Scientific Annals of Computer Science vol. 29 (2), 2019, pp. 185–201
doi: 10.7561/SACS.2019.2.185

# Identifying and Querying Regularly Visited Places

Ali Gholami Rudi[1]

**Abstract**

A stay point of a moving entity is a region in which it spends a significant amount of time. In this paper, we identify all stay points of an entity in a certain time interval, where the entity is allowed to leave the region but it should return within a given time limit. This definition of stay points seems more natural in many applications of trajectory analysis than those that do not limit the time of entity's absence from the region. We present an $O(n \log n)$ algorithm for trajectories in $R^1$ with $n$ vertices and a $(1+\epsilon)$-approximation algorithm for trajectories in $R^2$ to identify all such stay points. Our algorithm runs in $O(kn^2)$, where $k$ depends on $\epsilon$ and the ratio of the duration of the trajectory to the allowed gap time. We also present an algorithm to answer stay point queries in logarithmic time, after an $O(kn \log n)$ time preprocessing.

**Keywords:** Trajectory analysis, stay points, regular visits, geometric algorithm.

## 1 Introduction

The question, asking where a moving entity, like an animal or a vehicle, spends a significant amount of its time is very common in trajectory analysis [14]. These regions are usually called popular places, hotspots, interesting places, stops, or stay points in the literature. There are several definitions of stay

This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License

[1]Department of Electrical and Computer Engineering, Babol Noshirvani University of Technology, Babol, Mazandaran, Iran, Email: gholamirudi@nit.ac.ir

points and different techniques have been presented to find them [1, 3, 5, 8, 9, 10, 12, 13]. However, from a geometric perspective, which is the focus of the present paper, few papers are dedicated to a formal algorithmic study of this problem.

To analyse the movement of entities, it should be accurately represented. Usually, the location of an entity is recorded at certain points in time. The sequence of these locations, paired with the time at which the entity was observed at each location, describes the entity's path; this is usually called its trajectory. To find stay points, we require one or more such trajectories. In the rest of this section we review some of the available definitions of stay points, mostly for $R^2$, and the algorithms presented for finding them.

Benkert et al. [3] defined a stay point as an axis-aligned square of fixed side length in the plane, which is visited by the most number of distinct trajectories. They modelled a visit either as the inclusion of a trajectory vertex or the inclusion of any portion of a trajectory edge, and presented optimal algorithms for both cases. Gudmundsson et al. [9] introduced several different definitions of trajectory stay points. Unlike Benkert et al., Gudmundsson et al. considered the duration of visits for finding stay points. Their stay points are axis-aligned squares that contain a sub-trajectory with the maximum duration. The sub-trajectory can be either continuous (the entity has spent some time inside the region without leaving it) or non-continuous (where the entity is allowed to leave the region and return later). For stay points of fixed side length, they presented an $O(n \log n)$ time algorithm for the former, and an algorithm with the time complexity $O(n^2)$ for the latter, where $n$ is the number of trajectory vertices. For non-continuous presence and for orthogonal trajectories, in which the entity moves parallel to one of the axes of the coordinate system, Rudi [12] presented a 1/2-approximation algorithm with the time complexity $O(n \log^3 n)$.

However, in many applications that require spatio-temporal analysis, we are interested in finding regions that are never left for a long time. Examples include bird nests, animal resting place, player posts in sports, and bus stations. Arboleda et al. [1] studied a problem that takes as input, in addition to the trajectories, a set of polygons as potential stay points or interesting sites. They presented a simple algorithm to identify stay points among the given interesting sites; their algorithm computes the longest sub-trajectory visiting each interesting site, while allowing the entity to leave the site for some predefined amount of time. They also mentioned motivating real world examples to show that in some applications, it makes

sense to allow the entity to leave the site for short periods of time, like leaving a cinema for the bathroom. Also, Djordjevic et al. [6, 7] introduced the problem of checking if a region is visited almost regularly (in fixed periods of time) by an entity, and presented an algorithm to solve it.

Our goal is identifying all trajectory stay points, i.e. axis-aligned squares in which the entity is always present, except for short periods of time, where both the side length of the squares and the allowed gap time are specified as parameters and assumed to be fixed. Note that we ignore the duration in which the entity stays in a region. If, for instance, a region with the maximum duration among our stay points is desired, our algorithm can be combined with those that find a stay point with the maximum duration, but allow unbounded entity absence, like the ones presented by Gudmundsson et al. [9]. A preliminary version of this paper appeared in CCCG 2018 [11].

This paper is organized as follows. In Section 2, we introduce the notation and define some of the main concepts of this paper. In Section 3, we handle trajectories in $R^1$ and present an algorithm to find all stay points of such trajectories with the time complexity $O(n \log n)$. We focus on trajectories in $R^2$ in Section 4 and present an approximation algorithm for finding their stay points. We also show that the complexity of the stay map of two-dimensional trajectories can be $\Theta(n^2)$. In Section 5, we answer stay point queries in logarithmic time, and finally in Section 6, we conclude this paper.

## 2 Preliminaries

A trajectory $T$ describes the movement of an entity in a certain time interval. Trajectories can be modelled as a sequence of vertices and edges in the plane. Each vertex of $T$ represents a location at which the entity was observed. The time of this observation is indicated as the timestamp of the vertex. We assume that the entity moves in a straight line and with constant speed from a vertex to the next; the edges of the trajectory connect its contiguous vertices.

A sub-trajectory of $T$ for a time interval $(a, b)$ is denoted as $T(a, b)$, and describes the movement of the entity from time $a$ to time $b$. Except possibly the first and the last vertices of a sub-trajectory, which may fall on an edge of $T$, its set of vertices is a subset of those of $T$. The stay points considered in this paper are formally described in Definition 1. We use the symbols defined here, such as $g$ and $s$, throughout the paper without

repeating their description. Also, any square that appears in the rest of this paper is axis-aligned and has side length $s$.

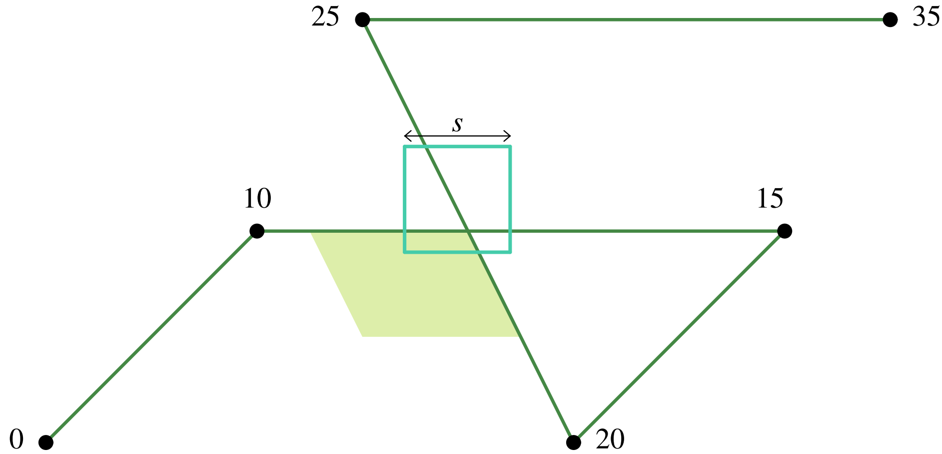


Figure 1: An example two-dimensional trajectory. The number near each vertex shows its timestamp. The green region is the stay map and the green square is a stay point ($g = 15$).

**Definition 1** *A stay point of a trajectory $T$ in $R^2$ is a square of fixed side length $s$ in the plane such that the entity never spends more than a given time limit $g$ outside it continuously.*

The goal of this paper is identifying all stay points of a trajectory, or its stay map (Definition 2). Note that the parameters $s$ and $g$ are assumed to be fixed and specified as inputs of the algorithm.

**Definition 2** *The stay map $M$ of a trajectory $T$ in $R^2$ is a subset of the plane such that every square of side length $s$ whose lower left corner is in $M$ is a stay point of $T$, and the lower left corners of all stay points of $T$ are in $M$.*

Figure 1 shows an example trajectory, its stay map, and one of its stay points. Note that every square, whose lower left corner is in the stay map, is a stay point. Although these definitions are presented for trajectories in $R^2$, they can be trivially adapted for one-dimensional trajectories, as we do in Section 3.

## 3 Stay Maps of One-Dimensional Trajectories

Let $T$ be a trajectory in $R^1$. A stay point of $T$ is an interval of length $s$ such that the entity never leaves it for a period of time longer than $g$. The stay map $M$ of $T$ is the region containing the left end points of all stay points of $T$. In this section, we present an algorithm for finding $M$.

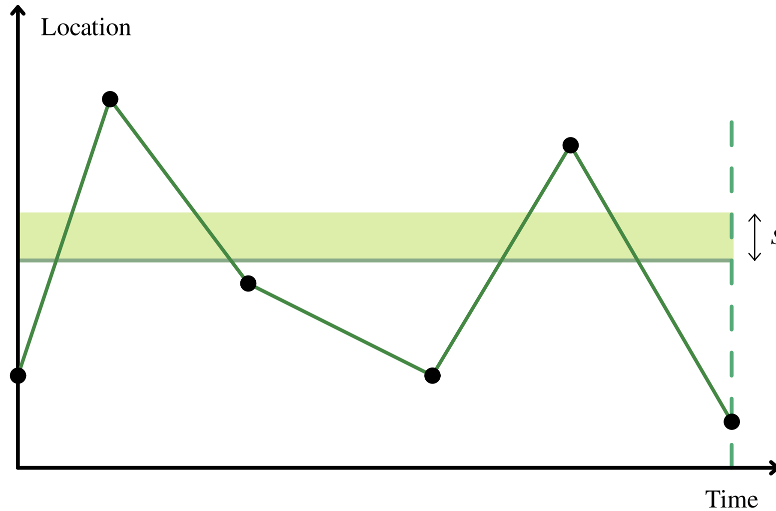


Figure 2: Mapping a one-dimensional trajectory to the time-location plane. The green rectangle of height $s$ shows a possible stay point.

**Lemma 1** *The stay map $M$ of a trajectory $T$ in $R^1$ is continuous.*

**Proof:** To obtain a contradiction, let points $p$ and $q$ be inside $M$ and $v$ be outside it such that $p < v < q$ (our assumption that $M$ is non-continuous implies the existence of this triple). Let $r_p$, $r_q$, and $r_v$ be three segments of length $s$, whose left corners are at $p$, $q$, and $v$, respectively. Clearly, $r_p$ and $r_q$ are stay points while $r_v$ is not. Whenever the entity moves to the left of $v$, it must return to $q$ before the time limit $g$ to visit $r_q$. Also, whenever the entity moves beyond the right end point of $r_v$ (which is outside $r_p$), it must return to $r_p$ before the time limit. Therefore, it can never be outside $r_v$ for more than time $g$ and this implies that $v$ is also a stay point and inside $M$, which yields the desired contradiction. □

**Lemma 2** *Given a trajectory $T$ with $n$ vertices in $R^1$, we can answer in $O(n)$ time whether a point $p$ is in the stay map or not, and if not, whether the stay map is on its left side or on its right side.*

**Proof:** Define $r$ as segment $pq$, in which $q$ is $p+s$. Testing each trajectory edge in order, we can compute the duration of each maximal sub-trajectory outside $r$ and check if it is at most $g$. Therefore, we can decide if $p$ is the left end point of a stay point in $O(n)$ time. If it is not a stay point, there is at least one time interval, in which the entity spends more than time $g$ on the left or on the right side of $r$. Without loss of generality, suppose it does so on the left side. Then, no point on the right of $r$ can be a stay point and therefore the whole stay map of $T$ must appear on the left of $p$. This again can be tested in $O(n)$ time by processing trajectory edges. □

To compute the stay map of a trajectory, we need its set of event points (Definition 3).

**Definition 3** *An event point of a trajectory $T$ in $R^1$ is a point on the line in which one of the following occurs: i) a trajectory vertex lies on that point, ii) the time gap between two contiguous visits to that point is exactly $g$.*

**Lemma 3** *The stay map $M$ of a trajectory $T$ starts and ends at an event point or at distance $s$ from one.*

**Proof:** By Lemma 1, $M$ is continuous. Let $p$ be the left end point of the stay map $M$. Let $r = pq$ be a segment such that $q = p + s$. Whenever the entity leaves $r$ through $p$, it returns by passing it again within the time limit $g$. Similarly, if the entity leaves $r$ through $q$, it visits $q$ again within time $g$. Suppose, for the sake of contradiction, that $p$ is not an event point. Then, we can move $r$ slightly to the left to obtain $r'$. $r'$ must also be a stay point because every time the entity leaves it from either of its end points, it returns within time $g$, because neither $p$ nor $q$ is an event point (the time between the contiguous visits of the entity is not exactly $g$ and they are not on a trajectory vertex). This contradicts the choice of $p$. A similar argument shows that the right endpoint of $M$ must also be an event point or at distance $s$ from one. □

**Lemma 4** *The set of event points of a trajectory with $n$ vertices can be computed in $O(n \log n)$ time.*

**Proof:** We map the trajectory to the time-location plane such that a trajectory vertex at position $p$ with timestamp $t$ is mapped to point $(t, p)$ (see Figure 2). Obviously, the polygonal path representing the trajectory in this plane is $y$-monotone. We perform a plane sweep by sweeping a line parallel to the $x$-axis in the positive direction of the $y$-axis in this plane. This is demonstrated in Figure 2.

The edges in this plane chop the sweep line into several segments (for instance, in Figure 2 the trajectory splits the horizontal line into five segments). We maintain the length of every such segment during the sweep line algorithm. When the sweep line intersects a trajectory vertex $v$, an event point is recorded and, based on the other end point of the edges that meet at that vertex, one of the following cases occurs (demonstrated in Figure 3).

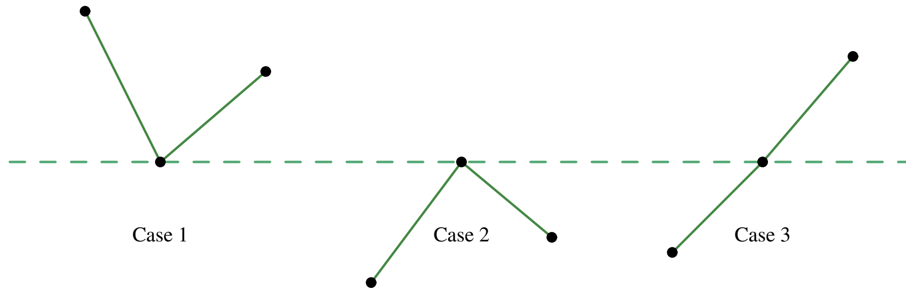


Figure 3: Different cases of sweep line intersecting a vertex in the algorithm on the proof of Lemma 4.

1. If $v$ is the lowest end point of both edges, two new segments are introduced. Based on the slope of the edges bounding each segment, we record an event at which the distance between the edges is exactly $g$, if they are long enough.

2. If $v$ is the highest end point of both edges that meet at $v$, three segments on the sweep line are merged (when the sweep line is before $v$, three segments are created by the edges incident to $v$, at $v$, there are two such segments, and after $v$, they merge into one). We also record an event for the location at which the length of the remaining segment becomes $g$ in the plane.

3. If $v$ is the highest end point of one edge and the lowest end point of another, the event scheduled for the location at which the length of

each of the two incident segments on the sweep line are $g$ may need to be updated.

Note that since the sweep line stops at every vertex and at each vertex only a constant number of event points are added, the total number of event points is $O(n)$. Therefore, these events can be generated in order and handled with the time complexity $O(n \log n)$. □

**Theorem 1** *The stay map $M$ of a trajectory $T$ with $n$ vertices in $R^1$ can be computed in $O(n \log n)$ time.*

**Proof:** Lemma 4 implies that the set of event points of $T$ can be computed with the time complexity $O(n \log n)$. From this set, we can obtain an ordered sequence of event points and points at distance exactly $s$ from them in $O(n \log n)$ time (note that the length of this sequence is still $O(n)$). Based on Lemma 3, $M$ starts and ends at a point of this sequence. Also, Lemma 2 implies that we can decide if any of the end points of $M$ appears before or after any point in $O(n)$ time. Therefore, we can perform a binary search on the sequence obtained from the event points of $T$ to find the left and the right end points of $M$. Since the length of the sequence is $O(n)$, the time complexity of the binary search is $O(n \log n)$. □

Unfortunately, this algorithm cannot be adapted for two-dimensional trajectories, because their stay maps may no longer be continuous.

## 4 Stay Maps of Two-Dimensional Trajectories

We use the notation $P(a, b)$ to denote the region that contains the lower left corners of all squares of side length $s$ that contain at least one point of the sub-trajectory $T(a, b)$. We also use $M(a, b)$ to indicate the stay map of the sub-trajectory $T(a, b)$. We assume that trajectory $T$ starts at time 0 and has total duration $D$. It is clear that every point in the stay map of $T$ must appear in $P(t, t + g)$ for any value of $t$, where $0 \leq t \leq D - g$ (because the entity cannot be outside a stay point of $T$ for more than time $g$). Therefore, the stay map of $T$ is the intersection of $P(t, t+g)$ for every possible value of $t$, $0 \leq t \leq D-g$. This suggests the general scheme demonstrated in Algorithm 1 for finding the stay map of a two-dimensional trajectory, assuming $D > g$.

**Algorithm 1** *Let $T$ be two-dimensional trajectory with $n$ edges and total duration $D$. Compute the stay map of $T$ ($M(0, D)$) as follows.*

1. *Compute $P(0, g)$, as the union of polygons $P(u, v)$, for all edges $uv$ in $T(0, g)$.*

2. *Let $M(0, g)$ be $P(0, g)$. This is not strictly correct as $M(0, t)$ must include the complete plane when $t \leq g$ and its value changes to a subset of $T(0, g)$ for any value of $t > g$. This simplifying assumption, however, does not affect the correctness of the algorithm, since $D > g$.*

3. *Incrementally compute $M(0, D)$ as follows. Suppose $M(0, a)$ is computed in previous steps of the algorithm. Let $b$ be the smallest value after $a$, such that $b - g$ or $b$ is the timestamp of a trajectory vertex. Compute $M(0, b)$ from $M(0, a)$ (note again that $M(0, b)$ is a subset of $M(0, a)$). Let $V$ be the difference between $M(0, a)$ and $M(0, b)$. After computing $V$, we obtain $M(0, b)$ by excluding $V$ from $M(0, a)$.*

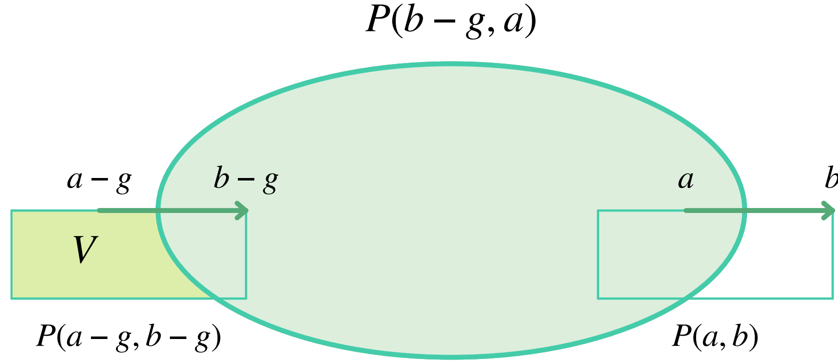


Figure 4: The difference $V$ in Algorithm 1, when $P(a - g, b - g)$ and $P(a, b)$ do not overlap.

The core of Algorithm 1 is the computation of the difference $V$. By the choice of $b$, $T(a - g, b - g)$ and $T(a, b)$ are both line segments. The value of $V$ depends on these segments and $T(b - g, a)$.

Let $r$ be a square, whose lower left corner is in $V$ and let $a - g + \delta$ be the time of entity's departure from $r$ before time $b - g$. Since the lower left corner of $r$ is in $V$, $r$ is not visited by the entity in the sub-trajectory $T(a - g + \delta, a + \delta)$. In other words, any point not in $P(a - g + \delta, b - g)$, $P(b - g, a)$, and $P(a, a + \delta)$ for any value of $\delta$ in $0 \leq \delta \leq g$ cannot be a stay point.

To make the computation of $V$ easier, we define $V'$ as follows ($V'$ is very similar to $V$, except that it ignores $P(b - g, a)$):

$$V' = \bigcup_{0 \le \delta \le g} P(a-g, a-g+\delta) \setminus (P(a-g+\delta, b-g) \cup P(a, a+\delta))$$

$V'$ contains the lower left corners of all squares that have been visited during the interval $(a-g, a-g+\delta)$, but have not been visited in $(a-g+\delta, b-g)$ or $(a, a+\delta)$ for some $\delta$ in $0 \le \delta \le g$. Then, $V = V' \setminus P(b-g, a)$.

If no square intersects both $T(a-g, b-g)$ and $T(a, b)$, $V'$ is $P(a-g, b-g)$. This case is shown in Figure 4, in which $V'$ is the rectangle on the left. Otherwise, $V'$ depends on the relative speed of the entity in these sub-trajectories. In both cases, $V'$ is a polygon of constant complexity and can be computed in constant time. We do not discuss the details of the computation of $V'$ in this paper, however. Since $T(b-g, a)$ consists of $O(n)$ edges, $P(b-g, a)$ is the union of $O(n)$ simple polygons. Therefore, $V' \setminus P(b-g, a)$ is also the union of a set of polygons with the total complexity $O(n)$. Let $V_t$ be the union of the differences $V$ for all iterations of the third step of Algorithm 1 (note that the complexity of $V_t$ is $O(n^2)$). When the algorithm finishes, $M(0, D)$ is $P(0, g) \setminus V_t$. Since the computation of $V_t$ requires finding the union of polygons with total complexity $O(n^2)$, an $O(n^2)$ implementation of this exact algorithm seems unlikely.

## 4.1 Approximate Stay Maps of Two-Dimensional Trajectories

In Algorithm 2, we consider $P(t, t+g)$ for limited discrete values of $t$ to compute *approximate stay maps* of a trajectory (Definitions 4 and 5), to improve the time complexity of Algorithm 1.

**Definition 4** *A $(1+\epsilon)$-approximate stay point of a trajectory $T$ in $R^2$ is a square of fixed side length $s$, such that the entity is never outside it for more than $g + \epsilon g$ time.*

**Definition 5** *A $(1+\epsilon)$-approximate stay map of a trajectory $T$ in $R^2$ is the region containing the lower left corners of all exact stay points of $T$ and possibly the lower left corners of some of its $(1+\epsilon)$-approximate stay points.*

**Algorithm 2** *Let $T$ be a trajectory in $R^2$ with $n$ edges and total duration $D$ and let $\epsilon$ be any real positive constant no greater than $D/g$. Compute a $(1+\epsilon)$-approximate stay map of $T$ as follows.*

1. *Compute $P(t, t+g)$ for $t = i\lambda$ for integral values of $i$ from 0 to $D/\lambda$, where $\lambda$ is $\epsilon g$. We call $P(t, t+g)$ for any value of $t$ a* snapshot *of $T$.*

2. *Compute the intersection of these snapshots. For this, we can use the topological sweep of Chazelle and Edelsbrunner [4] on the subdivision of the plane induced by the edges of the snapshots and include in the output the regions present in all snapshots.*

**Theorem 2** *For trajectory $T$ in $R^2$ with $n$ edges and total duration $D$ and any real positive constant $\epsilon$ no greater than $D/g$, Algorithm 2 computes a $(1+\epsilon)$-approximate stay map of $T$.*

**Proof:** Since the output of Algorithm 2 is the intersection of different snapshots of $T$, the lower left corner of every stay point must be inside it. Therefore, it suffices to show that every point in the output of the algorithm is the lower left corner of a $(1+\epsilon)$-approximate stay point.

Let $r$ be a square whose lower left corner is in the region reported by this algorithm. Suppose that the entity leaves $r$ at $t_b$ and reenters $r$ at $t_e$. We can set $t_b = 0$ for handling the initial part of the trajectory, and, if the entity never returns to $r$, we can set $t_e = D$. To prove the approximation factor, we show that $t_e \leq t_b + g + \epsilon g$. Let $i$ be the largest index such that $\lambda i \leq t_b$ and let $t_1 = \lambda i$. We show that the entity must return before time $t_1 + \lambda + \lambda/\epsilon$. Otherwise, $P(t_1 + \lambda, t_1 + \lambda + \lambda/\epsilon)$, which is a snapshot since $\lambda/\epsilon$ is equal to $g$, does not contain the lower left corner of $r$ (this is demonstrated in Figure 5) and this contradicts the assumption that it is included in the region returned by the algorithm. Therefore, the entity cannot be outside $r$ for longer than $\lambda/\epsilon + \lambda$, and $t_e \leq t_b + g + \epsilon g$. □

Figure 5: The entity leaves a square at $t_b$ and returns at $t_e$. If $t_e - t_b$ is larger than $g + g\epsilon$, there is a snapshot in which the entity is outside the square.

**Theorem 3** *The time complexity of Algorithm 2 is $O(n^2/\epsilon^2 + \sigma^2/\epsilon^2)$, in which $\sigma$ is $D/g$.*

**Proof:** $m$ line segments divide the plane into $O(m^2)$ regions (or faces). This subdivision of the plane can be can be swept with the time complexity $O(m^2)$ [4]. Moreover, the number of the segments of each snapshot

depends on the number of vertices of the sub-trajectory inside that snapshot (the region containing the lower left corners of the squares that intersect an edge of the sub-trajectory is a polygon with a constant number of sides). We, therefore, count the total number of vertices of the sub-trajectories in all snapshots. There are two types of trajectory vertices in each snapshot: those present in the original trajectory $T$ and the end points of the snapshot, which may not coincide with a trajectory vertex. Since the duration of each snapshot is $g$ and the difference between the start time of contiguous snapshots is $\epsilon g$, each trajectory vertex appears in at most $1/\epsilon$ snapshots. Therefore, the total number of vertices is at most $n/\epsilon + 2D/(\epsilon g)$ and the time complexity of Algorithm 2 is $O(n^2/\epsilon^2 + \sigma^2/\epsilon^2)$. □

It is not difficult to see that the stay map of a two-dimensional trajectory may contain $\Theta(n^2)$ faces and therefore we cannot expect an algorithm with the worst-case time complexity $o(n^2)$.

**Theorem 4** *The stay map of a trajectory in $R^2$ may contain $\Theta(n^2)$ faces.*

**Proof:** In what follows, we demonstrate a trajectory with $O(n)$ edges and a stay map of $\Theta(n^2)$ faces. Trajectory edges are added incrementally, as demonstrated in Figure 6, in which filled regions represent the stay map (except for $t \leq g$, in which they represent $P(0, t)$) and arrows show trajectory edges. We assume that the entity starts at time 0 and position $(0, 0)$.

1. Generate $m$ vertical strips as follows. Add the second vertex at $(2s, 0)$ with timestamp $g/2$ (Figure 6.a). Move the entity to its initial position using three vertices as shown in Figure 6.b; the position of the last vertex is $(0, 0)$ and its timestamp is $g - g/2m$. Create the vertical strips as follows: after every $g/m$ time, quickly move the entity by $s/m$ to the right (Figures 6.c–6.e). After $m$ such steps and waiting for at least $g$, the current stay map consists of $m$ vertical strips (Figure 6.f). Note that the excluded regions from the stay map are those regions that are not visited within the last interval of duration $g$.

2. The same trajectory we used for creating vertical strip can be used for creating horizontal strips after rotating the trajectory 90 degrees. If this is performed after the previous step, however, this would result in a stay map (Figure 6.g), which consists of $\Theta(m^2)$ small squares.

□

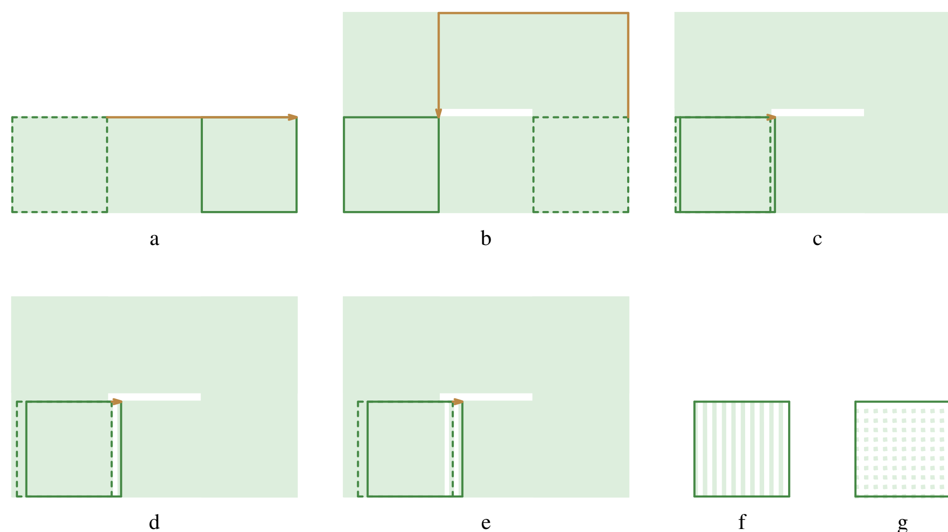


Figure 6: A trajectory with a stay map of $O(n^2)$ faces. The arrows indicate trajectory edges and filled regions indicate the stay map at each step.

## 5 Answering Stay Point Queries

In this section, we present an algorithm to answer stay point queries efficiently. After preprocessing a trajectory, each query asks if a point is the lower left corner of a stay point or not. In this section, we use the notation introduced in Section 4. We also assume that the edges of a sub-trajectory $T(a, a+g)$, for $0 \leq a \leq D-g$, intersect themselves $O(n)$ times (but note that this may not be true for $T(a, a+g')$, for any $g' > g$).

One way of answering such queries is computing and storing the (approximate) stay map of the input trajectory, and then using a point location data structure to answer each query in logarithmic time. However, the quadratic time complexity of preprocessing and quadratic space complexity of the algorithm makes it difficult to use in practice. The following algorithm is more efficient in this respect.

Let $\epsilon > 0$ be a real. To answer if a square is an $\epsilon$-approximate stay point or not, in the preprocessing step we store a point location data structure $L(t, t+g)$ for each snapshot $P(t, t+g)$, for $t = i\lambda$ for integral values of $i$ from 0 to $D/\lambda$, in which $\lambda = \epsilon g$. Each face of $P(t, t+g)$ contains the lower left corner of the squares visited by the same set of trajectory edges. We augment these data structures to store the last edge visiting the

squares corresponding to each face of $P(t, t+g)$. Algorithm 3 uses these data structures to answer stay point queries.

**Algorithm 3** *Let point $p$ be a stay point query. Find out whether $p$ is the lower left corner of a $(1+\epsilon)$-approximate stay point of trajectory $T$ or not.*

1. *Initialise $t$ to be 0.*
2. *Use $L(t, t+g)$ to find out $t'$, the time of the last visit of the sub-trajectory $T(t, t+g)$ to the square whose lower left corner is at $p$. If the sub-trajectory does not visit the square, $p$ is not the lower left corner of an approximate stay point.*
3. *Change the value of $t$ to $i\lambda$, in which $i$ is the smallest index such that $i\lambda > t' + g$.*
4. *If $t'$ is at least $D - 2g$, $p$ is an approximate stay point and we are done. Otherwise, repeat from step 2.*

**Theorem 5** *A trajectory with $n$ vertices can be preprocessed in time $O(\frac{\sigma}{\epsilon} n \log n)$, to answer stay point queries with the time complexity $O(\sigma \log n)$, in which $\sigma = D/g$.*

**Proof:** $L(t, t+g)$ for each snapshot contains $O(n)$ vertices and faces and can be initialized with the time complexity $O(n \log n)$. Since the total number of snapshots is $O(\sigma/\epsilon)$, the time complexity of initialising point location data structures is $O(\frac{\sigma}{\epsilon} n \log n)$.

To show the correctness of Algorithm 3, first note that $p$ is an $\epsilon$-approximate stay point, if it is in $P(t, t+g)$ for every snapshot. The algorithm, however, does not verify this for every snapshot. If a point $p$ is visited at time $t$, it is safe to skip all snapshots from time $t$ to $t+g$, because they are certainly present in all of them. We now obtain an upper bound on the number of times point location data structures are used for each query in Algorithm 3. Since the value of $t$ is increased at least by $g$ in each iteration of steps 2–4, step 3 is performed at most $D/g$ times. Therefore the time complexity of Algorithm 3 is $O(\sigma \log n)$. □

Alternatively, we can store a segment Voronoi diagram for the edges of each trajectory [2]. For each query, we can find the trajectory edge closest to the query point and see if this edge is close enough to the point or not. Also, by storing a Voronoi diagram for each sub-trajectory of each snapshot,

with $2^i$ vertices ($0 \leq i \leq \log n$), we can reduce the number of times point location queries should be made in Voronoi diagrams for each stay point query, as in Algorithm 3.

## 6 Concluding Remarks

In this paper we presented algorithms to find regions in which the entity is always present, except for short periods of time. We presented an exact algorithm for $R^1$, an approximation algorithm for $R^2$, and an approximation algorithm to verify if any given point is the lower left corner of a stay point or not in logarithmic time.

Our definition of stay points can be easily extended to multiple trajectories. Given a set of trajectories instead of only one, a multi-trajectory stay point is a square that is visited by at least one of the entities in any interval of duration $g$. It seems possible to compute such stay maps, by modifying Algorithm 2 to compute the intersection of the union of the snapshots of different entities. However, the time complexity of this algorithm may no longer be $O(n^2)$, where $n$ is the total number of trajectory vertices. Finding an efficient exact algorithm for the multi-trajectory version of the problem seems interesting.

As shown in Section 4, the complexity of a stay map can be $\Theta(n^2)$, rendering an algorithm with the time complexity $o(n^2)$ impossible. This bound however is not tight and a natural question is whether it is possible to find the exact stay map of two-dimensional trajectories in $O(n^2)$ time. Also, by limiting the size of the output, for instance by finding only one of the stay points, a more efficient algorithm is not unlikely. Furthermore, it seems interesting to study the problem in higher dimensions.

## References

[1] F. J. M. Arboleda, V. Bogorny, H. Patio. SMoT+NCS - Algorithm for Detecting Non-continuous Stops. Computing and Informatics 3(2), 283–306, 2017. doi:10.4149/cai_2017_2_283.

[2] S. W. Bae. An Almost Optimal Algorithm for Voronoi Diagrams of Non-Disjoint Line Segments. *Computational Geometry* 52, 34–43, 2016. doi:10.1016/J.COMGEO.2015.11.002.

[3] M. Benkert, B. Djordjevic, J. Gudmundsson, T. Wolle. Finding Popular Places. *International Journal of Computational Geometry and Applications* 20(1), 19–42, 2010. doi:10.1142/S0218195910003189.

[4] B. Chazelle, H. Edelsbrunner. An Optimal Algorithm for Intersecting Line Segments in the Plane. *Journal of the ACM* 39(1), 1–54, 1992. doi:10.1145/147508.147511.

[5] M. L. Damiani, H. Issa, F. Cagnacci. Extracting Stay Regions with Uncertain Boundaries from GPS Trajectories - A Case Study in Animal Ecology. In *ACM International Conference on Advances in Geographic Information Systems*, 253–262, 2014. doi:10.1145/2666310.2666417.

[6] B. Djordjevic, J. Gudmundsson. Detecting Areas Visited Regularly. In M.T. Thai, S. Sahni (Eds.) *Computing and Combinatorics* (COCOON 2010), *Lecture Notes in Computer Science* 6196, 244–253, 2010. doi:10.1007/978-3-642-14031-0_28.

[7] B. Djordjevic, J. Gudmundsson, A. Pham, T. Wolle. Detecting Regular Visit Patterns. *Algorithmica* 60(4), 829–852, 2011. doi:10.1007/S00453-009-9376-2.

[8] M. Fort, J. A. Sellarès, N. Valladares. Computing and Visualizing Popular Places. *Knowledge and Information Systems* 40(2), 411–437, 2014. doi:10.1007/s10115-013-0639-5.

[9] J. Gudmundsson, M. J. van Kreveld, F. Staals. Algorithms for Hotspot Computation on Trajectory Data. In *ACM International Conference on Advances in Geographic Information Systems* (SIGSPATIAL 2013), 134–143, 2013. doi:10.1145/2525314.2525359.

[10] R. Pérez-Torres, C. Torres-Huitzil, H. Galeana-Zapién. Full On-device Stay Points Detection in Smartphones for Location-based Mobile Applications. *Sensors* 16(10), 1693, 2016. doi:10.3390/s16101693.

[11] A. G. Rudi. Looking for Bird Nests: Identifying Stay Points with Bounded Gaps. In *The Canadian Conference on Computational Geometry*, 334–339, 2018. http://www.cs.umanitoba.ca/~cccg2018/papers/session7A-p2.pdf.

[12] A. G. Rudi. Approximate Hotspots of Orthogonal Trajectories. *Fundamenta Informaticae* 167(4), 271–285, 2019. doi:10.3233/FI-2019-1818.

[13] Y. Zhang, Y. Lin. An Interactive Method for Identifying the Stay Points of the Trajectory of Moving Objects. *Journal of Visual Communication and Image Representation* 59, 387–392, 2019. doi:10.1016/J.JVCIR.2019.01.038.

[14] Y. Zheng. Trajectory Data Mining - An Overview. *ACM Transactions on Intelligent Systems and Technology* 6(3), 29:1–29:41, 2015. doi:10.1145/2743025.

**© Scientific Annals of Computer Science 2019**