# Data structures for detecting rare variations in time series

Caio Valentim, Eduardo S. Laber, and David Sotelo

Departamento de Informática, PUC-Rio, Brazil.

**Abstract.** In this paper we study, from both a theoretical and an experimental perspective, algorithms and data structures to process queries that help in the detection of rare variations over time intervals that occur in time series. Our research is strongly motivated by applications in financial domain.

## 1 Introduction

The study of time series is motivated by applications that arise in different fields of human knowledge such as medicine, physics, meteorology and finance, just to cite a few. For some applications, involving time series, general purpose spreadsheets and databases provide a good enough solution to analyse them.

However, there are applications in which the time series are massive as in the analysis of data captured by a sensor in a milliseconds basis or in the analysis of a series of quotes and trades of stocks in an electronic financial market. For these series, traditional techniques for storing data may be inadequate due to its time/space consumption. This scenario motivates the research on data structures and algorithms to handle massive time series [13].

The focus of our research is to develop data structures that allow the identification of variations on time series that are rare, i.e., that occur just a few times over a long time period.

Let $A = (a_1, ..., a_n)$ be a time series with $n$ values and let a time index be an integer belonging to the set $\{1, \ldots, n\}$. We use two positive numbers, $t$ and $d$, to capture variations in the time series $A$ during a time interval. More precisely, we say that a pair of time indexes $(i, j)$ of $A$ is a $(t, d)$-event if $0 < j - i \leq t$ and $a_j - a_i \geq d$.

We want to design efficient algorithms/data structures to handle the following queries.

- $AllPairs(t, d)$. This query returns all $(t, d)$-events in time series $A$.
- $Beginning(t, d)$. This query returns all time indexes $i$ such that there is a time index $j$ for which $(i, j)$ is a $(t, d)$-event in time series $A$.

The reason why we focus on the above queries is because we believe that they are basic queries for the study of rare variations in time series so that the techniques developed to address them could be extended to many other queries with the same flavor.

To be more concrete, we describe an application where an efficient processing of our queries turns out to be useful. The prices of two stocks with similar characteristics (e.g. belonging to the same segment of the industry) usually have a strong correlation. A very popular trading strategy in financial stock markets is to estimate an expected ratio between the prices of two similar stocks and then bet that the observed ratio will return to the estimated one when it significantly deviates from it [8].

A strategy designer may be interested to study if rare variations between the ratio of two stocks tend to revert to its mean (normal value). For that he(she) may specify a possibly long list of pairs $(t, d)$ and then study the behavior of the time series right after the occurrence of a $(t, d)$-event, for each pair $(t, d)$ in the list. A naive scan over a time series with the aim of looking for $(t, d)$-events, for a fixed pair $(t, d)$, requires $O(nt)$ time. In some stock markets (e.g. NYSE), where over 3000 stocks are daily negotiated, and a much larger number of pairs of stocks can be formed, a naive approach turns out to be very expensive. Thus, cheaper alternatives to detect $(t, d)$-events may be desirable.

Still in this context, given a the length $t$ of a time interval and a threshold $p$, one may ask for the minimum $d$ for which the number of $(t, d)$-events is at most $p$, that is, $(t, d)$-events are rare. If we want to process this kind of query for many time intervals $t$ and many thresholds $p$, an efficient processing of our primitive queries is also required.

To make the presentation easier we focus on positive variations ($d > 0$) in our queries. However, all the techniques developed here also apply for processing negative variations. The following notation will be useful for our discussion. The $\Delta$-value of a pair $(i, j)$ is given by $j - i$ and its deviation is given by $a_j - a_i$. We say that $i$ is the startpoint of a pair $(i, j)$ while $j$ is its endpoint.

**Our Contributions.** We develop and analyze algorithms and data structures to process queries AllPairs$(t, d)$ and Beginning$(t, d)$. The relevant parameters for our analysis are the time required to preprocess the data structure, its space consumption and the query elapsed time.

To appreciate our contributions it is useful to discuss some naive approaches to process our queries.

A naive algorithm to process $AllPairs(t, d)$ scans the list $A$ and for each startpoint $i$, it reports the time indexes $j$ in the interval $[i + 1, i + t]$ for which $a_j - a_i \geq d$. This procedure handles query $AllPairs(t, d)$ in $O(nt)$ time. A similar procedure also process $Beginning(t, d)$ in $O(nt)$ time.

Another simple approach to process $Beginning(t, d)$ is to use a table $T$, indexed from 1 to $n - 1$; the entry $T[t]$, for each $t$, has a pointer to a list containing the pairs in the set $\{(1, m_1), (2, m_2), \dots, (n - 1, m_{n-1})\}$, where $m_i$ is the time index of the largest value among $a_{i+1}, a_{i+2}, \dots, a_{i+t}$. Each of these $n-1$ lists is sorted by decreasing order of the deviations of their pairs. To process Beginning$(t, d)$, we scan the list associated with $T[t]$ and report the startpoint of every pair that is a $(t, d)$-event. The scan is aborted when a pair that has deviation smaller than $d$ is found. The procedure process $Beginning(t, d)$ in $O(k + 1)$ time, where $k$ is the size of the solution set. Along these lines, it is not difficult to design a data structure that also handles query $AllPairs(t, d)$ in $O(1+k)$ time. Though this kind of approach achieves optimal querying time, its space consumption could be prohibitive for most of practical applications since we have to store $\Theta(n^2)$ pairs.

The solutions we propose/analyze in this paper make an intensive use of data structures that support range minimum(maximum) queries(RMQ's). Let $V$ be a vector of real entries, indexed from 1 to $n$. A range minimum(maximum) query (RMQ) over $V$, with input range $[i, j]$, returns the index of the entry of $V$ with minimum(maximum) value, among those with indexes in the range $[i, j]$. This type of query is very well understood by the computational geometry community [1, 6]. In fact, there are known data structures of size $O(n)$ that support RMQs in constant time.

A simple but crucial observation is that the queries $AllPairs(t, d)$ and $Beginning(t, d)$ can be translated into multiple RMQ's (details are given in the next section). Thus, by using data structures that support RMQ's, we can handle both queries in $O(n+k)$ time, where $k$ is the size of the solution set. In fact, $Beginning(t, d)$ can be handled with the same time complexity without any preprocessing as we explain in Section 4.1. These solutions have the advantage of being very space efficient, they require $O(n)$ space. Their main drawback, however, is that they are not output sensitive in the sense that they require $\Omega(n)$ time regardless to the size of the solution set.

When the list of $(t, d)$ pairs for which we are interested to find/study the $(t, d)$-events is long, it may be cheaper, from a computational point of view, to preprocess the time series into a data structure and then handling the queries. However, the design of a compact data structure that allows efficient query processing seems to be challenging task. The key idea of our approach is to index a set of specials pairs of time indexes rather than all possible pairs. We say that a pair of time indexes $(i, j)$ is <u>special</u> with respect to time series $A$ if the following conditions hold: $i < j$, $a_i < \min\{a_{i+1}, \ldots, a_j\}$ and $a_j > \max\{a_i, \ldots, a_{j-1}\}$. If a $(t, d)$-event is also a special pair we say that it is a special $(t, d)$-event.

Let $S$ be the set of special pairs of the time series $A$. We propose a data structure that requires $O(|S| + n)$ space/preprocessing time and handles query $AllPairs(t, d)$ in $O(k+1)$ time, where $k$ is the size of the solution set. In addition, we propose a structure that requires $O(|S| \log |S|)$ space/preprocessing time and process $Beginning(t, d)$ in $O(\log n + f(\log f + \log t) + k)$ time, where $f$ is the number of distinct time indexes that are endpoints of special $(t, d)$-events and $k$ is the number of startpoints to be reported. Because of the symmetry between startpoints and endpoints and the fact that $f$ is the number of endpoints for a restricted class of $(t, d)$-events(the special ones), we expect $f$ to be smaller than $k$, that is, we expect to pay logarithmic time per solution reported. In fact, our experimental results are in accordance with this hypothesis.

The preprocessing time/space consumption of some of our solutions heavily depend on the number of special pairs of the time series under consideration. Thus, we investigate the expected value of this quantity. First, we prove that the expected number of special pairs of a random permutation of $\{1, \ldots, n\}$, is $n - \ln n$ with high probability. This result is interesting in the sense that the number of special pairs of a time series with distinct values is equal to the number of special pairs of the permutation of $\{1, \ldots, n\}$ in which the time series can be naturally mapped on. In addition, we evaluated the number of special pairs for 96 time series consisting of stock prices, sampled in a minute basis, over a period of three years, from the Brazilian stock market. We observed that the number of special pairs is, in average, 2.5 times larger than the size of the corresponding time series.

Finally, we performed a set of experiments to evaluate the performance of our algorithms. These experiments confirm our theoretical results and also reveal some interesting aspects of our solutions that are not explicited by the underlying theory.

**Related Work.** In the last decade, a considerable amount of research has been carried on to develop data mining techniques for time series as thoroughly discussed in a recent survey by Fu [7]. An important research sub-field pointed out by this survey asks for how to identify a given pattern (subsequence matching) in the time series [5, 12, 14, 10]. Though the problem studied here is related with this line of research, our queries do not fit in the framework proposed in [5] so that the available techniques do not seem to be adequate to handle them. In fact, one single query (e.g. $Beginning(t, d)$) that we deal with could be translated into a family of queries involving subsequences.

Our work is also closely related to the line of research that focus on identifying technical patterns in financial time series [11, 14]. Technical patterns as head &shoulders, triangles, flags, among others, are shapes that are supposed to help in forecasting the behaviour of stocks prices. Here we focus on a specific type of pattern that is useful for the study of mean reversion processes that occur in financial markets [3].

The problem of developing data structures to support Range Minimum(Maximum) Queries (RMQ problem), as previously explained, is closely related to our problems. The first data structure of linear size that can be constructed in linear time and answer RMQ queries in constant time was discovered by Harel and Tarjan [9]. Their structure, while a huge improvement from the theoretical side, was difficult to implement. After this paper, some others were published proposing some simpler, though still complex, data structures. A work from Bender et al.[1] proposed the first uncomplicated solution for the RMQ problem. The work by Fischer and Heun[6] improves the solution by Bender et. al. and also present an experimental study that compares some available strategies.

Finally, our problem has some similarities to a classical computational geometry problem, named the fixed-radius near neighbors problem. Given a set of points $n$ in a $m$-dimensional Euclidean space and a distance $d$, the problem consists of reporting all pairs of points within distance not greater than $d$. Bentley et al [2] introduced an algorithm that reports all $k$ pairs that honor this property in $O(3^m n + k)$. The same approach can be applied if a set $\{d_1, d_2, \ldots, d_m\}$ of maximum distances is given, one for each dimension. However, it is not clear for us if similar ideas can be applied to generate all pairs within distance *not* *less* than $d$ or, as in the case of our problem, not greater than $d_1$ in the first dimension and not less than $d_2$ in the second.

**Paper Organization.** Section 2 introduces some basic concepts/notations that will be used to explain our approach. In Section 3 and 4, we present, respectively, our data structures to handle both queries $AllPairs(t, d)$ and $Beginnings(t, d)$. In Section 5, we discuss some theoretical and experimental results related to the number of special pairs in a time series and we also describe the experiments executed to evaluate the peformance of our solutions. Finally, in Section 6, we draw our conclusion and discuss some possible extensions.

## 2 Basic Concepts

In this section we present observations/results that are useful to handle both queries $AllPairs(t, d)$ and $Beginning(t, d)$.

Our first observation is that it is easy to find all $(t, d)$-events that have a given time index $i$ as a starting point. For that, we need a data structure that supports range maximum queries over the input $A$ in constant time and with linear preprocessing time/space, as discussed in the related work section. Having this data structure in hands, we call the procedure `GenEventStart`, presented in Figure 1, with parameters $(i, i+1, i+t)$. This procedure, when executed with parameters $(i, low, high)$, generates all $(t, d)$-events with startpoint $i$ and with endpoint in the range $[low, high]$. First the procedure uses the data structure to find the time index $j$ with maximum value in the range $[low, high]$ of vector $A$. If the deviation of $(i, j)$ is smaller than $d$ then the search is aborted because there are no $(t, d)$-events that satisfy the required conditions. Otherwise, it reports $(i, j)$ and recurses on intervals $[low, j-1]$ and $[j+1, high]$.

The running time of the procedure is proportional to the number of recursive calls it executes. The execution flow can be seen as a binary tree where each node corresponds to a recursive call. At each internal node a new $(t, d)$-event is generated. Since the number of nodes in a binary tree is at most twice the number of internal nodes, it follows that the number of recursive calls is at most $2k_i$, where $k_i$ is the number of $(t, d)$-events that have startpoint $i$ and endpoint in the range $[low, high]$. Our discussion is summarized in the following proposition.

**Proposition 1.** *Let $k_i$ be the number of $(t, d)$-events that have startpoint $i$ and endpoint in the range $[low, high]$. Then, the procedure* `GenEventsStart`*(i,low,high) generates all these $(t, d)$-events in $O(k_i + 1)$ time.*

We shall consider the existence of an analogous procedure, which we call `GenEventsEnd`, that receives as input a triple $(j, low, high)$ and generates all $(t, d)$ events that have $j$ as an endpoint and startpoints in the range $(low, high)$.

The following lemma motivates the focus on the special pairs in order to handle queries $AllPairs(t, d)$ and $Beginning(t, d)$.

**Lemma 1.** *Let $(i, j)$ be a $(t, d)$-event. Then, the following conditions hold*

```
GenEventsStart($i$,$low$,$high$)
    If $high \geq low$
        $j \leftarrow RMQ(low, high)$
        If $a_j - a_i \geq d$   (*)
            Add $(i,j)$ to the list of $(t,d)$-events   (**)
            GenEventsStart($i$,$low$,$j$-1)
            GenEventsStart($i$, $j$+1,$high$)
        End If
    End If
```

**Fig. 1.** Procedure to generate $(t, d)$-events that have a given startpoint

i *there is an index $i^*$ such that $(i^*, j)$ is a $(t, d)$-event and $i^*$ is the startpoint of a special $(t, d)$-event.*

ii *there is an index $j^*$ such that $(i, j^*)$ is a $(t, d)$-event and $j^*$ is the endpoint of a special $(t, d)$-event.*

*Proof.* We just need to prove (i) because the proof for (ii) is analogous. Let $i^*$ be the time index of the element of $A$ with minimum $a_{i^*}$ among those with time indexes in the set $\{i, \ldots, j-1\}$. In case of ties, we consider the largest index. Because $a_{i^*} \leq a_i$, $i \leq i^* < j$ and $(i, j)$ is a $(t, d)$-event, we have that $(i^*, j)$ is also a $(t, d)$-event. Let $k^*$ be the time index of the element of $A$ with maximum value among those with time indexes in the set $\{i^*+1, \ldots, j\}$. The pair $(i^*, k^*)$ is both a special pair and a $(t, d)$-event, which establishes (i). $\square$

Our last result in this section shows that the set $S$ of special pairs of a given time series of size $n$ can be generated in $O(|S| + n)$ time. This is accomplished by calling the pseudo-code presented in Figure 2 with parameters $(1, n)$. First, the procedure uses a RMQ data structure to compute in constant time the time index $i$ with the smallest value in the range $[low, high]$. At this point, it concludes that there are no special pairs $(x, y)$ with $x < i < y$ because $a_x > a_i$. Then, it generates all special pairs with startpoint $i$ by invoking a modified version of `GenEventsStart`. Next, it looks for, recursively, for special pairs with time indexes in the range $(low, i-1)$ and also for those with time indexes in the range $(i + 1, high)$.

The procedure `ModifiedGenEventsStart` is similar to `GenEventsStart` but for the following differences: it verifies whether $a_j > a_i$ in line (*) and it adds the pair to the list of special pairs in line (**) of the pseudo-code of Figure 1

This last result is summarized in the following proposition.

**Proposition 2.** *Let $S$ be the set of special pairs of a time series $A$ of size $n$. Then, $S$ can be constructed in $O(|S| + n)$ time*

## 3 The query AllPairs$(t, d)$

We propose two algorithms to process query *AllPairs*$(t, d)$, namely, AllPairs-RMQ and AllPairs-SP. The first one relies on a data structure of size $O(n)$ and

```
GenSpecialPairs($low$,$high$)
    If $high \geq low$
        $i \leftarrow RMinQ(low, high)$
        ModifiedGenEventsStart($i, i+1, high$)
        GenSpecialPairs($low, i-1$)
        GenSpecialPairs($i+1, high$)
    End If
```

**Fig. 2.** Procedure to generate all special pairs

process $AllPairs(t, d)$ in $O(n + k)$ time, where $k$ is the size of the solution set. The second one requires $O(n+|S|)$ space and process $AllPairs(t, d)$ in $O(k)$ time, where $|S|$ is the number of special pairs of the time series under consideration.

### 3.1 Algorithm AllPairs-RMQ

AllPairs-RMQ builds, during its preprocessing phase, a data structure to support range maximum queries over $A$. The algorithms spends $O(n)$ time in this phase to build a structure of $O(n)$ size.

To process $AllPairs(t, d)$, it scans the list $A$ and for each time index $i$, it calls `GenEventsStart`$(i, i+1, \min\{i+t, n\})$. It follows from Proposition 1 that it process $AllPairs(t, d)$ in $O(n+k)$ time, where $k$ is the number of $(t, d)$-events in the solution set.

### 3.2 Algorithm AllPairs-SP

**Preprocessing Phase.** Let $S$ denote the set of special pairs of the time series $A = (a_1, ..., a_n)$. In it preprocessing phase, AllPairs-SP generates the set $S$ and stores its pairs, sorted by increasing order of its $\Delta$-values, in a vector $V$; ties are arbitrarily broken. Furthermore, it builds an auxiliary RMQ data structure $\mathcal{D}$ of size $O(|S|)$ to be able to answer the following query: given two values $t^\star$ and $d^\star$, report the subset of special pairs with $\Delta$-value at most $t^\star$ and deviation at least $d^\star$. Finally, it builds two auxiliary RMQ data structures $\mathcal{D}_{min}$ and $\mathcal{D}_{max}$ of size $O(n)$ to support, respectively, `GenEventsEnd` and `GenEventsStart` procedure calls over $A$.

To handle the query $AllPairs(t, d)$, we execute two phases. In the first one, we retrieve all special $(t, d)$-events from $V$ by using the data structure $\mathcal{D}$. Then, in the second phase, we use these special $(t, d)$-events and the structures $\mathcal{D}_{min}$ and $\mathcal{D}_{max}$ to retrieve the other $(t, d)$-events. The two phases are detailed below.

**Phase 1.** Let $k^*$ be the time index of the last pair in $V$ among those with $\Delta$-value at most $t$. [1] We use $\mathcal{D}$ to perform a set of maximum range queries to find

[1] It must be observed that $k^*$ can be found in $O(1)$ time, for a given $t$, by preprocessing $V$ with $O(n)$ time/space consumption.

the $(t,d)$-events in $V[1,..,k^*]$. More precisely, let $i$ be the index of $V$ returned by a range maximum query over $V$ with input range $[1,k^*]$. If the deviation of the pair stored in $V[i]$ is smaller than $d$ we abort the search because no other pair in $V[1,..,k^*]$ has deviation larger than $d$. Otherwise, we add this pair to a list $\mathcal{L}$ and we recurse on subvectors $V[1,..,i-1]$ and $V[i+1,..,k^*]$. At the end of this phase the list $\mathcal{L}$ contains all special $(t,d)$-events.

**Phase 2.** This phase can be split into two subphases:

**Phase 2.1.** In this subphase, we generate the set of distinct startpoints of all $(t,d)$-events. First, we scan the list $\mathcal{L}$ to obtain a list $E_{\mathcal{L}}$ containing the distinct endpoints of the special $(t,d)$-events in $\mathcal{L}$. This can be accomplished in $O(|\mathcal{L}|)$ time by keeping an $0-1$ vector to avoid repetitions among endpoints. Then, we invoke the procedure `GenEventsEnd`$(j, j-t, j-1)$, which is supported by the data structure $\mathcal{D}_{min}$, for every $j \in E_{\mathcal{L}}$. We keep track of the startpoints of the $(t,d)$-events generated in this process by storing them in a list $B$. Again, we use a $0-1$ vector to avoid repetitions among startpoints. It follows from item (ii) of Lemma 1 that list $B$ contains the startpoints of all $(t,d)$-events.

**Phase 2.2.** In this subphase, we generate all $(t,d)$-events that have startpoints in $B$. For that, we invoke `GenEventsStart`$(i, i+1, i+t)$ for every every $i \in B$. One must recall that `GenEventsStart` is supported by the data structure $\mathcal{D}_{max}$. The list of pairs of indexes $(i,j)$ returned by `GenEventsStart` will be the $(t,d)$-events we were looking for.

The results of this section can be summarized in the following theorem.

**Theorem 1.** *The algorithm AllPairs-SP generates all the $k$ solutions of the query $AllPairs(t,d)$ in $O(1+k)$ time and with $O(|S|+n)$ preprocessing time/space.*

## 4 The Query Beginning(t,d)

We propose three algorithms that differ in the time/space required to process *Beginning*$(t,d)$.

### 4.1 Beg-Quick

This algorithm scans the time series $A$ checking, for each time index $i$, whether $i$ is the startpoint of a $(t,d)$-event. To accomplish that it makes use of a list $Q$ of size $O(t)$ that stores some specific time indexes. We say that a time index $j$ is <u>dominated</u> by a time index $k$ if and only if $k > j$ and $a_k \geq a_j$. The algorithm mantains the following invariants: right before testing whether $i-1$ is a startpoint of a $(t,d)$-event, the list $Q$ stores all time indexes in the set $\{i, \ldots, (i-1)+t\}$ that are not dominated by other time indexes from this same set. Moreover, $Q$ is simultaneously sorted by increasing order of time indexes and non-increasing order of values.

Due to these invariants, the time index of the largest $a_j$, with $j \in \{i, \dots, (i-1)+t\}$ is stored at the head of $Q$ so that we can decide whether $i-1$ belongs to the solution set by testing if $a_{Head(Q)} - a_{i-1}$ is at least $d$. In the positive case, $i-1$ is added to the solution set. To guarantee the maintainance of these invariants, for the next loop, the algorithm removes $i$ from the head of $Q$ (if it is still there); it removes all time indexes dominated by $i+t$ and, finally, it adds $i+t$ to the tail of $Q$.

The algorithm runs in $O(n)$ time. To see that note that each time index is added/removed to/from $Q$ exactly once. The pseudo-code for the algorithm is presented in Figure 3. The first block is used to intitialize the list $Q$ so that it respects the above mentioned invariants, right before testing whether 1 belongs to the solution set.

```
Add time index $t+1$ to $Q$
**For** $k = t, ..., 2$
        If $a_k$ is larger than $a_{Head(Q)}$ **then** Add $k$ to the head of $Q$
**End For**
**If** $(1, head(Q))$ is a $(t,d)$-event **then** Add time index 1 to the solution set
**For** $i = 2, .., n-1$
        **If** $head(Q) = i$, remove $i$ from the head of $Q$
        **If** $i + t \leq n$
            Traverse $Q$ from its tail to its head removing every time index $j$ that is
            dominated by $i+t$
            Add $(i+t)$ to the tail of Q
        **End If**
        **If** $(i, head(Q))$ is a $(t,d)$-event **then** Add $i$ to the solution set
**End For**
```

**Fig. 3.** The Procedure Beg-Quick

### 4.2 Algorithm Beg-SP

**Preprocessing Phase.** First, the set $S$ of special pairs is generated. Then, $S$ is sorted by decreasing order of deviations. Next, we build an ordered binary tree $\mathcal{T}$ where each of its nodes is associated with a subinterval of $[1, n-1]$. More precisely, the root of $\mathcal{T}$ is associated with the interval $[1, n-1]$. If a node $v$ is associated with an interval $[i, .., j]$ then the left child of $v$ is associated with $[i, \lfloor (i+j)/2 \rfloor]$ and its right child is associated with $[\lfloor (i+j)/2 \rfloor + 1, j]$. Furthermore, each node of $\mathcal{T}$ points to a list associated with its interval, that is, if $v$ is associated with an interval $[i, j]$ then $v$ points to a list $L_{i,j}$. Initially, all these lists are empty.

Then, the set $S$ is scanned in non-increasing order of deviations of its special pairs. If a special pair $s$ has $\Delta$-value $t$ then $s$ is appended to all lists $L_{i,j}$ such that $i \leq t \leq j$. By the end of this process each list $L_{i,j}$ is sorted by non-increasing order of the deviations of its special pairs. Finally, we scan each list $L_{i,j}$ and remove a special pair whenever its endpoint has already appeared as an endpoint of another special pair.

The size of this structure is $O(\min\{n^2, |S| \log n\})$ because each special pair may occur in at most $\log n$ lists $L_{i,j}$ and the size of each of these $n$ lists is upper bounded by $n$ because for each endpoint at most one special pair is stored.

To handle a query $Beginning(t, d)$ we execute two phases. In the first one, we retrieve the endpoints of the special $(t, d)$-events. In the second phase, we use these endpoints to retrieve the startpoints of the $(t, d)$-events.

The correctness of this approach relies on item (ii) of Lemma 1 because if $i$ is a startpoint of a $(t, d)$-event then there is an endpoint $j^*$ of a special $(t, d)$-event such that $(i, j^*)$ is a $(t, d)$-event. As a consequence, the endpoint $j^*$ is found at Phase 1 and it is used to reach $i$ at Phase 2. In the sequel, we detail these phases

**Phase 1.** First, we find a set $K$ of nodes in $\mathcal{T}$ whose associated intervals form a partition of the interval $[1, t]$. If $n = 8$ and $t = 6$, as an example, $K$ consists of two nodes: one associated with the interval $[1, 4]$ and the other with the interval $[5, 6]$. It can be shown that it is always possible to find in $O(\log n)$ time a set $K$ with at most $\log t$ nodes. One way to see that is to realize that tree $\mathcal{T}$ is exactly the first layer of a 2D range search tree (see e.g. Chapter 5 of [4]).

Then, for each node $v \in K$ we proceed as follows: we scan the list of special pairs associated with $v$; if the current special pair in our scan has deviation larger than $d$, we verify whether it has already been added to the list of endpoints $E_f$. In the negative case we add it to $E_f$; otherwise, we discard it. The scan is aborted when a special pair with deviation smaller than $d$ is found. This step spends $O(|E_f| \log t)$ because each endpoint may appear in at most $\log t$ lists.

**Phase 2.** The second phase can be split into three subphases:

**Phase 2.1.** In the first subphase, we sort the endpoints of $E_f$ in $O(\min\{n, |E_f| \log |E_f|\})$ time by using either a bucket sort or a heapsort algorithm depending whether $|E_f|$ is larger than $n/ \log n$ or not. Let $e_1 < e_2 < \ldots < e_f$ be the endpoints in $E_f$ after applying the sorting procedure.

**Phase 2.2.** To generate the beginnings efficiently, it will be useful to calculate the closest larger predecessor (*clp*) of each endpoint $e_j \in E_f$. For $j = 1, \ldots, |E_f|$, we define $clp(e_j) = e_{i^*}$, where $i^* = \max\{i | e_i \in E_f \text{ and } e_i < e_j \text{ and } a_{e_i} \geq a_{e_j}\}$. To guarantee that the *clp*'s are well defined we assume that $E_f$ contains an artificial endpoint $e_0$ such that $e_0 = 0$ and $a_{e_0} = \infty$. As an example, let $E_5 = \{e_1, e_2, e_3, e_4, e_5\}$ be a list of sorted endpoints, where $(e_1, e_2, e_3, e_4, e_5) = (2, 3, 5, 8, 11)$ and $(a_{e_1}, a_{e_2}, a_{e_3}, a_{e_4}, a_{e_5}) = (8, 5, 7, 6, 4)$. We have that $clp(e_4) = e_3$ and $clp(e_3) = e_1$. The *clp*'s can be calculated in $O(|E_f|)$ time.

**Phase 2.3.** To generate the beginnings of the $(t, d)$-events from the endpoints in $E_f$, the procedure presented in Figure 4 is executed. The procedure iterates over all endpoints in $E_f$ and it stores in the variable $CurrentHigh$ the smallest time index $j$ for which all beginnings larger than $j$ have already been generated.

This variable is initialized with value $e_f - 1$ because there are no beginnings larger that $e_f - 1$. For each endpoint $e$, the procedure looks for new startpoints in the range $[\max\{e_i - t, clp(e_i)\}, CurrentHigh]$. It does not look for startpoints in the range $[e_i - t, clp(e_i)]$ because every startpoint in this range that can be generated from $e_i$ can be also generated from $clp(e_i)$ while the opposite is not necessarily true. Indeed, this is the reason why we calculate the *clp*'s – they avoid to generate a startpoint more than once. Next, the variable $CurrrentHigh$ is updated and a new iteration starts. This subphase can be implemented in $O(|E_f| + k)$ time, where $k$ is the number of beginnings generated.

Our discussion is summarized in the following theorem.

**Theorem 2.** *The algorithm Beg-SP generates all the $k$ solutions of the query $Beginning(t, d)$ in $O(\log n + f \cdot (\log f + \log t) + k)$ time, where $f$ is the number of distinct endpoints of special $(t, d)$-events. Furthermore, the algorithm employs a data structure of size $O(\min\{n^2, |S| \log |S| + n\})$ that is built in $O(|S| \log |S| + n)$ time in the preprocessing phase.*

As we have already mentioned, we expect $f$ to be smaller than $k$ due to the symmetry between startpoints and endpoints, and the fact that $f$ is the number of endpoints for a restricted class of $(t, d)$-events(the special ones) while $k$ is the number of startpoints of unrestricted $(t, d)$-events. In fact, we have observed this behavior, $f < k$, for more than thousand queries executed over real time series as described in Section 5.2.

```
CurrentHigh ← $e_f - 1$
**For** $i = f$ downto 1.
        GenEventsEnd  $(e_i, \max\{e_i - t, clp(e_i)\}, CurrentHigh)$
        CurrentHigh ← $\max\{e_i - t, clp(e_i)\}$
**End For**
```

**Fig. 4.** Procedure to generate the beginnings

### 4.3 Beg-Hybrid

Another way to process query $Beginning(t, d)$ is to combine the algorithms AllPairs-SP and Beg-SP as follows:

1. Execute the preprocessing phase of AllPairs-SP, skipping the construction of data structure $\mathcal{D}_{max}$ since it is not useful for processing $Beginning(t, d)$.
2. Apply Phase 1 of AllPairs-SP. At the end of this phase we have a list $\mathcal{L}$ containing all the special $(t, d)$ events.
3. Scan the list $\mathcal{L}$ to obtain the list $E_f$ containing the distinct endpoints of the specials $(t, d)$-events. Apply Phase 2 of Beg-SP.

The main motivation of this approach, when contrasted with Beg-SP, is the economy of a $\log n$ factor in the space consumption of the underlying data structure. Another motivation is the fact that it uses the same date structure employed by AllPairs-SP so that both $AllPairs(t, d)$ and $Beginning(t, d)$ can be processed with a single data structure. Its disadvantage, however, is that it requires $O(k \min\{t, f\})$ time, rather than $(\log n + f \cdot (\log f + \log t) + k)$ time, to process $Beginning(t, d)$, where $k$ is the size of the solution set and $f$ is the number of distinct endpoints of special $(t, d)$-events.

The correctness of Beg-Hybrid follows directly from the correctness of both AllPairs-SP and Beg-SP.

## 5 Experimental Work

### 5.1 On the number of special pairs

Since the preprocessing time and space of some of our data structures depend on the number of special pairs, we dedicate this section to discuss some theoretical and practical results related to this quantity.

Clearly, the number of special pairs of a time series is at least 0 and at most $n(n-1)/2$, where the lower(upper) bound is reached for a decreasing(increasing) series. The following proposition shows that the expected number of special pairs for a random permutation of the $n$ first integers is $n - H_n$. This result is interesting in the sense that the number of special pairs of a time series, with distinct values, is equal to the number of special pairs of the permutation of $\{1, \ldots, n\}$ in which the time series can be naturally mapped on.

**Proposition 3.** *Let $S$ be a list of special pairs constructed from a time series taken uniformly at random from a set of $n$ elements. Then, the expected size of $S$ is $n - H_n$, where $H_n$ is the $n$-th harmonic number.*

*Proof.* Let $E[X]$ represent the expected size of list $S$ of special pairs. Furthermore, let $X_{i,j}$ denote a random indicator variable that stores 1 if the pair of time indexes $(i, j)$ belongs to $S$ and 0 otherwise.

From the previous definitions, $E[X] = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}]$.

By the linearity of expectation, it follows that:

$$E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}].$$

Since $E[X_{i,j}] = \frac{1}{(j-i+1)(j-i)} = \frac{1}{j-i} - \frac{1}{j-i+1}$, we have:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left( \frac{1}{j-i} - \frac{1}{j-i+1} \right) = \sum_{i=1}^{n-1} \left( 1 - \frac{1}{n-i+1} \right) = n - \sum_{i=1}^{n} \frac{1}{i} = n - H_n.$$

We shall mention that it is possible to show that the cardinality of $S$ is highly concentrated around its mean value.

Since our research is strongly motivated by applications that arise in finance domain, we have also evaluated the number of special pairs for time series containing the prices of 48 different Brazilian stocks, sampled in a minute basis, over a period of three years. We have also considered the number of special pairs for the inverted time series. These inverted series shall be used if one is looking for negative variations rather than positive ones. We measured the ratio between the number of special pairs and the size of the underlying time series for each of these 96 series. We observe that this ratio lies in the range $[0.5, 7]$, with median value 2.50, and average value 2.75.

The results of this section suggest that the data structures that depend on the number of special pairs shall have a very reasonable space consumption for practical applications. We shall note that we can always set an upper bound on the number of special pairs to be stored and recourse to a structure that do not use special pairs if this upper bound is reached.

## 5.2 On the preprocessing/querying elapsed time

In this section we present the experiments that we carried on to evaluate the performance of the proposed data structures.

For these experiments, we selected, among the 96 time series available, those with the largest size, which accounts to 38 times series, all of them with 229875 samples. Our codes were developed in `C++`, using compiler `o g++ 4.4.3`, with the optimization flag `-O2` activated. They were executed in a 64 bits Intel(R) Core(TM)2 Duo CPU, T6600 @ 2.20GHz, with `3GB` of RAM.

To support RMQ's we implemented one of the data structure that obtained the best results in the experimental study described in [6]. This data structure requires linear time/space preprocessing and handles RMQ's over the range $[i, j]$ in $O(\min\{\log n, j - i\})$ time.

The following table presents some statistics about the time(ms) spent in the preprocessing phase of our algorithms.

| **Algorithm** | **Min Time** | **Average Time** | **Max Time** |
|---|---|---|---|
| AllPairs-RMQ | 7.8 | 8.7 | 10.1 |
| Beg-Hybrid | 44 | 128 | 302 |
| Beg-SP | 165 | 681 | 1601 |
| Beg-Quick | N/A | N/A | N/A |

**Experiments with query Beginning$(t, d)$.** In our experiments, we also included a naive method, denoted by Beg-Naive, that for each time index $i$, it looks for the first time index $j$ in the interval $[i + 1, i + t]$ such that $a_j - a_i \geq d$. Clearly, this method spends $O(nt)$ to process *Beginning*$(t, d)$.

Since we are interested in rare events, we focused on pairs $(t, d)$ that induce a small number of solutions for *Beginning*$(t, d)$, that is, the size of the output is a small percentage of the input's size. For each of the 38 series, we executed 30 queries, with $1 \leq t \leq 61$.

The elapsed times are plotted in Figure 5. The horizontal axis shows the ratio (in percentage) between the size of the output and the size of the input series while the vertical axis shows the query elapsed time.

First, we observe that Beg-Naive is hugely outperformed by the other methods. The dispersion in the results of Beg-Quick is due to the dependence of its query elapsed time with the value of $t$. We observe also that algorithm Beg-SP outperforms Beg-Quick when the size of the output is much smaller than the input's size. As this ratio gets larger, the difference becomes less significant. In general, when the size of the output is $p\%$ of the input's size, Beg-SP is $10/p$ times faster, in average, than Beg-Quick. As an example, when $p = 0.1\%$, Beg-SP is 100 times faster.

It is also interesting to observe that Beg-Hybrid, though slower than Beg-SP, also yields a reasonable gain when compared with Beg-Quick to detect very rare variations.

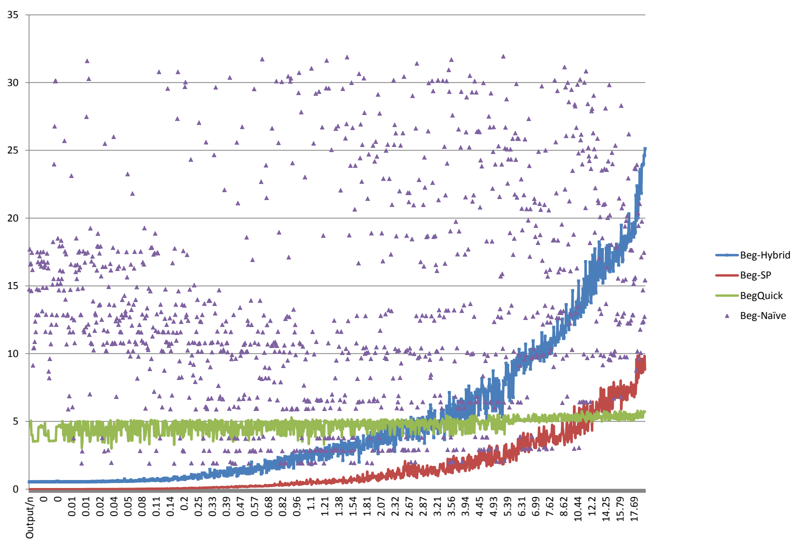

**Fig. 5.** Elapsed query times of the algorithms for processing Beginning$(t, d)$)

Recall that Theorem 2 states that Beg-SP spends $O(\log n + f \cdot (\log f + \log t) + k)$ time to process $Beginning(t, d)$, where $k$ is the size of the solution set and $f$ is the number of distinct endpoints of special $(t, d)$-events. Hence, it is interesting to understand the relation between $f$ and $k$. We observed that this ratio is smaller than 1 for all queries that produced at least 10 outputs. Thus, one should expect to pay $O(\log k + \log t)$ time per output generated by algorithm Beg-SP.

**Experiments with query AllPairs$(t, d)$.** We executed 30 queries AllPairs$(t, d)$ for each of the 38 series with 229875 samples. The elapsed times are plotted in Figure 6. As expected, AllPairs-SP outperforms AllPairs-RMQ when the size of the output is much smaller than the size of the time series. We have not performed other experiments due to the lack of space and because we understand that the theoretical analysis of AllPairs-SP and AllPairs-RMQ succeed in explaining their behavior.

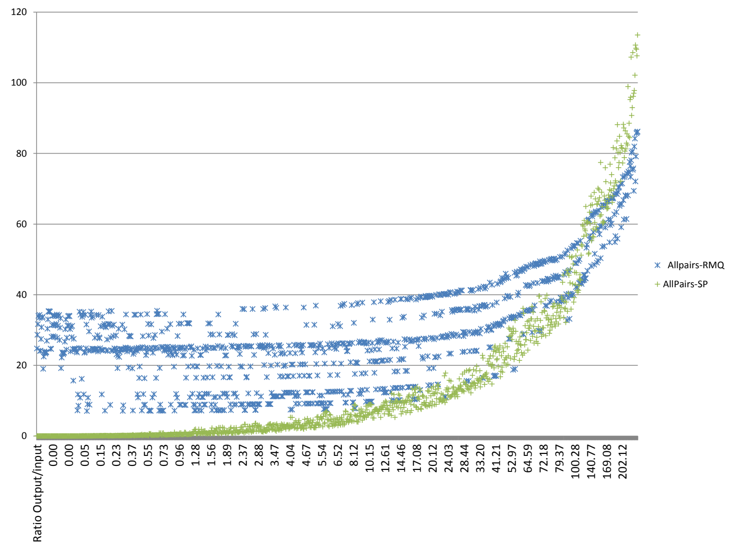

**Fig. 6.** Elapsed query times of AllPairs-SP and AllPairs-RMQ

## 6 Conclusions

In this paper, we proposed and analyzed algorithms/data structures to detect rare variations in time series. Our solutions combine data structures that are well known in the computational geometry community, as those that support RMQ's, with the notion of a special pair. We present some theoretical results and carried on a set of experiments that suggest that our solutions are very efficient in terms of query elapsed time and are compact enough to be used in practical situations.

Although we focused on two particular queries, $Beginning(t, d)$ and $AllPairs(t, d)$, the approach presented here can be extended to other queries with the same flavor. As a future work, we aim to investigate how to extend our techniques to deal efficiently with multiple time series and with queries that should count the number of solutions rather than reporting them.

## References

1. Michael A. Bender and Martn Farach-colton. The lca problem revisited. In In Latin American Theoretical INformatics, pages 88–94. Springer, 2000.
2. Jon Louis Bentley, Donald F. Stanat, and E. Hollings Williams Jr. The complexity of finding fixed-radius near neighbors. Inf. Process. Lett., 6(6):209–212, 1977.
3. Kausik Chaudhuri and Yangru Wu. Mean reversion in stock prices: evidence from emerging markets. Managerial Finance, 29(10):22–37, Mar 2003.
4. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry: Algorithms and Applications. Springer-Verlag, January 2000.
5. Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In Proceedings of ACM SIGMOD, pages 419–429, Minneapolis, MN, 1994.
6. Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In PROC. CPM. VOLUME 4009 OF LNCS, pages 36–48. Springer, 2006.
7. T. Fu. A review on time series data mining. Engineering Applications of Artificial Intelligence, (24):164–181, 2011.
8. E Gatev, WN Goetzman, and KG Rouwenhorst. Pairs trading: Performance of a relative-value arbitrage rule. Review of Financial Studies, 3(19):797–827, 2007.
9. Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. SIAM J. Comput., 13:338–355, May 1984.
10. Seung-Hwan Lim, Heejin Park, and Sang-Wook Kim. Using multiple indexes for efficient subsequence matching in time-series databases. Inf. Sci, 177(24):5691–5706, 2007.
11. Andrew Lo, H. Mamaysky, and J. Wang. Foundations of technical analysis: Computational algorithms, statistical inference, and empirical implementation. Journal of Finance, 2(1):1705–1770, Mar 2000.
12. Yang-Sae Moon, Kyu-Young Whang, and Wook-Shin Han. General match: a subsequence matching method in time-series databases based on generalized windows. In Michael Franklin, Bongki Moon, and Anastassia Ailamaki, editors, Proceedings of the ACM SIGMOD International Conference on Management of Data, June 3–6, 2002, Madison, WI, USA, pages 382–393, pub-ACM:adr, 2002. ACM Press.
13. D. Shasha and Y. Zhu. High performance discovery in time series: techniques and case studies. Springer-Verlag, 2004.
14. Huanmei Wu, Betty Salzberg, and Donghui Zhang. Online event-driven subsequence matching over financial data streams. In ACM, editor, Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data 2004, Paris, France, June 13–18, 2004, pages 23–34, pub-ACM:adr, 2004. ACM Press.