On Computing the Time-varying Distance between Moving Bodies

MAXIME SCHOEMANS, Université libre de Bruxelles, Belgium
MAHMOUD SAKR, Université Libre de Bruxelles, Belgium and Ain Shams University, Egypt
ESTEBAN ZIMÁNYI, Université libre de Bruxelles, Belgium

A moving body is a geometry that may translate and rotate over time. Computing the time-varying distance between moving bodies and surrounding static and moving objects is crucial to many application domains including safety at sea, logistics robots, and autonomous vehicles. Not only is it a relevant analytical operation in itself, but also it forms the basis of other operations, such as finding the nearest approach distance between two moving objects. Most moving objects databases represent moving objects using a point representation, and the computed temporal distance is thus inaccurate when working with large moving objects. This article presents an efficient algorithm to compute the temporal distance between a moving body and other static or moving geometries. We extend the idea of the V-Clip and Lin-Canney closest features algorithms of computational geometry to track the temporal evolution of the closest pair of features between two objects during their movement. We also present a working implementation of this algorithm in an open-source moving objects database and show, using a real-world example on AIS data, that this distance operator for moving bodies is only about 1.5 times as slow as the one for moving points while providing significant improvements in correctness and accuracy of the results.

CCS Concepts: • **Computing methodologies**; • **Information systems → Spatial-temporal systems**; • **Theory of computation → Computational geometry**; **Data structures and algorithms for data management**;

Additional Key Words and Phrases: Spatio-temporal databases, temporal distance, moving objects

ACM Reference format:
Maxime Schoemans, Mahmoud Sakr, and Esteban Zimányi. 2023. On Computing the Time-varying Distance between Moving Bodies. *ACM Trans. Spatial Algorithms Syst.* 9, 4, Article 29 (November 2023), 28 pages.
https://doi.org/10.1145/3611010

1 INTRODUCTION

When working with moving bodies, geometries that may translate and rotate over time, the time-varying (temporal) distance is of importance in many domains. For instance, it will help analyze near-collision cases among sea vessels or autonomous vehicles. It will also help to analyze the interaction of logistics robots and their surroundings. Existing solutions in the domain of

Authors' addresses: M. Schoemans and E. Zimányi, Université libre de Bruxelles, Brussels, Belgium; emails: maxime.schoemans@ulb.be, esteban.zimanyi@ulb.be; M. Sakr, Université Libre de Bruxelles, Brussels, Belgium, and Ain Shams University, Cairo, Egypt; email: mahmoud.sakr@ulb.be.

2374-0353/2023/11-ART29 $15.00
https://doi.org/10.1145/3611010

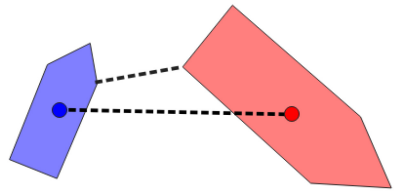


Fig. 1. Nearest approach distance in moving point and moving body representation. The moving point distance is 9.8m, compared to the 3.7m computed using moving bodies.

moving objects databases approximate the moving body by a moving point, e.g., its centroid. This representation, although memory efficient and easier to manipulate, completely disregards the spatial extent of the moving bodies and can result in inaccurate or wrong results. In the example shown in Figure 1, the difference between computing the distance using the moving point presentation vs. using the moving body representation is illustrated. The nearest approach distance between the objects is in reality less than half as much as what is computed using the moving point representation.

In the domain of computer graphics, computing the distance between two rigid bodies is a well-known and heavily researched problem. Some of the presented algorithms are optimized for distance computations on moving bodies [10, 16, 19]. These solutions, however, compute the evolution of the distance by iteratively calling a static distance function at subsequent snapshots. None of these solutions makes use of the parameters of the movement, mostly because they do not assume that the motion is known in advance. These solutions are thus not viable for tracking the continuous distance when the complete movement of the bodies is known, as is the case when analyzing historical moving objects data in a moving objects database. For example, when computing the nearest approach distance between two moving bodies, the minimum distance can be missed if it is reached between two calls of the static distance algorithm. To clarify, the term *distance* denotes the smallest Euclidean distance between two static geometries, and the *minimum distance* thus relates to the smallest distance value reached during the movement of the geometries.

In this article, we present an efficient algorithm to compute the temporal distance between two moving bodies in 2D, when both the geometry and the movement of the bodies are known. The algorithm computes a list storing the evolution of the closest features (vertices of edges) between the moving bodies. This list, together with the moving bodies themselves, completely defines the distance function between these objects. It can then be used to compute the distance at any time during the movement. This algorithm can be used to compute the distance between a pair of moving objects or between a static and a moving object. Note that the latter case is obtained by simply setting all the movement parameters of the static object to zero. We also implement and evaluate the proposed algorithm in a moving objects database.

Concretely, the main contributions of this article are as follows:

- Presenting an efficient algorithm to compute the time-varying distance between convex moving bodies
- Proposing an algorithm for non-convex polygons making use of the solution for convex polygons
- Developing optimizations for the algorithms in case the polygons are non-rotating
- Providing an implementation of multiple distance-related operators using the proposed algorithm in an open-source RDBMS
- Assessing the algorithms and their implementations on both synthetic and real-world data

The rest of the article is organized as follows: Section 2 discusses the related work. We define a representation of moving objects in Section 3, which is compatible with the existing representations of moving points and moving bodies in moving objects databases [12, 27]. Section 4 describes the proposed algorithm for computing the evolution of closest features between moving bodies. Section 5 describes an optimization for this algorithm. Section 6 then presents the implementation of the proposed algorithm in the open-source moving object database MobilityDB. Section 7 validates the theoretical complexity of the proposed algorithms and assesses their running time in an actual database use-case. To conclude, Section 8 summarizes the article and gives some final remarks.

2 RELATED WORK

The moving objects database research defines three main types of moving objects: moving points, deforming moving regions, and non-deforming moving regions (also called moving bodies). Moving points are commonly represented using piecewise linear functions, and computing the temporal distance between them is thus a trivial task. As the distance between two linearly moving points is a square root of a quadratic function, the difficult part consists of representing and storing this function in the database. In SECONDO [4], the data model allows to represent this function as a temporal float object. In MobilityDB [31], an approximation is computed and stored as a piecewise linear function that maintains the extreme points of the initial distance function. One important application of such a function is solving continuous nearest neighbor queries [9, 11, 28].

A distance function for deforming moving regions is presented in [4]. It accepts a moving point and a moving region, as well as a pair of moving regions. The algorithm combines a brute-force technique, computing the distance with every segment of the regions, with a filtering technique, which limits the actual number of segments to improve the performance. The data model represents the deforming moving region as a set of linearly moving segments, i.e., the region edges. This model cannot represent non-deforming moving regions that move by translating and rotating around a given rotation center. Data models have thus been proposed to represent this latter type [12, 27]. However, to our knowledge, the problem of computing the temporal distance between non-deforming moving regions, i.e., moving bodies, has not been addressed before. This article aims at developing such an algorithm.

In the field of computational geometry, there are efficient solutions for computing the distance between static 2D polygons. Using binary search and no prior information, it is possible to determine the distance between two convex static polygons in $O(log(n))$ time, where n is the total number of vertices of the two polygons [3, 7, 30]. This complexity is a lower bound when no extra information is given about these two polygons. When working with non-convex polygons, the main idea is to decompose the polygon into convex parts and build a bounding hierarchy to reduce the number of operations applied to each convex part. These bounding hierarchies can be made of spheres/circles [6, 14, 24, 26] or bounding boxes [17, 22], either axis aligned or not. [24] mentions an average complexity of $O(nlog(n))$ for 2D distance computations between non-convex polygons.

Here, in the case of moving polygons, we observe that the distance is computed multiple times at close time intervals. When working with convex polygons, it is thus often correct to assume that the closest points between these two polygons will not move much between two distance computations. Lin and Canney [16] describe an algorithm to compute the distance between two convex static polygons, as well as the two closest features (vertices or edges) of the two polygons that are at this minimum distance. This algorithm takes not only the two polygons as input but also an estimate of the closest features. In the worst case, the algorithm has a linear time complexity $O(n)$ but can finish in constant time if the estimate of the closest features was not far from the

real closest features. When iteratively computing the distance between two moving polygons, the closest feature of the previous computation is used as an estimate of the next one. This allows for significant time improvements.

Improvements on the Lin-Canney algorithm have also been made, such as the V-Clip [19] and H-Walk [10] algorithms. V-Clip is similar to Lin-Canney but more robust and easier to implement, while H-Walk tries to improve the linear worst-time complexity of both Lin-Canney and V-Clip by representing the object as a hierarchy of convex polygons. A similar hierarchical technique is used in [23] for the purpose of collision detection. An overview of existing static and iterative algorithms for collision detection and distance computation can be found in [18].

The algorithms presented in this article are based on ideas from V-Clip and Lin-Canney but differ from the existing algorithms by taking into account the movement parameters of the moving objects, as well as returning a temporal distance function instead of returning distance values at distinct timestamps. This setting is useful when the movement is known in advance, such as in a database setting storing historical movement data.

As detailed in Section 4, the presented algorithm starts by computing the initial closest features at the start of the movement. This is done using any one of the existing algorithms discussed above. Similarly, as discussed at the end of Section 3, the V-Clip algorithm can be used to check the validity of the result at the end of the algorithm in constant time.

3 PRELIMINARIES

In this section, we present a simple model for moving points and moving bodies that will be used throughout the article. This model is in agreement with more general models described in previous research [4, 8, 12, 27, 31] but restricts the movement of an object to a single segment. Indeed, if the movement is composed of a sequence of these segments (as is usually the case), the algorithm presented in this article can be applied independently to each segment. We discuss how this can be done for the data models presented in [27, 31] at the end of this section.

A 2D moving object is described using a static geometry and associated movement parameters. In this article, the static geometry will be either a point $p = (x_p, y_p)$ or a simple polygon $\mathcal{R}$, and the corresponding moving objects will thus be called *moving point* and *moving body,* respectively. The term *moving body* refers to the fact that the polygon is non-deforming during its movement. We will use the terms *body* and *polygon* interchangeably in the rest of the article.

A simple polygon $\mathcal{R}$ is represented using a list of n vertices stored in counter-clockwise order, together with a rotation center (x_c, y_c) assumed to be inside the polygon. This article does not take into account holes in the polygon:

$$\mathcal{R} = [(x_1, y_1), \ldots, (x_n, y_n) | (x_c, y_c)]. \tag{1}$$

To simplify the notation, we will assume that $(x_{n+i}, y_{n+i}) = (x_i, y_i)$. The modulo operations will thus be omitted in the rest of the equations. To further simplify the notation, when talking about a single vertex, we will use v_i and (x_i, y_i) interchangeably. Edges of the polygon are denoted e_i and represent the linear segment between v_i and v_{i+1}.

We define a point on the edge e_i as $e_i(s) = (x_i(s), y_i(s)) = v_i * (1-s) + v_{i+1} * s$, with $s \in [0, 1]$. The notation $v_i * s$ corresponds to a scalar multiplication and $v_i + v_{i+1}$ is the standard vector addition. This parametric definition of a point along an edge is the same as in [19]. The distance between a point $p = (x_p, y_p)$ and an edge e_i of the polygon can thus be computed using Equation (2). The notation $d(e_i, p)$ is an abuse of notation as e_i is not a vector, but it is to be understood as the minimum Euclidean distance between the edge e_i and p:

$$d(e_i, p) = min_{s \in [0,1]} \left\{ \sqrt{(x_i(s) - x_p)^2 + (y_i(s) - y_p)^2} \right\}. \tag{2}$$

As mentioned before, a moving object combines a static geometry with movement parameters related to that geometry. In the case of a moving point, its movement is defined by a single translation (dx, dy). The coordinates of a moving point $p(t) = (x_p(t), y_p(t))$ can thus be computed at a given $t \in [0, 1]$ using Equation (3). The parameter t represents time, normalized to $[0, 1]$. Thus, $t = 0$ corresponds to the start of the movement and $t = 1$ corresponds to the end of the movement. This will hold for all future equation in this article:

$$\begin{aligned} x_p(t) &= x_p + t * dx \\ y_p(t) &= y_p + t * dy. \end{aligned} \tag{3}$$

Similarly, the movement of a polygon is defined by a translation (dx, dy) and a rotation θ around its rotation center (x_c, y_c). In theory, the algorithm described in this article works for any value of θ. In practice, however, we will assume that θ is a (sufficiently small) constant and we can thus omit it from the complexity estimations of Section 4. In [27], for example, the value of θ for a single segment is restricted between $-\pi$ and π.

A moving polygon $\mathcal{R}(t)$ is thus represented by a list of moving vertices. At any given $t \in [0, 1]$, the coordinates of the vertices are computed using Equation (5):

$$\mathcal{R}(t) = [v_1(t), \ldots, v_n(t) | v_c(t)] \tag{4}$$

$$\begin{aligned} x_i(t) = &(x_i - x_c) * cos(t * \theta) - (y_i - y_c) * sin(t * \theta) \\ &+ x_c + t * dx \\ y_i(t) = &(x_i - x_c) * sin(t * \theta) + (y_i - y_c) * cos(t * \theta) \\ &+ y_c + t * dy. \end{aligned} \tag{5}$$

We define the moving edge $e_i(t)$ as being the moving linear segment between the moving vertices $v_i(t)$ and $v_{i+1}(t)$. Note that Equation (5) guarantees non-deformation. The length of the segments is thus constant during the movement. Like in the static case, we define a point on this edge as $e_i(s, t) = (x_i(s, t), y_i(s, t)) = v_i(t) * (1 - s) + v_{i+1}(t) * s$, with $s \in [0, 1]$.

These definitions of a moving point and moving polygon only represent a single "linear" segment of movement. In practice, however, the movement of an object is composed of a sequence of segments. Some data models group these segments together into a single data object [27, 31], while others define the movement as a set of individual segments [4, 8, 12]. In both cases, the algorithm can be applied on the individual segments independently. The presented algorithm is thus only described for a single segment.

Note that the algorithm requires an initialization step, as described in Section 4. This initialization step computes the initial set of closest features at time $t = 0$ (of the current segment), using existing algorithms. In case the movement is composed of a sequence of segments, the initial set of closest features of the next segment will be the same as the set of closest features at the end of the last segment. The initialization step is thus only required for the very first segment. Between two segments, the validity of the closest features can be tested in constant time using the V-Clip algorithm. In case of invalid features, the result of the V-Clip algorithm will be used as input to the next segment. This avoids propagating errors when working with long trajectories.

Lastly, when computing the distance between two segments having different start and end timestamps, the algorithm can only be applied on the period during which these segments overlap. Let's imagine that the movement of a first object is defined between t_1 and t_2, and the movement of a second object is defined between t_3 and t_4, with $t_1 < t_3 < t_2 < t_4$. The algorithm can thus only be applied on the period $[t_3, t_2]$. In the rest of this article, we always assume that this period is normalized to $[0, 1]$ for simplicity. All equations thus implicitly assume $t \in [0, 1]$.

4 EVOLUTION OF CLOSEST FEATURES

The problem of computing the distance between two static geometries can be reduced to finding the two closest features of these geometries. Features of a geometry are its vertices and edges. For a point geometry, the only existing feature is the point itself. Given the closest features of two geometries, their distance can then be computed in constant time. Indeed, both the distance between two points and the distance between a point and an edge can be computed in constant time.

The same conclusion can be made when computing the temporal distance between two moving objects. If the closest features are known at any time $t \in [0, 1]$, returning the distance between these objects given their closest features can be done in constant time. Knowing this, the remaining problem consists of computing the evolution of the closest features of the two moving objects from $t = 0$ to $t = 1$. In this section we present an algorithm that, given a pair of moving objects (points or polygons), returns a list representing the evolution of closest features during the movement of the objects.

The rest of this section is structured as follows. Section 4.1 first details how we can compute the distance between a moving point and a moving edge in constant time. This section also presents a fundamental equation (Equation (8)) used to detect changes in closest features. Then, Section 4.2 presents the algorithm to find the evolution of closest features between a moving point and a convex moving polygon. Section 4.3 then describes an equivalent algorithm applied on two convex moving polygons. Finally, Section 4.4 describes how the presented algorithms can also be used for non-convex polygons.

4.1 Point-to-edge Distance

This section describes the equations needed to compute the distance between a moving point $p(t) = (x_p(t), y_p(t))$ and a moving edge $e(t)$. These equations are crucial for the algorithms described in Sections 4.2 and 4.3. For simplicity, we omit the subscript of the edge and denote its start and end vertices as $v_s(t)$ and $v_e(t)$, respectively. A point on the moving edge is thus defined using Equation (6). As mentioned, this follows from the static parametric definition of a point along an edge, as used in [19]:

$$e(s, t) = v_s(t) * (1 - s) + v_e(t) * s, \ s \in [0, 1]. \tag{6}$$

The equations of the moving point $p(t)$ are not specified here, as they can in practice be any parametric function of time. In the context of this article, however, we can assume that they correspond to either Equation (3) or Equation (5).

As described in Section 3, the distance between a point and an edge can be computed by minimizing the distance between this moving point and any point on the moving edge (Equation (7)). Again, $d(e(t), p(t))$ is an abuse of notation and denotes the minimum Euclidean distance between the edge $e(t)$ and $p(t)$:

$$d(e(t), p(t)) = min_{s \in [0,1]}\{d(e(s, t), p(t))\}. \tag{7}$$

By differentiating $d(e(s, t), p(t))$ with respect to s and equating to 0, we can find the values of s where this minimum is obtained. This results in a function $s(t)$, given in Equation (8):

$$s(t) = \frac{(p(t) - v_s(t)) \bullet (v_e(t) - v_s(t))}{L^2}, \tag{8}$$

where $x \bullet y$ represent the standard dot operator, and $L^2 = (x_e - x_s)^2 + (y_e - y_s)^2$ is the squared length of the edge.

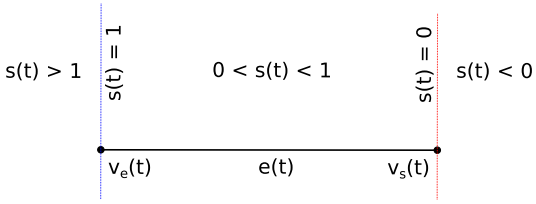


Fig. 2. Visualization of the value of $s(t)$ depending on the position of the point with respect to the edge.

Since we require $s \in [0, 1]$ for the point $e(s, t)$ to be on the edge, we can distinguish three cases:

- $s(t) <= 0$: Point p is closest to the start vertex v_s.
- $s(t) >= 1$: Point p is closest to the end vertex v_e.
- $0 < s(t) < 1$: Point p is closest to $e(s(t), t)$.

Using this equation, we can thus determine the point on the moving edge closest to the moving point. Equation (8) has two main uses. First, it can tell us when the closest point on the edge is one of the end vertices or a point inside the edge. This will be used to determine changes in closest features in the following sections. Second, knowing $s(t)$, we can compute the distance between a moving point $p(t)$ and a moving edge $e(t)$ at any given time $t \in [0, 1]$ in $O(1)$ time using Equation (9):

$$d(e(t), p(t)) = d(e(max(min(s(t), 1), 0), t), p(t)). \tag{9}$$

Figure 2 displays the lines where $s(t) = 0$ (in red) and $s(t) = 1$ (in blue) for a moving edge. The position of the moving point with respect to these lines will thus determine where the closest point on the edge lies. It is important to understand that these lines are not actually parameterized by t. Indeed, if the equation $s(t) = 0$ ($s(t) = 1$) is true for a given t^*, this simply means that the moving point is on the red (blue) line at $t = t^*$. The actual equations of the two lines are not given here as they are of no practical importance.

4.2 Point-to-polygon Distance

In this section, we present an algorithm to compute the evolution of closet features between a moving point and a convex moving polygon in $O(log(n) + k)$ time, where n is the number of vertices of the polygon and k is the size of the result. This algorithm returns a new data object of size k that allows successive computations of the distance to be done in $O(log(k))$ time. The relation between k and n is discussed at the end of the section.

The general idea of the algorithm is to track the closest feature of the moving polygon to the moving point. The features of a polygon are its vertices and edges, and the closest feature of a polygon to a point is thus the vertex or the edge closest to this point. The outer Voronoi diagram of a convex polygon can be used to determine the closest feature given the position of the point. An example of an outer Voronoi diagram is shown in Figure 3. In this figure, the Voronoi regions outside of the polygon are bounded by a red and a blue line, as well as an edge in half of the cases. The red and blue lines are identical to the ones in Figure 2 but applied to the different edges of the polygon. This representation assumes that the vertices of the polygon are listed in counter-clockwise order, as mentioned in Section 3. This assumption will hold in all future figures as well.

The return value of the algorithm is a mapping from time to feature as a list of tuples. Each tuple contains a timestamp $t \in [0, 1]$ and a feature $\mathcal{F}$, which can be either a vertex ($\mathcal{F} = v_j$) or an edge ($\mathcal{F} = e_j$) of the polygon. This list is sorted by increasing timestamp t. Note that this sorting does

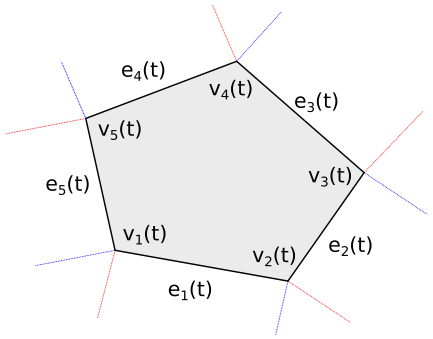


Fig. 3. Outer Voronoi diagram of a convex polygon.

not have to be done in practice, as the list is already being computed in sorted order:

$$\mathcal{L} = [(t_0 = 0, \mathcal{F}_0), \ldots, (t_{k-1}, \mathcal{F}_{k-1}), (t_k = 1, \mathcal{F}_{k-1})]. \tag{10}$$

At every timestamp t, with $t_0 \leq t < t_1$, the closest feature to the moving point is $\mathcal{F}_0$. At $t = t_1$, the closest feature becomes $\mathcal{F}_1$, and so on. In practice, the last tuple in $\mathcal{L}$ of Equation (10) will be omitted since it is only there to describe the fact that the last feature $\mathcal{F}_{k-1}$ is valid from $t = t_{k-1}$ to $t = 1$.

With this data, the distance between the moving point and the moving polygon can be computed in $O(log(k))$ time. Indeed, given any timestamp $t \in [0, 1]$, we can use binary search on the list to find the closest feature at that instant in $O(log(k))$ time. With the closest feature and the moving point, we can then compute the distance in constant time. If the closest feature is a point, the Euclidean distance is used. If it is an edge, the distance is computed using Equation (9). Note that in this case, we are certain that $0 < s(t) < 1$, and the *min* and *max* functions of Equation (9) can thus be omitted.

Next, we describe the algorithm to compute the list $\mathcal{L}$ starting from a convex moving polygon and a moving point.

4.2.1 Algorithm. The algorithm consists of two parts. First, the initial closest feature $\mathcal{F}_0$ is computed. This is a static problem and can be solved using known algorithms in $O(log(n))$ time [30]. Second, given $(t_i, \mathcal{F}_i)$, the algorithm finds the next pair $(t_{i+1}, \mathcal{F}_{i+1})$ as described below. Starting from $(t_0 = 0, \mathcal{F}_0)$, the second part is then called repeatedly using the output of the previous call as input to the next, and this continues until no new closest feature is found at a time $t_i < t < 1$. The remaining problem consists thus of computing $(t_{i+1}, \mathcal{F}_{i+1})$ given $(t_i, \mathcal{F}_i)$ and both moving objects ($p(t)$ and $\mathcal{R}(t)$).

The input feature is either a vertex ($\mathcal{F}_i = v_j$) or an edge of the polygon ($\mathcal{F}_i = e_j$). Figure 4 displays both cases.

The four possible transitions ((a) to (d)) of closest features are shown in Figure 5. If the input closest feature is a vertex of the polygon $\mathcal{F}_i = v_j$ (Figure 5, left), then the next closest feature will be either one of the two edges adjacent to the vertex: $\mathcal{F}_{i+1} = e_j$ or e_{j-1}. This corresponds to cases (a) and (b) of Figure 5, respectively. The boundary lines are defined as in Section 4.1, and determining when either case (a) or (b) happens can be done by solving Equation (11) with respect to t. In these equations, $s_j(t)$ corresponds to Equation (8) applied to the edge $e_j(t)$ and the point $p(t)$:

$$(a): \ s_j(t) = 0 \qquad\qquad (b): \ s_{j-1}(t) = 1. \tag{11}$$

If at least one of these equations has a solution in $[t_i, 1]$, then there will be a change in closest feature. Let's denote the respective solutions t^a (Equation (11), (a)) and t^b (Equation (11), (b)). The

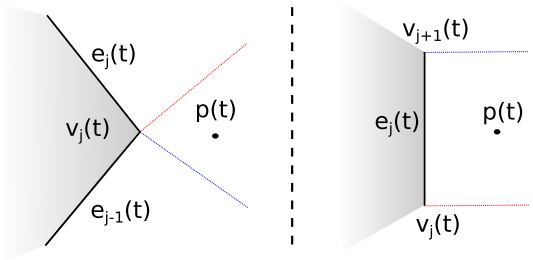


Fig. 4. Examples where the moving point is closest to a vertex (left) or an edge (right) of the moving polygon.

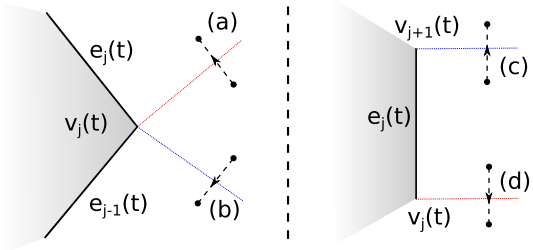


Fig. 5. Two possible transitions per type of initial closest feature.

algorithm will add (t^a, e_j) to the list $\mathcal{L}$ if $t^a < t^b$, and (t^b, e_{j-1}) otherwise. If neither equation has a solution in $[t_i, 1]$, this means that the closest feature will not change again before the end of the movement, and the algorithm will thus terminate and return the current list $\mathcal{L}$.

Similarly, if the initial closest feature is an edge of the polygon $\mathcal{F}_i = e_j$ (Figure 5, right), then the next closest feature will be either the end or the start vertex of this edge $\mathcal{F}_{i+1} = v_{j+1}$ or v_j. This corresponds to cases (c) and (d) of Figure 5, respectively. In these cases, the equations to solve are listed in Equation (12):

$$(c): \ s_j(t) = 1 \qquad\qquad (d): \ s_j(t) = 0. \tag{12}$$

As in the previous case, we denote the solutions t^c and t^d, respectively. The algorithm appends (t^c, v_{j+1}) to the list $\mathcal{L}$ if $t^c < t^d$, and (t^d, v_j) otherwise. If neither equation has a solution in $[t_i, 1]$, the algorithm terminates and returns $\mathcal{L}$.

From the definition of $s(t)$, these equations are non-linear if the rotation of the polygon is nonzero. To solve these equations, numerical methods have to be utilized to find numerical approximations of the solutions. Possible methods include the Newton-Raphson method [1] or bracketing methods such as the false position method or the ITP method [21]. The implementation used in Section 7 uses the false position method to solve these non-linear equations. In case the movement of the polygon contains no rotation, these equations become linear. Section 5 describes direct solutions to the equations under the assumption that the polygons move without rotation.

The examples and equations described in this section do not take into account special cases. For the point-to-polygon distance, there are three special cases that have to be taken into account:

(1) The moving point is on a boundary line of the Voronoi diagram at t_0.
(2) The moving point enters the polygon through the edge e_j in the right image of Figure 5.
(3) The moving point enters the polygon through the vertex v_j in the left image of Figure 5.

All three of these cases can be detected and handled accordingly. In Case 1, the algorithm will have to decide if $\mathcal{F}_0$ corresponds to the left or right case of Figure 4. This can be done by looking

at a timestamp $t_0^* = t_0 + \epsilon$, right after $t_0 = 0$. Both Cases 2 and 3 correspond to an intersection between the moving polygon and the moving point. This is not allowed in the algorithm, and if one of these two cases is detected, the algorithm will fail. This behavior also exists in the polygon-to-polygon algorithm and can thus be used to detect if two moving objects intersect or not. Case 2 can be detected using Equation (13), and Case 3 will happen if the solutions of both (a) and (b) in Equation (11) are true for the same value of t (the moving point crosses the two boundary lines of the Voronoi region at the same time):

$$(p(t) - v_j(t)) \times (v_{j+1}(t) - v_j(t)) = 0, \tag{13}$$

where $v_1 \times v_2$ represents the cross-product between two vectors.

Algorithm 1 summarizes the process of computing the list of closest features between a moving point and a convex moving polygon (without the special cases). The complexity of computing the first closest feature is $O(log(n))$ [30]. Assuming that solving the equations using numerical methods takes constant time, the complete algorithm will finish in $O(log(n)+k)$ time, where k is the length of the list returned by the algorithm. This follows from the fact that the loop only solves two equations per iteration. The cost of running this loop will thus be the number of iterations times the cost of an iteration. As mentioned, we assume that one iteration (solving one or two equations) takes constant time. Each iteration adds a single change of closest feature to the list, and we exit the loop when no new change is found. The number of iterations is $O(k)$, the result size. This gives us a total complexity of $O(log(n) + k)$. The assumption that solving the non-linear equations takes constant time is a simplification made to keep the notation of the complexity the same for both the rotating and the non-rotating case. As can be seen in Section 7.1, this simplification is coherent with the actual runtime of the algorithm. We observe that the non-linear solution is about 5 to 8 times slower than the direct solution, which does not influence the global complexity of the algorithm. The value of k corresponds to the number of times the closest features switched during the movement and is thus very dependent on the translation and rotation parameter of the moving objects. With the assumptions that $\theta \in [-\pi, \pi]$ for the moving polygon and that the moving point either translates linearly or has the equations of a vertex of another moving polygon, the complexity of k will in the worst case be linear in n. In practice, it is clear that if the polygon rotates at most 180 degrees, there cannot be more than n changes in closest features. The worst-case complexity of this algorithm is thus $O(n)$, and the best-case complexity is $O(log(n))$ (if $k = 1$).

4.3 Polygon-to-polygon Distance

This section describes an algorithm to compute the distance between two moving polygons $\mathcal{R}^A(t)$ and $\mathcal{R}^B(t)$, each having their own translation and rotation parameters $(dx^A, dy^A, \theta^A, x_c^A, y_c^A)$ and $(dx^B, dy^B, \theta^B, x_c^B, y_c^B)$. The algorithm is similar to the one described in Section 4.2 in the sense that it also tracks the closest feature between the moving objects. In the case of the distance between two polygons, however, we track two closest features instead of one.

The algorithm processes two moving polygons having n and m vertices, respectively, in $O(log(n) + log(m) + k)$ time and returns a data object that allows successive computations to be done in $O(log(k))$ time. As for the point-to-polygon distance algorithm, the value of k corresponds to the size of the result set and depends heavily on n, m and the movement of both polygons. In particular, $k = O(n + m)$ in the worst case, assuming that the rotation parameter of both polygons is a constant (e.g., $\theta^A \in [-\pi, \pi]$ and $\theta^B \in [-\pi, \pi]$).

The object returned by the algorithm is a list $\mathcal{L}$ of tuples, where each tuple contains a timestamp t and the closest features $\mathcal{F}^A$ and $\mathcal{F}^B$ of the two polygons at that instant:

$$\mathcal{L} = [(t_0 = 0, \mathcal{F}_0^A, \mathcal{F}_0^B), \ldots, (t_{k-1}, \mathcal{F}_{k-1}^A, \mathcal{F}_{k-1}^B)]. \tag{14}$$

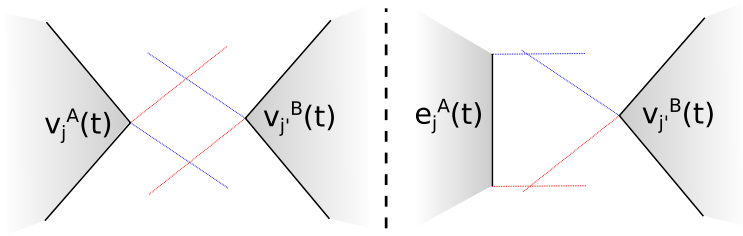


Fig. 6. Initial closest features between two moving polygons: vertex-vertex (left) or edge-vertex (right).

ALGORITHM 1: Point-to-polygon Distance Algorithm

Input: $\mathcal{R}(t), p(t)$
Output: $\mathcal{L}$, the list of closest features with timestamps.
begin
  $\mathcal{F}_0 \leftarrow$ Compute closest feature at $t_0 = 0$ [30];
  $\mathcal{L} \leftarrow [(0, \mathcal{F}^0)]$;
  while *True* **do**
    $(t_i, \mathcal{F}_i) \leftarrow$ Last element of $\mathcal{L}$;
    if $\mathcal{F}_i = v_j$ **then**
      $t^a, t^b \leftarrow$ Solve Equation (11);
      if $t_i < t^a <= 1$ *and* $t^a < t^b$ **then**
        Append (t^a, e_j) to $\mathcal{L}$;
      else if $t_i < t^b <= 1$ **then**
        Append (t^b, e_{j-1}) to $\mathcal{L}$;
      else break ;
    else
      $t^c, t^d \leftarrow$ Solve Equation (12);
      if $t_i < t^c <= 1$ *and* $t^c < t^d$ **then**
        Append (t^c, v_{j+1}) to $\mathcal{L}$;
      else if $t_i < t^d <= 1$ **then**
        Append (t^d, v_j) to $\mathcal{L}$;
      else break ;
  return $\mathcal{L}$;

The first step of the algorithm consists of computing the closest features $\mathcal{F}_0^A$ and $\mathcal{F}_0^B$ at $t_0 = 0$. This is a well-known static problem and can be solved in $O(log(n) + log(m))$ time [30]. The rest of the tuples in the list are then computed by iteratively determining when the next change in closest features happens and what the new closest features are. This process is similar to the one described in Algorithm 1. In this case, however, the closest feature transitions are more complex than in the point-to-polygon case. Each change in closest feature will still be detected in constant time, and the algorithm will exit when no new change in closest features is detected. The loop in the algorithm has thus a complexity of $O(k)$, which results in an algorithm of complexity $O(log(n) + log(m) + k)$.

The initial state of the closest features pair can be either vertex-vertex, vertex-edge, edge-vertex, or edge-edge, where the first and second terms denote the types of closest feature on polygons $\mathcal{R}^A$ and $\mathcal{R}^B$, respectively. The vertex-edge and edge-vertex cases can be handled similarly by simply swapping the two polygons in the equations, and the edge-edge case is a special case that will be discussed later. Figure 6 thus displays the two main cases for the types of initial closest features:

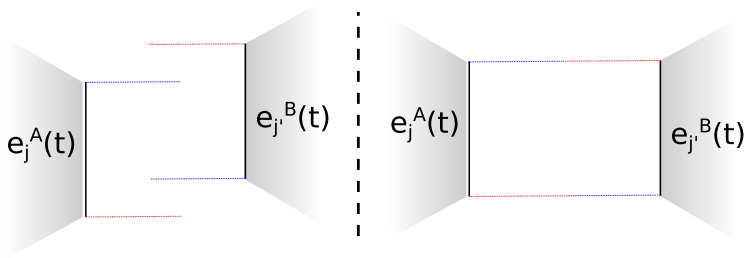


Fig. 7. Two examples of parallel edges as closest features between two polygons. (Right) is a special case of (left).

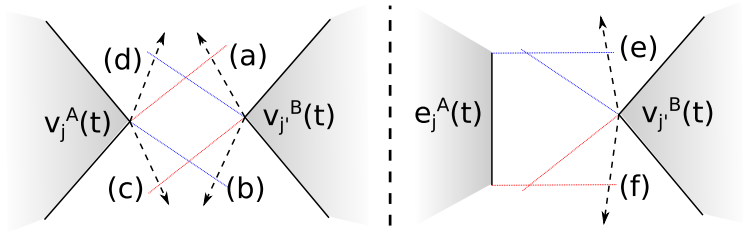


Fig. 8. Evolution of the closest features: vertex-vertex to edge-vertex (left) or edge-vertex to vertex-vertex (right).

vertex-vertex (left) and edge-vertex (right). Figure 7 shows two examples of the special case where the two closest features are parallel edges of the polygons.

In this section, the indices $j \in \{1, \ldots, n\}$ and $j' \in \{1, \ldots, m\}$ will be used for the vertices and edges of polygons $\mathcal{R}^{\mathcal{A}}$ and $\mathcal{R}^{B}$, respectively. The index $i \in \{0, \ldots, k-1\}$ will still be used to denote the different tuples of $\mathcal{L}$.

Starting from one of the two states in Figure 6, three transitions in closest features are possible:

- Vertex-vertex to edge-vertex
- Edge-vertex to vertex-vertex
- Edge-vertex to another edge-vertex

These transitions and their corresponding equations are explained in the following subsections. Since all of the transitions keep the closest feature pairs in either the vertex-vertex or edge-vertex state, the final list $\mathcal{L}$ will only contain vertex-vertex or edge-vertex pairs. This is important for future sections, and this constraint can be met even when handling special cases, e.g., when two edges are parallel and closest.

4.3.1 Vertex-vertex ↔ Edge-vertex. Figure 8 shows the possible transitions that will cause the closest features to go from the vertex-vertex case to the edge-vertex case or back. For clarity, we change the notation of Equation (8) once again and write function $s(t)$ as $s(v, e)(t)$, to specify which moving vertex/point and moving edge are being used. The cases (a) to (f) in Figure 8 can thus be detected by solving Equations (15) to (17):

$$(a): \ s(v_{j'}^B, e_j^A)(t) = 0 \qquad (b): \ s(v_{j'}^B, e_{j-1}^A)(t) = 1, \tag{15}$$

$$(c): \ s(v_j^A, e_{j'}^B)(t) = 0 \qquad (d): \ s(v_j^A, e_{j'-1}^B)(t) = 1, \tag{16}$$

$$(e): \ s(v_{j'}^B, e_j^A)(t) = 1 \qquad (f): \ s(v_{j'}^B, e_j^A)(t) = 0. \tag{17}$$

If $(\mathcal{F}_i^A, \mathcal{F}_i^B) = (v_j^A, v_{j'}^B)$, Equations (15) and (16) have to be solved to determine what the next pair of closest features will be. The edge-vertex pair used in the function $s(v, e)(t)$ corresponds to the next pair of closest features in that particular case. The equation in Equations (15) and (16) that has the earliest solution $t \in [t_i, 1]$ will thus determine uniquely the next pair of closest features. For example, if the equation corresponding to case (a) has the earliest solution $t^a \in [t_i, 1]$, then the next pair of closest features is $(\mathcal{F}_{i+1}^A, \mathcal{F}_{i+1}^B) = (e_j^A, v_{j'}^B)$.

In the edge-vertex case, the process is similar. Equation (17) has to be solved for cases (e) and (f) to determine what the next pair of features will be. If the equation for case (e) has the earliest solution between t_i and 1, the next pair of features is $(\mathcal{F}_{i+1}^A, \mathcal{F}_{i+1}^B) = (v_{j+1}^A, v_{j'}^B)$. Otherwise, the next pair of features is $(\mathcal{F}_{i+1}^A, \mathcal{F}_{i+1}^B) = (v_j^A, v_{j'}^B)$.

Special cases similar to the ones discussed in Section 4.2 can also happen here. Since they are solved in the same way as explained in Section 4.2, they are not discussed further.

4.3.2 Edge-vertex ↔ Edge-vertex. Another possibility when starting from the edge-vertex case is that the edge e_j^A becomes parallel to one of the edges adjacent to the vertex $v_{j'}^B$. If this happens, the edges will be parallel during a single timestamp t, and the next closest features will be a new edge-vertex pair. Figure 9 shows an example of this, where the edge e_i^A becomes parallel to the edge $e_{j'}^B$ during a single timestamp, and the closest features thus evolve from $(e_j^A, v_{j'}^B)$ to $(v_j^A, e_{j'}^B)$ at that timestamp. This is only one of the four possible transitions starting from $(e_j^A, v_{j'}^B)$. The four possible transitions are listed below:

(i) $(e_j^A, v_{j'}^B) \rightarrow (v_j^A, e_{j'}^B)$
(ii) $(e_j^A, v_{j'}^B) \rightarrow (v_{j+1}^A, e_{j'-1}^B)$
(iii) $(e_j^A, v_{j'}^B) \rightarrow (e_j^A, v_{j'+1}^B)$
(iv) $(e_j^A, v_{j'}^B) \rightarrow (e_j^A, v_{j'-1}^B)$

To determine when edge e_j^A becomes parallel to either $e_{j'}^B$ or $e_{j'-1}^B$, Equations (18) and (19) have to be solved:

$$(v_{j+1}^A(t) - v_j^A(t)) \times (v_{j'+1}^B(t) - v_{j'}^B(t)) = 0, \tag{18}$$

$$(v_{j+1}^A(t) - v_j^A(t)) \times (v_{j'}^B(t) - v_{j'-1}^B(t)) = 0. \tag{19}$$

If Equation (18) is true for some t, then e_j^A is parallel to $e_{j'}^B$ at that timestamp, and either case (i) or (iii) of the transitions listed above will happen. These two cases can be distinguished by determining which of the two vertices v_i^A or $v_{j'+1}^B$ is closest to the other edge at t. In the example given in Figure 9, v_i^A is closer to $e_{j'}^B$ than $v_{j'+1}^B$ is to e_i^A, and this corresponds to transition (i) in the list. With two possible transitions per adjacent edge of $\mathcal{R}^B$, we thus get to the four possible transitions listed above.

Just like in Section 4.2, the equations listed in this section are nonlinear in the general case and have to be solved using numerical methods. For all equations, only the first solution between t_i and 1 is required, with $0 \leq t_i < 1$. If the movement of the polygons does not involve a rotation, these equations become linear and Section 5 describes how they can be solved directly without using numerical methods.

4.3.3 Special Cases. Next to the two start states shown in Figure 6 and the transitions detailed in Sections 4.3.1 and 4.3.2, a multitude of special cases exist. Similarly to the ones in Section 4.2, they can be detected and handled in constant time. Below is a non-exhaustive list of these special cases:

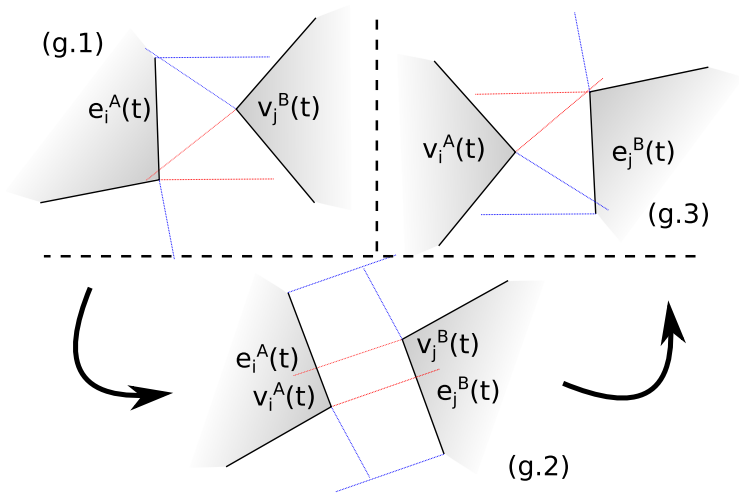


Fig. 9. Evolution from an edge-vertex pair (g.1) to another edge-vertex pair (g.3) by transitioning through an edge-edge pair (g.2).

- Two edges of the polygons are parallel and closest at $t_0 = 0$, as shown in Figure 7.
- The two polygons have the same rotation speed, and two edges are parallel during the complete movement instead of at a single timestamp.
- The polygons do not rotate. See Section 5.
- An edge-vertex pair evolves to a vertex-vertex pair by transitioning though an edge-edge pair. This is a special case of Section 4.3.2.

The first special case essentially corresponds to starting from state (g.2) in Figure 9. In this case, the problem consists of determining the next pair of closest features without knowing what the previous pair was. Notice that we are looking for state (g.3) without knowing state (g.1). This can be solved by looking at the closest features at a timestamp $t_0^* = t_0 + \epsilon$, right after $t_0 = 0$, similarly to what was done for a special case of Section 4.2.

The rest of the special cases are not detailed here but can also be solved in constant time using methods similar to the ones presented here and in Section 4.2. Note that these cases will for the most part never happen when running the algorithm on random or real-world data. For example, even when handling fast-rotating polygons with up to 500 vertices as in the experiments of Section 7.1, more than 99% of the runs did not encounter a single one of these special cases.

4.4 Non-convex Polygons

The algorithms presented in Sections 4.2 and 4.3 assume that the moving polygons are convex, which allows solutions in linear time in terms of number of vertices. In this section, we present an algorithm for non-convex polygons that makes use of the previously presented algorithms for convex polygons. The algorithm for non-convex polygons consists of three steps. First, the polygons are decomposed into convex parts (see Section 4.4.1). Second, partial solutions are computed on the convex parts (see Section 4.4.2). Last, the partial solutions are merged into a final solution (see Section 4.4.3).

4.4.1 Convex Decomposition. The first step of the algorithm consists of decomposing the non-convex polygons into convex parts. This has to be done for both polygons in the case of the

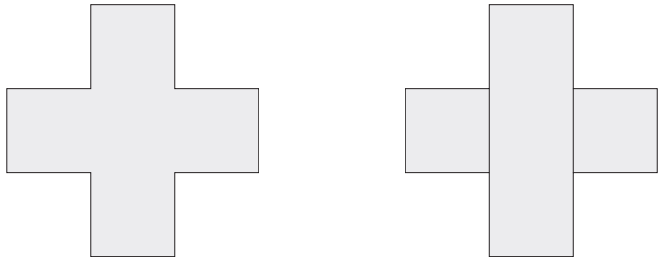


Fig. 10. Non-convex object (a), with an optimal decomposition in three convex parts (b).

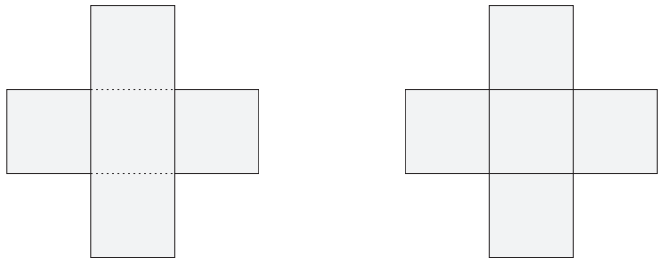


Fig. 11. (a) Overlapping or (b) superfluous parts in the convex decomposition.

polygon-to-polygon distance operation. Even if the polygons are moving, the decomposition only has to be done on the static polygon at $t = 0$. The movement of the convex parts will then be determined using the same parameters as the movement of the initial polygon. Specifically, if the initial polygon has a nonzero rotation, all convex parts will also have a nonzero rotation. The rotation parameters of each convex part are then defined using the same rotation center and rotation angle as the initial polygon.

Convex polygon decomposition is a well-known problem in computation geometry. Chazelle [2] and Keil [15] present two optimal solution in, respectively, $O(n+r^3)$ and $O(r^2 nlog(n))$ time, where r corresponds to the number of reflex vertices (with an inside angle $> 180°$) and n is the total number of vertices. A more practical algorithm is the Hertel-Mehlhorn algorithm [13], which returns a convex decomposition in $O(n + rlog(r))$ time, with at most 4 times the optimal number of convex parts. We will write the number of convex parts of a polygon using the uppercase letter of its number of vertices. For example, a polygon with n vertices will have N convex parts when decomposed using Hertel-Mehlhorn. Figure 10 shows an example of a non-convex polygon, as well as its optimal decomposition in convex parts. In the example shown, we have $N = 3$.

Note that the decomposition required to compute the temporal distance is less restrictive than a traditional convex decomposition problem. For example, Figure 11(a), shows an example of a decomposition in only two overlapping convex parts. This is less than the optimal convex decomposition without overlap, which is beneficial for the next step. Another case is when a convex part does not contain any edges of the initial non-convex polygon. In this case, that specific convex part can be omitted, as there will always be at least one other part with a smaller distance to the second object. An example of such a case is shown in Figure 11(b), where the center square can be omitted. In this specific case, the omitted part is convex as well, but this is not a requirement

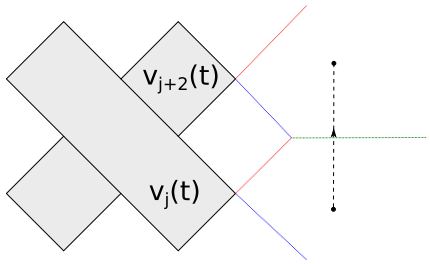


Fig. 12. Evolution of closest features for a non-convex polygon.

since it does not participate in the distance computation. Determining a good (or optimal) convex polygon partitioning, taking into account overlapping and superfluous parts, is left as future work.

4.4.2 Computing Partial Solutions. With the moving polygons being decomposed in convex parts, the next step consists of computing partial solutions using the previously explained algorithms for convex polygons. For the point-to-polygon distance, this has to be done once per convex part. For the polygon-to-polygon distance, every part from one polygon has to be combined with every part of the other polygon. With the two polygons having N and M convex parts, respectively, the algorithms for convex polygons will thus be called N and $N * M$ times for the point-to-polygon and polygon-to-polygon problems, respectively. As discussed in the previous subsection, only convex parts that share an edge with the initial non-convex polygon need to be part of this computation.

Section 2 mentions techniques for non-convex polygons making use of bounding hierarchies to reduce the total number of computations necessary. Similar techniques could be used in this step to improve the running time of the algorithm, but this is left as future work.

4.4.3 Merging Partial Solutions. The last step of the non-convex algorithm consists of merging the partial solutions into a final output. Let's denote the number of partial solutions as L. L will be equal to either N or $N * M$, depending on which problem is being solved. Every partial solution consists of a list of $k_l (l \in \{0, \ldots, L-1\})$ pairs of closest features with their corresponding timestamps. We will also assume that in the point-to-polygon distance problem, the second feature in the closest feature pair always contains the moving point. The two problems can thus be solved using the same algorithm. The remaining problem consists of finding for every timestamp between 0 and 1 which solution contains the actual closest features of the non-convex problem. Figure 12 shows an example of when a closest feature changes from one convex part to another. The green line is a type of boundary in the Voronoi diagram of the non-convex polygon that does not exist in the convex case (see Figure 3).

The problem is solved in multiple steps. First, for every partial solution, the distance is computed at every timestamp in the list of closest features, as well as at $t = 1$. In total, these are $K = \sum_{l=0}^{L-1} k_l + 1$ distance computations. We now have L lists of tuples, where each tuple contains a timestamp, a pair of closest features, and a distance value. The timestamp and distance values can thus be seen as piecewise-linear functions, where every linear piece has an associated pair of closest features. In this representation, the distance between two computed instants is assumed to be linearly interpolated between the previous and the next instant. Figure 13 shows an example with two partial solutions.

In the second step, we track the minimum of these piecewise-linear solutions using a sweep-line algorithm such as the Bentley–Ottmann algorithm [5]. The sweep line will have to maintain a binary search tree with L elements (the L solutions) throughout the complete algorithm. An event

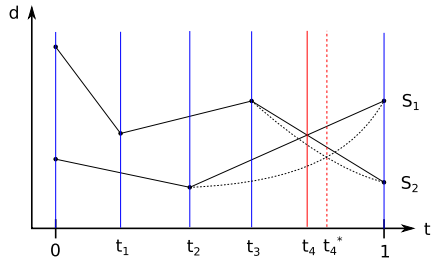


Fig. 13. Piece-wise linear functions of two partial solutions. The dotted red line corresponds to the actual timestamp t_4^* (computed using Equation (22)) at which the solutions switch.

will be either a change in segments from a single solution or a crossing between two solutions. The main purpose of this step is to determine when the actual closest features switch from one convex part to another. This corresponds in the sweep-line algorithm to a crossing between the two lowest elements in the binary search tree. These crossings can be detected using the linear approximation of the distance, but the actual timestamps at which these solutions swap still have to be computed. In Figure 13, for example, one swap in convex parts takes place at the red vertical line. To determine when the actual swap takes place (the red dotted line), however, the real distance functions (the black dotted lines) have to be used instead of the linear approximation.

Since every linear segment of the linear approximation is associated with a pair of closest features, this can be done by finding the timestamps where the distance between the pair associated to the first segment is equal to the distance between the pair associated to the second segment. Depending on the type of closest features, the distance functions will be different. If the closest features are two moving vertices $v^A(t) = (x^A(t), y^A(t))$ and $v^B(t) = (x^B(t), y^B(t))$, then the (squared) distance function corresponds to Equation (20). The distance between a moving vertex $v^A(t) = (x^A(t), y^A(t))$ and a moving edge $e^B(t)$ is equal to the distance between the vertex and the closest point on the edge. From Section 4.1, we know that the closest point on the edge is given by $e^B(s(t), t) = (x^B(s(t), t), y^B(s(t), t))$, where $e^B(s, t)$ is an arbitrary point on the edge, and $s(t)$ is given by Equation (8). The (squared) distance is thus given by Equation (21):

$$d_{AB}^2(t) = (x^B(t) - x^A(t))^2 + (y^B(t) - y^A(t))^2, \tag{20}$$

$$d_{AB}^2(t) = (x^B(s(t), t) - x^A(t))^2 + (y^B(s(t), t) - y^A(t))^2. \tag{21}$$

To determine when the closest features change from $(\mathcal{F}^A, \mathcal{F}^B)$ to $(\mathcal{F}^C, \mathcal{F}^D)$, we thus have to solve Equation (22) between t_i and t_{i+1}, where the actual distance function of both pairs of features is either Equation (20) or (21):

$$d_{AB}^2(t) - d_{CD}^2(t) = 0. \tag{22}$$

The final step of the merging part consists of creating the final list of closest features. This process is straightforward, since after solving Equation (22) for the different switches between minimum solutions, the minimum/closest solution is known for every instant between 0 and 1. In the example in Figure 13, S_1 is closest when $t \in [0, t_4^*]$ and S_2 is closest when $t \in [t_4^*, 1]$.

To summarize the more complex case of polygon-to-polygon distance, the first step consists of decomposing both polygons into convex parts. This will create N and M convex parts, respectively. The second step calls the algorithm for convex polygons using every combination of convex parts of the two initial polygons. This results in $N * M$ partial solutions. Finally, these solutions are merged together. A sweep-line algorithm is used to compute the intersections between the solutions, and the exact timestamps of the intersections between the two minimum solutions are

computed using Equation (22). The final solution is then created by combining the partial solutions at the times where they are the actual closest solution in the non-convex problem.

5 NON-ROTATING POLYGON OPTIMIZATION

Section 3 assumes that the movement of the polygons is a combination of a translation and a rotation. The rotation is the cause of the non-linearity of the equations in Section 4. If the movement of the polygons only contains a translation, then the movement of its vertices is linear, and most of the equations presented in this article become linear or polynomial. If this is the case, then the equations present direct solutions and can thus be solved without numerical methods. Without directly impacting the complexity of the algorithms, this can still significantly improve their performance. Real-world examples where this could be of use include logistics robots, objects moving on fixed tracks, cars on a highway, and more. Whenever an object has a rotation $\theta < \epsilon$ between two timestamps, these direct solutions can be used to speed up the algorithms.

With the movement of the polygons being defined using only translation parameters, the equations of the moving vertices defined in Equation (5) can be simplified to Equation (23):

$$\begin{aligned} x_i(t) &= x_i + t * dx \\ y_i(t) &= y_i + t * dy. \end{aligned} \tag{23}$$

With the equations of the vertices being linear, Equation (6) can be rewritten as Equation (24):

$$\begin{aligned} e(s,t) &= v_s(t) * (1-s) + v_e(t) * s, \ s \in [0,1] \\ &= v_s * (1-s) + v_e * s + t * (dx, dy). \end{aligned} \tag{24}$$

Two important equations that need solving are $s(t) = 0$ and $s(t) = 1$, with $s(t)$ as defined in Equation (8). As a reminder, $p(t) = (x_p(t), y_p(t))$ is the moving point, and $v_s(t)$ and $v_e(t)$ are the start and end points of the moving segment, respectively. We will call (dx_p, dy_p) the translation of the moving point and (dx, dy) the translation of the edge. Examples where these equations are required for different combinations of moving point and moving edge are Equations (11), (12), (15), (16), and (17). The direct solutions for these equations are shown in Equations (25) and (26). These equations only have solutions if the movement of the point is not perpendicular to the edge ($(dx_p - dx) * (x_e - x_s) + (dy_p - dy) * (y_e - y_s) \neq 0$):

$$s(t) = 0 \Leftrightarrow$$

$$t = \frac{(x_s - x_p) * (x_e - x_s) + (y_s - y_p) * (y_e - y_i)}{(dx_p - dx) * (x_e - x_s) + (dy_p - dy) * (y_e - y_s)}, \tag{25}$$

$$s(t) = 1 \Leftrightarrow$$

$$t = \frac{(x_e - x_p) * (x_e - x_s) + (y_e - y_p) * (y_e - y_s)}{(dx_p - dx) * (x_e - x_s) + (dy_p - dy) * (y_e - y_s)}. \tag{26}$$

When working with rotating polygons, it is possible that two edges become parallel, and that the closest features change at that moment (Section 4.3.2). In this case, since the polygons do not rotate, two edges of different polygons are either always or never parallel. Equations (18) and (19) thus do not need to be solved for t anymore.

Using Equations (25) and (26), the algorithms of Sections 4.2 and 4.3 can thus be run without using numerical methods. The experimental evaluation on real data in Section 7.2, shows that more than half the movement segments are non-rotating, and thus the impact of this performance optimization is significant.

6 IMPLEMENTATION IN A MOVING OBJECTS DATABASE

In Section 4 we presented a general solution to the problem of computing the temporal distance between two moving points or polygons. The described algorithms return a list of k closest features with their associated timestamps. This list of closest features, together with the corresponding moving objects, completely defines the equation of the temporal distance between these moving objects. Indeed, the temporal distance is a piece-wise defined function, where every piece is defined using either Equation (20) or Equation (21). To compute the distance value at a given timestamp t, we can thus determine the closest features at that timestamp in $O(log(k))$ time using binary search on the list of features. Given the closest features at t, we then use the corresponding equation (Equation (20) or (21)) to determine the distance value in $O(1)$ time.

When using these algorithms to implement a distance operator in a moving objects database, the returned distance function still needs to be transformed into the data model used in the database. In this section, we describe how the algorithm of Section 4 has been used to implement various distance operators in MobilityDB [31], an open-source moving objects database.

6.1 Linear Approximation of the Distance

MobilityDB defines a *temporal float* type (`tfloat`), which is used to store the temporal evolution of a real-valued parameter as a piecewise linear function. This temporal type is used, for example, to store temperature measurements, the speed of a moving object, and so forth. It is also the return type of the distance operators that involve temporal objects, e.g., the operator computing the distance between two moving points.

Since the temporal distance between two moving objects is not piecewise linear, we need to compute and store its linear approximation. This linear approximation will accurately store the correct start and end distance values of every movement segment, as well as the extreme points of the function, i.e., all minima and maxima. An example of such a linear approximation of a non-linear function is shown in Figure 14 as the red line. This choice of linear approximation is the same as the one already done in MobilityDB when computing the distance between two moving points [31]. Since this approximation maintains the extreme points of the distance function, it can be used to compute the exact nearest approach distance between two moving bodies. This is also explained in Section 6.2.

The previously presented distance algorithm returns a list $\mathcal{L}$ representing the evolution of closest features between two moving objects. In case the distance is computed between a moving point and a moving polygon, we can assume that the second feature always corresponds to the moving point itself:

$$\mathcal{L} = [(t_0 = 0, \mathcal{F}_0^A, \mathcal{F}_0^B), \ldots, (t_{k-1}, \mathcal{F}_{k-1}^A, \mathcal{F}_{k-1}^B), (t_k = 1, \mathcal{F}_{k-1}^A, \mathcal{F}_{k-1}^B)]. \tag{27}$$

As explained in Section 4.3, the closest feature pairs stored in the list can either be two moving vertices or a moving vertex and a moving edge. Between every pair of timestamps $[t_i, t_{i+1}]$, the closest features remain the same. In case these closest features are two moving vertices, the (squared) distance function is described by Equation (20). If the two features are a moving vertex and a moving edge, the function corresponds to Equation (21).

The linear approximation of the distance function is computed in three steps. First, for every period $[t_i, t_{i+1}], i \in \{0, \ldots, k-1\}$, the time coordinate of the extreme points is computed for the corresponding distance function. This is done by computing the derivative of the (squared) distance functions and finding the roots of this new function between t_i and t_{i+1}. Again, this has to be solved using numerical methods if the rotation is nonzero. In this case, however, we are looking for all the roots, instead of only the first. If there is no rotation, the distance functions

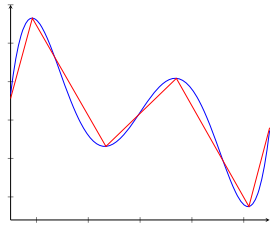


Fig. 14. Approximation of a nonlinear continuous function (in blue) using a piecewise-linear function (in red).

distance	tgeometry $\times \mathcal{A}$	$\rightarrow$	tfloat
nearestApproachDistance	tgeometry $\times \mathcal{A}$	$\rightarrow$	float
nearestApproachInstant	tgeometry $\times \mathcal{A}$	$\rightarrow$	timestamptz
shortestLine	tgeometry $\times \mathcal{A}$	$\rightarrow$	geometry

Fig. 15. List of implemented distance operators.

will be quadratic, and there will thus be a single extreme point. The developments for this direct solution are omitted here.

In a second step, the distance is computed at every $t \in \{t_0, \ldots, t_k\}$, as well as at every timestamp returned by the first step. These values are stored in a list of time-value pairs, sorted by increasing timestamp. Finally, the intermediate timestamps $t \in \{t_1, t_{k-1}\}$ that do not correspond to an extreme point are removed from the list. The extreme point condition can be checked by looking at the values of the previous, current, and next timestamp. The current point will then be extreme if its value is either smaller or larger than both other values. This last check is required in case the switch from one closest feature to another does not happen at an extreme point.

This algorithm results in a list of time-value pairs that stores a linear approximation of the distance between two moving bodies. This is also the MobilityDB representation of a `tfloat` value that does not contain temporal gaps. The temporal distance operator in MobilityDB will thus combine the algorithms described in Section 4 with the linear approximation algorithm to return a `tfloat` representing the temporal distance between its two operands.

6.2 Distance Operators

The previous section describes how we implement the `distance` operator in MobilityDB. Additionally, we also define three additional operators: `nearestApproachDistance`, `nearestApproachInstant`, and `shortestLine`, with the signature in Figure 15. All three of these operators are defined in the OGC Moving Features Access standard [20]. The types `tgeompoint` and `tgeometry` are the types representing moving points and moving polygons, respectively, in MobilityDB. For brevity, let's also use the notation $\mathcal{A}$ to denote the set of types `{tgeompoint, tgeometry, point, polygon}`.

NearestApproachDistance is an operator that, given two moving objects, returns the smallest distance ever obtained between them during their movement. This operator is important as it is used for nearest neighbor searches. Its implementation first computes the distance function using the *distance* operator and then finds the minimum value of the result using the MobilityDB *minValue* function. Indeed, as described in Section 6.1, the linear approximation of the distance maintains all the extreme points of the function. The *nearestApproachDistance* operator will thus return the correct value despite the linear approximation step.

The *nearestApproachInstant* operator is complementary to the *nearestApproachDistance* operator, as it computes the timestamp at which the nearest approach distance is obtained. If this happens more than once, the first timestamp is returned. Lastly, the *shortestLine* operator computes the shortest linestring between the two moving objects at the nearest approach instant.

7 EXPERIMENTAL VALIDATION

In this section, we evaluate the performance of the distance operator as it is described in Sections 4 and 6. The section is divided into two parts. First, we validate the analytical complexity of the algorithms described in Section 4. This is done on synthetic data, as this allows us to control the size and the movement parameters of the polygons. In a second step, we compare the existing MobilityDB distance function for moving points to the newly implemented distance operator for moving polygons. Since moving points have a more simple data model and distance function, we use them as a baseline to get insight into the performance of the proposed moving body distance function in practice. This second experimental section uses real-world AIS data.

7.1 Validation of Computational Complexity

In this section, we evaluate the complexity of the algorithms of Sections 4.2 and 4.3, as well as the speedup received by the optimized version when the polygons are non-rotating. The algorithms are coded in Python 3.6 and the experiments are repeated for both the point-to-polygon and the polygon-to-polygon problems. The code and explanations to replicate these results are available on GitHub.[1]

In the point-to-polygon case, a random convex polygon with n vertices is generated, using the algorithm in [29],[2] in the box $(x_{min}, y_{min}, x_{max}, y_{max}) = (0, 0, n, n)$, with a translation $(dx, dy) = (0, n)$ and a rotation θ. The random point is then generated in the box $(2n, n, 3n, 2n)$, with a translation $(0, -n)$, and the first algorithm is applied to these two moving objects. For the polygon-to-polygon problem, we apply the same generation technique but generate a second polygon with $m = n$ vertices in the box $(2n, n, 3n, 2n)$, with a translation $(0, -n)$ and the same rotation θ as the other polygon. In both cases, the rotation center of the polygons corresponds to their centroid. These starting and movement conditions allow us to vary the number of vertices n and the rotation θ of the polygons while making sure that the moving objects do not intersect. We vary the number of vertices n between 3 and 500 and the rotation θ between 0 and π.

Two elements of the algorithms can be analyzed: the size k of the result and the runtime t of the algorithm. First, we analyze the size of the result with respect to the parameters θ and n (remember that $m = n$ in the experiments). Figure 16 shows the graphs of k as a function of n (point-to-polygon problem) or $n + m$ (polygon-to-polygon problem), for varying values of θ. We can see that the size of the result is linear in the number of vertices but that the actual value of k also heavily depends on θ. The results are averaged over multiple runs with the same values for n and θ.

Second, we analyze the runtime t of the algorithms with respect to the result size k. This is done by running the algorithms for random values of n and θ and storing the size of the result and runtime of the algorithms as (k, t) pairs in a list. The list is then sorted by k and displayed on the graph. For this experiment, only the *While* loop of Algorithm 1 is timed. The computation of the initial closest feature in $O(log(n))$ is omitted, as it is done using existing algorithms. We thus expect the time t to be linear in k as detailed in Sections 4.2 and 4.3.

The results of these experiments can be seen in Figure 17. The blue and orange lines correspond to the point-to-polygon and the polygon-polygon cases, respectively, and Figures 17(a) and 17(b)

[1] https://github.com/mschoema/tgeometry_python
[2] https://cglab.ca/~sander/misc/ConvexGeneration/convex.html

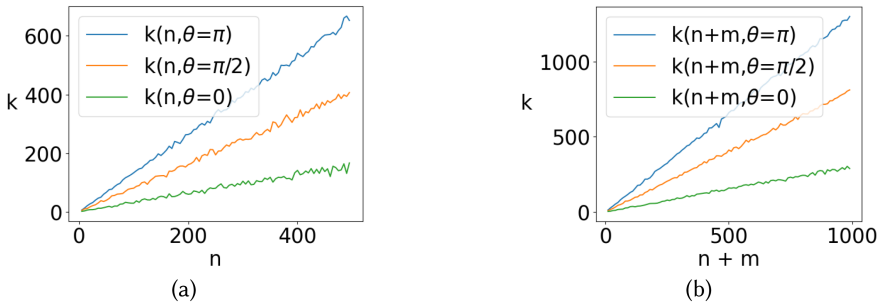


Fig. 16. Graphs of k in function of n (point-to-polygon, (a)) and n + m (polygon-to-polygon, (b)) for fixed value of θ.

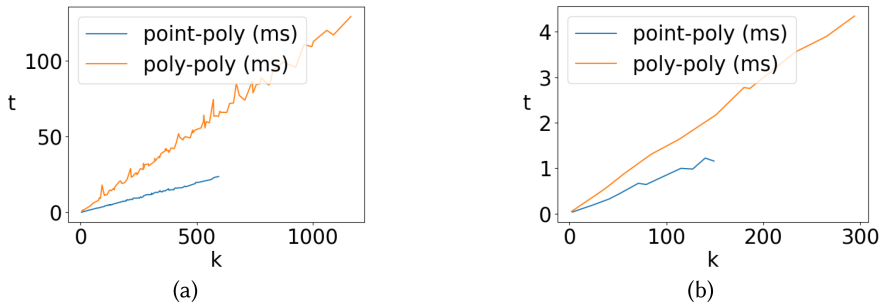


Fig. 17. Graphs of t (in ms) in function of k for the non-optimized ($\theta \neq 0$) (a) and optimized ($\theta = 0$) (b) algorithms.

show the result for the non-optimized ($\theta \neq 0$) and optimized ($\theta = 0$) algorithms, respectively. As expected, the durations of the different algorithms are all linear with respect to their result sizes ($t = O(k)$). The average durations per result (t/k) for the non-optimized algorithm are $t/k = 38\mu$s and 100μs for the point-to-polygon and polygon-to-polygon problems, respectively. The respective average durations per result for the optimized algorithms are $t/k = 7\mu$s and 13μs, respectively. The optimized algorithms are thus about 5 to 8 times faster than the non-optimized one for identical result sizes.

7.2 Danish AIS Use-case

With Section 7.1 confirming the computational complexities of the algorithms presented in Section 4, it remains to assess the running time of a distance query in a real-world use-case. For this, we test the MobilityDB implementation of the *distance* operator[3] on a real-world AIS dataset from the Danish Maritime Authority.[4] This dataset contains historic AIS data and is publicly distributed as CSV files each containing 1 full day of data. We use the CSV files from September 2020, which contain in total 250M AIS points in 40K ship tracks spread over 30 days, for a total of 61.6GB of raw data. The data is cleaned and loaded in MobilityDB in both moving point and moving polygon format. The moving point data is constructed using the timestamp, longitude, and latitude fields of the AIS data. To construct the moving polygons, we first construct the static geometry at each timestamp using the latitude, longitude, and heading and the fields `sizea`, `sizeb`, `sizec`, and

[3] https://github.com/mschoema/MobilityDB/tree/tgeometry
[4] https://dma.dk/safety-at-sea/navigational-information/ais-data

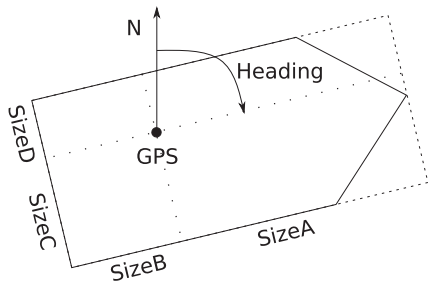
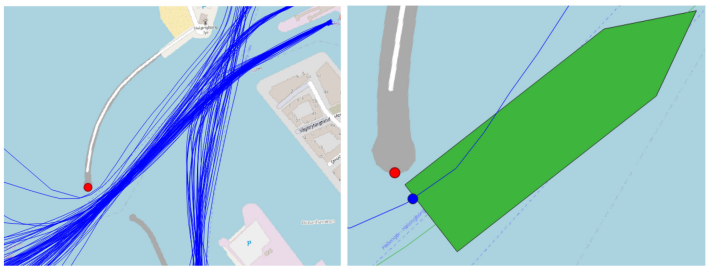


Fig. 18. Construction of the geometry of a vessel from its position, size, and heading information.



(a) (b)

Fig. 19. (a) Entrance to the port of Helsingborg, with the query point in red and the vessel trajectories in blue. (b) Visualization of the vessel with the closest approach in both the point (blue) and polygon (green) representation.

`sized`, giving the distances of the sides of the vessel to the GPS point at which the longitude and latitude are computed. The static geometry constructed from this data is a pentagon in the shape of a rectangle with an additional triangle on one side representing the front of the vessel. The construction of this polygon representation of a vessel can be seen in Figure 18. Since the position, size, and orientation information is extracted from the AIS data, we can assume that the moving polygon closely approximates the real movement and geometry of the vessel.

The experiments in this section thus correspond to the case where $n = 5$. The two constructed tables are the following:

`ships_point(mmsi integer, trip tgeompoint)`, and
`ships_poly(mmsi integer, trip tgeometry)`.

After this transformation into a MobilityDB data format, the table sizes are respectively 4.4GB and 5.5GB. We now compare the existing distance operator for moving points with the newly implemented operator for moving polygons.

As a first example use-case, let us analyze the entrance of the port of Helsingborg. For security reasons, when entering the port, the vessels should not come too close to the dyke at the entrance of the port. Query 1 thus computes the temporal distance between the vessels stored as moving points (1a) or polygons (1b) and a fixed point at the end of the dyke at the entrance of the Helsingborg port. Figure 19(a) shows the position of this point as well as the trajectories of the vessels around it. For demonstration purposes, the query is only run on the first day of data. The running time of Query 1a is 3.65 seconds (average of 10 runs), while the same query on moving bodies (1b) takes 5.06 seconds. Note that this is the total running time of applying the algorithm on the 6.5M segments that make up the movement of the vessels. The distance operator for moving polygons

Table 1. Running Time and Result of Query 2

Query	Running Time (s)	mmsi	min_dist (m)
2a	3.59	265610940	10.42
2b	4.94	265041000	5.21

is thus about 1.4 times slower than the operator for moving points, which is to be expected due to the increased complexity of the algorithm. Another thing to note here is that for the trips of table `ships_poly`, 65.7% of the segments have a rotation angle of 0. This indicates that the optimized solution for non-rotating polygons is heavily used in real-world use-cases.

```
Query 1a: SELECT distance(trip, 'Point(729493 6216766)')
    FROM ships_point;
Query 1b: SELECT distance(trip, 'Point(729493 6216766)')
    FROM ships_poly;
```

Let us also query for the vessel that came closest to the query point during its movement. Query 2 computes the nearest approach distance of every vessel to the query point and returns the vessel id (mmsi) and distance of the vessel with the smallest nearest approach distance. Again, this query is run with the vessels being represented as moving points (2a) and moving polygons (2b) for the first day of data. Table 1 shows the running time and results of both queries. Consistently with the results obtained in Query 1, the running times of Query 2 are similar since the processing of the `distance` operator takes the majority of the query time. Looking at the result values, we can see that the smallest distance is 10.42m when computed using moving points and 5.21m when computed using moving polygons, which illustrates the gain in precision. More important is that the vessels (identified by mmsi) returned by the queries are also different. This means that the impression introduced by computing distances using the moving point approximation led to returning a wrong vessel id in this query. This result is also visualized in Figure 19(b).

```
Query 2a: SELECT mmsi,
    nearestApproachDistance(trip,
        'Point(729493 6216766)') AS min_dist
    FROM ships_points
    ORDER BY min_dist LIMIT 1;
Query 2b: SELECT mmsi,
    nearestApproachDistance(trip,
        'Point(729493 6216766)') AS min_dist
    FROM ships_polys
    ORDER BY min_dist LIMIT 1;
```

In a second experiment, we demonstrate the scalability of the algorithms. As reference geometries, we generate a point and five convex polygons of varying sizes that we arbitrarily placed below the island of Læsø. Table 2 lists the generated geometries with their number of vertices, and Figure 20 shows four of these geometries on the map. Note that since we are working with convex polygons, the polygon with 75 vertices already has a smooth contour. We thus limit ourselves to polygons with at most 75 vertices. To link this experiment to a practical problem, the generated polygons can be seen as the extent of an offshore wind farm. European member states and, in particular, Denmark already possess many offshore wind farms[5] and are planning on adding more to

[5] www.4coffshore.com/windfarms/denmark/

Table 2. Sizes of the Generated Geometries

Geometry Name	Point A	Poly B	Poly C	Poly D	Poly E	Poly F
# of Vertices	1	5	10	25	50	75

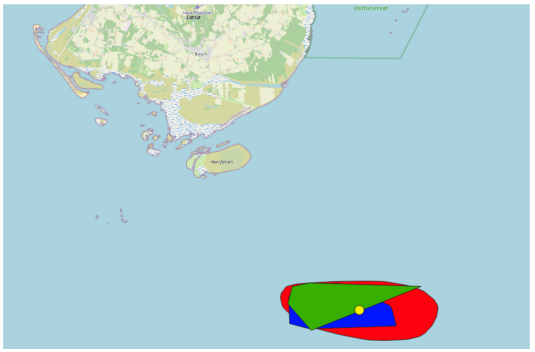


Fig. 20. Visualization of four generated geometries: Point A (yellow) and Poly B (green), C (Blue), and F (red).

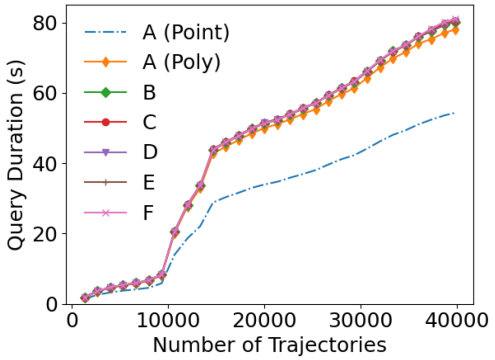


Fig. 21. Duration of temporal distance queries in function of the number of trajectories. Queries A (Poly) and B–F use the polygon representation of the vessels, while query A (Point) uses the point representation.

be able to meet ambitious goals related to renewable energy.[6,7] For security reasons, vessel traffic cannot come within a certain distance of these wind farms [25]. When planning a new project, it is thus important to know the distance between the planned wind farm and the surrounding vessel traffic.

In this experiment, we compute the temporal distance between the vessel trajectories in moving polygon representation and each of the generated reference geometries. Additionally, we also compute the temporal distance between the moving point representation of the trajectories and the first point geometry, since this is currently the only existing distance function for moving objects in MobilityDB. Figure 21 shows the query duration for data sizes varying from 1 day to a full month of data. For the full data size, computing the temporal distance between the moving polygons and a static geometry takes about 80 seconds.

The first thing we note in the figure is that computing the distance for the polygon representation of the vessels is at most 1.5 times slower than for the point representation. This is coherent

[6]energy.ec.europa.eu/news/member-states-agree-new-ambition-expanding-offshore-renewable-energy-2023-01-19_en.
[7]www.offshorewind.biz/2023/02/20/denmark-to-auction-off-9-gw-of-offshore-wind-in-2023/

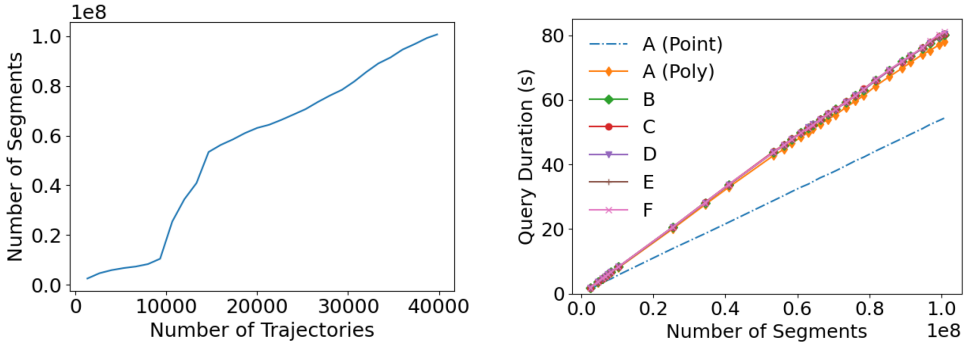


Fig. 22. (a) Relation between the number of trajectories and the number of segments. (b) Duration of temporal distance queries in function of the number of segments.

with the result from the previous experiment. Second, the polygon-to-point algorithm (A Poly) is ever so slightly faster than the polygon-to-polygon algorithm (B–F), which is expected since the polygon-to-polygon algorithm needs to solve more complex equations to determine changes in closest features. Lastly, the number of vertices of the second polygon does not visibly influence the duration of the query. Indeed, the duration of the algorithm is linked to the number of changes in closest features. For many trajectories that are relatively far away from the wind farm geometry, increasing the number of vertices of the geometry will not directly increase the number of changes in closest features during the movement.

The last remaining particularity of Figure 21 is the shape of the graph. Notice that the data size is given in terms of number of trajectories. Each trajectory represents a sequence of linear segments, as discussed at the end of Section 3. The distance algorithm is thus called subsequently on each individual segment. Querying the distance for trajectories with more segments would thus take longer than for trajectories with fewer segments. Looking at Figure 22(a), we can indeed see that the total number of segments does not increase linearly with the number of trajectories. This means that some trajectories contain more segments than others. Plotting now the query duration in function of the total number of segments (Figure 22(b)), we can see that the query duration is linear in terms of trajectory segments. The average duration per segment is 539ns, 775ns, and 800ns for the point-to-point, polygon-to-point, and polygon-to-polygon algorithms, respectively. The last value is computed as the average of the four polygon-to-polygon queries since they all have similar duration.

8 CONCLUSION

To conclude, we described the problem of computing the time-varying distance between a continuously moving body and other static and moving objects in 2D and presented algorithms to solve this problem efficiently. When the moving bodies are convex, the algorithm computes the evolution of their closest features in $O(log(n) + k)$ time, where n is the total number of vertices, and k is the number of times the closest features change (size of the result). This evolution is stored as a list of length k that, together with the initial moving bodies, completely determines the equation of the temporal distance between the moving objects. This list can thus be used to determine the distance at a given timestamp in $O(log(k))$ time.

Around this distance algorithm, multiple extensions have been described. First, we developed an optimization for the case where the movement contains no rotation. This case appeared to

be the most common one in the test we performed on a real dataset. Second, the algorithm was generalized for non-convex polygons. Finally, we described how this algorithm, together with an additional linear approximation step, is used to implement the distance operators specified in the OGC Moving Features Access standard in the moving object database MobilityDB.

The experiments confirm the linear complexity of the algorithm in terms of k and show a 5–8× speedup when using the optimized algorithms for non-rotating moving bodies. Additionally, we show that the increased complexity of representing vessels using moving polygons can result in increased precision during distance computations with only a 1.4x increase in computation time.

This article presents a solution for moving objects in 2D, but the idea of tracking the closest features between two objects can also be applied in 3D. In this case, the moving bodies are either 3D points or polyhedrons, and the features are vertices, edges, or faces of the objects. Future work would thus be to generalize this algorithm to compute the time-varying distance between 3D moving bodies.

ACKNOWLEDGMENTS

Maxime Schoemans is a Research Fellow of the Fonds de la Recherche Scientifique - FNRS.

REFERENCES

[1] Jean-Luc Chabert. 1999. *Newton's Methods*. Springer, Berlin, 169–197.
[2] Bernard Marie Chazelle. 1980. *Computational Geometry and Convexity*. Ph. D. Dissertation. Yale University.
[3] F. Chin and Cao An Wang. 1983. Optimal algorithms for the intersection and the minimum distance problems between planar polygons. *IEEE Transactions on Computers* 32 (1983), 1203–1207.
[4] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. 2003. Algorithms for moving objects databases. *Computer Journal* 46 (2003), 680–712.
[5] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry*. Springer-Verlag, Berlin.
[6] Angel P. Del Pobil, Miguel Angel Serna, and Juan Llovet. 1992. A new representation for collision avoidance and detection. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*. 246–247.
[7] H. Edelsbrunner. 1985. Computing the extreme distances between two convex polygons. *Journal of Algorithms* 6 (1985), 213–224.
[8] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. 2000. A data model and data structures for moving objects databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. 319–330.
[9] Yunjun Gao, Chun Li, Gencai Chen, Qing Li, and Chun Chen. 2007. Efficient algorithms for historical continuous k nn query processing over moving object trajectories. In *Proceedings of the Advances in Data and Web Management: Joint 9th Asia-Pacific Web Conference (APWeb'07), and 8th International Conference on Web-Age Information Management (WAIM'07)*. 188–199.
[10] Leonidas J. Guibas, David Hsu, and Li Zhang. 2000. A hierarchical method for real-time distance computation among moving convex bodies. *Computational Geometry: Theory and Applications* 15 (2000), 51–68.
[11] Ralf Hartmut Güting, Thomas Behr, and Jianqiu Xu. 2010. Efficient k-nearest neighbor search on moving object trajectories. *The VLDB Journal* 19 (2010), 687–714.
[12] Florian Heinz and Ralf Güting. 2018. A data model for moving regions of fixed shape in databases. *International Journal of Geographical Information Science* 32 (2018), 1737–1769.
[13] Stefan Hertel and Kurt Mehlhorn. 1983. Fast triangulation of simple polygons. In *Foundations of Computation Theory*. 207–218.
[14] Philip M. Hubbard. 1993. Interactive collision detection. In *Proceedings of 1993 IEEE Research Properties in Virtual Reality Symposium*. 24–31.
[15] J. Mark Keil. 1985. Decomposing a polygon into simpler components. *SIAM Journal on Computing* 14 (1985), 799–19.
[16] Ming C. Lin and John F. Canny. 1991. A fast algorithm for incremental distance calculation. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*. 1008–1014.
[17] Ming C. Lin, Dinesh Manocha, Jon Cohen, and Stefan Gottschalk. 1997. Collision detection: Algorithms and applications. In *Algorithms for Robotic Motion and Manipulation*. A K Peters/CRC Press, 129–142. https://www.taylorfrancis.com/books/mono/10.1201/9781439864524/algorithms-robotic-motion-manipulation-jean-paul-laumond-mark-overmars

[18] Ming C. Lin, Dinesh Manocha, and Young J. Kim. 2017. *Collision and Proximity Queries*. Chapman and Hall/CRC, 1029–1056.
[19] Brian Mirtich. 1998. V-Clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics* 17 (1998), 177–208.
[20] OGC Open Geospatial Consortium. 2016. *OGC Moving Features Access*. http://docs.opengeospatial.org/is/16-120r3/16-120r3.html
[21] I. F. D. Oliveira and R. H. C. Takahashi. 2020. An enhancement of the bisection method average performance preserving minmax optimality. *ACM Transactions on Mathematical Software* 47 (2020), 1–24.
[22] Jae Sung Park, Chonhyon Park, and Dinesh Manocha. 2017. Efficient probabilistic collision detection for non-convex shapes. In *2017 IEEE International Conference on Robotics and Automation (ICRA'17)*. 1944–1951.
[23] Madhav Ponamgi, Dinesh Manocha, and Ming C. Lin. 1995. Incremental algorithms for collision detection between solid models. In *Proceedings of the 3rd ACM Symposium on Solid Modeling and Applications*. 293–304.
[24] Sean Quinlan. 1994. Efficient distance computation between non-convex objects. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*. 3324–3329.
[25] Andrew Rawson and Edward Rogers. 2015. Assessing the impacts to vessel traffic from offshore wind farms in the Thames Estuary. *Zeszyty Naukowe Akademii Morskiej w Szczecinie* 43 (2015), 99–107.
[26] Yuichi Sato, Mitsunori Hirata, Tsugito Maruyama, and Yuichi Arita. 1996. Efficient collision detection using fast distance-calculation algorithms for convex and non-convex objects. In *Proceedings of IEEE International Conference on Robotics and Automation*. 771–778.
[27] Maxime Schoemans, Mahmoud Sakr, and Esteban Zimányi. 2021. Implementing rigid temporal geometries in moving object databases. In *Proceedings of the 37th IEEE International Conference on Data Engineering*. 12.
[28] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. 2002. Continuous nearest neighbor search. In *VLDB*. 287–298.
[29] Pavel Valtr. 1995. Probability that n random points are in convex position. *Discrete & Computational Geometry* 13 (1995), 637–643.
[30] Cheng-lei Yang, Meng Qi, Xiangxu Meng, Xue-qing Li, and Jia-ye Wang. 2006. New fast algorithm for computing the distance between two disjoint convex polygons based on Voronoi diagram. *Journal of Zhejiang University Science* 7 (2006), 1522–1529.
[31] Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. 2020. MobilityDB: A mobility database based on PostgreSQL and PostGIS. *ACM Transactions on Database Systems* 45 (2020), 42.

Received 12 September 2022; revised 30 April 2023; accepted 30 June 2023