

Finding Popular Places

Marc Benkert¹, Bojan Djordjevic², Joachim Gudmundsson², and Thomas Wolle²

¹ Department of Computer Science, Karlsruhe University, Germany.
mbenkert@ira.uka.de

² NICTA* Sydney, Australia
{joachim.gudmundsson,bojan.djordjevic,thomas.wolle}@nicta.com.au

Abstract. Widespread availability of location aware devices (such as GPS receivers) promotes capture of detailed movement trajectories of people, animals, vehicles and other moving objects, opening new options for a better understanding of the processes involved. We investigate spatio-temporal movement patterns in large tracking data sets. Specifically we study so-called ‘popular places’, that is, regions that are visited by many entities. We present upper and lower bounds.

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, 10, 17]. 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’ in [19, 20]) among geospatial lifelines. The input is a set P of n moving point objects A_1, \dots, A_n whose locations are known at τ consecutive time-steps t_1, \dots, t_τ , that is, the trajectory of each object is a polygonal line 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 A 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 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. 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 discrete time steps are considered and the continuous case where the polygonal lines connecting the input points are considered. Recently it has been argued [6, 13] that the continuous model is becoming more important since 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 one second. A popular place in the two different models is defined as follows (see Fig. 1).

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

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

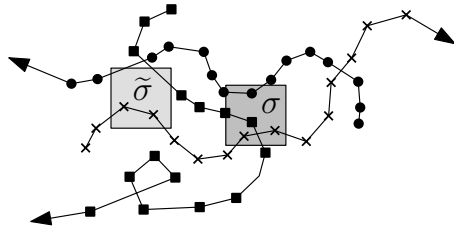


Fig. 1: An example where three entities Λ_1, Λ_2 and Λ_3 are traced during 16 time steps. For $k = 3$ the square $\tilde{\sigma}$ is a popular place only in the continuous model. While σ is a popular place for $k = 3$ in both the discrete and continuous model.

Recently 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, spatial or temporal nearest neighbours, see for example the work by Sältenis et al. [23] and Hadjieleftheriou et al. [18]. From a data mining perspective Verhein and Chawla [24] used association rule mining to detect patterns in spatio-temporal sets. 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. [21] studied periodic patterns, 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 [2, 3, 5, 14].

Precursory to this work Laube et al. [19, 20] proposed the REMO framework (Relative Motion) 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 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 the aim is to find or forecast a popular place (assuming the entities do not change their direction).

As mentioned in earlier work [5, 14, 15], specifying exactly which of the patterns should be reported is often a subject for discussion. 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 $\mathcal{H}(P)$ such that any axis-aligned unit square with centre in a polygon of $\mathcal{H}(P)$ is a popular place (report all).

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

In the Section 2 we present an algorithm for the discrete model, followed by an $\Omega(\tau n \log \tau n)$ time lower bound. In Section 4 we present an $O(\tau^2 n^2)$ time algorithm in the continuous model. Finally, in Section 5, we prove an $\Omega(n^2)$ lower bound in the continuous model.

2 A fast algorithm in the discrete model

We start with the discrete version of the problem. A set of n entities is traced over a period of τ time steps, generating τn points in the plane that correspond to the positions of the tracked entities. We will refer to the τn points as *input points*. The input parameter $k > 0$ defines the minimum number of entities defining a popular place, see Definition 1.

We design an algorithm that reports the popular place with the largest number of entities in $O(\tau n \log \tau n)$ time. In Section 2.4 we show how to modify it to produce more general output. This problem closely resembles the coloured range counting problem [16] 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 in [16]) *A set S of \bar{n} coloured points in the plane can be preprocessed into a data structure of size $O(\bar{n}^2 \log^2 \bar{n})$ 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 entities contained in a square of side length 1, i.e. 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 n coloured range counting queries where each query square has side length 2 and is centred at an input point. The algorithm requires $O(\tau^2 n^2 \log^2 \tau n)$ time and space. 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 that only uses $O(\tau n)$ space and runs in $O(\tau n \log \tau n)$ time.

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

For a start event, say that an input point p belonging to entity Λ enters str_ℓ , we update our data structures and check for the largest popular place located in str_ℓ that is visited by Λ . Such a popular place must obviously 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. 2a. If we check at every start event p for a largest popular place within R_p then we will find the maximum popular place. For an end event, say a point p belonging to Λ is about to leave str_ℓ , we simply remove it from the current data structures.

As described above, the general idea is to sweep a vertical strip str_ℓ of width 1 from left to right. Then, for each start event p the aim is to find a largest popular place containing p . This could be done by sweeping all the points 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 points 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 a set of trees such that given a y -interval $[a, b]$ (the y -coordinates of the top- and bottom side of R_p), we can find the y -value of the centre of a unit square that contains the largest number of different entities in $O(\log \tau n)$ time per query. 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 2.2 we show how we can merge this information into one tree T_{int} that can be queried and updated in $O(\log \tau n)$ time.

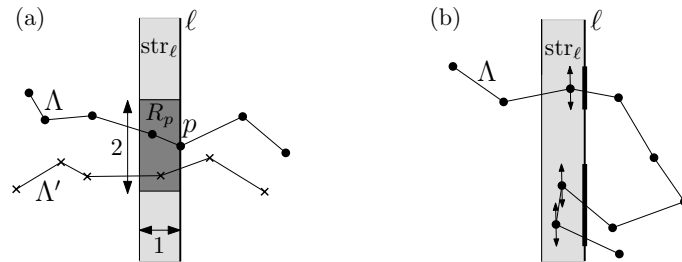


Fig. 2: (a) Finding the popular places in R_p visited by Λ . (b) The set I_A is indicated by the bold, solid line segments on ℓ .

2.1 One structure for each entity

We maintain a set of disjoint y -intervals I_A , for each entity A , such that at any event point during the sweep I_A contains exactly the y -intervals for which a square s in str_ℓ with centre in an interval in I_A contains a point of A , see Fig. 2b. The square containing a maximum popular place is a square whose centre is contained in a maximum number of such y -intervals.

Next we show how such a data structure can be maintained efficiently. We will maintain two trees B_A and T_A for each entity A . The tree B_A is a balanced binary search tree on the points of A currently within str_ℓ and ordered with respect to their y -coordinates. B_A can be queried and updated in $O(\log \tau)$ time using, for example, Red-Black trees (see e.g. [7]). The tree T_A will store the set I_A of intervals w.r.t. the current position of ℓ . The leaves of T_A store the endpoints of the intervals in I_A ordered on their y -coordinates. Each leaf also contains a pointer to the leaf in T_A containing the other endpoint. Inserting and deleting a new interval can be done in $O(\log \tau)$ time per update:

Assume we are about to process a start event $p_A = (x, y)$. Let $I = [y - 1/2, y + 1/2]$ and note that any unit square within str_ℓ with centre in I will contain p_A . The point is inserted into B_A and then a range query is performed in T_A that reports the intervals of I_A intersecting I . Since the intervals in I_A are disjoint and have length at least one, I may intersect at most two intervals in I_A . 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_A . 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.

In the case when str_ℓ is about to process an end event $p_A = (x, y)$, update the trees in a similar manner. Report the two adjacent neighbours $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ of p_A in B_A , and delete p_A . Assume without loss of generality that $y_1 < y < y_2$. Let I be as defined above and let I' be the interval in I_A containing I . We have to distinguish three cases:

1. If $|y_2 - y_1| \leq 1$ then we are done since I' does not change.
2. If $\min\{|y - y_1|, |y - y_2|\} > 1$ then $I' = I$ is deleted from T_A .
3. If $|y - y_1| > 1$ and $|y - y_2| \leq 1$ then I' is deleted from T_A and the interval $I' \setminus [y - 1/2, y_2 - 1/2]$ is inserted. The case when $|y - y_2| > 1$ and $|y - y_1| \leq 1$ is symmetric.

We denote the set of all trees T_A and B_A by \mathcal{T}^{ent} and \mathcal{B}^{ent} , respectively. Since the total number of events is $2\tau n$ the below corollary follows immediately.

Corollary 1. *Throughout the sweep, the sets \mathcal{T}^{ent} and \mathcal{B}^{ent} can be maintained in $O(\tau n \log \tau)$ time requiring $O(\tau n)$ space.*

2.2 Maintaining the status of the sweep

We store all intervals that are currently in str_ℓ in a balanced binary tree T_{int} . We will use T_{int} to perform the maximum popular place query for R_p . Let $\mathcal{I} = \bigcup_A I_A$. For simplicity we assume that all start and end points of intervals in \mathcal{I} are pairwise disjoint. The leaf set of T_{int} corresponds to the set of start and end points in \mathcal{I} ordered w.r.t. their y -coordinates.

Since there are τn points, T_{int} contains at most $O(\tau n)$ leaves and thus at most $O(\tau n)$ vertices at all times. During the sweep T_{int} is maintained as follows: every time a tree T_A is updated we perform the corresponding update operation in T_{int} . One update in T_A requires the deletion and insertion of a constant number of leaves in T_{int} . Thus, one update operation in T_A induces update operations in T_{int} that can be performed in $O(\log \tau n)$ time. Since the sweep conducts $2\tau n$ update operations, we have the following:

Lemma 1. *Throughout the sweep, the tree T_{int} can be maintained in $O(\tau n \log \tau n)$ time requiring $O(\tau n)$ space.*

Next, we show how we can store appropriate information in T_{int} in order to perform maximum popular place queries.

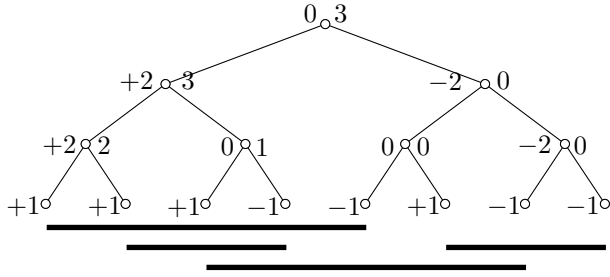


Fig. 3: The values sum (left) and max_{pre} (right) for the depicted tree T_{int} .

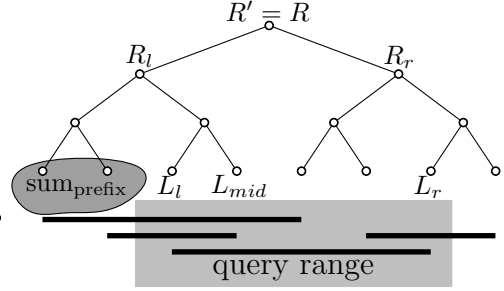


Fig. 4: Querying T_{int} for the maximum stabbing number $\text{max}_{l,r}$.

2.3 Extending T_{int} to allow for efficient queries

A point $p = (x, y)$ is said to *stab* an interval $[a, b]$ if $y \in [a, b]$. The *stabbing number* of p w.r.t. a set of intervals I is the number of intervals in I that p stabs. Note that for a start event p we have to find the point in $\ell \cap R_p$ having maximum stabbing number w.r.t. \mathcal{I} . Ideally we could store the number of intervals that a point corresponding to a leaf L in T_{int} stabs. 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, $\text{sum}(V)$ and $\text{max}_{\text{pre}}(V)$, with each vertex V in T_{int} , see Fig. 3. For leaves L we define $\text{sum}(L)$ and $\text{max}_{\text{pre}}(L)$ to be $+1$ if L corresponds to an interval start point and to be -1 otherwise. For inner vertices V let V_{left} and V_{right} denote the left and the right child of V , respectively. Then, sum and max_{pre} are defined as follows:

$$\text{sum}(V) := \text{sum}(V_{\text{left}}) + \text{sum}(V_{\text{right}}), \quad \text{and}$$

$$\text{max}_{\text{pre}}(V) := \max\{\text{max}_{\text{pre}}(V_{\text{left}}), \text{sum}(V_{\text{left}}) + \text{max}_{\text{pre}}(V_{\text{right}})\}.$$

Let $L_1(V), \dots, L_m(V)$ be the sequence of all leaves contained in the subtree rooted at V enumerated from left to right. The intuition of the definitions is that

$$\text{sum}(V) = \sum_{j=1}^m \text{sum}(L_j(V)) \quad \text{and} \quad \text{max}_{\text{pre}}(V) = \max_{1 \leq i \leq m} \sum_{j=1}^i \text{sum}(L_j(V)).$$

When performing an update operation in T_{int} , i.e. deleting or inserting a leaf L , updating sum and max_{pre} is only required on the path from L to the root. Hence, updating sum and max_{pre} takes $O(\log \tau n)$ time per update operation.

Lemma 2. *Throughout the sweep, the values sum and max_{pre} can be maintained in $O(\tau n \log \tau n)$ time requiring $O(\tau n)$ space.*

Recall the processing of the sweep. When we arrive at a start event $p = p_A$, we first perform the required updates in B_A and T_A , update T_{int} accordingly and then query T_{int} for the most popular place in R_p . Next, we show how the query can be done in $O(\log \tau n)$ time.

Let L_1, L_2, \dots be the set of leaves in T_{int} ordered from left to right. Each of the leaves in T_{int} is associated with the y -value of the stored start or end point. By performing two searches in T_{int} we can find the leftmost leaf L_l in T_{int} whose y -value is at least $y_p - 1/2$ and the rightmost leaf L_r whose y -value is at most $y_p + 1/2$ in $O(\log \tau n)$ time. This defines our query range within T_{int} : the goal is to find the leaf between (and including) L_l and L_r whose stored point stabs the maximum number of intervals among these leaves. We denote this maximum stabbing number by $\text{max}_{l,r}$. We have that $\text{max}_{l,r} = \max_{l \leq m \leq r} \{\sum_{j=1}^m \text{sum}(L_j)\}$. We claim that $\text{max}_{l,r}$ can be calculated by walking along the search paths from L_l and L_r , respectively, to the root R of T_{int} . We sketch how this is done. Note that the sketch comprises that a leaf with stabbing number $\text{max}_{l,r}$ can be found and reported in $O(\log \tau n)$ time.

Let R' be the least common ancestor of L_l and L_r and let mid , with $l \leq mid < r$, be the index such that L_{mid} is the rightmost leaf in the left subtree of R' , see Fig. 4. Set $\text{sum}_{\text{prefix}} := 0$, and consider the traversal of the search path from R to L_l . Every time the search path descends

into the right subtree of a vertex V add $\text{sum}(V_{\text{left}})$ to $\text{sum}_{\text{prefix}}$, see Fig. 4. When the search path reaches L_l we have $\text{sum}_{\text{prefix}} = \sum_{j=1}^{l-1} \text{sum}(L_j)$.

Let $\max_l = \max_{l \leq m \leq \text{mid}} \sum_{j=l}^m \text{sum}(L_j)$ and $\max_r = \max_{\text{mid}+1 \leq m \leq r} \sum_{j=\text{mid}+1}^m \text{sum}(L_j)$. Once we know \max_l and \max_r , the maximum stabbing number can be computed by $\max_{l,r} = \max\{\text{sum}_{\text{prefix}} + \max_l, \text{sum}(R'_{\text{left}}) + \max_r\}$. It remains to show how to compute \max_l and \max_r .

We compute \max_l by walking along the path from L_l to R'_{left} . Initially, we set $\max_l := \max_{\text{pre}}(L_l)$, and a helper variable $\text{sum}_{\text{seen}} := \text{sum}(L_l)$. Let $L_l = V^0, V^1, \dots, R'_{\text{left}}$ be the sequence of nodes that are traversed when walking from L_l to R'_{left} in T_{int} . Each time we encounter a vertex V^i having V^{i-1} as a left child we look for a new maximum in the right subtree, i.e. $\max_l := \max\{\max_l, \text{sum}_{\text{seen}} + \max_{\text{pre}}(V_{\text{right}}^i)\}$ and update sum_{seen} to $\text{sum}_{\text{seen}} := \text{sum}_{\text{seen}} + \text{sum}(V_{\text{right}}^i)$. It is not hard to see that in the end of the traversal \max_l holds the right value. There are $O(\log \tau n)$ vertices on the path from L_l to R' , and each vertex is processed in constant time. Thus, the time to compute \max_l is $O(\log \tau n)$.

In a similar fashion we compute \max_r . Here, we walk along the path from R'_{right} to L_r , let $R'_{\text{right}} = V^0, V^1, \dots, L_r$ be the sequence of traversed vertices. Initially, we set $\max_r := 0$ and $\text{sum}_{\text{seen}} := 0$. Each time we encounter a vertex V_i which is a right child of its parent we look for a new maximum in the left subtree, i.e. $\max_r := \max\{\max_r, \text{sum}_{\text{seen}} + \max_{\text{pre}}(V_{\text{left}}^{i-1})\}$ and update sum_{seen} to $\text{sum}_{\text{seen}} := \text{sum}_{\text{seen}} + \text{sum}(V_{\text{left}}^{i-1})$. Arriving at L_r we get \max_r by the final update $\max_r = \max\{\max_r, \text{sum}_{\text{seen}} + \max_{\text{pre}}(L_r)\}$.

Lemma 3. *Given T_{int} and a y -interval $[l, r]$ one can, in $O(\log \tau n)$ time, return the highest stabbing number $\max_{l,r}$ for any point in $[l, r]$ together with a point $p \in [l, r]$ having this stabbing number.*

Now we are ready to summarise the results obtained in this section.

Theorem 1. *Given a set P of n moving point objects 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.*

Proof. The proof follows by putting together Corollary 1, Lemma 1, Lemma 2 and Lemma 3. That is, all updates and queries can be done in $O(\log \tau n)$ per event, thus, in $O(\tau n \log \tau n)$ time in total. \square

2.4 Approximating and reporting all popular places

For a start event $p = p_A$ we have seen in Section 2.3 how we can detect a unit square in R_p that contains the maximum number k_{max} of different entities in the discrete model among all unit squares contained in R_p . If we now find that $k_{\text{max}} \geq k$ 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 P 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 of polygons $\mathcal{H}(P)$ such that any axis-aligned unit square with centre in a polygon of $\mathcal{H}(P)$ defines a popular place and each centre of a popular place is contained in $\mathcal{H}(P)$. Say that we process a start event p_A 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 makes the difficulty of the report-all problem.

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

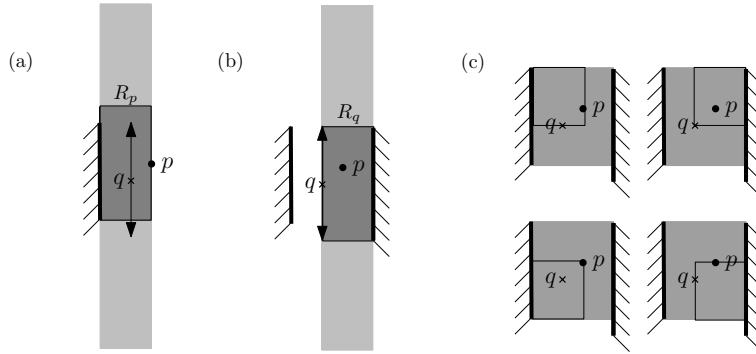


Fig. 5: (a) setting start flags, (b) setting end flags, and (c) the reported polygon (grey) in $\mathcal{H}(P)$.

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, meaning 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 exceeds 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 the polygons $\mathcal{H}(P)$ can simply 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 $\mathcal{H}(P)$. 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 simply have to connect the start and end flags accordingly to obtain the polygon set $\mathcal{H}(P)$.

Theorem 3. *Given a set P of n moving point objects in the plane the polygons $\mathcal{H}(P)$ 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 [4]. The decision version of the MIN-GAP problem also has an $\Omega(N \log N)$ lower bound [12].

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 step 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 $\tau n = \Theta(N)$. \square

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. In Section 5 it will be shown that there is an $\Omega(n^2)$ lower bound for the running time. Thus, the gap between our upper and lower bound is $O(\tau^2)$.

Consider the set P of n moving entities A_1, \dots, A_n in the plane, and let $\mathcal{H}(P)$ denote the minimal point set in the plane such that any axis-aligned unit square whose centre lies in $\mathcal{H}(P)$ intersects at least k trajectories. The output of the algorithm presented in this section will be the set $\mathcal{H}(P)$, i.e. a set of polygons. Note that in difference to the discrete case, the polygons $\mathcal{H}(P)$ are in general not rectilinear in the continuous case. In Section 4.1 we show how to construct an arrangement of lines such that $\mathcal{H}(P)$ can be constructed by a sweep line algorithm. The sweep itself is described in Section 4.2.

4.1 Constructing the line arrangement

Recall that a trajectory A is a polygonal path described by τ points, p_1, \dots, p_τ , and that two consecutive points p_i and p_{i+1} are connected by a straight-line segment e_i . Consider the following polygon construction: for a trajectory A sweep an axis-aligned unit square σ along the trajectory such that its centre moves on A as shown in Fig. 6a. The region swept by σ induces a polygon which we denote by $P(A)$, see Fig. 6b. Note that any axis-aligned unit square having its centre within $P(A)$, will intersect A . Thus, we can restrict ourselves to consider the centre locations for the popular-place defining squares.

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

Observation 1 $\mathcal{H}(P)$ consists of the regions having depth at least k w.r.t. $\{P(A_1), \dots, P(A_n)\}$.

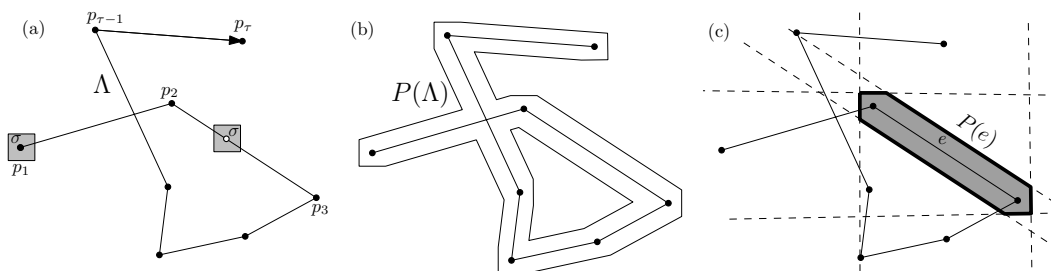


Fig. 6: (a) A trajectory Λ and the square σ that sweeps along Λ . (b) The polygon $P(\Lambda)$ obtained from the sweep, and (c) the polygon $P(e)$ and the lines l_i generated by $P(e)$ (dashed).

However, we do not explicitly compute the polygons $P(\Lambda)$ for each Λ , instead the following will be sufficient for our sweep. For each edge $e = (p_j, p_{j+1})$ of Λ consider the region swept along e by the square σ . This region is a polygon $P(e)$ with at most six edges, see Fig. 6c. For each edge f_i in $P(e)$ we construct an infinite line l_i that contains f_i . For each line l_i we store the start and end point of f_i on l_i , which side e lies to and Λ so that we later on know which entity this polygon belongs to. We refer to f_i as *visible segment* and to the rest of l_i as *invisible line*. The set of all lines constructed as above yields the line arrangement $\mathcal{A}(L)$ which we will sweep. Obviously, $\mathcal{A}(L)$ contains $O(\tau n)$ lines.

4.2 Sweeping the line arrangement

The general idea is to sweep the arrangement $\mathcal{A}(L)$ while simultaneously building $\mathcal{H}(P)$. Initially, we will 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. As the algorithm does not require a sweep by a straight-line we will later see that we can use a topological plane sweep by Edelsbrunner and Guibas [8] to improve the running time to $O(\tau^2 n^2)$ and space to $O(\tau n)$.

The algorithm will sweep the arrangement $\mathcal{A}(L)$ from left to right using a vertical line ℓ . The status of the sweep line is an array S of size $O(\tau n)$, which contains the current intersections between ℓ and the visible segments in $\mathcal{A}(L)$ ordered along ℓ . For each intersection between ℓ and a visible segment we store a *bracket* in S , with pointers between the segment and the bracket. A bracket is formally a tuple $\langle i, \text{type}, \text{level}, \text{count} \rangle$, where i is the entity number corresponding to the segment, *type* is *open* or *closed* depending on whether we are entering or exiting the polygon $P(e_i)$ as we go down ℓ . Note that the brackets will always come in open/closed pairs and that we can consider them as nested brackets. We keep track of matching pairs of brackets for the same entity. For example if going down ℓ we enter two polygons belonging to entity 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 with nesting level (*level*) equal to 1, and the second and third bracket are a different pair with nesting level 2. Note that the matching brackets do not have to come from the same polygon. *count* is the stabbing number immediately after the bracket. *count* only counts each entity once, so a point that stabs many polygons belonging to the same entity will only get a contribution of 1 to *count* from that entity. Therefore in $(())$ the second bracket will have *count* = 1 since the middle region only stabs one unique entity. An example is shown in Fig. 7.

The event points are the vertices of the arrangement $\mathcal{A}(L)$. Note that each vertex of a polygon $P(\Lambda)$ induces an intersection in $\mathcal{A}(L)$, thus the event points correctly includes all events for which the status changes.

The sweep line ℓ starts to the left of all vertices of $\mathcal{A}(L)$, which means initially S is empty. The moment at which the sweep line reaches an event point the status of the sweep line is updated.

When an event point x is reached the following process is performed. Recall that x is the intersection point between two consecutive lines along ℓ . We denote the two intersecting lines by l_1 and l_2 . For ease of description we assume that, to the left of x , l_1 lies above l_2 . We denote the

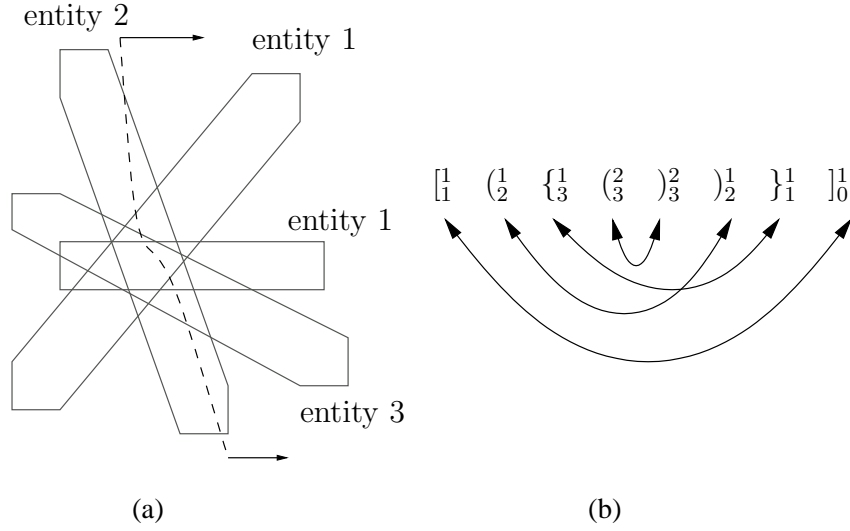


Fig. 7: (a) Example \mathcal{A}' (b) Brackets corresponding to the example. We use $(, [, \{$ for entities 1, 2 and 3 respectively. Superscripts and subscripts represent *level* and *count*. Matching brackets are shown with arrows.

half-lines of l_1 and l_2 to the right of x by l'_1 and l'_2 . The status of the sweep line ℓ just before reaching x is shown in Fig. 8.

Note that it could happen that three (or more) lines intersect in an event point. We can deal with this as if we processed three (or more) event points (in an appropriate order) for which only two lines intersect. When describing all the possible cases when processing x , we will for simplicity assume that only two lines of $\mathcal{A}(L)$ intersect in x .

Recall that each line l_i of $\mathcal{A}(L)$ corresponds to an edge f_i of $P(e)$, for some e of some A . Together with l_i , we store the two points on l_i that are the start and end points of the segment f_i , see Fig. 6c. Now, for l_i and the event point x we have to distinguish four cases:

- (i) x coincides with the start point of f_i
- (ii) x coincides with the end point of f_i
- (iii) x lies in the interior of f_i
- (iv) x and f_i are disjoint

For cases (i)–(iii) it is also of interest to which polygon $P(A)$ the line l_i is associated and to which side of l_i $P(A)$ lies. Let l_1 and l_2 be associated with the polygons $P(A_1)$ and $P(A_2)$, respectively. Note that not all combinations of the cases for l_1 and l_2 can occur, e.g. (i) and (ii) for l_1 can never be combined with (iii) or (iv) for l_2 , since a polygon cannot start/end in a single point. The six cases discussed below and illustrated in Fig. 8 are (modulo symmetry) the only cases that can indeed occur.

- (a) Both l_1 and l_2 are of type (iv). No changes are made to S .
- (b) l_1 is of type (iv), l_2 is of type (iii) and $P(A_2)$ lies above l_2 . No change to S because ℓ does not contain a bracket for invisible lines (l_1) so the position of the bracket corresponding to l_2 will not change.
- (c) l_1 is of type (ii), l_2 is of type (i) and $P(A_1)$ and $P(A_2)$ lie above l_1 and l_2 , respectively. Then $P(A_1) = P(A_2)$ and there is no change to S .
- (d) Both l_1 and l_2 are of type (i) and $P(A_1)$ lies below l_1 and $P(A_2)$ lies above l_2 . Again $P(A_1) = P(A_2)$. Insert a pair of brackets into S (which might require shifting parts of S by one position) and recalculate the *level* and *count* values for all the brackets in S . This can be done by traversing all the brackets in S , thus it requires $O(\tau n)$ time. For the symmetrically opposite

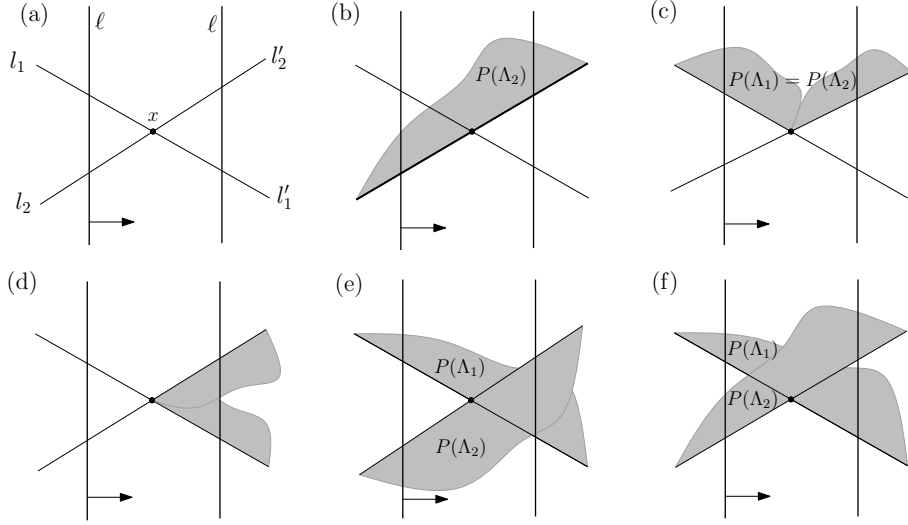


Fig. 8: The cases at an event point x during the line-arrangement sweep.

version of this case we remove the corresponding pair from S and recalculate values in S in the same way.

- (e) Both l_1 and l_2 are of type (iii), $P(\Lambda_1)$ lies above l_1 and $P(\Lambda_2)$ lies below l_2 . Swap the corresponding brackets in S (they will be adjacent) and update the two brackets in constant time. This will be described below.
- (f) Both l_1 and l_2 are of type (iii) and $P(\Lambda_1)$ and $P(\Lambda_2)$ lie above l_1 and l_2 , respectively. Same as (e), i.e. swap the corresponding brackets in S (they will be adjacent) and update the two brackets in constant time.

Since no updates are necessary in cases (a)-(c), and since we already established that case (d) requires $O(\tau n)$ time per update, it only remains to consider the cases (e) and (f). Note that the following description applies to both cases (e) and (f). We only need to look at the two brackets swapping positions since nothing else will change. Let a and b be the two brackets corresponding to lines l_1 and l_2 before the event point. Then we distinguish two cases: either they both belong to the same entity or not.

We start with the latter case: the brackets belong to different entities. In this case we can update S as follows. Let a' and b' refer to the new items in S (which are now in order b', a'). We set $a' \leftarrow a$, $b' \leftarrow b$ and then change the following values:

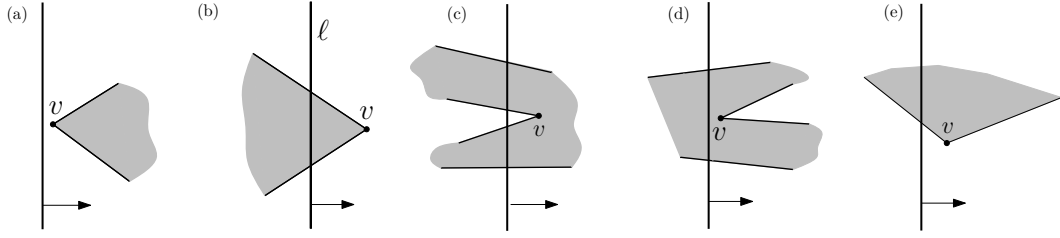
$$b'.count \leftarrow a.count + \Delta(b) - \Delta(a) \quad \text{and} \quad a'.count \leftarrow b.count,$$

where

$$\Delta(x) = \begin{cases} 0 & \text{if } x.level > 1 \\ +1 & \text{if } x.level = 1 \text{ and } x.type = open \\ -1 & \text{if } x.level = 1 \text{ and } x.type = closed \end{cases}$$

If the two brackets belong to the same entity then again we swap the brackets so we have b', a' in S and we update them as follows:

$$\begin{aligned} b'.count &\leftarrow a.count + \alpha, & a'.count &\leftarrow b.count \\ b'.level &\leftarrow a.level + \beta, & \text{and } a'.level &\leftarrow b.level + \beta \end{aligned}$$

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

where

$$\alpha = \begin{cases} +1 & \text{if } a.level = 1 \text{ and } a.type = closed \text{ and } b.type = open \\ 0 & \text{otherwise} \end{cases}$$

$$\beta = \begin{cases} 0 & \text{if } a.type = b.type \\ +1 & \text{if } a.type = closed \text{ and } b.type = open \\ -1 & \text{if } a.type = open \text{ and } b.type = closed \end{cases}$$

This only takes constant time so cases (e) and (f) can be done in constant time per event.

Lemma 4. *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. From above we have that the cases (a)-(c) and (e)-(f) can occur $O(\tau^2 n^2)$ times but can be handled in constant time. Thus, we only need to consider case (d). We can handle events (d) in $O(\tau n)$ time. However, there are only $O(\tau n)$ of these events. Therefore the total time is $O(\tau^2 n^2)$. There are at most $6\tau n$ elements in S because the sweepline intersects each line in the arrangement exactly once so we only need $O(\tau n)$ space. \square

It remains to argue how the polygon(s) $\mathcal{H}(P)$ are constructed. To simplify the description, we will describe the construction of $\mathcal{H}(P)$ by a postprocessing step. We may assume that the set of edges describing $\mathcal{H}(P)$ is known. More precisely, for all Λ we know the edges of $P(\Lambda)$, with one incident face having depth less than k and the other incident face having depth at least k . We call such edges *boundary edges*. Having information about all the boundary edges allows us to build $\mathcal{H}(P)$ by performing a second sweep traversing the set of boundary edges with a vertical sweep line ℓ . Note that by definition, no two boundary edges can intersect. An event point v 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 $\mathcal{H}(P)$, see Fig. 9. For an event point v 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 v yield. Hence, the sweep and the construction of $\mathcal{H}(P)$ can be done in $O(|\mathcal{H}(P)|)$, where $|\mathcal{H}(P)|$ denotes the complexity of $\mathcal{H}(P)$. Note that, by the line-arrangement sweep, we get the boundary edges already in order.

Recall that the line-arrangement sweep processed $O(\tau^2 n^2)$ events, so together we have:

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

If we examine the above sweep-line algorithm we can observe that there is no need to process the points from left to right. As long as a well-defined sweep line is maintained the order of the event points is not important. This observation suggests the use of a topological sweep line introduced by Edelsbrunner and Guibas [8]. As a result the above bounds can be improved.

Theorem 6. *Given a set S of n moving points in the plane, 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 [11]. 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 [9]. 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 follows. For each integer s of the input, we create a line $\ell_s : y = d_1x + d_2$, where d_1 and d_2 depend on s and which set s belongs to. We sort the set $A = \{a_1, \dots, a_{|A|}\}$ 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 Figure 10(a), for an example of three such lines that intersect in one point.

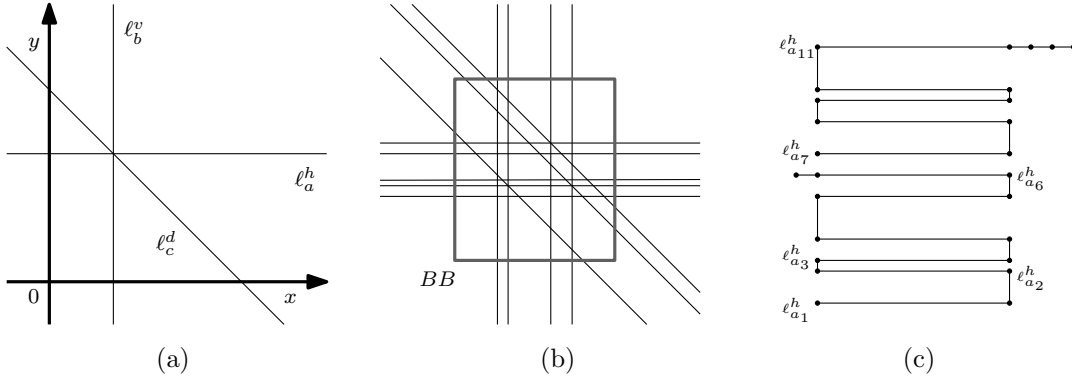


Fig. 10: Examples illustrating the transformation for the lower bound: (a) three lines $\ell_a^h \in L^A$, $\ell_b^v \in L^B$ and $\ell_c^d \in L^C$, which intersect in one point; (b) the bounding box BB containing all intersections between all lines; (c) constructing trajectories of length $\tau = 12$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

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 Figure 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 $L^A = \{\ell_{a_1}^h, \dots, \ell_{a_{|A|}}^h\}$, 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 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 Figure 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 5. *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, i.e. $3\text{-SUM2} \leq \text{POP-PLACE2}$.*

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. 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 .*

Acknowledgements

We would like to thank Jyrki Katajainen and Hans Bodlaender for very useful discussions.

References

1. Wildlife tracking projects with gps gsm collars. www.environmental-studies.de/projects/projects.html, 2006.
2. 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*. ACM, 2007.
3. M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle. Reporting leadership patterns among trajectories. In *Proceedings of the 22nd ACM Symposium on Applied Computing*. ACM, 2007.
4. M. Ben-Or. Lower bounds for algebraic computation trees. In *15th Annual ACM Symposium on Theory of Computation*, pages 80–86, 1983.
5. M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. In *Proceedings of the 14th European Symposium on Algorithms (ESA 2006)*, volume 4168 of *Lecture Notes in Computer Science*, pages 660–671. Springer, 2006.
6. H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *VLDB J.*, 15(3):211–228, 2006.
7. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.
8. H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *Journal of Computer and System Sciences*, 38:165–194, 1989.

9. J. Erickson and R. Seidel. Better lower bounds on detecting affine and spherical degeneracies. *Discrete & Computational Geometry*, 13:41–57, 1995.
10. 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.
11. A. Gajentaan and M. H. Overmars. n^2 -hard problems in computational geometry. Technical Report 1993-15, Department of Computer Science, Utrecht University, The Netherlands, 1993.
12. J. Gudmundsson, T. Husfeldt, and C. Levkopoulos. *Information Processing Letters*, 81(3):137–141, 2002.
13. J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. Submitted, April 2007.
14. J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *To appear in Proceedings 14th ACM Symposium on Advances in GIS*, 2006.
15. J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal sets. *To appear in Geoinformatica*, 2007.
16. 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.
17. R.H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.
18. M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatio-temporal archives. *The VLDB Journal*, 15(2):143–164, 2006.
19. 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.
20. 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.
21. 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.
22. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.
23. S. Sältenis, 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.
24. 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.