World Wide Web (2021) 24:51–83
https://doi.org/10.1007/s11280-020-00831-w

# Indexing and progressive top-*k* similarity retrieval of trajectories

**Nikolaos Pliakis**[1] · **Eleftherios Tiakas**[1] · **Yannis Manolopoulos**[2]

Received: 26 July 2019 / Revised: 18 April 2020 / Accepted: 15 June 2020 /
Published online: 6 July 2020

**Abstract**
In this work, we study the performance of state-of-the-art access methods to efficiently store and retrieve trajectories in spatial networks. First, we study how efficiently such methods can manage trajectory data to support indexing for data demanding applications where trajectory retrieval must be fast. At the same time, trajectory insertions, deletions and modifications should also be executed efficiently. Secondly, we compare the performance of *progressive* processing of trajectory similarity top-*k* queries, which is a common query in spatial applications. Specifically, we examine FNR-trees (Frentzos 2003) and MON-trees (de Almeida and Gueting, 2005), which have been proposed for trajectory management, against a novel variation of our proposed Cluster-extended Adjacency Lists (CeAL) (Tiakas and Rafailidis 2015). In particular: (a) we extend the above access methods to efficiently handle trajectories of objects that move in large spatial networks, and (b) to enhance their performance, we create an entirely new implementation framework to generate trajectories and to test the trajectory management and retrieval for each approach. With respect to the generation of trajectories, we extend the generator by Brinkhoff (2000) to efficiently support very large spatial networks. Finally, we conduct extensive experimentation which demonstrates that the proposed method CeAL prevails in space and time complexity.

**Keywords** Location-based services · Trajectory · indexing · Spatio-temporal queries · Progressive processing · Top-*k* queries · Similarity retrieval

✉ Eleftherios Tiakas
tiakas@csd.auth.gr

Nikolaos Pliakis
pliakis.nick@gmail.com

Yannis Manolopoulos
yannis.manolopoulos@ouc.ac.cy

[1] Department of Informatics, Aristotle University, Thessaloniki 54124, Greece

[2] Faculty of Pure and Applied Sciences, Open University of Cyprus, Nicosia 2220, Cyprus

## 1 Introduction

During the last decade there has been an increased demand for applications in *spatial networks* with moving objects and trajectories; from *navigation systems*: road/ river/railway networks, traffic analysis, map destination, shortest routes, location-based services, to *energy-resources networks*: oil, electrical power network, natural gas, telephone lines, water-sewer systems, etc.

Spatial networks are characterized by topological restrictions, since moving objects follow specific routes to reach a destination. Even in aviation, flights respect predefined air routes. *Moving objects* may have properties which may affect their status through the network (e.g. speed, congestion tolerance and priorities, obeying certain network restrictions) or may not (e.g. ID, label, color).

A spatial network can be modeled as a *graph* with a set of nodes connected with edges.[1] The main advantage in graph representations is that usually there is no need for the graph topology to perfectly match the real geography. Most applications require only the existence of nodes and weighted edges. Usually a moving object has a starting node and a destination node, plus it may have to visit some in-between network nodes. The path followed by every object, respecting the restrictions enforced by the network, is called object *trajectory*.

Modern spatial applications involve big data; thus sophisticated indexing and processing methods are required for efficient data management. When new trajectories have to be inserted, or old trajectories have to be modified/deleted the indexes must efficiently support these operations. Among the spacial access methods, a well honored popular family of such indexes are the R-tree-based methods.

Another vital requirement in spatial applications is the efficient query processing of trajectories. In particular, a popular query is top-$k$ similarity retrieval of trajectories, i.e. given a trajectory (or some spatial locations in the network and time restrictions for the transition between nodes) we want the top-$k$ most similar trajectories to the given one (or passing close to the spatial locations).

In emergency applications the query response time is crucial. Therefore, another desired property is the *progressive* query processing, i.e. the results are provided to the user in a incremental manner: when a trajectory satisfies the space and time restrictions, it is provided to the user, while the next results are being prepared.

In the area of trajectory query processing there are some major challenges. A large number of the proposed approaches require a preprocessing to precompute all-to-all pairwise shortest paths in the spatial network, with significant space and time costs, especially in large networks. To avoid this preprocessing some methods ignore the network restrictions by taking the Euclidean node distances to provide preliminary results, which require subsequent filtering.

Another issue is that real-life applications generate big amounts of data in short time. For instance, we can consider the number of trajectories generated in the road network of a medium-sized city during a single day. In such a case, it is crucial to efficiently suggest trajectories to the moving objects in the network. While the existing spatiotemporal indexing methods may be quite sufficient in small networks and trajectory datasets, their performance degrades and becomes quite inefficient in very large networks. Moreover, complex indexes require even higher processing cost to manage the trajectories. Therefore, the challenge is to have simple and efficient indexes for big data.

[1]The terms *network* and *graph* will be used alternatively, as well as *node* and *vertex*.

The most popular spatial access methods for such settings are either tree-based (R-tree, M-tree, etc.) or methods that exploit other structures. A distinct category is comprised of progressive trajectory similarity search methods, which can highly reduce the on-line query processing time, due to the fact that not all top-$k$ results need to be retrieved, if users find the already retrieved results satisfactory.

To overcome the weaknesses of existing trajectory similarity search approaches, we propose an elaborated variation of our method *Cluster-extended Adjacency List (CeAL)* [32]. CeAL has been used to enhance location-based trajectory similarity top-$k$ queries, where a user provides the query locations in a spatial network along with time restrictions, and the top-$k$ similar trajectories to the locations that satisfy the time restrictions are provided to the user in a progressive manner. Here, the original CeAL method is modified and applied as a core indexing scheme to support efficient trajectory management and retrieval in very large spatial networks. This is achieved by facilitating trajectory similarity searching by taking into account both spatial and temporal restrictions between nodes.

The proposed variation of CeAL inherits the advantages of its original version [32] and, in addition, it is enhanced with the following characteristics:

(a) User-defined locations and time restrictions have been replaced by recorded spatial positions and timestamps of moving objects. Automatically calculating and recording locations and time restrictions through the applied moving objects framework enhances CeAL.
(b) Trajectory similarity calculations have been significantly reduced, by avoiding the computation of all-to-all shortest path distances as most of the previous methods do, by limiting the calculation of pairwise node distances from the small set of the selected spatial locations to the nodes of the spatial graph.
(c) CeAL consumes linear space, since adjacent nodes are connected directly to stored trajectory data on disk, without building any complex index.
(d) The trajectories can be provided in a personalized manner by means of a proposed spatio-temporal similarity measure adaptable to the user preferences by tuning the query to be more spatial- or more temporal-oriented. Additionally, users can set weights to spatially prioritize the selected spatial locations.
(e) finally, CeAL is established with all necessary theorems and proofs for its properties and its complexity.

The main contributions of this study are:

- We propose a novel variation of algorithm CeAL for the progressive processing of top-$k$ trajectory similarity queries in spatial networks. CeAL can operate in on-line environments, as the on-line query processing cost has been reduced due to its progressive approach, which enables early termination if adequate results have been reported to the user.
- We propose a new spatio-temporal similarity measure, which satisfies the generalized metric properties and can also be used in other related problems, where the metric properties are applied to efficiently prune the search space.
- We provide an extensive comparison of CeAL against other state-of-the-art methods with respect to their space and time performance. To this end, we use an enhancement of the Brinkhoff's classic generator [4], which takes into account the spatial and temporal restrictions.

The structure of the sequel is as follows. The next section provides an overview of the relevant literature, whereas Section 3 gives basic definitions, notations and assumptions. Alternative access methods are described in Section 4, whereas Section 5 focuses on the particular implementations. Section 6 reports the results of an extensive experimentation. Finally, the last two sections conclude the paper and discuss further extensions.

## 2 Related research

### 2.1 Trajectory generation

Generating trajectories of moving objects in spatial networks must be realistically designed. A classic trajectory generator has been proposed by Brinkhoff [4], which takes into account the spatial restrictions imposed by the underlying network and the temporal restrictions due to characteristics of the moving objects. Some important aspects in the process of the generation are the maximum speed of the moving objects, the influence of the other moving objects to the speed and the route of an object, the maximum capacity of connections, the adequate determination of the start and destination of an object, the influence of external objects and events, and time-scheduled traffic. The generator is written in Java; to enhance further the generator performance in very large networks, we extended this method in a new implementation framework (C++, Boost Graph library [5]).

### 2.2 Trajectory indexing and retrieval

Access methods for trajectory management must be efficient during query processing in spatial networks. To this end, there are several relevant works proposed in the literature. In [20, 27] the trajectory of a moving object is represented as a set of graph edges followed by the object during its lifetime. Also, of interest is the time interval during which the moving object traverses a specific edge. Additionally, two kinds of transformation techniques are proposed for network data and for trajectory data. Both techniques store the data in R-trees. An alternative way to represent the trajectory data is to store the visited nodes along with the corresponding time instant when the visit takes place [11].

In [37] the notion of multi-attribute trajectories is studied, i.e. standard trajectories with descriptive attributes. Multi-attribute trajectories are indexed in a 3D R-tree and a composite structure which can be adapted to work with any R-tree-based or Grid-based index. However, the article focuses in the problem of continuous *k* nearest neighbor queries over the data trajectories and proposes efficient algorithms for query processing.

Several studies model trajectories as time series using transformation techniques, where trajectory similarity search is performed by using either distance measures or subsequence matching [1, 7, 8, 14, 21, 25, 29, 35, 38]. However, these works suffer from a high cost of similarity calculations. Therefore, pruning or approximation methods have been proposed to decrease the computational cost. Most of the works on similarity search assume Euclidean spaces, either transformed or not, using R-trees or variants [24]. The works of [9, 16, 22] introduce query processing algorithms for similarity search in trajectory data ignoring, however, the temporal domain.

The works of [6, 17] retrieve trajectories similar to a query trajectory in both spatial and temporal domains. In particular, similarity calculations and optimization techniques, such as pruning and bounding, are performed in Euclidean space, which contradicts the nature of spatial networks. On the other hand, the works of [33, 34] perform trajectory similarity search in spatial networks using M-trees [10], to prune the search space based on metric functions.

With respect to the storage of trajectories, several methods have been proposed. Frentzos et al. use FNR-trees which is a 2-d R-tree storing graph edges, whereas its leaves point to the roots of 1-d R-trees, which store the visits of each specific edge [15]. In [31], trajectories are considered as sets of points in the Euclidean space and are indexed with R-trees. The algorithm returns the $k$ most similar trajectories by a set of predefined point locations, and uses a heap to retrieve candidate trajectories from each individual query point. In the sequel, the candidate trajectories are refined according to specific bounds. The particular algorithm is based on the methodology of Fagin's Threshold Algorithm [13]. The main distance measure is an aggregation of the distances from the query points to the corresponding shortest trajectory points. The idea of this aggregation has been also studied in [26].

With respect to similarity measures, mainly the spatial attributes of the trajectories are taken into consideration, with temporal data becoming relevant only occasionally. A query processing algorithm returns the most similar trajectories by searching over a set of candidate trajectories. Although a variety of trajectory similarity measures has been proposed, most of them apply specific measures. For instance, some widely used similarity measures are: Euclidean distance [1], Discrete Fourier Transformation and Wavelets [7], Edit Distance and its variations [8, 9], Longest Common Subsequence [35] and Dynamic Time Warping [38].

The existing spatiotemporal indexing methods are quite sufficient in small networks and trajectory datasets, but their performance gradually degrades and becomes inefficient in very large networks. Moreover, the more complex the index, higher the processing cost to manage the trajectories. Tiakas et al. alternatively suggest that the network is represented by a structure based on adjacency lists at a preprocessing step [32]. This is the original version of CeAL. In each edge formed at preprocessing, a cluster is assigned, which contains references to all the trajectories that pass from that particular edge. This way a simple and efficient index is constructed to handle large data.

## 3 Preliminaries

Here we introduce the basic terminology and assumptions, as well as we present the main tasks and formulate the main problem.

### 3.1 Definitions

A spatial network can be represented as a *graph* $G(V, E)$ consisting of a set of *vertices* $V$ and a set of *edges* $E$. On a 2-d plane every vertex can also be defined by its coordinates as $(x_i, y_i) \in V$. Every edge can be defined as $(v_i, v_j) \in E$ and represents a connection between $v_i$ and $v_j$. We assume that the network is a static connected undirected graph. For real data, edges are weighted, i.e. a weight $w(v_i, v_j)$ is given to any pair of neighbor nodes $v_i, v_j$, to represent the distance between them or the time spent to travel from one to the other, and so on.

We also assume that the distance between two non-neighbor nodes $v_i, v_n$ equals the sum of the weights of all the edges in the path: $w(v_i, v_j) + w(v_j, v_k) + \ldots + w(v_m, v_n)$. If there are several paths connecting the two nodes, then of importance is the *geodesic* path which is the shortest one with a distance called *network distance*. We normalize this distance by dividing with the *network diameter* (the maximum shortest path distance between any two nodes), to have a distance measure in the interval [0, 1]. We will denote this normalized distance between any two nodes $v_a, v_b$ of the network as $d(v_a, v_b)$.

This distance measure $d(\cdot)$ satisfies the metric properties:

- *Non-Negativity:* Any transition from vertex $v_a$ to vertex $v_b$ has a non-negative cost. Therefore, it holds that: $d(v_a, v_b) \geq 0$, whereas $d(v_a, v_b) = 0 \Leftrightarrow v_a = v_b$.
- *Triangular Inequality:* For any three nodes $v_a, v_b, v_c$ it holds that: $d(v_a, v_b) \leq d(v_a, v_c) + d(v_c, v_b)$.
- *Symmetry:* Since the network is undirected, the distances from node $v_a$ to node $v_b$ and vice-versa are equal: $d(v_a, v_b) = d(v_b, v_a)$.

We denote a trajectory as $T_i$ which is part of a trajectory set $T$: $T_i \in T$. Each trajectory $T_i$ has its own length $r_i$ of spatial points, which is called *description length*. We assume that the trajectories have an arbitrary description length, which means that for two different trajectories $T_i, T_j$, it may hold that $r_i \neq r_j$.

We assume that the spatial points of the trajectories lay on the nodes of the spatial network. Otherwise, if the spatial points of the trajectories lie on the edges, then they can be aligned to the closest nodes using map-matching methods [2, 3, 18, 23, 36]. This matching does not affect the proposed methods, since it can be performed in a preprocessing step, while generating the trajectory data.

Therefore we can define a trajectory as an ordered set of $r_i$ pairs, which correspond to nodes $v_i$ visited during the network traversal along with the time instances $t_{v_i}$ that the visit takes place:

$$T_i = \{(v_{i1}, t_{v_{i1}}), (v_{i2}, t_{v_{i2}}), \ldots, (v_{ir_i}, t_{v_{ir_i}})\} \tag{1}$$

We consider a node visit to be an instantaneous event with zero time elapsed, i.e. we ignore the time spent by an object in any node. Defining a trajectory requires only the total time spend by the object when moving from one node to another within the network limits.

Finally, the *multiset*[2] of the spatial points from all trajectories is denoted as $R$, and the multiset of all trajectory edges as $RE$. Both multisets $R$ and $RE$ represent the raw trajectory data, and it holds that:

$$|R| = \sum_{i=1}^{|T|} r_i \qquad |RE| = \sum_{i=1}^{|T|} (r_i - 1) = |R| - |T|$$

### 3.2 Indexing and managing trajectories data

A focus of this work is to estimate the efficiency of the examined indexing methods to manage trajectories. In particular, we study their efficiency in supporting dynamic environments where new trajectories have to be inserted or old trajectories have to be modified/deleted, as well as their consumed space to index data. We will use large networks to study their behavior along with the respective algorithms. We will also conduct experiments in very large real road networks, which are relatively sparse but with millions of nodes and edges.

### 3.3 Problem definition for trajectory similarity top-*k* queries

Let $G$ be the underlying graph of an undirected network and $T$ a trajectories dataset. Let $Q$ be a set of query locations $q_1, q_2, \ldots, q_m$ which are spatial points (nodes of $G$), that the

[2]Multiset is a generalization of the notion of set in which members are allowed to appear more than once.

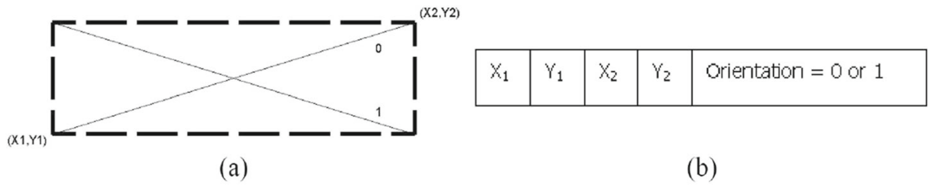


**Figure 1** **a** Line segment of a corresponding edge encapsulated by an $MBR$, **b** Index node representing the $MBR$ containing the edge

resulted trajectories have to pass as close as possible. Let also $qt_2, qt_3, \ldots, qt_m$ be the corresponding inter-arrival times which are $m-1$ tolerance time intervals, acceptable by users for travelling between the query locations ($qt_i = \infty$ denotes the lack of time restriction for the transition to location $q_i$). Let $w_1, w_2, \ldots, w_m$ be the users' predefined weights, expressing the personal preference of importance to the $m$ query locations, where $0 < w_j < 1$ for $j = 1, ..., m$ and $\sum_{j=1}^{m} w_j = 1$. Given a similarity function $sim(Q, T_i)$ between the set $Q$ of query locations and a trajectory $T_i \in T$, the goal is to find the $k$ most similar trajectories in $T$ with the highest similarity score to $Q$.

For this study the query locations $q_1, q_2, \ldots, q_m$ and the corresponding inter-arrival times $qt_2, qt_3, \ldots, qt_m$ can alternatively be given through a query trajectory $T_q$, where its nodes define the query locations and its time instances define the corresponding inter-arrival times.

## 4 Indexing methods and algorithms

The mostly used indexes for trajectories are based on R-trees [19] and their variants [24]. R-trees group nearby spatial objects in a minimum bounding rectangle, $MBR$, which is a key concept in all R-tree-based algorithms. Figure 1 illustrates an example of $MBR$, which is a rectangle that encapsulates an edge in such a way that each of $min(x), max(x), min(y), max(y)$ will be in contact with the respective rectangle side. An $MBR$ can also encapsulate a trajectory object with all its nodes and edges. Moreover, in upper R-tree levels there are also $MBR$'s that enclose lower level $MBR$s.

In the sequel, we will present three indexing methods for moving objects, two of them based on R-trees, and one based on adjacency lists. These methods have been previously tested experimentally in small-scale networks. However, here the efficiency and the performance of these methods will be stressed in networks of large sizes, e.g. in the order of millions of links/edges.

### 4.1 Fixed network R-trees

FNR-trees are height balanced structures based on R-trees [15]. The idea is that any network with $n$ links can be represented as a forest of 1-d R-trees,[3] having a single 2-d R-tree on top. The 2-d R-tree is used to index the graph edges; i.e., every 2-d R-tree leaf represents a single graph edge and stores a pointer to a 1-d R-tree, which indexes the temporal intervals during which a moving object traveled through the particular edge represented by that leaf.

[3] 1-d R-trees can be viewed as having flat $MBR$s to store points in 1-d space, i.e. on a line.

The FNR-tree can support efficient insertion and deletion of trajectories data. For the insertion process it uses Guttman's search algorithm on the top-level 2-d R-tree to find the relevant graph edge encapsulated by the appropriate $MBR$. This leads to a 1D R-tree containing the object visits. Since time is increasingly monotonously, time intervals will be inserted in an increasing order. Thus, we can insert the new element on the bottom-most, right-most tree node without performing a search. This optimization leads to full 1-d R-tree leaves, and minimizes the leaf overlap. Without this optimization, space utilization in the 1-d R-trees is around 65%, whereas with this implementation it increases to 96% [15].

To perform a spatio-temporal query against an FNR-tree, a 3-d interval is used that consists of two spatial points and one temporal point. Thus, the query can be defined as: $((x_1, y_1), (x_2, y_2), (t_1, t_2))$. For the search process, Guttman's search algorithm is executed on the top level 2-d R-tree, and the edges bound by the spatial interval represented by the rectangle provided by the user as a query are identified. After recovering the leaf nodes representing these edges, the edges are stored in memory. Then Guttman's search algorithm is executed in each of the 1-d R-trees which are pointed by the previously recovered leaves, and the corresponding edges are retrieved. If there are edges that are completely outside the query spatial window rectangle, they are discarded.

## 4.2 Moving objects in network trees

MON-trees comprise of a 2-d R-tree with leaves pointing to lower level 2-d R-trees (see Figure 2), which index the moving objects and their trajectories [11]. At the upper level of MON-trees, there is a hash structure with entries in the form *(polyid, bottreeptr)*, where *polyid* is the unique trajectory ID and acts as key, whereas *bottreeptr* points to the lower level R-tree which indexes that current trajectory.

The upper R-tree leaves are of the form *($MBR$, polypt, treept)*, where *MBR* is the $MBR$ acting as a box for the trajectory, *polypt* is a pointer to the trajectory itself, and *treept* is a

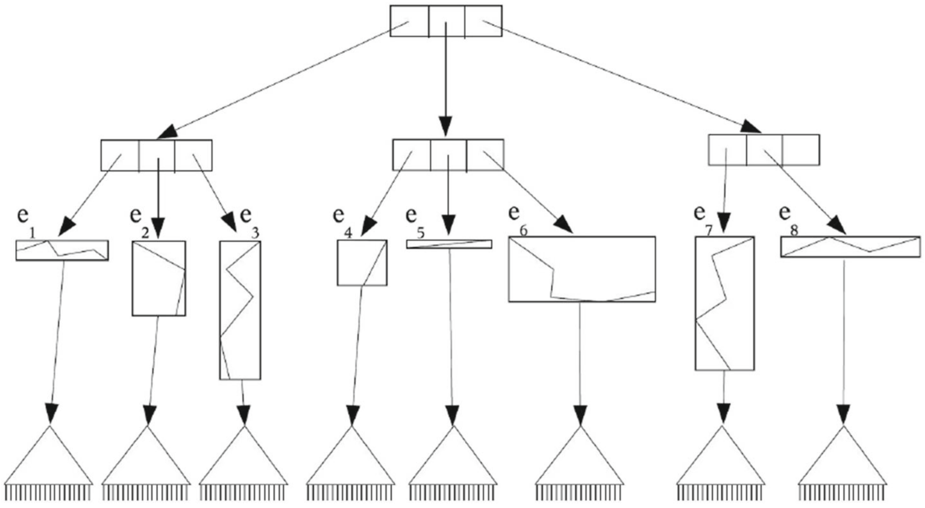


**Figure 2** MON-tree overview [11]

pointer to the lower level R-tree. The internal nodes have the form *(MBR, childpt)*, where *MBR* is the $MBR$ enclosing the $MBR$s of descendant nodes, and *childpt* is a pointer to the descendant node.

The lower level R-trees index the object trajectories. This is achieved with two intervals: the spatial interval $(p_1, p_2)$ (where $0 \leq p_1, p_2 \leq 1$), and the temporal interval $(t_1, t_2)$. A combination of the two intervals gives the position of the moving object within the time interval defined by the two time points $t_1$ and $t_2$.

Searching is based on a spatio-temporal window: $wnd = (x_1, x_2, y_1, y_2, t_1, t_2)$ and can be interpreted as: retrieve the moving objects within the space bounded by the rectangle $r = (x_1, x_2, y_1, y_2)$, during the time period $t = (t_1, t_2)$. To this end, the process is split into its orthogonal parts, the spatial and the temporal one. First, a search is performed on the top R-tree to retrieve all $MBR$s which intersect the rectangle defined by the spatial part of $wnd$. The result is a set of windows: $wnd' = \{(p_{1_1}, p_{1_2}, t_1, t_2), \ldots, (p_{n_1}, p_{n_2}, t_1, t_2)\}$ as shown in Figure 3, where $n$ is the number of elements, $p_n$ the position of the moving object, and $t_1, t_2$ the time interval given as input to the query. After retrieving these network portions, a search is performed on each of them, based on the time interval. As seen in Figure 3, the trajectories retrieved by the initial search are examined to determine on which parts they intersect the time interval provided as part of the spatio-temporal window.

The insertion process takes a trajectory ID as input and uses the hash structure to discover the lower level R-tree which corresponds to that trajectory. Searching is accomplished by a spatio-temporal window, which defines the spatial and temporal intervals of interest. The search algorithm begins from the top level R-tree root, which narrows the search down to the $MBR$s of each trajectory. If there is no bottom level R-tree for inserting this trajectory, then a new R-tree node is created, and the pointer of the trajectory is inserted in the hash structure.

## 4.3 Cluster-extended adjacency lists

CeAL uses an adjacency list to model the network on which the trajectories will be mapped. Trajectory clusters are assigned to each node of the list, storing the trajectories that pass through it. In our variation of CeAL, query processing can be done in both following ways: (a) the user can define specific spatial locations and time restrictions as well optional weights of importance for the locations, (b) the user can input a query trajectory.

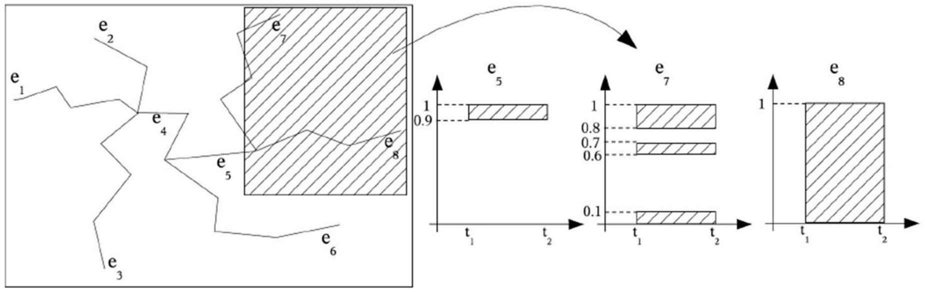


**Figure 3** MON-tree search by spatio-temporal window [11]

### 4.3.1 Creating the CeAL scheme

CeAl has a preprocessing phase to index the trajectories, either during generation, or extracted from a pre-compiled dataset [32]. This phase indexes the edges with adjacency lists: for each edge $(v_i, v_j)$ a cluster $C_{ij}$ is created to store all the trajectories passing through this edge. If no trajectory passes through a specific edge, then the relevant cluster remains empty. The created cluster is assigned to the node $v_j$, the ending point of the edge. Thus, the final structure comprises of $V$ adjacency lists, representing the edges from a specific network node, extended by the clusters containing all the trajectories passing through the edge (see Figure 4).

The clusters are implemented as dynamic lists. Therefore, an initial traversal of the trajectory dataset $T$ is required. For each trajectory $T_i$, its ID is passed as a parameter to a pre-selected hash function. In particular, we used the simple hash function: $ID$ mod $|T|$ to get the disk page location of the trajectory. Then, the trajectory $T_i$ is traversed; during this traversal, we retrieve its edges and store the trajectory's ID in each of the clusters associated with it.

Algorithm 1 presents the preprocessing procedure. The time complexity for reading the trajectory data and the spatial complexity of the preprocessing phase is linear: $O(|V|+|E|+|RE|)$. Also, the created trajectory clusters are generally smaller in size than the structure proposed in [28], where clustering is based on network nodes, which leads to larger clusters. In CeAL, clustering is performed based on the graph edges. Additionally, since hashing is used to store the trajectories, they can be efficiently retrieved when required by using the same hash function.

**Algorithm 1** Preprocessing.

```
1: for all trajectories $T_i \in T$ do
2:     allocate data of $T_i$ through $hash(T_i.Id)$
3:     for all edges $(v_i, v_j) \in T_i$ do
4:         insert $T_i.Id$ in cluster $C_{ij}$
5:     end for
6: end for
```

Figure 5 depicts a small-scale example of a spatial network with 14 nodes, 21 edges and 3 trajectories (see Tables 1 and 2). The outcome of Algorithm 1 is the structure of Figure 6.

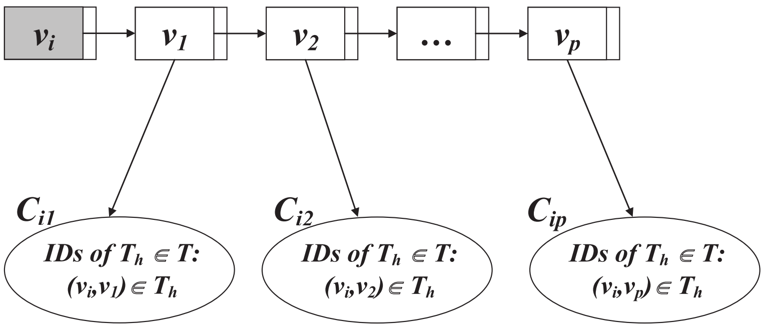
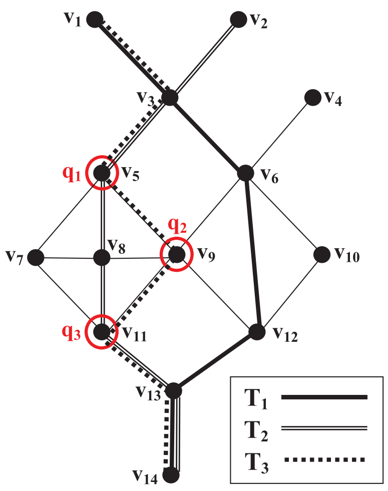

$v_i$
$v_1$
$v_2$
...
$v_p$
$C_{i1}$
$C_{i2}$
$C_{ip}$
IDs of $T_h \in T$: $(v_i,v_1) \in T_h$
IDs of $T_h \in T$: $(v_i,v_2) \in T_h$
IDs of $T_h \in T$: $(v_i,v_p) \in T_h$

**Figure 4** Extended adjacency list index of node $v_i$ with $p$ adjacent nodes and trajectory clusters

**Figure 5** An illustrative small-scale example



### 4.3.2 Trajectory similarity measures

In the proposed CeAL method, trajectory retrieval is based on the similarity between the trajectories and the selected spatial positions and timestamps of the moving objects. Therefore, trajectory retrieval ignites a location-based query with time restrictions. To facilitate searching within CeAL, two new trajectory similarity metrics, $D_s(.)$ and $D_t(.)$, are proposed for the spatial and the temporal dimension, respectively. The spatial similarity measure is used to assess how close a trajectory is to the selected spatial positions $Q$ with respect to the restriction the network imposes on the movement of objects. The spatial distance of a specific location $q_i \in Q$ from a trajectory $T_j \in T$, which passes through the nodes $v_1, v_2, \ldots, v_n$, is defined as the minimum among the distances between the location and each node of the trajectory. This measure is:

$$d_s(q_i, T_j) = \min_{(h=1,\ldots,n)} d(q_i, v_h) = d(q_i, v_{\min}) \tag{2}$$

**Proposition 1** *$d_s(\cdot)$ is a generalized metric function that satisfies the generalized triangular inequality with values in the range of* [0, 1].

**Table 1** Edge weights of the graph in Figure 5

| | | | |
|---|---|---|---|
| $w(v_1, v_3) = 4$ | $w(v_2, v_3) = 3$ | $w(v_3, v_5) = 4$ | $w(v_3, v_6) = 5$ |
| $w(v_4, v_6) = 6$ | $w(v_5, v_7) = 8$ | $w(v_5, v_8) = 5$ | $w(v_5, v_9) = 8$ |
| $w(v_6, v_9) = 3$ | $w(v_6, v_{12}) = 7$ | $w(v_6, v_{10}) = 4$ | $w(v_7, v_8) = 5$ |
| $w(v_7, v_{11}) = 8$ | $w(v_8, v_9) = 5$ | $w(v_8, v_{11}) = 5$ | $w(v_9, v_{11}) = 8$ |
| $w(v_9, v_{12}) = 2$ | $w(v_{10}, v_{12}) = 5$ | $w(v_{11}, v_{13}) = 2$ | $w(v_{12}, v_{13}) = 9$ |
| $w(v_{13}, v_{14}) = 7$ | | | |

**Table 2** Trajectories of the graph in Figure 5

| | |
|---|---|
| $T_1$ | $(v_1, 0) \rightarrow (v_3, 2) \rightarrow (v_6, 4) \rightarrow (v_{12}, 7) \rightarrow (v_{13}, 11) \rightarrow (v_{14}, 14)$ |
| $T_2$ | $(v_2, 0) \rightarrow (v_3, 1) \rightarrow (v_5, 3) \rightarrow (v_8, 5) \rightarrow (v_{11}, 7) \rightarrow (v_{13}, 8) \rightarrow (v_{14}, 11)$ |
| $T_3$ | $(v_1, 0) \rightarrow (v_3, 3) \rightarrow (v_5, 6) \rightarrow (v_9, 11) \rightarrow (v_{11}, 16) \rightarrow (v_{13}, 17) \rightarrow (v_{14}, 21)$ |

*Proof* It is sufficient to prove that the following properties hold for any location $q_j \in Q$, for any node $x \in V$ and for any trajectory $T_i \in T$:

1. $0 \leq d_s(q_j, T_i) \leq 1$
2. $d_s(q_j, T_i) = 0 \Leftrightarrow q_j \in T_i$
3. $d_s(q_j, T_i) \leq d(q_j, x) + d_s(x, T_i)$

Let $vmin_j$ be the corresponding node of the trajectory $T_i$ with the minimum distance from the location $q_j$ (see Figure 7). Since $d_s(q_j, T_i) = d(q_j, vmin_j)$ and the spatial function $d(\cdot)$ is in the range of [0, 1], therefore the same holds for the $d_s(\cdot)$ function, i.e. $0 \leq d_s(q_j, T_i) \leq 1$. Moreover, it holds that: $d_s(q_j, T_i) = 0 \Leftrightarrow d(q_j, vmin_j) = 0 \Leftrightarrow q_j = vmin_j$ (property of the $d(\cdot)$ function). Therefore, since $vmin_j$ is a node of $T_i$ we have: $q_j \in T_i$.

For the proof of the generalized triangular inequality, $x$ is a random graph node where the closest node of trajectory $T_i$ to $x$ is not necessary node $vmin_j$ (Figure 7). Let $vmin_x$ be the corresponding node of the trajectory $T_i$ which has the minimum distance from node $x$. Then, we have: $d_s(q_j, T_i) = d(q_j, vmin_j)$ and $d_s(x, T_i) = d(x, vmin_x)$. Thus, it is sufficient to prove:

$$d_s(q_j, T_i) \leq d(q_j, x) + d_s(x, T_i) \;\Leftrightarrow\; d(q_j, vmin_j) \leq d(q_j, x) + d(x, vmin_x)$$

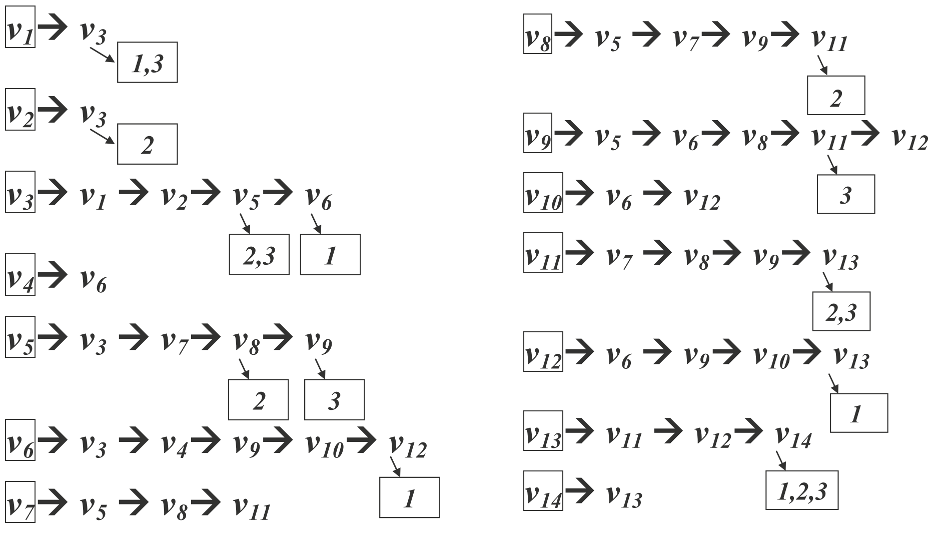

v1 → v3 [1,3]
v2 → v3 [2]
v3 → v1 → v2 → v5 → v6 [2,3] [1]
v4 → v6
v5 → v3 → v7 → v8 → v9 [2] [3]
v6 → v3 → v4 → v9 → v10 → v12 [1]
v7 → v5 → v8 → v11
v8 → v5 → v7 → v9 → v11 [2]
v9 → v5 → v6 → v8 → v11 → v12 [3]
v10 → v6 → v12
v11 → v7 → v8 → v9 → v13 [2,3]
v12 → v6 → v9 → v10 → v13 [1]
v13 → v11 → v12 → v14 [1,2,3]
v14 → v13

**Figure 6** Outcome of Algorithm 1 on the structure of Figure 5

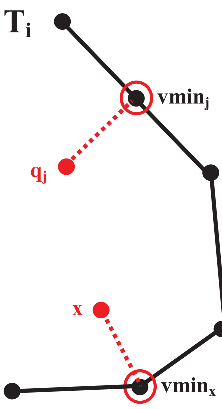


**Figure 7** Proof of generalized triangular inequality

Since $vmin_j$ is the closest node between the rest of nodes of trajectory $T_i$ to $q_j$ (including $vmin_x$), it holds that:

$$d(q_j, vmin_j) \leq d(q_j, vmin_x)$$

Moreover, since function $d(\cdot)$ satisfies the triangular inequality for node $x$, it holds that:

$$d(q_j, vmin_x) \leq d(q_j, x) + d(x, vmin_x)$$

By combining the last two inequalities we reach to the generalized triangular inequality:

$$d(q_j, vmin_j) \leq d(q_j, x) + d(x, vmin_x)$$

□

Each included location may have a different distance from the trajectory, which means that this distance will be calculated separately for each location. Our objective is to have at least $j$ nodes as close to the location $q_j$ as possible. Therefore, we calculate the sum of the distances of all locations from the trajectory, which allows approximating the total distance of the trajectory from the spatial locations into consideration. Consequently, the spatial similarity metric is defined as the average distance between all locations and the trajectory, and can be calculated as:

$$D_s(Q, T_i) = \frac{1}{m} \sum_{j=1}^{m} d_s(q_j, T_i) \tag{3}$$

An alternative approach is when the user can provide the importance on each spatial location. In this case, the calculated spatial similarity distances are multiplied by the assigned weight of importance, which take values in the interval (0,1) and have sum 1, and show the contribution of each location to the total similarity. The more the weight approaches 0, the less important it is; on the contrary, the more it approaches 1, the more it contributes to the final distance calculation. In this case, the metric is:

$$D_s(Q, T_i) = w_1 d_s(q_1, T_i) + \ldots + w_n d_s(q_n, T_i) \tag{4}$$

Thus, if a spatial location is considered more important, then its weight will be close to 1. If the distance of that location is large, it will affect the spatial measure adversely. The opposite case, where even though the distance is large, the weight is close to 0, will mean that its effect on the final value of the spatial measure will be less severe, reducing the impact its distance has to the final trajectory similarity score. It goes without saying that the opposite is also true.

**Proposition 2** $D_s(\cdot)$ *is a generalized metric function that satisfies the generalized triangular inequality in the range* [0, 1].

*Proof* It is sufficient to prove that the following properties hold for any node $x \in V$ and for any trajectory $T_i \in T$:

1. $0 \leq D_s(Q, T_i) \leq 1$
2. $D_s(Q, T_i) = 0 \Leftrightarrow q_j \in T_i, \forall j = 1, \ldots, m$
3. $D_s(Q, T_i) \leq d_q(Q, x) + d_s(x, T_i)$, where $d_q(Q, x) = \sum_{j=1}^{m} w_j \cdot d(q_j, x)$

From Proposition 1 we have that: $0 \leq d_s(q_j, T_i) \leq 1, \forall j = 1, \ldots, m$. Since $w_j > 0$, $\forall j = 1, \ldots, m$, we have that: $0 \leq w_j \cdot d_s(q_j, T_i) \leq w_j, \forall j = 1, \ldots, m$. By summing the above $m$ inequalities we derive:

$$0 \leq \sum_{j=1}^{m} w_j \cdot d_s(q_j, T_i) \leq \sum_{j=1}^{m} w_j \Leftrightarrow 0 \leq D_s(Q, T_i) \leq 1$$

Moreover, $D_s(Q, T_i) = 0 \Leftrightarrow \sum_{j=1}^{m} w_j \cdot d_s(q_j, T_i) = 0$, and since $d_s(q_j, T_i) \geq 0$ and $w_j > 0, \forall j = 1, \ldots, m$, the sum will be zero in case that all terms become zero, i.e. $d_s(q_j, T_i) = 0, \forall j = 1, \ldots, m \Leftrightarrow q_j \in T_i, \forall j = 1, \ldots, m$ (Proposition 1).

Finally, if $x$ is a random graph node, according to Proposition 1, we have: $d_s(q_j, T_i) \leq d(q_j, x) + d_s(x, T_i), \forall j = 1, \ldots, m$. Thus: $w_j \cdot d_s(q_j, T_i) \leq w_j \cdot d(q_j, x) + w_j \cdot d_s(x, T_i), \forall j = 1, \ldots, m$. By summing these $m$ inequalities, we get:

$$\sum_{j=1}^{m} w_j \cdot d_s(q_j, T_i) \leq \sum_{j=1}^{m} w_j \cdot d(q_j, x) + d_s(x, T_i) \cdot \sum_{j=1}^{m} w_j$$

$$\Leftrightarrow D_s(Q, T_i) \leq d_q(Q, x) + d_s(x, T_i)$$

□

An advantage of this methodology is that the proposed similarity measures express the similarity between a trajectory $T_i \in T$ and the selected spatial locations in $Q$. Therefore, the proposed measures are functions in the $|Q| \times |T|$ space, instead of the $|T| \times |T|$ space, by significantly speeding up computations. Moreover, the computation of all-to-all geodesic path distances is avoided, by limiting the calculation of pairwise node distances from the small set of spatial positions $Q$ to the nodes of $T$. This is in contrast to the majority of previous methods for trajectory similarity search, which require a computationally intensive preprocessing step with all-to-all geodesic path distance calculations.

A significant property of the proposed method is that it is not required that all times points are stored. The reason is that time restrictions set by the recorded timestamps between the location visits and the resulting delay is what defines the temporal restrictions in an absolute manner.

To calculate the temporal similarity we obtain the nearest nodes $vmin_j$ for $j = 1, \dots, m$ of the trajectory to the spatial locations, as described above. Then, we calculate the inter-arrival times on each of these nodes. This can be calculated instantly, by summing the arrival times of these nodes. More specifically, if we set $t_{vmin_j}$ for $j = 1, \dots, m$ as the time points, which correspond to each closest node, we calculate the corresponding inter-arrival times $dt_2, dt_3, \dots, dt_m$ in the above fashion. We observe three distinct cases:

1. $dt_j = qt_j$: The actual temporal distance of the location from the trajectory is equal to the time tolerance based on the recorded timestamps. The temporal distance is equal to 0.
2. $dt_j > qt_j$: The temporal distance is greater than the time tolerance. Thus, more time is needed to pass through the trajectory, which means that the temporal difference must be taken into consideration and is equal to $|qt_j - dt_j|$.
3. $dt_j < qt_j$: The temporal distance is less than the time tolerance. In this case, the temporal distance is not taken into consideration, since less time is needed to traverse the trajectory. This case is treated like the first one, i.e. the temporal distance is 0.

Based on the above, the temporal distance metric is:

$$D_t(Q, T_i) = \frac{1}{m-1} \sum_{j=2}^{m} \frac{|qt_j - dt_j|}{\max_{2<j<m}\{qt_j, dt_j\}} \tag{5}$$

**Proposition 3** *$D_t(\cdot)$ is a generalized metric function that satisfies the generalized triangular inequality in the range* [0, 1].

*Proof* For $j = 2, \dots, m$, by considering the values $qt_j$ and $dt_j$ as $m - 1$ couples of real values, the proof is the same as presented in [34]. □

The two previous similarity metrics are then combined into a spatio-temporal metric $sim(.)$ as follows:

$$sim(Q, T) = 1 - dist(Q, t) \tag{6}$$

where:

$$dist(Q, T) = a * D_s(Q, T) + (1 - a) * D_t(Q, T) \tag{7}$$

Parameter $a \in [0, 1]$ expresses the preference to one of these two metrics, depending on how close its value approaches 0 or 1. Thus, an application can tune which of the two, or any combination of the two, should be applied. If $a = 0$, then an absolute preference for the temporal distance is expressed. Oppositely, $a = 1$ means that only the spatial distance will be included in the spatio-temporal similarity measure calculation.

**Algorithm 2** Progressive trajectory retrieval algorithm.

```
1: $L = 0, Lcurr = 0, top = 1, H = \emptyset, HQ_j = \emptyset (j = 1, ..., m), B[i] = false (i = 1, ..., |T|)$
2: initialize $vQ_j = q_j, \forall j = 1, ..., m$
3: initialize $distQ_j[vQ_j] = 0, distQ_j[\text{other nodes}] = \infty, \forall j = 1, ..., m$
4: $HQ_j.insert(vQ_j, distQ_j[vQ_j]), \forall j = 1, ..., m$
5: while $HQ_j.size > 0, \forall j = 1, \ldots, m$ do
6:     for $j = 1$ to $m$ do
7:         $vQ_j = HQ_j.extractMin(), vmin_j = vQ_j$
8:         for all neighbors $uQ_j$ of $vQ_j$ in the adjacency list do
9:             if $distQ_j[uQ_j] > distQ_j[vQ_j] + w(vQ_j, uQ_j)$ then
10:                $distQ_j[uQ_j] = distQ_j[vQ_j] + w(vQ_j, uQ_j)$
11:                if $uQ_j \in HQ_j$ then
12:                    $HQ_j.decreaseKey(uQ_j, distQ_j[uQ_j]$
13:                end if
14:            end if
15:            if $uQ_j \notin HQ_j$ then
16:                $HQ_j.insert(uQ_j, distQ_j[uQ_j]$
17:            end if
18:            for all trajectory IDs $T_h.ID$ in cluster $C_{(vQ_j,uQ_j)}$ from edge $(vQ_j, uQ_j)$ do
19:                if $B[T_h.ID] = false$ then
20:                    retrieve data of trajectory $T_h$ from raw file through $hash(T_h.ID)$
21:                    compute the spatio-temporal distance $dst = dist(Q, T_h)$
22:                    $B[T_h.ID] = true$
23:                    $H.insert(T_h.ID, dst)$
24:                end if
25:            end for
26:        end for
27:    end for
28:    $Lcurr = \frac{a}{m} \sum_{j=1}^{m} d(q_j, vmin_j)$
29:    if $Lcurr > L$ then
30:        $L = Lcurr$
31:    end if
32:    for $i = top$ to $k$ do
33:        if $H[i].dst < L$ then
34:            $top++$, return trajectory $H[i].ID$
35:        end if
36:    end for
37:    if $L > H[k].dst$ or $B.count = |T|$ then
38:        Stop execution
39:    end if
40: end while
```

Algorithm 2 presents the progressive trajectory retrieval process from the spatial locations. The main strategy is the following: from each location, perform an incremental Dijkstra expansion step following a round-robin strategy, collect the trajectory IDs that are included in the trajectory clusters of the visited edges, compute the spatiotemporal similarities based on the proposed measures and progressively return the top-$k$ retrieved trajectories when an updated threshold value is satisfied. The threshold and top-$k$ process is similar with Fagin's Threshold Algorithm [13].

When the algorithm begins, variables and structures are initialized (lines 1–4). Variable $L$ keeps the threshold value. The ordered structure $H$ keeps the retrieved trajectory ID's ordered by their calculated spatiotemporal distance. Each location $q_j$ uses a Fibonacci Heap $HQ_j$, in which the corresponding shortest-path distances from the Dijkstra expansion are updated. To avoid recalculations of spatiotemporal distances in any step of the algorithm, a bit-set $B$ with $|T|$ bits in memory is used where the corresponding bit of each calculated trajectory distance is enabled on-the-fly. Therefore, during the query processing, the distances are calculated only once for each trajectory. The five main steps of the proposed algorithm are the following:

**(S1):** From each location $q_j$, in a round-robin manner (initially $vQ_j = q_j$), each neighbor node $uQ_j$ of $vQ_j$ is retrieved in the Dijkstra expansion step (lines 5–17). The heaps $HQ_j$ are updated with the relevant shortest-path distances from the Dijkstra expansion. The candidate trajectories $T_h$ are collected from the corresponding edge clusters $C_{(vQ_j, uQ_j)}$ of the extended adjacency list index (line 18).

**(S2):** The spatiotemporal distances $dist$ between the collected candidate trajectories $T_h$ and the location set $Q$ are calculated (7). In bit-set $B$ the corresponding bits of each calculated trajectory distance are enabled (lines 19–24). The currently calculated trajectory distances and their corresponding trajectory Ids are preserved and updated in $H$ (ordered by $dist$) on-the-fly (line 23).

**(S3):** The threshold $L$ is updated according to the aggregated network distances between the locations and the set of $vmin_j$ nodes: $L = \frac{a}{m} \sum_{j=1}^{m} d(q_j, vmin_j)$, where $vmin_j$ is the closest node to location $q_j$ in the current Dijkstra expansion level, i.e. $vmin_j$ has the shortest path distance to $q_j$ among all the detected nodes in the current round from $q_j$. The threshold $L$ is a lower bound of the final distance function $dist$ and it is used for generating the results. In each round, $L$ is increased, (when the expansion level is changed), by comparing the current $Lcurr$ value with the previously calculated one. In particular, if the currently computed $Lcurr$ value is greater than the previous $L$ value of the last round, then the $Lcurr$ value of the current round is updated accordingly (lines 28–31). Since the temporal distances $D_t$ are aggregated with the spatial distances $D_s$ in the final distance function $dist(\cdot)$, $L$ is a lower bound for both spatial and spatiotemporal distances. Moreover, in case that $w_j$ weights are used (4), then threshold $L$ is calculated as: $L = a \cdot \sum_{j=1}^{m} w_j \cdot d(q_j, vmin_j)$ (alternative line 28).

**(S4):** After the end of each round, the trajectories in the current top-$k$ list in $H$ are examined based on condition that they have a distance $dist$ lower than $L$. If the condition is satisfied for a subset of trajectories in $H$, then these trajectories are instantly added to the top results list (lines 32–36). The trajectory extraction proceeds progressively until $L$ reaches a value greater than the distance of the $k$-th element in $H$ or in the extreme case that the spatiotemporal distances of all trajectories in $T$ have been calculated (stopping condition, line 37).

**(S5):** If not all top-$k$ results have been retrieved, the algorithm proceeds to the next expansion round, where the algorithm repeats the loop in lines 5–40.

**Correctness:** As the threshold $L$ is a lower bound of the distance function $dist$, all trajectories that have not been discovered yet will have spatiotemporal distances greater than or equal to $L$. This means that the trajectories that have been stored into $H$ will definitely have smaller distances to any not discovered yet trajectory in all next expansion levels. Also, when a trajectory is inserted into $H$, its calculated spatiotemporal distance is a final distance and will not be modified (bitset $B$ ensures that will not even be recalculated). Therefore, as $L$ is increasing and there are some trajectories in the top positions of $H$ that have distances smaller to $L$, they can safely returned to the user.

The time complexity of Algorithm 2 is: $O(m * (|V| * \log |V| + |E|) + |RE|)$, where the part $O(m * (|V| * \log |V| + |E|))$ corresponds to the Dijkstra expansion. On the other hand, the term $O(|RE|)$ represents the number of trajectory edges the algorithm will take into account; it is at maximum $|RE|$, since the control bitmap will be storing the IDs of the trajectories with distances already calculated.

### 4.3.3 Trajectory retrieval for the illustrative small-scale example

The diameter of the graph in Figure 5 is $D_G = 27$, which is the geodesic distance between the most distant nodes $v_1$ and $v_{14}$. The closest nodes of trajectory $T_1$ from locations $q_1, q_2, q_3$ are nodes $v_3, v_{12}, v_{13}$, with distances 4,2,2, respectively. Then, the spatial distance between the set $Q$ of locations and the nodes of trajectory $T_1$ are calculated as: $D_s(Q, T_1) = \frac{1}{3} \cdot \frac{4+2+2}{27} \approx 0.099$ (considering equal weights $w_j = \frac{1}{3}$). The closest nodes of trajectory $T_2$ from locations $q_1, q_2, q_3$ are nodes $v_5, v_8, v_{11}$, with distances 0,5,0, respectively. Then, $D_s(Q, T_2)$ is: $D_s(Q, T_2) = \frac{1}{3} \cdot \frac{0+5+0}{27} \approx 0.062$. The closest nodes of trajectory $T_3$ from locations $q_1, q_2, q_3$ are nodes $v_5, v_9, v_{11}$, which are the nodes that the trajectory passes through all the locations, resulting thus in $D_s(Q, T_3) = 0$. Therefore, if $a = 1$, i.e. only the spatial similarity contributes to the final score of $sim(\cdot)$, the top-3 similarity list is $[T_3, T_2, T_1]$.

If the time tolerance is 3 time units for the transition from $q_1$ to $q_2$ and 3 time units for the transition from $q_2$ to $q_3$, i.e. $qt_2 = qt_3 = 3$, then the corresponding inter-arrival times for $T_1$ are $dt_2 = 5 > 3$ (for the transition from $v_3$ to $v_{12}$), and $dt_3 = 4 > 3$ (for the transition from $v_{12}$ to $v_{13}$). Therefore, $D_t(Q, T_1)$ is: $D_t(Q, T_1) = \frac{1}{2} \cdot \left(\frac{|3-5|}{5} + \frac{|3-4|}{4}\right) = 0.325$. The corresponding inter-arrival times for $T_2$ are $dt_2 = 2 < 3$ (for the transition from $v_5$ to $v_8$), and $dt_3 = 2 < 3$ (for the transition from $v_8$ to $v_{11}$). Therefore, we have: $D_t(Q, T_2) = 0$, since $dt$ values are set equal to $qt$. The corresponding inter-arrival times for $T_3$ are $dt_2 = 5 > 3$ (for the transition from $v_5$ to $v_9$), and $dt_3 = 5 > 3$ (for the transition from $v_9$ to $v_{11}$). Therefore, $D_t(Q, T_3)$ is: $D_t(Q, T_3) = \frac{1}{2} \cdot \left(\frac{|3-5|}{5} + \frac{|3-5|}{5}\right) = 0.4$.

If $a = 0.5$, the final spatio-temporal distances of the trajectories are equal to: $dist(Q, T_1) \approx 0.212$, $dist(Q, T_2) \approx 0.031$, $dist(Q, T_3) = 0.2$. By considering both temporal and spatial domains, the top-3 similarity list becomes $[T_2, T_3, T_1]$. In contrast to trajectory $T_3$ (case $a = 1$), by considering both spatial and temporal domains (case $a = 0.5$), $T_2$ is the top trajectory result which does not pass from all the locations.

## 5 Unified framework and extensions for the studied methods

Here we provide additional information about the unified framework implemented for the three studied methods, plus more details about the extensions. The studied methods and the respective algorithms were implemented so that memory manipulation and pointer creation to objects and values is allowed, instead of copying whole objects across functions and methods. This way, the involved classes and structures can communicate with each other and have access to the needed objects.

We model the graphs with adjacency lists because the tested networks are sparse plus they are rather stable without any changes to their initial topology. Thus, graph data are stored with size analogous to that of the network. Especially for CeAL, each edge has an additional property, corresponding to the cluster containing the IDs of the trajectories passing through this edge.

We used the Dijkstra algorithm [12] for pairwise shortest path calculations. We provided the initial node to the Dijkstra algorithm, so that it begins its expansion steps, as well as an array for the distances assigned to each edge. This results in an efficient derivation of the shortest paths, even in extremely large networks. The particular implementation of the Dijkstra algorithm uses the concept of a virtual *visitor*. We provided a visitor as parameter to the function call, which evaluates the potential options from the nodes adjacent to the one it is on, and selects one according to the specific algorithmic restrictions.

Most implementations use the predefined Dijkstra visitor, which will only perform the default set of actions on each step. The main advantage is that we can replace this predefined visitor with a visitor of our own preference to perform modified tasks on each step. Therefore, especially for CeAL that uses the Dijkstra algorithm to perform a set of actions on each expansion step, we implement our own visitor class and override its behavior when it finishes handling a node. Thus, the Dijkstra expansion step will be overridden and the trajectories will be discovered as per the function of the algorithm described above. Before any expansion from each location, we call the Dijkstra algorithm by passing the predefined visitor as a parameter, and then we pass the resulting distance array to our custom visitor. This allows to perform the distance calculations based on either the Euclidean distance, or the actual distance obeying the network constraints as adopted here.

The trajectories collected through query processing are stored in a min-heap. At the end of the process, the min-heap contains all the discovered trajectories in ascending order. We then extract the top-$k$ trajectories, where $k$ is user-defined, either in an expanded (all nodes that comprise a trajectory), or in a compressed form (number of nodes in trajectory), and the total spatio-temporal cost.

Regarding the R-tree-based methods, we implemented the bottom R-trees aiming at maximizing the control and the ability for modifications. Each R-tree node consists of a 1-d array containing the tree elements. Thus, the nodes are separated from the elements to be stored; this provides the flexibility to store whatever is necessary. Notably, in our implementation the edges follow a bidirectional rationale, i.e. the flag for the edge direction is ignored.

Our initial approach was to insert the *TopTreeElement* elements into each node, for both internal and leaf nodes. The first tests showed that this approach was not very efficient with respect to creating and inserting elements in each node. For this reason, the *TopTreeElement* elements were removed from the internal nodes, i.e. in our current R-tree implementation, elements of this type can only be found in the leaves, whereas internal nodes store an 1-d vector, which can store elements of any type. Thus, we can store either *TopTreeElement* elements, or pointers at R-tree lower levels. We use R-trees to store the edges. This provides a robust implementation with respect to time and storage efficiency. These extensions are important for enhancing the performance of the two studied R-tree based methods.

The trajectory generator by Brinkhoff [4] was extended to further enhance its performance for very large networks. Especially for the generation process of the trajectories, a class has been implemented to model time with discrete clock ticks, which denote a new movement cycle for the visitors, where each visitor moves only as far as its speed allows.

Each time instance is numbered, which allows tying visits to nodes and to edges to a time instance or a time interval. Thus, it is possible for an object, depending on its speed, to pass more than one clock ticks on the same edge; however, visiting a node is considered always instantaneous.

The creation of a visitor is based on a probability, which can be adjusted to model bustling or deserted networks. This probability is examined on each time instant; in addition to the visitors already moving in the network, new visitors may also appear. Each visitor *knows* the specific trajectory to follow in the network, which is indexed and stored in each particular studied method. Additionally, each visitor is assigned a specific speed, which determines the distance traversed on one clock tick. The way used visitors can be modified so that moving objects can be studied on graphs. In our case, each visitor decides on the trajectory to follow, and it traces this trajectory with increasing time. The trajectory is stored as a sequence of nodes, and thus a visitor is used only to trace trajectories on the network.

## 6 Experimental evaluation

We performed a series of exhaustive experiments on the studied methods. The datasets retrieved from the 9th DIMACS Challenge webpage [30] (last update in 2010) represent various specific portions of the US road network (see Table 3). Even though we appreciate the realistic network topology, which helps in providing the real distances between nodes instead of calculating them, either during the graph construction in a preprocessing step, or on the fly during the algorithm execution, our interest primarily lies in the network sizes, which cover a wide spectrum. By using these datasets, we trust that our conclusions are realistic.

All methods and the unified framework were implemented in C++. The Boost Graph library [5] was also used for several primitive structure types and graph algorithms. The experiments were performed on a personal computer with an Intel Core i7-6700K quad-core processor clocked at 4.00 GHz with 8 MB Cache, 16 GB (8GBx2) DDR4 main RAM memory clocked at 2133 MHz, and an SSD drive, with a read speed of 550 MB/s, a write speed of 520 MB/s, and a capacity of 120 GB. To avoid any throttling on the processor or any other system part, the computer remained plugged in the power outlet throughout the whole experimentation.

**Table 3** Datasets used for experimental evaluation

| Dataset | Description | Nodes | Edges |
|---|---|---|---|
| NY | New York City | 264346 | 733846 |
| BAY | San Francisco Bay Area | 321270 | 800172 |
| COL | Colorado | 435666 | 1057066 |
| FLA | Florida | 1070376 | 2712798 |
| NW | Northwest USA | 1207945 | 2840208 |
| NE | Northeast USA | 1524453 | 3897636 |
| CAL | California & Nevada | 1890815 | 4657742 |
| LKS | Great Lakes | 2758119 | 6885658 |
| E | Eastern USA | 3598623 | 8778114 |
| W | Western USA | 6262104 | 15248146 |
| CTR | Central USA | 14081816 | 34292496 |

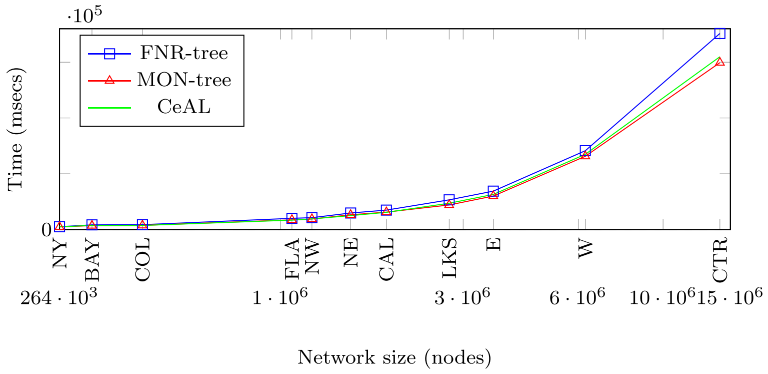


**Figure 8** Time for constructing the trajectory indexes - 1K trajectories

## 6.1 Preprocessing - indexing and storing

Here we calculate the required space and time to construct the corresponding indexes of each method. In each case, we construct the network and store it in the relevant index. Afterwards, we insert a variable number of trajectories and observe the time and storage required. The number of trajectories varies from 1K to 10K, 100K and 1M (Figs. 8, 9, 10 and 11). The trajectories are created by our enhanced generator as described in Section 5.

We observe a more or less similar behavior, when bulk trajectory insertions are made in the networks. The time differences are small in all three methods; however, MON-tree and CeAL have a distinct advantage when storing an empty network over FNR-tree, as seen in Table 4.

With respect to the storage space needed while constructing the network, for brevity, we show results only for the NY network since our experiments indicate that the same behavior is observed across all datasets. Table 5 shows the increase of memory space required for a variable number of trajectories. We observe that CeAL displays the best behavior in comparison to the other methods. When the number of trajectories is small (e.g. 1000) there is

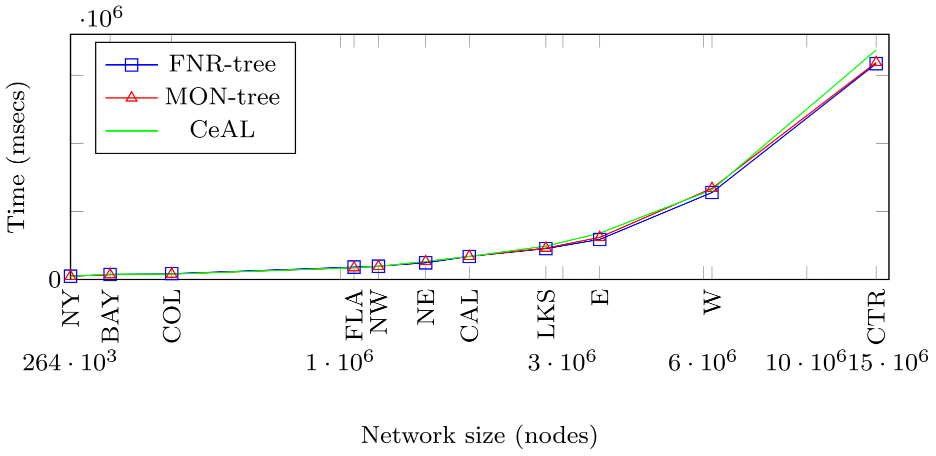


**Figure 9** Time for constructing the trajectory indexes - 10K trajectories

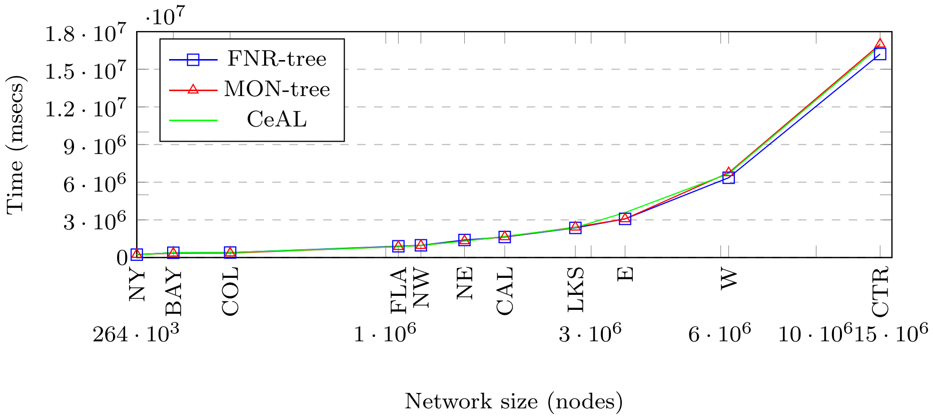


**Figure 10** Time for constructing the trajectory indexes - 100K trajectories

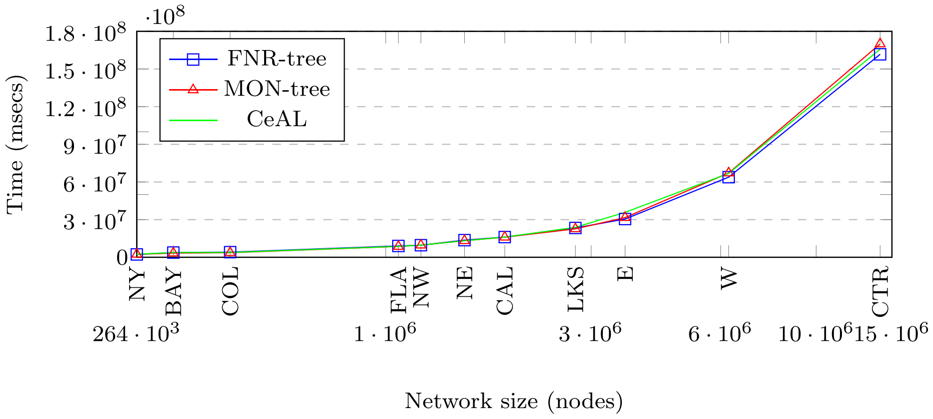


**Figure 11** Time for constructing the trajectory indexes - 1M trajectories

**Table 4** Time needed to construct networks for each index (secs)

| Dataset | FNR-tree | MON-tree | CeAL |
|---|---|---|---|
| NY | 303.68 | 22.32 | 24.21 |
| BAY | 330.99 | 26.65 | 29.91 |
| COL | 436.70 | 32.52 | 35.83 |
| FLA | 1158.06 | 84.47 | 97.94 |
| NW | 1246.27 | 89.66 | 100.27 |
| NE | 1536.19 | 108.95 | 118.08 |
| CAL | 2018.57 | 138.20 | 151.80 |
| LKS | 3023.64 | 203.88 | 226.88 |
| E | 3832.78 | 275.69 | 302.58 |
| W | 6847.95 | 446.74 | 499.31 |
| CTR | 15533.41 | 1041.21 | 1225.30 |

**Table 5** Network storage space for each index (MB)

| Trajectories | FNR-tree | MON-tree | CeAL |
|---|---|---|---|
| 1000 | 353.10 | 339.50 | 355.50 |
| 10000 | 467.80 | 482.80 | 376.90 |
| 100000 | 1651.80 | 1916.80 | 617.90 |
| 1000000 | 13162.80 | 17045.80 | 2646.90 |

not much difference between the examined methods; however, as the number of trajectories increases, the performance gap becomes more obvious.

Figure 12 shows the comparison of the time needed to insert a trajectory in each index. It can be seen that MON-tree and CeAL outperform FNR-tree in all cases. In networks larger than the Great Lakes network, CeAL increases linearly in time, whereas MON-tree retains its performance. This is due to the index construction. FNR-tree stores the entire network; thus, inserting a new trajectory requires a search, which becomes more expensive as the network increases in size. Similarly, CeAL's adjacency list grows in size as the network nodes increase, leading to larger search times for storing the new trajectory in the correct network edges. On the other hand, MON-tree does not store any part of the network until it becomes relevant (by being part of a new trajectory, which means that it is less sensitive to network size increases). Similar results are retrieved for a trajectory deletion.

Next, we examine the time to retrieve a trajectory from each index. In particular, first we compare the results between FNR-tree and MON-tree, and then between MON-tree and CeAL. Figure 13 shows that the performance of FNR-tree is inferior of that of MON-tree in all cases. In particular, its performance degrades seriously as the network size increases. Therefore, the only meaningful comparison is between MON-tree and CeAL. Figure 14 shows that CeAL outperforms MON-tree in retrieving the trajectory edges. Retrieving a trajectory from a MON-tree requires a traversal of the generated R-tree, which is sensitive to the network size, while doing the same in a CeAL index requires the traversal of a single adjacency list path, which doesn't require any comparisons and path decisions.

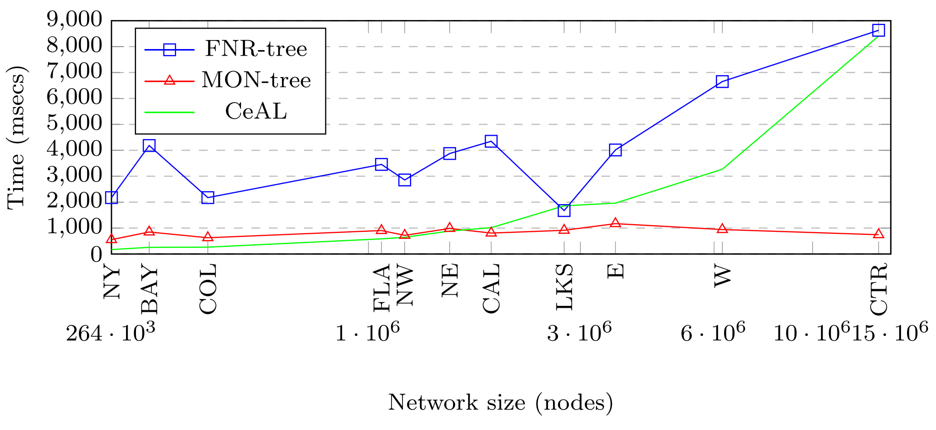


**Figure 12** Comparison between FNR-tree, MON-tree and CeAL - trajectory insertion

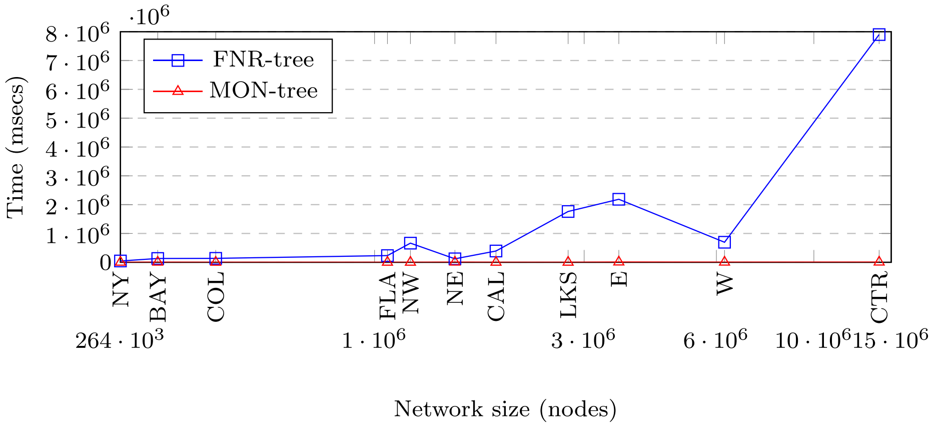


**Figure 13** Comparison between FNR-tree and MON-tree - trajectory discovery

## 6.2 Trajectory similarity query processing

Here, we test the performance of the methods under examination during trajectory similarity query processing. We measure the total time performance (CPU time and I/O cost) as well as the total number of distance computations / shortest path calculations for searching and retrieving trajectories stored within each index.

### 6.2.1 Results for variation of $|T|$

We check how the examined methods behave with increasing number of stored trajectories since real-life applications deal with a large number of trajectories, and dynamic data can lead to increased load. Figs. 15, 16, 17 and 18 show how the algorithms behave across the different networks.

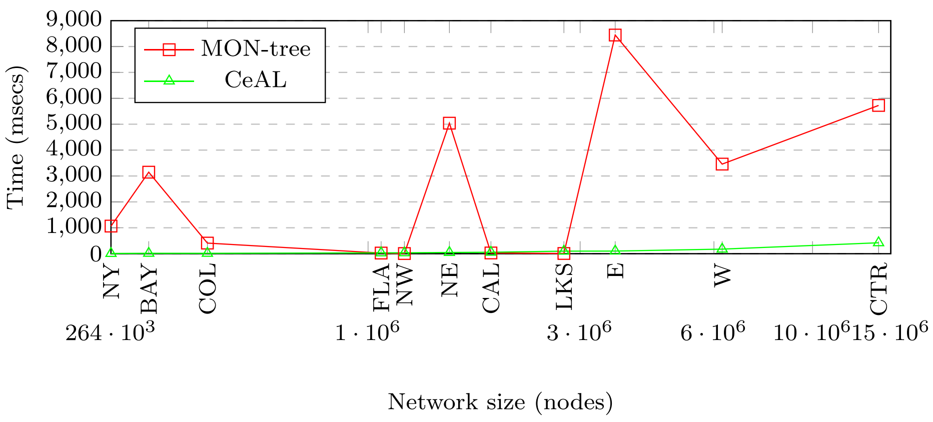


**Figure 14** Comparison between MON-tree and CeAL - trajectory discovery

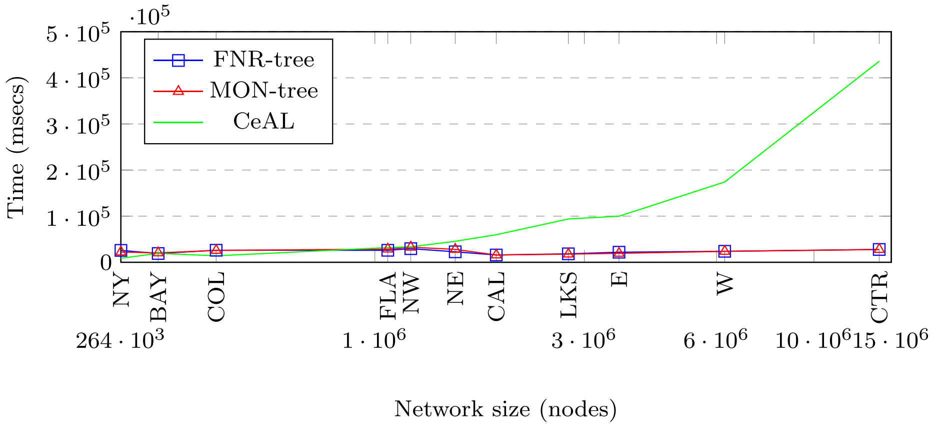


**Figure 15** FNR-tree, MON-tree and CeAL - 1000 trajectories

As can be seen from these figures, small trajectory numbers favor the two spatial methods, which exhibit a more or less stable behavior. On the other hand, CeAL scales linearly with increasing network size, a behavior that is maintained across all cases of trajectory numbers. A small trajectory number and a smaller network still favors CeAL, since it performs as well as, and in some cases better, than the spatial methods.

CeAL outperforms the other methods when the number of stored trajectories is in the order of millions. Figure 18 shows that CeAL outperforms the two spatial methods in small, medium and large networks. Assuming that a real-life application will have millions of stored trajectories to provide accurate information to the users, we can conclude that despite the fact that the initial results don't favor CeAL across the whole dataset, it performs better in the crucial tests.

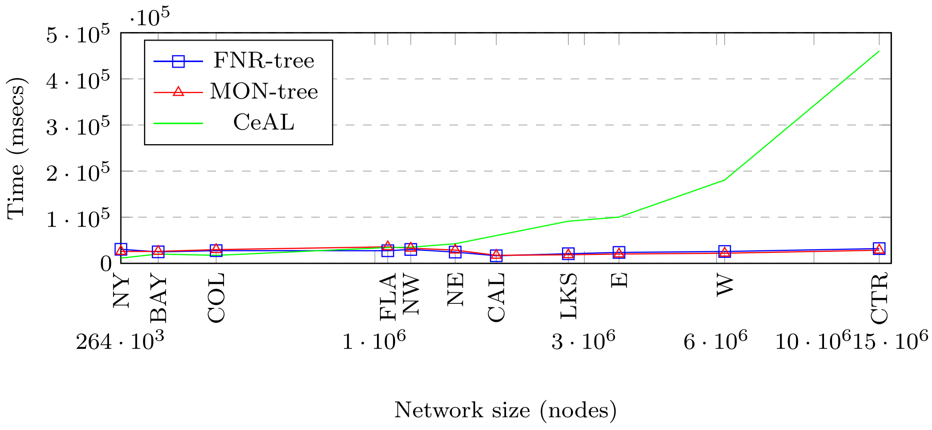


**Figure 16** FNR-tree, MON-tree and CeAL - 10000 trajectories

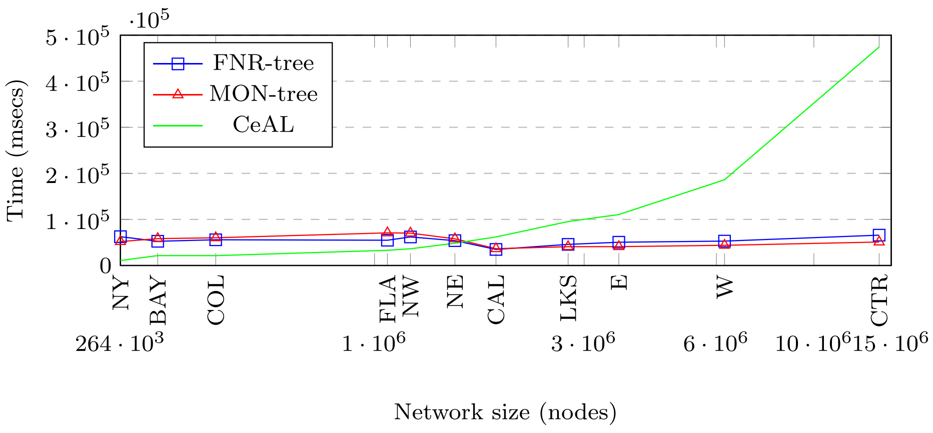


**Figure 17** FNR-tree, MON-tree and CeAL - 100000 trajectories

### 6.2.2 Results for variation of *k*

Next, we examine how the methods are affected with varying $k$. To stress the methods we use networks with 1000000 trajectories, for reasons mentioned in previous. The results can be seen in Figure 19. It should be noted that the FNR-tree and MON-tree indexes do not support a top-$k$ query, but rather work by providing a desired spatiotemporal box and retrieving all trajectories in it. Nevertheless, we include the relevant measurements for both methods in the figure, so that a comparison can be made about the relative efficiency between the algorithms.

### 6.2.3 Results for variation of *q*

Next, we examine the effect of increasing the query locations $q = 5, 10, 20, 40, 80$ when $|T| = 100$ K and $k = 10$. As seen in Figure 20, this has the most significant impact on

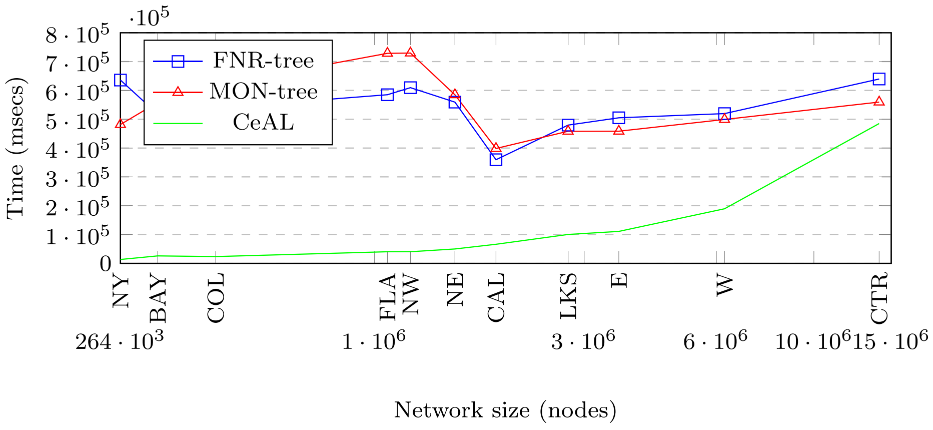


**Figure 18** FNR-tree, MON-tree and CeAL - 1000000 trajectories

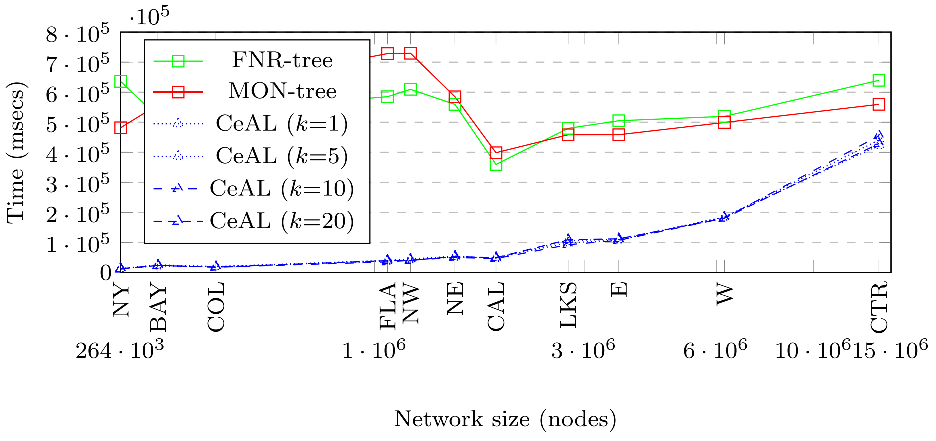


**Figure 19** Comparison between FNR-tree, MON-tree and CeAL - trajectory discovery, number of top-$k$ trajectories requested

CeAL's performance, due to the increased number of shortest path calculations that need to be performed. Again, FNR-tree and MON-tree relevant measurements are included for comparison purposes.

### 6.2.4 Results for progressiveness of top-$K$ result discovery

To show CeAL's progressiveness in fetching the trajectory results, we logged the time when each trajectory was retrieved during the algorithm's execution. We ran this experiment with 100K trajectories stored in the data structure, with 20 query points of interest, requesting 10 trajectories. In further experiments, the algorithm's behavior remains consistent with what is shown below. Table 6 shows how trajectories are obtained in a progressive manner when

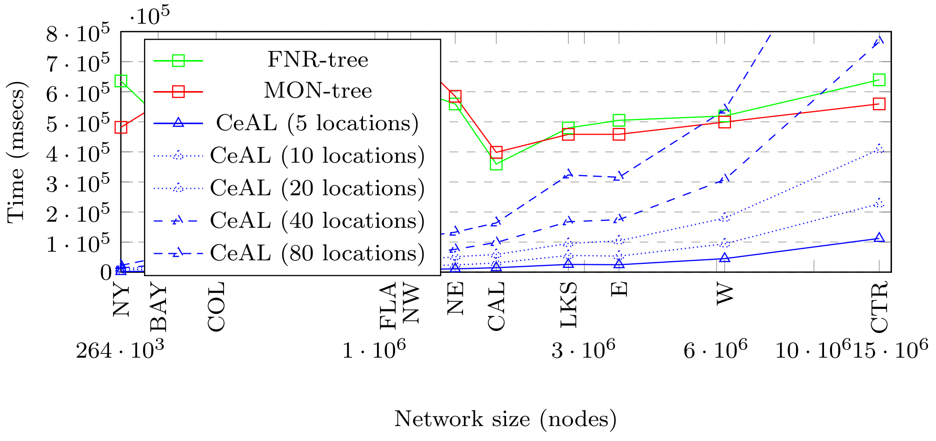


**Figure 20** Comparison between FNR-tree, MON-tree and CeAL - trajectory discovery, number of query locations

**Table 6** Progressive discovery of top-$k$ results (msecs)

| Datasets | $k = 1$ | $k = 2$ | $k = 3$ | $k = 4$ | $k = 5$ | $k = 6$ | $k = 7$ | $k = 8$ | $k = 9$ | $k = 10$ |
|---|---|---|---|---|---|---|---|---|---|---|
| NY | 780 | 1630 | 2730 | 3580 | 4310 | 5070 | 5810 | 6550 | 7310 | 10830 |
| BAY | 3440 | 5470 | 6360 | 7900 | 10420 | 12727 | 14811 | 16138 | 18211 | 21470 |
| COL | 1860 | 3150 | 4370 | 5780 | 7300 | 8630 | 10886 | 10981 | 12475 | 21390 |
| FLA | 2950 | 5920 | 8880 | 12120 | 17718 | 19434 | 21329 | 26069 | 26936 | 32990 |
| NW | 3500 | 6880 | 10525 | 13665 | 16624 | 19570 | 22654 | 28340 | 30458 | 36280 |
| NE | 4210 | 11569 | 16554 | 20852 | 24701 | 31242 | 31341 | 40189 | 44571 | 48390 |
| CAL | 6730 | 13758 | 18114 | 23886 | 27316 | 33902 | 42918 | 43164 | 53660 | 61930 |
| LKS | 7550 | 18231 | 31361 | 36461 | 50865 | 58873 | 62047 | 84586 | 92230 | 95410 |
| E | 9696 | 17882 | 27360 | 39821 | 51731 | 61662 | 70105 | 85332 | 83030 | 110620 |
| W | 12844 | 31548 | 56940 | 66848 | 82191 | 95167 | 101184 | 127592 | 156489 | 186080 |
| CTR | 13485 | 54912 | 97495 | 149182 | 196548 | 227682 | 299883 | 361241 | 401088 | 474390 |

performing a top-$k$ query in CeAL. For each dataset, the time when a particular trajectory was obtained is shown. The above results show that CeAL retrieves its results progressively, thus being able to provide results in a more efficient manner.

### 6.2.5 Results for I/O activity

Lastly, we present results on the average number of page accesses (I/O activity) needed by each method for each network to retrieve a trajectory, as seen in Table 7. Note that for this comparison, we use the results of MON-tree for 1 million stored trajectories. Also, for completeness in Table 8 we present the average number of disk accesses of MON-tree, which depends on the number of inserted trajectories into the structure.

As can be seen from the above results, CeAL's adjacency list-based structure offers a distinct advantage as far as disk accesses are concerned, as it is not sensitive to the increase of the network size. The spatial methods demonstrate different behaviors. FNR-tree constructs

**Table 7** Average number of disk accesses for all methods

| Datasets | FNR-tree | MON-tree (T = 1M) | CeAL |
|---|---|---|---|
| NY | 5.00 | 7.03 | 2.00 |
| BAY | 5.00 | 7.78 | 2.00 |
| COL | 6.00 | 7.29 | 2.00 |
| FLA | 6.00 | 7.65 | 2.00 |
| NW | 6.00 | 7.50 | 2.00 |
| NE | 6.00 | 7.99 | 2.00 |
| CAL | 6.00 | 7.34 | 2.00 |
| LKS | 6.00 | 7.38 | 2.00 |
| E | 6.00 | 7.08 | 2.00 |
| W | 6.00 | 7.00 | 2.00 |
| CTR | 7.00 | 7.07 | 2.00 |

**Table 8** Average number of disk accesses for a MON-tree with varying trajectory number

| Datasets | 1K | 10K | 100K | 1M |
|---|---|---|---|---|
| NY | 4.45 | 5.84 | 6.43 | 7.03 |
| BAY | 4.28 | 5.04 | 6.78 | 7.78 |
| COL | 4.93 | 5.90 | 6.25 | 7.29 |
| FLA | 4.29 | 5.40 | 6.99 | 7.65 |
| NW | 4.21 | 5.32 | 6.52 | 7.50 |
| NE | 4.92 | 5.19 | 6.00 | 7.99 |
| CAL | 4.65 | 5.37 | 6.14 | 7.34 |
| LKS | 4.32 | 5.77 | 6.51 | 7.38 |
| E | 4.78 | 5.83 | 6.28 | 7.08 |
| W | 4.46 | 5.06 | 6.85 | 7.00 |
| CTR | 4.01 | 5.41 | 6.43 | 7.07 |

the entire network from the beginning; thus, the network size determines how deep the resulting R-tree's leaf nodes are stored, and the total number of accesses needed to retrieve each level. On the other hand, MON-tree constructs only the relevant parts of the network, i.e. only the parts with existing trajectories. This means that the network size itself doesn't affect the number of disk accesses, but the number of trajectories does.

### 6.2.6 Results for distance calculations

CeAL uses Dijkstra's Shortest Path algorithm to find the closest trajectories to the points of interest $q$. Table 9 shows the number of performed calculations during the query trajectory processing across all of the network datasets. These results were obtained on a network with 100000 stored trajectories, a query of 20 locations, and requesting 10 top-$k$ trajectories. The R-tree-based indexes do not use any shortest-path algorithm, since they return trajectories contained within a provided bounding box.

**Table 9** Number of distance/shortest paths calculations for CeAL

| Datasets | Min | Max | Average |
|---|---|---|---|
| NY | 113675.00 | 323566.00 | 250235.12 |
| BAY | 81091.00 | 203337.00 | 195235.04 |
| COL | 218356.00 | 316043.00 | 238825.64 |
| FLA | 477433.00 | 1113304.00 | 781027.00 |
| NW | 302554.00 | 596789.00 | 526012.95 |
| NE | 823218.00 | 1480823.00 | 1137468.47 |
| CAL | 1135749.00 | 1449519.00 | 1135748.25 |
| LKS | 1908613.00 | 2225704.00 | 1908612.55 |
| E | 1573461.00 | 2286826.00 | 2238569.60 |
| W | 2028448.00 | 5810474.00 | 4014867.38 |
| CTR | 8650571.00 | 10667951.00 | 9424987.52 |

## 7 Conclusions

There is an ever increasing need for spatial and temporal data (and their spatio-temporal synthesis), which a user needs to retrieve efficiently, even in real time. We have examined three approaches towards the solution of the problem of trajectory storing, indexing and retrieval in large spatial networks, through the implementation of three different methods. Any application dealing with multi-dimensional spaces can be based on the methods using the R-trees family. On the other hand, our approach based on simpler structures and algorithms, removes the need for intermediate R-trees and leads to better time performance.

FNR-trees displays some issues when compared to its antecedents. The main disadvantage of this method is that it copies the whole network on its top level R-tree at the beginning of its execution, which increases the construction time and makes its reconstruction infeasible in case it is ever needed.

MON-trees improves upon the above, by not storing the entire network beforehand. Instead, edges are stored in the R-tree only when needed, i.e. only when storing a trajectory with edges that are not yet stored in the R-tree. If the edge has already been stored, then the temporal data insertion will proceed as usual. As seen in the experimental section, from the point of view of construction time, MON-trees should be favored. A direct consequence of this is that each time we insert a new edge or search for an existing one, this is always performed on a top level R-tree of smaller size than the FNR-tree, thereby generally decreasing the time needed for these operations.

Our CeAL method is an alternative approach which is characterized be the following advantages:

– Decreased storage space and construction time in comparison to the other two methods. In particular, this advantage is more significant when compared with FNR-trees.
– By using a generalized metric similarity function, the trajectories are ranked according to their relevance to the user query. This allows the user to retrieve the best trajectory according to his needs. This could be achieved by the other two methods by manipulating the query rectangle but this should come along with another set of difficult problems.
– It returns to the user the desired number of trajectories ($k$) in a progressive manner, whereas the other two methods return all the results they reach. This characteristic is important not only in a real-world setting but performance-wise as well.
– It prevails in retrieving complete trajectories, although the other two methods perform better at retrieving these parts of a trajectory which pass through a specific part of the graph.

The effectiveness of all three methods depends on the number of the stored trajectories. If this number is small, then the results will be poor. In particular, for the first two methods, the user query might contain a small number of trajectories, or even none at all. On the other hand, the CeAL method will always return a number of trajectories.

## 8 Future work

In this paper we focused in the problem of trajectory storing, indexing and retrieval in spatial networks aiming at delivering an efficient and effective solution. Future research could capitalize on the following ideas.

The drawbacks of FNR-trees have been faced by MON-trees. Thus, one could invest in improving further MON-trees instead of FNR-trees. MON-trees could be augmented with a ranking mechanism to deliver the trajectories according to the user input. Locations of interest could be integrated on this method as well, which could be translated to a representative trajectory. The trajectories discovered by the method would then be ranked based on spatial and temporal distances before returned to the user.

Another improvement of this method could be to provide the capability to define the desired number of the trajectories, ensuring that the user will receive only the most relevant results. This requires a metric function to rank the trajectories and return only the closest ones. As seen, the number of requested trajectories does not affect their discovery speed; e.g. a query requesting $k$ trajectories and one requesting $k + 1000$ trajectories will need approximately the same time.

MON-trees could be enhanced with a user-defined or derived threshold. For example, if the user requests $k$ trajectories, and several trajectories have a distance less than the threshold, then the execution would stop and the trajectories would be delivered to the user as good enough answers to query. This concept needs testing to come up with reasonable policies as to what should be the threshold set.

As explained before, all examined methods are based on the existence of a significant number of pre-stored trajectories to ensure that the results are relevant. If the number of trajectories is small, then the results, although algorithmically correct, will suffer in terms of quality and usefulness to the user. For this reason, an abundance of pre-existing trajectories is required, either real or generated. Using minimum spanning trees could help in separating the graph in segments, or neighborhoods, which can communicate with each other, and generating trajectories on each of these segments.

Another alternative could be based on the wide proliferation of social networks and on using spatio-temporal data gathered and aggregated from them. Proper anonymization techniques could be used to avoid invoking personal information issues. Unfortunately, there is lack of such datasets extracted from major social networks.

## References

1. Agrawal, R., Faloutsos, C., Swami, A.N.: Efficient similarity search in sequence databases. In: Proceedings of the 4th International Conference on Foundations of Data Organization & Algorithms (FODO), pp. 69–84. Chicago (1993)
2. Alt, H., Efrat, A., Rote, G., Wenk, C.: Matching planar maps. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 589–598. Baltimore (2003)
3. Brakatsoulas, S., Pfoser, D., Salas, R., Wenk, C.: On map-matching vehicle tracking data. In: Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), pp. 853–864. Trondheim (2005)
4. Brinkhoff, T.: Generating network-based moving objects. In: Proceedings of the 12th International Conference on Scientific & Statistical Database Management (SSDBM), pp. 253–255. Berlin (2000)
5. Boost Graph Library Index: http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/index.html
6. Cai, Y., Ng, R.: Indexing spatio-temporal trajectories with Chebyshev polynomials. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 599–610. Paris (2004)
7. Chan, K., Fu, A.W.: Efficient time series matching by wavelets. In: Proceedings of the 15th International Conference on Data Engineering (ICDE), pp. 126–133. Sydney (1999)
8. Chen, L., Ng, R.: On the marriage of Lp-norms and edit distance. In: Proceedings of the 13th International Conference on Very Large Data Bases (VLDB), pp. 792–803. Toronto (2004)
9. Chen, L., Ozsu, T., Oria, V.: Robust and fast similarity search for moving object trajectories. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 491–502. Baltimore (2005)

10. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB), pp. 426–435. Athens (1997)
11. de Almeida, V.T., Gueting, R.H.: Indexing the trajectories of moving objects in networks. GeoInformatica **9**(1), 33–60 (2005)
12. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
13. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS), pp. 102–113. Santa Barbara (2001)
14. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast subsequence matching in time-series databases. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 419–429. Minneapolis (1994)
15. Frentzos, E.: Indexing objects moving on fixed networks. In: Proceedings of the 8th International Symposium on Spatial & Temporal Databases (SSTD), pp. 289–305. Santorini (2003)
16. Frentzos, E., Gratsias, K., Pelekis, N., Theodoridis, Y.: Algorithms for nearest neighbor search on moving object trajectories. Geoinformatica **11**(2), 159–193 (2007)
17. Frentzos, E., Gratsias, K., Theodoridis, Y.: Index-based most similar trajectory search. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE), pp. 816–825. Istanbul (2007)
18. Greenfeld, J.S.: Matching GPS observations to locations on a digital map. In: Proceedings of the 81th Annual Meeting of the Transportation Research Board, Washington DC (2002)
19. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 47–57. Boston (1984)
20. Hadjieleftheriou, M., Kollios, G., Tsotras, V., Gunopulos, D.: Efficient indexing of spatiotemporal objects. In: Proceedings of the 8th International Conference on Extending Database Technology (EDBT), pp. 251–268. Prague (2002)
21. Keogh, E.: Exact indexing of dynamic time warping. In: Proceedings of the 28th International conference on Very Large Data Bases (VLDB), pp. 406–417. Hong Kong (2002)
22. Lin, B., Su, J.: Shapes based trajectory queries for moving objects. In: Proceedings of the 13th Annual ACM International Workshop on Geographic information systems (GIS), pp. 21–30. Bremen (2005)
23. Liu, K., Li, Y., He, F., Xu, J., Ding, Z.: Effective map-matching on the most simplified road network. In: Proceedings of the 20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL), pp. 609–612. Redondo Beach (2012)
24. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A., Theodoridis, Y.: R-trees: Theory and Applications. Springer, Berlin (2006)
25. Morse, M.D., Patel, J.M.: An efficient and accurate method for evaluating time series similarity. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 569–580. Beijing (2007)
26. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate nearest neighbor queries in spatial databases. ACM Trans. Database Syst. **30**(2), 529–576 (2005)
27. Pfoser, D., Jensen, C.S.: Indexing of network constrained moving objects. In: Proceedings of the 11th International Symposium on Advances in Geographic Information Systems (GIS), pp. 25–32. New Orleans (2003)
28. Shang, S., Ding, R., Zheng, K., Jensen, C.S., Kalnis, P., Zhou, X.: Personalized trajectory matching in spatial networks. VLDB J. **23**(3), 449–468 (2014)
29. Sherkat, R., Rafiei, D.: On efficiently searching trajectories and archival data for historical similarities. Proc. VLDB Endow. **1**(1), 896–908 (2008)
30. Shortest Paths implementation benchmarks, 9th DIMACS Implementation Challenge, http://www.dis.uniroma1.it/challenge9/download.shtml
31. Tang, L., Zheng, Y., Xie, X., Yuan, J., Yu, X., Han, J.: Retrieving $k$-nearest neighboring trajectories by a set of point locations. In: Proceedings of the 12th International Conference on Advances in Spatial & Temporal Databases (SSTD), pp. 223–241. Minneapolis (2011)
32. Tiakas, E., Rafailidis, D.: Scalable trajectory similarity search based on locations in spatial networks. In: Proceedings of the 5th International Conference Model & Data Engineering (MEDI), pp. 213–224. Rhodes (2015)
33. Tiakas, E., Papadopoulos, A.N., Nanopoulos, A., Manolopoulos, Y., Stojanovic, D., Djordjevic-Kajan, S.: Trajectory similarity search in spatial networks. In: Proceedings of the 10th International Database Engineering & Applications Symposium (IDEAS), pp. 185–192. New Delhi (2006)
34. Tiakas, E., Papadopoulos, A.N., Nanopoulos, A., Manolopoulos, Y., Stojanovic, D., Djordjevic-Kajan, S.: Searching for similar trajectories in spatial networks. J. Syst. Softw. **82**(5), 772–788 (2009)

35. Vlachos, M., Gunopoulos, D., Kollios, G.: Discovering similar multidimensional trajectories. In: Proceedings of the 18th International Conference on Data Engineering (ICDE), pp. 673–684. San Jose (2002)
36. Wenk, C., Salas, R., Pfoser, D.: Addressing the need for map-matching speed: localizing global curve-matching algorithms. In: Proceedings of the 18th International Conference on Scientific & Statistical Database Management (SSDBM), pp. 379–388. Vienna (2006)
37. Xu, J., Güting, R.H., Gao, Y.: Continuous *k* nearest neighbor queries over large multi-attribute trajectories: a systematic approach. GeoInformatica **22**(4), 723–766 (2018)
38. Yi, B., Jagadish, H.V., Faloutsos, C.: Efficient retrieval of similar time sequences under time warping. In: Proceedings of the 14th International Conference on Data Engineering (ICDE), pp. 201–208. Orlando (1998)