Emerging Technologies*, 15(5):312 – 328, 2007.
[14] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, and S. Vial. Indexing in-network trajectory flows. *The VLDB Journal—The International Journal on Very Large Data Bases*, 20(5):643–669, 2011.
[15] M. P. Seybold. Robust map matching for heterogeneous data via dominance decompositions. In *Proc. SIAM International Conference on Data Mining*, pages 813–821, 2017.
[16] R. Song, W. Sun, B. Zheng, and Y. Zheng. Press: A novel framework of trajectory compression in road networks. *Proc. VLDB Endow.*, 7(9):661–672, May 2014.
[17] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. Sadiq. Sharkdb: An in-memory column-oriented trajectory storage. In *Proc. 23rd ACM Int. Conf. on Information and Knowledge Management*, pages 1409–1418. ACM, 2014.
[18] X. Yang, B. Wang, K. Yang, C. Liu, and B. Zheng. A novel representation and compression for queries on trajectories in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 30(4):613–629, 2018.
[19] Y. Zheng. Trajectory data mining: An overview. *ACM Trans. Intell. Syst. Technol.*, 6(3):29:1–29:41, May 2015.