

Detecting Regular Visit Patterns

Bojan Djordjevic^{1,2}, Joachim Gudmundsson², Anh Pham¹, and Thomas Wolle²

¹ School of Information Technology, Sydney University, Sydney, Australia
{bojan}@cs.usyd.edu.au

² NICTA*** Sydney, Locked Bag 9013, Alexandria NSW 1435, Australia
{bojan.djordjevic, joachim.gudmundsson, thomas.wolle}@nicta.com.au

Abstract. We are given a trajectory \mathcal{T} and an area \mathcal{A} . \mathcal{T} might intersect \mathcal{A} several times, and our aim is to detect whether \mathcal{T} visits \mathcal{A} with some regularity, e.g. what is the longest time span that a GPS-GSM equipped elephant visited a specific lake on a daily (weekly or yearly) basis, where the elephant has to visit the lake *most* of the days (weeks or years), but not necessarily on *every* day (week or year).

During the modelling of such applications, we encountered an elementary problem on bit-strings, that we call LDS (LONGESTDENSESUBSTRING). The bits of the bitstring correspond to a sequence of regular time points, in which a bit is set to 1 iff the trajectory \mathcal{T} intersects the area \mathcal{A} at the corresponding time point. For the LDS problem, we are given a string s as input and want to output a longest substring of s , such that the ratio of 1's in the substring is at least a certain threshold.

In our model, LDS is a core problem for many applications that aim at detecting regularity of \mathcal{T} intersecting \mathcal{A} . We propose an optimal algorithm to solve LDS, and also for related problems that are closer to applications, we provide efficient algorithms for detecting regularity.

1 Introduction

Recent technological advances of location-aware devices and mobile phone networks provide increasing opportunities to trace moving individuals. As a result many different areas including geography, market research, database research, animal behaviour research, surveillance, security and transport analysis involve to some extent the study of movement patterns of entities [2, 8, 12]. This has triggered an increasing amount of research into developing algorithms and tools to support the analysis of trajectory data [9]. Examples are detection of flock movements [3, 5, 10], leadership patterns [4], commuting patterns [13–15] and identification of popular places [11].

In this paper, we introduce and study a problem called the *Longest Dense Substring* problem which originally stems from a problem concerning the analysis of trajectories, namely detecting regular visits to an area. Consider a trajectory obtained by tracking an elephant [1]; it is easy to detect which areas are important for the elephant, i.e. where it spends a certain amount of its time. However, ideally we would like to be able to detect if this area is visited with some regularity which might indicate that it could be used as a grazing or mating area during certain times of year. Another example occurs when tracking a person. Again, it is easy to detect which areas are important for her, such as home, work, local shopping areas and the cricket ground. But it would be interesting to find out if she goes to the cricket ground with some regularity, for example batting practice every Wednesday night. Note however, that the visits may be regular even though the cricket ground is not visited *every* Wednesday evening. It might be a regular event even though it only takes place 50% of all Wednesday evenings.

The above examples give rise to the following problem, as illustrated in Fig. 1. Given an area \mathcal{A} in some space and a trajectory \mathcal{T} , i.e. a time-stamped sequence of points in the space, one can generate a sequence of n time intervals $\mathcal{I} = \langle I_1 = [t_1^s, t_1^e], \dots, I_n = [t_n^s, t_n^e] \rangle$ for which \mathcal{T} intersects the area \mathcal{A} . Some of these intervals might be important for our application, while others are not

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

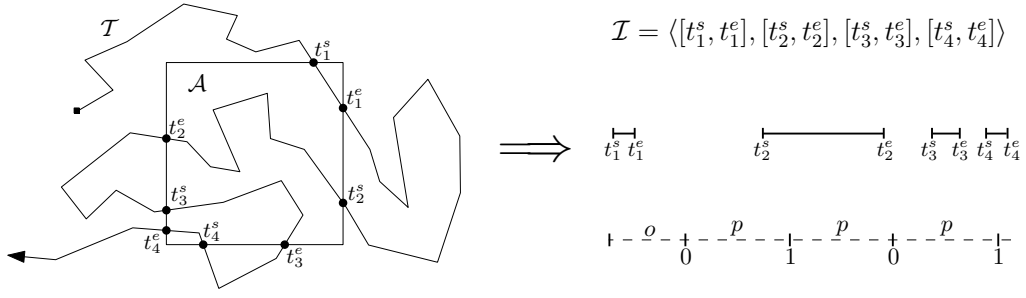


Fig. 1. A trajectory \mathcal{T} and an area \mathcal{A} is shown. From this, we derive the sequence of intervals \mathcal{I} from which we obtain the sequence of regular time points and also $s(o, p)$.

(e.g. a person’s visit to the cricket ground on a Sunday to watch a match is not interesting for detecting regular practice that occurs on Wednesdays). Hence, we look at whether \mathcal{T} intersects \mathcal{A} for a sequence of regular time points. For modelling regularity among the sequence of time points, we introduce two important notions: the *period length* p and the *offset* o . For fixed period length p and offset o (with $o < t_1^s + p$), we have a sequence of time points between t_1^s and t_n^e uniquely defined in the following way: All time points are equidistant with distance p and the first time point is at time o (e.g. if o is chosen to be a ‘Wednesday at 19:30’, and p equals 7 days, then all these time points correspond to all ‘Wednesday at 19:30’ for all weeks). Having the entire sequence of regular time points, the problem is to find the longest subsequence of consecutive time points such that \mathcal{T} intersects \mathcal{A} with high density (e.g. among the last three years, find the longest time span that a person was at the cricket ground on Wednesdays at 19:30). The density allows us to model exceptional occurrences of \mathcal{T} not intersecting \mathcal{A} (we still would like to detect the regular Wednesday’s practice, even though the person called in sick on some Wednesdays). We model the density as a value $c \in [0, 1]$ and require that \mathcal{T} intersects \mathcal{A} for at least $(100 \cdot c)\%$ of the times. To further formalise the above, we associate to each regular time point a value in $\{0, 1\}$. This value is 1 iff there exists an interval $I \in \mathcal{I}$ such that the time point is inside I , i.e. iff \mathcal{T} intersects \mathcal{A} at this time point (this can be extended to ‘approximate’ versions where the value is set to 1 if \mathcal{A} is visited approximately at this time point). These values *generate* the bitstring $s(o, p)$ for fixed p and o (for the example of batting practice, the bits indicate that a person has or has not been at the cricket ground on the corresponding Wednesday at 19:30). Now the aim is to compute a longest substring s^{opt} of $s(o, p)$ with the ratio of 1’s being at least c . We refer to such an optimal substring s^{opt} as a *longest dense substring*. Hence, a longest dense substring represents a longest time span where \mathcal{T} visited \mathcal{A} with regularity and high density (defined by o, p and c). This leads to the following problem. Let $length(s)$ denote the length of bitstring s , and $ratio(s)$ is the number of 1’s in s divided by $length(s)$. We denote the substring relation by \subseteq .

Problem: LDS (LONGESTDENSESUBSTRING)

Input: A string s and a real number $c \in (0, 1]$.

Question: What is a longest dense substring $s^{opt} \subseteq s$, i.e. what is a longest substring $s^{opt} \subseteq s$ with $ratio(s^{opt}) \geq c$?

LDS plays an important role for most of our applications for detecting regularity. Despite the fact that it seems fundamental, nothing is known about it to the best of the authors’ knowledge. Motivated from the analysis of trajectories, we have different variants for our applications. For an application, we may or may not be given a set of offsets and/or a set of period lengths. Thus together with the LDS-problem, we will also focus on finding the offsets and/or period lengths that generate the string that contains a longest dense substring. The perhaps given offsets and/or period lengths then generate an entire set of strings. Depending on the application and our approach, we can then compute a longest dense substring *for each string* or *over all strings* in this set of strings. The exact definitions are given in the appropriate section.

First, in Section 2, we propose an optimal algorithm to solve the LDS problem. This algorithm is used in Section 3, where we consider the case when both a set of possible offsets and a set of possible period lengths are given as input. For example if we know the first time of the period we would have a fixed offset, or set of possible offsets. In the most common scenario, one is given a set of possible period lengths; for example, we know that an elephant migrates to the same area on a monthly, quarterly or maybe yearly basis, or that a person goes to the cricket ground on a weekly, or biweekly, basis. This variant is considered in Sections 4 and 5. Finally, in Section 6, we consider the general version where nothing is known regarding the offset and only a lower bound is given for the period length. In the same section we also argue that, in the case when the period length may vary over a great range of discrete values, it seems hard to find a solution efficiently. We conclude the paper with some final remarks in Section 7.

2 Optimally Solving LDS

We present an algorithm to solve LDS for a string $s(o, p)$, for given period length p , offset o and density ratio c . Note that the length of the string $s(o, p)$ can be much larger than the number of intervals in \mathcal{I} . This motivates the study of a flavour of LDS that deals with compressed strings, where runs of 1s or 0s are compressed.

A compressed bit-string is easy to obtain from a normal bitstring or a set \mathcal{I} of intervals: runs of 0s or 1s are represented as pairs $(count, value)$; for example, a run (of maximal length) of x bits whose value is 0 is represented by $(x, 0)$. A compressed bitstring s^{comp} that arises from \mathcal{I} for an offset o and a period length p is a sequence of such pairs:

$$s^{comp}(o, p) = (count_1, value_1)(count_2, value_2)(count_3, value_3)\dots(count_k, value_k)$$

The number of bits in s^{comp} is $\sum_{i=1}^k count_i$, where the number k of pairs depends on the intervals and on the parameters o and p . The advantage of a compressed bit-string which is derived from a set of n intervals, is that its size is linear in n , i.e. $k = \mathcal{O}(n)$. To benefit from this advantage, we restate our problem and present an algorithm that works with compressed bitstrings. Note that the results of this section carry over to the problem with non-compressed bitstrings.

Problem: LDS^{comp}

Input: A compressed string s^{comp} represented by k pairs and a real number $c \in (0, 1]$.

Question: What is a compressed longest dense substring $s^{opt} \subseteq s^{comp}$ with $ratio(s^{opt}) \geq c$?

Our general approach to solve LDS^{comp} is to transform it in the following way. First, we consider a function $f_1(i)$ that is the number of 1's in s^{comp} from the first position to position i , see Figure 2(a). Next, we define a second function f_2 that is obtained from f_1 by skewing the coordinate system: $f_2(i) := f_1(i) - c \cdot i$, see Figure 2(b). Now we observe that a longest dense substring in s^{comp} corresponds to two indices $i_1 \leq i_2$, such that $i_2 - i_1$ is as large as possible and $f_2(i_1) \leq f_2(i_2)$. To compute such indices efficiently, we define a *lower left envelope* $LLE(i) := \min_{1 \leq j \leq i} f_2(j)$. In a symmetric way, we also define an *upper right envelope* $URE(i) := \max_{i \leq j} f_2(j)$. These envelopes are indicated in Figure 2(c). Finding two indices $i_1 \leq i_2$ where $i_2 - i_1$ is as large as possible and $f_2(i_1) \leq f_2(i_2)$ can be done by a 'walking along' these envelopes with two pointers i_1 and i_2 , initially both are at bit position 1. With i_2 we walk along URE as long as $URE(i_2) \geq LLE(i_1)$. We then walk forward with i_1 on LLE and repeat the process until both pointers reached the end of the envelopes. During this process, we keep track of the largest difference between these two pointers.

Theorem 1. *Let s^{comp} be a compressed bitstring represented by k pairs. There is an $\mathcal{O}(k)$ time algorithm to solve LDS^{comp} for s^{comp} , for a given c .*

Proof. The proof follows the general outline of the algorithm. The function f_1 is defined over the bits of s^{comp} . This function is piece-wise linear and changes from one line-segment to the next

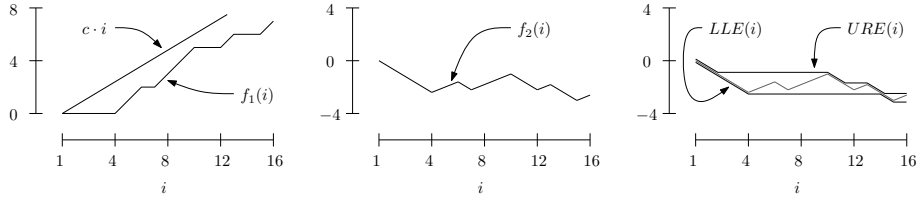


Fig. 2. Exemplary illustrations of $f_1(i)$, $f_2(i)$, $LLE(i)$ and $URE(i)$ for $c = 0.6$ and the compressed bit string $(4,0)(2,1)(1,0)(3,1)(2,0)(1,1)(2,0)(1,1)$, which equals the uncompressed bit string 0000110111001001 .

whenever the corresponding bit index changes from one pair of s^{comp} to the next.

$$f_1(i) = \sum_{j=1}^i \text{jth bit value in } s^{comp}$$

The problem LDS^{comp} can now be reformulated: we are looking for two indices $i_1 \leq i_2$ such that $i_2 - i_1$ is as large as possible and $\frac{f_1(i_2) - f_1(i_1)}{i_2 - i_1} \geq c$ (see Figure 2(a)). To ease the processing, we introduced another piece-wise linear function f_2 over the bits of s^{comp} . Graphically, f_2 is obtained from f_1 by skewing the coordinate system such that the line $c \cdot i$ will be horizontal (see Figure 2(b)).

$$f_2(i) = f_1(i) - c \cdot i$$

Now, solving LDS^{comp} means finding two indices $i_1 \leq i_2$ such that $i_2 - i_1$ is as large as possible and $f_2(i_1) \leq f_2(i_2)$. Recall that LLE and URE are defined as follows (where n' is the number of bits in s):

$$LLE(i) = \min_{1 \leq j \leq i} f_2(j) \quad \text{and} \quad URE(i) = \max_{i \leq j \leq n'} f_2(j)$$

LDS^{comp} can be solved by a single pass over the two envelopes. However, storing or processing the envelopes for every bit of s would result in an algorithm with running time $\Omega(n')$, but we aim for $\mathcal{O}(k)$. That is why we compute and store the envelopes in the following way, which is sufficient for the algorithm. We only describe this for LLE (it is similar for URE).

Since LLE is piece-wise linear it can be stored as a sequence of line-segments, which are represented by quadruples $(index_1, f_2(index_1), index_2, f_2(index_2))$. Such a quadruple corresponds to the line segment between the points $(index_1, f_2(index_1))$ and $(index_2, f_2(index_2))$. We compute all LLE -quadruples from left to right by passing over the pairs of the compressed bitstring s^{comp} . We start with the first quadruple, which is $(1, f_2(1), 1, f_2(1))$, i.e. it will represent just one point, and we initialise the current minimum to $f_2(1)$. Note that the current minimum corresponds to the current value of LLE . Also note that each pair $(count, value)$ contributes $(1 - c) \cdot count$ to f_2 if $value = 1$, and $-c \cdot count$ if $value = 0$. We keep track of that value by updating it incrementally for every pair. Now we process pairs until we encounter the pair $(count_h, value_h)$, such that h is the smallest index, such that $f_2(\sum_{j=1}^h count_j)$ is smaller than the current minimum. That means that the pair $(count_h, value_h)$ contains a bit index g , such that $f_2(g - 1) \geq \text{current minimum} > f_2(g)$. We compute this bit index g , which is possible in constant time. And we create and insert a new quadruple $(g, f_2(g), \sum_{j=1}^h count_j, f_2(\sum_{j=1}^h count_j))$ to LLE .

(One subtlety involved here is the following. For the pass over the envelopes described next, it is desirable if for each start- or end-point $(index, f_2(index))$ of a line segment in one envelope, there exists a line segment in the other envelope that has a start- or end-point $(index', f_2(index'))$ with $f_2(index) = f_2(index')$. This can be realised by breaking up the line segments accordingly, which can be done in $\mathcal{O}(k)$ time. Having the line segments with this property ensures that we will stop at all positions with $LLE(i_1) = URE(i_2)$ during the pass over the envelopes.)

To solve LDS^{comp} , we now make a single pass with two pointers i_1 and i_2 from left to right through the two envelopes LLE and URE . The pointer i_1 refers to the point $(i_1, LLE(i_1))$ and

i_2 refers to the point $(i_2, URE(i_2))$; initially, $i_1 = 1$ and $i_2 = 1$. We increment i_2 as long as $LLE(i_1) \leq URE(i_2)$. Incrementing i_2 means jumping from a start-point of a quadruple to the end-point of that quadruple, and then to the start-point of the next quadruple in the sequence, and so forth. When i_2 cannot be incremented without obtaining $LLE(i_1) > URE(i_2)$, we increment i_1 instead and continue the process. We keep track of and report the pointers for largest difference between i_1 and i_2 that was found in this way. All this can be done in time linear in the number k of pairs, because the size of the envelopes is linear in k .

For the correctness note that the algorithm will find two indices $i_1 \leq i_2$ such that $i_2 - i_1$ is as large as possible and $f_2(i_1) \leq f_2(i_2)$ (this is sufficient; see the argumentation above). The inequality $f_2(i_1) \leq f_2(i_2)$ is guaranteed by ensuring $LLE(i_1) \leq URE(i_2)$. Having $i_2 - i_1$ as large as possible follows from using the envelopes. \square

3 Given Offsets and Period Lengths

If we are given a sequence \mathcal{I} of n intervals, a set P of period lengths and a set O of offsets, we can use the results in Section 2 and run the algorithm of Theorem 1 on every string $s(o, p)$ that is generated by any combination of $p \in P$ and $o \in O$. We obtain the following lemma.

Lemma 1. *Let O be a set of offsets and P be a set of period lengths. In $\mathcal{O}(|O| \cdot |P| \cdot n)$ time, we can compute the overall longest dense substring over all $p \in P$ and $o \in O$. In the same time, we can also compute the longest dense substrings for all combinations of $p \in P$ and $o \in O$.*

Proof. There are $|O| \cdot |P|$ combinations of offsets and period lengths. For each possible combination, we compute the compressed bitstring $s(o, p)$ from the set \mathcal{I} of n intervals. This can be done in $\mathcal{O}(n)$ time. Since the length of the compressed bitstring is $\mathcal{O}(n)$, we now spend $\mathcal{O}(n)$ time to solve LDS, see Theorem 1. During this process, we can keep track of the longest substring encountered. \square

Compressed bitstrings have the advantage that their representation takes as much space as the set of intervals that was given as input. In situations where using uncompressed bitstrings does not become prohibitively expensive, we can use another approach, which results in certain cases in faster running times. For example, if our data (the start/end-points of intervals in \mathcal{I}) has a resolution of one day (i.e. disregarding the time of the day), then having period lengths and offsets as multiples of one day seems an obvious choice. Also, if the total number of days is small enough to be represented in an array, we can compute all longest dense substrings for all offsets for a fixed period length in linear time, as formalised in the following lemma.

Lemma 2. *Let O be a sorted sequence of offsets, P be a set of period lengths and q be a constant. If every period length $p \in P$ is a multiple of q , and if for every offset $o \in O$, it holds that the distance between t_1^s and o is a multiple of q , then we can compute all longest dense substrings for all $p \in P$ and $o \in O$ in $\mathcal{O}(n + |P| \cdot \text{length}(s(0, q)))$ time.*

Proof. As a first step, we compute the uncompressed bitstring $s(0, q)$ and store it in an array, which can be done in $\mathcal{O}(\text{length}(s(0, q)) + n)$ time. For each $p \in P$ and for each possible $o \in O$, we compute the uncompressed bitstring $s(o, p)$. Since we have the string $s(0, q)$ stored in an array, this can be done in $\mathcal{O}(\frac{\text{length}(s(0, q))}{p})$ time for any o . The time for computing the longest dense substring in $s(o, p)$ is linear in $\text{length}(s(o, p))$, hence $\mathcal{O}(\frac{\text{length}(s(0, q))}{p})$ time. Now note that the number of offsets that are valid for period length p , is at most p . Since these offsets are sorted, the total time we spend for all offsets is $\mathcal{O}(\text{length}(s(0, q)))$ per period length. \square

4 Given Period Lengths (Approximate)

Here, we are given a set of period lengths P together with the set of intervals \mathcal{I} . We propose an efficient algorithm that computes for *each* period length $p \in P$ *all* approximate longest dense substrings – one for each possible offset. Note that the approach as proposed in Section 3 to solve

the current problem will fail, because we do not know the offsets and even worse: on first sight, it seems we have to deal with an infinite number of offsets here. The general approach is to run an algorithm for every $p \in P$. For a fixed p , the algorithm is as follows: We cut the time-line and intervals between t_1^e and t_n^e into $\frac{t_n^e - t_1^e}{p}$ pieces of length p , see Figure 3(a). The last piece might have length less than p . We arrange all the pieces into a two dimensional structure by putting them above each other, see Figure 3(b).

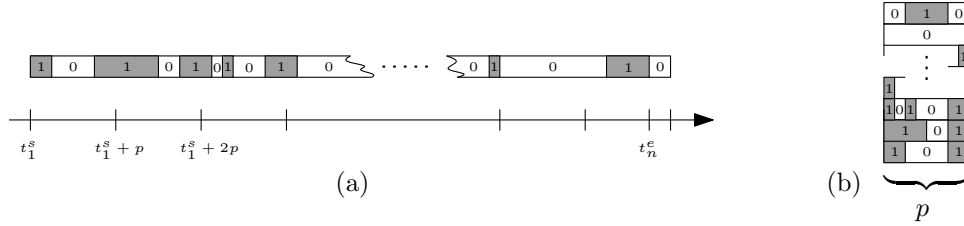


Fig. 3. (a) the time-line with indicated intervals (grey); (b) the time-line cut into pieces that are arranged underneath each other

Now we scan this structure from left to right with a vertical scan line. The scan line stops at *events*, which are moments during the scan, where the scan line leaves or enters any interval of any piece. At each event, we perform certain computations. We can interpret inside an interval as bit 1, and outside as bit 0. Hence, each vertical position o of the scan line cutting through the pieces represents a bit-string $s(o, p)$ of length $\frac{t_n^e - t_1^s}{p}$. Note that this bitstring does not change in between two events; and hence, we only have at most $2 \cdot n$ offsets to consider. If we apply the algorithm of Section 2 onto this string $s(o, p)$, we can compute an exact solution to the current problem in $\mathcal{O}(|P| \cdot (n \log n + n \cdot \frac{t_n^e - t_1^s}{p}))$ time and $\mathcal{O}(n + \frac{t_n^e - t_1^s}{p})$ space.¹ However, we aim for a better running time.

Remarkable is the following observation on the scan described above. From one event to the next, exactly one bit will flip in $s(o, p)$. (In the case that two or more intervals have the same endpoint modulo p , we define an order on them according to their absolute time.) In the just mentioned approach, we spend $\mathcal{O}(\frac{t_n^e - t_1^s}{p})$ time for each bitstring, even though this bitstring differs only by one bit from the previous bitstring. The ultimate goal would be an algorithm that consumes only constant time for each bit flip. It seems hard to derive such an algorithm that computes the exact solution. We will present an approximation algorithm that spends $\mathcal{O}(\log \frac{t_n^e - t_1^s}{p})$ time per bit flip. The main idea is that the bitstring $s(o, p)$ is represented in the leaves of a binary tree T . For a bitflip, we have to update the corresponding leaf of T and also information stored in nodes along the path from that leaf to the root of T . After that we can query the tree T to find an approximate solution for the current bitstring. Updating and querying can be done in logarithmic time, if T is balanced. In addition to T , we store in an event queue the $\mathcal{O}(n)$ events in sorted order. We have one event for each endpoint of an interval, each of which causes a bitflip.

For our approximation algorithm, we assume that we are given a constant real ε , $0 < \varepsilon < 1$. To clarify what we mean by an approximation let s_{opt} be an optimal solution to the LDS problem on a string s , i.e. let s_{opt} be a longest dense substring of s . A $(1 - \varepsilon)$ -approximate solution to this problem is a string $s' \subseteq s$ with $ratio(s') \geq (1 - \varepsilon) \cdot c$ and $length(s') \geq length(s_{opt})$. Note that s_{opt} and s' can be disjoint and that s' can be much longer than s_{opt} , and $\frac{ratio(s')}{ratio(s_{opt})}$ can be smaller than $(1 - \varepsilon)$.

¹ Note that if we use compressed bitstrings and do not explicitly compute this two-dimensional structure, we can prove the following bounds: $\mathcal{O}(|P| \cdot n^2)$ time and $\mathcal{O}(n)$ space.

The Data Structure T : The structure T is a binary tree with the leaves representing the bits of the current string s , see Figure 4. To ease the description, we assume w.l.o.g. that T is a complete binary tree.² We define leaves of T to have height-level 0 and the root of T has height-level $\log(\text{length}(s))$. Each node v_1 of the tree represents a substring s_{v_1} of s that contains exactly those bits of s that are represented in the leaves of the subtree rooted at v_1 . The length of s_{v_1} depends on the height-level $\text{level}(v_1)$ of v_1 , i.e. $\text{length}(s_{v_1}) = 2^{\text{level}(v_1)}$.

Each node in T will store four values and two links as described next. For each node v_1 , we store the ratio $\text{ratio}(s_{v_1})$ and the length $\text{length}(s_{v_1})$ of the corresponding string s_{v_1} . We also create *level-links* (indicated in Figure 4) that connect consecutive nodes in T on the same height-level into a doubly-linked list. On top of that, we attach two other values to every node v_1 . Since the tree is level-linked, we can easily access the nodes v_2, v_3, \dots on the same height-level as v_1 and to the right of v_1 in T , see Figure 4. Hence, we can easily compute the ratio and length of the substrings obtained by concatenating (\circ -operation) the substrings $s_{v_1}, s_{v_2}, \dots, s_{v_i}$. For $1 \leq i \leq m$, we obtain m such substrings, where $m := \lceil \frac{4}{\varepsilon} - 4 \rceil$. And among those we store at v_1 the length $\text{length}_{\max}(v_1)$ of the longest such substring with ratio at least $(1 - \varepsilon) \cdot c$. Note that $\text{length}_{\max}(v_1)$ is always a multiple of $\text{length}(s_{v_1})$. Additionally, we store at v_1 the maximum $\text{length}_{\max}^{\text{tree}}(v_1)$ of this value over all nodes of the subtree rooted at v_1 . Formally, we have:

$$\text{length}_{\max}(v_1) := \max_{1 \leq i \leq m} \{ \text{length}(s_{v_1} \circ \dots \circ s_{v_i}) \mid \text{ratio}(s_{v_1} \circ \dots \circ s_{v_i}) \geq (1 - \varepsilon) \cdot c \}, \text{ and}$$

$$\text{length}_{\max}^{\text{tree}}(v_1) := \max \{ \text{length}_{\max}(v) \mid v \text{ is in the subtree rooted at } v_1 \}$$

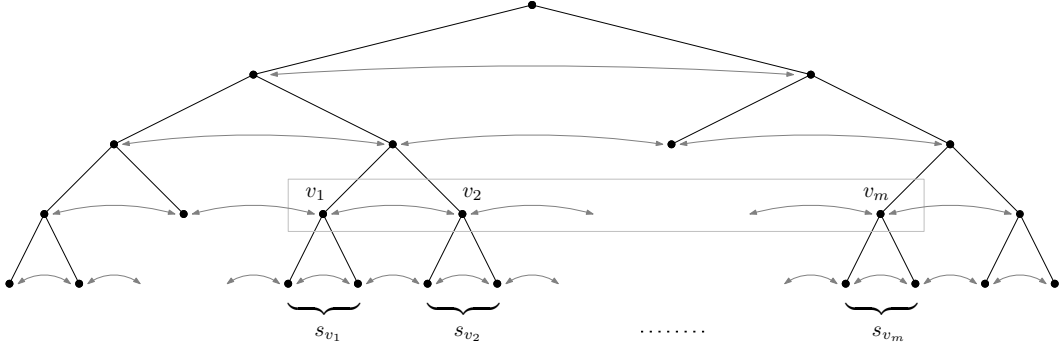


Fig. 4. The tree T for string s is shown (grey arrows are the level-links). Also the list of m nodes v_1, \dots, v_m on the same level as v_1 and starting with v_1 is indicated (grey rectangle), as well as the substrings corresponding to those nodes.

Using elementary techniques and basic results, we can conclude with the following lemma.

Lemma 3. *Constructing the tree T and the event queue can be done in $\mathcal{O}(\frac{1}{\varepsilon^2} \cdot \frac{t_n^e - t_1^s}{p} + n \log n)$ time and $\mathcal{O}(\frac{t_n^e - t_1^s}{p} + n)$ space.*

Once we have the tree T for string s , we will see that an optimal solution $s_{\text{opt}} \subseteq s$ to the LDS problem can be approximated by concatenating at most m substrings that correspond to consecutive tree-nodes on the same height-level.

Lemma 4. *Let lvl be the smallest height level in T such that $\text{length}(s_{\text{opt}}) \leq m \cdot 2^{lvl}$. Among the nodes of T on level lvl , let v_1 be the left-most node with $s_{v_1} \cap s_{\text{opt}} \neq \emptyset$, and let v_i be the right-most node with $s_{v_i} \cap s_{\text{opt}} \neq \emptyset$, see Figure 5. Then $\text{ratio}(s_{v_1} \circ \dots \circ s_{v_i}) \geq (1 - \varepsilon) \cdot c$ and $s_{\text{opt}} \subseteq s_{v_1} \circ \dots \circ s_{v_i}$.*

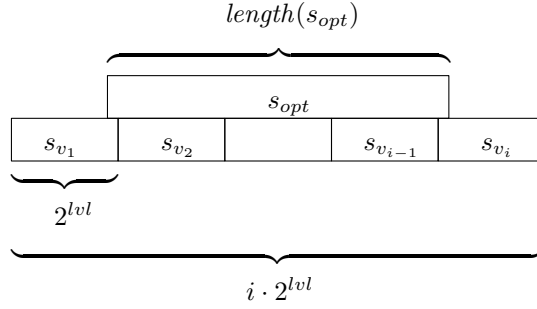


Fig. 5. The relations between the strings s_{opt} and $s_{v_1} \circ \dots \circ s_{v_i}$, $i \leq m$, as well as their lengths.

Proof. First, we observe the following, see Figure 5:

$$ratio(s) = \frac{\text{number of 1 in } s}{length(s)}, \quad \text{for any string } s \quad (1)$$

$$length(s_{v_1} \circ \dots \circ s_{v_i}) < length(s_{opt}) + 2 \cdot 2^{l_{v_1}}, \quad \text{because of the definition of } v_1 \text{ and } v_i \quad (2)$$

$$length(s_{opt}) > m \cdot 2^{l_{v_1}-1}, \quad \text{by our definition of } l_{v_1} \quad (3)$$

Now, we will prove that $ratio(s_{v_1} \circ \dots \circ s_{v_i}) \geq (1 - \varepsilon) \cdot c$:

$$\begin{aligned} ratio(s_{v_1} \circ \dots \circ s_{v_i}) &\geq \frac{ratio(s_{opt}) \cdot length(s_{opt})}{length(s_{v_1} \circ \dots \circ s_{v_i})}, \quad \text{using (1)} \\ &\geq \frac{ratio(s_{opt}) \cdot length(s_{opt})}{length(s_{opt}) + 2 \cdot 2^{l_{v_1}}}, \quad \text{using (2)} \\ &\geq \frac{ratio(s_{opt}) \cdot length(s_{opt})}{length(s_{opt}) + 4 \cdot \frac{length(s_{opt})}{m}}, \quad \text{using (3)} \\ &\geq \frac{ratio(s_{opt})}{1 + \frac{4}{m}} = \frac{m}{m+4} \cdot ratio(s_{opt}) \geq (1 - \varepsilon) \cdot c \end{aligned}$$

By the definition of $s_{v_1} \circ \dots \circ s_{v_i}$, it follows that $s_{opt} \subseteq s_{v_1} \circ \dots \circ s_{v_i}$, which completes the proof. \square

Note that the previous lemma also holds for any string with ratio at least c . Nevertheless, we formulated it with respect to an optimal solution s_{opt} , and we can conclude the existence of approximate solutions. For finding an approximate solution, recall that for each node v in T , we store the $length_{max}(v)$ values.

Lemma 5. *Let l_{v_1} and v_1, \dots, v_i be defined as in Lemma 4, and let s' be the string defined by $s' := s_{v_1} \circ s_{v_2} \circ \dots \circ s_{v_i}$ and $length(s') = length_{max}(v_1)$. Then $s' \supseteq s_{v_1} \circ \dots \circ s_{v_i} \supseteq s_{opt}$.*

Proof. From s_{opt} being an optimal solution and Lemma 4, we know that $ratio(s_{v_1} \circ \dots \circ s_{v_i}) \geq (1 - \varepsilon) \cdot c$ and $s_{v_1} \circ \dots \circ s_{v_i} \supseteq s_{opt}$. Now from the definition of $length_{max}$, it follows that $length_{max}(v_1) \geq length(s_{v_1} \circ \dots \circ s_{v_i})$. The lemma follows. \square

So far, we have considered the approximate longest dense substrings only for one string for the current position of the scan line (i.e. for the current offset). Now, we will move the scan line to the right until it either leaves an interval or enters a new interval. This results in one bitflip. We have to update T accordingly, and we have to perform a query to find an approximate solution for the new position of the scanline (i.e. for the new offset).

² We can append 0-bits to $s(o, p)$ to obtain a bit string whose length is a power of 2. This does not influence the solution, because an approximate solution that contains appended 0-bits, indicates another approximate solution without appended 0-bits.

Lemma 6. *For a single bitflip, T can be updated in $\mathcal{O}(\frac{1}{\varepsilon^2} \cdot \log \frac{t_n^e - t_1^s}{p})$ time.*

Proof. Following the path from the leaf, where the bit flipped, to the root, we may have to update the tree on all $\mathcal{O}(\log \frac{t_n^e - t_1^s}{p})$ levels; and on each level we have to recompute $\mathcal{O}(m)$ max-length values corresponding to consecutive nodes on that level. Each computation of such a value requires $m = \mathcal{O}(\frac{1}{\varepsilon})$ time. The time bound follows. \square

Updating the $length_{max}$ and $length_{max}^{tree}$ values is important, because these values can be used for navigating in the tree. Given the tree at any time, an approximate solution to the LDS problem can be found by simply following the path from the root of the tree to children with highest $length_{max}^{tree}$ value. From this and Lemmas 5 and 6, we derive the following theorem.

Theorem 2. *Let P be a set of period lengths and \mathcal{I} be a set of n intervals. We can compute approximate solutions for all longest dense substrings for all period lengths in P and all possible offsets in $\mathcal{O}(|P| \cdot (n(\log n + \frac{1}{\varepsilon^2} \log(t_n^e - t_1^s)) + (t_n^e - t_1^s)))$ time and $\mathcal{O}((t_n^e - t_1^s) + n)$ space.*

Remark 1. For a given period length and a fixed offset, the problem of reporting *all* approximate substrings with ratio $\geq (1 - \varepsilon) \cdot c$ and length at least a certain threshold can also be solved with the above approach. Whenever we query the tree, we follow all paths from the root to the children whose $length_{max}^{tree}$ value is at least the length threshold.

Remark 2. If we are not interested in computing approximate solutions for all longest dense substrings for *each* possible offset, but rather have the overall longest dense substring *over all* possible offsets, the above approach can be slightly modified, and the running time can be improved by replacing the factor $\frac{1}{\varepsilon^2}$ with $\frac{1}{\varepsilon}$. In this case, we can start with an initially approximate overall longest dense substring, and during all updates to the underlying bitstring and the tree T , we keep track of possible longer dense substrings. For this, we do not need the values $length_{max}(v)$ and $length_{max}^{tree}(v)$, and the procedure behind Lemma 6 can be modified in the following way. Instead of computing $\mathcal{O}(m)$ values at a node v , we only compute one value, which is the longest dense substring including s_v and having length at most $m \cdot length(s_v)$. This problem is similar to our basic problem introduced in Section 2, and in fact, having the *ratio* values of s_v and of v 's neighbours on the same height-level, the algorithm proposed in Section 2 can be modified to solve this in $\mathcal{O}(\frac{1}{\varepsilon})$ linear time.

5 Given Period Lengths (Exact)

As in the previous section we are given a set of period lengths P together with a sequence of intervals $\mathcal{I} = \langle I_1 = [t_1^s, t_1^e], \dots, I_n = [t_n^s, t_n^e] \rangle$, see Figure 6(a). We refer to the intervals in \mathcal{I} as *grey intervals* and to the intervals between them as *white intervals*. We give an efficient algorithm that computes a longest dense substring for *each* period length $p \in P$ over all possible offsets. The general approach is also the same as in the previous section. For every $p \in P$ cut the time-line and intervals between t_1^s and t_n^e into $\rho = \frac{t_n^e - t_1^s}{p}$ pieces of length p , denoted ℓ_1, \dots, ℓ_ρ . The last piece might have length less than p . We arrange all the pieces into a two dimensional structure by putting them above each other, as shown in Figure 6(b).

5.1 Transform into a weighted range query problem

We will show how the original problem can be transformed into the problem of finding the tallest grounded rectangle among a set of weighted points such that the total weight of the points within the rectangle is at least zero. A grounded rectangle is a rectilinear rectangle whose bottom left corner lies at $(0, o)$ for some offset o . The tallest rectangle is the one with the maximum height. In the following section it will be shown how this problem can be solved efficiently.

The part of a piece that is grey or white is called a *grey part* or *white part*, respectively, see Figure 6(b). Each part has an index i that indicates the position along the vertical axis, i.e. which

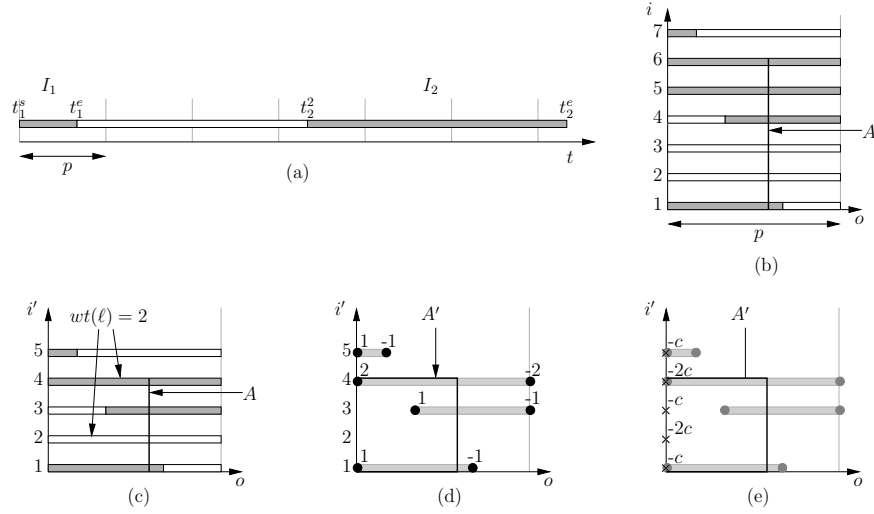


Fig. 6. (a) Input is a set of intervals and a period length p . (b) Cut off pieces of length p and place them above each other. (c) Compressed segments and best line. (d) Segments replaced by their endpoints, best line becomes best grounded rectangle. (e) The additional points.

piece it belongs to. The problem now is to find a longest vertical segment $A = \{o\} \times [i_1, i_2]$ that intersects at least $c(i_2 - i_1 + 1)$ grey parts.

Note that a grey or white interval that is much longer than p would create a large number of adjacent pieces that each contain just one grey or white part of length p , such as pieces 2 – 3 and 5 – 6 in Fig. 6(b). To overcome this problem we compress them into a single piece with just one grey or white part. The pieces at $i = 2$ and $i = 3$ (respectively at $i = 5$ and $i = 6$) get compressed into the piece at $i' = 2$ (respectively at $i' = 4$) in Fig. 6(c). When pieces are compressed, all pieces on top are moved downward to fill the gap. Each compressed piece ℓ' is assigned a weight $wt(\ell')$, equal to the number of pieces that were compressed. For each uncompressed piece ℓ' we set $wt(\ell') = 1$. Note that this compression is valid since an optimal pattern will always contain either all or none of the grey parts in a compressed piece.³

The obtained set of pieces is denoted $\Lambda = \langle \ell'_1, \dots, \ell'_{p'} \rangle$. We further assign to each grey part g the weight $wt(g)$, equal to the weight $wt(\ell')$ of the compressed piece ℓ' it belongs to.

When pieces with indices in the range $[i_1, i_2]$ are compressed into a single piece at i' we lose information about the vertical position of pieces so we define two mappings $\alpha_1(i')$ and $\alpha_2(i')$ that return the lowest and highest index of the compressed pieces, i.e. $\alpha_1(i')$ returns i_1 and $\alpha_2(i')$ returns i_2 .

Note that we can go from the input in Fig. 6(a) to the compressed version in Fig. 6(c) directly in $\mathcal{O}(n)$ time, rather than building the uncompressed structure in Fig. 6(b) in $\mathcal{O}(n + \frac{t_n^e - t_1^s}{p})$ time.

We can now rewrite the original problem as follows. Find an offset o and integers i'_1 and i'_2 such that $\alpha_2(i'_2) - \alpha_1(i'_1)$ is maximised and the sum of the weights over all grey parts intersected by the segment $\{o\} \times [i'_1, i'_2]$ is at least $c(\alpha_2(i'_2) - \alpha_1(i'_1) + 1)$.

Since there are only $\mathcal{O}(n)$ different offsets that might give different bitstrings we can use the approach presented in Section 3 to obtain the following observation.

Observation 1 *Given a period length and a set of n time intervals a longest pattern for any offset can be computed in $\mathcal{O}(n^2)$ time.*

In the rest of this section we will prove that the running time can be improved to $\mathcal{O}(n^{\frac{3}{2}} \log^2 n)$.

³ A longest pattern may contain only a part of the white parts in a compressed piece at the end but this can be trivially handled in constant time.

We replace each grey part l by two points $q_{left}(l)$ and $q_{right}(l)$. The point $q_{left}(l)$ has weight $wt(l)$ and is placed at the left endpoint of l and $q_{right}(l)$ has weight $-wt(l)$ and is placed at the right endpoint of l , as illustrated in Fig. 6(d). The set of weighted points is denoted A' .

We observe that the sum over all the weights of the grey parts in A intersected by a vertical segment A is equal to the sum over the points in A' in the rectilinear rectangle A' with top right corner at the top endpoint of A , bottom right corner at the bottom endpoint of A and unbounded to the left, see Fig. 6(c) and (d). The condition on A' is that $\sum_{q \in A'} wt(q) \geq c(\alpha_2(i'_2) - \alpha_1(i'_1) + 1)$.

Finally, for each piece ℓ' , insert into A' a point $(0, i')$, where i' is the index of ℓ' , as shown in Fig. 6(e) with weight $-wt(\ell') \cdot c$. The sum over the new points in the region A' will be equal to $-c(\alpha_2(i'_2) - \alpha_1(i'_1) + 1)$, so the constraint on A' becomes that the sum over all points in A' is at least zero. Note that the constraint on the sum is now independent of the height of A' , i.e. i'_1 and i'_2 .

We have now transformed the original problem into the following problem:

Problem 1. Given a set of weighted points in the plane find the grounded rectangle $[0, o'] \times [i'_1, i'_2]$ of total weight at least zero that maximises $\alpha_2(i'_2) - \alpha_1(i'_1)$.

5.2 Finding a longest pattern

Partition the points in A' with respect to their i' -coordinates into \sqrt{n} sets $A'_1, \dots, A'_{\sqrt{n}}$, each set having at most \sqrt{n} points, as shown in Fig. 7(a).

An optimal solution will either only contain points from a set A'_i , for some $1 \leq i \leq \sqrt{n}$, or contain points from several consecutive sets. In the former case we can find an optimal solution by simply running the quadratic time algorithm from Observation 1 on each set A'_i , to find the longest pattern that does not cross a boundary between the sets.

It remains to handle the case when the optimal solution contains points from several consecutive sets. Recall that the height of A in the uncompressed setting is $\alpha_2(i'_2) - \alpha_1(i'_1)$. For a bottom side fixed at i'_1 , finding the tallest rectangle is equivalent to finding one that maximises i'_2 . We do this for every value of i'_1 and at the end report the rectangle with the largest height in the uncompressed setting.

We first need to define a new problem. A point q dominates a point q' if and only if $q_x \geq q'_x$ and $q_y \geq q'_y$. Let $\text{dom}(q)$ be the set of points dominated by a point q . We define the dominance sum, denoted $\sigma(q)$ to be the sum of $wt(q')$ over all points q' dominated by q . Finally, the *Highest Valid Dominance Point*, or *HVDP*(x_1, x_2, S), is the point q with the following properties: $\sigma(q) \geq S$, $q_x \in [x_1, x_2)$, and q_y is maximised. The proof of the following lemma can be found in Section 5.3. BOJAN SAYS: Change to Appendix for conference version. ←

Lemma 7. *We can preprocess a set of n weighted points in $\mathcal{O}(n \log^2 n)$ time using $\mathcal{O}(n \log n)$ space such that HVDP queries can be answered in $\mathcal{O}(\log^2 n)$ time.*

Now, for each set A'_i , $1 \leq i < \sqrt{n}$, we build a HVDP data structure D_i on all the points in $A'_{i+1}, \dots, A'_{\sqrt{n}}$. We are going to search for grounded rectangles with the bottom edge in A'_i and the top edge using the structure D_i . Note that such a rectangle can be partitioned into a top left quadrant in A'_i and a bottom left quadrant in D_i , as shown in Fig. 7(b).

Let $o_1 < \dots < o_m$ be the ordered set of o -coordinates of the points in A'_i , where $m \leq \sqrt{n}$, and let I' be the set of the corresponding i' -coordinates. For a fixed $i'_1 \in I'$ we partition the horizontal line $i' = i'_1$ into $\mathcal{O}(\sqrt{n})$ half-open segments ($L_1 = [o_1, o_2) \times \{i'_1\}, \dots, L_{m-1} = [o_{m-1}, o_m) \times \{i'_1\}, L_m = [o_m, \infty) \times \{i'_1\}$), such as segment L in Fig. 7(b).

Let q be a point and $\text{dom}'(q_o, q_{i'})$ be the set of points q' in A'_i with $q'_o \leq q_o$ and $q'_{i'} \geq q_{i'}$, i.e. the set of points in the top-left quadrant from q . Note that $\text{dom}'(q_o, q_{i'})$ will be the same for any point q on the half-open segment $[o_a, o_{a+1}) \times \{i'_1\}$, and will therefore be equal to $\text{dom}'(o_a, i'_1)$.

For each segment L_j we want to find the tallest rectangle with the bottom right corner on L_j . Let S be the sum over points in $\text{dom}'(x_j, i'_1)$, which we can find using semigroup dominance searching BOJAN SAYS: Joachim, do you know who to cite for this? It's easy to do and we can even ←

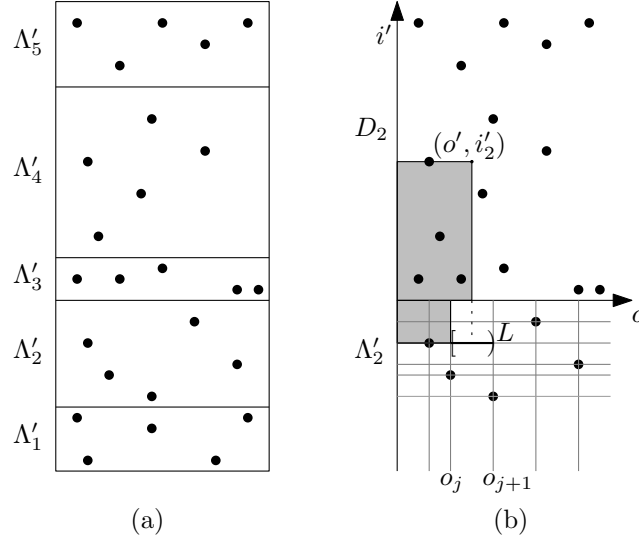


Fig. 7. (a) The points in Λ' are split into \sqrt{n} sets. (b) The grounded rectangle is split into two quadrants. Any point on L covers the same set of points in Λ'_i .

use HSMD but I couldn't find a published result. Now we query D_i for the highest point (o', i'_2) with $o' \in [o_j, o_{j+1})$ and whose dominance sum is at least $-S$. We calculate the uncompressed height of each rectangle, $\alpha_2(i'_2) - \alpha_1(i'_1)$, and report only the tallest one. This process is repeated for each $i'_1 \in I'$.

Theorem 3. *Let P be a set of period lengths and \mathcal{I} be a set of n intervals. For each period length in P , we can compute a longest dense substring over all possible offsets in $\mathcal{O}(|P| \cdot n^{\frac{3}{2}} \log^2 n)$ time and $\mathcal{O}(n \log n)$ space.*

Proof. In the preprocessing step of each of the $|P|$ iterations we use $\mathcal{O}(n)$ time and space to build the compressed representation.

In each iteration, we only use one HVDP data structure at a time so the total space used is $\mathcal{O}(n \log n)$. Running the quadratic algorithm on each Λ'_i takes $\mathcal{O}(\sqrt{n}^2)$, which is $\mathcal{O}(n^{\frac{3}{2}})$ in total. Constructing each of the $\mathcal{O}(\sqrt{n})$ HVDP data structures takes $\mathcal{O}(n \log^2 n)$ time, which is $\mathcal{O}(n^{\frac{3}{2}} \log^2 n)$ in total. For each Λ'_i we perform $\mathcal{O}(n)$ semigroup dominance searching queries and HVDP queries which takes $\mathcal{O}(\log^2 n)$. In total this amounts to $\mathcal{O}(\sqrt{n} \cdot n \cdot \log^2 n) = \mathcal{O}(n^{\frac{3}{2}} \log^2 n)$ time. \square

5.3 HVDP Problem

We first define the *horizontal segment maximum dominance*, or $\text{HSMD}(I)$, of a given half-open horizontal segment $I = [x_1, x_2) \times \{y_1\}$ to be the maximum dominance sum for any point on I . Formally,

$$\sigma(p) = \sum_{q \in \text{dom}(p)} wt(q)$$

$$\text{HSMD}(I) = \max_{p \in I} \sigma(p)$$

Problem 2. Preprocess a set P of n points so that we can efficiently find HSMD of a given half-open horizontal segment $I = [x_1, x_2) \times \{y_1\}$.

Lemma 8. *We can preprocess n points in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(n \log n)$ space such that HSMD queries can be answered in $\mathcal{O}(\log n)$ time.*

Proof. We build a balanced binary search tree T on the set of x values. We will then sweep the points from bottom to top and update T . After each sweep event we would like T to contain the information about dominance sums for every point on the sweep line. Since inserting a point would require updating $\mathcal{O}(n)$ values, we need to store the dominance sum indirectly. At each node v we store a real value $\delta(v)$, initially set to zero. Dominance sum at value x_0 will be represented by the sum of the δ -values on the path from the root to the leaf for x_0 . When a point p is reached by the sweep line we only need to increase by $wt(p)$ the $\mathcal{O}(\log n)$ δ -values for the nodes that cover the region $[p_x, \infty)$, i.e. we search for p_x in the tree and for every left turn at a node v we add $wt(p)$ to the δ -value of the right child of v .

At each node v in T we also store a real value $M(v)$, which is equal to $\delta(v)$ for the leaf nodes. For other nodes it is defined recursively as $M(v) = \delta(v) + \max\{M(v_l), M(v_r)\}$, i.e. it is equal to the maximum sum of δ -values on any path from v to a leaf.

When a point p is inserted, only the $\mathcal{O}(\log n)$ nodes on the path from the root to p_x are affected and can be updated by walking from the leaf to the root.

To find the maximum dominance sum in the region $[x_1, x_2)$, we initialise s and m to zero and search with x_1 and x_2 in T until we get to the split node v_{split} where the search paths split. For every node v from the root to v_{split} we add $\delta(v)$ to s . We set s_{left} and s_{right} equal to s . From the left child of v_{split} we continue to search for x_1 , and at each step we increase s_{left} in the same way as before. At each left turn we set $m = \max\{m, s_{left} + M(v_r)\}$. At the leaf node w for x_1 we set $m = \max\{m, s_{left} + M(w)\}$. Similar process is repeated with the right child of v_{split} and s_{right} . The only difference is that m is updated on right turns using $m = \max\{m, s_{right} + M(v_l)\}$. At the leaf node for x_2 m is not updated.

At the moment this only allows queries for the current position of the sweep line. However, since at each event point only $\mathcal{O}(\log n)$ values are changed, we can make T partially persistent using $\mathcal{O}(n \log n)$ space, while still keeping the $\mathcal{O}(\log n)$ query time [7]. To find the HSMD for an interval $I = [x_1, x_2) \times \{y_1\}$ we use y_1 as the time stamp in the persistent structure and then query it using the region $[x_1, x_2)$. If there are multiple points with the same y value, the time stamp refers to the last point inserted. \square

Recall that *Highest Valid Dominance Point*, or $HVDP(x_1, x_2, S)$, is the point (x_0, y_0) with maximal y_0 whose dominance sum is at least S and $x_0 \in [x_1, x_2)$.

We build a balanced binary search tree T on the set of points P with respect to their y values. For each node v , define $\mathcal{X}(v)$ and $\mathcal{Y}(v)$ to be the sets of x and y values in the canonical subset $\mathcal{C}(v)$ of v . We create an associated data structure $D(v)$ as a balanced binary search tree on $\mathcal{X}(v)$, as shown in Figure 8.

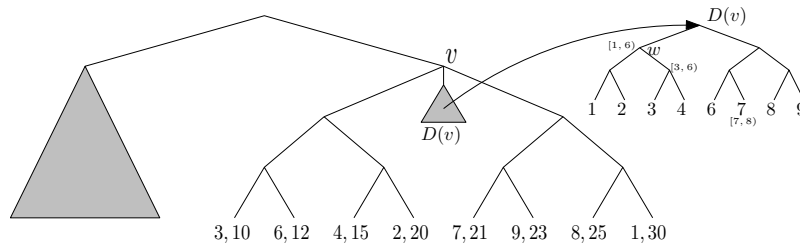


Fig. 8. The primary and secondary data structures. $[w_s, w_e)$ is shown for some nodes in the associated data structure.

Given a node v in T , we define the boolean function *Valid Dominance Point*, or $VDP(v, x_1, x_2, S)$ that evaluates to true if and only if there is a real value $y_0 \in \mathcal{Y}(v)$ and $x_0 \in [x_1, x_2)$ such that the point (x_0, y_0) has a dominance sum of at least S .

Lemma 9. *We can build a tree T on n points in $\mathcal{O}(n \log^2 n)$ time using $\mathcal{O}(n \log n)$ space such that VDP queries on T can be answered in $\mathcal{O}(\log n)$ time.*

Proof. Let w be a node in $D(v)$. We say that w represents the half-open range $[w_s, w_e)$, where w_s is the minimal value in $\mathcal{C}(w)$ and w_e is the next value to the right of $\mathcal{C}(w)$, as shown in Figure 8. Formally if w_m is the maximal value in $\mathcal{C}(w)$ then w_e is the minimal value in $D(v)$ that is greater than w_m . At each leaf w of D we store real values $\text{best}(w)$ and $\text{best-inside}(w)$.

$\text{best}(w)$ is the maximum dominance sum of a point (x_0, y_0) with $y_0 \in \mathcal{Y}(v)$ and $x_0 \in [w_s, w_e)$. $\text{best-inside}(w)$ is the maximum dominance sum, *restricted to the set $\mathcal{C}(v)$* , of a point (w_s, y_0) with $y_0 \in \mathcal{Y}(v)$.

Formally,

$$\begin{aligned} \text{best-inside}(w) &= \max_{y_0 \in \mathcal{Y}(v)} \sum_{p \in \mathcal{C}(v) \cap \text{dom}(w_s, y)} p_s \\ \text{best}(w) &= \max_{y_0 \in \mathcal{Y}(v)} \max_{x_0 \in [w_s, w_e)} \sum_{p \in \text{dom}(x_0, y_0)} p_s \\ &= \text{best-pre}(w_s, w_e, \lambda(v)) + \text{best-inside}(w) \end{aligned}$$

where $\lambda(v)$ is the first point below $\mathcal{C}(v)$ and $\text{best-pre}(x_1, x_2, y_1)$ is the maximum dominance sum for any point on the half-open segment $[x_1, x_2) \times \{y_1\}$, which is exactly the HSMD problem. The last equality follows from the fact that in $\mathcal{C}(v)$ any $x \in [w_s, w_e)$ gives the same dominance sum.

$\text{best-inside}(w)$ can be calculated for each leaf w in D , i.e. for every $x \in \mathcal{X}(v)$ in total time $\mathcal{O}(|\mathcal{C}(v)| \log |\mathcal{C}(v)|)$ using the same approach as for the HSMD problem. However here we sweep $\mathcal{C}(v)$ from left to right and after each update we record the maximum dominance sum for the current value of x . The data structure does not need to be made persistent so we can do this in $\mathcal{O}(|\mathcal{C}(v)|)$ space.

At each internal node w in $D(v)$ we store $\text{best} = \max\{\text{best}(w_l), \text{best}(w_r)\}$, where w_l and w_r are the children of w .

Note that if we partition the interval $[x_1, x_2)$ into a set of intervals $\mathcal{I} = \langle I_1, \dots, I_{k-1} \rangle = \langle [\tau_1, \tau_2), [\tau_2, \tau_3), \dots, [\tau_{k-1}, \tau_k) \rangle$, then $\text{VDP}(v, x_1, x_2, S)$ returns true if and only if $\text{VDP}(v, \tau_i, \tau_{i+1}, S)$ returns true for some $i \in \{1, \dots, k-1\}$.

To answer a VDP query we search with x_1 and x_2 in $D(v)$ until we get to the split node v_{split} . We then search for x_1 starting from the left child of v_{split} and whenever we turn left at a node w' we check if $\text{best}(w'_r) \geq S$. If it is then we stop and return true.

If x_1 is not present in $D(v)$ then the search path will take us to the leaf node w with $w_s < x_1 < w_e$. Since $D(v)$ does not contain any x values in the range (w_s, w_e) , the highest dominance sum in the range $[x_1, w_e)$ will be equal to $\text{best-inside}(w) + \text{best-pre}(x_1, w_e, \lambda(v))$. If this sum is at least S then we stop and return true.

The search path for y_2 is handled similarly. □

We can now prove the lemma stated in Section 5.

Lemma 7. *We can preprocess a set of n weighted points in $\mathcal{O}(n \log^2 n)$ time using $\mathcal{O}(n \log n)$ space such that HVDP queries can be answered in $\mathcal{O}(\log^2 n)$ time.*

Proof. To solve the HVDP problem in Lemma 7 we build a VDP data structure T . Let v be the root of T . If $\text{VDP}(v, x_1, x_2, S)$ returns false then we are done because there is no valid point with the desired dominance sum. Otherwise we walk from v to its right child v_r if $\text{VDP}(v_r, x_1, x_2, S)$ returns true, and to the left child v_l otherwise. We continue the process recursively until we reach a leaf node, which will contain the highest valid point with the desired dominance sum.

Since we perform $\mathcal{O}(\log n)$ VDP queries the total query time is $\mathcal{O}(\log^2 n)$. □

6 Nothing Given

Even if we are not given a set of possible offsets nor a set of possible period lengths, we still can tackle the problem of computing the *overall* longest dense substring over all period lengths and over all offsets. We also observe that in some cases it might be hard to develop efficient algorithms.

6.1 Solution

We propose an algorithm to solve this problem, where we make the following assumptions: the period length p can be any value in $[0, t_n^e - t_1^s]$, the offset can be any value between t_1^s and $t_1^s + p$, and the intervals are shorter than the period length p . Hence, a single visit can only contribute once to a bit-string. If no lower bound on the period length is given, then an arbitrarily good solution (w.r.t. the number of bits) can always be found by setting p to be infinitesimally small. Usually the lower bound is related to the length of the shortest interval between two visits. We believe that considering these constraints is meaningful from an application point of view.

We argue that we only have to check a polynomial (in n – the number of intervals) number of possible period lengths, which enables the use of the algorithms in Section 5 or Observation 1. To see this, consider an offset o and a period length p , such that $s(o, p)$ contains an overall longest dense substring. Any change of p and/or o by an infinitesimally small amount, may or may not change the string $s(o, p)$. Now, we repetitively change p and o by infinitesimal amounts until we obtain p' and o' , such that $s(o', p') = s(o, p)$ and any infinitesimal change of o' to obtain o'' results in $s(o', p') \neq s(o'', p')$. From this, we can conclude the following lemma. Recall that $s(o, p)$ is the string that contains an overall longest dense substring.

Lemma 10. *There exist o' and p' with $s(o', p') = s(o, p)$, and there exists a start-point t' of an interval in \mathcal{I} and an end-point $t'' > t'$ of an interval in \mathcal{I} , such that $t' - t_1^s \equiv t'' - t_1^s \equiv o' \pmod{p'}$.*

We conclude the existence of o' and p' such that $s(o', p')$ contains an overall longest dense substring, and with any infinitesimal change to o' we lose this property. Hence, we conclude the existence of a pair of interval start/end-points (namely t' and t'') that specify o' and p' . We also conclude that $i := \frac{t'' - t'}{p'}$ is an integer. The integer $i - 1$ is the number of bits in $s(o', p')$ between the bits corresponding to t' and t'' . Hence, for the period length p' , it holds that $p' = \frac{t'' - t'}{i}$.

We do not know the values of t' and t'' , and hence, we try all possible combinations of start-/end-points. For a fixed pair of start/end-points (i.e. for fixed t' and t''), we compute the period length $p' = \frac{t'' - t'}{i}$. However, we do not know the integer i , but we can bound it by: $1 \leq i \leq \frac{n}{c}$. The first inequality follows from $t' < t''$. To see the second inequality, note that any substring with ratio at least c has length at most $\frac{n}{c}$, because the number of 1's is at most n . Therefore, if $i > \frac{n}{c}$ then the bits corresponding to t' and t'' could not belong to the same dense substring. So for a fixed pair of start/end-points, we thus have $\frac{n}{c}$ potential period lengths to test. Since there are $\mathcal{O}(n^2)$ pairs of start/end-points, there are in total $\mathcal{O}(\frac{n^3}{c})$ potential period lengths.

Theorem 4. *There exist $\mathcal{O}(n)$ space, $\mathcal{O}(\frac{n^5}{c})$ running time and $\mathcal{O}(n \log n)$ space, $\mathcal{O}(\frac{n^{4\frac{1}{2}}}{c})$ running time algorithms to compute an overall longest dense substring, assuming that the period length p can be any value in $[0, t_n^e - t_1^s]$, the offset can be any value between t_1^s and $t_1^s + p$, and that the intervals are shorter than the period length.*

Proof. From the discussion above, we know that there is a pair t', t'' of start/end-points of intervals, such that this pair specifies a period length p' and an offset o' , such that $s(o', p')$ contains an overall longest dense substring. Therefore, we test all pairs. For each pair, we test all potential period lengths, and for each potential period length, we apply the algorithm in Section 5 (respectively, Observation 1) to give $\mathcal{O}(\frac{n^{4\frac{1}{2}}}{c})$ (respectively, $\mathcal{O}(\frac{n^5}{c})$) running time. \square

6.2 Hardness in Some Cases

It often is the case in algorithm design and complexity that a problem becomes harder if the solution or restricting parameter are required to be integers. In this section, we argue that this might also be true for the possible period lengths P . We have seen that if P is a set of period lengths then our algorithms have a running time that is linear in $|P|$. On the other hand, if P is a continuous range of period lengths then our algorithm to solve the problem in some cases has a running time that is independent of P . When solving LDS for a given set P of period lengths, we argue that it is hard to develop an algorithm whose running time is independent of $|P|$.

Observation 2 *Suppose we are given a set of period lengths and a fixed offset. Computing an overall longest dense substring over all period lengths is at least as hard as integer factorisation.*

To see this, let q be any integer that we aim to decompose into two integer factors. We construct an instance \mathcal{I} , O and P of LDS in the Discrete Model. The set of intervals is $\mathcal{I} = \{[0, 0], [q + \frac{1}{2}, q + \frac{1}{2}]\}$, the set of offsets is $O = \{\frac{1}{2}\}$ and the set of period lengths is $P = \{2, \dots, q - 1\}$. Note that the size of \mathcal{I} and O is constant, while the size of P is $\mathcal{O}(q)$. Hence, if all elements of P have to be given explicitly, then this construction takes $\mathcal{O}(q)$ time, but if the elements of P can be given implicitly as ‘all integers between 2 and $q - 1$ ’, then this construction can be done in constant time. (The latter is equivalent to setting the range of period lengths to $P = [2, q - 1]$, and requiring a period length to be an integer.) Now finding a period length $p \in P$, such that $s(o, p)$ has a substring of ratio greater than 0 means finding a divisor of q . Note that $o = \frac{1}{2}$. This would solve the integer factorisation problem for which no efficient algorithm is publicly known [6].

7 Concluding Remarks

In our applications, we look for regularities when a trajectory \mathcal{T} visit an area \mathcal{A} . To this end, we generate a bit string from \mathcal{T} and \mathcal{A} that reflects regularity by specifying a period length and an offset. Note that the here proposed approaches are not confined to regular visit patterns, but can be used for finding regularities in anything that can be expressed as a bit string.

During the course of this application driven research, we encountered the elementary problem, called LDS, of computing a longest dense substring, which is at the core of many applications. We provided an optimal algorithm to solve this basic LDS problem, see Theorem 1. To solve our more applied problems, we proposed efficient (approximation) algorithms that compute longest dense substrings, and hence, longest regular visit patterns for the cases where we are given a set of possible offsets and/or a set of possible period lengths.

It is often a topic for discussion to specify what our algorithms should produce as output. We chose to maximise the length of a substring, while the density has to be above a certain threshold. From an application point of view it might be a good choice to output all substrings of a string of which the length *and* the density are above certain thresholds. Some of our algorithms can be easily extended (with increased running time) to this *report all* version of our problems, while other algorithms require more research for such an extension.

When generating bit strings, we used a sequence of *time points* to define the bit values. It is also possible to use *time spans* instead (e.g. each bit represents an entire day and is set to 1, iff the person has been to the cricket ground on that day at *any* time). This is appropriate for many applications, and because our results also hold for this modelling, we can conclude the practical relevance of our algorithms.

Worthwhile directions for further research include the consideration of LDS and the related applications when we have streaming data. Also if we do not know the area(s) \mathcal{A} , we can consider the problem of computing the area(s) \mathcal{A} that are visited with regularity (perhaps specified by length and density thresholds).

Acknowledgements

We would like to thank Mark de Berg, Sergio Cabello and Damian Merrick for sharing their insights and for very useful discussions. Some of these discussions took place during the GAD-GET Workshop on Geometric Algorithms and Spatial Data Mining, funded by the Netherlands Organisation for Scientific Research (NWO) under BRICKS/Focus grant number 642.065.503.

References

1. Save the elephants. www.save-the-elephants.org.
2. Wildlife tracking projects with GPS GSM collars. <http://www.environmental-studies.de/projects/projects.html>, 2006.
3. 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.
4. M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle. Reporting leadership patterns among trajectories. In *Proceedings of the 22nd ACM Symposium on Applied Computing*, pages 3–7. ACM, 2007.
5. M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. Reporting flock patterns. *Computational Geometry—Theory and Applications*, 2007.
6. Richard P. Brent. Recent progress and prospects for integer factorisation algorithms. *Lecture Notes in Computer Science*, 1858:3+, 2000.
7. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.
8. 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.
9. J. Gudmundsson, P. Laube, and T. Wolle. *Encyclopedia of GIS*, chapter Movement Patterns in Spatio-Temporal Data. Springer, 2008. To appear.
10. 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.
11. J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal sets. *GeoInformatica*, 11(2):195–215, 2007.
12. R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.
13. Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604, New York, NY, USA, 2007. ACM Press.
14. 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.
15. M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 673–684, 2002.