International Journal of Computational Geometry & Applications

FINDING POPULAR PLACES

MARC BENKERT
Department of Computer Science, Faculty of Informatics, University Karlsruhe
ITI Wagner, Box 6980, 76128 Karlsruhe, Germany
mbenkert@ira.uka.de

BOJAN DJORDJEVIC
NICTA, Sydney*
Locked Bag 9013, Alexandria NSW 1435, Australia
and
School of Information Technologies, University of Sydney, NSW 2006, Australia
bojan@it.usyd.edu.au

JOACHIM GUDMUNDSSON,
THOMAS WOLLE
NICTA, Sydney*
Locked Bag 9013, Alexandria NSW 1435, Australia
{*joachim.gudmundsson, thomas.wolle*}*@nicta.com.au*

Received (received date)
Revised (revised date)
Communicated by (Name)

Widespread availability of location aware devices (such as GPS receivers) promotes capture of detailed movement trajectories of people, animals, vehicles and other moving objects. We investigate spatio-temporal movement patterns in large tracking data sets, i.e. in large sets of polygonal paths. Specifically, we study so-called 'popular places', that is, regions that are visited by many entities.

Given a set of polygonal paths with a total of $\bar{n}$ vertices, we look at the problem of computing such popular places in two different settings. For the discrete model, where only the vertices of the polygonal paths are considered, we propose an $O(\bar{n} \log \bar{n})$ algorithm; and for the continuous model, where also the straight line segments between the vertices of a polygonal path are considered, we develop an $O(\bar{n}^2)$ algorithm. We also present lower bounds and hardness results.

Keywords: Moving point objects; spatio-temporal data; trajectories; movement pattern.

*National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

1. Introduction

Technological advances of location-aware devices, surveillance systems and electronic transaction networks produce more and more opportunities to trace moving individuals. Consequently, an eclectic set of disciplines including geography, market research, data base research, animal behaviour research, surveillance, security and transport analysis shows an increasing interest in movement patterns of entities moving in various spaces over various times scales.[1,2,3,4] In the case of moving animals, movement patterns can be viewed as the spatio-temporal expression of behaviours, e.g. in flocking sheep or birds assembling for the seasonal migration. In a transportation context, a movement pattern could be a traffic jam.

In this paper we will focus on the problem of computing *popular places* (also called *convergence patterns*[5,6]) among geospatial lifelines. The input is a set E of n moving point objects $\Lambda_1, \ldots, \Lambda_n$ whose locations are known at τ consecutive time-steps $t_1, \ldots, t_\tau$, that is, the *trajectory* of each object is a polygonal path that can self-intersect, see Fig. 1. For brevity, we will call moving point objects *entities* from now on, and when it is clear from the context, we use Λ to denote an entity or its trajectory. It is assumed that an entity moves between two consecutive time steps on a straight line, and the velocity of an entity along a line segment of the trajectory is constant. Given a set E of n moving entities in the plane, an integer $k > 0$ and a real value $r > 0$, a *popular place* is a square of side length r, that is visited by at least k entities in E. Throughout the article we will for simplicity assume $r = 1$. Note that the entities do not have to be in the square simultaneously. Spatio-temporal patterns have traditionally been considered in two settings: the discrete case, where only the *trajectory vertices* according to the discrete time steps are considered, and the continuous case, where the polygonal paths connecting the trajectory vertices are considered. Recently it has been argued that the continuous model is becoming more important,[7,8] because trajectories will have to be compressed (simplified) to allow for fast computations. Nowadays it is not unusual that the coordinates are recorded with a frequency of once per second. A popular place in the two different models is defined as follows (see Fig. 1).

Definition 1. Let E be a set of n moving entities in the plane, $k > 0$ be an integer and $r > 0$ a real value. An axis aligned square σ of side length r is a *popular place* in the *discrete model* if σ contains trajectory vertices from at least k different entities in E. In the *continuous model*, σ is a *popular place* if it is intersected by polylines from at least k different entities in E.

There has been considerable research in the area of analysing and modelling spatio-temporal data. In the database community, research has mainly focussed on indexing databases so that basic spatio-temporal queries concerning the data can be answered efficiently. Typical queries are spatio-temporal range queries and spatial or temporal nearest neighbour queries (see for example the work by Săltenis et al. and Hadjieleftheriou et al.[9,10]). From a data mining perspective, Verhein and Chawla

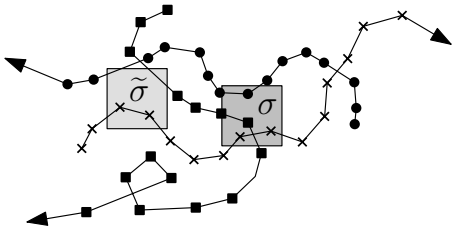


Fig. 1: An example where three entities Λ_1, Λ_2 and Λ_3 are traced during 16 time steps. For $k = 3$ the square $\widetilde{\sigma}$ is a popular place only in the continuous model, while σ is a popular place for $k = 3$ in both the discrete and continuous model.

used association rule mining to detect patterns in spatio-temporal sets.[11] They defined a region to be a *source, sink* or a *thoroughfare* depending on the number of objects entering, exiting or passing through the region. Mamoulis et al. studied periodic patterns,[12] e.g. yearly migration patterns or daily commuting patterns. Recently, there have been several papers considering the problem of detecting flock patterns and leadership patterns.[13,14,15,16]

Precursory to this work, Laube et al. proposed the REMO framework (RElative MOtion),[5,6] which defines similar behaviour in groups of entities. They defined patterns such as 'flock', 'convergence', 'trend-setting' and 'leadership' based on similar movement properties such as speed, acceleration or movement direction; and they gave algorithms to compute them efficiently. They proposed an input model, where a ray was drawn from the current position of each entity that corresponds to its direction. That is, the coordinates and direction of the entities are known at the initial time step, and their aim is to find or forecast a popular place (assuming the entities do not change their direction).

As mentioned in earlier work, specifying exactly which of the patterns should be reported is often a subject for discussion.[15,16,17] For the discrete model we design a general algorithm that can generate the following output:

- the popular place with the most number of entities (detect maximum),
- a set of rectangles of width 1 and height 2 such that each reported rectangle contains a popular place and all popular places are covered by the reported rectangles (approximate),
- a set of polygons $\mathcal{H}(E)$ such that any axis-aligned unit square with centre in a polygon of $\mathcal{H}(E)$ is a popular place (report all).

In the continuous model we only describe how to find the set $\mathcal{H}(E)$. However, one can easily modify the proposed algorithm to comply with any of the output models listed above.

In Section 2, we present an algorithm for the discrete model, followed by an $\Omega(\tau n \log \tau n)$ time lower bound in Section 3. In Section 4, we present an $O(\tau^2 n^2)$

time algorithm in the continuous model. Finally, in Section 5, we argue on an $\Omega(n^2\tau^2)$ hardness in the continuous model.

2. A Fast Algorithm in the Discrete Model

We start with the discrete version of the problem. A set E of n entities is traced over a period of τ discrete time steps, generating τn trajectory vertices in the plane that correspond to the positions of the tracked entities at the discrete time steps. The input parameter $k > 0$ defines the minimum number of entities defining a popular place, see Definition 1. In the next sections, we describe a plane sweep algorithm and data structures with which we can report the popular place with the largest number of entities in $O(\tau n \log \tau n)$ time. In Section 2.6, we show how to modify this algorithm to produce more general output. But first, we discuss a trivial algorithm using coloured range searching.

The current problem closely resembles the coloured range counting problem,[18] where the input is a set of $\bar{n}$ points, each point having one of $\bar{m}$ possible colours, and the aim is to preprocess the points such that the number of different colours inside a given query range can be reported.

Fact 1 (Theorem 5.1 by Gupta et al.[18]). A set S of $\bar{n}$ coloured points in the plane can be preprocessed into a data structure that requires $O(\bar{n}^2 \log^2 \bar{n})$ space and preprocessing time, such that for any axis-parallel query rectangle $q = [a, b] \times [c, d]$, the number of distinctly-coloured points in q can be reported in $O(\log^2 \bar{n})$ time.

This result can be used to obtain a simple approximation algorithm. Let $k_{\max}$ be the largest number of trajectory vertices belonging to different entities contained in a square of side length 1. In other words, $k_{\max}$ is the size of a maximum popular place. We say that an algorithm is an α-approximation algorithm if it returns a square of side length α that contains at least $k_{\max}$ entities. A 2-approximation algorithm can be obtained by performing τn coloured range counting queries, where each query square has side length 2 and is centred at a trajectory vertex. The algorithm requires $O(\tau^2 n^2 \log^2 \tau n)$ time and space for the preprocessing and all the queries. The main drawback is the space usage. In the applications considered, the size of the input may be very large as noted in the introduction. We will present an exact algorithm in the next section that only uses $O(\tau n)$ space and runs in $O(\tau n \log \tau n)$ time.

2.1. *Sweeping the trajectory vertices*

The general idea of our algorithm is to use a vertical sweep line that sweeps the trajectory vertices from left to right. Together with the sweep line, we sweep a vertical strip str_ℓ of width 1 whose right boundary is the sweep line. Each of the τn trajectory vertices induces two event points, one when the trajectory vertex enters str_ℓ and one when the trajectory vertex leaves str_ℓ. We refer to these event points as *start events* and *end events*.

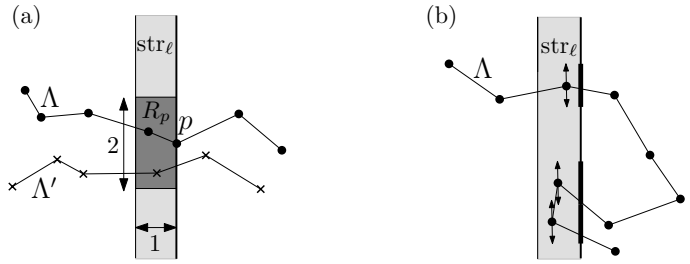


Fig. 2: (a) Finding the popular places in R_p visited by Λ. (b) The set I_Λ is indicated by the bold, solid line segments on the right side of str_ℓ.

For a start event, say that a trajectory vertex p belonging to entity Λ enters str_ℓ, we update our data structures and check for the largest popular place that is located in str_ℓ and that contains p. Such a popular place must be contained in the axis-aligned rectangle R_p having width 1, height 2 and p on the midpoint of its right vertical segment, see Fig. 2(a). Now, whenever we process a start event for a trajectory vertex p, we check for a largest popular place within R_p. In this way, we will find the maximum popular place. For an end event, say a trajectory vertex p belonging to Λ is about to leave str_ℓ, we simply remove it from the current data structures.

Finding a largest popular place within R_p could be done by sweeping all the trajectory vertices within R_p with a unit square s, while keeping track of the number of entities within s at any time. However, since the number of trajectory vertices within R_p is $O(\tau n)$ this might take $O(\tau n \log \tau n)$ time per event. Instead, we are going to show how we can maintain trees such that given a y-interval $[a, b]$, we can find a y-value within that interval, such that the unit square in R_p with centre at this y-value contains the largest number of different entities among all unit squares in R_p. We also show that such a query can be done in $O(\log \tau n)$ time. Below we will show how we can achieve this by maintaining a tree for each entity separately in $O(\log \tau)$ time per event, and in Section 2.3, we show how we can merge this information into one tree T that can be queried and updated in $O(\log \tau n)$ time.

2.2. *One structure for each entity*

For each entity $\Lambda \in E$, we maintain a set I_Λ of disjoint y-intervals, such that at any event point during the sweep, I_Λ contains exactly the y-intervals for which a square s in str_ℓ with centre in an interval in I_Λ contains a trajectory vertex of Λ, see Fig. 2(b). Hence, a square containing a maximum popular place is a square whose centre is contained in a maximum number of such y-intervals. We store I_Λ in a balanced binary search tree T_Λ. More specifically, the tree T_Λ stores the set I_Λ of intervals w.r.t. the current position of str_ℓ. The leaves of T_Λ store the endpoints of the intervals in I_Λ ordered on their y-coordinates. Each leaf also contains a pointer

to the leaf in T_Λ containing the other endpoint. Inserting and deleting a new interval can be done in $O(\log \tau)$ time per update; and hence, we have the following lemma.

Lemma 1. *For any entity Λ and throughout the entire sweep, T_Λ can be maintained in $O(\tau \log \tau)$ total time requiring $O(\tau)$ space.*

Proof. Before we go into details on how updates are performed on T_Λ, we introduce another balanced binary search tree T'_Λ as an auxiliary data structure. T'_Λ stores Λ's trajectory vertices that are currently within str_ℓ and ordered with respect to their y-coordinates. Note that it is straight forward to query and update T'_Λ in $O(\log \tau)$ time and linear space. We only need T'_Λ to efficiently update T_Λ, which is described next.

Assume we are about to process a start event $p = (x, y)$, where p is a trajectory vertex of Λ. That means that any unit square within str_ℓ with centre in the interval $i := [y - \frac{1}{2}, y + \frac{1}{2}]$ will contain p. Hence, we have to update I_Λ to make sure that the interval i is contained in an interval in I_Λ. To this end, we perform a range query in T_Λ that reports the intervals of I_Λ intersecting i. Since the intervals in I_Λ are disjoint and have length at least one, i may intersect at most two intervals in I_Λ. Thus, finding the intersecting intervals can be done in $O(\log \tau)$ time. If the number of intersecting intervals is zero then i is inserted into T_Λ. If i intersects one interval i_1 then i_1 is deleted and the interval $i \cup i_1$ is inserted. Finally, if two intervals i_1 and i_2 are intersected then i_1 and i_2 are deleted and $i \cup i_1 \cup i_2$ is inserted. We also have to update T'_Λ, which simply means to perform an insertion of p.

In the case where $p = (x, y)$ is an end event, we update the trees in a similar manner. We first report the two adjacent neighbours $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ of p in T'_Λ, and then delete p from T'_Λ (p_1 and p_2 are important for correctly updating the intervals in T_Λ). Assume without loss of generality that $y_1 < y < y_2$. Let i be the interval defined above and let i' be the interval in I_Λ containing i. We have to distinguish the following cases:

- If $|y_2 - y_1| \leq 1$, then we are done since i' does not change.
- If $\min\{|y - y_1|, |y - y_2|\} > 1$, then $i' = i$ is deleted from T_Λ.
- If $|y - y_1| > 1$ and $|y - y_2| \leq 1$, then i' is deleted from T_Λ and the interval $i' \setminus [y - \frac{1}{2}, y_2 - \frac{1}{2})$ is inserted. The case when $|y - y_2| > 1$ and $|y - y_1| \leq 1$ is symmetric.
- If $|y_2 - y_1| > 1$ and $|y - y_1| \leq 1$ and $|y - y_2| \leq 1$, then i' is deleted from T_Λ and the interval $i' \setminus (y_1 + \frac{1}{2}, y_2 - \frac{1}{2})$ is inserted.

Since the total number of events is $2 \cdot \tau$ the lemma follows. □

We need the trees T_Λ for easily and efficiently updating the status structure of the sweep line.

2.3. *Maintaining the status of the sweep*

To speed up finding the largest popular place in R_p, we will maintain a status structure T of str_ℓ. The status for the current position of str_ℓ is described by the trees T_Λ for all $\Lambda \in E$. Hence, as our status structure, we use yet another balanced binary search tree T, which combines the information represented by the trees T_Λ for all $\Lambda \in E$. More precisely, T stores all intervals that are currently represented in all the trees T_Λ, i.e. all intervals in $\bigcup_\Lambda I_\Lambda$. Note that in contrast to T_Λ, the tree T represents intervals that may intersect. The leaf set of T corresponds to the set of start and end points of intervals in $\bigcup_\Lambda I_\Lambda$ ordered w.r.t. their y-coordinates. For simplicity, we assume w.l.o.g. that all start and end points of intervals are pairwise disjoint. Each interval in T_Λ is also contained in T; and for each interval in T, there exists a T_Λ that contains it. Using the trees T_Λ, we can efficiently maintain T.

Lemma 2. *Throughout the sweep, the status structure T can be maintained in $O(\tau n \log \tau n)$ time requiring $O(\tau n)$ space.*

Proof. Since there are τn trajectory vertices, T contains at most $O(\tau n)$ leaves and thus at most $O(\tau n)$ nodes at all times. During the sweep, T is maintained as follows: We first perform the update in the corresponding tree T_Λ, which identifies exactly which intervals will be deleted and/or inserted. Then, exactly those intervals will also be deleted and/or inserted in T. One update in T_Λ requires the deletion and insertion of a constant number of leaves in T. Thus, one update operation in T_Λ induces update operations in T that can be performed in $O(\log \tau n)$ time. Since the sweep performs $2\tau n$ update operations, the lemma follows. □

We will use T to perform the maximum popular place queries with the axis-aligned rectangle R_p. To do this, we show in the next section how we can store additional information within T in order to efficiently perform such queries.

2.4. *Extending T to allow for efficient queries*

A number y is said to *stab* an interval $[a, b]$ if $y \in [a, b]$. The *stabbing number* of y w.r.t. a set of intervals I is the number of intervals in I that y stabs. Recall that the leaves of T have y-values associated with them, and hence, we extend this notion to leaves: the *stabbing number of a leaf* L_i of T is the stabbing number of the y-value associated with L_i w.r.t. the set of intervals represented by T.

Note that for a start event $p = (x, y)$, we have to find the y-value in $[y - \frac{1}{2}, y + \frac{1}{2}]$ having maximum stabbing number w.r.t. $\bigcup_\Lambda I_\Lambda$, since this establishes the maximum popular place within R_p. Ideally, we could store the stabbing numbers of the leaves in T. However, one event could require the update of $O(\tau n)$ of these numbers which would be too costly to maintain. Instead, we maintain this information implicitly in order to do updates as well as queries in $O(\log \tau n)$ time.

For this, we store two values, $sum(v)$ and $max_{\mathrm{pre}}(v)$, with each node v in T, see Fig. 3. The idea is that these numbers will help us to find a leaf with the highest stab-

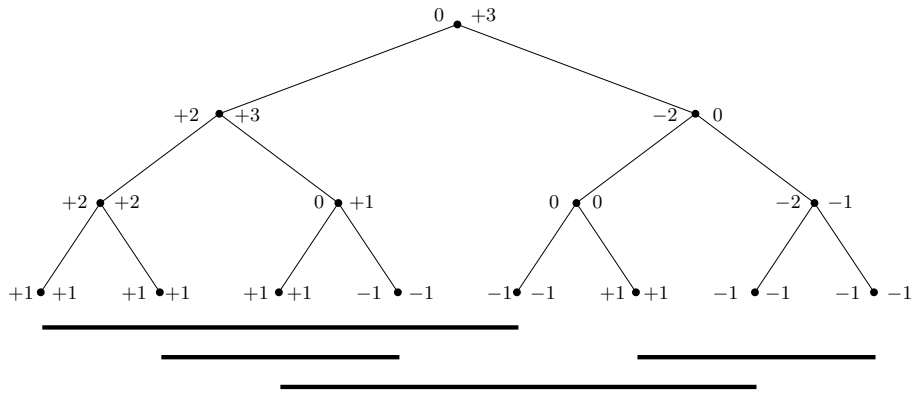


Fig. 3: Four intervals and the tree T representing them and the values *sum* (left) and max_{pre} (right) for the depicted tree T.

bing number. For a leaf v, we define $sum(v)$ and $max_{\text{pre}}(v)$ to be $+1$, if v corresponds to an interval start point, and to be -1, if v corresponds to an interval end point. For an internal node v, the intuition is that $sum(v)$ is the sum over all the *sum*-values of leaves that are in the subtree rooted at v, and $max_{\text{pre}}(v)$ is the maximum sum of the *sum*-values over all prefixes of the sequence of leaves that are in the subtree rooted at v. More formally, let $L_1(v), \ldots, L_m(v)$ be the sequence of all leaves contained in the subtree rooted at v enumerated according to increasing y-coordinates. Then we define $sum(v) := \sum_{j=1}^{m} sum(L_j(v))$ and $max_{\text{pre}}(v) := \max_{1 \le i \le m} \sum_{j=1}^{i} sum(L_j(v))$. For internal nodes v let v^{left} and v^{right} denote the left and the right child of v, respectively. Then, the *sum*- and max_{pre}-values can be recursively computed as follows:

$$sum(v) := sum(v^{left}) + sum(v^{right})$$

$$max_{\text{pre}}(v) := \max\{max_{\text{pre}}(v^{left}), sum(v^{left}) + max_{\text{pre}}(v^{right})\}$$

When performing an update operation in T, i.e. deleting or inserting a leaf v, updating *sum* and max_{pre} is only required on the path from v to the root r of T. Hence, updating *sum* and max_{pre} takes $O(\log \tau n)$ time per update operation, as is stated in the following lemma.

Lemma 3. *Throughout the sweep, the values sum and* max_{pre} *can be maintained in* $O(\tau n \log \tau n)$ *total time requiring* $O(\tau n)$ *space.*

2.5. *Querying T for the maximum stabbing number*

Recall the processing of the sweep. When we arrive at a start event p, we first perform the required update in T and then query T for the most popular place in R_p. Next, we show how the query can be done in $O(\log \tau n)$ time.

Each of the leaves in T is associated with the y-value of the stored start or end point. Let $L_1, L_2, \ldots$ be the leaves in T ordered from left to right according to increasing y-coordinate. We first state two lemmas that are needed when we compute the stabbing numbers by computing the sum over *sum* and max_{pre}-values.

Lemma 4. *The stabbing number of a leaf L_i is*

$$\max\Big(\sum_{j=1}^{i-1} sum(L_j), \sum_{j=1}^{i} sum(L_j)\Big)$$

Proof. Let y be any number, and let $y_1, y_2, ...$ be the y-values associated with $L_1, L_2, ...$ If y is a number with $y_m < y < y_{m+1}$, then the stabbing number of y is equal to $\sum_{j=1}^{m} sum(L_j)$. If $y = y_i$ is a start point of an interval, then y stabs $\sum_{j=1}^{i-1} sum(L_j)$ intervals plus one additional interval, namely the one that starts at y. Hence, y stabs $\sum_{j=1}^{i} sum(L_j) = 1 + \sum_{j=1}^{i-1} sum(L_j)$ intervals, since $sum(L_i) = +1$. If $y = y_i$ is an end point of an interval, then y stabs $\sum_{j=1}^{i-1} sum(L_j)$ intervals, which includes the interval that ends at y. Hence, y stabs $\sum_{j=1}^{i-1} sum(L_j) = 1 + \sum_{j=1}^{i} sum(L_j)$ intervals, since $sum(L_i) = -1$. □

By performing two searches in T, we can find the leftmost leaf L_l in T whose y-value is at least $y_p - \frac{1}{2}$ and the rightmost leaf L_r whose y-value is at most $y_p + \frac{1}{2}$ in $O(\log \tau n)$ time. Our goal is to find among the leaves $L_l, ..., L_r$ a leaf whose associated y-value has maximum stabbing number. However, since we will use sums over *sum* and max_{pre}-values to compute this leaf, we have to adjust this query range from the index range $\{l, ..., r\}$ to the index range $\{l-1, l, ..., r\}$. This is formalised in the next lemma.

Lemma 5. *Let i be an index with $i \in \{l-1, l, ..., r\}$ such that $\sum_{j=1}^{i} sum(L_j) = \max_{i' \in \{l-1,l,...,r\}} \sum_{j=1}^{i'} sum(L_j)$. If $i \geq l$, then L_i is a leaf with maximum stabbing number among the leaves $L_l, ..., L_r$. If $i = l-1$, then L_l is a leaf with maximum stabbing number among the leaves $L_l, ..., L_r$.*

Proof. Note that i is the index that realises $\max_{i' \in \{l-1,l,...,r\}} \sum_{j=1}^{i'} sum(L_j)$.

If $i \in \{l, ..., r-1\}$, then from Lemma 4 and the choice of i, we have that L_i and L_{i+1} have the same stabbing number and this stabbing number is maximal among the leaves $L_l, ..., L_r$. If $i = r$, then from Lemma 4 and the choice of i, we have that L_r has maximal stabbing number among the leaves $L_l, ..., L_r$. Hence, in both cases we conclude that L_i is a leaf with maximum stabbing number.

If $i = l-1$, then from Lemma 4 and the choice of i, we have that $L_{i+1} = L_l$ has maximal stabbing number among the leaves $L_l, ..., L_r$. □

Hence, $\{L_{l-1}, L_l, .., L_r\}$ defines our query range in T; and our goal now is to find the leaf L_i among these leaves such that $\sum_{j=1}^{i} sum(L_j)$ is maximised. We denote

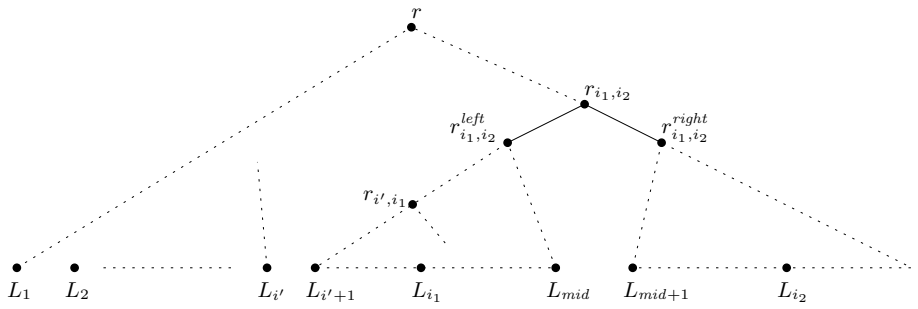


Fig. 4: Querying T for max_{i_1,i_2}.

this number by $max_{l-1,r}$, and more generally, we define:

$$max_{i_1,i_2} := \max_{i_1 \le m \le i_2} \Big\{ \sum_{j=1}^{m} sum(L_j) \Big\}$$

Lemma 6. *Given T and two leaves L_{i_1} and L_{i_2} of T, one can, in $O(\log \tau n)$ time, return a leaf in $\{L_{i_1}, ..., L_{i_2}\}$ that realises max_{i_1,i_2}.*

Proof. Let r_{i_1,i_2} be the least common ancestor in T of L_{i_1} and L_{i_2}, and let L_{mid}, with $i_1 \le mid < i_2$, be the rightmost leaf in the left subtree of r_{i_1,i_2}, see Fig. 4. A leaf that realises max_{i_1,i_2} is either contained in $\{L_{i_1}, ..., L_{mid}\}$ or in $\{L_{mid+1}, ..., L_{i_2}\}$. Hence, we can divide our query into two queries. Let $max_1 := \max_{i_1 \le m \le mid} \sum_{j=i_1}^{m} sum(L_j)$ and $max_2 := \max_{mid+1 \le m \le i_2} \sum_{j=mid+1}^{m} sum(L_j)$. Once we know max_1 and max_2, we can compute max_{i_1,i_2} by

$$max_{i_1,i_2} = \max \Big\{ \sum_{j=1}^{i_1-1} sum(L_j) \; + \; max_1 \; , \; \sum_{j=1}^{mid} sum(L_j) \; + \; max_2 \Big\}.$$

Hence, it remains to show how each of the terms of these sums can be computed. The general idea for each of those terms is that we accumulate certain values stored at nodes, when walking along a path between the root r of T and a leaf.

The term $\sum_{j=1}^{i_1-1} sum(L_j)$ can be computed in the following way. We use a temporary variable s, initially set to 0. We walk along the path from the root r of T to L_{i_1-1}, and each time we walk from a node v to the right child v^{right} of v, we perform an update: $s := s + sum(v^{left})$. When we reached the leaf L_{i_1-1}, we update s one last time: $s := s + sum(L_{i_1-1})$. Now, we have that $s = \sum_{j=1}^{i_1-1} sum(L_j)$. The term $\sum_{j=1}^{mid} sum(L_j)$ can be computed in a very similar way.

We compute max_1 by walking up the tree along the path from L_{i_1} to $r^{left}_{i_1,i_2}$. Initially, we set $max_1 := max_{\text{pre}}(L_{i_1})$. Again, we use a helper variable s initialised to $s := sum(L_{i_1})$. Let $v_0, v_1, \ldots, r^{left}_{i_1,i_2}$ (with $v_0 = L_{i_1}$) be the sequence of nodes that are traversed when walking from L_{i_1} to $r^{left}_{i_1,i_2}$ in T. Each time we encounter

a node v_i having v_{i-1} as a left child, we look for a new maximum in the right subtree, i.e. $max_1 := \max\{max_1\ ,\ s + max_{\text{pre}}(v_i^{right})\}$ and then update s to $s := s + sum(v_i^{right})$. At the end of the traversal max_1 holds the right value.

To compute max_2, we walk along the path from r_{i_1,i_2}^{right} to L_{i_2}. Let $r_{i_1,i_2}^{right} = v_0, v_1, \ldots, L_{i_2}$ be the sequence of traversed nodes. Initially, we set $max_2 := -\infty$ and a temporary helper variable $s := 0$. Each time we encounter a node v_i which is a right child of its parent, we look for a new maximum in the left subtree, i.e. we perform a possible update: $max_2 := \max\{max_2\ ,\ s + max_{\text{pre}}(v_{i-1}^{left})\}$ and then update s to $s := s + sum(v_{i-1}^{left})$. After arriving and performing the updates at L_{i_2}, we obtain max_2 by one additional update: $max_2 = \max\{max_2, s + max_{\text{pre}}(L_{i_2})\}$.

There are $O(\log \tau n)$ nodes on any path from a leaf to the root r of T, and each node is processed in constant time. Thus, the total time to compute max_{i_1,i_2} is $O(\log \tau n)$. □

Now we are ready to state the main result of this section. The correctness follows by putting together Lemmas 2, 3, 5 and 6.

Theorem 1. *Given a set E of n entities in the plane, the unit square containing the maximum number of different entities in the discrete model can be computed in $O(\tau n \log \tau n)$ time using $O(\tau n)$ space.*

2.6. *Approximating and reporting all popular places*

Recall that according to Definition 1, k is the minimum number of different entities that a unit square s must contain for s to be a popular place. For a start event p, we have seen in Section 2.5 how we can detect a unit square s in R_p that contains the maximum number of different entities in the discrete model among all unit squares contained in R_p. If we now find that this number is at least k, we can report R_p as an approximation for (potentially) all popular places that are contained in R_p. This leads directly to the following result.

Theorem 2. *Given a set E of n entities in the plane, we can report rectangles of width 1 and height 2 such that each reported rectangle contains a popular place and all popular places are covered by the reported rectangles. This requires $O(\tau n \log \tau n)$ time and $O(\tau n)$ space.*

It gets more involved when we want to report the set $\mathcal{H}(E)$ of polygons such that any axis-aligned unit square with centre in a polygon in $\mathcal{H}(E)$ is a popular place and each centre of a popular place is contained in a polygon in $\mathcal{H}(E)$. Say that we process a start event p and find a popular place $s \subset R_p$. Then, s will most likely still be a popular place if we shift it slightly to the right. To find out the rightmost position of s still being a popular place is the difficulty of the problem to report all popular places.

We first show how we can find all popular places in R_p when processing a start event $p = (x_p, y_p)$. We will report them as an interval set $\mathcal{I}_{p,\text{start}}$ such that for each

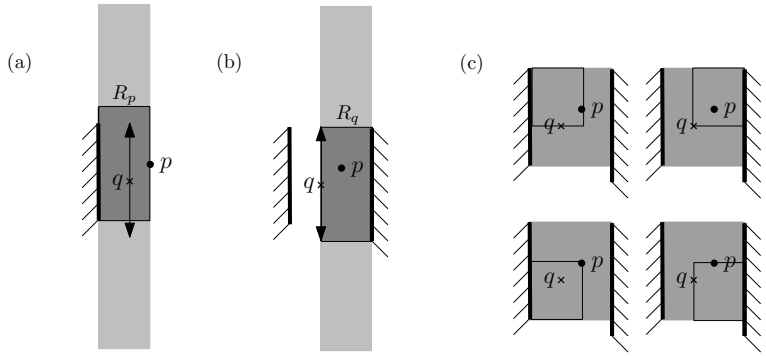


Fig. 5: (a) setting start flags, (b) setting end flags, and (c) the reported polygon (grey) in $\mathcal{H}(E)$.

$I \in \mathcal{I}_{p,\text{start}}$ it holds that $I \subseteq [a, b]$, where a and b are the bottom and topmost y-coordinates defining R_p, and furthermore, the squares with centres on any $y \in I$ give all popular places. The task is to find all leaves between L_l and L_r (that define the query range induced by R_p) whose stabbing number is at least k. Using the above techniques, we can find a leaf that induces a popular place in $O(\log \tau n)$ time. This means that we can find all leaves in $O(M \log \tau n)$ time per event, where M is the number of all leaves between L_l and L_r whose stabbing number is at least k.

Note that each leaf L_j is associated with a y-value y_j, which means if we recognise L_j as a popular place, we will actually report the line segment $x_p \times [y_j, y_{j+1}]$ as a popular place defining interval. To find out if the leaves L_l or L_r induce popular places, we actually have to be a bit more careful and determine whether we have to extend the reported line segment to the lower end a or the upper end b of the query range, by checking how many intervals the leaf stabs; and in case this number is exactly k whether an interval starts or ends at L_l or L_r, respectively.

Next, we make the following observations: let $\mathcal{I}_{p,\text{start}}$ be the popular place defining interval set that we have found for a start event p and let s be a popular place associated with $\mathcal{I}_{p,\text{start}}$. Let s' denote the position of the unit square s when we slightly shift it to the left. If s' is also a popular place, we will have recognised this popular place earlier on when we processed a start event to the left of the current sweep line position. So, finding the left boundary of a polygon in $\mathcal{H}(E)$ can be done by setting start flags whenever we find the set $\mathcal{I}_{p,\text{start}}$ for a start event p: for each $I \in \mathcal{I}_{p,\text{start}}$, we set the start flag $[x_p - 1] \times I$, see Fig. 5. However, we still have to determine the right boundary of the polygons in $\mathcal{H}(E)$. We do this by setting end flags: for an end event q, we also compute the set of popular place defining intervals. Now, let $\mathcal{I}_{q,\text{end}}$ be the complement of this set in R_q extended with the popular place defining intervals whose stabbing number is exactly k (q is about to leave). For each $I \in \mathcal{I}_{q,\text{end}}$ we set the end flag $x_p \times I$, see Fig. 5. Now, we have to connect the start and end flags accordingly to obtain the polygons in $\mathcal{H}(E)$.

Theorem 3. *Given a set E of n entities in the plane, the polygons $\mathcal{H}(E)$ can be reported in $O(\tau n \log \tau n + M \log \tau n)$ time using $O(\tau n + M)$ space, where M is the number of all popular place defining intervals that we find throughout the algorithm.*

3. A Lower Bound in the Discrete Model

One of the first problems shown to have an $\Omega(N \log N)$ lower bound was the MIN-GAP problem.[19] The decision version of the MIN-GAP problem also has an $\Omega(N \log N)$ lower bound.[20]

Problem 1. (MIN-GAP, decision version)
Given a vector $x \in \mathbb{R}^N$ of reals and a positive real value δ, is $\min_{i \neq j} |x_i - x_j| > \delta$?

Now consider the decision version of the discrete popular place problem in one dimension for n entities, τ time steps and $k = 2$.

Problem 2. (POP-PLACE1, decision version)
Given a set Y of n vectors $y \in \mathbb{R}^\tau$ of reals and a positive real value δ, is there a popular place of side length δ and $k \geq 2$?

Theorem 4. *Problem 2 has an $\Omega(\tau n \log \tau n)$ lower bound.*

Proof. Consider an algorithm $\mathcal{A}(Y)$ that solves Problem 2 in $T_{\mathcal{A}}(Y)$ time. We show how algorithm $\mathcal{A}$ can be used to solve Problem 1.

Let $1 \leq a \leq N$ be a fixed integer. We transform an instance of Problem 1 into an instance of Problem 2. Suppose we are given a vector $x \in \mathbb{R}^N$ of reals and a positive real value δ as input to Problem 1. Place the values in x in a matrix M with a columns and $b = \lceil \frac{N}{a} \rceil$ rows, in any order. (In the case that $\frac{N}{a}$ is not an integer, fill up the matrix with appropriate dummy values.)

Consider the columns in M as a set of a vectors of length b. This set of vectors can be interpreted as a set Y_1 of $n = a$ trajectories over $\tau = b$ time steps in one dimension. Run algorithm $\mathcal{A}$ with Y_1 as input. If the algorithm $\mathcal{A}$ returns 'Yes' then return 'Yes' as the answer to Problem 1. Otherwise, consider the rows in M as a set Y_2 of $n = b$ trajectories over $\tau = a$ time steps. Run algorithm $\mathcal{A}$ again, but this time with Y_2 as input. If the algorithm $\mathcal{A}$ returns 'Yes' then return 'Yes', otherwise return 'No' as the answer to Problem 1.

To prove the correctness of this transformation consider the pair of real numbers x_i, x_j in x with the smallest gap in x. If x_i and x_j are in different columns then algorithm $\mathcal{A}(Y_1)$ will report 'Yes' if $|x_i - x_j| \leq \delta$. If x_i and x_j are in the same column they must belong to different rows, thus $\mathcal{A}(Y_2)$ will report 'Yes' if $|x_i - x_j| \leq \delta$, otherwise 'No'.

Thus, we can solve Problem 1 with algorithm $\mathcal{A}$ in $O(N) + T_{\mathcal{A}}(Y_1) + T_{\mathcal{A}}(Y_2)$ time. As there is a lower bound for this time of $\Omega(N \log N)$, this implies that $\max\{T_{\mathcal{A}}(Y_1), T_{\mathcal{A}}(Y_2)\} = \Omega(N \log N) = \Omega(\tau n \log \tau n)$, because $\tau n = \Theta(N)$. □

4. An Exact Algorithm in the Continuous Model

In this section, we consider the continuous model and present an $O(\tau^2 n^2)$ time algorithm using $O(\tau n)$ space. Later in Section 5 we will argue that it is unlikely to find an asymptotically faster algorithm. Our algorithm takes as input a set E of n entities $\Lambda_1, \ldots, \Lambda_n$ moving in the plane over τ time steps. The output of the algorithm will be a set $\mathcal{H}(E)$ of polygons. $\mathcal{H}(E)$ is the maximal point set in the plane such that any axis-aligned unit square whose centre lies in $\mathcal{H}(E)$ intersects at least k trajectories of E. Note that these polygons are in general not rectilinear polygons.

We first show how to construct an arrangement $\mathcal{A}$ of lines. The general idea is to sweep the arrangement $\mathcal{A}$ and then building $\mathcal{H}(E)$. For ease of presentation, we will initially describe an algorithm using a standard sweep-line technique with running time $O(\tau^2 n^2 \log \tau n)$ using $O(\tau^2 n^2)$ space. This sweep-line algorithm identifies the edges that contribute to the polygons in $\mathcal{H}(E)$. In a second sweep over these edges, we will construct the polygons in $\mathcal{H}(E)$. Note that the presented algorithms work for inputs with degeneracies and are easy to implement. We then observe that our methods do not require a sweep by a straight line. Hence, we can use a topological plane sweep, which was introduced by Edelsbrunner and Guibas,[21] to improve the running time to $O(\tau^2 n^2)$ and the used space to $O(\tau n)$.

4.1. *Constructing the Line Arrangement*

Recall that we use Λ to denote both an entity and its trajectory. Also recall that a trajectory is a polygonal path described by τ trajectory vertices, and that two consecutive trajectory vertices are connected by a straight-line segment s. Consider the following polygon construction: for a trajectory Λ sweep an axis-aligned unit square σ along the trajectory such that σ's centre moves on Λ as shown in Fig. 6(a). The region swept by σ induces a polygon which we denote by $P(\Lambda)$, see Fig. 6(b). Note that any axis-aligned unit square having its centre within $P(\Lambda)$, will intersect Λ. Thus, we can restrict ourselves to consider the centre locations for the popular-place defining squares.

Consider a set W of polygons and a point q in the plane. The *depth* of q with respect to W is the number of polygons in W intersecting q. This definition allows us to describe $\mathcal{H}(E)$ as follows:

Observation 1. $\mathcal{H}(E)$ consists of the set of points having depth at least k with respect to $\{P(\Lambda_1), \ldots, P(\Lambda_n)\}$.

However, we do not explicitly store the polygons $P(\Lambda)$ for each Λ, because $P(\Lambda)$ might have $O(\tau^2)$ holes, which in turn results in $O(\tau^2)$ storage space for each $P(\Lambda)$. Instead, the following approach will result in $\Theta(\tau)$ line-segments per trajectory, which guarantees linear space usage and fewer event points during the sweep. For each segment s of Λ consider the region swept along s by the unit-square σ. This region is a polygon $P(s)$ with at most six edges, see Fig. 6(c). When considering

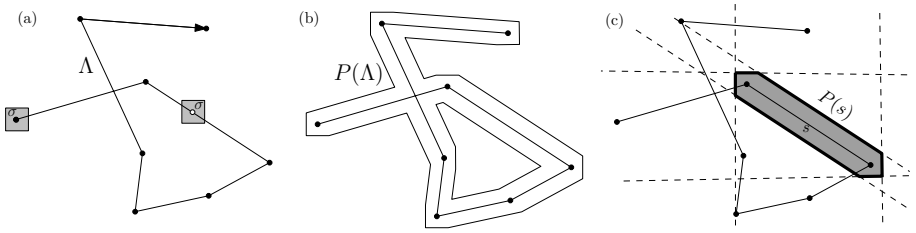


Fig. 6: (a) a trajectory Λ and the square σ that sweeps along Λ, (b) the polygon $P(\Lambda)$ obtained from the sweep, (c) the polygon $P(s)$ and the lines l_i generated by $P(s)$ (dashed)

the two points that specify an edge e_i of $P(s)$, we call the point with the smaller x-coordinate (or smaller y-coordinate, in case the x-coordinates are equal) the *start point* of e_i and the other one the *end point* of e_i. For each edge e_i of $P(s)$, we construct an infinite line l_i that contains e_i. For each line l_i, we store the start and end point of e_i on l_i, and to which side s lies, and Λ. We refer to e_i as a *visible edge* and to the rest of l_i as the *invisible line*. The set of all lines constructed as above yields the line arrangement $\mathcal{A}$, which we will sweep. Note that $\mathcal{A}$ contains $O(\tau n)$ lines.

4.2. *The First Sweep*

The algorithm will sweep the arrangement $\mathcal{A}$ from left to right using a vertical sweep line ℓ. The status structure of the sweep line is stored in a list S of size $O(\tau n)$. For convenience, we sometimes use S as if it was an ordered string. It contains the current intersections between ℓ and the visible edges in $\mathcal{A}$ ordered along ℓ from top to bottom. Initially, S is empty. For each intersection between ℓ and a visible edge e, we store a *bracket* br in S with pointers between br, the corresponding edge e and the line corresponding to e. A bracket is formally a tuple $\langle i, type, level, depth\rangle$, where i is the entity number corresponding to the edge; *type* is either *open* or *closed* depending on whether we are entering or exiting the polygon $P(s)$ as we go down ℓ. The brackets will always come in open-closed pairs, because $P(s)$, for a segment s, is a convex region. Note that we can consider them as matching pairs of brackets for the same entity, and that the matching brackets do not have to come from the same polygon $P(s)$. For example in Fig. 7(a), if going down ℓ then we enter two polygons belonging to entity Λ_1 and then exit both of them. We get the following sequence of brackets: `(())`. We say that the first and last are one matching pair (event though they do not correspond to the same polygon) with nesting level (*level*) equal to 1, and the second and third bracket are a different pair with nesting level 2. The *depth* value of a bracket is the depth with respect to $\{P(\Lambda_1), \ldots, P(\Lambda_n)\}$ of points immediately after this bracket until the next bracket as we go down ℓ. Note that

depth only counts each entity once, so a point that stabs many polygons belonging to the same trajectory will only get a contribution of 1 to *depth* from that entity. Therefore in `(())` the second bracket will have $depth = 1$ since the middle region only stabs one unique entity. An example in shown in Fig. 7(b).

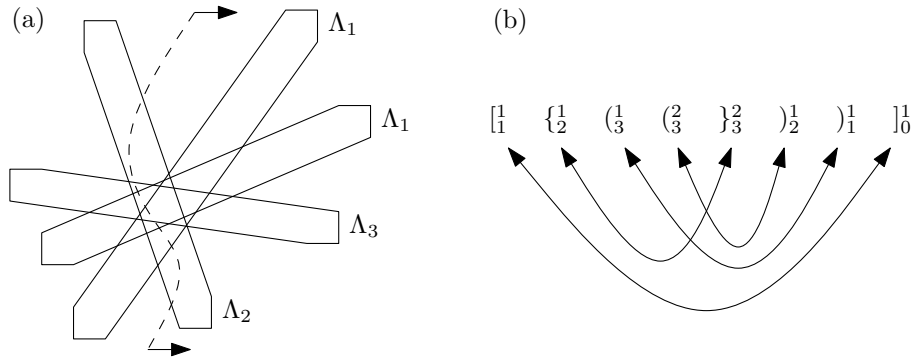


Fig. 7: (a) Four polygons for trajectory segments of three entities Λ_1, Λ_2 and Λ_3. The dashed line indicates the (topological) sweep line that sweeps the polygons from left to right. (b) Brackets corresponding to the sweep shown in (a). We use $(, [, \{$ for Λ_1, Λ_2 and Λ_3 respectively. Superscripts and subscripts represent *level* and *depth*. Matching brackets are shown with arrows.

An *event* of the sweep is the intersection of exactly two lines of the arrangement $\mathcal{A}$. These events happen at the vertices of the arrangement $\mathcal{A}$, which we call *event points* from now on. By our construction it will happen that three or more lines of $\mathcal{A}$ intersect in one point. Hence, multiple events can occur at a single event point. The moment at which the sweep line reaches an event point x the status of the sweep line is updated according to all events that happen at that event point. All events and event points are computed, sorted and stored beforehand.

Recall that each line l_i of $\mathcal{A}$ corresponds to an edge e_i of $P(s)$, for some s of some Λ. Also recall that together with l_i, we stored the two points on l_i that are the start and end points of the edge e_i, see Fig. 6(c). Now, we can distinguish the following categories for a line l_i that passes through an event point x:

(I) x coincides with the end point of e_i
(II) x coincides with the start point of e_i
(III) x lies in the interior of e_i
(IV) x and e_i are disjoint

During the sweep, we process all event points in turn, and for each event point x, we will do the following.

- Let L be the set of lines of $\mathcal{A}$ that go through x. Note that there are $\Theta(|L|^2)$ events happening at the event point x.
- Let S' be the substring of S that contains all the brackets that correspond to lines that go through x. Note that all these brackets appear consecutively in S, and that S' represents brackets before the event point x is processed.
- Let S'' be a new string that contains copies of exactly those brackets contained in S'. From now on, we will modify (brackets of) S'' using the information represented in S'. At the end S'' will represent brackets after x has been processed.
- For all events, i.e. for all pairs of lines l_1 and l_2 in L, let l_1 and l_2 be associated with an edge of $P(s_1)$ and $P(s_2)$ for some segments s_1 and s_2, respectively. We distinguish the following cases (see Fig. 8):

  (a) Both l_1 and l_2 are of category (II). If $P(s_1) = P(s_2)$ then we encountered a new polygon $P(s)$ and we insert a pair of brackets (anywhere) into S''. (The position of these brackets does not matter at this time, because in the next step the right order will be established by sorting all brackets.)

  (b) Both l_1 and l_2 are of category (I). If $P(s_1) = P(s_2)$ then we finished sweeping a polygon $P(s)$ and we delete the corresponding pair of brackets in S''.

  (c) l_1 is of type (I), l_2 is of type (II). If $P(s_1) = P(s_2)$ then we need to change a pointer. More specifically, let br be the bracket in S'' corresponding to the visible segment of l_1. We have to change the pointer of br that points to l_2 so that it now points to l_1. The symmetric opposite case can be handled in a similar way.

  (*) For all other cases, we do not make any changes.

  Note, that there are only $O(\tau n)$ events of the first three cases in total.
- We sort the brackets in S'' in non-increasing order according to the slope of their corresponding lines. This can be done in $O(|L| \log |L|)$ time.
- Recalculate all the *level* values from scratch for all the brackets in S''. This can be done for each entity Λ_i independently. Let br' and br'' be the first brackets that correspond to Λ_i in S' and S'', respectively. Note that br' and br'' must have the same *level* value. All other *level* values in S'' can be computed by traversing S'' starting at br''. Whenever we encounter an open or closed bracket corresponding to Λ_i, we set its *level* to be the level of the previous bracket plus or minus one, respectively. This costs $O(|L|)$ time per entity. Hence, in total this can be done in $O(|L|^2)$ time.
- Recalculate all the *depth* values from scratch for all the brackets in S''. Having the *depth* values of the first brackets and all the *level* values, we can calculate the *depth* values of all brackets in a way that is similar to the previous step. Also this can be done in $O(|L|^2)$ time in total.
- Replace S' by S'' in S.

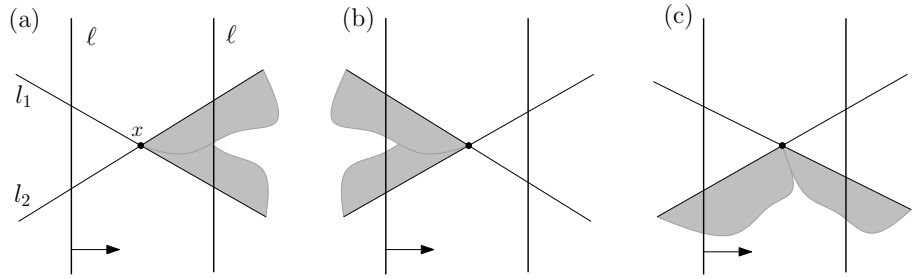


Fig. 8: Cases at an event point x during the line-arrangement sweep.

Lemma 7. *The status of the sweep line can be maintained during the sweep in total time $O(\tau^2 n^2)$ and $O(\tau n)$ space.*

Proof. The above procedure describes how to process an event point x in which $|L|$ lines intersect. Note that there are $\Theta(|L|^2)$ events happening at x, and all of them can be processed in $O(|L|^2)$ time and $O(|L|)$ space. Hence, each single event can be processed in $O(1)$ amortised time. Since each trajectory induces 6τ lines in $\mathcal{A}$ and there are $O(\tau^2 n^2)$ events in total, the lemma follows. □

4.3. *Constructing the Output*

In the above, we did not describe what our algorithm outputs. In this section, we will make up for this by adding a few steps to the previous sweep. Recall that the output of our algorithm will be a set $\mathcal{H}(E)$ which consists of all points having depth at least k with respect to $\{P(\Lambda_1), \ldots, P(\Lambda_n)\}$.

Recall that by definition any edge of $\mathcal{A}$ has a start point that is to the left of its end point. Hence, for each edge e and corresponding bracket br we also have two unique faces of $\mathcal{A}$, one below e, the other above e. The *depth* value of br gives us the depth of (the points in) the face that is below br, while the *depth* value of the bracket in S that is just before br gives us the depth of (the points in) the face above br. Whenever we have a bracket br, where the two corresponding faces have depth $k-1$ and k then we call that bracket a *boundary bracket*. A boundary bracket br corresponds to a *boundary edge*, which is a part of the edge e corresponding to br that is an edge of a polygon in $\mathcal{H}(E)$. The start and end points of boundary edges are the event points of the sweep, where a bracket becomes a boundary bracket, or where a boundary bracket ceases to be a boundary bracket, or where a boundary bracket is inserted, or where a boundary bracket is deleted.

To identify the set B of all boundary edges, we extend the procedure of the previous section. Recall that this procedure handles all events at a given event point. More specifically, we add the following steps to that procedure just before we replace S' by S''.

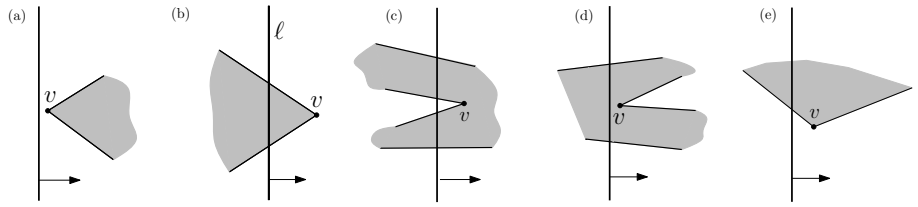


Fig. 9: The cases that may occur while building $\mathcal{H}(E)$.

- For all brackets $br \in S'' \setminus S'$:
  - If br is a boundary bracket, then add a new boundary edge b to B. The start point of b will be the current event point. Associate b with the boundary bracket br.
- For all brackets $br \in S' \setminus S''$:
  - If br is a boundary bracket, then the current event point is the end point of the boundary edge associated with br.
- For all brackets $br \in S' \cap S''$:
  - If br is a boundary bracket in S'' but not in S', then add a new boundary edge b to B. The start point of b will be the current event point. Associate b with the boundary bracket br.
  - If br is a boundary bracket in S' but not in S'', then the current event point is the end point of the boundary edge associated with br.
  - If br is a boundary bracket in S' and in S'', but br corresponds to different lines in S' and S'', then x is a start and end point of boundary edges. The current event point x is the end point of the boundary edge associated with br. We also add a new boundary edge b to B with start point the current event point x. Associate b with the boundary bracket br in S''.

Having the information about all the boundary edges allows us to build $\mathcal{H}(E)$ by performing a second sweep traversing the set B of boundary edges with a vertical sweep line ℓ. Note that no two boundary edges can intersect. An event point x of this second sweep will either be (a) a start vertex, (b) an end vertex, (c) a merge vertex, (d) a split vertex or (e) a regular vertex of a polygon in $\mathcal{H}(E)$, see Fig. 9. For an event point x it is not hard to see that we can decide which kind of polygon vertex it induces in constant time by looking at the information that the two boundary edges emanating from x yield. Hence, the sweep and the construction of $\mathcal{H}(E)$ can be done in $O(|\mathcal{H}(E)|)$ time, where $|\mathcal{H}(E)|$ denotes the complexity of $\mathcal{H}(E)$. Note that, by the first sweep, we get the boundary edges already in order. Recall that the first sweep processed $O(\tau^2 n^2)$ events, so together we have:

Theorem 5. *Given a set E of n entities moving in the plane over τ time steps, the set $\mathcal{H}(E)$ can be computed in $O(\tau^2 n^2 \log \tau n)$ time using $O(\tau^2 n^2)$ space.*

4.4. *Using a Topological Sweep*

If we examine the above sweep-line algorithms, we can observe that there is no need to process the event points strictly from left to right. As long as a well-defined sweep line is maintained, the order in which the event points are processed is not important. Also, we only need to keep the events and event points in memory that correspond to the current sweep line. This observation suggests the use of a topological sweep line introduced by Edelsbrunner and Guibas.[21] Recall that our arrangement contains degenerate cases, such as identical/parallel lines and multiple lines intersecting in the same point. Also note that our method processes all events that happen at the same event point at once. Hence, we cannot directly apply the results on topological sweeping[21]. We need a topological sweep approach that correctly processes all degeneracies; especially, the approach should recognise and handle at once all events that happen at the same event point. Exactly this can be done with an approach suggested by Rafalin et al.,[22] which is an extension of the work by Edelsbrunner and Guibas.[21] As a result the above bounds can be improved.

Theorem 6. *Given a set E of n entities moving in the plane over τ time steps, the set $\mathcal{H}(P)$ can be computed in $O(\tau^2 n^2)$ time using $O(\tau n + |\mathcal{H}(P)|)$ space.*

5. Hardness in the Continuous Model

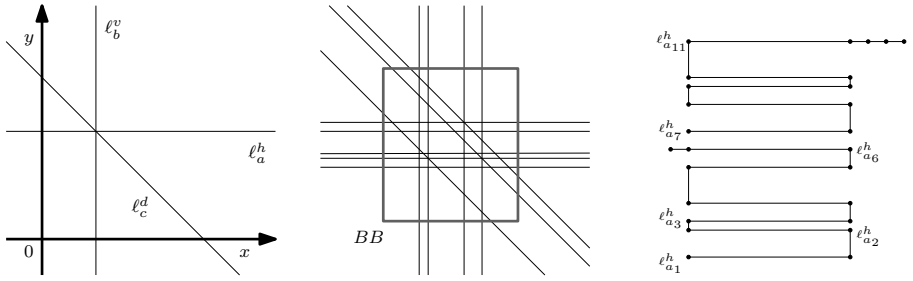
We argue that it is likely that every algorithm in the continuous model of the problem requires $\Omega(n^2\tau^2)$ time in the worst case. We will present a transformation from the problem 3-SUM2 to a special case of our problem.

Problem 3. (3-SUM2)
Given three sets A, B and C of integers with $|A| + |B| + |C| = N$, are there $a \in A$, $b \in B$ and $c \in C$ with $a + b = c$?

The 3-SUM2 problem is closely related to the classic 3-SUM problem and has been proven to be 3-SUM-hard.[23] This means that it is at least as hard as 3-SUM (with input size N) for which no subquadratic time algorithm has been found yet, which is an indication for an inherent hardness of the problems. For a weak model of computation a lower bound of $\Omega(N^2)$ for those problems exists.[24] The problem we will prove to be 3-SUM-hard is the following version of the popular places problem.

Problem 4. (POP-PLACE2)
Given the trajectories of n entities over τ time steps, is there a popular place of at least three entities with side length zero?

Let (A, B, C) be a 3-SUM2 instance and let τ be a fixed positive integer with $1 \leq \tau \leq N$. The transformation from 3-SUM2 to the POP-PLACE2 problem is as



(a) (b) (c)

Fig. 10: Examples illustrating the transformation for the lower bound: (a) three lines $\ell_a^h \in L^A$, $\ell_b^v \in L^B$ and $\ell_c^d \in L^C$, which intersect in one point; (b) the bounding box BB containing all intersections between all lines; (c) constructing trajectories of length $\tau = 12$ using the line segments of set L^A with $|L^A| = |A| = 11$.

follows. For each integer s of the input, we create a line $\ell_s : y = d_1 x + d_2$, where d_1 and d_2 depend on s and which set s belongs to. We sort the set $A = \{a_1, ..., a_{|A|}\}$ such that it is indexed in increasing order of its elements. We then create a set L^A of horizontal lines $L^A := \{\ell_a^h : y = a \mid a \in A\}$. We do the same for the sets B and C and create a set of vertical lines $L^B := \{\ell_b^v : x = b \mid b \in B\}$, and a set of diagonal lines $L^C := \{\ell_c^d : y = c - x \mid c \in C\}$. See Fig. 10(a), for an example of three such lines that intersect in one point.

Observation 2. There exist $a \in A$, $b \in B$ and $c \in C$ with $a + b = c$, iff the three lines ℓ_a^h, ℓ_b^v and ℓ_c^d intersect in one point.

Now, we transform these lines into line segments. We compute an axis-aligned bounding box BB whose interior contains all intersections between all lines, as is illustrated in Fig. 10(b). For each line, we then cut off everything outside BB, resulting in N line segments. For the sake of simplicity, we consider the sets L^A, L^B and L^C as sets of these line segments from now on. Note that no two line segments belonging to the same set can intersect.

As a last step we construct trajectories using the line segments. We will only describe this for the set of horizontal line segments $L^A = \{\ell_{a_1}^h, ..., \ell_{a_{|A|}}^h\}$, because for the other sets it is analogous. To create a trajectory of length τ, we start with $\ell_{a_1}^h$ and connect it to $\ell_{a_2}^h$ with a vertical connector line segment joining the right endpoints of $\ell_{a_1}^h$ and $\ell_{a_2}^h$. We then join the left endpoints of $\ell_{a_2}^h$ and $\ell_{a_3}^h$ by another vertical connector line segment. We continue to add line segments $\ell_{a_i}^h$ and vertical connector segments either on the left or right side until we obtain a trajectory of length τ or $\tau - 1$. If τ is even, we extend the current trajectory by adding a unit-length horizontal dummy line segment. With the remaining line segments in L^A, we create more trajectories, and we continue the process until all line segments of L^A

are used in trajectories. If, for the last trajectory that was created, there are not enough line segments in L^A for it to have length τ, we add unit-length horizontal dummy segments to it to achieve length τ. An example is shown in Fig. 10(c). Note that none of the added horizontal dummy segments and none of vertical connector segments intersects with any other line segment. The line segments of the sets L^B and L^C are used to create trajectories of length τ in a similar way.

The result of this transformation is a set T of n trajectories over τ time steps. From the above, the following lemma becomes evident.

Lemma 8. *There exist $a \in A$, $b \in B$ and $c \in C$ with $a + b = c$, iff there exists a popular place in T of at least three entities with side length zero.*

Note that in our trajectories T, we have N segments originating from A, B and C, at most N connector segments and at most $3\tau \leq 3N$ dummy segments. Hence, the size of T is $n\tau = \Theta(N)$. As the transformation holds for any τ and can be done in $O(N \log N)$ time, we can conclude with the following theorem.

Theorem 7. *Let T be a set of n trajectories over τ time-steps, for any $\tau \geq 1$. There exists no $o(n^2\tau^2)$ time algorithm to decide* POP-PLACE2 *with input T, unless there exists an $o(N^2)$ time algorithm to decide* 3-SUM2 *for an input of total size N.*

6. Concluding Remarks

Popular places are spatio-temporal patterns that can be interesting for biologists, marketing and surveillance analysts. Such patterns often come in two different settings: discrete and continuous. For both of them, we proposed algorithms to compute popular places, and we presented arguments that in the worst case it is unlikely that asymptotically faster algorithms can be derived.

In practice, however, it might be possible to have a faster algorithm. Recall, that in the continuous case we sweep an arrangement of $\Theta(\tau n)$ lines. This arrangement has $\Theta(\tau^2 n^2)$ intersections, but in practice we might expect that most of these intersections will not involve visible edges. A different approach would be not to sweep these lines, but the $\Theta(\tau n)$ visible edges. This sweep can be done in $O((\tau n + I) \log(\tau n))$ time, where I is the number of intersections among the line segments.[25] As a result, we have a running time which is smaller than $\Theta(\tau^2 n^2)$, if I is small, which we expect in practice. On the other hand, if I is large, sweeping the arrangement of the lines will be more efficient.

Throughout the paper, we have assumed that a popular place is defined by an axis-aligned square. Another natural way to look at this is to have a popular place defined by a disk. However, this problem variant has to be solved by more sophisticated sweep techniques, because our algorithms do not adapt to this setting in a straight-forward manner: in the discrete model our algorithms heavily rely on the fact that the entire area that is considered visited by a single entity is an axis-aligned rectilinear polygon which breaks down for disks. In the continuous model our algorithms were based on the fact that $P(\Lambda)$, the area visited by an entity Λ,

was a polygon as this enabled a sweep of the line arrangement. Again this result breaks down for disks as $P(\Lambda)$ will feature arcs then.

Acknowledgements

We would like to thank Hans Bodlaender, Jyrki Katajainen, Damian Merrick and Anh Pham for very useful discussions; and also many thanks to the reviewers of an earlier version of this article.

References

1. Wildlife tracking projects with GPS GSM collars. `http://www.environmental-studies.de/projects/projects.html`, 2006.
2. A. U. Frank. Socio-Economic Units: Their Life and Motion. In A. U. Frank, J. Raper, and J. P. Cheylan, editors, *Life and motion of socio-economic units*, volume 8 of *GISDATA*, pages 21–34. Taylor & Francis, London, 2001.
3. R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.
4. J. Gudmundsson, P. Laube, and T. Wolle. *Encyclopedia of GIS*, chapter Movement Patterns in Spatio-temporal Data, pages 726–732. Springer, 2008.
5. P. Laube, S. Imfeld, and R. Weibel. Discovering relative motion patterns in groups of moving point objects. *International Journal of Geographical Information Science*, 19(6):639–668, 2005.
6. P. Laube, M. van Kreveld, and S. Imfeld. Finding REMO – detecting relative motion patterns in geospatial lifelines. In P. F. Fisher, editor, *Developments in Spatial Data Handling: Proceedings of the 11th International Symposium on Spatial Data Handling*, pages 201–214, Berlin, 2004. Springer.
7. H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, 2006.
8. J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. In *Proceedings of the the 18th Annual International Symposium on Algorithms and Computation (ISAAC)*, volume 4835 of *Lecture Notes in Computer Science*, pages 763–775, Berlin Heidelberg, 2007. Springer-Verlag.
9. S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, 2000.
10. M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatio-temporal archives. *The VLDB Journal*, 15(2):143–164, 2006.
11. F. Verhein and S. Chawla. Mining spatio-temporal association rules, sources, sinks, stationary regions and thoroughfares in object mobility databases. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA)*, volume 3882 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2006.
12. N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *Proceedings of the 10th ACM SIGKDD International Conference On Knowledge Discovery and Data Mining*, pages 236–245. ACM, 2004.
13. G. Al-Naymat, S. Chawla, and J. Gudmundsson. Dimensionality reduction for long duration and complex spatio-temporal queries. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, pages 393–397. ACM, 2007.

14. M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle. Reporting leaders and followers among trajectories of moving point objects. *GeoInformatica*, 2007.
15. M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Computational Geometry—Theory and Applications*, 2007.
16. J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of the 14th ACM Symposium on Advances in GIS*, pages 35–42, 2006.
17. J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal sets. *GeoInformatica*, 11(2):195–215, 2007.
18. P. Gupta, R. Janardan, and M. Smid. *Handbook of Data Structures and Applications*, chapter Computational geometry: generalized intersection searching, pages 64–1 – 64–17. Chapman & Hall/CRC, 2005.
19. M. Ben-Or. Lower bounds for algebraic computation trees. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 80–86, New York, NY, USA, 1983. ACM Press.
20. J. Gudmundsson, T. Husfeldt, and C. Levcopoulos. Lower bounds for approximate polygon decomposition and minimum gap. *Information Processing Letters*, 81(3):137–141, 2002.
21. H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *Journal of Computer and System Sciences*, 38:165–194, 1989.
22. E. Rafalin, D. Souvaine, and I. Streinu. Topological sweep in degenerate cases. In *Proceedings of the 4th international workshop on Algorithm Engineering and Experiments, ALENEX 02*, volume 2409 of *Lecture Notes in Computer Science*, pages 155–156, Berlin, 2002. Springer.
23. A. Gajentaan and M. H. Overmars. n^2-hard problems in computational geometry. Technical Report 1993-15, Department of Coumputer Science, Utrecht University, The Netherlands, 1993.
24. J. Erickson and R. Seidel. Better lower bounds on detecting affine and spherical degeneracies. *Discrete & Computational Geometry*, 13:41–57, 1995.
25. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.