Summary

AIS provides detailed information on ships, their trajectories and purpose of venture. However, AIS data contains vast amount of data points that is ever-increasing, as new data is recorded every day. This presents the need for a modern and efficient database structures, that can handle large datasets efficiently. A lot of research have gone into this, but none have yet attempted to utilize a network structure to index the data. Therefore, this paper sets out to create an efficient network structure to index data. It takes inspiration from land-based road networks, which is an extensively researched area. This paper therefore both seeks to optimize current index structures for spatial data and as an attempt to see if land-based methods can be applied in the maritime domain.

This paper contains three main sections: Extracting the network from AIS trajectories, map-matching trajectories and querying using the proposed index. The network extraction consist of a series of steps:

First, a rectangular area, encapsulating Kattegat, is partitioned into uniform grids, which we then build a heatmap from. The heatmap represents the number of trajectories intersecting any given cell and visualizes the maritime routes. The result is referred to as a density grid. A thinning process then eliminates pixels from the density grid, forming a skeleton of the original density grid. Then crossing and paths between crossings is computed from the skeleton, and finally used to extract the network.

The second step is map-matching. Map-matching is the process of associating a trajectory to an edge in the network. The map-matching method, introduced in this paper, differs from the traditional one used in land-based road networks, as we do not require all trajectories to be mapped to an edge. As well as the fact that we split trajectories, that can be partly mapped. Furthermore, a nautical network, as proposed in this paper, is not required to be connected.

Finally, as the network work as an index, we present four new algorithms to insert, update, delete and query data.

108 different configurations of a nautical network are extracted from $44,509$ trajectories from January and February 2021, with each trajectory having a length of at least ten kilometers. The 108 different configurations are tested by indexing $71,941$ trajectories from January 2022, to find the five best configurations for a nautical network. The $71,941$ trajectories have traversed a combined distance of $2,104,914$ kilometers. The result is the five best configurations, reaching upwards of 72% improvement compared to the GiST index implemented in PostGIS.

Maritime Indexing from Nautical Networks

Christopher Colberg Jensen, Jonas Noermark Falkesgaard, Kristian Morsing Pedersen

Computer Science Department, Aalborg University,
Aalborg, Denmark

Emails: {cjen19, jnoerm19, kpede19}@student.aau.dk

***Abstract*—This paper proposes Maritime Indexing from Nautical Networks (MINN), which is an indexing method utilizing a nautical network to improve query response times. The main contributions of this paper are a method to build a nautical network, a method to map-match trajectories to the nautical network and a method to utilize the nautical network for indexing trajectories. The performance of MINN is highly reliant on the nautical network. Therefore, we test 108 different configurations of a nautical network to find the configurations that provides the best performance. The nautical network is built from $44,509$ trajectories from January and February 2021, with each trajectory having a length of at least ten kilometers. The trajectories cover a combined distance of $4,273,441$ kilometers. The index is tested with $71,941$ trajectories from January 2022, traversing a combined distance of $2,104,914$ kilometers. The $71,941$ trajectories are map-matched using a novel approach. This is done for each configuration of the nautical network. We test MINN by executing spatial range queries, and comparing the results to a GiST index. MINN improves query response times by upwards of 72% compared to the GiST index.**

1. Introduction

Within the maritime domain, massive amounts of vessel data is being transmitted every day. Each transmitted data message contains detailed information about a vessel, such as the spatial location, speed and heading. As vessel data volumes are growing rapidly when storing historical vessel data, it is necessary to consider scalability to ensure reasonable query response times. Traditionally, this is accomplished by utilizing an index based on an R-tree [1] or variations thereof [2, 3].

The focus of this paper is to extract a nautical network to index vessel trajectories in order to enhance query response times. To the best of our knowledge, no existing research on this subject exists. Solutions to utilizing a network can be found in the land-based domain, where this subject has been extensively researched [2, 4, 5, 6, 7]. In this paper, we modify existing road network methods used to index trajectories from vehicles to accommodate the maritime domain, i.e., indexing trajectories from vessels, using a nautical network. Inherently, a number of challenges arises from this. First, movement at sea is unconstrained, unlike road networks [2, 4]. Therefore, we extend the capabilities of a network to accommodate unconstrained movement. Other challenges arising from the unconstrained movement are map-matching vessel trajectories, and implementing a nautical network aware index.

Extracting a nautical network is non-trivial. While land-based road networks are restricted by physical roads, a nautical

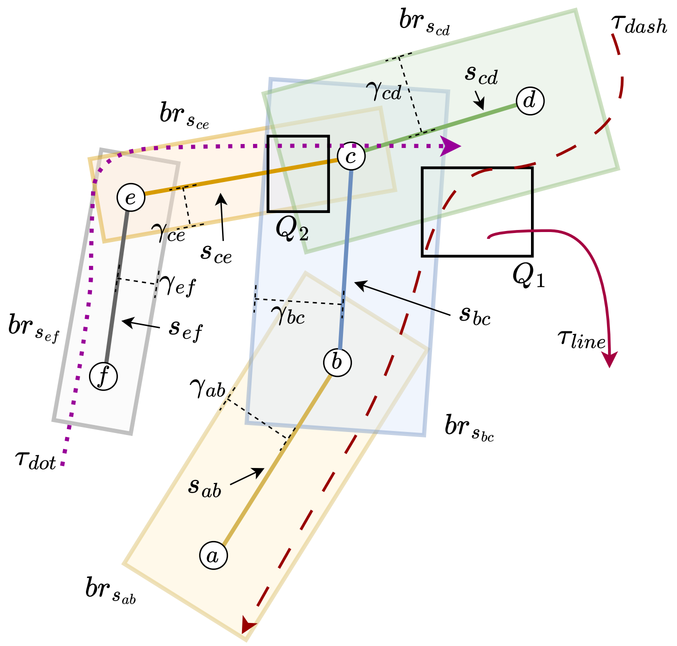

Figure 1: Visualization of MiNN, including spatial queries and trajectories.

network may be restricted by a fairway, but is often not, as is the case at open sea. Therefore, Maritime Indexing from Nautical Networks (MiNN) leverage heatmaps to discover the most popular routes, based on the assumption that vessels follow similar routes. The heatmap is utilized to construct a nautical network through a series of steps and refinements. Hereafter, we use map-matching to associate each vessel trajectory with nautical network edges, using their spatial segments. The segments work as the index for the given vessel trajectory in a network-aware index, such as the one presented by Krogh et al. [4].

Extracting nautical networks from vessel trajectories has, to the best of our knowledge, not yet been used to improve database performance by creating indexes based on the extracted nautical networks. In general, other works [8, 9, 10, 11] examine more than just the longitude and latitude of the vessels to enable a more elaborate analysis of the vessels' movements. The proposed solution, MiNN, does not concern itself with semantics or more complex relationships between vessels and routes, but instead only uses the Global Positioning System (GPS) coordinates of the trajectories with their respective timestamps, to construct a nautical network. MiNN, can therefore also be utilized in other domains beside the maritime domain.

The contributions of this paper are the following:

- A method to create a nautical network from trajectory data
- A method to map-match vessel trajectories to a nautical network
- A novel graph network based indexing method
- An empirical performance study indicating up to a 72% improvement, compared to the PostGIS Generalized Search Tree (GiST) index [12]

The methods for creating nautical networks, map-matching, and the index are described in Section 4, Section 5, and Section 6, respectively. In Section 7, 108 different configurations are used to build nautical networks and the resulting networks are evaluated on a set of trajectories extracted from Automatic Idenfication System (AIS). Each nautical network configuration is built from $44,509$ trajectories from January and February 2021, where trajectories that have covered less than ten kilometers are excluded. Combined, they cover $4,273,441$ kilometers. To evaluate MiNN, a total of $71,941$ trajectories from January 2022 are indexed, covering a combined $2,104,914$ kilometers. Eight spatial range queries have been evaluated on each configuration of the nautical network. MiNN is compared to PostGIS's GiST index [12]. The results show that the best configurations of MiNN outperforms the GiST index by upwards of 72%.

2. Related Work

Krogh et al. [4] and de Almeida and Güting [2] utilize a network structure with edges that forms spatial polylines, and are used to represent trajectories. This is possible, as the trajectories are physically constrained to a network, e.g., roads. de Almeida and Güting associate each trajectory with an edge, which allows a spatial range query to prune irrelevant trajectories by using the intersection between the query and edge polylines. Krogh et al. utilize a shortest path algorithm to represent trajectories as the shortest path between edges in a constrained network. This leads to an efficient in-memory indexing method, while also improving query response times. The idea of utilizing edge polylines to represent trajectories provides the foundation of our solution. In MiNN, instead of polylines to represent edges, we use spatial segments. To accommodate the unconstrained movement, we add a buffer [13] around an edge segment, to indicate that mapped trajectories lie in a region around an edge segment.

Several methods to extract a nautical network from AIS data have been proposed [8, 14]. Filipiak et al. [8] propose using a parallel genetic algorithm on spatially partitioned AIS data, to discover waypoints and construct a directed route graph (i.e., a directed nautical network). While the proposed method by Filipiak et al. exhibits good results, the choice of hyperparameters for the genetic algorithm is area-specific, there are no general values that would work well for all areas, and also requires high computational demands.

A study by Ren et al. [14] propose a method that implements a high-dimensional approach using the CLIQUE [15] and BIRCH [16] algorithms, to consider both the spatial dimension, as well as including course over ground information, extracted from AIS data. The CLIQUE clustering algorithm divides the data space into cells, and identifies dense cells based on a density threshold, i.e., the cells represent areas with frequent maritime traffic, which is used to identify possible waypoints. However, much like the method proposed by Filipiak et al., the method is dependent on parameters chosen for the clustering algorithms.

In contrast, Biagioni and Eriksson [7] propose a hybrid map-inference method, which combines kernel density estimation and trajectory-based refinement to produce a road network. The steps involved consist of creating a density map and using a thinning algorithm to extract road centerlines. Furthermore, Biagioni and Eriksson employ a technique proposed by Shi et al. [17] to determine if each pixel should either be part of an edge or a node. They then refine the road network by map-matching trajectories to the edges in the network and pruning edges that has a low (or zero) trajectories mapped to them.

In MiNN, we aim to construct a nautical network in a fast and efficient manner, and the genetic algorithm proposed in Filipiak et al. shows promising results, but suffers from high computational requirements in order to extract a network in a timely manner. The work by Ren et al. also shows promising results, as does the work by Biagioni and Eriksson, where both methods require significantly less computational resources compared to Filipiak et al.. Therefore, in order to construct the nautical network in MiNN, we adopt the concept of density grids akin to Ren et al. and Biagioni and Eriksson, along with the method for extracting vertices and edges as proposed by Biagioni and Eriksson. As the method proposed by Biagioni and Eriksson does not use AIS data, but rather vehicle data, using it in MiNN will also act as an experiment whether such a method can be utilized in the maritime domain, as it only requires spatial information (i.e., longitude and latitude coordinates), and not, for example, course over ground data as in Ren et al.. The MiNN method will therefore be able to be used with any trajectory data.

To utilize the nautical network for indexing, we use map-matching to map the trajectories to edges in the nautical network. Map-matching has been a research topic for many years, especially for in-vehicle navigation systems in cars, where it must determine exactly which road the vehicle is traversing, often in real time. Some of the more notable being the works by Newson and Krumm [18] and Lou et al. [19]. These works are made for land-based vehicles, such as cars, which are physically restricted to a road. However, in MiNN, we operate on maritime vessels, which we cannot guarantee has actually traversed any actual path in our nautical network, as a vessel is often not physically limited to any path at sea. Another important difference to consider is that a nautical network may not be fully connected, which means utilizing shortest path algorithms on a network can be an issue. Therefore, we do not rely on traditional shortest path algorithms, but we use the concept of shortest distance between network edges and trajectory segments, which is also the basis of the works by Newson and Krumm and Lou et al..

3. Preliminaries

The following section presents preliminaries. First, we define the spatial domain MiNN operates in (Definition 1–5). Second, we define a set of functions, that allows us to extract, modify, and delete elements in the spatial domain (Definition 6–11). Finally, we conclude this section with Definition 12, which describes the base structure of MiNN.

Definition 1 (Spatial Point)**.** *A spatial point* $p = (lng, lat)$ *is a pair, where* lng *is a longitude and* lat *is a latitude. Furthermore, the function* $\textsc{Transform}(p)$ *transforms the coordinate* (lng, lat) *to 2D Cartesian coordinates* x *and* y*, using a map-projection that scales the coordinates to meters (e.g., Universal Transverse Mercator (UTM) [20])*

The $\textsc{Transform}$(p) function is essential for MiNN, as it requires extensive distance computations measured in meters. We use the implementation provided by the PROJ library [21].

Definition 2 (Spatial Segment)**.** *A spatial segment* $s = (p_1, p_2)$ *is defined by a pair of spatial points.*

Definition 3 (Spatio-Temporal Segment)**.** *A spatio-temporal segment* $tps = ((p_1, ts_1), (p_2, ts_2))$ *is defined by a pair of pairs of spatial points and timestamps, where* $ts_1 < ts_2$*.*

Definition 4 (Trajectory)**.** *A trajectory* $\tau = \{tps_1, \ldots, tps_n\}$*, is a set of spatio-temporal segments. From a trajectory* τ *we can extract its spatial segments using the function* $\textsc{Segments}(\tau) = \{(p_i, p_{i+1}) \mid ((p_i, ts_i), (p_{i+1}, ts_{i+1})) \in \tau \wedge i \in [1, n-1]\}$*, i.e., its spatio-temporal segments, without the timestamps.*

We differentiate between spatial- and spatio-temporal segments, as trajectories have to have temporally ordered segments. Besides this, only spatial segments are used for the remainder of the paper, and no ordering is required for the rest of the definitions. Therefore, when referring to segments further on, it refers to spatial segments.

Definition 5 (Density Grid)**.** *In a Euclidean space, a density grid* $\mathcal{DG}$ *is a uniform partitioning of a rectangular region into* $m \times k$ *equal-sized cells. A cell* c *consists of a 4-tuple* $(z, q, square, d)$*, where* z *and* q *is the column- and row index of c, respectively,* $square$ *is the square describing the spatial area of c, and d is the density of cell c, capturing the number of trajectories intersecting it. A cell* c' *is connected to a cell c if it is part of the Moore neighborhood [22] of c. To lookup a cell, we define the function* $\mathcal{L}(z, q, \mathcal{DG})$*, where z and q is the column- and row index of the wanted cell c. It returns cell* $c \in \mathcal{DG}$*, where* $c.z = z \wedge c.q = q$*.*

We use a density grid to form a heatmap, to find high density areas, such that we can extract a nautical network

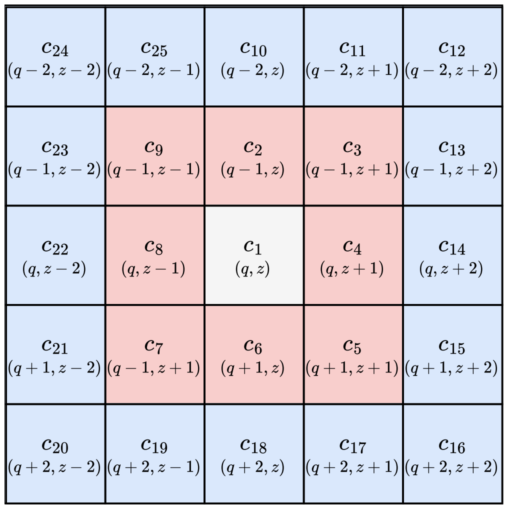

Figure 2: Cell Neighborhood of $\mathcal{N}(c_1, \alpha)$: $\alpha = 1$ yields the red and white cells, and $\alpha = 2$ yields all cells.

that represents the most traffic dense areas, i.e., forming a representative network of the most common traversed routes at sea.

Definition 6 (Cell Neighborhood - modified from Chen and Hsu [23])**.** *Given a cell* $c \in \mathcal{DG}$*, the Cell Neighborhood* $\mathcal{N}(c, \alpha) = \bigcup_{z' \in \{z-\alpha, \ldots, z+\alpha\} \wedge q' \in \{q-\alpha, \ldots, q+\alpha\}} \mathcal{L}(z', q', \mathcal{DG})$*, where* $\alpha \in \{1, 2\}$*.*

A visualization of Definition 6 is shown in Figure 2. If a cell has missing neighbors, e.g., if a cell is in the corner of a density grid, imaginary cells with a density of zero replaces the missing cells. Chen and Hsu only works on the 1-wide neighborhood $\mathcal{N}(c, 1)$, whereas we extend the neighborhood to be a generic term given by α.

Definition 7 (Fringe)**.** *The Fringe* $\mathcal{F}(c)$ *of a cell* $c \in \mathcal{DG}$ *is the set of cells in the outer neighborhood of* $\mathcal{N}(c, 2)$*. That is,* $\mathcal{F}(c) = \mathcal{N}(c, 2) \setminus (\mathcal{N}(c, 1) \cup \{c\})$*.*

An example of a fringe $\mathcal{F}(c_1)$ can be seen in Figure 2, where the blue shaded area corresponds to the fringe of c_1.

In order to form a nautical network, we consider for each cell in a density grid whether it should be a part of the final network or should be removed. To determine this, our algorithms require information about the surrounding cells, to which end the definition of a cell's neighborhood and fringe is used.

Definition 8 (Shortest Distance)**.** *Given two spatial segments* s_1 *and* s_2*, the distance function* $\textsc{MinDist}(s_1, s_2)$ *returns the shortest distance between a pair of a spatial point in* s_1 *and a spatial point in* s_2*. The algorithm computing the* $\textsc{MinDist}(s_1, s_2)$ *function can be seen in Appendix A.*

Definition 9 (Longest Shortest Distance)**.** *Given two spatial segments* s_1 *and* s_2*, the distance function* $\textsc{MaxDist}(s_1, s_2)$ *returns the longest shortest distance between a pair of a spatial* *point in s_1 and a spatial point in s_2. The algorithm computing the* MaxDist(s_1, s_2) *function can be seen in Appendix B.*

The two distance functions are used in our map-matching algorithm to determine whether a trajectory can be matched to an edge. This is done by using the two different types of distance calculations between trajectories and edge segments, to find and filter candidate edges to match to a given trajectory.

Definition 10 (Bounding rectangle)**.** *The bounding rectangle of a segment s with a distance ds,* BUFFER(s, ds) *is the smallest rectangle that contains the Minkowski sum [24] of s, using a circle with a radius of ds.*

An example of a bounding rectangle can be seen in Figure 1, where any of the shaded rectangles corresponds to a bounding rectangle of an edge.

Definition 11 (Boundary)**.** *The boundary of a rectangle rt is the set of line segments that make up its perimeter. The boundary of rt is obtained by the function $\mathcal{BD}(rt)$.*

Definition 12 (Nautical Network)**.** *A Nautical Network Ψ is an undirected graph $G(V, E, \mathrm{T}_{bucket})$, where $V = \{p_1, \dots, p_j\}$ is a set of spatial points, $E = \{(s_1, \mathrm{T}_{s_1}, \gamma_{s_1}, br_{s_1}), \dots, (s_r, \mathrm{T}_{s_r}, \gamma_{s_r}, br_{s_r})\}$ is a set of edges and T_{bucket} is a set of trajectories. Each $e \in E$ is a 4-tuple containing a segment s, referred to as the edge segment, a set of trajectories T_s, a distance γ_s and a bounding rectangle br_s. The set of trajectories T_s is the trajectories that have been mapped to the edge. γ_s denotes the longest shortest distance from s to any trajectory segment, i.e., $\gamma_s \leftarrow \max\{\textsc{MaxDist}(s', s) \mid s' \in \bigcup_{\tau \in \mathrm{T}_s, \textsc{Segments}(\tau)}, s' \cap br_s\}$. The bounding rectangle br_s is computed using the edge segment and γ_s, i.e., $br_s \leftarrow$ BUFFER(s, γ_s). The last element in a nautical network is T_{bucket}, which is the set of trajectories that contains at least one segment, that could not be mapped to any edge in the given nautical network.*

To utilize a nautical network as an index, we extend the notion of an edge. The first element, s, is the segment of the edge, referred to as the edge segment. The second element, T_s, is the set of trajectories associated with the edge. Next, γ_s is a distance threshold, used to calculate a bounding rectangle br_s. The bounding rectangle enables the nautical network to use the edge segments to represent trajectories, while still maintaining their spatial position when querying. For example, in Figure 1, Q_1 does not spatially intersect s_{bc} and s_{cd}, but it does intersect their bounding rectangles. Therefore, we know that some trajectories intersecting Q_1 are represented by either s_{bc}, s_{cd}, or both. Note that a trajectory is not necessarily represented exclusively by one edge segment, but can be represented by multiple.

In the map-matching algorithm, only part of a trajectory segment may be within the maximum mapping distance to an edge segment. Therefore, we split such a segment into sub-segments, by calculating the intersection with the boundary of the bounding rectangle of the candidate edge. An example of such a case is shown in Figure 8 in Section 5, where a trajectory segment s_2 is split into sub-segments, s_4 and s_5. We use the implementation provided by PostGIS PSC, OSGeo [25], for calculating the boundary.

4. Constructing the Nautical Network

The nautical network is structured as a graph, but with the caveat that an edge contains additional information, such as the bounding rectangle of the edge segment, and a set of trajectories mapped to the edge segment, along with the maximum mapping distance. This information is included such that the segments of the edges in a nautical network can be used as an index for trajectories. Extending the edge information with the maximum distance allows us to form a bounding rectangle of an edge. The bounding rectangle is used to prune irrelevant edges based on the intersection between a spatial query region and the bounding rectangles. The bounding rectangles are visualized in Figure 1 where τ_{dot} is mapped to s_{ef}, s_{ce}, s_{cd} and s_{bc}. Thereby, when we perform a spatial query, e.g. Q_2, we find that it intersects the bounding rectangles $br_{s_{ce}}$, $br_{s_{bc}}$ and $br_{s_{cd}}$, and therefore we only examine trajectories mapped to the edge segments s_{ce},s_{cd}, and s_{bc} resulting in τ_{dot} and τ_{dash}.

The following section will describe the process of constructing the nautical network (i.e., the index). An overview of each step in the process can be seen in Figure 3.

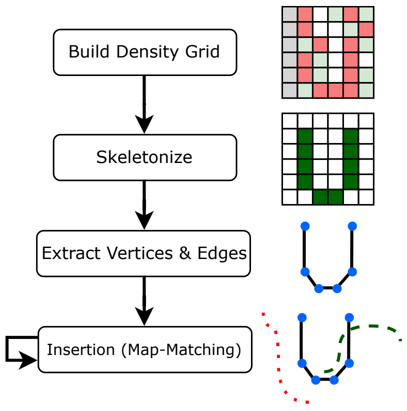

Figure 3: The steps involved in constructing the nautical network (index), along with insertion.

As shown in Figure 3, the first step is to construct the density grid (Definition 5), which is then thinned, until only the skeleton of the grid remains. The skeleton can then be used to extract the vertices and edges, which makes up the nautical network. Finally, the trajectories are map-matched and inserted to the edges in the network.

4.1 Constructing and Thinning the Density Grid

To construct a density grid, a set of trajectories is used in combination with a grid of cells, where each cell's density is incremented by each trajectory spatially intersecting with it (Definition 5). The constructed density grid is then thinned using a thinning algorithm, which is based on the work by Zhang and Suen [26] with the improvements proposed by

Chen and Hsu [23]. Furthermore, we modify the thinning algorithm such that it will work with a density grid $\mathcal{DG}$. The density grid is akin to a bitmap, where each bit corresponds to a cell $c \in \mathcal{DG}$. Furthermore, a bit b can only contain the values $b \in \{0, 1\}$, whereas we use the density of each cell to represents its value, where $c.d \in \{0\} \cup \mathbb{N}$.

The thinning process is iterative, meaning it applies a set of rules to the density grid repeatedly. It terminates when no further changes occur. During an iteration, the algorithm looks at each cell in a density grid and determines whether it should keep its density value or have its density value set to 0 (indicating that the cell should be removed). To simplify the formula of the conditions, we refer to Figure 2 for notation of cells. The cells $c_1, \ldots, c_9$ in the figure, corresponds to the cells $c_1, \ldots, c_9$ in Equation 1–5. Each iteration consists of two sub-iterations, and in the first sub-iteration, the conditions presented in Equations 1–3 are applied.

In Equation 1, $B(c_1)$ is the number of neighbor cells of a cell c_1 with density greater than zero, i.e., $B(c_1) \leftarrow \{c' \mid c' \in \mathcal{N}(c_1, 1), c'.d > 0\}$. Therefore, the first equation states, that a cell c_1 must have at least two and maximum seven cell neighbors, with a density greater than zero, in order to be keep its density value.

In Equation 2 and 3, $A(c_1)$ refers to the number of $(0, d > 0)$ pairs in the sequence $\langle c_2.d, c_3.d, ..., c_9.d, c_2.d \rangle$ where $d > 0$. I.e., let $SP = \{c_2.d, \ldots, c_9.d, c_2.d\}$, $A(c_1)$ is the number of pairs $(c_a.d, c_{a+1}.d) \in SP$, where $c_a.d = 0$ and $c_{a+1}.d > 0$. Note that $c_2.d$ is added at the end such that we also examine the pair $(c_9.d, c_2.d)$.

$$2 \leq B(c_1) \leq 7 \tag{1}$$

$$A(c_1) = 1 \Rightarrow c_2.d \cdot c_4.d \cdot c_6.d = 0 \wedge c_4.d \cdot c_6.d \cdot c_8.d = 0 \tag{2}$$

$$\begin{aligned} A(c_1) = 2 \Rightarrow (c_2.d \cdot c_4.d \geq 1 \wedge c_6.d + c_7.d + c_8.d = 0) \vee \\ (c_4.d \cdot c_6.d \geq 1 \wedge c_2.d + c_8.d + c_9.d = 0) \end{aligned} \tag{3}$$

If the number of $(0, d > 0)$ pairs of a cell is equal to one, Equation 2 is applied. If the number of $(0, d > 0)$ pairs is equal to two, Equation 3 is applied. If all the conditions hold, the density of the examined cell is set to zero, i.e., $c_1.d \leftarrow 0$.

In the second sub-iteration, Equation 1 is applied, along with Equations 4 and 5.

$$A(c_1) = 1 \Rightarrow c_2.d \cdot c_4.d \cdot c_8.d = 0 \wedge c_2.d \cdot c_6.d \cdot c_8.d = 0 \tag{4}$$

$$\begin{aligned} A(c_1) = 2 \Rightarrow (c_2.d \cdot c_8.d \geq 1 \wedge c_4.d + c_5.d + c_6.d = 0) \vee \\ (c_6.d \cdot c_8.d \geq 1 \wedge c_2.d + c_3.d + c_4.d = 0) \end{aligned} \tag{5}$$

The algorithm terminates when no density of any cell in the density grid is changed in an iteration. We denote the output of the algorithm as a skeletonized grid $\mathcal{SG}$. Thinning of the density grid is from hereon referred to as skeletonization.

To visualize the skeletonization process, we refer to Figure 4, where red cells indicate that they will be removed

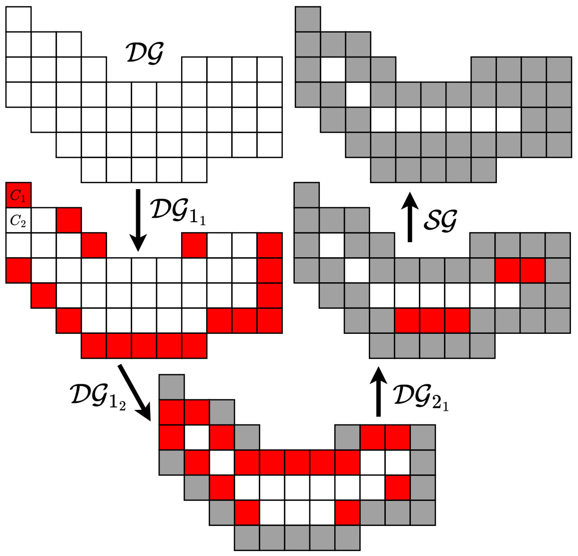

Figure 4: The thinning process (skeletonization) of a density grid $\mathcal{DG}$.

during the current (sub)iteration, gray cells indicate that they have been removed in a previous (sub)iteration and white cells indicate that they are yet to be examined or that they are currently part of the output. As input, the density grid $\mathcal{DG}$ is provided. After applying the first set of rules in the first sub-iteration, the red shaded cells in $\mathcal{DG}_{1_1}$ are removed. For example, the cell c_1 in $\mathcal{DG}_{1_1}$ is marked for removal, as it satisfies Equations 1, 2 and 3, however, cell c_2 is kept in this iteration, as it does not satisfy Equation 3 because $c_6 + c_7 + c_8 \neq 0$ and $c_2 + c_8 + c_9 \neq 0$.

The second sub-iteration is then applied to the output of $\mathcal{DG}_{1_1}$, where more cells are marked for removal, as seen in $\mathcal{DG}_{1_2}$. The next iteration is then applied, with the first sub-iteration being $\mathcal{DG}_{2_1}$, where the last couple of cells are marked for removal. Note, the figure does not show the second sub-iteration of the last iteration, (i.e., $\mathcal{DG}_{2_2}$) as no more cells would be marked for removal. The output is the skeletonized grid $\mathcal{SG}$, which is a subset of the original grid $\mathcal{DG}$.

4.2 *Extracting Vertices and Edges*

Similar to Biagioni and Eriksson [7], we employ a modified version of the method introduced in Shi et al. [17] to extract vertices and edges from a skeletonized grid $\mathcal{SG}$. Shi et al. work on binary pixel images. Therefore, we use the same strategy as in Section 4.1 and extend the set of possible density values of any cell $c \in \mathcal{SG}$ to be $\{-1, 0\} \cup \mathbb{N}$. The output of the following process is denoted a crossing grid $\mathcal{CG}$.

To extract vertices and edges, we iteratively employ Algorithm 1 on all cells in $\mathcal{SG}$. Algorithm 1 is modified based on algorithm 1 by Shi et al.. The algorithm looks at a cell c and a copy of its cell neighborhood $\mathcal{N}(c, 2)$ and determines whether c is a crossing ($c.d \leftarrow -1$), a path ($c.d$ keeps its value), or, in the case that it has no neighboring cells, deletes it from the skeleton grid ($c.d \leftarrow 0$). To visualize this, we refer to Figure 2 and set $c = c_1$. Then if $c_1, c_2, c_{10}, c_{c25}, c_6$ and c_{18} all have a density > 0 and the rest have density 0, c_1 is marked as a path ($c_1.d \leftarrow c_1.d$). If we add $c_4.d > 0$ then c_1 is marked as a crossing ($c_1.d \leftarrow -1$). Finally, if $\forall c' \in \{c_2, c_3, \ldots, c_9\} : c'.d = 0$ then c_1 has no connections and is deleted ($c_1.d \leftarrow 0$). From $\mathcal{SG}$, we build a crossing grid $\mathcal{CG}$, by iteratively running Algorithm 1 on each $c \in \mathcal{SG}$: $\mathcal{CG} = \{c \mid c \in \mathcal{SG}, c.d \leftarrow \text{COMBUST}(c)\}$. Algorithm 1 initial-

Algorithm 1 Combust

Input: A cell c
Output: integer representing crossing, path or deleted

```
1:  function COMBUST($c$)
2:      queue $combusting \leftarrow c$
3:      set $fringe \leftarrow \emptyset$
4:      while $combusting \neq \emptyset$ do
5:          $c' \leftarrow$ DEQUEUE($combusting$)
6:          for all $c'' \in \mathcal{N}(c', 1)$ do
7:              if $c''.d > 0$ then
8:                  $c''.d \leftarrow 0$
9:                  if $c'' \notin \mathcal{F}(c)$ then
10:                     ENQUEUE($combusting, c''$)
11:                 else if $c'' \notin fringe \wedge c'' \in \mathcal{F}(c)$ then
12:                     $fringe \leftarrow fringe \cup \{c''\}$
13:                     ENQUEUE($combusting, c''$)
14:     $zp \leftarrow$ ZEROONEPAIRS($fringe$)
15:     if $zp = 1 \vee zp > 2$ then
16:         return $-1$
17:     else if $zp = 2$ then
18:         return $c.d$
19:     else
20:         return 0
```

izes two variables: a queue *combusting* and a set *fringe*. Then, in the while loop (Line 4), the first element of *combusting* is dequeued (Line 5), such that we can extract and examine its neighbors $c'' \in \mathcal{N}(c', 1)$ in the for loop on Line 6. If c'' has density greater than zero, we set $c.d \leftarrow 0$ (Lines 7–8), marking that it has been examined. If c'' is not part of the fringe of c (Line 9), c'' is added to the queue *combusting* (Line 10). If it is part of the fringe $\mathcal{F}(c)$, but not part of the set *fringe* (Line 11), it is added to the set *fringe* (Line 12) and the queue *combusting* (Line 13). Note that the function fringe $\mathcal{F}(c)$ returns all cells in the fringe of c, whereas the set *fringe* only contains cells in the fringe that have a path to c, where all cells in the path have a density greater than zero. If none of the if-cases hold, we simply continue to the next $c'' \in \mathcal{N}(c', 1)$. In Line 14 we determine the number of $(0, d > 0)$ pairs in the set *fringe*. Note that the function ZEROONEPAIRS(*fringe*) is similar to the function $A(c)$ for Equation 2 - 5, but only looks at cells in the fringe that has a path to c. Finally, in Lines 14–20, the number of $(0, d > 0)$ pairs in the set *fringe* determines if c is a crossing (Line 16), path (Line 18) or should be deleted (Line 20).

Figure 5 illustrates a visual example of Algorithm 1. The black square is the cell under evaluation, the blue squares are marked crossings and the white squares are paths. Initially, we start with a skeletonized grid $\mathcal{SG}$, and examine the black cell, referred to as c_{black}. Again, note the difference between the set *fringe* and the function $\mathcal{F}(c)$. We see that the set *fringe* of c_{black} contains one element, and the density is thereby set to -1, marking it as a crossing. This provides us with $\mathcal{SG}_1$, which has one cell marked as a crossing (blue) and one black cell, which is the next cell to be examined. After examining each cell $c \in \mathcal{SG}$, we obtain the crossing grid $\mathcal{CG}$.

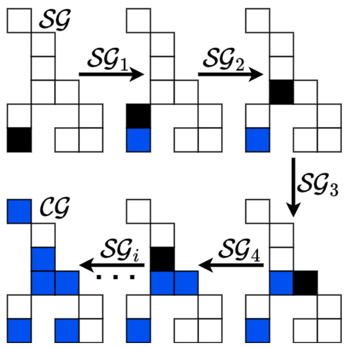

Figure 5: Illustration of Algorithm 1 that finds crossings and paths.

A crossing grid $\mathcal{CG}$ may contain long paths, which forms a small curve. This curve loses its shape if we create a direct edge between the two crossings connected by the path. Therefore, we add another crossing after encountering a given number of path cells in a sequence, such that the curvature of the path is maintained. Note, that the process of adding crossings after a set amount of path cells is not shown in Algorithm 1.

A crossing grid $\mathcal{CG}$ provides the foundation of extracting vertices and edges. Algorithm 2 depicts the overall structure of the process. It utilizes algorithm Algorithm 3 to extract a set of cells forming a crossing.

Algorithm 2 loops through all $c \in \mathcal{CG}$ (Line 3). If a crossing is found (Line 4), we call EXTENDEDCOMBUST(c) (Line 5) which finds every cell in a crossing denoted CS_{vertex}. A further explanation of EXTENDEDCOMBUST(c) (Algorithm 3) follows in the next paragraph. This creates what is known as a main crossing, for which we find all crossings with connecting paths, denoted subordinate crossings [17]. The subordinate crossings are found using the function FOLLOWPATH(c'') (Line 8) which follows each path extending from CS_{vertex} (Lines 7–8). Once a crossing cell c_{cros} is found, EXTENDEDCOMBUST(c_{cros}) is called and the full crossing set extending from c_{cros} is added to the main crossing CS_{vertex}. Finally, we add each main crossing CS_{vertex}, to the set of main crossings $\mathcal{MC}$ (Line 9).

Algorithm 3 is modified and based on algorithm 2 by Shi et al., such that it works with density grids. We extend the set of values that the density of a cell can be to $c.d \in \{-2, -1, 0\} \cup \mathbb{N}$, whereas Shi et al. allows a pixel to have the values $\{0, 1, 2, 3\}$. Algorithm 3 uses a copy of a cell c

Algorithm 2 Build Network Vertices

```
Input: A crossing grid $\mathcal{CG}$
Output: Set $\mathcal{MC}$ containing main crossings and their connected subordinate crossings $CS_i$.
 1: function BUILDNETWORKVERTICES($\mathcal{CG}$)
 2:     set $\mathcal{MC} \leftarrow \emptyset$
 3:     for all $c \in \mathcal{CG}$ do
 4:         if $c.d = -1$ then
 5:             $CS_{vertex} \leftarrow$ EXTENDEDCOMBUST($c$)
 6:             set $\pi \leftarrow \{n \mid c' \in CS_{vertex},$
                        $c'' \in \mathcal{N}(c', 1),\ c'' > 0\}$
 7:             for all $c'' \in \pi$ do
 8:                 $CS_{vertex} \leftarrow CS_{vertex} \cup$ FOLLOWPATH($c''$)
 9:             $\mathcal{MC} \leftarrow \mathcal{MC} \cup \{CS_{vertex}\}$
10:             $vertex \leftarrow vertex + 1$
11:     return $\mathcal{MC}$
```

marked as a crossing as input. It returns a set of cells that form a full crossing CS, i.e., cells with density -1 connected by their Moore neighborhoods. In Line 2 the density of c is set to -2 marking c as part of a main crossing. Line 4 initializes the queue $combusting$. $combusting$ is used to keep track of the cells whose neighbors have not yet been examined. The while loop in Line 5 runs until all cells in the crossing set CS have been found. In the while loop, the head of $combusting$ is extracted such that we can examine its one-wide neighborhood for cells marked as a crossing. We look at each cell c'' in a copy of the neighborhood of c' (Line 7). If $c''.d = -1$, we set $c''.d \leftarrow -2$ marking that it has been examined, add it to the crossing set CS and enqueue c'' to $combusting$, such that c'''s neighbors is examined in a later iteration. If c'' is not marked as a crossing, the loop continues to the next iteration (Line 13). Finally, the set CS is returned in Line 14.

Algorithm 3 Extended Combust

```
Input: A cell $c$
Output: A set of cells $CS$, representing a group of cells marked as one crossing
 1: function EXTENDEDCOMBUST($c$)
 2:     $c.d \leftarrow -2$
 3:     set $CS \leftarrow c$
 4:     queue $combusting \leftarrow c$
 5:     while $combusting \neq \emptyset$ do
 6:         $c' \leftarrow$ DEQUEUE($combusting$)
 7:         for all $c'' \in \mathcal{N}(c', 1)$ do
 8:             if $c''.d = -1$ then
 9:                 $c''.d \leftarrow -2$
10:                 $CS \leftarrow CS \cup \{c''\}$
11:                 ENQUEUE($combusting, c''$)
12:             else
13:                 continue
14:     return $CS$
```

Algorithm 2 is visualized in Figure 6, where gray marks a set of cells as the current main crossing, and blue marks a set of cells as a subordinate crossing. Note that the output of Figure 5 is used as input in Figure 6.

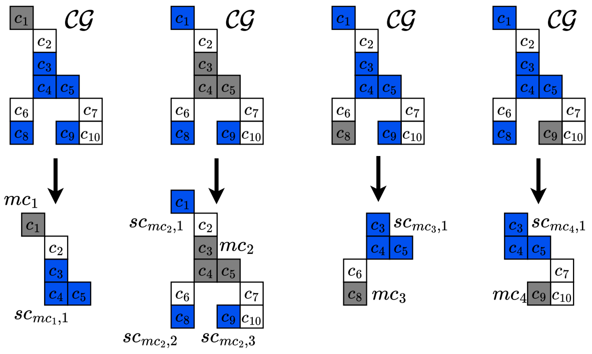

Figure 6: Main crossings mc and their subordinate crossings sc.

First, we look at cell c_1 and mark it as a main crossing mc_1, visualized in gray. Hereafter, we follow the surrounding path cells $c'' \in \pi$ (Lines 7–8 in Algorithm 2). In this case, the only surrounding path cell is c_2. From c_2, we reach a crossing cell c_3 and call EXTENDEDCOMBUST(c_3) (Algorithm 3). This returns the set $\{c_3, c_4, c_5\}$ which is made into the subordinate crossing $sc_{m_1,1}(c_3, c_4, c_5)$ of the main crossing mc_1. This process is repeated for each main crossing (mc_2, mc_3 and mc_4), creating the final set of main crossings with its subordinate crossings $\mathcal{MC} = \{\{mc_1, sc_{mc_1,1}\}, \{mc_2, sc_{mc_2,1}, sc_{mc_2,2} sc_{mc_2,3}\}, \{mc_3, sc_{mc_3,1}\}, \{mc_4, sc_{mc_4,1}\}\}$.

4.2.1 Connecting Vertices: Connecting vertices is a multi-step process. First, the centroid of each set of cells CS_{vertex} in $CS_{vertex} \in \mathcal{MC}$ is extracted, and the subordinate crossings gets connected to the main crossing of CS_{vertex}. A visualization of this is presented in Figure 7. Note that the output of Figure 6 is the input of Figure 7. First, the centroids of the main crossings and subordinate crossing are computed and connected. As any subordinate crossing in any $CS_{vertex} \in \mathcal{MC}$ is a main crossing of another $CS'_{vertex} \in \mathcal{MC}$ it is possible to connect the main crossings through their subordinate crossings as depicted in the final step in Figure 7.

5. Map-Matching

The map-matching algorithm works by iterating through each segment in a trajectory, and for each segment, it finds the edge segments in a nautical network that are within some distance threshold th (i.e., the maximum mapping distance). If no edge segments are within the distance threshold, the algorithm adds the trajectory to $\Psi.\mathrm{T}_{bucket}$, and continues to the next segment. If some candidate edges are found, the algorithm calculates the area of the bounding rectangles required to fully cover the trajectory segment. The edge with the smallest area of its bounding rectangle is selected as the best candidate. The algorithm then calculates the shortest- and longest shortest distance (Definition 8–Definition 9) between

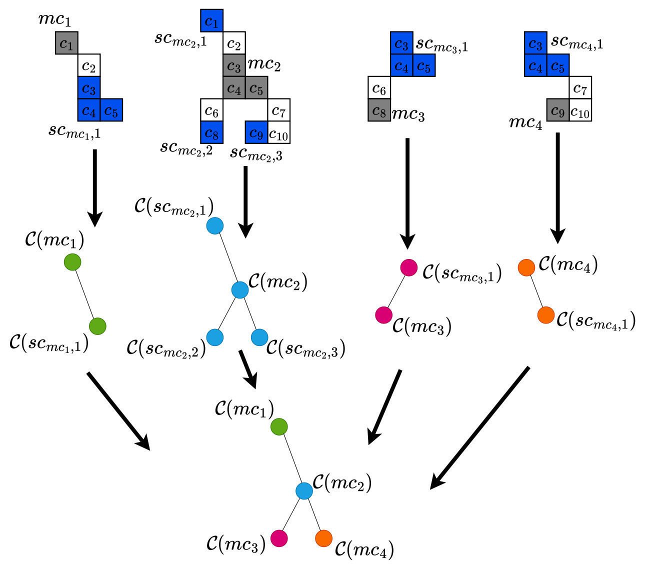
mc_1
c_1
c_2
c_3
c_4 c_5
$sc_{mc_1,1}$
$sc_{mc_2,1}$
mc_2
$sc_{mc_2,2}$ $sc_{mc_2,3}$
$sc_{mc_3,1}$
mc_3
$sc_{mc_4,1}$
mc_4
$\mathcal{C}(mc_1)$
$\mathcal{C}(sc_{mc_1,1})$
$\mathcal{C}(sc_{mc_2,1})$
$\mathcal{C}(mc_2)$
$\mathcal{C}(sc_{mc_2,2})$
$\mathcal{C}(sc_{mc_2,3})$
$\mathcal{C}(sc_{mc_3,1})$
$\mathcal{C}(mc_3)$
$\mathcal{C}(mc_4)$
$\mathcal{C}(sc_{mc_4,1})$

Figure 7: Extracting vertices from main and subordinate crossings.

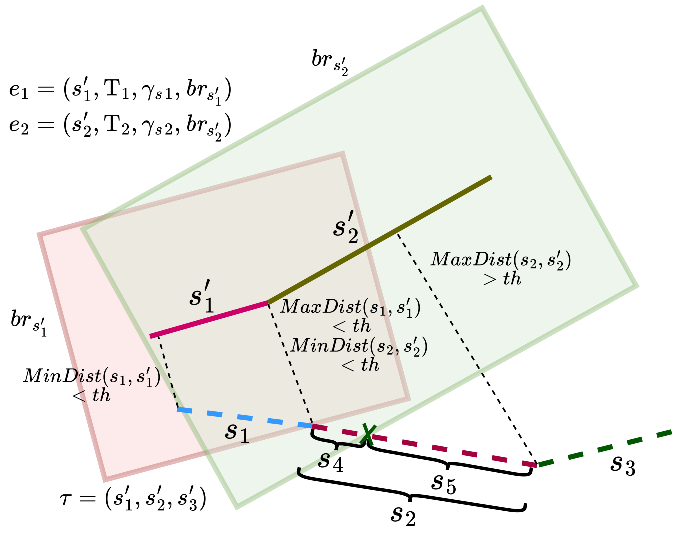
$e_1 = (s'_1, \mathrm{T}_1, \gamma_{s1}, br_{s'_1})$
$e_2 = (s'_2, \mathrm{T}_2, \gamma_{s2}, br_{s'_2})$
$br_{s'_2}$
$br_{s'_1}$
s'_2
s'_1
$MaxDist(s_2, s'_2) > th$
$MaxDist(s_1, s'_1) < th$
$MinDist(s_2, s'_2) < th$
$MinDist(s_1, s'_1) < th$
s_1
s_4
s_5
s_2
s_3
$\tau = (s'_1, s'_2, s'_3)$

Figure 8: Map Matching a trajectory τ containing three segments s_1, s_2 and s_3, to two edge segments s'_1 and s'_2 of the edges e_1 and e_2. Note the trajectory is described using its spatial segments.

the trajectory segment and the candidate segment. The longest shortest distance is the shortest distance from the spatial point in a segment that is farthest away from the other segment (The algorithm can be seen in Appendix B). If the longest shortest distance is within the threshold, we map-match the segment to the selected candidate edge. If the shortest distance is below the threshold, but the longest shortest is not, we enlarge the bounding rectangle of the edge to be equal to the maximum mapping distance threshold, and then split the segment at its intersection with the boundary of the bounding rectangle. This creates new (sub)segments, which are added back to the trajectory, such that they also can be matched, if possible. Should both the shortest- and longest shortest distance be greater than the threshold, the trajectory segment cannot be matched to any edge in the nautical network, and the algorithm moves on to the next segment in the trajectory. The output of the algorithm is a set of pairs $\{(s'_1, ds_1), \ldots, (s'_n, ds_n)\}$, where s' is an edge segment in a nautical network and ds is the longest shortest distance between s' and the trajectory segment that was matched to it. A visual example is presented in Figure 8, and the algorithm is shown in Algorithm 4.

In Figure 8, the nautical network contains two edges e_1 and e_2, and are represented visually by their edge segments s'_1 and s'_2, respectively. The trajectory τ is represented using it spatial segments s_1, s_2, s_3. Looking at segment s_1, both the shortest distance (denoted by $MinDist$), and the longest shortest distance (denoted by $MaxDist$) to the edge segment s'_1 is below the given threshold th. This means, that the bounding rectangle of s'_1 (the red shaded rectangle) can cover the entire segment, and therefore the resulting pair $(s'_1, MaxDist(s, s'_1))$ will be part of the result.

The following trajectory segment s_2 has a shortest distance to s'_2 that is within the threshold th, but the longest shortest distance is greater than the threshold. Therefore, we enlarge the bounding rectangle of s'_2 (the green shaded area) to be equal to the threshold value th, and we split the segment s_2 where the boundary of the bounding rectangle intersects with the segment (the green X in the figure). This creates two new (sub)segments, s_4 and s_5, where both will be added to the trajectory, such that they will be map-matched in a later iteration, i.e., τ now consists of the four segments s_1, s_4, s_5, s_3. The final trajectory segment s_3 is too far away from both s'_1 and s'_2, and can therefore not be map-matched. Therefore, the trajectory shown in Figure 8 can be map-matched to the edge segments s'_1 and s'_2. However, since the trajectory contains a segment that could not be map-matched, it is also added to $\Psi.\mathrm{T}_{bucket}$, i.e., $\Psi.\mathrm{T}_{bucket} \leftarrow \Psi.\mathrm{T}_{bucket} \cup \{\tau\}$.

The map-matching algorithm, shown in Algorithm 4, uses the following input: a trajectory τ, a nautical network Ψ, and a maximum mapping distance threshold th. The output is a set of pairs $\{(s'_1, ds_1), \ldots, (s'_n, ds_n)\}$, where s' is an edge segment, and ds is the longest shortest distance between the edge segment s' and the trajectory segment that was matched to it. In Line 2-3 the result set is initialized to be the empty set, and the set of segments is extracted from the trajectory τ. Then, the for-loop iterates through each segment in Line 4, and in Line 5 we extract a set of candidate edge segments CS from the set E, where the distance between the edge segment s' and the trajectory segment s has a shortest distance that is less than the distance threshold th. In Line 6 a bounding rectangle is computed for each candidate edge, that fully covers the trajectory segment, and the edge segment with the smallest area of its bounding rectangle is chosen as the best candidate edge. In Lines 7–8, we calculate the shortest- and longest shortest distance between the trajectory segment s and s', respectively. In Line 9, the if-statement checks if the longest

Algorithm 4 Map-Matching a Trajectory τ

```
Input: A trajectory $\tau$, a Nautical Network $\Psi$, a maximum
    mapping distance threshold $th$
Output: A set of pairs of segment $s$ and distances $ds$
    $\{(s'_1, ds_1), \dots, (s'_n, ds_n)\}$, that $\tau$ is mapped to.
 1: function MAPMATCH($\tau, \Psi, th$)
 2:     set $result \leftarrow \emptyset$
 3:     set $segments \leftarrow$ SEGMENTS($\tau$)
 4:     for all $s \in segments$ do
 5:         $CS \leftarrow \{s' \mid s' \in \Psi.E, \textsc{MinDist}(s, s') < th\}$
 6:         segment $s' \leftarrow \arg\min_{s' \in CS} \textsc{MinBRArea}(s', s)$
 7:         distance $ds_{min} \leftarrow \textsc{MinDist}(s, s')$
 8:         distance $ds_{max} \leftarrow \textsc{MaxDist}(s, s')$
 9:         if $d_{max} < th$ then
10:             $result \leftarrow result \cup (s', ds_{max})$
11:         else if $d_{min} < th$ then
12:             $segments \leftarrow segments$
                        $\cup$ SPLITSEGMENT($s, s', th$)
13:         else
14:             $\Psi.\mathrm{T}_{bucket} \cup \{\tau\}$
15:             $result \leftarrow result$
16:     return $result$
```

shortest distance is within the maximum mapping threshold. If that is the case, τ is mapped to that edge segment, and it is added to the *result* set. If d_{max} is greater than the threshold, the else-if in Line 11, checks if the shortest distance is less than the threshold. If so, we split the trajectory segment into sub-segments (Line 12), and add the new sub-segments back to the *segment* set, to be processed in a future iteration. If neither of the distances are within the threshold, the trajectory τ gets added to the bucket of trajectories, as seen in Line 14, and the *result* set is not updated with any new pairs, as shown in Line 15. Finally, the result is returned in Line 16.

The function $\textsc{MinBRArea}(s, s')$ can be seen in Appendix C, and the function $\textsc{SplitSegment}(s, s', th)$ is shown in Algorithm 5, where it takes the segment to be split s_1, a segment s_2 and distance ds, where s_2 and ds is used in combination to find the intersection to split the segment s_1 at. The output is a set of segments S. Note that, when the operators $\cup, \cap, \subseteq$ and $\setminus$ are used on spatial elements, such as rectangles and segments, it is equivalent to `ST_Union` [27], `ST_Intersection` [28], `ST_Within` [29], and `ST_Difference` [30], as defined by PostGIS PSC, OSGeo [31], respectively.

In Line 2, in Algorithm 5, a sequence PS is created, with the first spatial point of segment s_1. In Line 3, we create a bounding rectangle around the segment s_2, with the given threshold value th. The boundary of the rectangle is computed in Line 4, which is then used to calculate the intersecting spatial points in Line 5. The segments can then be created using the spatial points, as shown in Line 6. The result is then returned on Line 7.

Algorithm 5 Splits a segment into multiple sub-segments

```
Input: Two segments $s_1$ and $s_2$, and a distance threshold $th$
Output: A set of segments $S$
1: function SPLITSEGMENT($s_1, s_2, th$)
2:     sequence $PS \leftarrow s_1.p_1$
3:     rectangle $bf \leftarrow$ BUFFER($s_2, th$)
4:     boundary $bd \leftarrow \mathcal{BD}(bf)$
5:     $PS \leftarrow PS \cup (s_1 \cap bd) \cup \{s_1.p_2\}$
6:     $S \leftarrow \{(p_i, p_{i+1}) \mid p_i \in PS \land 1 \leq i < |PS|\}$
7:     return $S$
```

6. THE INDEX

Figure 1 visualizes the idea of MiNN. It consists of a nautical network with edge segments and their respective bounding rectangles. Every time a trajectory is map-matched and inserted, it is determined if the bounding rectangle should be expanded to cover the trajectory. For example, τ_{dot} is map-matched to segment s_{ef} and s_{ce} and upon insertion, they expand their bounding rectangles depicted as the gray and orange shaded area, respectively. The same happens for edge segments s_{ab}, s_{bc} and s_{cd} when inserting τ_{dash}. The resulting network and the bounding rectangles of the edge segments is then utilized when executing a spatial range query, e.g. Q_1 and Q_2 from Figure 1. Q_2 is fully contained in the union of the bounding rectangles it intersects. We can therefore guarantee that any trajectory segment intersecting Q_2 is mapped to edge segments s_{ef}, s_{ce}, s_{bc} and s_{cd}, and thereby avoiding evaluating Q_2 against τ_{line}, limiting the amount of trajectories to examine.

6.1 Insertion, Deletion, and Updates

Inserting trajectories into the index is described in Algorithm 6. The algorithm takes a trajectory τ to be inserted, a nautical network Ψ, and a distance threshold th as input. In Line 2 the trajectory τ is map-matched using Algorithm 4, and the output is stored in the set M_τ. Then, in Line 3, a set E_{match} is created. E_{match} contains the pairs of the corresponding edges e, which the trajectory has been mapped to, along with the distances $map.ds$. $map.ds$ is the longest shortest distance between the edge segments, and the trajectory segment that was mapped to it. In Line 4, each pair in E_{match} is looped trough, where the set of trajectories mapped to each edge is updated to include the input trajectory τ (Line 5), along with updating the longest shortest distance γ of the edge, as seen in Line 6.

Deletion of a trajectory is defined in Algorithm 7, which takes the trajectory τ to be removed as input, as well as the nautical network Ψ, where the trajectory should be removed from. Then, in Line 3, for each edge in $E_{contains}$, it removes the trajectory τ from the set T, thereby updating the nautical network.

To update a trajectory, the trajectory is first deleted using Algorithm 7, and then inserted again using Algorithm 6.

Algorithm 6 Trajectory insertion

```
Input: A trajectory $\tau$ to be inserted, a nautical network $\Psi$, a threshold $th$
1: function INSERT($\tau, \Psi, th$)
2:     $M_\tau \leftarrow$ MAPMATCH($\tau, \Psi, th$)
3:     $E_{match} \leftarrow \{(e, map.ds) \mid e \in \Psi.E, map \in M_\tau, e.s = map.s\}$
4:     for all $(e, map.ds) \in E_{match}$ do
5:         $e.\mathrm{T} \leftarrow e.\mathrm{T} \cup \{\tau\}$
6:         $e.\gamma \leftarrow$ MAX($e.\gamma, map.ds$)
```

Algorithm 7 Trajectory Deletion

```
Input: A trajectory $\tau$ to be removed, a nautical network $\Psi$
1: function DELETE($\tau, \Psi$)
2:     $E_{contains} \leftarrow \{(s, \mathrm{T}, \gamma_s) \mid (s, \mathrm{T}, \gamma_s) \in \Psi, \tau \in \mathrm{T}\}$
3:     for all $(s, \mathrm{T}, \gamma_s) \in E_{contains}$ do
4:         $\mathrm{T} \leftarrow \mathrm{T} \setminus \{\tau\}$
5:     $\Psi.\mathrm{T}_{bucket} \leftarrow \Psi.\mathrm{T}_{bucket} \setminus \{\tau\}$
```

6.2 Spatial Range Queries

To perform a spatial range query, MiNN uses Algorithm 8. As input, it takes a spatial range query Q and a nautical network Ψ. It returns a set of trajectories T' intersecting Q. In Line 3 we find the edge segments that have a bounding rectangle intersecting Q and in Line 4 we create the union of those bounding rectangles. Finding intersecting trajectories is done one of two ways: Either Q is fully covered by the union of the intersecting bounding rectangles, determined in Line 6, or it is not fully contained. If it is fully contained, only the trajectories mapped to the edges with intersecting bounding rectangles, is extracted and evaluated against Q (Lines 6-7). If it is not fully contained, we examine two different sets and return the combined result. First, we find the intersection between Q and the bounding rectangles in $E_{intersect}$ (Line 9). Second, we find the spatial difference between $Q_{intersection}$ and Q (Line 10) denoted Q_{diff}. We then find the trajectories in T_P that intersects $Q_{intersection}$ and combine it with the set of trajectories in T_{bucket} intersecting Q_{diff} (Line 11-12). Finally, we return the set T' in Line 13.

To visualize Algorithm 8, we refer to Figure 1, and set $Q = Q_2$. In this case, Q_2 is fully contained within the bounding rectangles it intersects, i.e., $Q_2 \subseteq br_{ce} \cup br_{cd} \cup br_{bc}$. Therefore, Algorithm 8 apply Lines 6–7 which builds the set T' from all trajectories mapped to any edge $e \in P_{intersect}$ that intersects Q_2. For Figure 1 t_{dot} is mapped to s_{ce}, s_{cd} and s_{bc} and intersects Q_2. Thereby, $\mathrm{T}' = \{\tau_{dot}\}$.

For the case that $Q = Q_1$, we see that $Q \nsubseteq P_{union}$ (Line 6 Algorithm 8), and we therefore apply Lines 9–12. We see that $Q_1 \nsubseteq Q_{union}$ and therefore, we first find the set of trajectories intersecting $Q_{intersection}$, which is $\{\tau_{dash}\}$, and then find the set of trajectories intersecting Q_{diff}, which is $\{\tau_{line}\}$, thereby getting the resulting set $\mathrm{T}' = \{\tau_{dash}, \tau_{line}\}$.

Algorithm 8 Evaluation of Spatial Range Queries

```
Input: Spatial range query $Q$ and a nautical network $\Psi$
Output: A set of trajectories intersecting $Q$ denoted $\mathrm{T}'$
1: function RANGEQUERY($Q$, $\Psi$)
2:     $\mathrm{T}' \leftarrow \emptyset$
3:     $E_{intersect} \leftarrow \{e \mid e \in \Psi.E, e.br \cap Q \neq \emptyset\}$
4:     $P_{union} \leftarrow \bigcup_{e.br, e \in E_{intersect}}$
5:     $\mathrm{T}_P \leftarrow \bigcup_{e.\mathrm{T} \in E_{intersect}}$
6:     if $Q \subseteq P_{union}$ then
7:         $\mathrm{T}' \leftarrow \mathrm{T}' \cup \{\tau \mid \tau \in \mathrm{T}_P, \tau \cap Q\}$
8:     else
9:         $Q_{intersection} \leftarrow Q \cap \bigcup_{e.s \in E_{intersect}}$
10:        $Q_{diff} \leftarrow Q \setminus Q_{intersection}$
11:        $\mathrm{T}' \leftarrow \mathrm{T}' \cup \{\tau \mid \tau \in \mathrm{T}_P, \tau \cap Q_{intersection}\}$
12:        $\mathrm{T}' \leftarrow \mathrm{T}' \cup \{\tau \mid \tau \in \Psi_{bucket}, \tau \cap Q_{diff}\}$
13:    return $\mathrm{T}'$
```

7. Experiments and Results

In order to evaluate the performance of MiNN, we perform a set of spatial range queries. Since MiNN heavily relies on the nautical network, different configurations for the nautical network are tested, to find the best of the tested configurations.

7.1 Hardware, Software and Data

The hardware used for building and testing our solution is two machines. One machine is responsible for creating the nautical network as well as running map-matching algorithm. This machine has the following hardware:

- 6-core, 12-thread Intel i7-9750H 2.6Ghz
- 16 GB DDR4 memory
- 512 GB NVMe SSD

The second machine contains the database and therefore the MiNN index. This machine has the following hardware:

- 8-core, 16-thread AMD Ryzen 7 5800x 3.8Ghz
- 32 GB DDR4 memory
- 4 TB NVMe SSD

The database is running the following software:

- PostgreSQL 15.3 [32]
- PostGIS 3.3.2 [31]
- MobilityDB 1.1.0 [33]

The implementation of skeletonization, combustion (Algorithm 1), extended combustion (Algorithm 3), finding vertices and edges (Algorithm 2), and map-matching (Algorithm 4–5) is written in C++ using the GEOS library [34] for spatial calculations. Both skeletonization and map-matching have been implemented with parallelization, significantly improving the run-time of both algorithms. Furthermore, the map-matching algorithm utilizes an in-memory R-tree [1, 34] to index edge segments in a given nautical network, making it significantly faster when computing possible candidate edges. Constructing the density grid, connecting vertices in the nautical network and building the index is done in PostgreSQL [32] using the PostGIS add-on [31]. In PostgreSQL, we assign each edge and trajectory a unique identifier, and connect trajectories and edges through a bridge table. This allows us to index the identifiers using a B-tree [12]. Furthermore, a demo website is developed in TypeScript [35] with React [36] and Leaflet [37]. The website comes with an Application Programming Interface (API) developed in Python [38] with FastAPI [39]. The website is able to show nautical networks with different configurations, show density grids, and manually execute a spatial range query using both the MiNN- and the GiST Index, providing performance numbers and comparisons for both indexes. We use trajectory data from AIS [40], and the trajectories are cleansed. We use $44,509$ trajectories from January and February 2021 to extract a nautical network, where each trajectory must have covered at least 10 kilometers. MiNN is evaluated on $77,941$ trajectories from January 2022.

7.2 Nautical Network

There are several adjustable parameters that influence the resulting nautical network. In order to find the most suitable choice of value for each parameter, several configurations are tested. The adjustable parameters and their possible values can be seen in Table 1, which is ordered by the best mean time improvement.

`Min Density` is a threshold made to filter out cells in the density grid with a density lower than the `Min Density` value. The parameter `Max Path Length` determines how many cells we traverse during the combustion step before adding a crossing, and together with `Cell Size` and `Min Density`, it affects the number of nodes and edges in the nautical network. Lastly, `Max Mapping Distance` affects the maximum size of the bounding polygon of the edges in the nautical network.

To extract the nautical network, we include trajectories from vessels of type Cargo and Tanker since the vessels often rely on routes and fairways to save on fuel costs and minimize travelling time, in theory providing a more accurate representation of the routes. If a trajectory is very short, the nautical network would not gain much information of routes from the trajectory. Therefore, we use trajectories longer than ten kilometers to extract the nautical network. MiNN is built and tested on trajectory data that lies within the gray boundary depicted in Figure 9. MiNN are evaluated by the spatial range queries, denoted SRQ, in Figure 9.

7.3 MINN Index

To evaluate the performance of MiNN, we examine a set of parameters listed below:

- Query response times for MiNN vs PostGIS GiST index [12]
- Size of MiNN bounding rectangles vs size of Minimum Bounding Rectangle (MBR) in GiST
- Query response time performance per query area
- The distance from trajectories to the edge segments they are mapped to
- Amount of trajectories mapped to an edge segment

We perform spatial queries for eight different areas seen in Figure 9. The eight areas are chosen such that we can examine performance on large query areas (area 2), high traffic volume areas (area 1, 5 and 6), harbors (area 4 and 6), and some random areas (3, 7 and 8). Each spatial range query is executed ten times to collect an average reading.

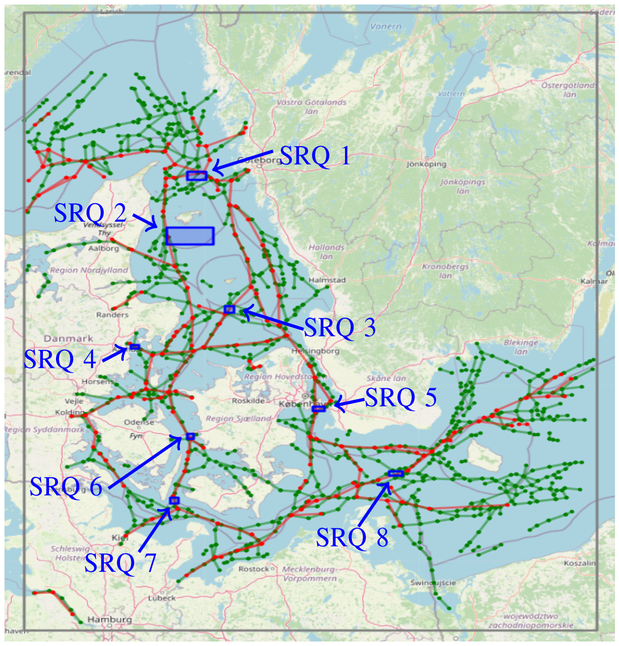

Figure 9: Nautical networks and spatial range queries.

7.4 Results

The spatial range queries are executed on 108 configurations of a nautical network. Table 2 shows the results for the five configurations with the best mean time improvement. Table 2 show that the best configurations for the nautical network does not include networks with a cell size of four kilometers and does also not include any configurations with a max mapping distance lower than five kilometers. The nautical network performs better when the cell size is one or two kilometers. In our ranking, the mean time is valued the highest. Configurations for one kilometer perform almost equal in mean time, but shows a better minimum (№3) or maximum (№4) time improvement. We also see that the number of vertices and edges for a configuration with one kilometer is far greater than the configurations using a cell size of two kilometers. The number of edges in particular can have influence on how fast map-matching can be done. However, map-matching is not necessarily slower because the number of edges and vertices are greater, as Table 6 shows. The map-matching time is mainly determined by the maximum mapping distance and less the number of edges. The max mapping distance also has the largest impact on the percentage of trajectories that can be mapped. To visualize a nautical network, Figure 9 show №1 and №3, from Table 2, colored red and green, respectively.

Parameter	Values	Description
Cell Size	1, 2, 4	The side length (in km) of each cell in the density grid
Min Density	Q1*, Q2*, Q3*	The minimum density a cell must have to be considered
Max Path Length	$\frac{12km}{Cell\ Size}$, $\frac{24km}{Cell\ Size}$, $\frac{48km}{Cell\ Size}$	The maximum number of cells traversed in a path before manually adding a crossing
Max Mapping Distance	1, 2.5, 5, 10	The maximum distance (in km) between a trajectory segment and an edge segment

Table 1: An overview over the parameters and their configuration values. Q1*, Q2*, Q3* is the 25%, 50% and 75% quartile of density values respectively.

№	Cell Size (km)	Max Path Length	Min Density	Max Mapping Distance (km)	Mean Time Improvement	# Vertices	# Edges
№1	2	6	114 (Q3)	10	19%	283	229
№2	2	12	114 (Q3)	10	18%	223	169
№3	1	12	16 (Q2)	5	17%	1, 239	920
№4	2	24	114 (Q3)	10	13%	196	143
№5	1	48	4 (Q1)	5	0%	1, 557	1, 098

Table 2: Best configurations for MiNN, given mean time improvement between MiNN and GiST.

№	Mean Time MiNN (ms)	Mean Time GIST (ms)	Min time improvement	Max Time Improvement	Mean Time Improvement	Median Time Improvement
№1	**73.83**	90.94	−27%	57%	**19%**	**30%**
№2	75.3	90.94	−15%	59%	18%	23%
№3	75.5	90.94	**−11%**	63%	17%	18%
№4	79.9	90.94	−23%	62%	13%	14%
№5	91.1	90.94	−73%	**72%**	0%	12%

Table 3: Time measurements for the best configurations of MiNN. The ranking number refers to a corresponding configuration in Table 2. The best time difference in each column is highlighted in bold and underlined.

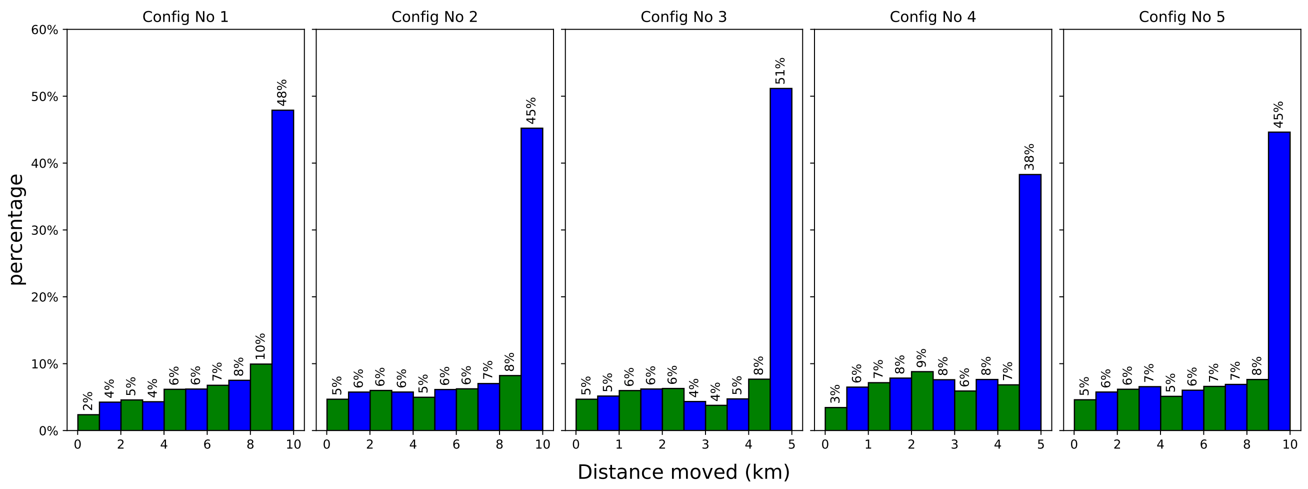

Figure 10: The distance from a trajectory segment to and edge segment it has been mapped to.

	Trajectories	Total Trajectories	% of Total Trajectories
SRQ 1	896	71,941	1.25%
SRQ 2	265	71,941	0.37%
SRQ 3	675	71,941	0.94%
SRQ 4	610	71,941	0.85%
SRQ 5	1,268	71,941	1.76%
SRQ 6	1,226	71,941	1.70%
SRQ 7	663	71,941	0.92%
SRQ 8	1,218	71,941	1.69%

Table 4: Output trajectories for each spatial range query.

	№1	№2	№3	№4	№5
SRQ 1	−27.2%	−15.4%	**−11.5%**	−17.0%	−58.3%
SRQ 2	44.3%	**44.5%**	14.1%	42%	−0.1%
SRQ 3	**36.0%**	22.8%	32.0%	6.9%	−43.3%
SRQ 4	57.0%	59.8%	63.7%	62.1%	**72.2%**
SRQ 5	31.5%	26.7%	34.5%	23.7%	**35.7%**
SRQ 6	**30.0%**	23.6%	22.9%	22.6%	29.9%
SRQ 7	−9.9%	−8.6%	−2.8%	−11.4%	**25.2%**
SRQ 8	**−6.6%**	−7.4%	−10.8%	−23.4%	−73.1%

Table 5: Query Time improvement per area (row) and nautical network configuration (column).

The goal of testing MiNN is to see if there exist a configuration of a network, where the query response time is improved compared to the GiST index [12]. Table 3 show that the five best configurations have a query speed averaging upwards of 19% improvement compared to GiST. Table 3 provides minimum-, maximum-, mean- and median time for each configuration. Note that the mean GiST time is the same for all five configurations, as it is not dependent on the configuration. Instead, the mean GiST time is found by running the eight queries using the GiST index, independently of the configurations. Table 6 shows that we are able to map a majority of the trajectories to the nautical network. We expect the majority of trajectory segments to have a mapping distance nearing the max mapping distance due to the splitting of trajectory segments in the map-matching algorithm (Algorithm 4), which is indicated by Figure 10.

An interesting observation is the low number of vertices and edges in the network. As seen in Table 2 the number of edges and vertices is mainly determined by the cell size, but also the maximum path length and minimum density. Moreover, the low number of edges and vertices is also influenced by the type of vessels used. Since we use cargo and tankers, their size forces them to follow certain routes. This is not inherently

№	Mean Time (ms)	Mapped	Not Mapped	% Mapped
№1	9.23	**57,897**	**14,044**	**80.5%**
№2	7.75	54,195	17,746	75.3%
№3	**7.19**	56,269	15,672	78.2%
№4	9.14	57,819	14,122	80.4%
№5	9.42	57,490	14,451	79.9%

Table 6: Average time to map-match a trajectory, and the number of trajectories that were mapped and not mapped to an edge segment for the top 5 configurations.

seen as a drawback, as the maximum mapping distance ensures that vessels travelling outside these routes are also mapped to an edge. However, this also means, indexing vessel types such as yachts or pleasure crafts, could be an issue, as they do not exhibit the same behavior as tanker and cargo vessels, meaning there likely will not be an edge that their trajectories could be mapped to. I.e., pleasure crafts often sail close to shore, compared to tanker and cargo vessels, and would therefore either not be mapped at all, or require a very large maximum mapping distance, which has a negative performance impact.

Table 4 show the number of trajectories retrieved by each spatial range query. This is important, as the output from a query should be a small fraction of the total data in order to best utilize an index [41]. Table 5 show the percentage improvement gained by using MiNN for each spatial range query compared to GiST [12]. Here we see, that for SRQ 1, SRQ 7 and SRQ 8 MiNN is not better than the original GiST index. However, in most cases MiNN outperforms GiST, and for SRQ 2, which is a spatial range query not fully covered by any of the five nautical networks, MiNN still outperforms GiST by up to 44.5%. The strength of MiNN lies in its ability to reduce the search area compared to GiST, which uses minimum bounding rectangles. This can be seen in Table 7, where, for all spatial range queries encapsulated by MiNN (Appendix D), MiNN reduces the area between 98.3% and 99.8%. Even in regions where the spatial range query is not fully encapsulated (Appendix D) we still reduce the area compared to GiST

PostGIS's GiST index [12] outperforms MiNN for the spatial queries SRQ 1, SRQ 7, and SRQ 8. The explanation for the subpar performance for SRQ 1, SRQ 7 and SRQ 8 is the large amount of intersecting bounding rectangles, affecting the performance of the index negatively.

8. Conclusion

We present a new method to index trajectories named MiNN. We describe three overall process: building a nautical network from a partitioned grid, map-matching trajectories to the nautical network and insert, update, delete and performing spatial queries on a network. The networks are built from 44,509 trajectories from January and February 2021 and tested on 71,941 trajectories from January 2022. MiNN utilizes a

	№1	№2	№3	№4	№5
SRQ 1	98.6%	98.8%	98.8%	**<u>99.5%</u>**	3.9%
SRQ 2	85.1%	85.1%	85.7%	85.1%	88.1%
SRQ 3	99.3%	**<u>99.5%</u>**	99%	98.6%	1.4%
SRQ 4	99.1%	99.1%	99.1%	**<u>99.5%</u>**	99.4%
SRQ 5	99.2%	99.0%	99.0%	**<u>99.7%</u>**	**<u>99.7%</u>**
SRQ 6	**<u>99.6%</u>**	**<u>99.6%</u>**	99.3%	99.2%	99.3%
SRQ 7	**<u>99.8%</u>**	98.2%	98.6%	99.5%	99.0%
SRQ 8	**<u>99.4%</u>**	99.3%	98.3%	98.3%	49.4%

Table 7: Size reduction of area needed to represent trajectories in MiNN vs GiST.

graph structure and introduces the idea of querying on bounding rectangles to optimize query time performance. MiNN reached upwards of 19% mean time improvement and 72% maximum improvement compared to GiST on well-chosen query areas. It shows promising results on large and small query areas, as well as within harbors. Overall, MiNN proved better than GiST with exceptions in the outer regions of the tested area. We believe, that the performance of MiNN in these areas could be improved and outperform GiST if more data was provided in the outer regions.

Future research should focus on creating networks in regions where numerous trajectories intersect, resulting in distorted route patterns. Here, the direction or other filtering measures could assist in finding patterns. Another interesting research area is to implement localized maximum mapping distances, rather than having a global threshold.

Acknowledgments

We would like to thank Kristian Torp[1] and Christian Søndergaard Jensen[2] for their supervision and guidance throughout the project. This paper would not be possible without their expertise.

[1]https://vbn.aau.dk/da/persons/108522

[2]https://vbn.aau.dk/da/persons/christian-s-jensen

References

[1] A. Guttman, "R-trees: a dynamic index structure for spatial searching," ser. SIGMOD '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266

[2] V. T. de Almeida and R. H. Güting, "Indexing the trajectories of moving objects in networks*," *GeoInformatica*, vol. 9, no. 1, pp. 33–60, Mar 2005. [Online]. Available: https://doi.org/10.1007/s10707-004-5621-7

[3] D. Pfoser, C. S. Jensen, Y. Theodoridis *et al.*, "Novel approaches to the indexing of moving object trajectories." in *VLDB*, vol. 2000. Citeseer, 2000, pp. 395–406.

[4] B. Krogh, C. S. Jensen, and K. Torp, "Efficient in-memory indexing of network-constrained trajectories," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPACIAL '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2996913.2996972

[5] H. He, R. Li, S. Ruan, T. He, J. Bao, T. Li, and Y. Zheng, "Trass: Efficient trajectory similarity search based on key-value data stores," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2306–2318.

[6] T. Li, R. Huang, L. Chen, C. S. Jensen, and T. B. Pedersen, "Compression of uncertain trajectories in road networks," *Proc. VLDB Endow.*, vol. 13, no. 7, p. 1050–1063, mar 2020. [Online]. Available: https://doi.org/10.14778/3384345.3384353

[7] J. Biagioni and J. Eriksson. (2012) Map inference in the face of noise and disparity. Accessed February 28th, 2024. [Online]. Available: https://www.cs.uic.edu/~jakob/papers/biagioni-gis12.pdf

[8] D. Filipiak, K. Wecel, M. Stróżyna, M. Stróżyna, and W. Abramowicz, "Extracting maritime traffic networks from ais data using evolutionary algorithm," *Business & Information Systems Engineering*, vol. 62, no. 5, pp. 435–450, Oct 2020. [Online]. Available: https://doi.org/10.1007/s12599-020-00661-0

[9] R. Vettor and C. Guedes Soares, "Detection and analysis of the main routes of voluntary observing ships in the north atlantic," *Journal of Navigation*, vol. 68, no. 2, p. 397–410, 2015.

[10] Z. Yan, Y. Xiao, L. Cheng, R. He, X. Ruan, X. Zhou, M. Li, and R. Bin, "Exploring ais data for intelligent maritime routes extraction," *Applied Ocean Research*, vol. 101, p. 102271, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141118720303631

[11] G. Pallotta, M. Vespe, and K. Bryan, "Traffic route extraction and anomaly detection from ais data," 06 2013.

[12] T. P. G. D. Group. (2024) Postgresql index types. Accessed May 21th, 2024. [Online]. Available: https:

//www.postgresql.org/docs/current/indexes-types.html

[13] PostGIS PSC, OSGeo. (2024) Postgis st_buffer documentation. Accessed May 19th, 2024. [Online]. Available: https://postgis.net/docs/ST_Buffer.html

[14] F. Ren, Y. Han, S. Wang, and H. Jiang, "A novel high-dimensional trajectories construction network based on multi-clustering algorithm," *EURASIP Journal on Wireless Communications and Networking*, vol. 2022, no. 1, p. 18, 2022. [Online]. Available: https://doi.org/10.1186/s13638-022-02108-4

[15] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," *SIGMOD Rec.*, vol. 27, no. 2, p. 94–105, jun 1998. [Online]. Available: https://doi.org/10.1145/276305.276314

[16] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 103–114. [Online]. Available: https://doi.org/10.1145/233269.233324

[17] W. Shi, S. Shen, and Y. Liu, "Automatic generation of road network map from massive gps vehicle trajectories," 11 2009, pp. 1 – 6.

[18] P. Newson and J. Krumm, "Hidden markov map matching through noise and sparseness," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 336–343. [Online]. Available: https://doi.org/10.1145/1653771.1653818

[19] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang, "Map-matching for low-sampling-rate gps trajectories," in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. GIS '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 352–361. [Online]. Available: https://doi.org/10.1145/1653771.1653820

[20] "The universal transverse mercator (utm) grid," Reston, VA, Tech. Rep., 2001, report. [Online]. Available: https://pubs.usgs.gov/publication/fs07701

[21] PROJ contributors, *PROJ coordinate transformation software library*, Open Source Geospatial Foundation, 2024. [Online]. Available: https://proj.org/

[22] E. W. Weisstein, "Moore neighborhood," retrieved 2024. [Online]. Available: https://mathworld.wolfram.com/MooreNeighborhood.html

[23] Y.-S. Chen and W.-H. Hsu, "A modified fast parallel algorithm for thinning digital patterns," *Pattern Recognition Letters*, vol. 7, no. 2, pp. 99–106, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167865588901249

[24] R. Schneider, *Convex Bodies: The Brunn–Minkowski Theory*, 2nd ed., ser. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2013.

[25] PostGIS PSC, OSGeo. (2024) Postgis st_buffer documentation. Accessed May 20th, 2024. [Online]. Available: https://postgis.net/docs/ST_Boundary.html

[26] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Commun. ACM*, vol. 27, no. 3, p. 236–239, mar 1984. [Online]. Available: https://doi.org/10.1145/357994.358023

[27] PostGIS PSC, OSGeo. (2024) Postgis st_union documentation. Accessed May 19th, 2024. [Online]. Available: https://postgis.net/docs/ST_Union.html

[28] ——. (2024) Postgis st_intersection documentation. Accessed May 19th, 2024. [Online]. Available: https://postgis.net/docs/ST_Intersection.html

[29] ——. (2024) Postgis st_within documentation. Accessed May 19th, 2024. [Online]. Available: https://postgis.net/docs/ST_Within.html

[30] ——. (2024) Postgis st_difference documentation. Accessed May 19th, 2024. [Online]. Available: https://postgis.net/docs/ST_Difference.html

[31] ——. (2024) Postgis documentation. Accessed May 5th, 2024. [Online]. Available: https://postgis.net/

[32] The PostgreSQL Global Development Group. (2024) Postgresql documentation. Accessed May 5th, 2024. [Online]. Available: https://www.postgresql.org/docs/15/index.html

[33] E. Zimányi, M. Sakr, and A. Lesuisse, "Mobilitydb: A mobility database based on postgresql and postgis," *ACM Trans. Database Syst.*, vol. 45, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3406534

[34] (2024) Postgis documentation. Accessed May 5th, 2024. [Online]. Available: https://libgeos.org/

[35] Microsoft. (2024) Typescript. Accessed May 18th, 2024. [Online]. Available: https://www.typescriptlang.org/

[36] Meta Open Source. (2024) React. Accessed May 18th, 2024. [Online]. Available: https://react.dev/

[37] Volodymyr Agafonkin et al. (2024) Leaflet. Accessed May 18th, 2024. [Online]. Available: https://leafletjs.com/

[38] (2024) Python documentation. Accessed May 19th, 2024. [Online]. Available: https://docs.python.org/3/

[39] S. Ramírez. (2024) Fastapi documentation. Accessed May 19th, 2024. [Online]. Available: https://fastapi.tiangolo.com/

[40] Llyods List Intelligence. The essential guide to the automatic identification system (ais). Accessed May 5th, 2024. [Online]. Available: https://www.lloydslistintelligence.com/knowledge-hub/data-storytelling/essential-guide-automatic-identification-system-ais-signals

[41] T. P. G. D. Group. (2024) Postgresql documentation: Indexes and order by. Accessed May 21th, 2024. [Online]. Available: https://www.postgresql.org/docs/current/indexes-ordering.html

[42] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and*

Applications, 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008. [Online]. Available: https://link.springer.com/book/10.1007/978-3-540-77974-2

Acronyms

AIS Automatic Idenfication System. 2, 11
API Application Programming Interface. 11

GiST Generalized Search Tree. 2, 11–14
GPS Global Positioning System. 2

MBR Minimum Bounding Rectangle. 11
MiNN Maritime Indexing from Nautical Networks. 1–3, 9–14, 20, 21

UTM Universal Transverse Mercator. 3

Appendix A
Shortest Distance between two Segments

The following algorithm is based on the work by de Berg et al. [42].

Algorithm 9 Calculates the shortest distance between two segments

```
Input: Two segments $s_1$ and $s_2$
Output: Shortest distance between $s_1$ and $s_2$
1: function MinDist($s_1, s_2$)
2:     Transform($s_1.p.1$)
3:     Transform($s_1.p.2$)
4:     Transform($s_2.p.1$)
5:     Transform($s_2.p.2$)
6:     $Ax \leftarrow s_1.p_1.x$
7:     $Ay \leftarrow s_1.p_1.y$
8:     $Bx \leftarrow s_1.p_2.x$
9:     $By \leftarrow s_1.p_2.y$
10:    $Cx \leftarrow s_2.p_1.x$
11:    $Cy \leftarrow s_2.p_1.y$
12:    $Dx \leftarrow s_2.p_2.x$
13:    $Dy \leftarrow s_2.p_2.y$
14:    $r \leftarrow \frac{(Ay - Cy)(Dx - Cx) - (Ax - Cx)(Dy - Cy)}{(Bx - Ax)(Dy - Cy) - (By - Ay)(Dx - Cx)}$
15:    $s \leftarrow \frac{(Ay - Cy)(Bx - Ax) - (Ax - Cx)(By - Ay)}{(Bx - Ax)(Dy - Cy) - (By - Ay)(Dx - Cx)}$
16:    if $0 \leq r \leq 1$ and $0 \leq s \leq 1$ then
17:        return 0
18:    else
19:        Spatial Point $A \leftarrow s_1.p_1$
20:        Spatial Point $B \leftarrow s_1.p_2$
21:        Spatial Point $C \leftarrow s_2.p_1$
22:        Spatial Point $D \leftarrow s_2.p_2$
23:        Distance $A_{s_2} \leftarrow$ PointToSegmentDist($A, s_2$)
24:        Distance $B_{s_2} \leftarrow$ PointToSegmentDist($B, s_2$)
25:        Distance $C_{s_1} \leftarrow$ PointToSegmentDist($C, s_1$)
26:        Distance $D_{s_1} \leftarrow$ PointToSegmentDist($D, s_1$)
27:        return $min(A_{s_2}, B_{s_2}, C_{s_1}, D_{s_1})$
```

Algorithm 10 Calculates the Shortest Distance between a Spatial Point and a Segment

Input: A spatial point p and a segment s
Output: Shortest distance between p and s

```
1:  function PointToSegmentDist($p, s$)
2:      transform($p$)
3:      transform($s.p.1$)
4:      transform($s.p.2$)
5:      $Px \leftarrow p.x$
6:      $Py \leftarrow p.y$
7:      $Ax \leftarrow s.p_1.x$
8:      $Ay \leftarrow s.p_1.y$
9:      $Bx \leftarrow s.p_2.x$
10:     $By \leftarrow s.p_2.y$
11:     $r \leftarrow \frac{(Px - Ax)(Bx - Ax) + (Py - Ay)(By - Ay)}{(Bx - Ax)^2 + (By - Ay)^2}$
12:     if $r \leq 0$ then
13:         $result \leftarrow \sqrt{(Px - Ax)^2 + (Py - Ay)^2}$
14:         return $result$
15:     else if $r \geq 1$ then
16:         $result \leftarrow \sqrt{(Px - Bx)^2 + (Py - By)^2}$
17:         return $result$
18:     else
19:         $s \leftarrow \frac{(Ay - Py)(Bx - Ax) - (Ax - Px)(By - Ay)}{(Bx - Ax)^2 + (By - Ay)^2}$
20:         $result \leftarrow |s| \cdot \sqrt{(Bx - Ax)^2 + (By - Ay)^2}$
21:         return $result$
```

Appendix B
Longest Shortest Distance from a segment to another segment

Algorithm 11 Calculates the longest shortest distance from one segment to another

```
Input: Two segments $s_1$ and $s_2$
Output: Longest Shortest distance from $s_1$ to $s_2$
1: function MaxDist($s_1, s_2$)
2:     $A \leftarrow s_1.p_1$
3:     $B \leftarrow s_1.p_2$
4:     $A_{s_2} \leftarrow$ PointToSegment($A, s_2$)
5:     $B_{s_2} \leftarrow$ PointToSegment($B, s_2$)
6:     return $max(A_{s_2}, B_{s_2})$
```

Appendix C
Area of bounding rectangle of a segment s_1, that is large enough to encapsulate a segment s_2

Algorithm 12 Calculates the area of Boundary Rectangle of segment s_1, that encapsulates a segment s_2

```
Input: Two segments $s_1$ and $s_2$
Output: Area of bounding rectangle of $s_1$, that encapsulates $s_2$.
1: function MinBRArea($s_1, s_2$)
2:     Distance $ds \leftarrow$ MaxDist($s_1, s_2$)
3:     Bounding rectangle $r \leftarrow \mathcal{BR}(s_1, ds)$
4:     width $w \leftarrow \sqrt{(r.p_2.x - r.p_1.x)^2 + (r.p_2.y - r.p_1.y)^2}$
5:     length $l \leftarrow \sqrt{(r.p_4.x - r.p_1.x)^2 - (r.p_4.y - r.p_1.y)^2}$
6:     return $w \cdot l$
```

Appendix D
Query area is encapsulated in MiNN

	№1	**№2**	**№3**	**№4**	**№5**
SRQ 1	Yes	Yes	Yes	Yes	No
SRQ 2	No	No	No	No	No
SRQ 3	Yes	Yes	Yes	Yes	No
SRQ 4	Yes	Yes	Yes	Yes	Yes
SRQ 5	Yes	Yes	Yes	Yes	Yes
SRQ 6	Yes	Yes	Yes	Yes	Yes
SRQ 7	Yes	Yes	Yes	Yes	Yes
SRQ 8	Yes	Yes	Yes	Yes	No

Table 8: Specifies if a query is encapsulated by MiNN.

APPENDIX E
TIME FOR EACH STEP BUILDING MINN

	№1	**№2**	**№3**	**№4**	**№5**
Skeletonize	99	95	$8,168$	99	$20,174$
Computing crossings and connections	236	202	$5,197$	190	$9,998$
Building MiNN database index	$1,537$	$1,516$	$3,576$	$1,559$	$5,445$

Table 9: Time (ms) for each step of building MiNN per configuration.