Politecnico di Torino

Master's degree in Computer Engineering

DEGREE THESIS

Anomaly detection by means of consecutive pattern discovery

Supervisor:
Luca Cagliero

Candidate:
Stefano Tata

October 2021

Contents

List of Tables

List of Figures

Acronyms

ACM	Association for Computing Machinery
CPDA	Consecutive Pattern Discovery Algorithm
ECG	Electrocardiogram
FFT	Fast Fourier Transform
GPPDA	General Purpose Pattern Discovery Algorithm
ICDM	International Conference on Data Mining
IEEE	Institute of Electrical and Electronics Engineers
MASS	Mueen's ultra-fast Algorithm for Similarity Search
NAB	Numenta Anomaly Benchmark
PIP	Perceptually Important Point
PVC	Premature Ventricular Contraction
SCAMP	SCAlable Matrix Profile
SCRIMP	Scalable Column Independent Matrix Profile
SoCC	Symposium on Cloud Computing
SoTA	State of The Art
STAMP	Scalable Time series Anytime Matrix Profile
STOMP	Scalable Time series Ordered-search Matrix Profile
XAUUSD	Gold price in dollars

Acknowledgements

The ideas that drive this thesis work are the foundation for a lifelong project and derive from the author's passion in automation, combined with a favorable series of events. Before graduating from the Polytechnic University of Turin, I worked for several companies both in Turin and Pinerolo, the town I was born and raised in. The experiences I made lit a fire in me: I could not keep selling my passion to companies and let them annihilate my energy.

I quit on September 30th, 2019 and bought a motorcycle with the money I had made. I already had an idea for my future, but I needed to breathe and remember what life was really about: I hopped on my Kawasaki Z650 on my way to Madrid, forgetting all about the past and getting ready for the present. My energies would finally become mine: I was the only one that would benefit from them, getting paid happiness instead of money. As soon as I got there, I met some friends and hanged out, discussing history and politics all throughout the night: "History always repeats itself", said Nikolay. My brain, already covered with memories of failed attempts in creating intelligent trading automation code, was tickled in just the right way: "History repeats itself both in short and long term", I thought, "Ice ages, seasons, conflicts, financial crises, it all comes to patterns! If I identified and classified them, I could develop logics to predict them!". This is the way it started: I had finally found happiness.

Life as a young adult in the third millennium is a dip in the open sea: if one's energies are entirely spent in panic and frustration the risk of drowning before reaching the land is very high. No one knows for sure they will, the only certainty is being in the open sea with the desperate need for a

psychological dimension. The science behind computers and algorithms is my raft, the one that grants access to order in a disordered world, the one my creative energies would crash violently onto my psychological health had I lost. Everyone in this world has a purpose, mine is to create: we only have one chance to be remembered, mine started October 1st, 2019.

1. Introduction

In recent years, machine learning and data mining have revolutionized researchers' approach to decision-making in data-related problems. The possibility of portraying the physical world through digital data introduces a considerable deal of disparity between the amount of valuable information available and the extent of computational power needed to analyze data the conventional way. Researchers' endeavour in data analysis has recently targeted the exploitation of machine learning and data mining techniques to properly address the issue, relying on "intelligent" code rather than standard algorithms.

In a research cluster as heterogeneous as data science, time series analysis covers a conspicuous number of branches. Time series, also known as "signals", are data sequences that have a temporal collocation (they can be framed as bidimensional arrays reporting time in abscissa - with or without unit of measure - and values in ordinates): data from sensors or medical equipment, stocks and even DNA sequences are examples of time series. The analysis spectrum that involves this particular kind of data is constantly growing and varies with context, ranging from forecasting in statistics, mathematical finance, meteorology, seismology and geophysics to classification, clustering, similarity search and anomaly detection in machine learning and data mining, all by means of a Pattern Discovery Algorithm.

Pattern Discovery in time series has received a lot of attention from researchers recently, especially in a world where hardware technology keeps improving and state-of-the-art computational potential starts satisfying the huge amount of processing power the analyses in such a complex field require. Researchers have been focusing their attention on finding the best way to compare, in a given time series, each existing sub-signal of a known length (a time series' portion composed of n data points) to every other sub-signal of the same length the time series is possibly composed of, with the intent of spotting meaningful similarities. This pairwise comparison mostly consists in computing relative distances between sub-signals, task that is practically unfeasible if performed "brutally", both in terms of time and memory consumption. State-of-the-art Pattern Discovery Algorithms are versatile tools that perform the above task and provide the complete set of pairwise comparisons' results in astonishingly short execution times.

Although state-of-the-art algorithms claim fast execution times for smaller datasets, they still have to deal with computational complexities being quadratic (doubling the size of the datasets results in quadrupled execution times) and poor scalability attributes as a consequence. Complexities cannot be improved in research areas where the globality of results is mandatory for the main objective to be reached; yet, there is the need to explore the situations when time series exploration and pattern discovery can be tailored to particular cases. A notable example is the anomaly detection domain, where the goal is to observe behaviors that deviate from the time series' normality: the approach described above makes analyses unattainable and results unsatisfactory for moderately hefty datasets.

My thesis work studies the relationships across consecutive sub-signals in the Anomaly Detection Research field proposing GPPDA (General Purpose Pattern Discovery Algorithm), a novel algorithm that targets the similarity searches in the sub-signals' neighborhood and is configurable to solve the majority of the Pattern Discovery problems nonetheless. The proposed approach guarantees a linear computational complexity and more meaningful results, under the assumption that similarities occur consecutively in an ordinary scenario.

As discussed in *chapters 4 and 5*, the process that brings to the introduction of GPPDA has been handled with a bottom-up approach, making the main method revolve around the implementation of a Consecutive Pattern Discovery Algorithm: given an input time series, the desired output is a list of similarities that consists of couples, each one defining the temporal collocation, within the input signal, of the pairs that triggered a similarity, commonly referred to as "Consecutive Patterns". The Consecutive Pattern Discovery is an unexplored subject that does not address a specific problem directly, but has the potential to set the stage for a wide range of applications, amongst which Anomaly Detection is the most emblematic.

The first part of the thesis (*section 4.1*) presents the problem from a very high-level standpoint, introducing the basics of the algorithm that is going to be engineered and the datasets that are going to be used throughout the research in order to test the validity of the implemented concepts.

Once the problem has been defined, the Consecutive Pattern Discovery Algorithm can be implemented as a very simple task (*section 4.2*): an input time series is iterated from start to end (i.e. chronologically) by means of a conceptual sliding window that, at each step of the iteration, performs computations on the current portion of data: the sub-signal's most meaningful data points are identified by analyzing its positive and negative peaks. Peaks can be used to summarize a signal's features into a more confined set of information, that will be compared with the same kind of information retrieved when analyzing previous "shots" of the input signal. If the normalized coordinates of two sub-signals' most relevant peaks match within a predefined threshold, the two sub-signals trigger a similarity. This version of the algorithm is very trivial for it performs repeated computations on sub-signals that share most features with each other, without exploiting any data continuity property. Furthermore, it is worth noticing that the same, complete iteration is repeated as many times as the number of sliding windows that are considered: not only do sub-signals show similarities in “shape”, but in duration (i.e. size) too.

The actions that follow include the exploitation of data continuity properties. Data continuity is exploited on two fronts: the computation of the sub-signal’s absolute maxima and minima (needed to perform the peaks’ coordinates’ normalization), and the computation of its positive and negative peaks. The first action that is needed in order to make the process successful is enveloping the iterations of multiple sliding sub-windows into a single iteration of a “bigger” sliding super-window, making it easier to exploit data continuity even at a higher level. Absolute maxima and minima are now stored in a hierarchical global variable that allows useful computations to be performed only once (even if they involve multiple sub-windows) and repetitive computations to be safely skipped, saving a lot of execution time. Although this is true, the biggest improvement in speed derives, in fact, by the optimization of the positive and negative peaks retrieval process: in order not to repeat heavy and preventable tasks, a solution similar to the one presented for absolute maxima and minima was employed, detaching calculations from library functions (that needed to be called multiple times without the possibility to incrementally compute results) and storing all the needed information in a hierarchical global variable.

The last improvement that needs to be performed, at this point, is the parallelization of the whole process. Each core of the machine the Consecutive Pattern Detection Algorithm runs on is conceptually assigned a sub-set of sub-windows that are going to be analyzed independently, exploiting the fact that the compare operations are sub-window independent and can be run in parallel.

Once the algorithm has been completely developed and thoroughly tested for bugs (a dataset including half a trillion data points was tested in *sections 4.3 and 4.4*) a generalization of the concepts that drive it can be made, giving birth to the General Purpose Pattern Discovery Algorithm in *chapter 5*. GPPDA is a tool offering eighteen different configurations, one of which being the configuration useful to solve Anomaly Detection problems.

As we can read in *chapter 6*, what emerged from a set of tests that aimed to cover a heterogeneous collection of datasets in which anomalies could be found, is the multitude of advantages the used approach lets us draw benefits from using. GPPDA's results were compared to the state-of-the-art algorithms in the Anomaly Detection field, showing overall better performances. Not only does comparing sub-signals to their close "neighborhood" take execution times and scalability to another level for Anomaly Detection problems, but it enhances precision and recall too. Results show state-of-the-art algorithms failing in spotting two or more similar anomalies, as well as in detecting continuous changes in sub-signals that are meant to show regularities. These two events were demonstrated happening more often than one could think, showing themselves as a real condition in datasets that are, clamorously, of completely different nature.

I believe the presented work is a preliminary step towards further research on scalable time series analyses, as discussed in *chapter 7*. The sequential nature of the algorithm, together with its cutting-edge performance, could guarantee the real-time detection of anomalies in continuous streams of data. More ambitious research involve the real-time prediction of patterns showing up consecutively in irregular data or anomalies occurring in supposedly regular streams, providing a view of prediction problems from a whole new perspective.

2. Pattern Discovery

Pattern Discovery in data science consists in the employment of algorithms capable of providing information about the objective analogies that can be drawn across different portions of data.

Time series analysis covers a solid number of research branches where Pattern Discovery Algorithms can be employed: time series, also known as ”signals” or ”digital signals” in signal processing, are sequences of values retrieved at subsequent instants, each one being chronologically equidistant from the adjacent samples. We can frame time series as bidimensional arrays reporting time in abscissa - with or without unit of measure - and values in ordinates, having a cardinality C_{ts} equal to the sampling frequency employed during the measurement that generated the array, multiplied by its duration:

$$C_{ts} = f_s \cdot T_m \tag{2.1}$$

We quickly grasp the vastness of the application spectrum Pattern Discovery in time series analysis presents itself with: data from sensors or medical equipment, stocks and even DNA sequences are valid candidates for research branches that make use of Pattern Discovery Algorithms, amidst which Similarity Search and Anomaly Detection are the most representative.

Pattern Discovery aims at developing efficient procedures that compare, in a given time series, each existing sub-signal of a known length (a time series' portion composed of q data points) to every other q-long sub-signal the time series is possibly composed of, with the intent of spotting meaningful similarities (Similarity Search) or anomalies (Anomaly Detection).

If we were to carry out the above procedure on a C_{ts}-long time series, we would have to deal with a number of comparisons Γ that results from the following equation:

$$\Gamma = \frac{(C_{ts} - q)(C_{ts} - q + 1)}{2}, \tag{2.2}$$

bringing the effort needed to complete the procedure beyond state-of-the-art computational capabilities.

Let us consider, as an example, the number Γ_1 resulting from the analysis of a 2^{26}-long time series we need to extract the complete set of 2^8-wide sub-signals' pairwise comparisons out of:

$$\Gamma_1 = \frac{(C_{ts} - q)(C_{ts} - q + 1)}{2} = \frac{(2^{26} - 2^8)(2^{26} - 2^8 + 1)}{2} \simeq 2.25 \cdot 10^{15} \tag{2.3}$$

If each comparison consists in computing the Euclidean Distance between two 2^8-long sub-signals, we could portray a utopian scenario where a $3\,GHz$ machine is capable of perfectly pipelining the lines of code used to perform the evaluation, taking approximately 0.3 nanoseconds to compute each of the $2^8 \cdot \Gamma_1$ operations the analysis is made up of: in these conditions, more than 6 years would be required to complete the whole process.

Deriving the Euclidean Distance between two 2^8-long sub-signals consists, in reality, in computing $2 \cdot 2^8$ squares, $(2 \cdot 2^8 - 1)$ sums and one final square root. Considering the overall memory access time the aforesaid operations imply, together with the total number of CPU instructions they will be decomposed into, we quickly come to the conclusion that the first estimate is at least ten times more optimistic than a realistic one, even if we take parallelization into account.

The Matrix Profile project, started in 2016 by the University of California, Riverside, has received a great deal of attention from researchers worldwide: it is considered the state-of-the-art set of algorithms for Pattern Discovery, as it provides procedures that solve problems, such as the one above, in exceptionally short execution times.

[1] is the first algorithm introduced by researchers presenting the Matrix Profile project at the IEEE International Conference on Data Mining (ICDM) 2016. The target of Scalable Time series Anytime Matrix Profile's (STAMP) analysis, as the name suggests, is the definition of an input signal's Matrix Profile, an array of values summarizing the set of comparisons whose cardinality we have been reffering to as Γ. The principle such a hefty set of comparison results is summarized upon is simple: every q-long sub-signal's nearest neighbor is a numeric element in the Matrix Profile describing its Euclidean Distance from the original sub-signal that creates, together with the other sub-signals' nearest neighbors, a complete representation of the set of pairwise comparisons through a $(C_{ts} - q + 1)$-long result array.

From an analytical standpoint, *we call the Matrix Profile $P(mT_s)$ of a signal $T(nT_s)$ the signal resulting from the computation, for each $x, q \in \mathbb{Z}$ such that both $T(nT_s)\,|\,n = x$ and $T(nT_s)\,|\,n = x + q$ exist, of the lowest Euclidean Distance between the sub-signal $T(nT_s)\,|\,x \leq n < x + q$ and every possible sub-signal $T(nT_s)\,|\,x' \leq n < x' + q$, $x' \geq x + q \vee x' \leq x - q$, with q being the fixed, pre-chosen length of the sub-signals (subsequences) to be analyzed.*

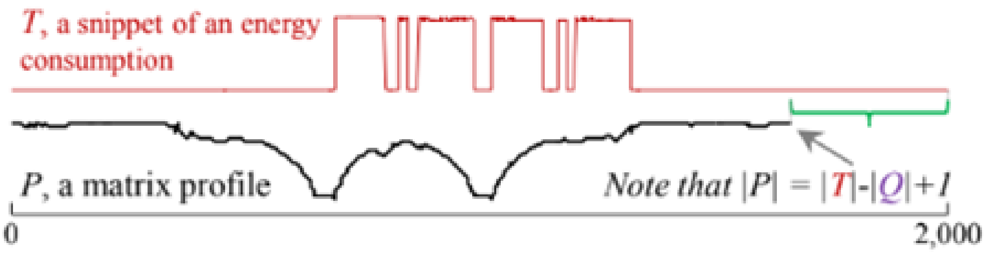


Figure 2.1: A Matrix Profile visualized

Source: [1]

The way STAMP operates is by iteratively performing Mueen's ultra fast

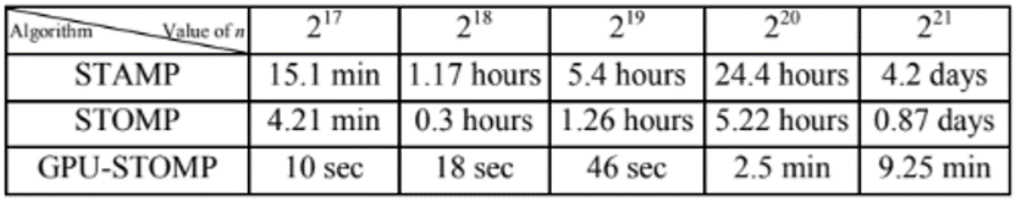
Algorithm for Similarity Search (MASS) on the signal $T(nT_s)$ for every $n \leq |T| - q$, with $|T| = C_{ts}$. MASS receives a query Q_x of length $|Q_x| = q$ and computes the Euclidean Distances between Q_x and every other q-long sub-signal $Q_{x'}$ belonging to T that does not intersect with Q_x, exploiting the Fast Fourier Transform (FFT) of T and storing the lowest Euclidean Distance in P at position x.

Although STAMP improves the Pattern Discovery brute-force solution considerably, it would still take about a decade to cover the equivalent of a classic solution's Γ_1 pairwise comparisons with a modern machine, according to [1]. As for what concerns scalability issues, STAMP's overall $O(n^2 \log_2 n)$ worst-case complexity (resulting from the repetition of the $O(n \log_2 n)$ similarity searches across the time series) carries the burden of a Pattern Discovery Algorithm that cannot scale as a result of its quadratic nature.

[2] was presented alongside STAMP at the IEEE ICDM 2016. The Scalable Time series Ordered-search Matrix Profile algorithm (STOMP, and its GPU-accelerated version GPU-STOMP) is an improvement of the Anytime algorithm that exploits data locality and brings the complexity down to $O(n^2)$. The main difference between the two algorithms concerns STAMP's anytime property, that is absent in STOMP and allows the execution to be stopped whenever the solution is considered satisfactory enough.

According to [10], STAMP has the advantage to provide, for *"favorable scenarios"*, an accurate approximation of the Matrix Profile in one twentieth of the total execution time, a property that has almost no impact in practical terms: if the best-case scenario for a 2^{26}-long time series analysis implies a six-month-long execution, the worst-case scenario still makes the analysis unatteinable.

If we look at the speed and scalability comparison between STAMP, STOMP and GPU-STOMP, we come to the realization that no sensible improvements were made in terms of scalability (although GPU-STOMP covers this for smaller datasets), while an important breakthrough is attributable to speed enhancements:



Algorithm Value of n	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
STAMP	15.1 min	1.17 hours	5.4 hours	24.4 hours	4.2 days
STOMP	4.21 min	0.3 hours	1.26 hours	5.22 hours	0.87 days
GPU-STOMP	10 sec	18 sec	46 sec	2.5 min	9.25 min

Figure 2.2: Speed and scalability comparison between STAMP, STOMP and GPU-STOMP

Source: [2]

[10] and [3] were presented two years later at the IEEE ICDM 2018 and three years later at the ACM Symposium on Cloud Computing (SoCC) 2019 respectively. Scalable Column Independent Matrix Profile (SCRIMP++) is the combination of the best previous algorithms' best features: while maintaining STOMP's $O(n^2)$ worst-case complexity, SCRIMP++ manages to offer STAMP's anytime property, cutting execution times significantly in *"favorable scenarios"*.

According to [2], the anytime property of the Matrix Profile algorithms is in conflict with GPU acceleration, so GPU-STOMP will not benefit from this additional feature. Still, GPU-STOMP is the fastest Matrix Profile computer in 2018. The SCAlable Matrix Profile algorithm (SCAMP), introduced in 2019, consists in an improved version of GPU-STOMP that offers better execution times, yet no improvements in scalability whatsoever.

As of 2021, if we were to decide which algorithm to use to perform Similarity Search or Anomaly Detection through Pattern Discovery, we would ultimately opt for SCRIMP++ in scenarios where GPU computation is not available and the anytime property can be exploited, SCAMP otherwise.

3. Anomaly Detection

The same way a lower Euclidean Distance suggests a closer resemblance between two time series, a higher distance value is an indicator of stronger dissimilarity: we can apply this concept to the Matrix Profile context if we define the potential indicator of an anomaly in a time series as the highest value in a Matrix Profile neighborhood.

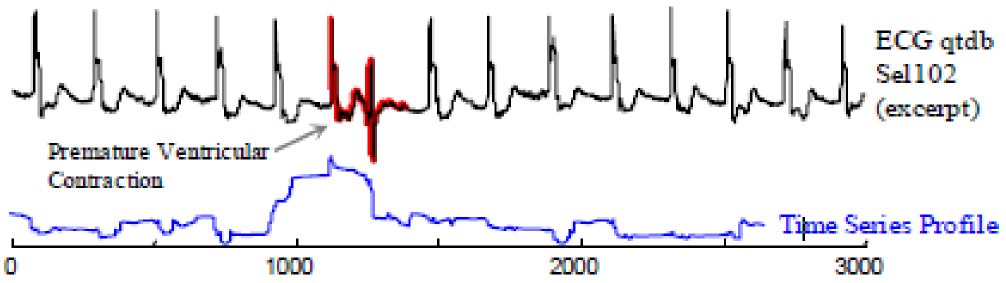


Figure 3.1: A snippet of an ECG recording showing a PVC (in red) and its Matrix Profile (in blue)

Source: [1]

As we can clearly see from the electrocardiogram sample above, the red-highlighted sub-signal corresponding to a Premature Ventricular Contraction exactly matches an evident local maximum (which is, coincidentally, also the absolute maximum) in the resulting Matrix Profile.

The Pattern Discovery Algorithms based on the Matrix Profile, in addition to suffering from the scalability issues illustrated in the previous chapter, introduce important concerns, in terms of result meaningfulness, for Anomaly Detection problems. As the globality of results can be very important for many similarity search problems (meaning that similarities between far-in-time sub-signals are as meaningful as similarities between close-in-time ones), the same may not be valid in the anomaly detection field.

Let us consider, as an example, a hypothetical thirty-minute-long ECG recording of a patient suspected of having anomalies in their heartbeat. If we wanted to have, as long as heartbeat regularity is concerned, an accurate picture of the patient's condition, it would be beneficial to receive the information regarding the irregularities' time collocation and frequency from the candidate Pattern Discovery Algorithm. A Matrix-Profile-based algorithm would do its job when potential multiple anomalies are clearly different from one another; yet, a Premature Ventricular Contraction is a well-known disruption of the heart's regular rhythm, often resulting in similar electrocardiogram shapes, even across patients.

If we investigate further, we will be able to demonstrate the presence of regularities amongst apparently unrelated anomalies across several different domains, portraying Matrix-Profile-based algorithms as unreliable tools for many Anomaly Detection problems. This issue acquires particular relevance when considering the alternatives the research world provides us with: as we read from [1], there are several Anomaly Detection algorithms present in the literature, all of which fall onto the "brute-force curse" in the worst-case scenario. According to Bart Goethals, Professor of Computer Science at the University of Antwerp, the Matrix Profile algorithms are the state-of-the-art tools to solve Anomaly Detection problems [4].

If the presence of regularities in anomalies was not the case in the real world, would it still be beneficial at all to compare, for example, a heartbeat H_i to the ones that precede H_{i-1}, including them in the Pattern Discovery equation and potentially sabotaging results? Is the price we pay in scalability worth the results we obtain when detecting anomalies?

This thesis work provides answer to all the above questions, illustrating the possibility for the anomaly detection research field to detach from the standard Pattern Discovery procedures and offering a solution that was engineered for a specific problem.

4. Consecutive Patterns

4.1 Overview

4.1.1 Definitions and Algorithm

Before exploring the Consecutive Pattern Discovery Algorithm in details, we should provide a proper definition of "Consecutive Pattern" in signals and a high-level description of the earliest algorithm to extract them from a time series.

We call "Consecutive Pattern" in time series analysis a pattern occurring, with a certain degree of approximation, twice in a consecutive manner.

A pattern, also referred to as "motif" in academic literature, is a ω-long signal's portion, with ω being a pre-chosen value. If we wanted to identify every existing pattern in order to detect their eventual consecutive repetitions in a time series we should, mathematically, consider every ω-wide window and make it slide throughout the signal, giving:

$$0 < \omega \leq \frac{n}{2}\,, \tag{4.1}$$

with n being the time series' length.

Our Consecutive Pattern Discovery Algorithm (and its General Purpose version GPPDA) makes use of an approach reconductable to the above process, introducing the following parameters and definitions:

ω_m: the analysis' minimum window width;

ω_M: the analysis' maximum window width;

λ: the number of windows considered in the analysis;

η: the difference in width across windows;

$\epsilon_\omega = \lfloor \frac{1}{\beta} \cdot \omega \rfloor$: the shift, expressed in number of data points, each window performs when sliding through the time series (β is a pre-chosen value);

$\sigma(\omega)$: the set of pairwise comparison operations that are carried out, for a single window at a given point in the sliding process, in order to detect a Consecutive Pattern;

$\phi(\omega)$: the sub-signal elaborations that are carried out, for a single window at a given point in the sliding process, in order to provide the elements for $\sigma(\omega)$ to operate.

The proposed Algorithm (together with its improvements and generalization) conceptually follows the flow hereunder, with modifications aiming to improve overall speed and scalability through actions on $\sigma(\omega)$ and $\phi(\omega)$ as well as code reorganization:

Algorithm 1 CPDA (GPPDA)

Input: T (a time series) and its length n, ω_m, ω_M, η and β

Output: A list of similarity couples grouped by ω

```
procedure Main
    for ($\omega_i = \omega_m$; $\omega_i <= \omega_M$; $\omega_i += \eta$) do
        for ($j = \omega_i + 1$; $j <= n$; $j += \lfloor \frac{\omega_i}{\beta} \rfloor$) do
            if $isSubSignalValid(T[j - \omega_i : j])$ then
                $P[j] \leftarrow \phi(\omega_i, T, j)$
            if ($j >= 2\omega_i$) then
                $results[\omega_i] \leftarrow \sigma(\omega_i, P, j)$
    return $results$

procedure $\phi(\omega, T, j)$
return $elaborateSubSignal(T[j - \omega : j])$

procedure $\sigma(\omega, P, j)$
    for ($i = dichotomicSearchOnP(\omega, P, j)$; $i >= j - 2\omega$; $i--$) do
        if ($isConsecutivePattern(P[i], P[j])$) then return $i$, $j$
```

While further details on the algorithm are provided in the "Consecutive Pattern Discovery Algorithm" section, our aim in these introductory sections is to provide the foundation for an easier and thorough understanding of the processes that follow.

We should now have a glance at the preliminary worst-case computational complexity $\Omega(n, \lambda)$ of the algorithm, in which σ and ϕ are, for the sake of simplicity, the complexity functions corresponding to the homonymous code procedures:

The algorithm's overall complexity $\Omega(n, \lambda)$ is given by the sum of the λ single-windowed executions' complexities, that are defined, in turn, as the i^{th} number of iterations throughout the complete dataset ($\lfloor \frac{n-\omega_i}{\epsilon_i} \rfloor + 1$), multiplied by the sum of the sub-signal elaborations complexity ϕ and the pairwise comparison operations complexity σ, both calculated in ω_i.

$$\Omega(n,\lambda)|_{\omega_m,\eta,\beta} = O(\sum_{i=1}^{\lambda} \sum_{j=0}^{\lfloor \frac{n-\omega_i}{\epsilon_{\omega_i}} \rfloor} (\phi(\omega_i) + \sigma(\omega_i, \tau_{1,j}, \tau_{2,j}))), \tag{4.2}$$

$$\sigma(\omega, \tau_1, \tau_2) = 1 + \lfloor log_2 \lceil 1 + \tau_1 \frac{2\omega}{\epsilon_\omega} \rceil \rfloor + \lfloor \tau_2 \frac{\omega}{\epsilon_\omega} \rfloor, \tag{4.3}$$

$$\lambda \ll n, \quad \omega_1 = \omega_m, \quad \omega_i = \omega_{i-1} + \eta \quad \forall\, i > 1, \tag{4.4}$$

$$\tau_{1,j} = \frac{j \cdot \epsilon_{\omega_i}}{2\omega_i} \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} < 2\omega_i, \quad \tau_{1,j} = 1 \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} \geq 2\omega_i, \tag{4.5}$$

$$\tau_{2,j} = 0 \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} < \omega_i, \quad \tau_{2,j} = (j - \lfloor \frac{\omega_i}{\epsilon_{\omega_i}} \rfloor) \cdot \frac{\epsilon_{\omega_i}}{\omega_i} \quad \forall\, j \,|\, \omega_i \leq j \cdot \epsilon_{\omega_i} < 2\omega_i,$$

$$\tau_{2,j} = 1 \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} \geq 2\omega_i, \tag{4.6}$$

$$\epsilon_\omega = \lfloor \frac{1}{\beta} \cdot \omega \rfloor \tag{4.7}$$

The above function's purpose is to introduce the reader to the linear nature of the algorithm, which is already visible from the inner sum: although the linearity of the algorithm is not particularly relevant at the moment (ω was not considered and ϕ could be anything from logaritmic to quadratic, as we do not know exactly what it does yet), its benefits will become clearer throughout the following sections.

4.1.2 Data Source Choices

In order to properly get to a more practical definition of "Consecutive Pattern" we must agree on the source we will get a first picture from. Financial markets are indeed an interesting starting point, as they are the most diversified data source available as far as pattern occurrence is concerned: financial assets have seen a significant number of recurring geopolitical events that have influenced their nature over the years. Interestingly enough, financial markets best represent the concept of unavoidable mechanistic harmony that is as evident in nature as hidden in a world run by complex individuals, whose global behavioral nature follows the harmonious pattern schema nonetheless. In other words, consecutive patterns may be occurring in financial assets without humans realizing it, opening an infinite list of possibilities both in academic research and business contexts: more elucidations of future possibilities will be given in the "Conclusions and future work" section.

As for what concerns this thesis work and the application of Consecutive Patterns in the Anomaly Detection field, considerations and experiments will be conducted on different data sources and contexts, providing material backed by the scientific community that can be framed in both the fields of statistics and biomedical sciences.

Financial data has been downloaded from dukascopy.com, a Swiss forex broker: the choice resides in the precision this broker offers historical data with. The temporal window the downloaded data refers to covers the all-time Dukascopy's archive of XAUUSD (Gold price in dollars) fluctuations with tick precision, starting May 5^{th}, 2003 and ending September 30^{th}, 2019.

Since the maximum temporal extension of a single file download that this broker allows is one day, it would take a large amount of time to cover a sixteen-year period with single-day downloads. GitHub has come in help:

there exists a Python script called “duka” (downloadable through pip) that receives the ends of the desired temporal window and downloads data for us. The script creates an output .csv file containing as much rows as the number of variations (ticks) the commodity market saw in the desired temporal window.

Each row is then composed of the following:

- **Date and Time** of the variation (YYYY-MM-DD HH-MM-SS.xxx000): GMT time
- **Ask**: the maximum price the sellers were willing to ask at the time
- **Bid**: the maximum price the buyers were willing to offer at the time
- **Ask Volume**: the number of sell contracts that have been traded on Dukascopy since the last tick
- **Bid Volume**: the number of buy contracts that have been traded on Dukascopy since the last tick

The version used for downloading test data is duka-0.2.0. A few edits have been made to this version in order to improve the resulting output text formatting:

- duka\core\csv_dumper.py, line 12:

return format(number, ’.5f’)

$$\Downarrow$$

return format(number * 100, ’.3f’)

– duka\core\csv_dumper.py, line 85:

with open(join(self.folder, file_name), ’w’) as csv_file:

$\Downarrow$

with open(join(self.folder, file_name), ’w’, newline = ”)
as csv_file:

Since the script progressively downloads and collects data from the source without dumping it on disk until completion, RA Memory saturates very quickly: a divide and conquer approach has been used in order to address this problem properly. Instead of downloading the entire sixteen-year window, as many sub-windows as the number of available XAUUSD one-day charts in the above-mentioned period have been created: a PowerShell script came in helpful at this point. Each “duka” execution is structured as follows:

“duka **XAUUSD** -t **1** -s **{START}** -e **{END}** -f **{FOLDER}**”

XAUUSD: the commodity market that we want to download historical data of
-t **1**: the number of threads used for the execution
-s **{START}**: the inferior end of the download window (YYYY-MM-DD)
-e **{END}**: the superior end of the download window (YYYY-MM-DD)
-f **{FOLDER}**: the output’s desired absolute path

Some of the downloaded charts were empty due to the fact that XAUUSD markets are not open for business on special days (Christmas, New Year’s Eve, etc.): these files have been manually deleted.

Dataset Refinement Operations

The historical XAUUSD market dataset we are using has a tick-precision, meaning new information is stored whenever ask and/or bid values change: this condition implies a non-homogeneous data density in our dataset (i.e. variations can occur more or less frequently depending on a lot of factors). In order to work with a homogeneous dataset, we will have to "fill the gaps": since the date and time information of each price variation is given with a $10^{-3}s$ precision, we must replicate the i^{th} dataset entry (gold ask/bid price) as many times as the milliseconds that exist between the i^{th} and the $(i+1)^{\text{th}}$ entry minus one. Doing this will allow us to work with a recreated 1kHz-sampled signal that gives information about the gold price trend over a specific period. This is how we want our data to be transformed, more practically speaking:

```
[1] 2019-10-01 00:00:00.000000, 1500.000, 1500.000, 0, 0
[2] 2019-10-01 00:00:00.003000, 1500.100, 1500.100, 1, 1
```

Into:

```
[1] 1500.000
[2] 1500.000
[3] 1500.000
[4] 1500.100
```

This is achievable in a few Python code lines. The script originates the ask price signal only: the bid price signal is pretty much the same, so we will ignore it.

Let us look at the ask price signal resulting from the analysis of the XAUUSD chart on October 1st, 2019 [19:25:00 GMT, 19:35:00 GMT]:

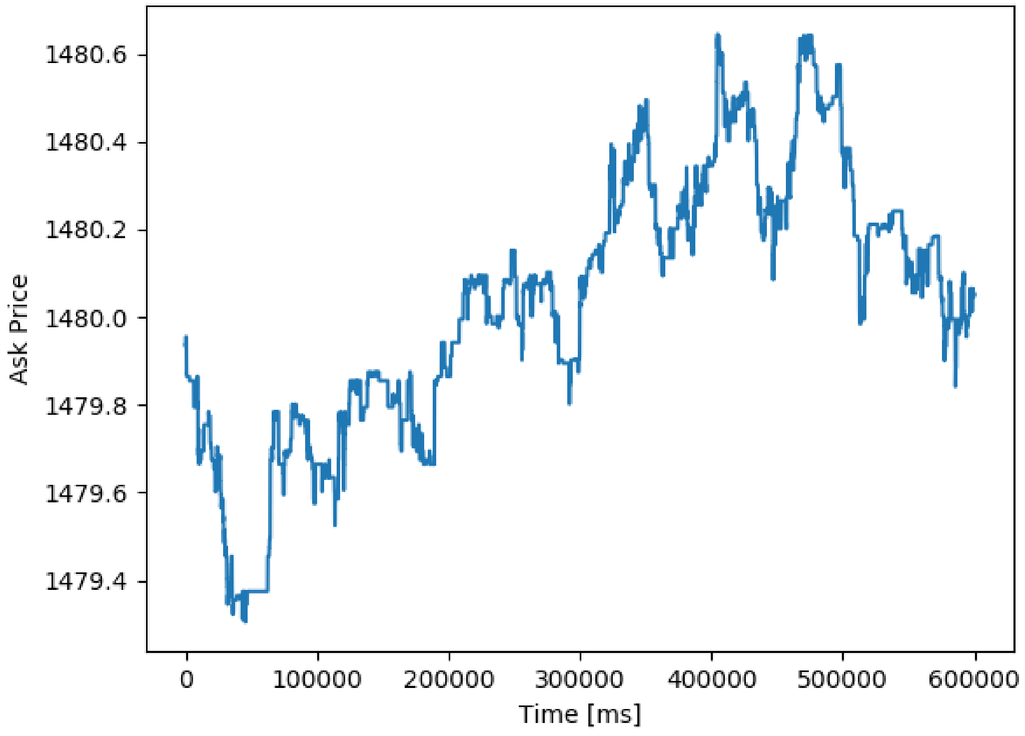


Figure 4.1: A snippet of our dataset

A first glance at Consecutive Patterns

In order to describe the desired output in a clearer way, we should consider a limited set of information: let us consider, for example, the .csv file that refers to the variations of the XAUUSD market throughout Tuesday, October 1st, 2019: Dukascopy makes both a .csv file and a chart available, so we can take advantage of the graphical interface to manually detect consecutive patterns throughout the chart. The search algorithm should then be able to detect at least the patterns that we have spotted by eye sliding through the chart.

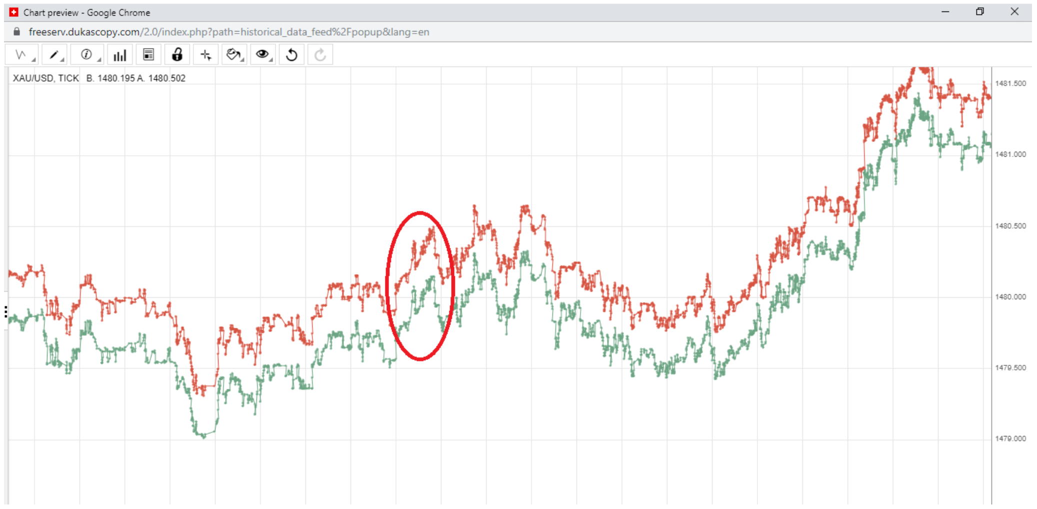


Figure 4.2: Visual example of a Consecutive Pattern

The highlighted one-minute pattern in the picture above (19:30 GMT) clearly repeats itself twice after it appears the first time. It gets fairly different one time from the other, but we can see how prices (both ask and bid) pretty much recreate the same picture: price shows a positive peak, a quick descend and a final climb by the end of the pattern (let it be strong or weak).

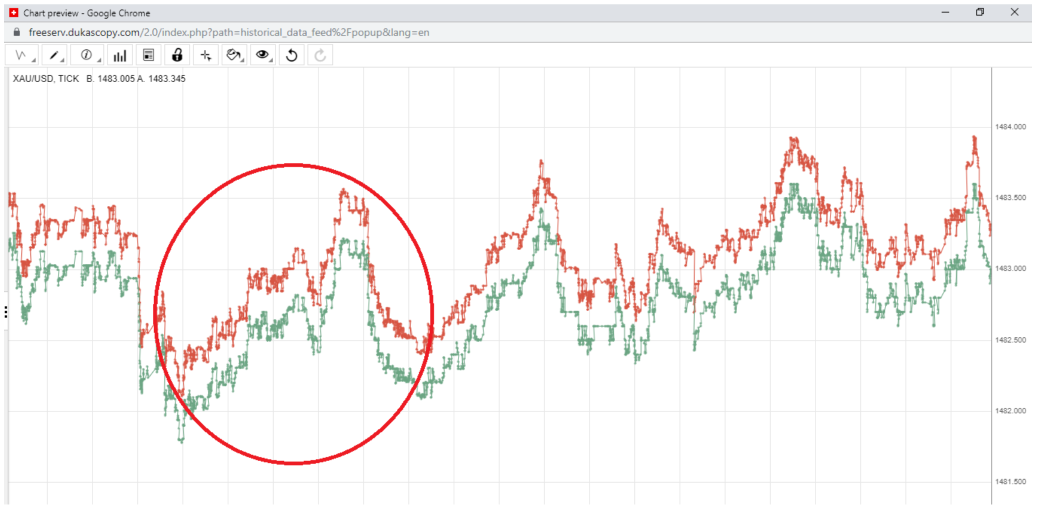


Figure 4.3: Visual example of a Consecutive Pattern (2)

The picture above shows something similar happening to a bigger pattern earlier that day (17:20 GMT).

4.2 Consecutive Pattern Discovery Algorithm

4.2.1 Introduction

Once our dataset has been properly initialized, we can start developing the Python algorithm for discovering patterns. We will start describing the algorithm step-by-step and we will end this chapter with some real case analyses: the smaller ones will serve as a demonstration of the procedure's proper behavior; one bigger analysis will then provide a bunch of useful statistics regarding the algorithm's application in a wider execution spectrum. Let us dive right into it: consider an initialized dataset like the one we have just seen in the previous section. If we wanted to identify every existing pattern in order to detect their eventual consecutive repetitions we should, mathematically, consider every ω-wide window and make it slide throughout the entire dataset, giving:

$$0\,[ms] < \omega\,[ms] \leq \frac{width_{dataset}}{2}\,[ms] \tag{4.8}$$

This is because not only do patterns come in different shapes, but sizes too. In this regard, we should make some decisions that are highly dependent on the application domain, deciding ω's minimum and maximum possible values. If we were to compute the average price variation frequency in the XAUUSD market throughout the years, we would recognize $1\,Hz$ to be a truthful value for a rough estimate. Since sub-signals that have less than a few dozen variations risk providing insufficient information for a pattern to be identified, we will consider ω's minimum possible value to be $60\,000\,\pm$ (sixty thousand), choice that translates into smallest patterns being one-minute-wide. On the other hand, we do not want ω to be excessively big: ten minutes is reasonable enough. ω's maximum possible value will be then set to $600\,000\,\pm$ (six hundred thousand) when $width_{dataset}$ is greater than ten minutes, $\frac{width_{dataset}}{2}$ otherwise. These values are purely indicative: we may decide for the interval to be either shrunk or expanded, together with further decisions regarding the number of milliseconds we would like our sliding window size to be increased by each time. Decisions will be mostly affected by execution times: since the algorithm repeats complex processing operations multiple times over, we had better expect times to be very long. All the parameters that are indicative and may vary are followed by a "±" symbol.

Let us see what we would like our code to do:

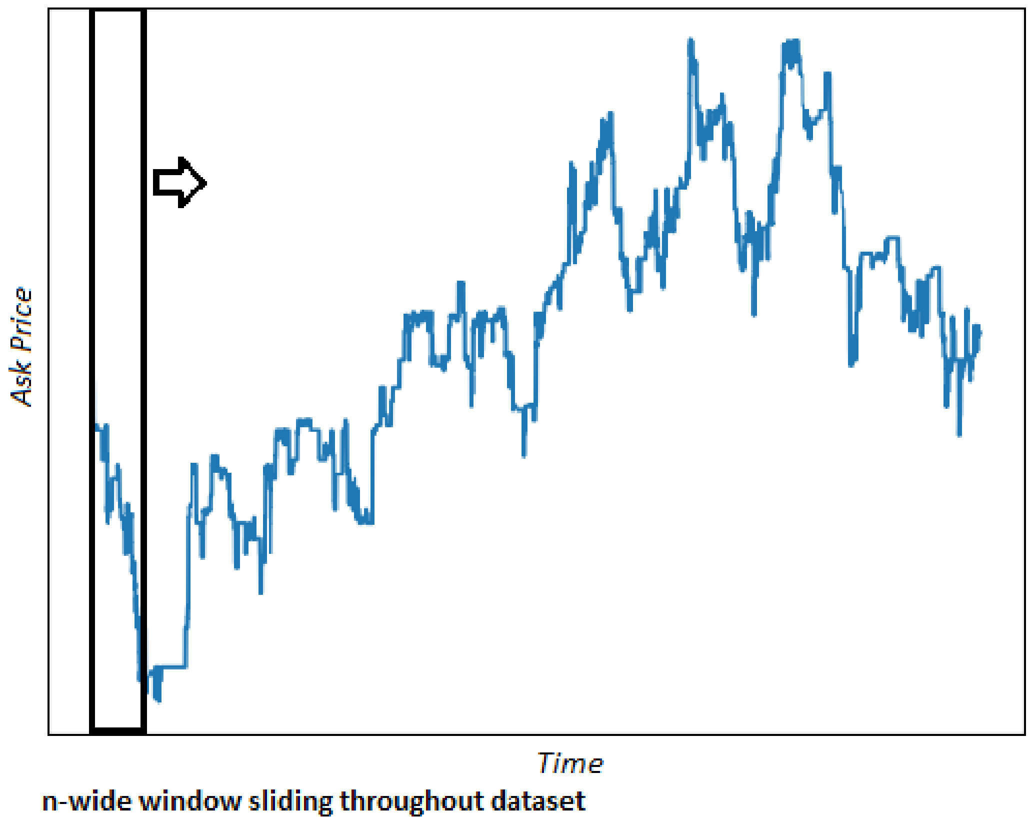


Figure 4.4: Simple visualization of the algorithm

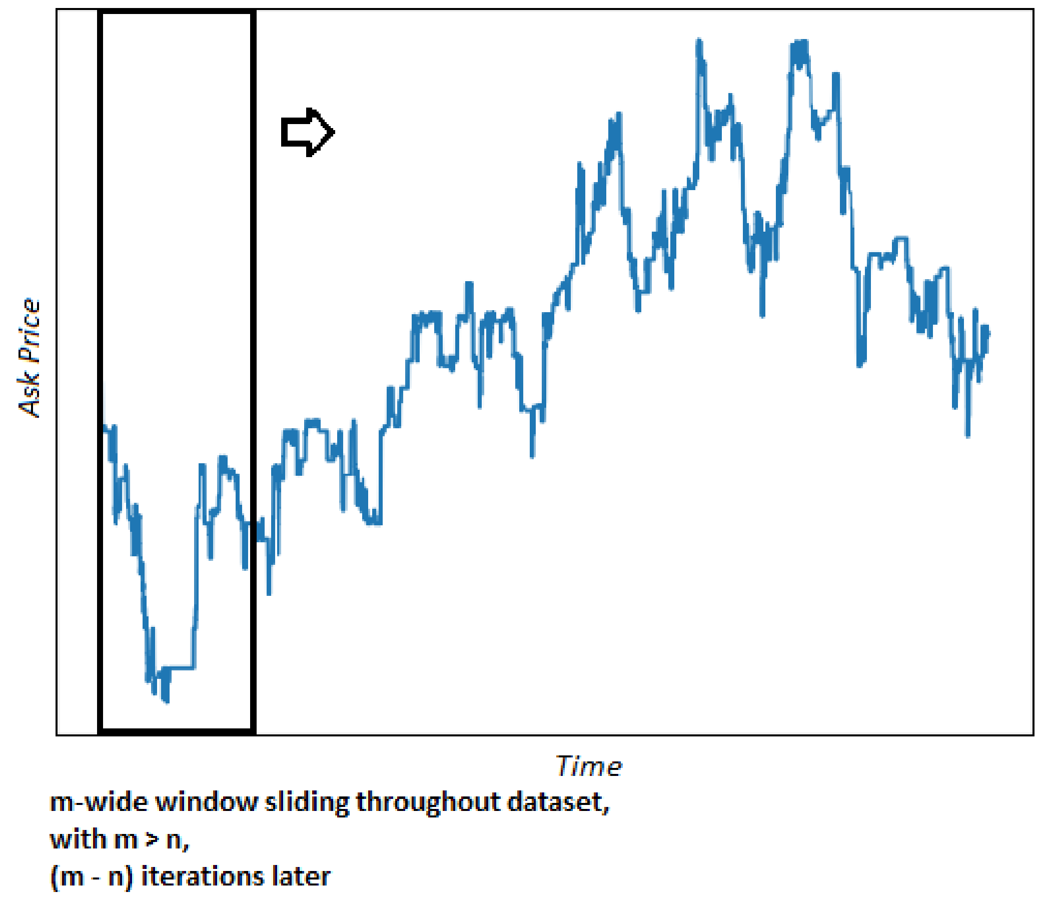


Figure 4.5: Simple visualization of the algorithm (2)

Two main for-cycles are needed: an external cycle providing the sliding window dimension and an internal cycle sliding the provided window throughout the dataset.

4.2.2 Pattern Discovery

This section's purpose is to illustrate the logics that rule the Pattern Discovery operations within the sliding window. We will use Python as the main programming language: amongst the libraries that will be used, "scipy.signal" is going to be the one used for retrieving important information about the signal portion we will be analyzing. The first action we must perform is finding the current signal portion's relative maxima and minima. Let us consider the case we are sliding a one-minute window throughout our dataset and we find ourselves in the middle of the Pattern Discovery algorithm. Let this be our current signal portion (window):

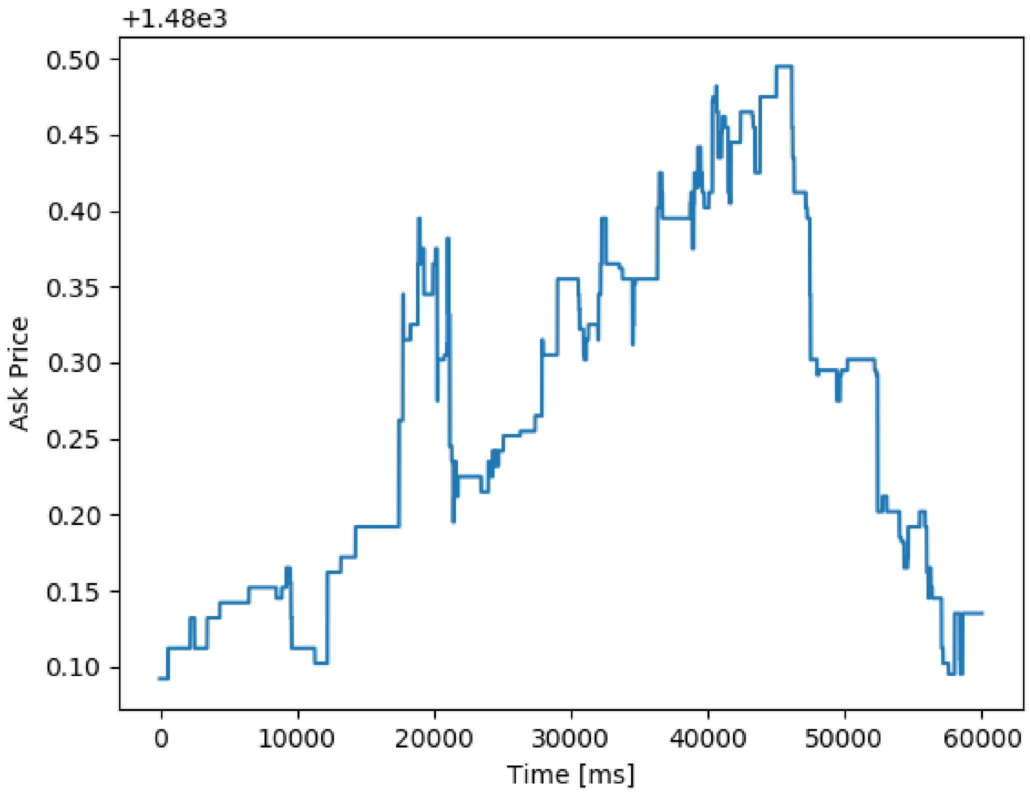


Figure 4.6: A "shot" of the input signal as seen by our sliding window

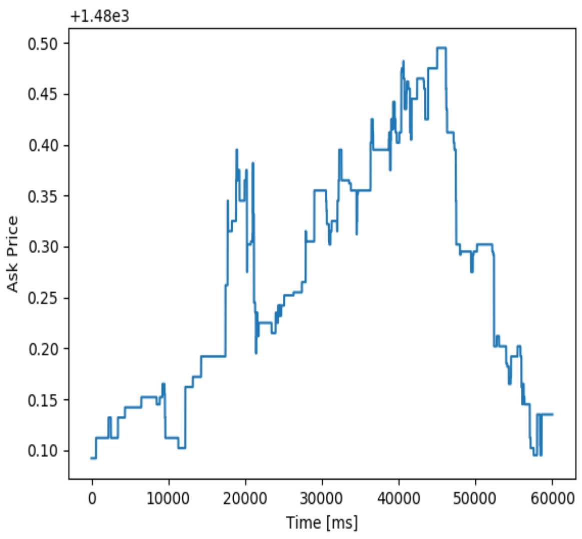
The easiest way to detect relative maxima and minima (positive and negative peaks) is calling scipy.signal.find_peaks function on our window.

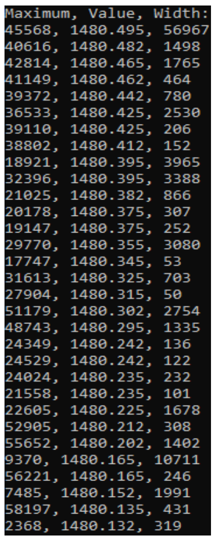
find_peaks() will be called twice: once on the $y = x(n)$ window signal for finding relative maxima, once on the $y' = -x(n)$ window signal for finding relative minima.

This may not be everything we need in order to detect patterns, though. Another important information we can retrieve is the peak width: it is easily retrievable calling peak_widths(), another function offered by scipy.signal.

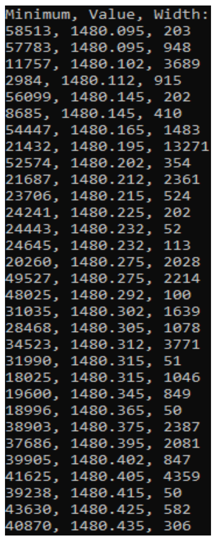
Once we have got all the data we need, we can organize it the way we think is more convenient for us to visually understand: since the most relevant information is the peak's value on the y-axis and its width, we could sort our (peak_position, peak_value, peak_width) tuples in a convenient order, considering the couple (peak_value, peak_width).

We will use a descending order for relative maxima and an ascending order for relative minima.





```
Maximum, Value, Width:
45568, 1480.495, 56967
40616, 1480.482, 1498
42814, 1480.465, 1765
41149, 1480.462, 464
39372, 1480.442, 780
36533, 1480.425, 2530
39110, 1480.425, 206
38802, 1480.412, 152
18921, 1480.395, 3965
32396, 1480.395, 3388
21025, 1480.382, 866
20178, 1480.375, 307
19147, 1480.375, 252
29770, 1480.355, 3080
17747, 1480.345, 53
31613, 1480.325, 703
27904, 1480.315, 50
51179, 1480.302, 2754
48743, 1480.295, 1335
24349, 1480.242, 136
24529, 1480.242, 122
24024, 1480.235, 232
21558, 1480.235, 101
22605, 1480.225, 1678
52905, 1480.212, 308
55652, 1480.202, 1402
9370, 1480.165, 10711
56221, 1480.165, 246
7485, 1480.152, 1991
58197, 1480.135, 431
2368, 1480.132, 319
```



```
Minimum, Value, Width:
58513, 1480.095, 203
57783, 1480.095, 948
11757, 1480.102, 3689
2984, 1480.112, 915
56099, 1480.145, 202
8685, 1480.145, 410
54447, 1480.165, 1483
21432, 1480.195, 13271
52574, 1480.202, 354
21687, 1480.212, 2361
23706, 1480.215, 524
24241, 1480.225, 202
24443, 1480.232, 52
24645, 1480.232, 113
20260, 1480.275, 2028
49527, 1480.275, 2214
48025, 1480.292, 100
31035, 1480.302, 1639
28468, 1480.305, 1078
34523, 1480.312, 3771
31990, 1480.315, 51
18025, 1480.315, 1046
19600, 1480.345, 849
18996, 1480.365, 50
38903, 1480.375, 2387
37686, 1480.395, 2081
39905, 1480.402, 847
41625, 1480.405, 4359
39238, 1480.415, 50
43630, 1480.425, 582
40870, 1480.435, 306
```

Figure 4.7: List of maxima and minima for the current window "shot"

As we can see from above, many relative maxima and minima have come up, even though the analyzed window is rather small. This is normal since a commodity market value changes very frequently.

It is worth noticing that the widest positive peak has been detected at position 45568 ms with a 56967 ms width, information that is very intuitive if we look at the chart. We now have all the important material we will need in order to outline our first complete definition of pattern:

1. The complete set of positive and negative peaks' positions
2. The peaks' corresponding ask price values
3. The peak widths

Peaks and peak widths are meaningful in the identification process of "Perceptually Important Points" (PIPs), elements introduced in [15] that coincide with our definition of "Filtered Peaks" (see the "Peak Filtering" section). Yet, how can we determine whether a window contains information that resembles something we have already seen? How can we store pattern information?

Before saving a pattern there are a few things we should verify: if some conditions are not met, the pattern will be labeled as invalid and ignored.

These are the conditions that a pattern must respect in order to be labeled as valid and stored:

1. The signal must not be a monotonic function, meaning we must be able to detect at least one relative maximum or minimum in it;
2. The signal's start and end values ($x(0)$ and $x(window_{width} - 1)$) must be fairly similar, meaning the signal cannot be approximated to a monotonic function with any down-sampling operation (provided that the chosen sampling frequency used to carry out the aforesaid operation is such that the resulting function's support size is at least three and samples were taken in correspondence of all the existing relative maxima).

The constraints we have just defined will help us direct computations to meaningful data, pruning the algorithm's search space and lowering execution times. Although this kind of validation has never been performed in previous literature, it is of extreme importance: the concept of "pattern" we are building our project upon is a period of time that starts with a setting and terminates with the same, no matter what happens inside of it (as long as something happens). It is the same if we think of patterns in weather: a season will come back 9 months after it ends, no matter what happens during that time.

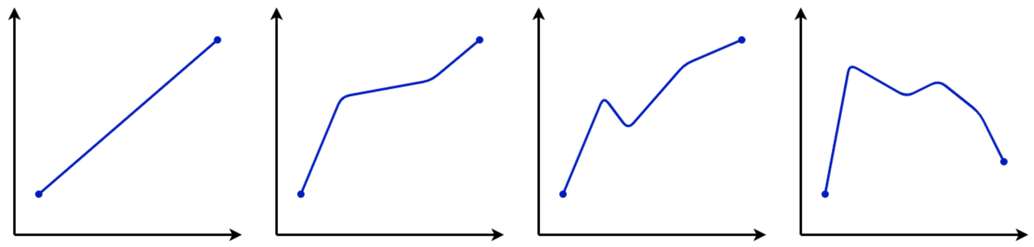


Figure 4.8: Pattern validation examples

The pictures above show four examples of patterns, amongst which the first two are invalid both for conditions 1 and 2, the third is valid for condition 1 only, the fourth is valid overall.

We can now dive right into the operations we want to perform in order to properly store and retrieve pattern information easily.

Storage

The data we refer to in this section is to be intended as the temporary information we want to store in order to eventually detect a correlation between two consecutive patterns. The actions performed once the correlation has been detected will be analyzed later.

The information we need to save and eventually restore is the following:

1. **Timestamps**:

   The pattern's end timestamp in milliseconds (relative to the analyzed dataset)

2. **Start and End coefficients**:

   The pattern's start and end normalized values on the y-axis (given the pattern's lowest value and its height, the start and end normalized values are obtained from the ratio $\frac{s/e\,value - lowest\,value}{height}$)

3. **Positive peaks** (filtered – see the "Peak Filtering" section) **and absolute maximum**

4. **Negative peaks** (filtered) **and absolute minimum**

This information is stored every time a new signal is generated from the sliding window, being aware of the conditions a pattern must respect in order to be saved: if the signal does not have any peaks or if the absolute value of ($start\,coefficient - end\,coefficient$) is greater than or equal to $\frac{1}{3}$, the current signal is ignored and the window is shifted one position to the right. It is worth reminding that in such a case not only is not the pattern stored, but the retrieval operations are no longer executed.

Let us now focus on the amount of stored data we want to keep handy: we cannot store every single pattern we detect throughout the execution without any intelligent disposal operation. As we slide our window throughout the dataset, we will then get rid of unnecessary data: patterns that are older than 2 times the window width are discarded, for we want to detect consecutive patterns only (almost consecutive patterns, at least). Needless to say, each time a window has been slid across the entire dataset and a new window width is generated from the external cycle, all the data relative to the Pattern Discovery and valid for that window is discarded.

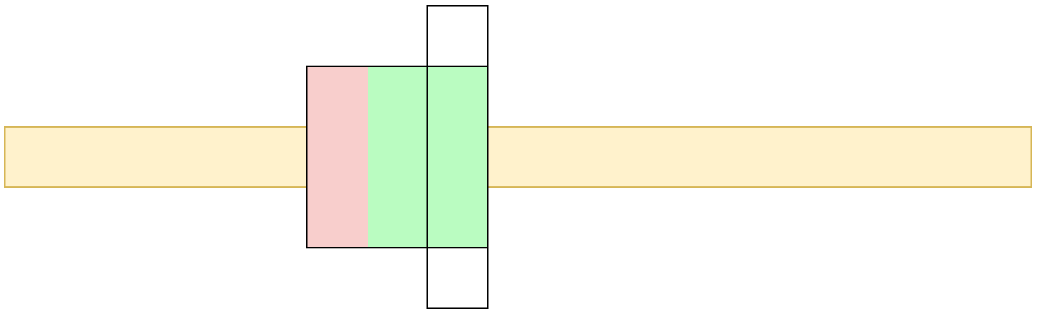


Figure 4.9: A conceptual visualization of the storage window

Let us look at the picture above for a better understanding.
The yellow, horizontal rectangle is our dataset.
The transparent, vertical rectangle is our sliding window.
The multicolor square is our storage window: the green portion represents the actual width of our stored data; the red portion is the oldest pattern's data: although its end timestamp (information 1) is within the green window, the important information (2, 3, 4) refers to the portion of data that is right behind.

Retrieval

Once we have ascertained the validity of the current pattern and stored its information for what comes next, we are ready to perform the most delicate operation of the entire project: understanding whether what we are seeing is something we have already seen. The actions we are going to perform revolve around some basic comparison concepts, but the risk of not choosing an intelligent way to carry them out is to fall into an inefficient procedure. The first question that should pop up is: do we have to visit the entire storage window (the multicolor square) every time we shift the sliding window (the transparent, vertical rectangle) in order to carry out the compare operations? Yes, if a brief answer is what we are looking for.

Without putting much effort into choosing the best way to carry things out and opting for a normal approach, running times grow in such a way that the algorithm's worst-case complexity becomes $O(n \cdot 2\omega)$ for a single sliding window to go through the entire dataset, provided the window width ω. We will analyze quicker approaches later, let us just consider the normal approach for now:

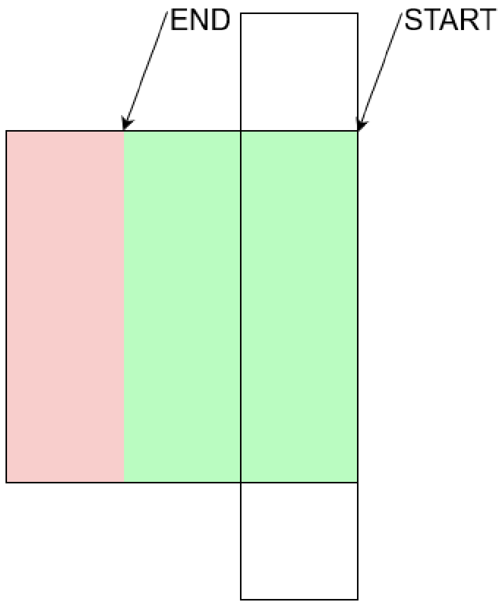


Figure 4.10: A conceptual visualization of the storage window (2)

The picture explains it all: we must compare the current pattern to the already existing patterns (within the storage window) in order to detect similarities.

It is evident that, since patterns that intersect themselves cannot be defined as consecutive, the first half of the storage window (coming from the right side) is not worth considering. "Why do we have to visit it anyways then? Why can we not jump right in the middle and take it from there?", someone may wonder. The answer to these questions is: the storage window does not have a fixed size (we will motivate this decision further in this section, let us just see it as a matter of fact for now).

What we can make faster is the retrieval of the first non-intersecting pattern inside the storage window (the first entry past the window's ideal middle cut line): binary search is what we are looking for. The algorithm's worst-case complexity becomes $O(n \cdot (log_2 2\omega + \omega))$, which is much faster in a real case. This can still be improved, though: we will get a little deeper in the performance analysis throughout the "Code Optimization (v1.1): unnecessary iterations over storage window" section.

Let us go back a little now, questioning the validity of our initial choice not to make the storage window fixed in size. As we may have noticed, only patterns that meet precise conditions are stored, otherwise they are discarded: this means that the storage window can be anything from completely empty to completely full in every moment of the execution. The risk of using the approach we have intentionally avoided is to visit half the storage window entirely and repeatedly even when it would not be necessary at all, undoubtedly lowering performances. Unnecessary visit operations could be easily avoided despite using this method, but would it be worth it to sacrifice memory over useless stuff just to see that logarithm disappear from our time complexity function?

It is now time to dive a little deeper in order to understand the pattern recognition logics.

A little premise is essential: the algorithm makes use of a "greedy" approach, so whenever a pattern repetition is detected while visiting the storage window from right to left the inspection terminates with a success status (no further patterns are compared) and the sliding window is shifted one position to the right.

Pattern recognition is a series of comparison operations that are made out of four important subsequent levels:

1. **Start coefficients check**:
   If two patterns share the same start coefficient (with a $\frac{1}{9}$ tolerance) the algorithm brings the comparison to the next level, otherwise it returns failure

2. **End coefficients check**:
   If two patterns share the same end coefficient (with a $\frac{1}{9}$ tolerance) the algorithm brings the comparison to the next level, otherwise it returns failure

3. **Positive peaks check**:
   After the two patterns' positive peaks sets are filtered properly in order to remove unnecessary data, if they share the same positive peak positions (bidimensional, normalized) within their own window (with a $\frac{1}{9}$ tolerance) the algorithm brings the comparison to the next level, otherwise it returns failure

4. **Negative peaks check**:
   After the two patterns' negative peaks sets are filtered properly in order to remove unnecessary data, if they share the same negative peak positions (bidimensional, normalized) within their own window (with a $\frac{1}{9}$ tolerance) the algorithm returns success, otherwise it returns failure

When a pattern repetition is detected the two similar patterns' timestamps are saved onto a variable (local within the internal cycle) containing every timestamp couple that identifies two consecutive patterns, so that as soon as the sliding window has reached the end of the dataset it is very easy to understand how many patterns have repeated themselves and where.

This is a complete example of the pattern's components that are taken into consideration during the pattern comparison operations:

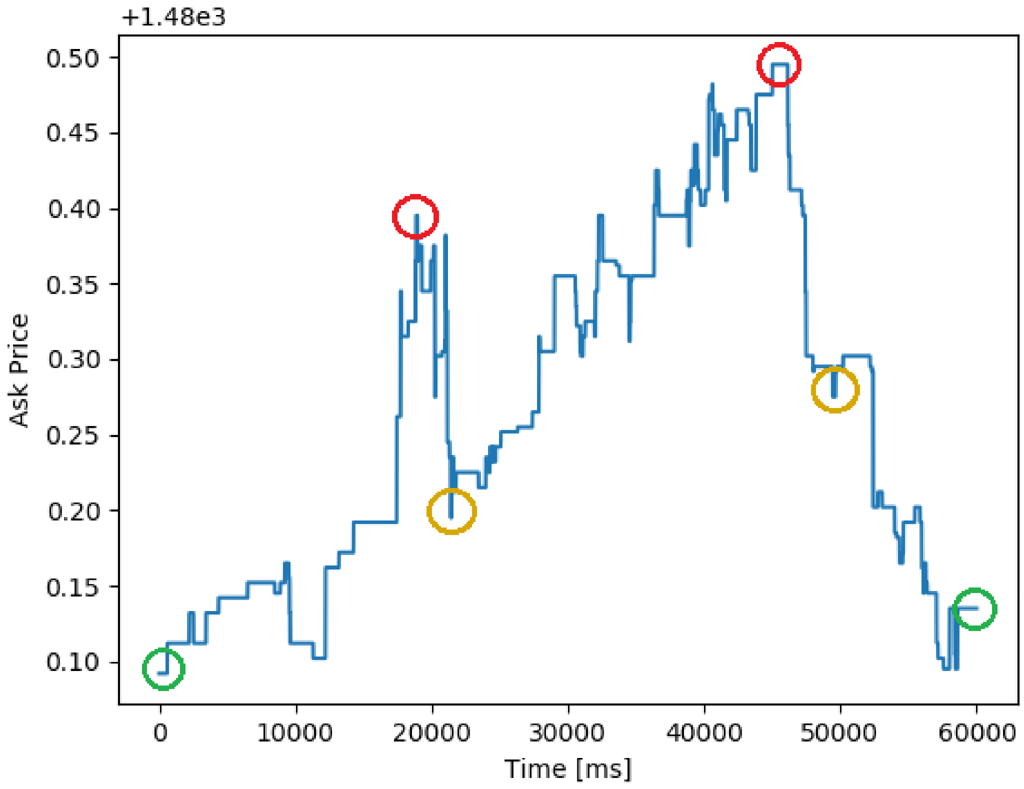
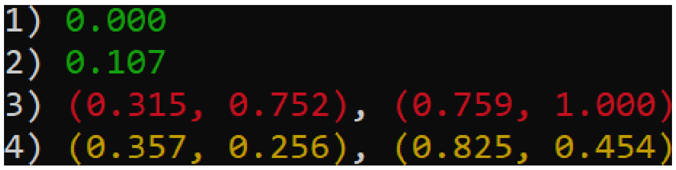


Figure 4.11: Visual example of a pattern's key points in pattern comparison operations



```
1) 0.000
2) 0.107
3) (0.315, 0.752), (0.759, 1.000)
4) (0.357, 0.256), (0.825, 0.454)
```

Figure 4.12: Visual example of a pattern's key points in pattern comparison operations (2)

Let us examine, step by step, what kind of patterns would successfully pass the comparison with the pattern shown in figure 4.11:

1. Only patterns having a start coefficient between 0.000 and 0.111 would be allowed to the next comparison level

2. Only patterns having an end coefficient between 0.000 and 0.218 would be allowed to the next comparison level

3. Only patterns satisfying one of the following conditions would be allowed to the next comparison level:
   - The pattern does not have any positive peak
   - The pattern has one positive peak located within the circumference of radius 0.111 and center in (0.315, 0.752) or within the circumference of radius 0.111 and center in (0.759, 1.000)
   - The pattern has two or more positive peaks, one of which located within the circumference of radius 0.111 and center in (0.315, 0.752) and one within the circumference of radius 0.111 and center in (0.759, 1.000)

4. Only patterns satisfying one of the following conditions would pass the comparison with the pattern shown in the picture successfully:
   - The pattern does not have any negative peak
   - The pattern has one negative peak located within the circumference of radius 0.111 and center in (0.357, 0.256) or within the circumference of radius 0.111 and center in (0.825, 0.454)
   - The pattern has two or more negative peaks, one of which located within the circumference of radius 0.111 and center in (0.357, 0.256) and one within the circumference of radius 0.111 and center in (0.825, 0.454)

Peak Filtering

We have talked about storing, retrieving and comparing patterns alluding to the fact that peaks have to be filtered prior to anything, but it is not clear how it could be done easily. Let us clarify the peak filtering algorithm starting off with a picture:

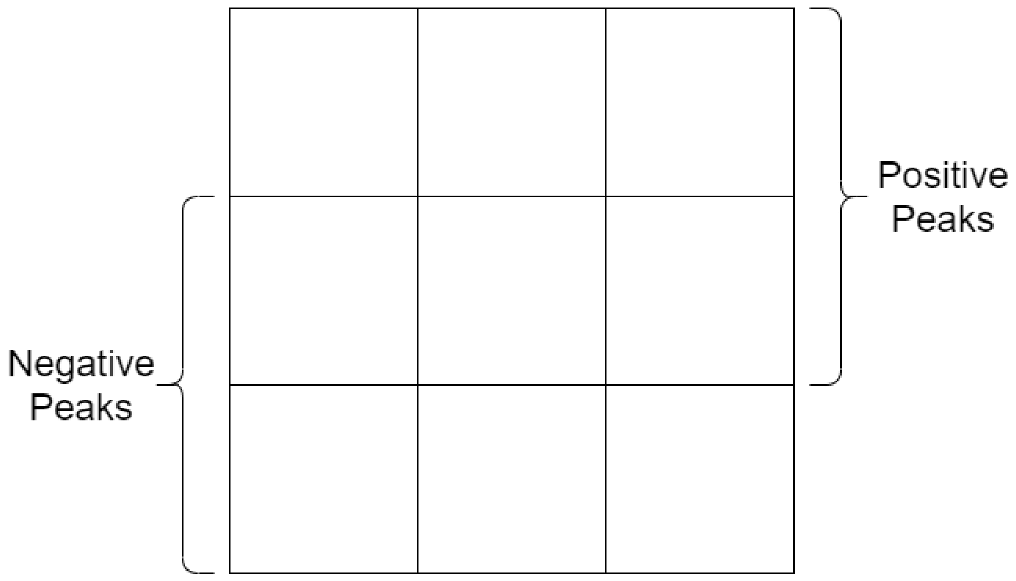


Figure 4.13: Grid schema of the filtered peaks' locations

The normalized pattern is divided into nine equal squares, each of which can contain a number of relevant peaks within its own limits. If we think of the pattern as a matrix called M, the equal squares behave as follows:

- $M_{0,j}$ (any square belonging to the upper row of squares) can contain at most one positive peak;
- $M_{2,j}$ (any square belonging to the lower row of squares) can contain at most one negative peak;
- $M_{1,j}$ (any square belonging to the row of squares in the middle) can contain at most one positive peak (if the square above – $M_{0,j}$ – does

not contain any) and one negative peak (if the square below – $M_{2,j}$ – does not contain any).

We quickly understand that, if we were to identify patterns with a $3x3$ matrix, we could differentiate them in a number of combinations $C = 3^6 = 729$.

How can we decide whether a peak should be of more importance compared to the ones that surround it? The information about the peaks' widths is fundamental at this point: every peak that has a "neighbor" with a bigger width within a $\frac{1}{3}$ radius is considered irrelevant and discarded.

4.2.3 Code Optimization: unnecessary iterations over Storage Window (CPDA v1.1)

Reducing the algorithm's worst-case complexity to almost half the original complexity (as explained in the "Retrieval" section) was a good start, but we do not want to rely on that improvement only: we will try thinking of other simple, potential time-saving upgrades applicable to the linearly complex part of the algorithm. Let us imagine a storage window containing six patterns in its left side (the green-highlighted portion):

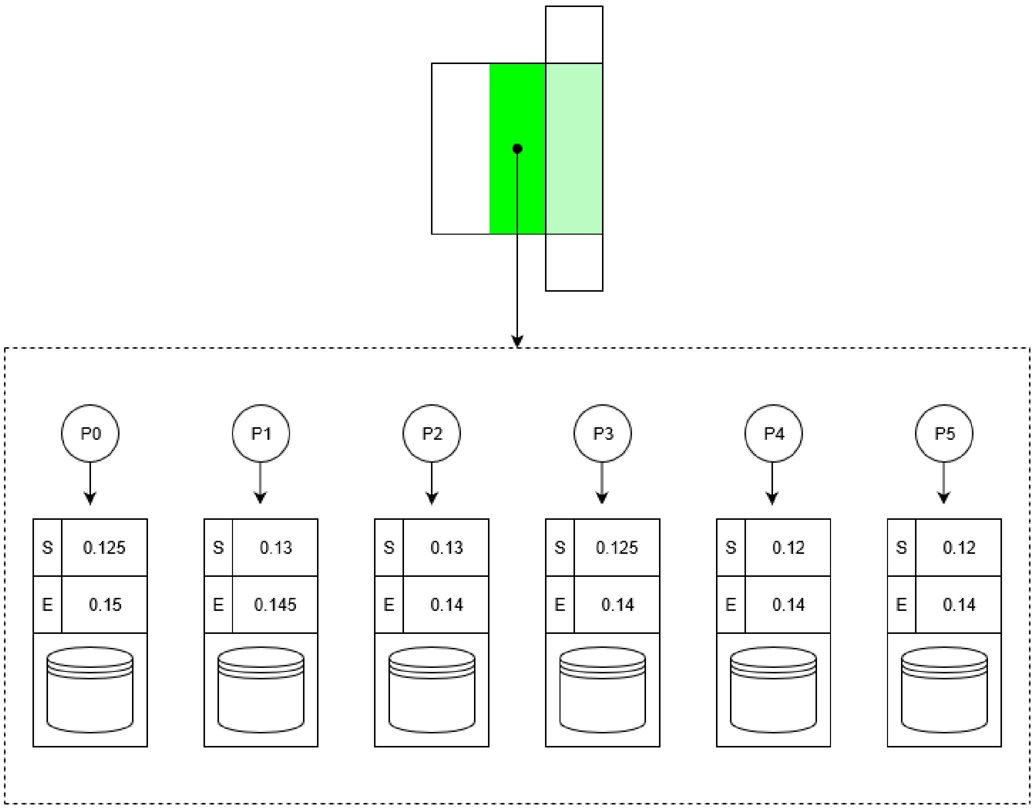


Figure 4.14: A storage window detailed representation

Each pattern contains the start/end coefficients (S/E) and the peaks information, as we can see from the picture above. Let us suppose that the current pattern P_k (the one originated by the sliding window) is compatible with P_0 only: the portion of the algorithm responsible for the pattern retrieval operations has an $O(6)$ complexity, because all the patterns within the storage window get visited in order for P_0 to be identified as P_k's "brother". If we had saved one more information during each pattern's storage operation the situation would be quite different:

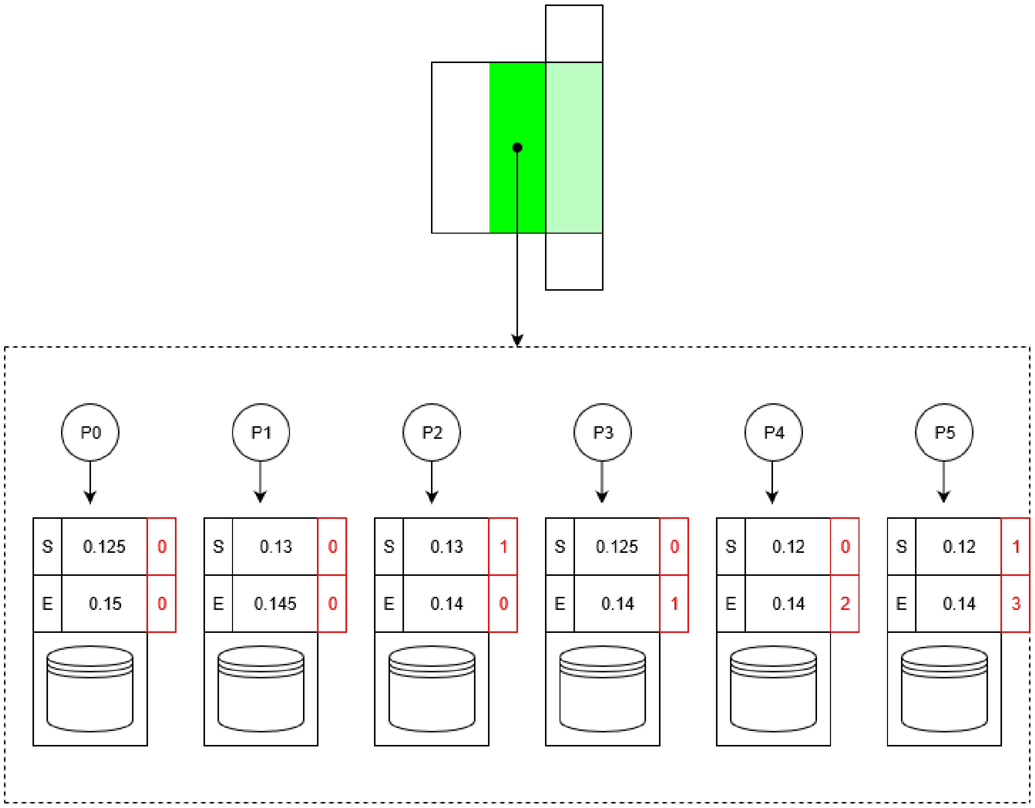


Figure 4.15: A storage window detailed representation (2)

Each start/end coefficient now has an extra information providing the number of times the same coefficient repeats itself in the previously stored patterns. Let us examine what would happen in each case P_k and P_5 should be identified as non-compatible patterns:

- P_k's and P_5's end coefficients are compatible, but their start coefficients are not $\rightarrow$ P_4 is skipped and complexity becomes $O(5)$ or faster;

- P_k's and P_5's start coefficients are compatible, but their end coefficients are not $\rightarrow P_4$, P_3 and P_2 are skipped and complexity becomes $O(3)$;
- P_k's and P_5's start coefficients are not compatible, as well as their end coefficients $\rightarrow P_4$, P_3 and P_2 are skipped and complexity becomes $O(3)$.

Although the algorithm's worst-case complexity does not change a bit, its average-case complexity becomes $O(n \cdot (log_2 2\omega + \omega - X|_\omega))$, assuming the number of elements contained in the storage window being equiprobable at the instant t and $X|_\omega$ being the probabilistic average of a discrete random value ranging from 0 to $\omega - 1$.

It is worth noticing that the probability of this improvement to be successful grows with the sliding window shifting operations being smaller (see the "Early Decisions" section). This is because it is more likely to find similarity in values if they are closer, since the starting signal has been heavily oversampled.

4.2.4 Early Decisions

Before executing the procedure we have been describing so far for the first time, there are a few decisions we must make in order to simplify the overall results and their manageability.

1. Sliding window widths can be multiples of 1000 milliseconds only:
   It is quite difficult to talk about patterns with a millisecond precision, let it be between us developers or to the final user. Furthermore, since the Consecutive Pattern Discovery Algorithm has an intrinsic margin in stating the similarity of two patterns, the information provided by similar windows would be way too redundant, without considering the huge execution times;

2. Sliding windows are shifted along the dataset a fixed number of milliseconds each time:
   Due to the algorithm's intrinsic margin in stating the similarity of two patterns, it is not necessary to shift the sliding window one millisecond each time: the operation would result in high levels of redundancy and large execution times. We will choose a value that is suitable for the algorithm's margin in stating similarities, given the window width. Since the aforesaid operation introduces a new error ϵ, we must prevent it being too big: if the algorithm's margin is $\frac{1}{9} \cdot \omega$, we will decide for the newly introduced error ϵ to be $\frac{1}{81} \cdot \omega \pm$, that is one ninth of the margin (this arbitrary choice derives from the interpretation of ϵ as a second-level margin that is obtainable squaring the previous). The final value of ϵ will be given by the following equation:

   $$2\epsilon\,[ms] \leq \frac{1}{81} \cdot \omega\,[ms] \quad \rightarrow \quad \epsilon\,[ms] \leq \frac{1}{162} \cdot \omega\,[ms] \tag{4.9}$$

   The reason for the error to be halved resides in the fact that two patterns (both affected by the same error) must be compared each time by the algorithm. This is valid as long as we are in this phase: an eventual final implementation of the algorithm may require high precision in detecting patterns, hence smaller shift operations could be taken back into consideration (accordingly to the system's capabilities in terms of performance).

4.2.5 First Results - CPDA v1.1

Table 4.1: Test Details - CPDA v1.1

Execution Start Date	Friday, September 18th, 2020
Execution Start Time	19:05:02 GMT
Execution End Date	Saturday, September 19th, 2020
Execution End Time	13:12:13 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	18 hours, 7 minutes, 11 seconds
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor \; [ms]$
OS	Windows 10 Home v2004, 64 bits
Threads used	1
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
Python version	3.8

The following charts provide a highlight of the most significant results for this test:

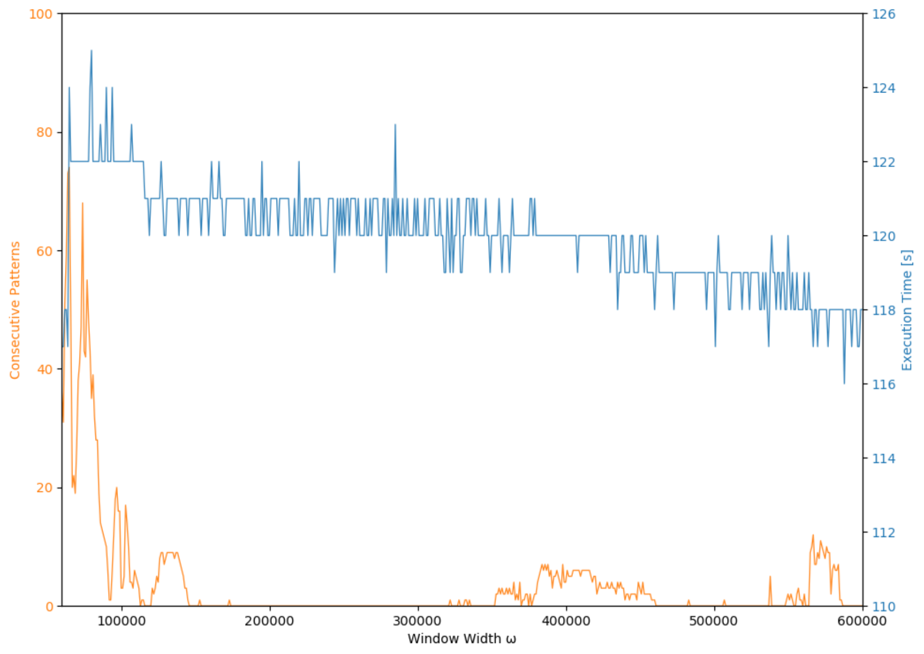


Figure 4.16: Test results overview

The picture above shows the execution time and the number of consecutive patterns found for each of the 541 sliding windows.

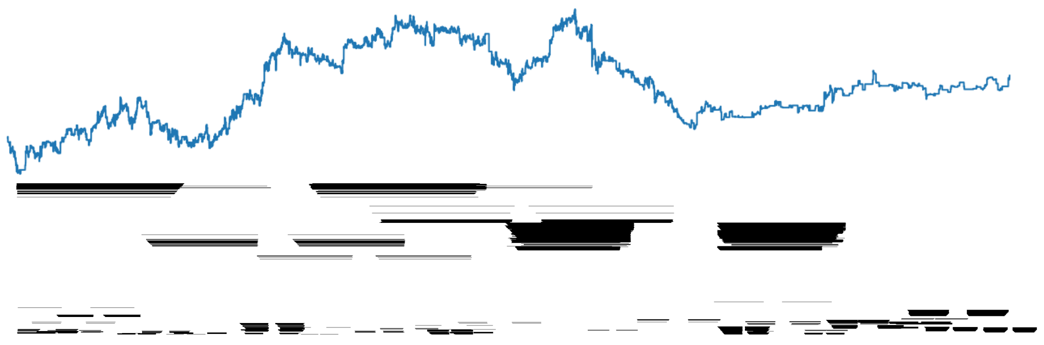


Figure 4.17: Test results overview (2)

The pictures above are the most significant in terms of comprehensibility of our results.

The upper one is a simple chart of the complete dataset, the lower one is a spectrum representing the areas of the dataset that the algorithm has interpreted as repeating patterns. The spectrum must be read as follows:

- The picture is a 541 x 3600 matrix: there are as many rows as the number of sliding windows used to perform the test and as many columns as the number of seconds the complete dataset is wide;
- Each segment corresponds to a sub-signal;
- Each segment is followed by a segment of equal length in its proximity: the two corresponding sub-signals are supposed to be fairly similar in shape;
- There are as many segments in one row as the value of consecutive patterns found with the corresponding sliding window, multiplied by two (a consecutive pattern must be interpreted as two repeating sub-signals – which we have been referring to as “patterns”, so far – in this context);
- Segments can overlap;

- Segments in the spectrum's upper area are obviously larger than the ones in its lower area.

Considerations

We have always considered the algorithm's worst-case complexity as the amount of time needed for a single window to be shifted along the entire dataset, ignoring the fact that there are a lot of windows to be taken into account and that we made some important decisions throughout the project development.

Let us examine the complete, exact worst-case computational complexity $\Omega(n, \lambda)$ of the algorithm we have just executed:

$$\Omega(n,\lambda)|_{\omega_m,\eta,\beta} = O(\sum_{i=1}^{\lambda} \sum_{j=0}^{\lfloor \frac{n-\omega_i}{\epsilon_{\omega_i}} \rfloor} (\phi(\omega_i) + \sigma(\omega_i, \tau_{1,j}, \tau_{2,j}))), \tag{4.10}$$

$$\sigma(\omega, \tau_1, \tau_2) = 1 + \lfloor log_2 \lceil 1 + \tau_1 \frac{2\omega}{\epsilon_\omega} \rceil \rfloor + \lfloor \tau_2 \frac{\omega}{\epsilon_\omega} \rfloor, \tag{4.11}$$

$$\lambda \ll n, \quad \omega_1 = \omega_m, \quad \omega_i = \omega_{i-1} + \eta \quad \forall\, i > 1, \tag{4.12}$$

$$\tau_{1,j} = \frac{j \cdot \epsilon_{\omega_i}}{2\omega_i} \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} < 2\omega_i, \quad \tau_{1,j} = 1 \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} \geq 2\omega_i, \tag{4.13}$$

$$\tau_{2,j} = 0 \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} < \omega_i, \quad \tau_{2,j} = (j - \lfloor \frac{\omega_i}{\epsilon_{\omega_i}} \rfloor) \cdot \frac{\epsilon_{\omega_i}}{\omega_i} \quad \forall\, j \,|\, \omega_i \leq j \cdot \epsilon_{\omega_i} < 2\omega_i,$$

$$\tau_{2,j} = 1 \quad \forall\, j \,|\, j \cdot \epsilon_{\omega_i} \geq 2\omega_i, \tag{4.14}$$

$$\epsilon_\omega = \lfloor \frac{1}{\beta} \cdot \omega \rfloor \tag{4.15}$$

Since the expression seems to differ a lot from the original one (at least from a first look), we could use a rather explanatory, simplified definition:

The algorithm's overall complexity $\Omega(n, \lambda)$ is given by the sum of the λ single-windowed executions' complexities, that are defined, in turn, as the i^{th} number of iterations throughout the complete dataset ($\lfloor \frac{n-\omega_i}{\epsilon_i} \rfloor + 1$), multiplied by the sum of the signal processing operations complexity ϕ and the Pattern Discovery operations complexity σ, both calculated in ω_i.

This should be easy to understand if the mechanism explained in the "Introduction" section has been comprehended: given a certain number of windows, we must make them slide throughout the dataset in order to detect the presence of consecutive patterns. This results in the aforesaid windows capturing ($\lfloor \frac{n-\omega_i}{\epsilon_i} \rfloor + 1$) different dataset portions each. Each portion requires some signal processing operations in order to be properly identified and characterized, along with the Pattern Discovery operations that we analyzed in the "Storage" and "Retrieval" sections.

The number of windows λ is a combination of ω_m, ω_M and η:

$$\lambda = \frac{\omega_M - \omega_m}{\eta} + 1 \tag{4.16}$$

The decision of ω_m, ω_M and η follows common sense and can be trivial, as long as the expert knows the input signal's sampling frequency. Well-defined parameters favor complete results that show low levels of redundancy.

Our time complexity function $\Omega(n, \lambda)$ is intuitively linear with respect to n, but we cannot be as sure for λ. With n being set to a fixed value, would incrementing λ result in a linear increment of Ω? In other words: is the algorithm equally fast for two different values of ω? If not, how slower (or faster) is it for higher ωs? And why does the execution time grow (or decrease) with values of ω being higher?

Although the test results provide an approximate answer to all these questions, we might be interested in a more complete explanation of what is happening inside our code.

Let us write the time complexity function in a different way:

$$\Omega(n,\lambda)|_{\omega_m,\eta,\beta} = O(\sum_{i=1}^{\lambda}\sum_{j=0}^{\lfloor\frac{n-\omega_i}{\epsilon_{\omega_i}}\rfloor}(\phi(\omega_i)+\sigma(\omega_i,\tau_{1,j},\tau_{2,j}))) = O(\sum_{i=1}^{\lambda}\Psi(n,\omega_i))), \tag{4.17}$$

$$\Psi(n,\omega) = (\lfloor\frac{n-\omega}{\epsilon_\omega}\rfloor+1)\cdot\phi(\omega) + \sum_{j=0}^{\lfloor\frac{n-\omega}{\epsilon_\omega}\rfloor}\sigma(\omega,\tau_{1,j},\tau_{2,j}) = \phi'(n,\omega)+\sigma'(n,\omega) \tag{4.18}$$

In order for $\Omega(n,\lambda)$ to be (sub)linear with respect to λ, $\Psi(n,\omega)$ has to be constant (or decreasing) with respect to ω, meaning the sum of its contributions $\phi'(n,\omega)$ and $\sigma'(n,\omega)$ has to be constant (or decreasing) with respect to ω itself.

We do not have a problem in comprehending that the Pattern Discovery function $\sigma'(n,\omega)$ is a decreasing function, having the following inferior limit:

$$\lim_{\omega\to\ n}\sigma'(n,\omega) = \lim_{\omega\to n}\sum_{j=0}^{\lfloor\frac{n-\omega}{\epsilon_\omega}\rfloor}\sigma(\omega,\tau_{1,j},\tau_{2,j}) \cong$$

$$\cong \lim_{\omega\to n}\sum_{j=0}^{\lfloor\frac{n-\omega}{\epsilon_\omega}\rfloor}(1+\lfloor log_2\lceil 1+\tau_1\frac{2\omega}{\epsilon_\omega}\rceil\rfloor+\lfloor\tau_2\frac{\omega}{\epsilon_\omega}\rfloor) = 1 \tag{4.19}$$

What about $\phi'(n,\omega)$? We do not know the signal processing functions' complexity because scipy.signal documentation does not provide any information about it: we can draw some conclusions, though.

Let us suppose $\phi(\omega)$ being linear and approximate $\phi'(n,\omega)$ as follows:

$$\phi'(n,\omega) = (\lfloor\frac{n-\omega}{\epsilon_\omega}\rfloor+1)\cdot\phi(\omega) \cong \frac{n-\omega+\epsilon_\omega}{\epsilon_\omega}\cdot\phi(\omega) \cong \frac{n-\omega+\frac{\omega}{\beta}}{\frac{\omega}{\beta}}\cdot\phi(\omega) =$$

$$= \frac{\beta n-\beta\omega+\omega}{\omega}\cdot\phi(\omega) = \frac{\beta n-\beta\omega+\omega}{\omega}\cdot\omega = \beta n-(\beta-1)\cdot\omega \Rightarrow$$

$$\Rightarrow \phi'(n,\omega) \cong \beta n - (\beta - 1) \cdot \omega \tag{4.20}$$

$$\lim_{\omega \to n} \phi'(n,\omega) = \lim_{\omega \to n} (\beta n - (\beta - 1) \cdot \omega) = n \tag{4.21}$$

Our conclusions are: Both $\sigma'(n,\omega)$ and $\phi'(n,\omega)$ are decreasing functions with respect to ω, so $\Psi(n,\omega)$ is decreasing with respect to ω itself. Since the latter is a decreasing function, $\Omega(n,\lambda)$ is sublinear with respect to λ.

The above statements are explainable in a less mathematical way: since our dataset has a start and an end (i.e. is limited), the initial creation of the sliding window subtracts more computational matter to the dataset with the sliding window being wider. This would not be valid if the algorithm operated on a continuous stream of data, obviously. Will Ψ be constant or linear, then?

The only missing piece to our time complexity puzzle is the information about the ratio between $\phi'(n,\omega)$ and $\sigma'(n,\omega)$ in this context: how complex are both compared to each other?

The same test has been performed a second time without the Pattern Discovery contribution $\sigma'(n,\omega)$ for this question to be answered properly.

The ratio between the first execution time T1 and the second execution time T_2 is the following:

$$\frac{T_1}{T_2} = \frac{65\,231\,[s]}{65\,040\,[s]} \cong 1.003 \tag{4.22}$$

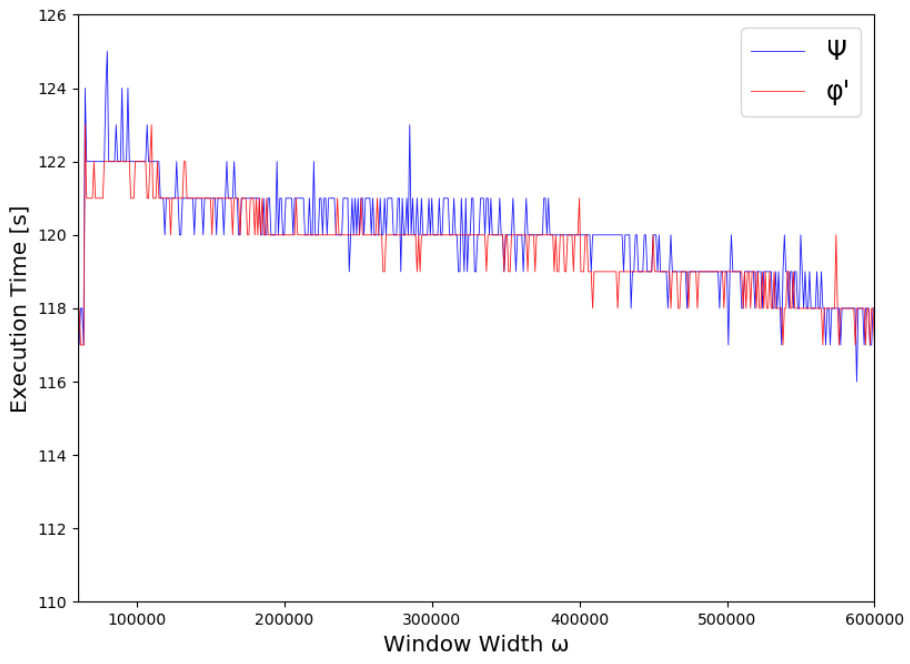


Figure 4.18: Test results overview (3)

Because of this analysis we can say that the portion of the algorithm relative to the Pattern Discovery $\sigma'(n,\omega)$ is negligible with respect to the overall algorithm's complexity:

$$\frac{\sum_i \sigma'(n,\omega_i)}{\sum_i \Psi(n,\omega_i)} \to 0 \tag{4.23}$$

Subtracting its contribution from the overall complexity was, in fact, insufficient for inevitable random factors not to take over and corrupt some results, making us think that removing $\sigma'(n,\omega_i)$ from the algorithm adds complexity to it (nonsense statement). These results are probably caused by $\phi'(n,\omega)$'s nature being much more important than $\sigma'(n,\omega)$'s and, eventually, the storage window being close to empty all throughout the execution: either way, it will not matter whether we improve $\sigma'(n,\omega)$ to its best potential until we invest our resources in increasing the ratio between its contribution and the overall complexity.

Once we have gained enough awareness about the algorithm's behavior in terms of performance and analyzed the technical results, we can move to a much interesting part of our "post-test" section: the pattern analysis output.

We will pick five samples from our result set:

- The very first consecutive pattern found;
- Three random samples from specific areas of the result spectrum, as shown in the pictures below;
- The very last consecutive pattern found.

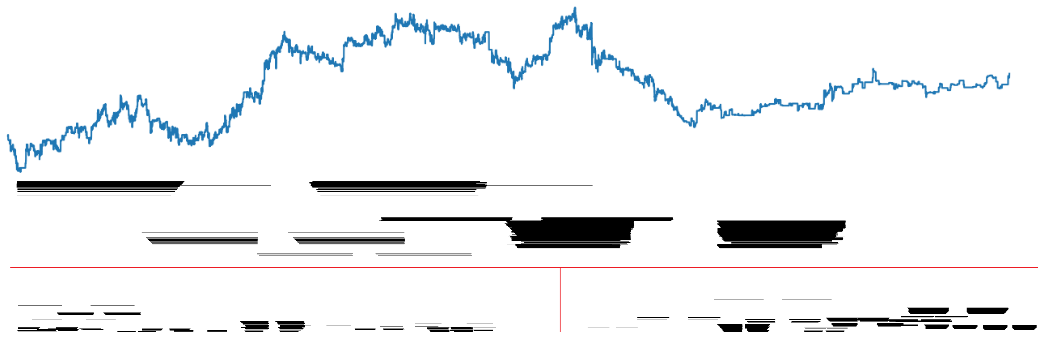


Figure 4.19: Test results overview (4)

Let us start from the very first sample available (belonging to the lower left area of the result spectrum) and compare the two sub-signals identified as repeating patterns:

$\omega = 60000$

$Pattern\,1\,relative\,location:\ [380360, 440359]$ (left)

$Pattern\,2\,relative\,location:\ [450290, 510289]$ (right)

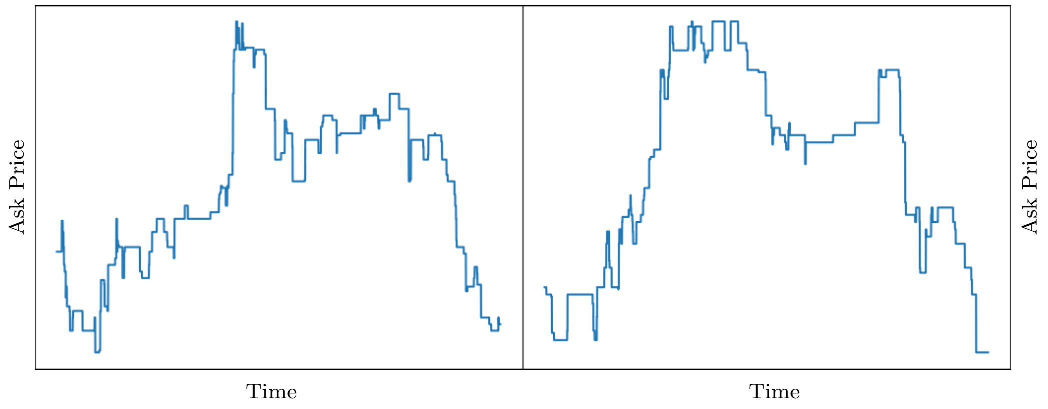


Figure 4.20: Visualization of the very first result (lower left area)

We cannot complain about this first result, for it belongs to the very first example of consecutive patterns we presented in this document.

Let us move to the remaining four samples and see if we feel the same way for each of them.

The following are three random samples picked from the result spectrum's red-circled areas, ordered by ω and relative location:

$\omega = 74000$

$Pattern\,1\,relative\,location: \; [811680, 885679]$ (left)

$Pattern\,2\,relative\,location: \; [941640, 1015639]$ (right)

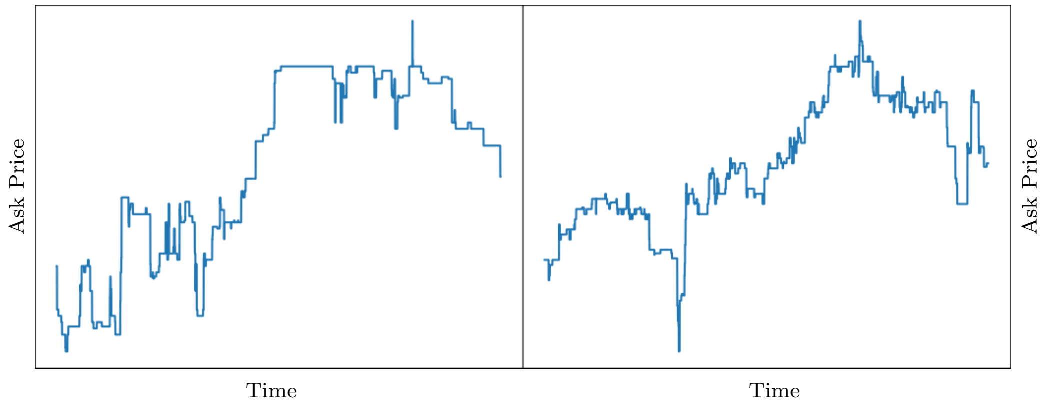


Figure 4.21: Visualization of a random result (lower left area)

$\omega = 74000$

$Pattern\,1\,relative\,location: \; [3415440, 3489439]$ (left)

$Pattern\,2\,relative\,location: \; [3518040, 3592039]$ (right)

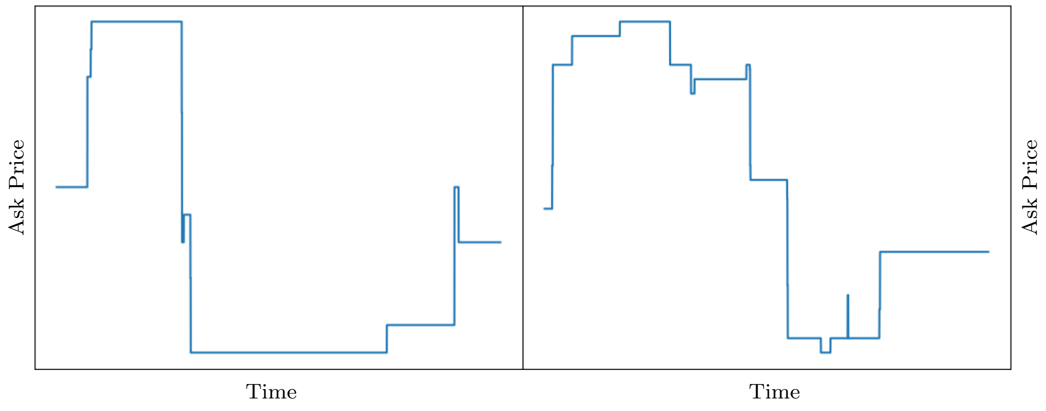


Figure 4.22: Visualization of a random result (lower right area)

$\omega = 407000$

$Pattern\,1\,relative\,location:\ [1765936, 2172935]$ (left)

$Pattern\,2\,relative\,location:\ [2484368, 2891367]$ (right)

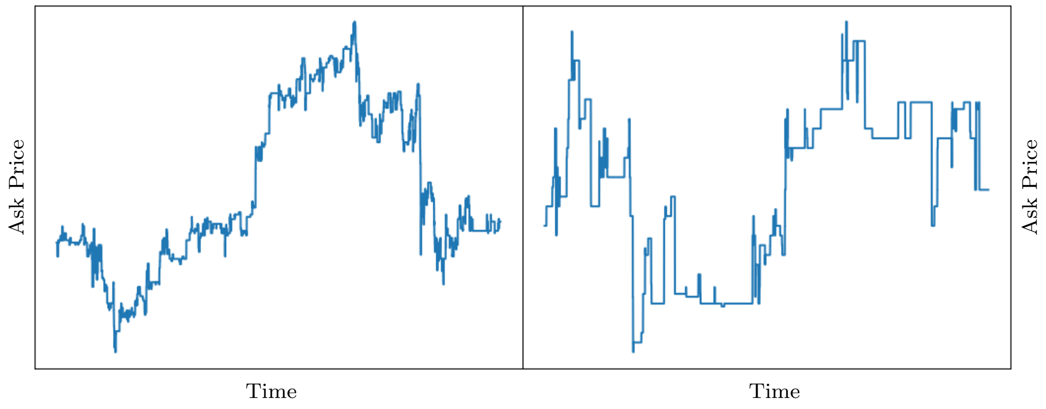


Figure 4.23: Visualization of a random result (upper area)

The following is the very last sample of our result set (belonging to the upper area of the result spectrum):

$\omega = 586000$

$Pattern\,1\,relative\,location:\ [25319, 611318]$ (left)

$Pattern\,2\,relative\,location:\ [1059781, 1645780]$ (right)

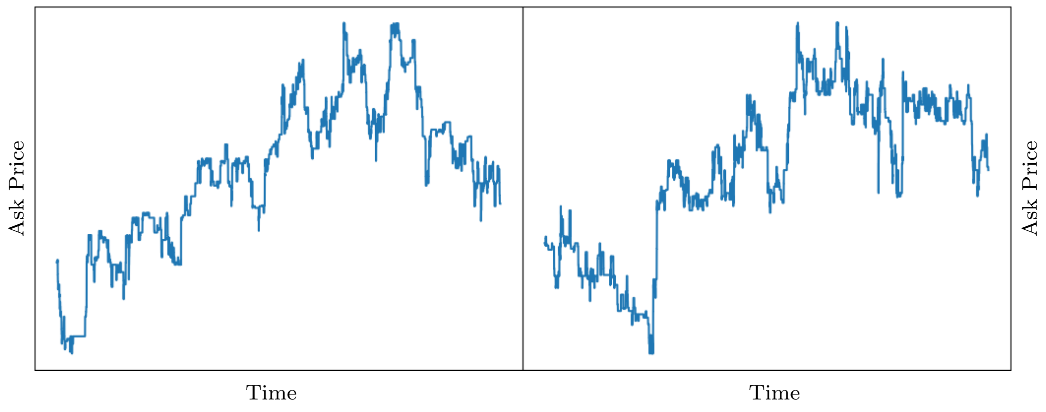


Figure 4.24: Visualization of the very last result (upper area)

Here are some notes about what we have seen so far and the consequent actions that we will perform.

- **Absence of some expected values**:

  We contemplated the idea of creating an algorithm that was capable of detecting at least the repeating patterns that we were able to spot by eye.

  Nevertheless, the area interested by three consecutive patterns analyzed in the "Preliminary Considerations" section (October 1st, 2019, 19:30 – 19:33 GMT) did not receive the same attention by the algorithm as we would expect.

  Let us refresh our memory and identify that area in our test case:

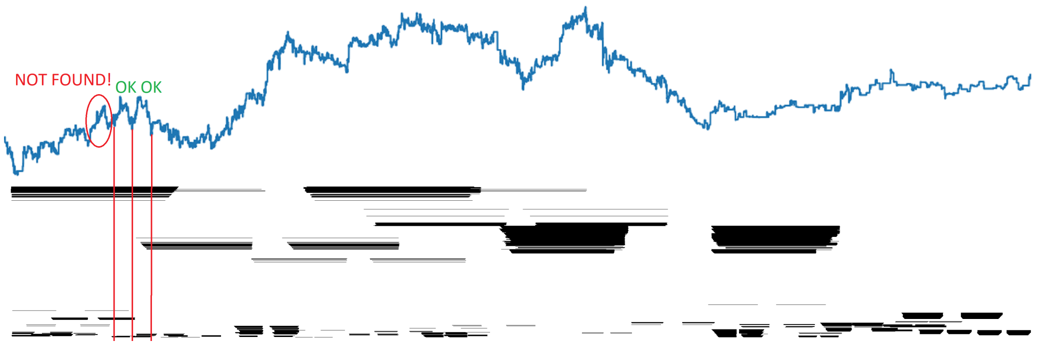


Figure 4.25: An example of missing result

  The algorithm works as it was designed for what we had previously identified as the second and the third repeating patterns. What about the first, though? It looked pretty similar to the others.

  It is worth noticing that the first pattern has something different from the others: its second peak prevails over the first one, probably resulting with the algorithm considering the pattern unique in its shape. It is hard to say whether the current algorithm's behavior is the best approach to a similar problem. We should think through the whole

Pattern Discovery process in a systematic way: the topic will be thoroughly elaborated in the “Pattern Discovery: Tuning & Score System (CPDA v5)” section.

- **Scale**:

  The algorithm is very strict in differentiating patterns on a horizontal scale: patterns of different widths are not compared (we do not want to consider consecutive patterns that are too far apart in terms of width, as little variations are already amortized by a combination of low values of ϵ and η). Can we say the same for its behavior on a vertical scale?

  Let us consider the consecutive patterns identified in the upper area of our result spectrum: it is hard to notice an immediate correspondence between the first and the second pattern, if one scale only is given. It is when we analyze samples individually that we find similarities: this is because vertical scales are different (we must remember that sub-signals are normalized before being stored).

  Is it any good not to care about vertical scales?

- **Redundancy**:

  Let us stay focused on the upper area of the result spectrum. The first thought that comes up when looking at the thickness of these blocks is: “How redundant is this information?”.

  As we approach higher values of ω, redundancy starts growing. This is because, unlike what happens for the shifting value ϵ, the window widths difference value η does not grow alongside ω: we have handled redundancy halfway through (badly).

  Redundancy will be taken care of in the “General Improvements (CPDA v4)” section.

There is a lot of work to do, as we can easily understand.

Leaving redundancy out for a moment, the most legitimate and challenging question is: "How can we best say whether a signal resembles another?".

The best answer to this question is a score system where patterns are related to each other with weights: weights determine the "confidence" of a relationship and help us better understand how important some consecutive patterns are with respect to other, different consecutive patterns.

The following is a prospect of the current and the desired functionalities of the algorithm, along with the sections where the specific improvements are carried on.

Functionality	**Current Behavior**	**Desired Behavior**	**Section**
Redundancy Handling	Rather Permissive	Strict	General Improvements (CPDA v4)
4-Step Comparison	Binary (Yes/No)	Score-Based	Pattern Discovery: Tuning & Score System (CPDA v5)
Vertical Scaling	Premissive	Score-Based	Pattern Discovery: Tuning & Score System (CPDA v5)
Gaps Handling	Permissive (below ω)	Score-Based	Pattern Discovery: Tuning & Score System (CPDA v5)

Table 4.2: Summary of CPDA's actual and desired functionalities

4.2.6 Code Optimization: Single Window (CPDA v2.x)

This section, like many others following, deals with code enhancements that must be tested to provide version congruency. A section called “CPDA: Performance and Congruency tests with different versions” exists for this matter: the same data used in the previous test is analyzed with newer versions and the same results are expected every time, unless the newer version consists of interventions that inevitably change the results’ nature.

Introduction

Transforming the algorithm in such a way that one single visit of our dataset is enough to obtain the same results does not come with effortless performance enhancements. Besides, we must execute the same operations as we did before with a greater amount of flow complexity. Why should we bother considering this option, then?

We said enhancements are not effortless, but they are surely worth the effort: the more we can comprehend their potentiality in such a context, the smoother the transition will be.

The idea is to get the closest we can to a real-time algorithm, inside of which incoming data is stored and analyzed at once, fast: we must be able to make the most out of new data, consuming it before other information arrives. Having a high-level view of the current work domain (every single sliding and storage window) grants us an excellent amount of data exploitation techniques.

Let us see what we would like our code to do in practice:

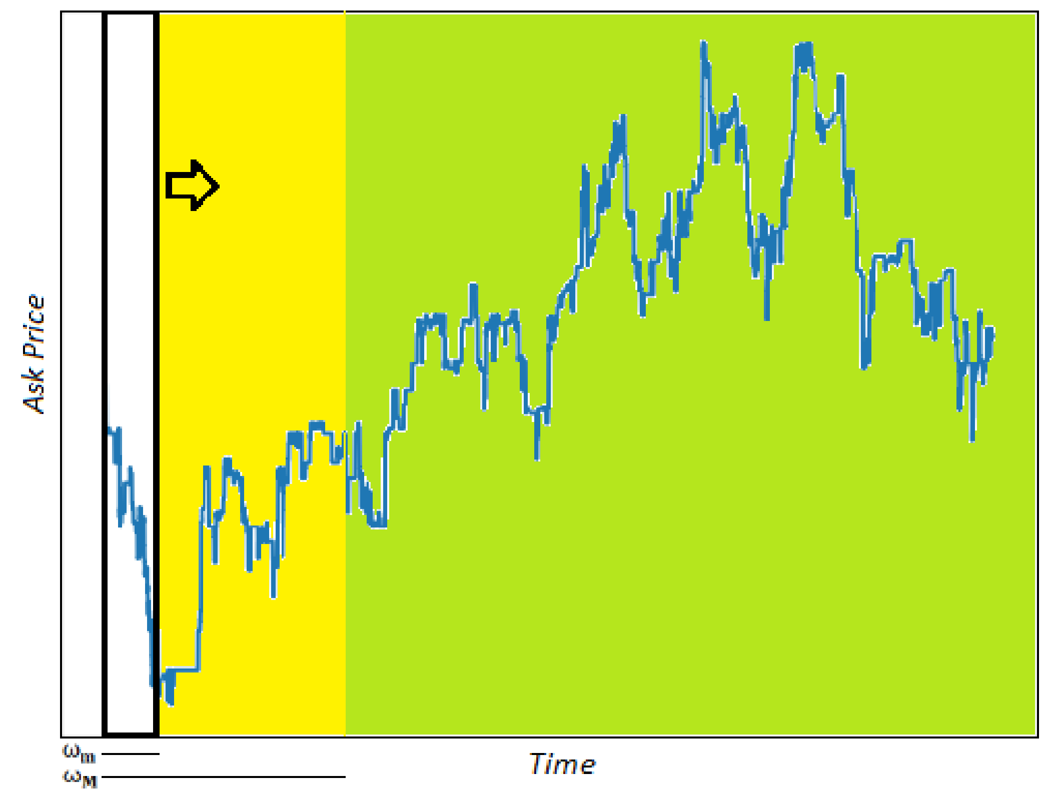


Figure 4.26: High-level visualization of CPDA v2.x

The (supposedly) ω_m-wide sliding window created at the beginning grows bigger until it is ω_M-wide, instead of sliding. While it grows, the super-window incorporates more sub-windows.

It then slides until it reaches the end of the dataset.

First approach to a new way of thinking (CPDA v2.0)

This section explains the criteria that we will adopt translating our code from v1.1 to v2.0, without focusing on optimizing code: this is because there are new concepts that should be introduced beforehand and we do not want to bite off more than we can chew.

Rather than deciding which data structures to be used for storing information (that is intuitive enough), we should question ourselves about the strategies to be applied for handling the shifting operations when sliding our super-window throughout the dataset. Let us visualize the problem in an easier way:

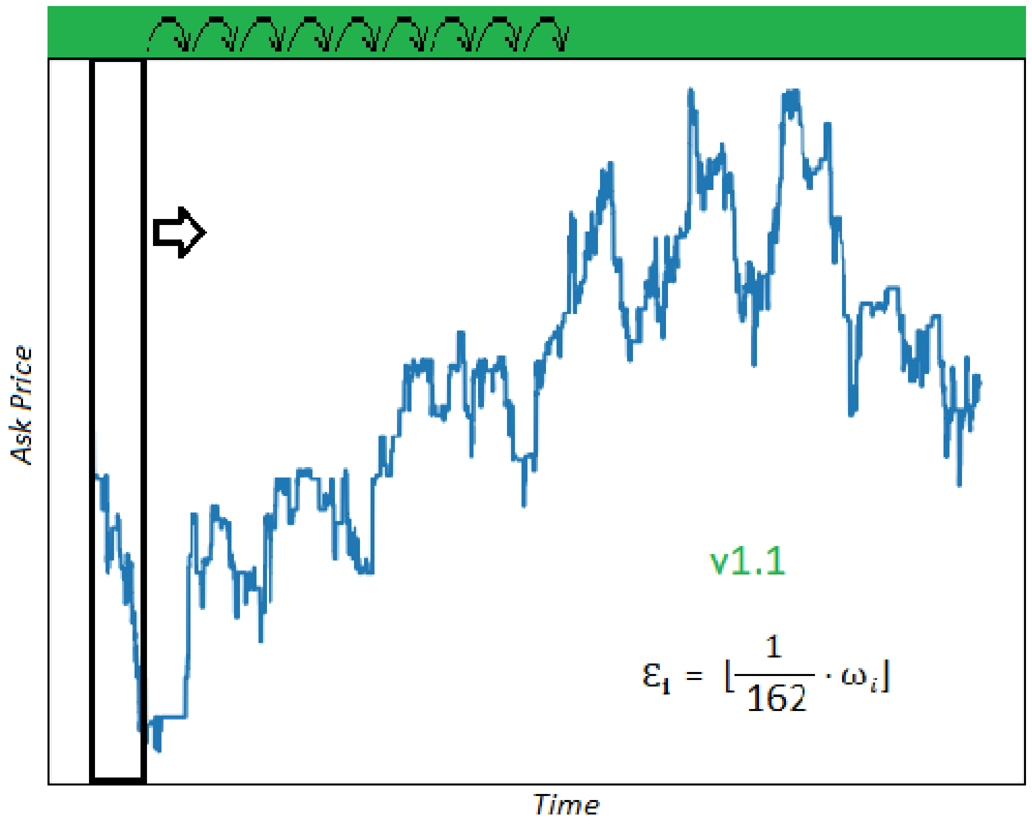


Figure 4.27: Shifting was easy in v1.1 because the jump was determined by the window width

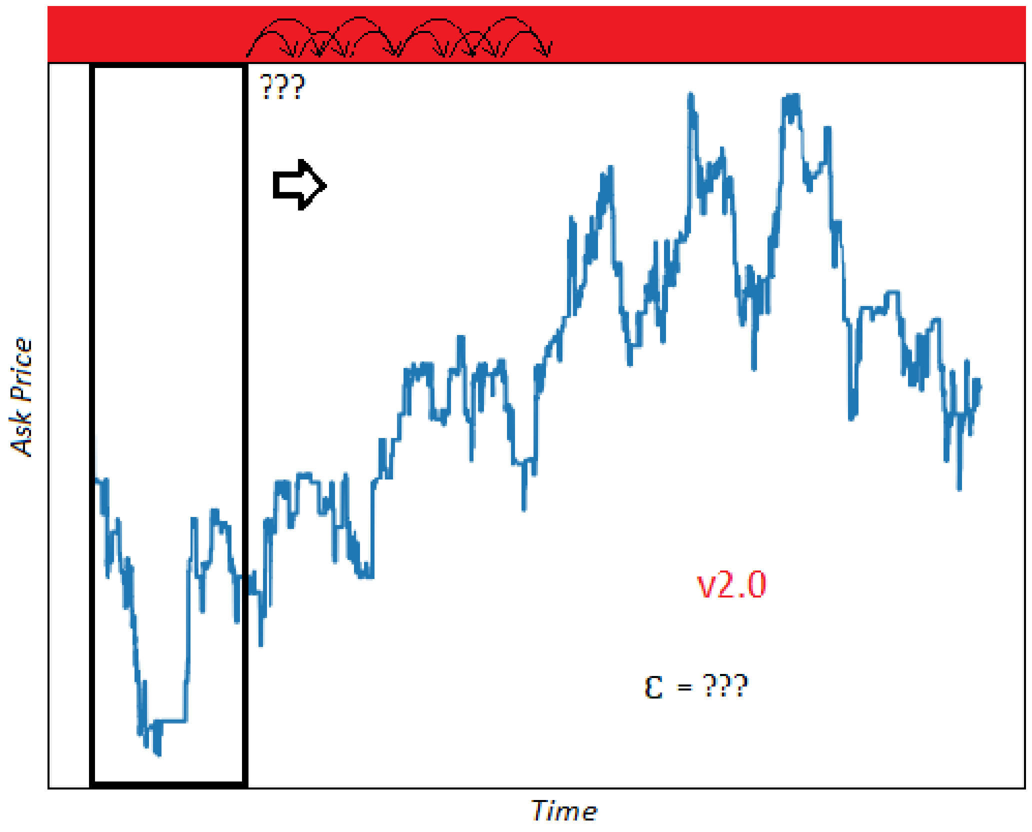


Figure 4.28: Deciding the number of milliseconds for the super-window to be shifted is cumbersome, since it incorporates more sub-windows

There are many ways this problem can be addressed.

It may be a good idea to find a compromise between all the sub-window-specific ϵ_i values.

Which one should we choose anyway? There is no answer, unfortunately: every sub-window has its own pace, so choosing one over another implies an inevitable overlapping of steps. This complicates things significantly, because once we make our step, we should ask ourselves whether we crossed a point another sub-window would have stopped at. It is worth noticing that, in our case, there is no ϵ_i small enough to prevent this uncomfortable situation.

Although this solution is not impossible, we prefer not to consider it due to its complexity.

Still, there is an interesting use of the above-mentioned solution when it is taken to its borderline case: what if we chose ϵ to be one millisecond and asked ourselves each time whether an instant is interesting to any sub-window? Although this may sound counterintuitive in terms of performance, a solution using a similar approach will be chosen by the end of this section. The decision is not trivial, so we will get there through some considerations that will clarify the reasons we want to opt for this solution.

Could we implement a mechanism such that we always know the next shifting hops without the need of minimal shift operations and complex polling? The answer is a booking mechanism: a chronologically ordered data structure providing information about the next shifting hops (bookings) and the sub-windows that booked them (bookers).

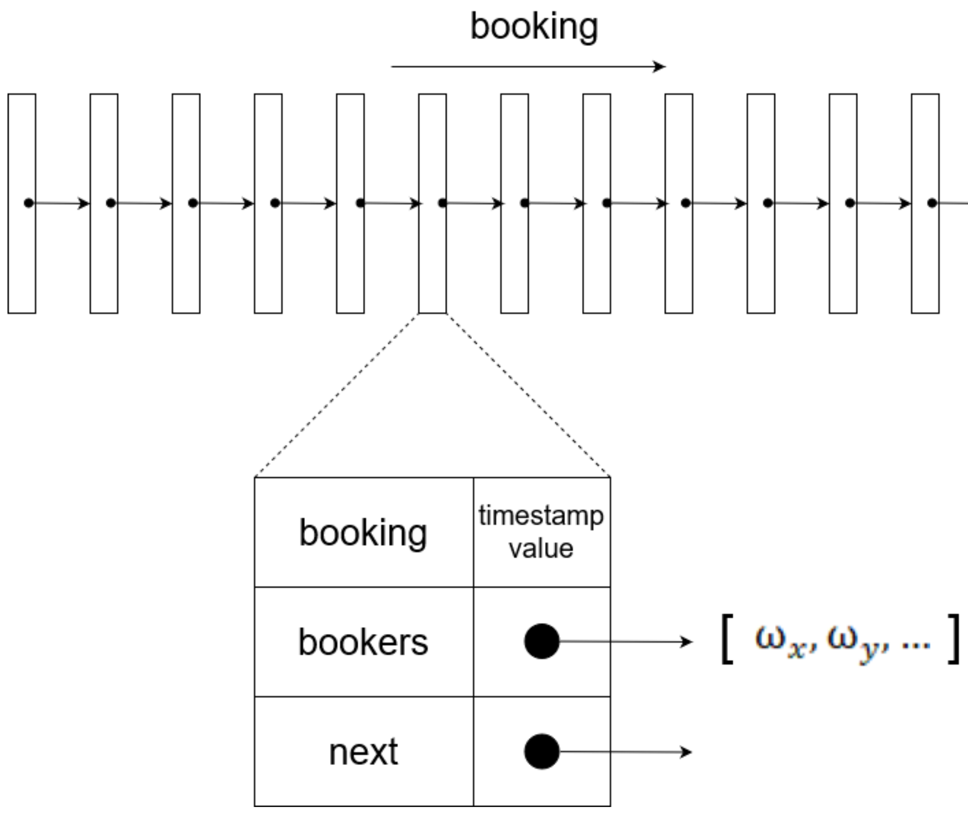


Figure 4.29: A chronologically ordered data structure providing a booking mechanism

How would we implement this solution?

When the execution begins, every sub-window ω_i books the first timestamp they would like to consider for starting their analysis (in our case, $\omega_{60\,000}$ books timestamp $t_0 = 59\,999\,ms$, $\omega_{61\,000}$ books timestamp $t_1 = 60\,999\,ms$ and so on).

When every sub-window has made their reservation, the slot associated to the lowest booking timestamp is popped out of the queue and the related bookers are retrieved (the slot associated to $t_0 = 59\,999\,ms$ is popped out and the only booker $\omega_{60\,000}$ is retrieved).

The (primitive) sliding super-window is populated with dataset values associated to timestamps ranging from 0 to 59 999 and $\omega_{60\,000}$ starts its analysis.

When $\omega_{60\,000}$ is done, a new slot is pushed into the queue as a new reservation for timestamp $t_n = t_0 + \epsilon_{60\,000} = 59\,999\,ms + 370\,ms = 60\,369\,ms$ is made on $\omega_{60\,000}$'s behalf.

At this point, the slot associated to the lowest booking timestamp ($t_n = 60\,369\,ms$) is popped out of the queue, the related bookers are retrieved ($\omega_{60\,000}$ only) and the sliding super-window is populated with new dataset values.

Bookers perform their analyses and the procedure repeats itself until valid slots are present in the queue.

This approach is very appealing, but is it worth the effort?

The following 3 600-pixels-wide picture emphasizes the density of bookings across an hour-wide dataset (same conditions as for v1.1 test): every vertical stripe of pixels illustrates the quantity of booked timestamps (bookings) in a second ([0; 1 000]) through 1 out of 256 shades of red.

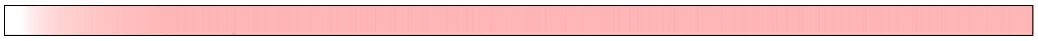


Figure 4.30: Density of bookings

As we can see, no bookings are present in the first minute. The number of bookings per second grows rather fast afterwards, remaining around 300 all throughout the execution.

What if, in the future, we decided that the error $\epsilon\,[ms] = \lfloor \frac{1}{162} \cdot \omega_i \rfloor\,[ms]$ was too big for our needs and we replaced it with the error $2\epsilon'\,[ms] = \lfloor \frac{1}{9} \cdot \frac{1}{100} \cdot \omega_i \rfloor \rightarrow \epsilon'\,[ms] = \lfloor \frac{1}{1800} \cdot \omega_i \rfloor\,[ms]$?

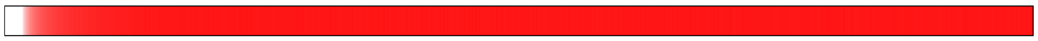


Figure 4.31: Density of bookings (2)

Under these conditions, almost every timestamp would be booked by at least one sub-window, making the use of a dynamically growing, chronologically ordered data structure a burden, besides being unnecessary. Note that the above is not the only case this situation presents itself: the number of bookings per second also changes when the number of windows λ or the window widths difference η are changed.

It would be much easier to pre-allocate a list of slots containing all the bookable timestamps and their relative bookers empty lists right from the start: at this point, the approach is very similar to what we have described previously, without the insertion and removal of slots in and out of the structure.

It is worth noticing that the insertion of bookers into slots is still dynamic: slots are statically allocated when the algorithm starts, but it is still on sub-windows to make their own reservations. This choice will help us prevent computational thickenings in a real-time algorithm, where renewal and disposal operations on the above-mentioned data structure will be needed anyways, at some point.

Lastly, it is important to remember that, in version 2.0, sub-windows still work separately one from the other: every sub-window has its own work domain (sliding and storage sub-windows) and performs analyses locally.

Let us visualize, for both the cases presented above (in order, from left to right), the total number of bookers per second, together with their cumulative cardinality:

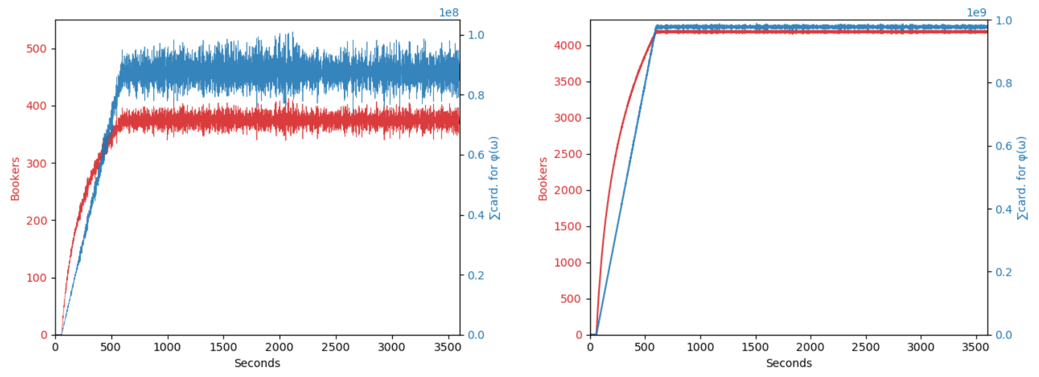


Figure 4.32: Total number of bookers per second and cardinality for two different cases

When the number of bookings stabilizes, the system should handle a considerable amount of independent procedures per second (400 in the first case, 4 000 in the second), aiming to obtain every sub-window's absolute maximum and minimum, peaks and relative widths (ϕ), without considering the Pattern Discovery operations (σ). This is very hard to manage in a real-time context: our job is to make it look easy.

Avoiding repetition of expensive actions in $\phi(\omega)$ - Absolute Maxima and Minima (CPDA v2.1)

What we are concerned the most about is the mind-blowing cardinality that the system must handle. Let us focus on the cause of this happening and suppose we have four bookers in a given timestamp: the algorithm computes absolute maxima and minima for each one of them brutally, starting from scratch every time a sub-window has been dealt with. Would it not be computationally easier to perform one iteration for all the sub-windows at once, keeping track of absolute maxima (or minima) progressively? Afterall, sub-windows are natively hierarchical:

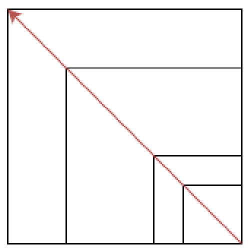


Figure 4.33: The hierarchical nature of sub-windows

Although this bidimensional illustration is not accurate, it will help us conceive the nature of the situation.

If the algorithm finds itself where the red arrow originates, only one single iteration on the biggest sub-window is needed: the smaller sub-windows' absolute maxima are lower than or equal to the bigger sub-windows', as well as the smaller sub-windows' minima are greater than or equal to the bigger ones'.

It is obvious that, for this to work, we need to iterate backwards.

This is a very simple solution that would have a good impact on the overall absolute maxima (or minima) computational cardinality: we would reduce it from $\simeq \frac{10^6}{ms} = \frac{10^9}{s}$ to $\simeq \frac{3.5 \cdot 10^5}{ms} = \frac{3.5 \cdot 10^8}{s}$, removing useless iterations for an approximate 65% of the total cardinality. This strange-looking complexity has been calculated performing the running average (in fully operational conditions) of the biggest booker per millisecond.

Although this may sound it, we must remember that the calculation of absolute maxima and minima is a small part of $\phi(\omega)$: there is still a big component of it (the one that handles the true signal analysis) that does not benefit at all from this improvement.

Besides, is this solution satisfying enough? We could potentially range over solutions of different kinds, each of which being better than the others for some aspects and worse for others: we will analyze one alternative only (solution 'b') and compare it to the one we have just introduced (solution ‘a’), providing pros and cons of both and making a final decision.

The alternative that we are about to introduce is more articulated than the first one, because it aims to reduce the local (booking-specific) complexity to 0. Let us picture a new data structure that keeps track of every sub-window’s absolute maximum (or minimum):

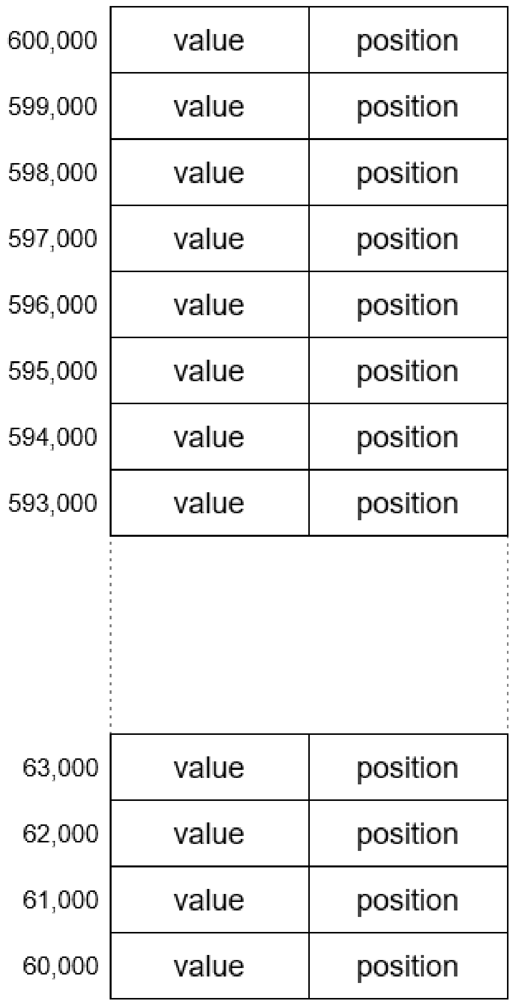


Figure 4.34: Data structure used for solution ’b’

In our case, this structure is a 541-elements-long dictionary containing, for each sub-window, two pieces of information: its absolute maximum (or minimum) and the relative position. Solution ‘b’ constantly updates the dictionary so that, when it comes to retrieving a sub-window’s maximum (or minimum), direct access is guaranteed.

If it were easy to keep the structure updated, we would not have any doubt in choosing this solution over the other. The problem is that, when new data is collected (i.e. when the super-window shifts), we may require operations of different nature to be carried on: we may need to update the dictionary after a new maximum (or minimum) has arrived, as well as re-iterate portions of already collected data after a maximum (or minimum) has expired. We may not need to do anything at all because the structure is up to date, enjoying direct access on the structure without the necessity of updating it.

This is enough to convince ourselves that we cannot think of an average millisecond cardinality as easily as we did for solution 'a', because it varies with respect to the incoming data. Although it may not be clear for the reader yet, we can be sure of this solution's complexity for three cases only: the incoming data is constant ($O(541)$ for both maxima and minima), increasing ($O(541)$ for maxima, $O(6 \cdot 10^5)$ for minima) or decreasing ($O(541)$ for minima, $O(6 \cdot 10^5)$ for maxima). Every intermediate case is hard to classify.

How can we determine which solution is better, then?

Before making the decision, it is important to understand exactly how solution 'b' works.

What makes this solution most interesting is the nature of the vocabulary's values: since sub-windows are hierarchical, maxima and minima are naturally ordered. Let us keep in mind the above illustration of the vocabulary and try to contextualize it inside the execution's maxima management: if the incoming data contains a value that is greater than (or equal to) the smallest sub-window's maximum, we can update the structure upwards until a greater value is found, explaining why complexity is $O(541)$ for incoming data being constant or increasing. If, otherwise, we find ourselves dealing with a stream of decreasing data, we will have to re-iterate the portion of already collected information corresponding to the super-window ($6 \cdot 10^5$ elements), because maxima are positioned at the very beginning of each sub-window and expire, as a consequence, every time the super-window is shifted.

What happens when the incoming data contains a value that is greater than (or equal to) some sub-windows' maxima and lower than all the other bigger sub-windows'? The following illustrations will help us comprehend how the solution behaves in a general case.

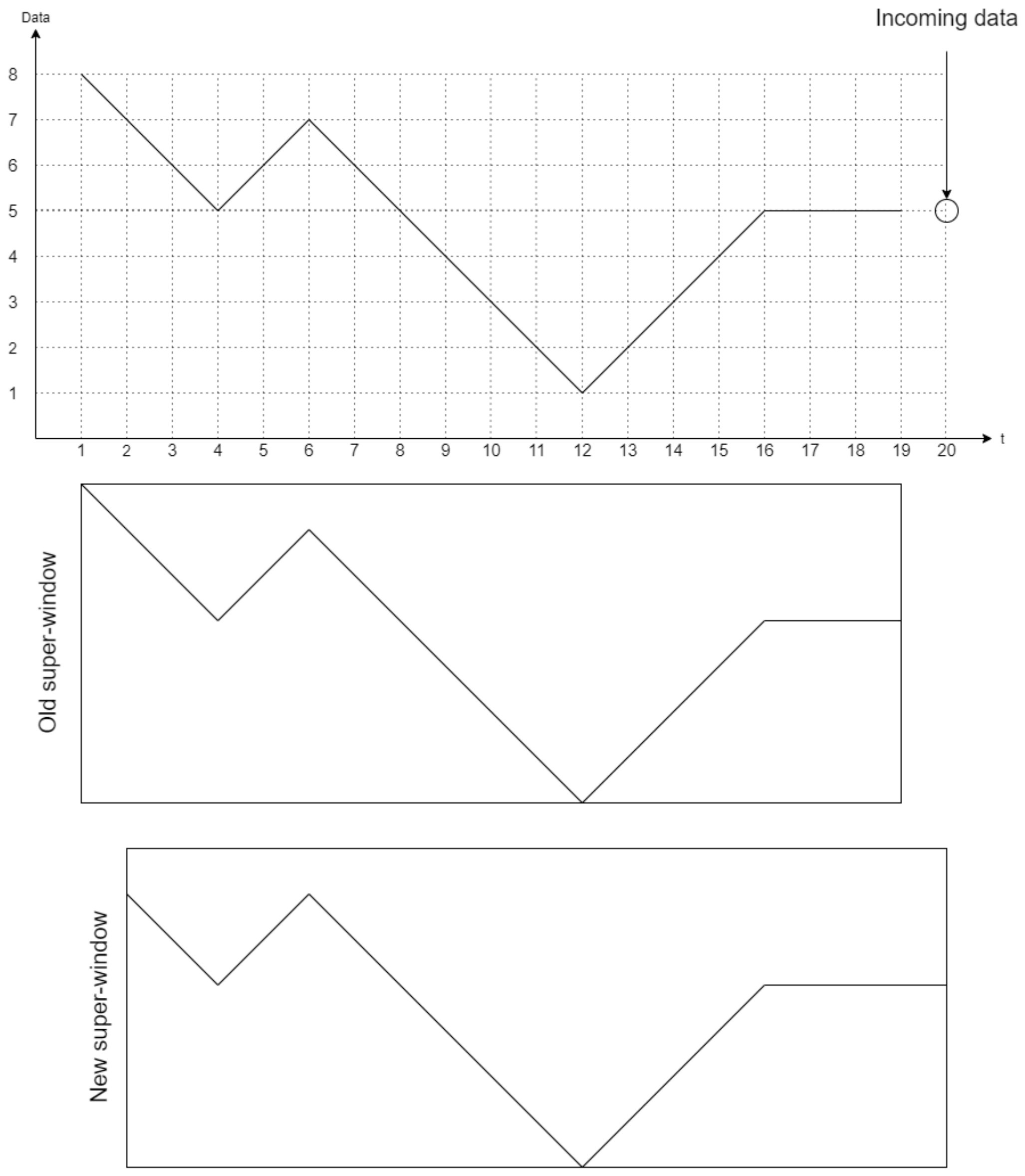
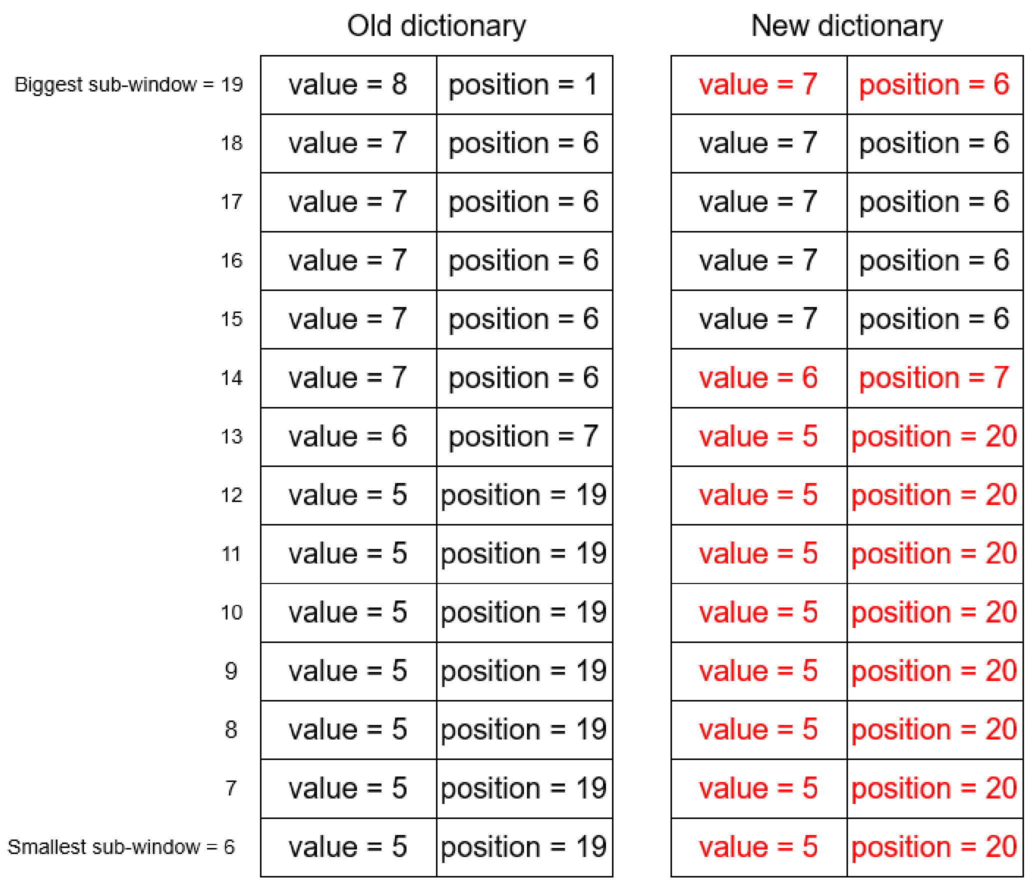
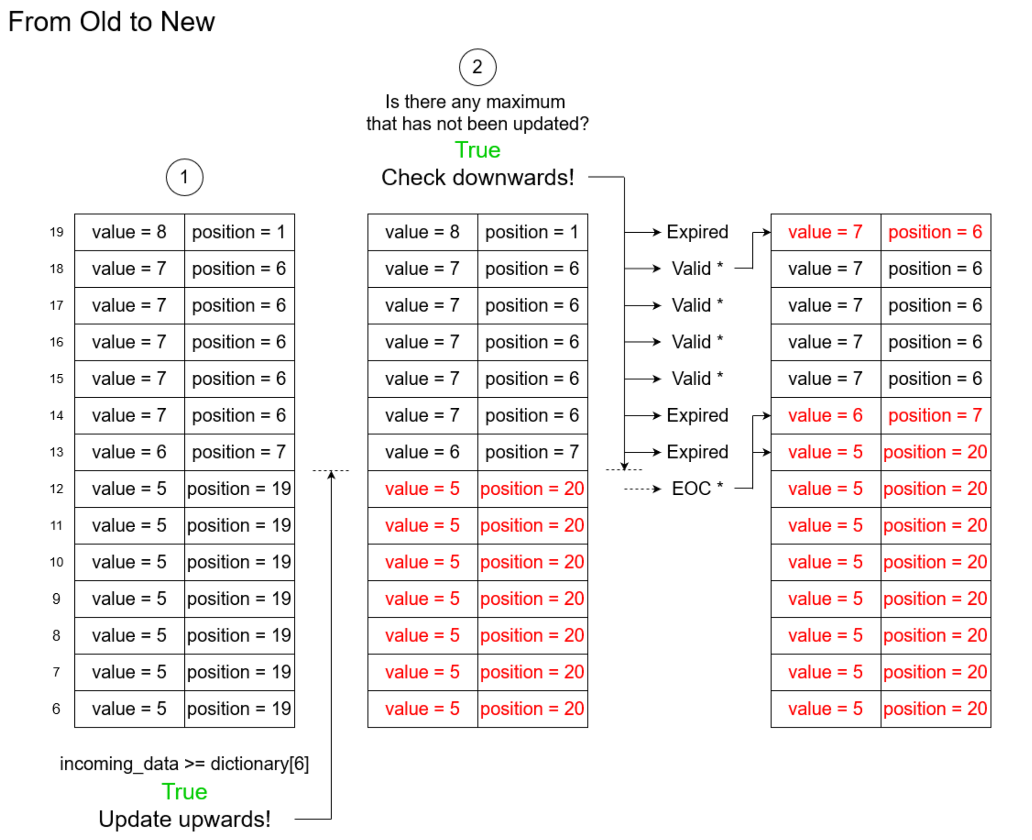


Figure 4.35: Solution 'b' visualized



	Old dictionary		New dictionary	
Biggest sub-window = 19	value = 8	position = 1	value = 7	position = 6
18	value = 7	position = 6	value = 7	position = 6
17	value = 7	position = 6	value = 7	position = 6
16	value = 7	position = 6	value = 7	position = 6
15	value = 7	position = 6	value = 7	position = 6
14	value = 7	position = 6	value = 6	position = 7
13	value = 6	position = 7	value = 5	position = 20
12	value = 5	position = 19	value = 5	position = 20
11	value = 5	position = 19	value = 5	position = 20
10	value = 5	position = 19	value = 5	position = 20
9	value = 5	position = 19	value = 5	position = 20
8	value = 5	position = 19	value = 5	position = 20
7	value = 5	position = 19	value = 5	position = 20
Smallest sub-window = 6	value = 5	position = 19	value = 5	position = 20

Figure 4.36: Solution 'b' visualized (2)



19	value = 8	position = 1	value = 8	position = 1	value = 7	position = 6
18	value = 7	position = 6	value = 7	position = 6	value = 7	position = 6
17	value = 7	position = 6	value = 7	position = 6	value = 7	position = 6
16	value = 7	position = 6	value = 7	position = 6	value = 7	position = 6
15	value = 7	position = 6	value = 7	position = 6	value = 7	position = 6
14	value = 7	position = 6	value = 7	position = 6	value = 6	position = 7
13	value = 6	position = 7	value = 6	position = 7	value = 5	position = 20
12	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20
11	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20
10	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20
9	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20
8	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20
7	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20
6	value = 5	position = 19	value = 5	position = 20	value = 5	position = 20

Figure 4.37: Solution 'b' visualized (3)

* When a checked value results valid or the downwards cycle ends, re-iteration is carried out backwards over the cumulative portions of data for which values resulted expired consecutively, exploiting hierarchy the most. Let us consider the end of our downwards cycle as an example:

1. The missing (expired) portion of data relative to the 13-wide sub-window is checked (position 8 only) and the relative maximum is acquired (5 at position 8)

2. The resulting relative maximum is compared to dictionary[12]: the greatest (or the most recent if equal) is assigned to dictionary[13]

3. The missing (expired) portion of data relative to the 14-wide sub-

window is checked (position 7 only) and the relative maximum is acquired (6 at position 7)

4. The resulting relative maximum is compared to dictionary[13]: the greatest (or the most recent if equal) is assigned to dictionary[14]

In our case, the downwards cycle ends before the bottom of the dictionary is reached: this is because some values were updated by the upwards cycle. What if no maximum was updated, the downwards check reached the bottom of the dictionary and the last value resulted invalid? The End Of Cycle would trigger the backwards re-iteration of data, but the number of re-iterated elements would obviously change. The difference between our example and the mentioned case is described below: the smallest sub-window is completely rechecked in the latter.

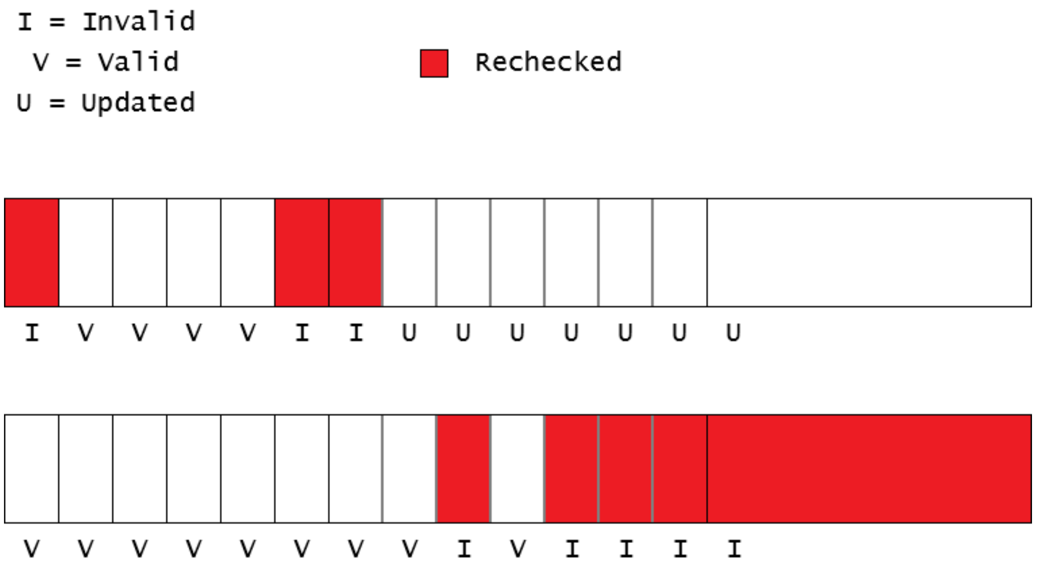


Figure 4.38: Solution 'b' visualized (4)

Everything being explained, it is important to clarify a few things before proceeding. (1) What is the incoming data? (2) Why do we update values and positions even when it is constant?

As we know, not every timestamp is booked, especially when the algorithm starts running. We may have to let the super-window slide without performing any analysis: (1) the incoming data shown in the previous pictures is the absolute maximum of all the values corresponding to consecutive instants without bookers.

(2) When incoming data is constant, maxima and minima get updated for their expiration to be postponed, so that no unnecessary re-iteration is made.

There is something more about expiration that has not been mentioned so far: it is mandatory to iterate the dictionaries in order to find expired values, but it is not necessary to do it in correspondence of every booking timestamp. We may exploit an additional variable that keeps track of the lowest expiration (both for maxima and minima) and prevents us from iterating the dictionaries needlessly, bringing the booking-specific complexity to 0 in some execution areas.

We are now ready to make our final decision between solution 'a' and solution 'b': how can we proclaim the winner? There is no better way to do this than running both, getting a sense of both solutions' behavior in terms of speed. Surprisingly enough, solution 'a' is visibly slower than both v2.1b and v2.0, meaning it worsened performances instead of improving them.

Two changes were made to v2.0 in order to obtain v2.1a:

- Creation of functions used to get sub-windows' maxima and minima instead of built-in max() and min()
- Invocation of sort() on bookers in order to perform backwards scan

Since we cannot give up on any of these edits (nor can we improve them somehow), solution 'b' will be the one used to perform v2.1 test.

Avoiding repetition of expensive actions in $\phi(\omega)$ - Signal Analysis (CPDA v2.2)

What follows is one of the most important improvements that the learning phase of the project will see. As we mentioned previously, the computationally thickest part of $\phi(\omega)$ resides in the signal analysis operations: the peaks finding and the peak widths retrieval.

As of v2.1, every booker performs its computations independently and repeatedly. We will develop a solution that exploits data continuity to avoid the repetition of expensive actions, but we will first understand how peaks are found and peak widths are retrieved.

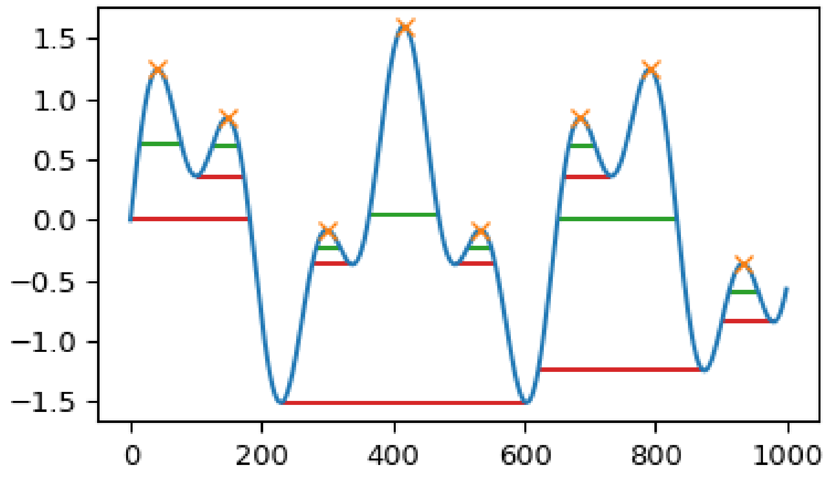


Figure 4.39: Peaks and peak widths

Source: docs.scipy.org

The picture above shows a signal's peaks (marked with an 'x') and peak widths (illustrated as a horizontal red line).

The scipy.signal's peak_widths function receives a signal together with its peak positions and performs analyses on it: peak prominences are computed first, peak widths are retrieved right afterwards. This is an incredibly heavy process, if it is repeated multiple times.

How can we invest our resources in such a way that the algorithm avoids visiting portions of the dataset multiple times over? We will make use of a strategy that we call "the backwards tunnel", for it operates very intuitively: it all resembles the action of placing flags on the way up to a mountain peak and digging an horizontal tunnel leading back to the flags on the way down to the flat land. Digging a tunnel is effortless in our context: it corresponds to a removal operation from a stack-like structure (LIFO) containing information about the flag position on the x-axis. How are we going to use the easily obtainable width anyways? How can we relate it to a specific peak? How can we retrieve every peak width in each part of the execution if they mutate so easily when changing sub-window sizes and positions? We must notice that widths stretch and compress continuously regardless of their peak horizontal position.

The following illustration (together with a step-by-step analysis of the operations performed to handle positive peaks and their widths) should clear up every doubt.

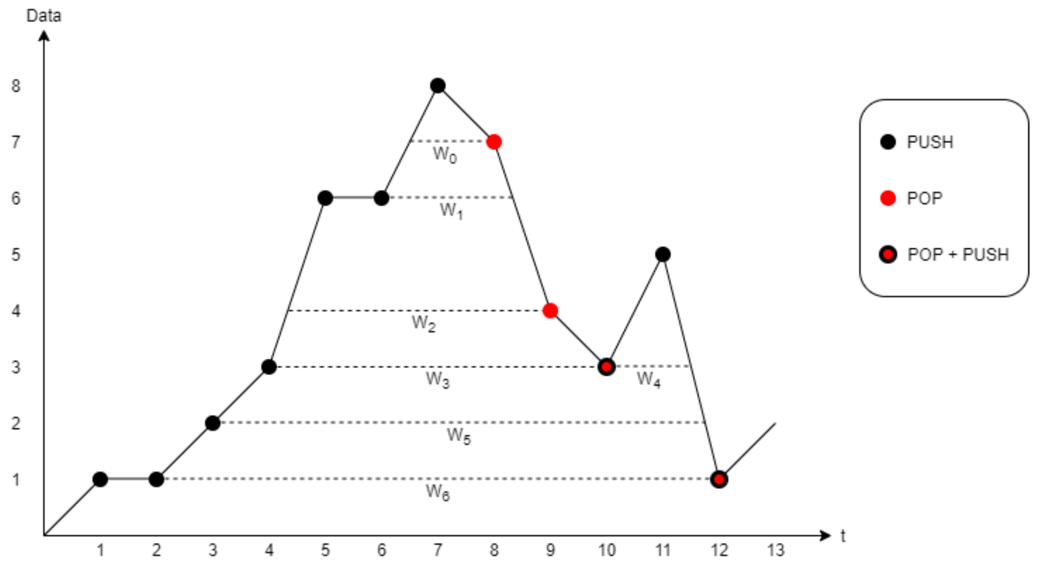


Figure 4.40: Data continuity exploitation in peaks and peak widths

Position	**Action**	**SP[2]**
1	Push: (1, 1, 0)[1]	(1, 1, 0)
2	Push: (1, 2, 0)	(1, 2, 0)
3	Push: (2, 3, 0)	(2, 3, 0)
4	Push: (3, 4, 0)	(3, 4, 0)
5	Push: (6, 5, 0)	(6, 5, 0)
6	Push: (6, 6, 0)	(6, 6, 0)
7	Push: (8, 7, 0)	(8, 7, 0)
8	Pop: (8, 7, 0); Creation of peak (7, 8)[3] associated to label 0[4]; Access to SP[5]; Computation of width $w_0 = 8 - 6.5 = 1.5$ and association of (6, 1.5)[6] to peak 0[7]	(6, 6, 0)

9	Pop: (6, 6, 0),(6, 5, 0); Access to SP; Computation of width $w_1 = 8.33 - 6 = 2.33$ and association of (5, 2.33),(6, 2.33)[8] to peak 0; Computation of width $w_2 = 9 - 4.33 = 4.67$ and association of (4, 4.67) to peak 0	(3, 4, 0)
10	Pop: (3, 4, 0); Computation of width $w_3 = 10 - 4 = 6$ and association of (4, 6) to peak 0; Push[9]: (3, 10, 1)	(3, 10, 1)
11	Push: (5, 11, 1)	(5, 11, 1)
12	Pop: (5, 11, 1); Creation of peak (11, 5) associated to label 1; Pop: (3, 10, 1); Computation of width $w_4 = 11.5 - 10 = 1.5$ and association of (10, 1.5) to peak 1; Pop: (2, 3, 0); Computation of width $w_5 = 11.75 - 3 = 8.75$ and association of (3, 8.75) to peak 0; Pop: (1, 2, 0),(1, 1, 0); Computation of width $w_6 = 12 - 2 = 10$ and association of (1, 10),(2, 10) to peak 0; Push: (1, 12, 2)	(1, 12, 2)

Table 4.3: Data continuity exploitation in peaks and peak widths

[1] (*Value*, *Position*, *Label*)

[2] *Stack Pointer*

[3] (*Position*, *Value*)

[4] *Label→Peak*

[5] *First element in stack is accessed for retrieving the straight-line equation used to calculate the peak width*

[6] $(Left\,position,\; Peak\,width)$

[7] $Peak \rightarrow (Left\,position,\; Peak\,width)$

[8] *Newer associations replace older ones*

[9] *The operation is performed in the next step*

As we can easily visualize, every peak is stored into memory. Bookers will access a data structure that contains peaks (both positive and negative) ordered by position, retrieving their information accordingly to the sub-window's left end: different bookers will potentially get different widths for the same peaks. In order to fully exploit data continuity, the retrieval operation is performed only once per booking (multiple bookers participate to the same visit).

There are still a few questions the illustration could not answer and should be examined:

1. What position is assigned to a peak that maintained a constant value once or more?

   Since the dataset signal has been oversampled at the beginning of the execution, two different values never occur consecutively.

   Replacing scipy.signal's find_peaks() and peak_widths() with our own real-time functionalities means that we must emulate the original functions in the most precise way. Peaks that maintained a constant value for several instants are, for this reason, assigned the average instant as their position (rounded to the lowest integer).

2. What if two or more peaks have the same value?

What is understandable from the picture provided by SciPy (the one at the beginning of this chapter) is that higher peaks win over lower ones regarding the assignment of the underlying width (i.e. when higher peaks are found, some of the labels present in the stack structure must be changed). What if we stumble upon a peak that has the exact same value of the previous peak? The answer is very simple: instead of changing some of the labels, we place newest label by their side. This allows the algorithm to assign widths to multiple peaks.

3. Is there the need to always push information into the stack if incoming data is constant?

   No, only the information related to the first and last values of a series of constant values is pushed into the stack.

At this point, the only thing left to do is testing our algorithm.
The reader is invited to consult the testing related section.

4.2.7 Code Optimization: Multi-Threading (CPDA v3)

The reasons the whole project was given life in a Python environment are countless: “great tools and ease of use” is the proper description to sum up the advantages that lead us to opt for such an approach. Now that we have adapted the code to our needs getting rid of the Python-specific libraries, it is the moment to question the idea of maintaining the same approach.

Why would we want to change it in the first place, though? In order for Python to be as simple to use as it is, lots of things are handled through an intricate layer of control mechanisms that lower performances significantly. Not only does this complex layer flatten the optimization potential, but it saturates memory much quicker than the average programming language does. C++, on the other hand, offers a good level of abstraction without compromising performance or memory usage, still allowing low level programming techniques to be implemented as needed.

A fully optimized C++ version of the algorithm (v2.2) runs in about 25% less time with the same conditions applied to it as the ones the Python version ran in:

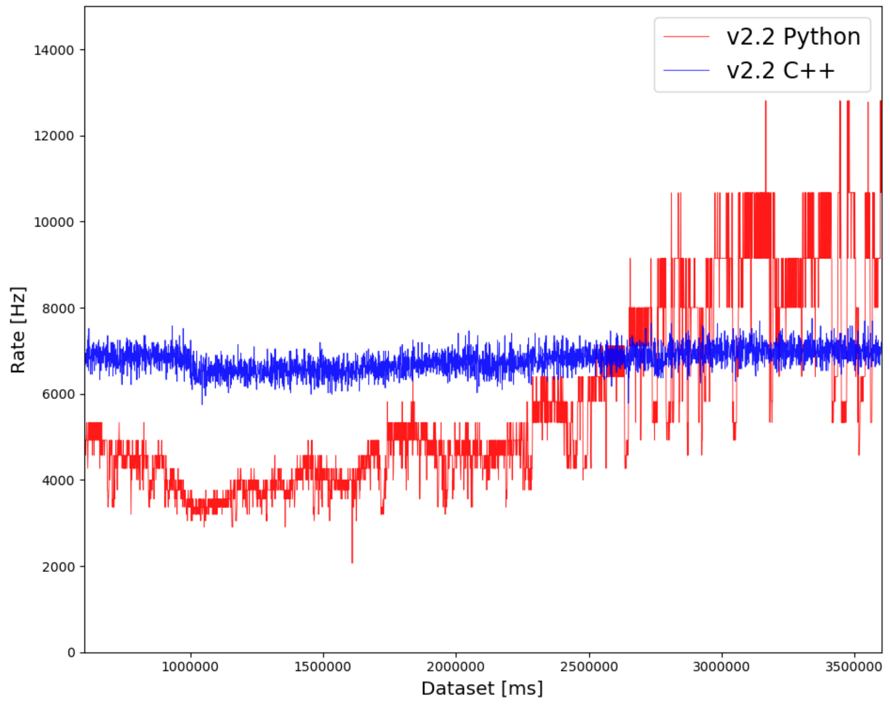


Figure 4.41: Comparison between v2.2 C++ and v2.2 Python

Interestingly enough, v2.2 C++ shows a much higher level of stability in terms of execution speed, in addition to an overall greater performance: although the Python version tends to stand out in some instances, we obviously prefer a more stable algorithm that runs a little slower in some circumstances. On top of this, the tools C++ provides us with allow us to easily create a complex and synchronized multi-threading environment that makes the potential improvement very appealing and rewarding.

Multi-Threading Implementation

The algorithm must dispose of incoming data before new data arrives. The only way to guarantee this is to handle an incoming new value in less than one millisecond, since the frequency of incoming data is $1\,kHz$. Although the algorithm already runs faster than needed for the one-hour dataset we analyzed, additional improvements can be done to better handle potentially complex situations in which what we have done may not be enough. In order to plan our next steps, we need to visualize the problem: the following illustration examines, in details, all the partial execution times of v2.2 C++. This serves as a tool to properly decide the direction the multi-threading optimization should be addressed in:

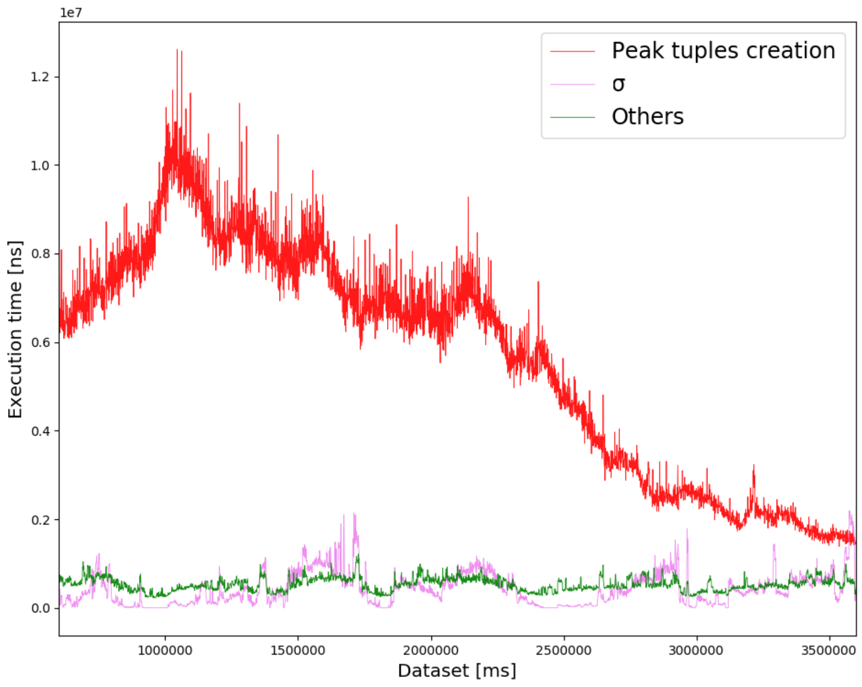


Figure 4.42: Partial execution times in v2.2 C++

What stands out the most from this chart is the peak tuples creation.

What does this mean?

Peaks are computed exploiting continuity (as explained in the "Avoiding repetition of expensive actions in $\phi(\omega)$ – Signal Analysis (CPDA v2.2)" section), but there is a moment when we have to retrieve the specific bookers' peaks in a tuple format (see the "Pattern Discovery" section) in order to study the current patterns. We should consider the fact that, at this point, the information retrieval does not have to be sequential: with the access to the shared peaks data structure being read-only, every booker could retrieve their own information at the same time without the risk of getting in the way of other bookers. Unfortunately, despite partitioning the above jobs across bookers being helpful, we cannot achieve our goal by the mean of this only: allocating a new stack for each booker once every millisecond is very expensive, without forgetting the fact that spreading the execution across different physical CPUs implies the unavoidable repetition of slower memory accesses, caused by CPU caches being empty (although this is not as true in a machine with properly cached DDRAM).

What if we juxtaposed this internal multi-threading technique with a higher-level, external parallelism? Every logical CPU executes the whole algorithm on its own, taking care of a subset of bookers that have been equally distributed across threads; meanwhile, the main thread orchestrates the process coordinating the execution and putting results together.

The question is, at this point: if each one of the external threads is potentially capable, in one millisecond, of allocating as many internal threads as two times the number of bookers present in that slot (for the retrieval of both positive and negative peaks), how many external threads can we use without overloading the machine? The answer depends indeed on the architecture of the physical chip we are running the algorithm on, but it is very hard to predict the behavior of the system with accuracy unless tested.

A high efficiency Intel Core i7-9750H with six hyper-threaded physical cores seems to be able to handle the full load of twelve external threads pretty well (as many as the number of available logical CPUs). There are no guarantees about the possibility of executing the algorithm on a lower tier CPU, as workflows of such complexity tend to bring CPU usage to the limit:

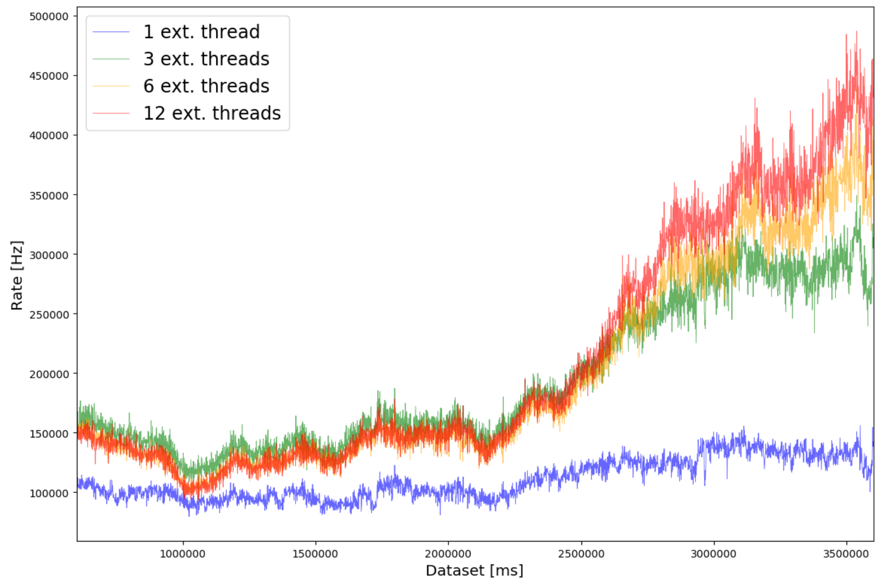


Figure 4.43: External multi-threading performances

Four versions of the algorithm were tested, each of which having both internal and external multi-threading options enabled. As we can tell from the combination of the picture above and the one at the beginning of this section, the performance enhancements introduced by the external multi-threading approach are more considerable with the peak tuples creation complexity being lower. As for the internal multi-threading approach alone, let us consider the blue line in the above chart (the one corresponding to the execution of the algorithm with one external thread only, setup equivalent to the external multi-threading option being disabled): although the rate is rather constant all throughout the execution, we can notice how the improvement introduced by the internal multi-threading functionality results in a tenfold decrease in execution times (in comparison with v2.2 C++).

After discarding the one-threaded setup, which of the three remaining configurations should we choose for future tests? The answer is not obvious, because setups with more external threads result in higher performances at higher rates, while setups with less external threads result in higher performances at lower rates. How can we decide whether to privilege some rate intervals over others and vice versa?

In order to make the right decision, we will consider the fact that results show minimum running rates being more than one hundred times higher than required, together with the fact that the average running rates are higher for configurations with the number of external threads being higher: the solution we will choose is a 12-threaded setup (red line in the chart above).

The reader is invited to consult the testing related section for more details about the execution of the 12-threaded version.

4.2.8 General Improvements (CPDA v4)

Redundancy Management

We have introduced redundancy before, noticing how results amongst adjacent sub-windows tend to repeat themselves more with ω being higher, because η is constant. On the other hand, we noticed how redundancy is rather constant for results provided by a single sub-window, because ϵ grows alongside ω. We could try controlling redundancy with an attempt to determine the optimal values for η and ϵ, but it would not be a wise approach as such values cannot be determined scientifically.

Instead, we should keep those values as low as possible (i.e. we should not modify them to handle redundancy) in order to provide the best precision, while preventing redundancy on another level: if we keep global track of the associations between patterns, we should be able to prevent bookers from adding the same association to the final result twice. In other words, when the similarity of two consequent patterns has been stated, the result is not added if it is already present.

A quick way to identify an association is to retrieve the global timestamps (not the sub-window local ones) of the two patterns' widest peaks: if the same association is found multiple times, the resulting pair of timestamps is the same every time.

Since the data structure containing the associations is shared amongst threads, the algorithm loses its deterministic property: if one thread stores an association, every other thread will not be able to do the same, so results may differ in form (not in overall content) from one execution to another.

Memory Management

Since the XAUUSD asset is not always open (like every other existing asset), data structures must be reset several times. Nevertheless, it is not a good idea to leave unused data in memory, because there is a high probability that a continuous 23-hour execution (the ordinary number of hours the XAUUSD market is open on a business day) brings memory usage to its limits and beyond. There are three data structures that require more attention than the others, as long as memory usage is concerned:

- The data that must be analyzed;
- The booking data structure;
- The peaks containers.

It is worth noticing that the peaks containers are local to every thread, so the amount of memory that a single-threaded application would require must be multiplied by the number of operating threads. Doing the math would immediately make us comprehend that unused data must be disposed of, under penalty of dozens of gigabytes of memory usage. If nothing can be done easily, in a test environment, for the data that we want to analyze – as it is read from a file all in once –, some actions can be carried out for the booking data structure (partially) and for the peaks containers: from this version on, every booker will remove their reservation once all the tasks in it have been completed, leaving the booking slot empty (the booking slots do not cease to exist); peaks containers, on the other hand, are easy to handle because they are ordered by position, and, consequently, they just need to be erased when it is sure that no booker will need them ever again.

These partially solved problems open a new set of modifications to the code structure that are to be taken proper care of as soon as a real-time implementation of the algorithm is developed.

4.2.9 Pattern Discovery: Tuning & Score System (CPDA v5)

The modifications of the pattern recognition procedure towards an efficient form of computational intelligence consist of a set of fine adjustments. Rather than limiting ourselves to simple, as well as very helpful and sometimes necessary, changes to some of the threshold values and whatnot, we must take a small step back and do some meticulous thinking.

We will analyze the new approach on a higher level in this section, redirecting the reader to the "In-deep Flow-Chart - CPDA v5" section for implementation details.

So far, patterns have been compared to each other with a 4-step binary approach: if the comparison between two patterns passed all of the four steps, it was considered successful and the couple of compared signals was stored into the results data structure. No additional information about the degree of similarity was given at the end of comparison processes, conferring no different levels of reliability to such assumptions.

The new version of the algorithm aims to provide such information, considering not only resemblance in shape, but also in height.

A logarithmic function was used to measure resemblance values, both for shape and height:

$$f(x) = log_{10}(1 + x), \quad x \in [0, 9] \tag{4.24}$$

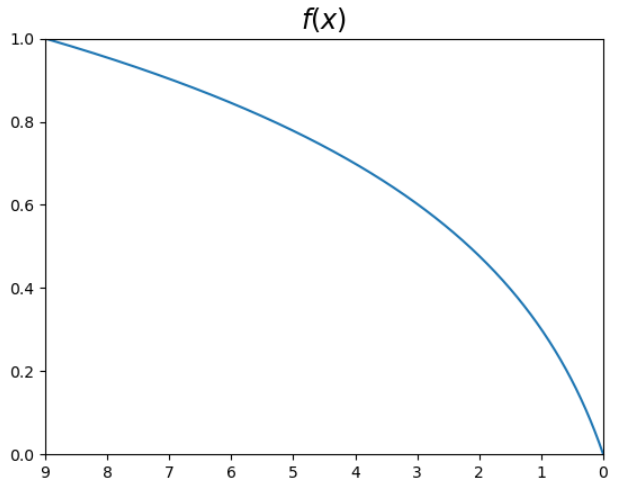


Figure 4.44: Resemblance values function

As we can see from the picture above, lower values of resemblance x result in lower values of confidence $f(x)$, decreasing faster as we approach $x = 0$. A resemblance value $x = 9$ when comparing shape means that two patterns have peaks located in the same coordinates, $x = 0$ that the last two steps of the 4-step comparison between two patterns were passed by satisfying the minimum requirements. A resemblance value $x = 9$ when comparing height means that two patterns share the same height, $x = 0$ that the computed height ratio of two patterns satisfied the minimum requirements.

This is very good, but the shape and height confidence evaluations are not enough to provide a meaningful overall confidence value; we would like another measure to contribute to it: the temporal distance between patterns.

An exponential function was used to measure distance values, as it is reasonable to say that the repetition of a specific pattern loses relevance exponentially in proportion to an increasing delay:

$$g(x) = e^{-2x}, \quad x \in [0,1] \tag{4.25}$$

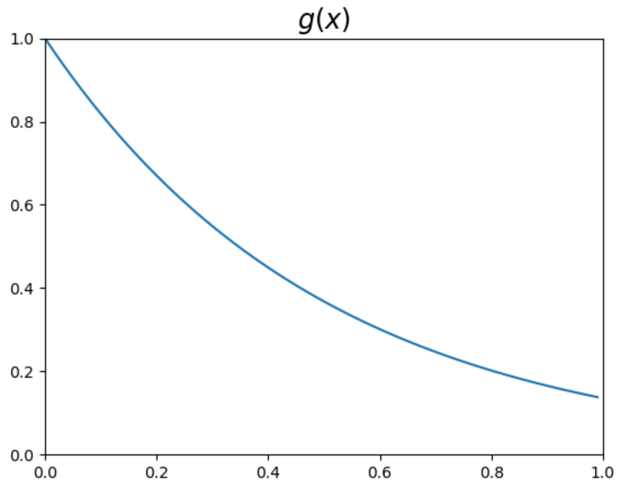
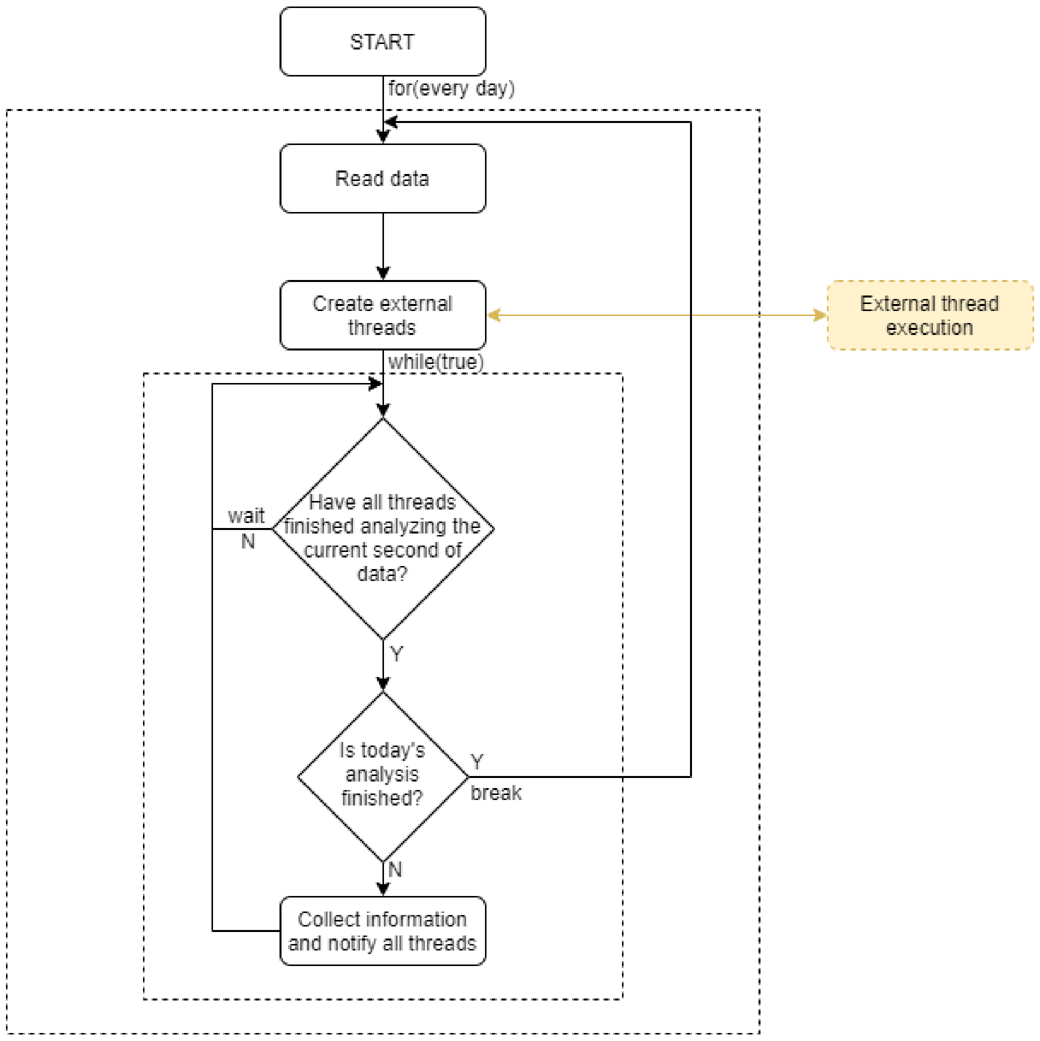


Figure 4.45: Distance values function

A distance value $x = 0$ means that two patterns are chronologically juxtaposed, $x = 1$ that one pattern started t time units after the other ended, with t being an amount of time equal to the duration of the patterns that are being evaluated. It is worth noticing that the minimum value of $g(x)$ is $e^{-2} \neq 0$: if we defined the overall confidence as the arithmetic average of the three contributions (shape, height and distance) and marked as valid patterns the ones that have a minimum confidence value $minconf = \frac{2}{3}$, we could be sure that associations of very similar patterns would not be discarded, regardless of their distance.

4.2.10 In-deep Flow-Chart - CPDA v5

Main Thread



START
for(every day)
Read data
Create external threads
External thread execution
while(true)
wait
N
Have all threads finished analyzing the current second of data?
Y
Is today's analysis finished?
Y
break
N
Collect information and notify all threads

Figure 4.46: Main Thread Flow-Chart

External Thread

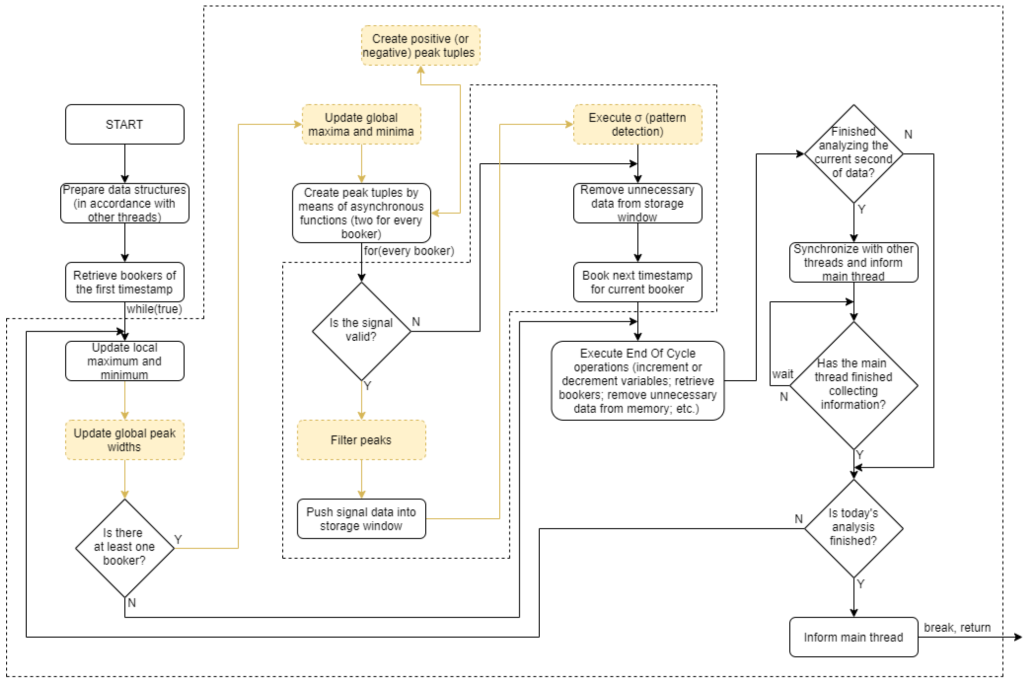
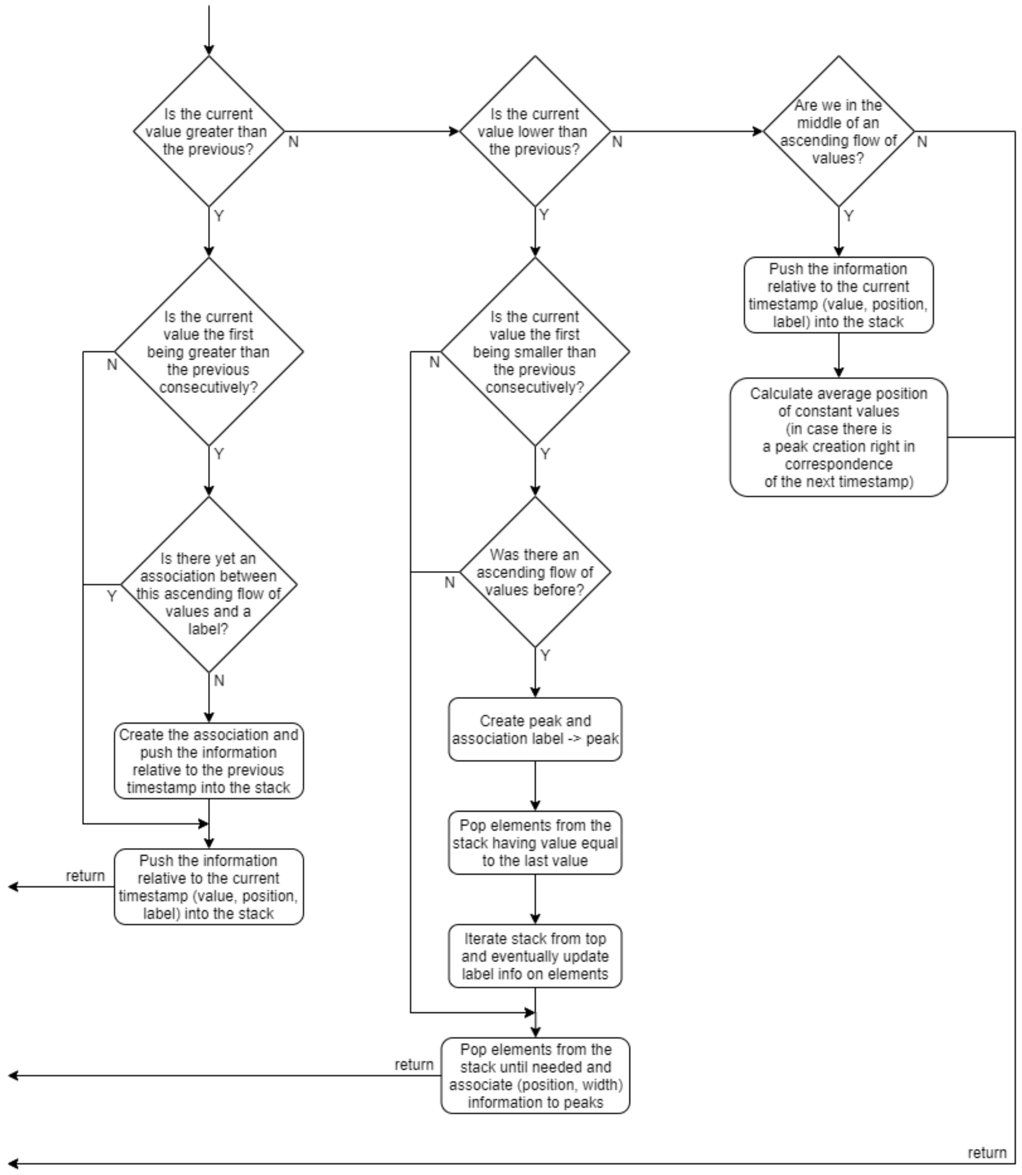

START
Prepare data structures (in accordance with other threads)
Retrieve bookers of the first timestamp
while(true)
Update local maximum and minimum
Update global peak widths
Is there at least one booker?
Y
N
Update global maxima and minima
Create positive (or negative) peak tuples
Create peak tuples by means of asynchronous functions (two for every booker)
for(every booker)
Is the signal valid?
N
Y
Filter peaks
Push signal data into storage window
Execute σ (pattern detection)
Remove unnecessary data from storage window
Book next timestamp for current booker
Execute End Of Cycle operations (increment or decrement variables; retrieve bookers; remove unnecessary data from memory; etc.)
Finished analyzing the current second of data?
N
Y
Synchronize with other threads and inform main thread
Has the main thread finished collecting information?
wait
N
Y
Is today's analysis finished?
N
Y
Inform main thread
break, return

Figure 4.47: External Thread Flow-Chart

Global positive peak widths update (applicable to negative peaks)



Is the current value greater than the previous?
N
Y
Is the current value lower than the previous?
N
Y
Are we in the middle of an ascending flow of values?
N
Y
Push the information relative to the current timestamp (value, position, label) into the stack
Calculate average position of constant values (in case there is a peak creation right in correspondence of the next timestamp)
Is the current value the first being greater than the previous consecutively?
N
Y
Is there yet an association between this ascending flow of values and a label?
Y
N
Create the association and push the information relative to the previous timestamp into the stack
Push the information relative to the current timestamp (value, position, label) into the stack
return
Is the current value the first being smaller than the previous consecutively?
N
Y
Was there an ascending flow of values before?
N
Y
Create peak and association label -> peak
Pop elements from the stack having value equal to the last value
Iterate stack from top and eventually update label info on elements
Pop elements from the stack until needed and associate (position, width) information to peaks
return
return

Figure 4.48: Global positive peak widths update Flow-Chart

Global maxima update (applicable to minima)

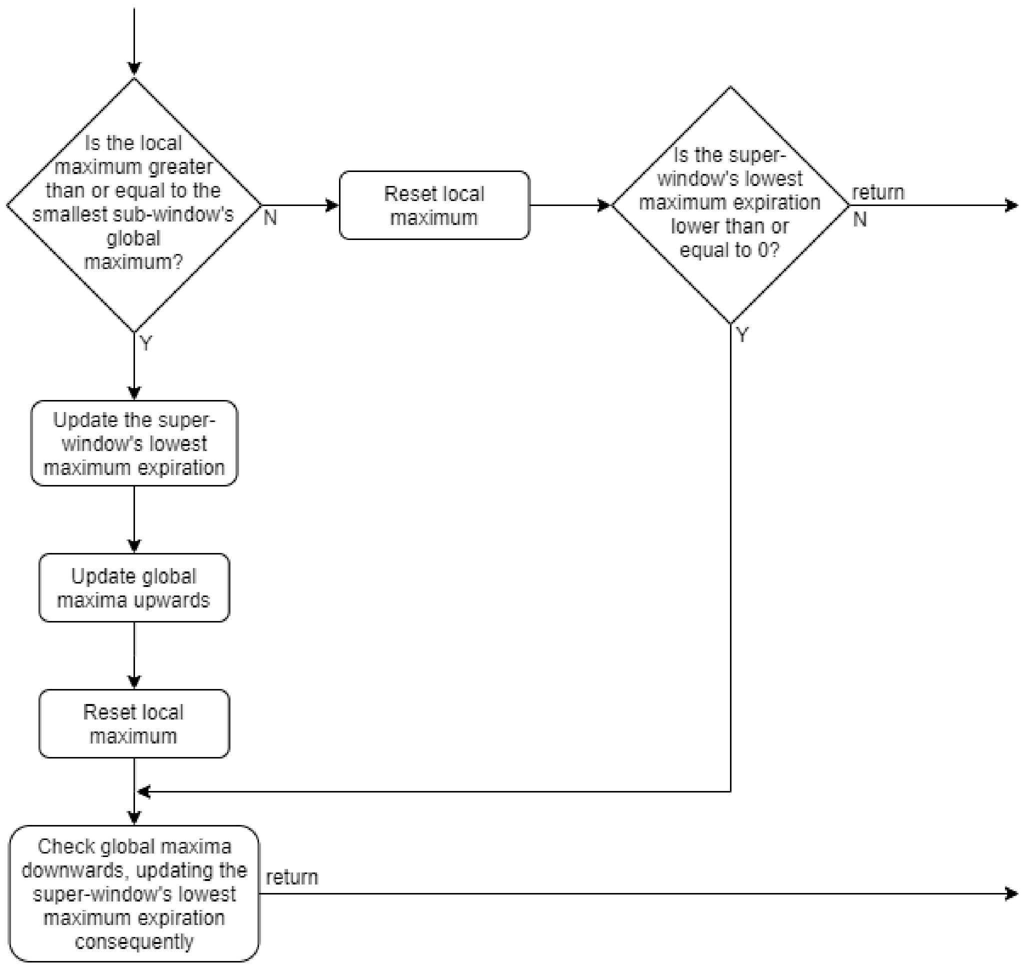
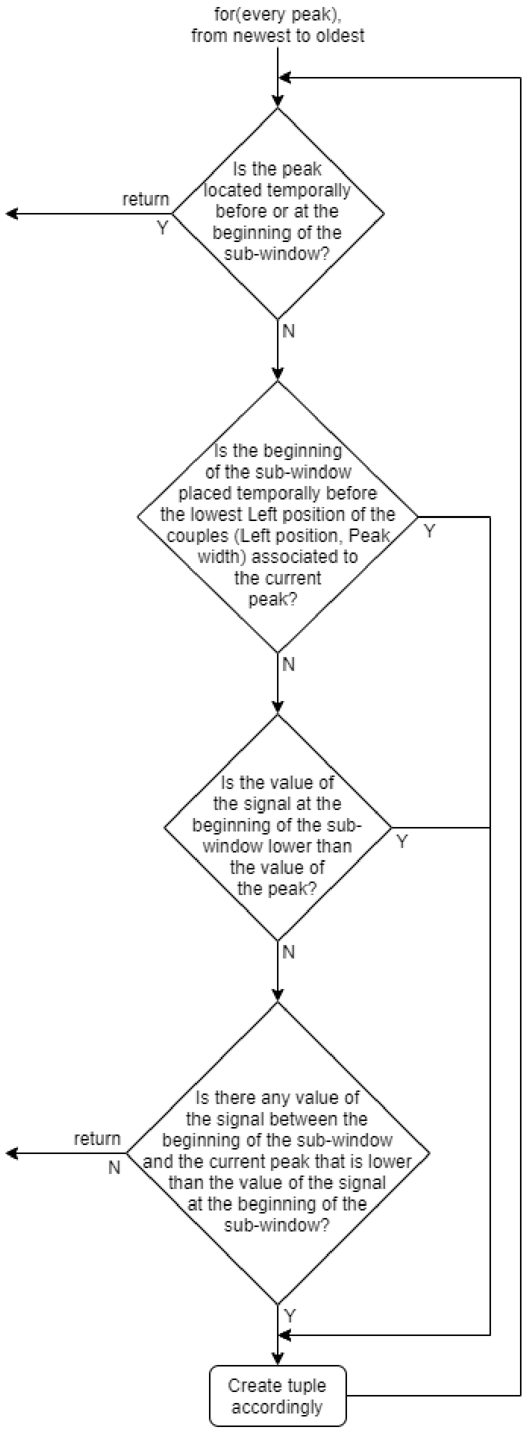

Is the local maximum greater than or equal to the smallest sub-window's global maximum?
N
Reset local maximum
Is the super-window's lowest maximum expiration lower than or equal to 0?
return
N
Y
Y
Update the super-window's lowest maximum expiration
Update global maxima upwards
Reset local maximum
Check global maxima downwards, updating the super-window's lowest maximum expiration consequently
return

Figure 4.49: Global maxima update Flow-Chart

Positive peak tuples creation (applicable to negative peaks)



for(every peak),
from newest to oldest
Is the peak located temporally before or at the beginning of the sub-window?
return
Y
N
Is the beginning of the sub-window placed temporally before the lowest Left position of the couples (Left position, Peak width) associated to the current peak?
Y
N
Is the value of the signal at the beginning of the sub-window lower than the value of the peak?
Y
N
Is there any value of the signal between the beginning of the sub-window and the current peak that is lower than the value of the signal at the beginning of the sub-window?
return
N
Y
Create tuple accordingly

Figure 4.50: Positive peak tuples creation Flow-Chart

Positive peak filtering (applicable to negative peaks)

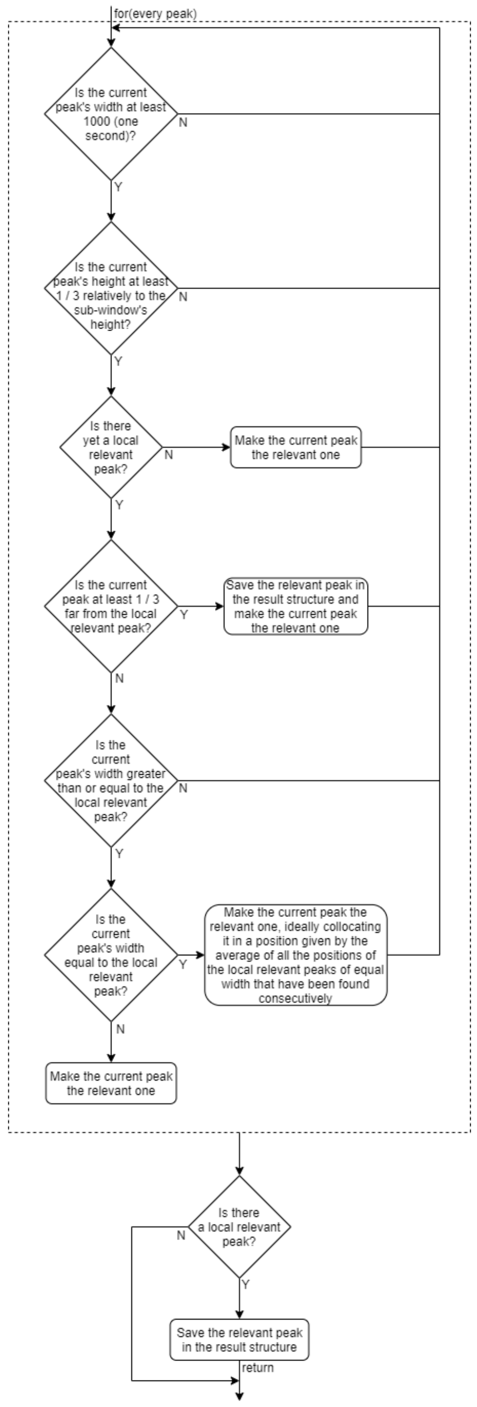

for(every peak)
Is the current peak's width at least 1000 (one second)?
N
Y
Is the current peak's height at least 1 / 3 relatively to the sub-window's height?
N
Y
Is there yet a local relevant peak?
N
Make the current peak the relevant one
Y
Is the current peak at least 1 / 3 far from the local relevant peak?
Y
Save the relevant peak in the result structure and make the current peak the relevant one
N
Is the current peak's width greater than or equal to the local relevant peak?
N
Y
Is the current peak's width equal to the local relevant peak?
Y
Make the current peak the relevant one, ideally collocating it in a position given by the average of all the positions of the local relevant peaks of equal width that have been found consecutively
N
Make the current peak the relevant one
Is there a local relevant peak?
N
Y
Save the relevant peak in the result structure
return

Figure 4.51: Positive peak filtering Flow-Chart

σ (Pattern Discovery) execution

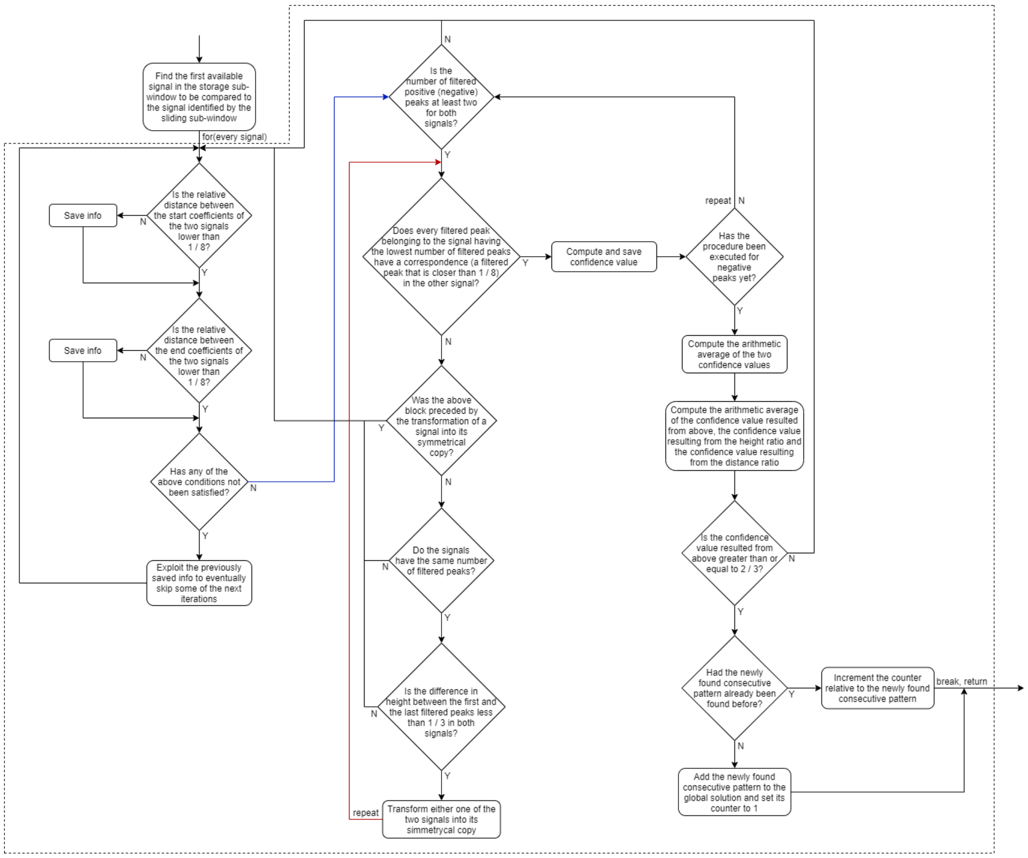

Find the first available signal in the storage sub-window to be compared to the signal identified by the sliding sub-window
for(every signal)
Is the relative distance between the start coefficients of the two signals lower than 1 / 8?
Save info
Is the relative distance between the end coefficients of the two signals lower than 1 / 8?
Save info
Has any of the above conditions not been satisfied?
Exploit the previously saved info to eventually skip some of the next iterations
Is the number of filtered positive (negative) peaks at least two for both signals?
Does every filtered peak belonging to the signal having the lowest number of filtered peaks have a correspondence (a filtered peak that is closer than 1 / 8) in the other signal?
Compute and save confidence value
Has the procedure been executed for negative peaks yet?
repeat
Was the above block preceded by the transformation of a signal into its symmetrical copy?
Do the signals have the same number of filtered peaks?
Is the difference in height between the first and the last filtered peaks less than 1 / 3 in both signals?
Transform either one of the two signals into its simmetrycal copy
repeat
Compute the arithmetic average of the two confidence values
Compute the arithmetic average of the confidence value resulted from above, the confidence value resulting from the height ratio and the confidence value resulting from the distance ratio
Is the confidence value resulted from above greater than or equal to 2 / 3?
Had the newly found consecutive pattern already been found before?
Increment the counter relative to the newly found consecutive pattern
break, return
Add the newly found consecutive pattern to the global solution and set its counter to 1
Y
N

Figure 4.52: σ execution Flow-Chart

4.3 CPDA: Performance and Congruency tests with different versions

The following tests have been performed in order to validate newer versions of the algorithm over v1.1.

A script has been used for stating congruency between versions: if the same patterns have been found by two different versions, they are congruent with each other.

Newly added entries to the test details table are highlighted in orange.

4.3.1 CPDA v2.0

Table 4.4: Test Details - CPDA v2.0

Additional Modifications (since last version)	None
Congruent with last version	Yes
Execution Start Date	Sunday, September 20th, 2020
Execution Start Time	19:20:01 GMT
Execution End Date	Monday, September 21st, 2020
Execution End Time	14:23:50 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	19 hours, 3 minutes, 49 seconds
Speedup from v1.1	N/A
Speedup from last version	N/A
Ideal Running Rate	1 000 Hz or higher
Average Running Rate	47.89 Hz
Maximum Running Rate	59.60 Hz

Minimum Running Rate	$37.72\,Hz$
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor \; [ms]$
OS	Windows 10 Home v2004, 64 bits
Threads used	1
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
Python version	3.8

Considerations

See the “First approach to a new way of thinking (CPDA v2.0)” section.

4.3.2 CPDA v2.1

Table 4.5: Test Details - CPDA v2.1

Additional Modifications (since last version)	None
Congruent with last version	Yes
Execution Start Date	Monday, September 21st, 2020
Execution Start Time	14:50:01 GMT
Execution End Date	Tuesday, September 22nd, 2020
Execution End Time	07:50:49 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	17 hours, 48 seconds
Speedup from v1.1	1.065x
Speedup from last version	1.121x
Ideal Running Rate	$1\,000\,Hz$ or higher
Average Running Rate	$53.70\,Hz$
Maximum Running Rate	$66.89\,Hz$
Minimum Running Rate	$41.62\,Hz$
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor \; [ms]$

OS	Windows 10 Home v2004, 64 bits
Threads used	1
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
Python version	3.8

Considerations

As new versions are being developed, several new measures must be introduced. The total execution time does not provide us with enough information anymore: we need to know the speed of the algorithm in every part of the execution if we want to develop a reliable real-time application. How many milliseconds of data can the algorithm deal with in a second? Are there portions of data that require more computation? The average, maximum and minimum running rates provide information about the relative speeds of the algorithm, monitoring its performance in a stabilized environment (when the number of bookers is constant).

Not only does the running rate tell us whether the system works homogeneously, but it also provides a tool that enables us to comprehend how different versions work in relation to each other: a version that is generally faster than another is not automatically proclaimed the best overall.

Let us look at the comparison between v2.1 and v2.0 in terms of relative speeds:

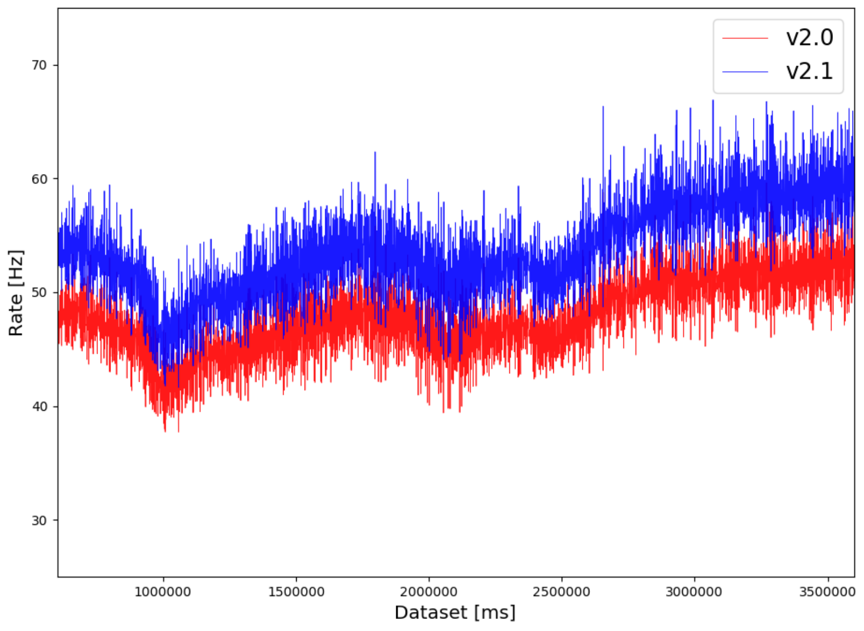


Figure 4.53: Comparison between v2.1 and v2.0 (Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT)

What matters the most is the fact that v2.1 is faster than v2.0 in every instant, with the margin being rather constant all throughout the execution. This allows us to officially validate v2.1 over the other versions, accounting it as the best version of the algorithm so far. Can we do better?

4.3.3 CPDA v2.2

Table 4.6: Test Details - CPDA v2.2

Additional Modifications (since last version)	- Unused data structure eliminated
Congruent with last version	Yes
Execution Start Date	Tuesday, September 22nd, 2020
Execution Start Time	08:15:01 GMT
Execution End Date	Tuesday, September 22nd, 2020
Execution End Time	08:25:57 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	10 minutes, 56 seconds
Speedup from v1.1	99.44x
Speedup from last version	93.37x
Ideal Running Rate	$1\,000\,Hz$ or higher
Average Running Rate	$5\,725\,Hz$
Maximum Running Rate	$12.81\,kHz$
Minimum Running Rate	$2\,065\,Hz$
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor \; [ms]$

OS	Windows 10 Home v2004, 64 bits
Threads used	1
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
Python version	3.8

Considerations

Let us look at the comparison between v2.2 and v2.1:

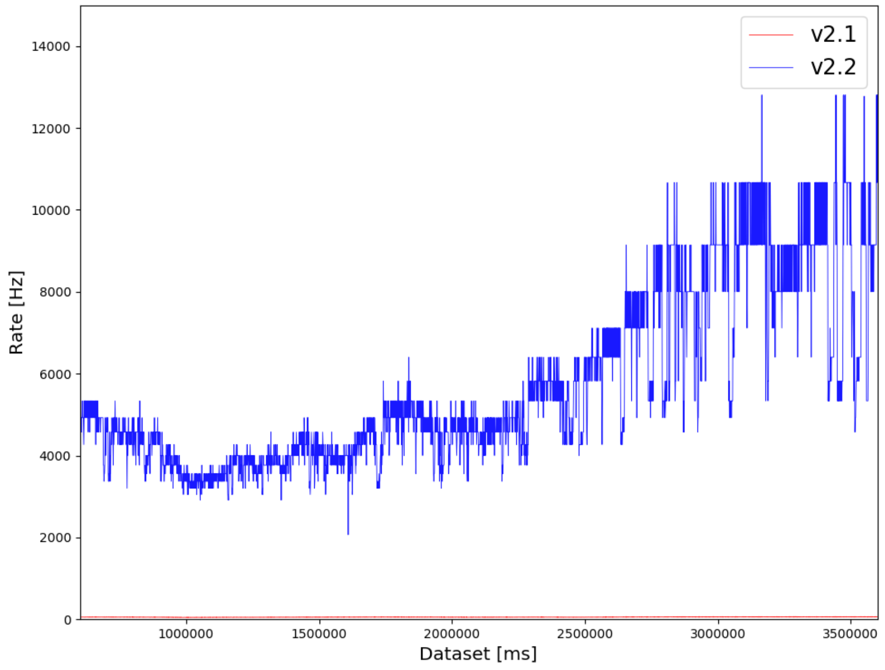


Figure 4.54: Comparison between v2.2 and v2.1 (Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT)

These results show something very interesting.

The new chart resembles the previous one in the first half, while it shows something different happening afterwards: it seems that the improvements presented with this version, while giving a great advantage in terms of performance to every single part of the execution, worked slightly better for the second half of the dataset. It is worth noticing that, to a higher rate, corresponds a higher level of instability in the rate itself.

Since v2.2 operates generally much faster than v2.1, we will validate it as the best new version of the algorithm.

We achieved one important goal: the algorithm could have run real-time without any problems. Nevertheless, we cannot be sure it would work smoothly in every situation because of the unpredictable nature of data shapes in a financial context: a real-time algorithm should prevent unpredictable events from slowing itself down.

4.3.4 CPDA v3

Table 4.7: Test Details - CPDA v3

Additional Modifications (since last version)	None
Congruent with last version	Yes
Execution Start Date	Thursday, September 24th, 2020
Execution Start Time	09:30:01 GMT
Execution End Date	Thursday, September 24th, 2020
Execution End Time	09:30:21 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	20 seconds
Speedup from v1.1	3 262x
Speedup from last version	25.00x
Ideal Running Rate	$1\,000\,Hz$ or higher
Average Running Rate	$210.6\,kHz$
Maximum Running Rate	$486.9\,kHz$
Minimum Running Rate	$93.59\,kHz$
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor \, [ms]$

OS	Windows 10 Home v2004, 64 bits
Threads used	12+
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
C++ version	C++ 11

Considerations

See the “Multi-Threading Implementation” section.

4.3.5 CPDA v4

Table 4.8: Test Details - CPDA v4

Additional Modifications (since last version)	None
Congruent with last version	Yes
Execution Start Date	Thursday, September 24th, 2020
Execution Start Time	09:35:00 GMT
Execution End Date	Thursday, September 24th, 2020
Execution End Time	09:35:18 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	18 seconds
Ideal Running Rate	$1\,000\,Hz$ or higher
Average Running Rate	$232.2\,kHz$
Maximum Running Rate	$599.9\,kHz$
Minimum Running Rate	$94.86\,kHz$
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor \; [ms]$
OS	Windows 10 Home v2004, 64 bits
Threads used	12+

CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
C++ version	C++ 11

Considerations

Most of the entries of the test details table related to the comparison with previous versions were removed, because we do not expect results to be congruent nor the execution times to be any faster than v3's. What we expect, on the other hand, is results to be less redundant and memory usage to be less significant: both characteristics have been largely satisfied by this new version of the algorithm. Results present negligible amounts of redundancy (as we can see from the image below), while memory usage during execution has been reduced by up to 50%.

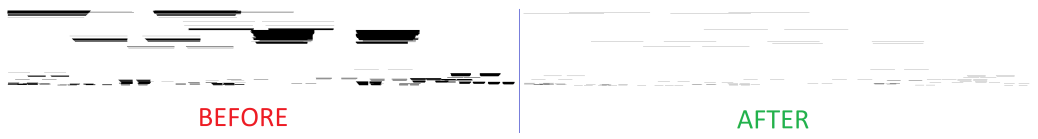


Figure 4.55: Redundancy comparison between v2.4 and v2.3

4.3.6 CPDA v5

Table 4.9: Test Details - CPDA v5

Additional Modifications (since last version)	- Code divided into multiple sequential sub-executions in order to differentiate results of different days
Congruent with last version	Yes
Execution Start Date	Thursday, September 24th, 2020
Execution Start Time	09:40:00 GMT
Execution End Date	Thursday, September 24th, 2020
Execution End Time	09:40:18 GMT
Ideal Total Execution Time	1 hour or less
Actual Total Execution Time	18 seconds
Ideal Running Rate	$1\,000\,Hz$ or higher
Average Running Rate	$247.3\,kHz$
Maximum Running Rate	$682.1\,kHz$
Minimum Running Rate	$94.25\,kHz$
Dataset	Gold Ask Price - October 1st, 2019, 19:25:00 GMT until October 1st, 2019, 20:25:00 GMT
Dataset Width n	$\sim 3\,600\,000\,ms$
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{162} \cdot \omega_i \rfloor\ [ms]$
OS	Windows 10 Home v2004, 64 bits

Threads used	12+
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
C++ version	C++ 17

Considerations

With minimum confidence set to $\frac{2}{3}$, each of the four resulting consecutive patterns was spotted within the first third of the dataset. Although the results seem to be accurate enough for us to feel satisfied, we should not delude ourselves into thinking we found the silver bullet for Pattern Discovery: there may be need for additional tuning procedures, since one hour is quite an insufficient amount of data to tune the algorithm with for us to expect satisfying results out of any kind of dataset.

The aim of the following picture is to show the first third of the dataset, delivering some context to results and giving the reader a more complete overview; the chronologically ordered results will follow right after, providing information about the exact locations of patterns together with their confidence value.

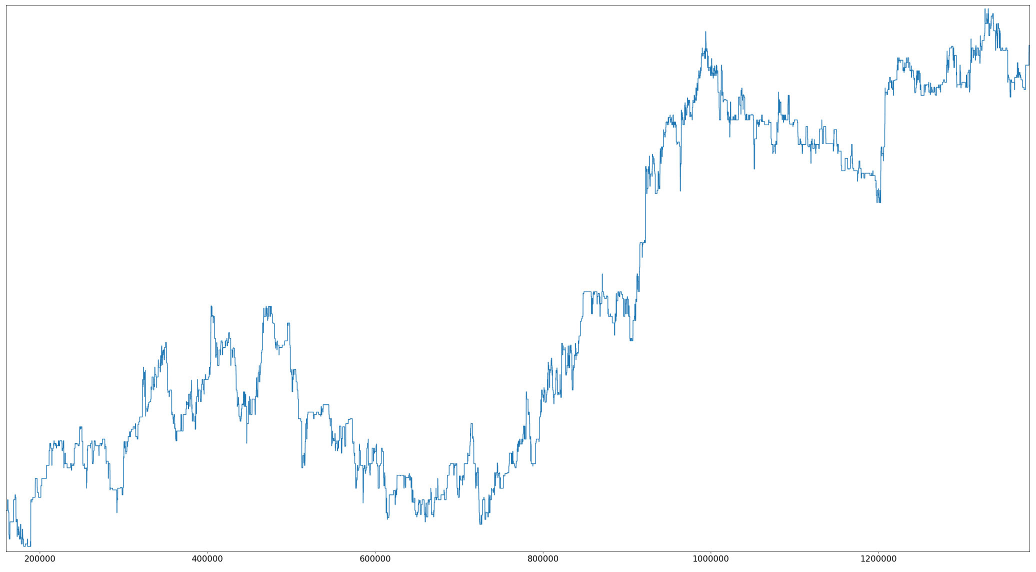


Figure 4.56: Visualization of the first third of the dataset used for this test

$\omega = 94000$

$Pattern\,1\,relative\,location:$ [162214, 256213] (left)

$Pattern\,2\,relative\,location:$ [259102, 353101] (right)

$Confidence = 0.75$

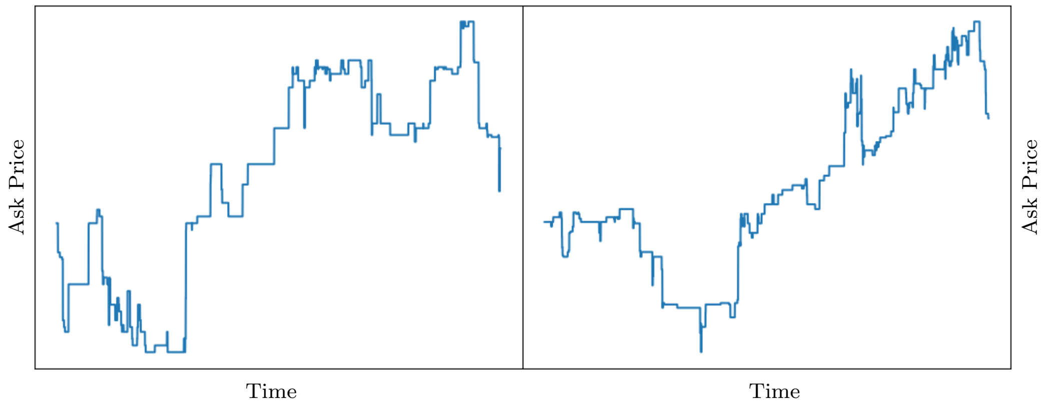


Figure 4.57: Visualization of the first result

$\omega = 60000$

$Pattern\,1\,relative\,location:$ $[311688, 371687]$ (left)

$Pattern\,2\,relative\,location:$ $[381888, 441887]$ (right)

$Confidence = 0.71$

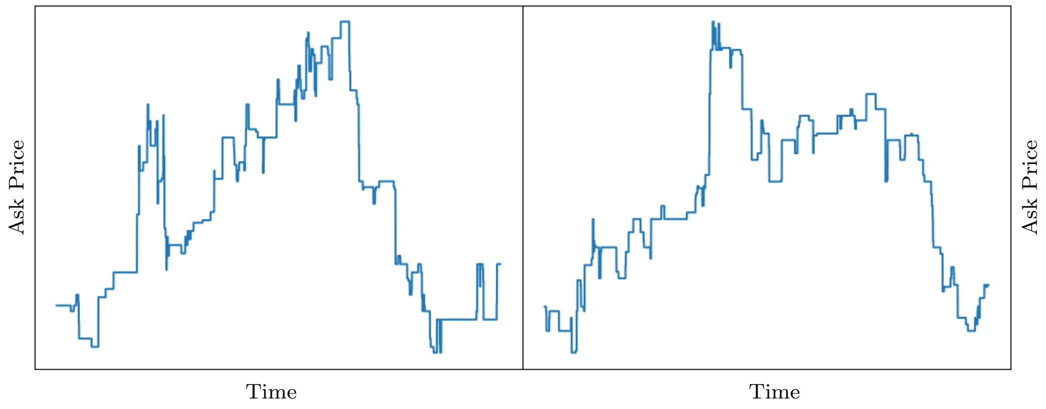


Figure 4.58: Visualization of the second result

$\omega = 60000$

$Pattern\,1\,relative\,location:$ $[374400, 434399]$ (left)

$Pattern\,2\,relative\,location:$ $[447408, 507407]$ (right)

$Confidence = 0.74$

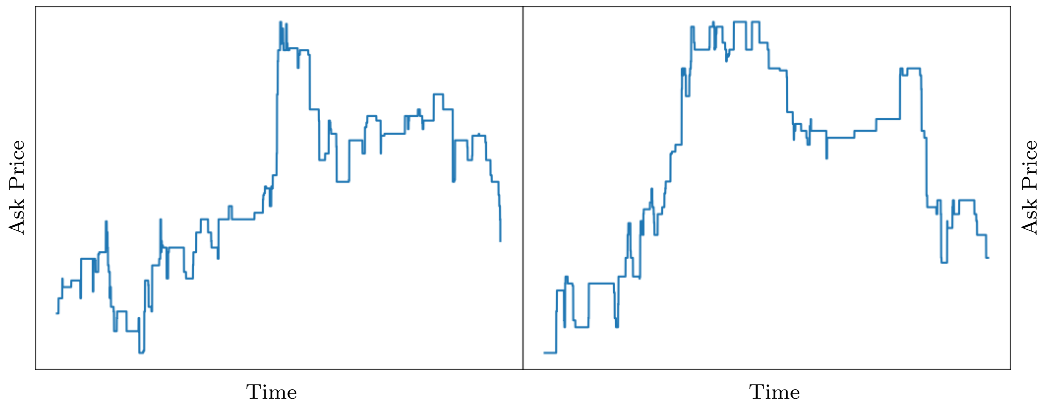


Figure 4.59: Visualization of the third result

$\omega = 87000$

$Pattern\,1\,relative\,location:$ [816837, 903836] (left)

$Pattern\,2\,relative\,location:$ [940415, 1027414] (right)

$Confidence = 0.67$

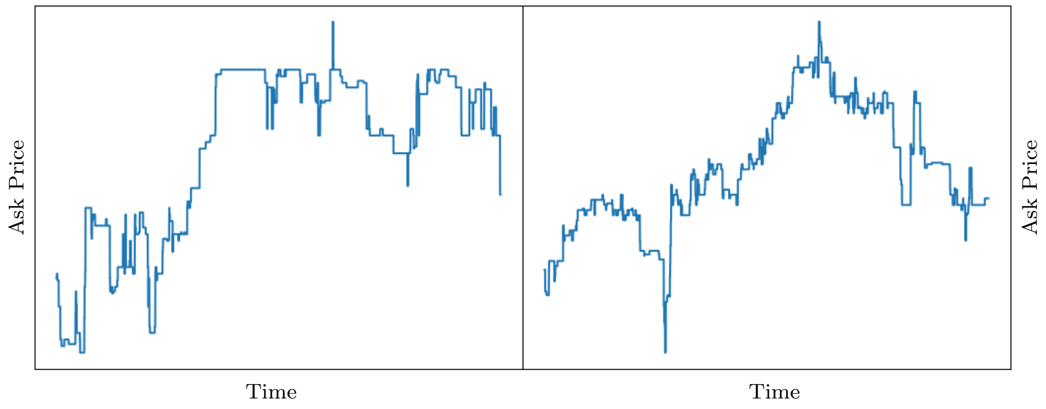


Figure 4.60: Visualization of the fourth result

4.3.7 Performance History

Below is a chart showing the execution times of the performed tests by version.

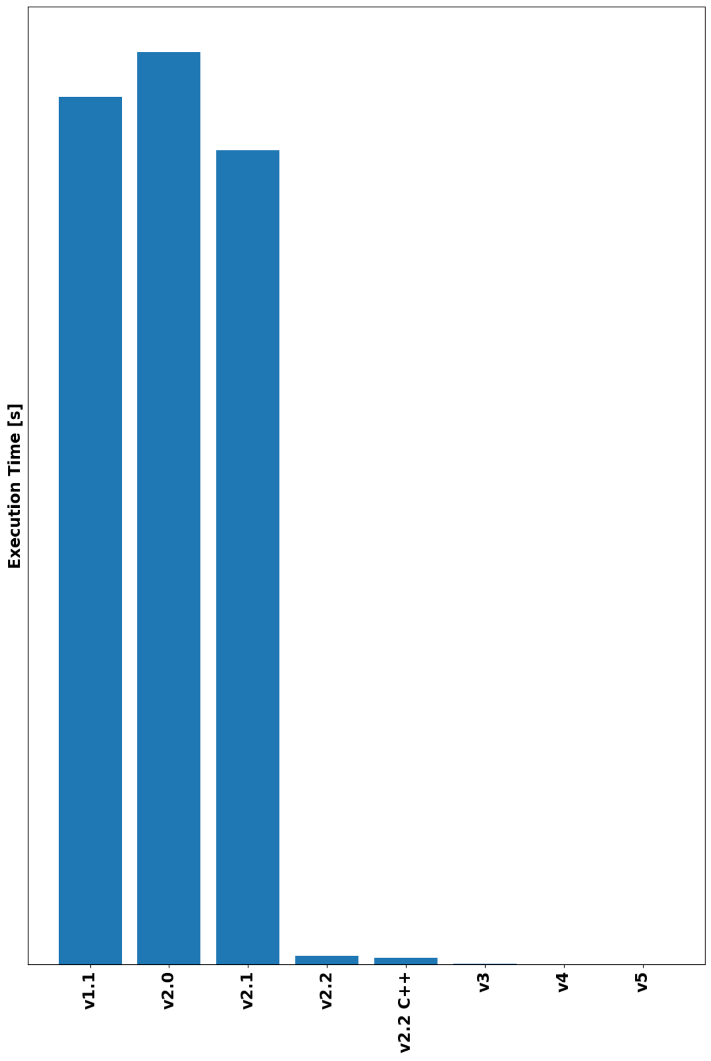


Figure 4.61: Histogram of the performance history throughout versions

4.4 CPDA: Output test with v5: all-time XAUUSD dataset

Table 4.10: Test Details - CPDA v5 (all-time XAUUSD dataset)

Additional Modifications (since last version)	- Added check on single-day signal length (must be at least twice ω_M
Execution Start Date	Thursday, October 1st, 2020
Execution Start Time	10:00:00 GMT
Execution End Date	Sunday, October 25th, 2020
Execution End Time	11:52:16 GMT
Ideal Running Rate	$1\,000\,Hz$ or higher
Average Running Rate	$327.0\,kHz$
Maximum Running Rate	$1\,232\,kHz$
Minimum Running Rate	$131.9\,Hz$
Real-time satisfaction	99.9997%
Dataset	Gold Ask Price - May 5th, 2003, 00:00:00 GMT until September 30th, 2020, 23:00:00 GMT
Minimum Window Width ω_m	$60\,000\,ms$
Maximum Window Width ω_M	$600\,000\,ms$
Number of Windows λ	541
Window Widths Difference η	$1\,000\,ms$
Error ϵ_i	$\lfloor \frac{1}{128} \cdot \omega_i \rfloor\ [ms]$
Consecutive patterns found	1 458 189
OS	Windows 10 Home v2004, 64 bits

Threads used	12+
CPU	Intel Core i7-9750H @ 2.60 GHz
RAM	16 GB DDR4 2 667 MT/s
Pagefile	32+ GB SSD
C++ version	C++ 17

Considerations

Something has changed in the test details table: the entries related to the dataset width and the total execution time have been removed, as such information becomes meaningless and hard to retrieve for bigger datasets. On the other hand, we can see newly added entries that help us look at the bigger picture, amongst which the real-time satisfaction percentage is surely important, as it describes the portion of the analysis that ran at the ideal rate or above (we will consider a 99% real-time satisfaction or above a good percentage). Another element that is worth our attention is the pagefile-related information: is not 16 gigabytes enough for the algorithm to run without incurring in memory problems? The answer is no, unfortunately. We have to remember that there are twelve threads operating on a one day period with a millisecond precision! This is not a big deal, though: old data that we cannot delete yet is safely transferred from RAM to disk and is deleted as soon as the algorithm finishes analyzing the current day.

What about the results? Are they satisfying enough? Was everything we worked on worth the effort? We cannot properly answer this question investigating a few random results: although verifying the validity of more than one million consecutive patterns is still close to impossible, we should try to have all the results available for a fast visual check. A script was used to generate as many images as the number of consecutive patterns that were found during the test execution, grouped by day (5,423 folders – the number of days of data the dataset was composed of). Every image displays the couple of patterns that resulted similar, together with information regarding the window width and the confidence of the result. The output of the script empirically confirms the overall validity of the algorithm.

Nevertheless, important differences in results have been spotted across years of data: early data presents lower frequency tick information, often resulting in less meaningful outcomes. The cause of some results not being very significant is to be addressed to the inability of the algorithm to approximate some patterns to three relative maxima and three relative minima (or less) as we previously described in the "Peak Filtering" section. This brings us to the conclusion (as one could have easily imagined) that there cannot be a silver bullet for such a diverse and complex problem: the spectrum of data properties the algorithm could find itself dealing with is as unpredictable as the data itself. This should not surprise us, nor should it make us lose heart. Once we are ready to acknowledge there exists no magic wand, we can move forward in our research and find a way to adapt the algorithm to any possible shape the input signal might have. The steps we will take include a generalization of the Consecutive Pattern Discovery Algorithm so that it adapts to the Pattern Discovery state-of-the-art algorithms. These actions are required to add a scientific meaning to this research, which so far has been backed by empirical results only: the reader is invited to refer to the "Expanding our horizons: General Purpose Pattern Discovery Algorithm" section.

The images below show the nine-squared approximation fail on May 5th, 2003:

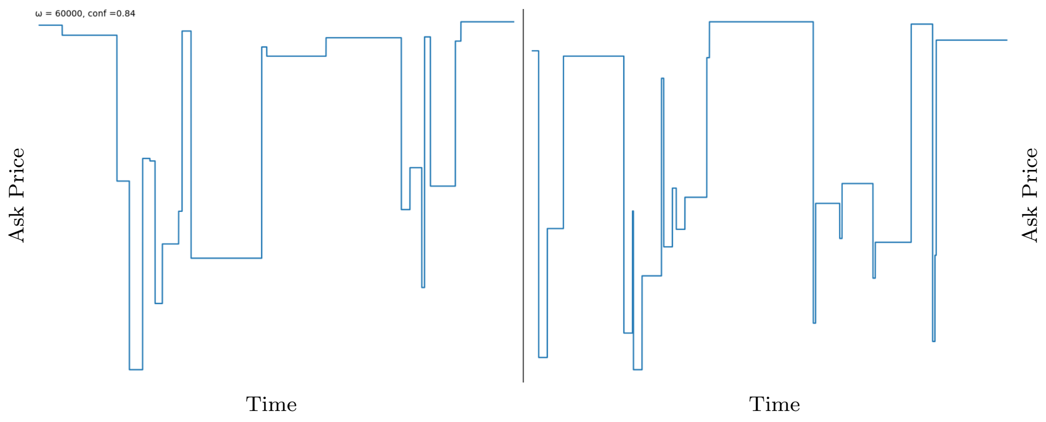


Figure 4.62: Nine-squared approximation failure

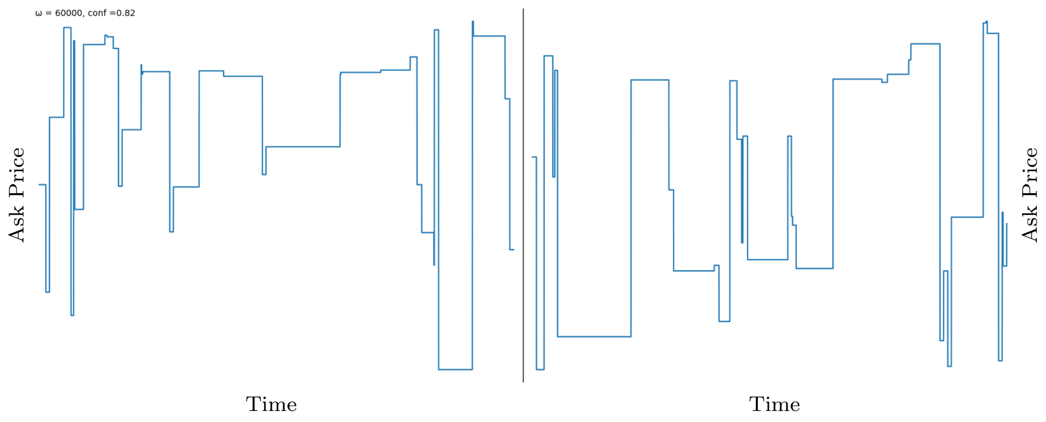


Figure 4.63: Nine-squared approximation failure (2)

5. Application of Consecutive Patterns to address Anomaly Detection

5.1 Expanding our horizons: General Purpose Pattern Discovery Algorithm

5.1.1 GPPDA: Features

What if one's intent was to approximate sub-signals into four, five or ten meaningful points? Not only do different needs shape the degree of approximation, but they can change the whole algorithm's approach as far as interpretation of results is concerned. There can even be situations in which considering approximation as an option is conceptually wrong in the first place.

GPPDA differentiates between eighteen possible configurations (that become countless if we take numeric parameters into account), allowing CPDA v5 to become a generalized, complete and efficient Pattern Discovery Algorithm, as well as presenting itself as the first choice algorithm for Pattern Discovery in highly repetitive signals.

Before diving straight into the operational possibilities GPPDA has to offer, there are some newly introduced features that are worthy of a detailed explanation. Code has been reordered in such a way that execution times are almost halved for approximate searches, as well as reorganized into sections that are executed only if the related configuration entry is enabled. The compiled GPPDA executable is plug and play, meaning it works on every Windows machine provided that a valid JSON configuration file exists in the same path the executable is stored.

Amongst the new available features, the possibility for the user to perform an exact search is the one that deserves most attention: if the entry "approximationEnabled" is set to false in the configuration file, peak-related information is not stored, nor are peak widths computed. In return, a global lossless summary of the signal is constantly updated, providing a quick way to retrieve detailed information about specific portions of the signal. The summary consists of a set of chronologically ordered entries, each describing a specific point in the signal and the quantitative extension of its repetition:

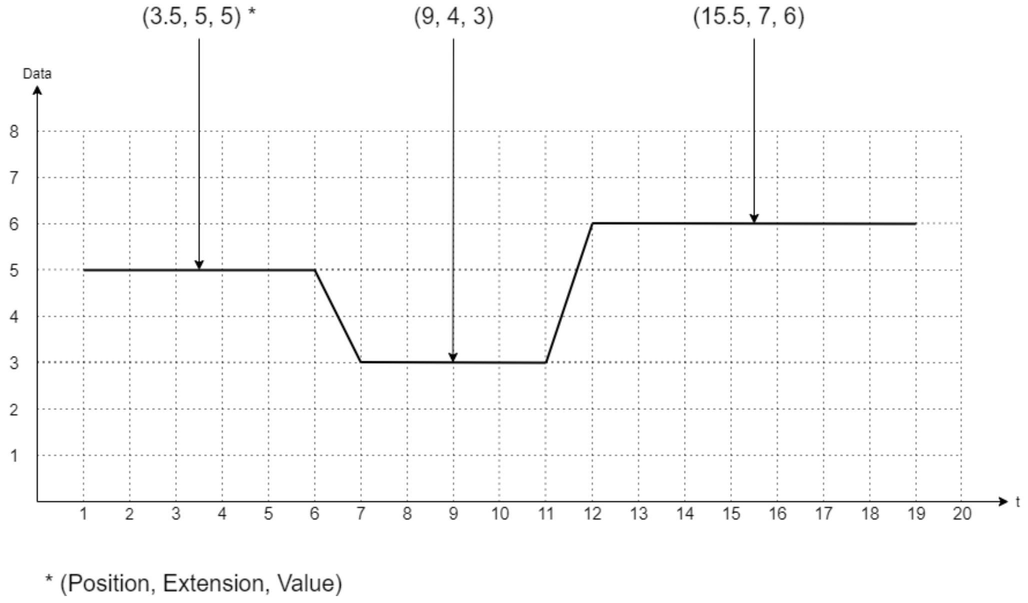


Figure 5.1: Visualization of GPPDA's summary structure

If we wanted to brutally adapt this solution to the existing code, we should save a sub-set of the global signal summary in correspondence of every booked timestamp, for each booker onto its storage sub-window. This kind of approach was convenient when we had to deal with peaks, but it would now result in high redundancy and inefficiency as no filtering operation must be performed. Comparisons are directly carried out reading the global (local to every thread) data structure, using two iterators (one for each sub-window) and computing incremental Euclidean distances. It is clear that computational times decrease with input signals being more repetitive.

It is now the moment to list all the possible GPPDA configurations and to provide some usage references. The following eighteen points describe every possible set of configurations that makes GPPDA unique in its versatility:

1. Approximation enabled, greedy approach in σ enabled, height differences in patterns affects confidence value, distance between patterns affects confidence value, partial search enabled (the configuration we have used so far);

2. Approximation enabled, greedy approach in σ enabled, height differences in patterns affects confidence value, distance between patterns does not affect confidence value, partial search enabled;

3. Approximation enabled, greedy approach in σ enabled, height differences in patterns does not affect confidence value, distance between patterns affects confidence value, partial search enabled;

4. Approximation enabled, greedy approach in σ enabled, height differences in patterns does not affect confidence value, distance between patterns does not affect confidence value, partial search enabled;

5. Approximation enabled, greedy approach in σ disabled, height differences in patterns affects confidence value, distance between patterns affects confidence value, partial search enabled;

6. Approximation enabled, greedy approach in σ disabled, height differences in patterns affects confidence value, distance between patterns does not affect confidence value, partial search enabled;

7. Approximation enabled, greedy approach in σ disabled, height differences in patterns does not affect confidence value, distance between patterns affects confidence value, partial search enabled;

8. Approximation enabled, greedy approach in σ disabled, height differences in patterns does not affect confidence value, distance between patterns does not affect confidence value, partial search enabled;

9. Approximation disabled (greedy approach in σ disabled as a consequence), partial search enabled;

$10 - 18$. The same as points 1-9, with partial search disabled.

Points 9 and 18 show us how switching from the approximate approach to the exact one prevents us from using a greedy approach in detecting similarities between patterns: this choice resides in the definition of "exact", which is intrinsically in contrast with the definition of "greedy".

Since disabling the greedy approach, as we will soon ascertain, results in the use of the K-NN paradigm, redundancy management is disabled in non-greedy configurations not to interfere with the cardinality of the results provided by the K-NN model.

What has not been mentioned above is the fact that enabling some features implies the definition of other parameters. Let us look at an example for configuration 1:

```
{
”year”: ”2020”,
”month”: ”01”,
”day”: ”01”,
”hours”: ”00”,
”minutes”: ”00”,
”seconds”: ”00”,

”minSubwindow”: ”60000”,
”maxSubwindow”: ”600000”,

”approximationEnabled”: true,
”partialSearch”: true,
”dukascopyStockInputData”: true,

”ifApproximationEnabled”: {
”differenceBetweenSubwindows”: ”1000”,
”beta”: ”128”,
”startEndDislocationLimit”: ”1 / 3”,
”maxPeaksPerPattern”: ”3”,
”minPeakWidth”: ”1000”,
”pointsDislocationLimit”: ”1 / 8”,
”minConf”: ”2 / 3”,
"heightAffectsConf": true,
"distanceAffectsConf": true,
"greedySigmaEnabled": true
},

"ifNotGreedySigmaEnabled": {
"knn": "1"
},

"ifPartialSearch": {
"storageWindow": "2"
},

"dataPath": "D:/Branch Prediction Based Automatic Trading System/-
DataTest/Other/",
"logsPath": "D:/Branch Prediction Based Automatic Trading Sys-
tem/TestResults/GPPDA/Logs/",
"ratesPath": "D:/Branch Prediction Based Automatic Trading Sys-
tem/TestResults/GPPDA/Rates/"
}
```

We can see how enabling approximation practically means deciding the numeric value of several parameters:

- "differenceBetweenSubwindows": What we have been referring to as η;
- "beta": What we have been referring to as β;
- "startEndDislocationLimit": The threshold the absolute difference be-

tween a pattern's start and end coefficients must fall below in order for a pattern to be saved onto the storage sub-window;

- "maxPeaksPerPattern": The number of peaks (valid both for positive and negative peaks) the algorithm will try to approximate patterns into;
- "minPeakWidth": A static value below which the algorithm will always ignore peaks during filtering (peaks having widths below this value will never be considered relevant peaks);
- "pointsDislocationLimit": The threshold each of the absolute differences between two patterns' relevant peaks must fall below in order for them to trigger a potential similarity;
- "minConf": A static value below which the algorithm will always ignore potential similarities.

Disabling approximation, on the other hand, implicitly fixes "differenceBetweenSubwindows" to 1. The usage of the algorithm in its exact variant also implies the introduction of the "knn" parameter, which tells the number of nearest neighbors we want to be returned from a single storage sub-window backwards iteration. It is worth noting that the "knn" parameter becomes meaningful for approximate searches that do not apply a greedy approach in σ too: the implicit difference between the exact "knn" and the approximate "knn" resides in the number of the nearest neighbors returned, which is always the same in the first case while there are no guarantees for approximate searches. In both cases, if "knn" is set to 0 all neighbors are returned.

What is surely independent from the above decisions is the "storageWindow" parameter, which is obviously meaningful in partial searches only. This variable is expressed in number of sub-windows and decides how far back in the dataset we want patterns to be compared.

There are a few parameters we have left behind. The following descriptions close the circle giving the necessary information for GPPDA to be used properly:

- "year", "month", "day", "hours", "minutes", "seconds": The set of parameters that schedule date and time of execution;
- "minSubwindow": What we have been referring to as ω_m;
- "maxSubwindow": What we have been referring to as ω_M;
- "dukascopyStockInputData": Decides how input data is retrieved (if this option is enabled data is read following Dukascopy's format and is oversampled afterwards, otherwise data is read as it is without further actions);
- "dataPath": The folder where data files are read sequentially;
- "logsPath": The folder where results are written for each data file analyzed;
- "ratesPath": The folder where execution times are written for each data file analyzed.

When the configuration file is read a set of very strict validation checks are performed: if something unexpected is read an exception is thrown and an error message is displayed. The following constraints must be respected in order for the algorithm to start:

- "year": Must be a string containing four digits;
- "month", "day", "hours", "minutes", "seconds": Must be strings containing two digits;

- “minSubwindow”: Must be a positive integer, must be lower than or equal to “maxSubwindow”, must be a multiple of “differenceBetweenSubwindows” (unless it is equal to “maxSubwindow”);
- “maxSubwindow”: Must be a positive integer, must be greater than or equal to “minSubwindow”, must be a multiple of “differenceBetweenSubwindows” (unless it is equal to “minSubwindow”);
- “approximationEnabled”, “partialSearch”, “dukascopyStockInputData”: Must be Booleans;
- “differenceBetweenSubwindows”: Must be a positive integer (unless “minSubwindow” is equal to “maxSubwindow” – in this case its value is forced to 1 for the sake of compatibility);
- “beta”: Must be a positive integer between 1 and “minSubwindow”;
- “startEndDislocationLimit”: Must be greater than 0 and lower than (or equal to) 1, must be in the format “x / y”;
- “maxPeaksPerPattern”: Must be a positive integer;
- “minPeakWidth”: Must not be negative;
- “pointsDislocationLimit”: Must be greater than 0 and lower than (or equal to) 1, must be in the format “x / y”;
- “minConf”: Must be between 0 and 1, must be in the format “x / y”;
- “heightAffectsConf”, “distanceAffectsConf”, “greedySigmaEnabled”: Must be Booleans;
- “knn”: Must be a positive integer (or 0);
- “storageWindow”: Must be at least 2;
- “dataPath”, “logsPath”, “ratesPath”: Must be strings, must exist.

5.1.2 GPPDA: Targeted Improvements

As stated above, code has seen reorganizational changes for the sake of execution speed so that GPPDA has become the fastest possible implementation of the algorithm described so far. Not only is the algorithm fast in multi-window Pattern Discovery problems for specific kinds of signals, but it shows potential for other Pattern Discovery fields too, as we will see in the next section.

Creating a versatile tool like we did welcomes new challenges that we will face victoriously: while the case of an analysis involving a number of sub-windows lower than the machine's hardware concurrency is possible, the chances of it being single-windowed are very high in some research fields. We are undoubtably referring to the possibility of the parallelization potential not to be fully exploited, an event that presents itself as a limit we need to overcome.

A lot of work has been done to make the algorithm as parallelizable as possible, focusing on multiple levels (see the "Code Optimization: Multi-Threading (CPDA v3)" section for details). A great contribution to parallelization is provided by what we have referred to as "external multi-threading" alone: the sub-windows' tasks are equally spread amongst threads, making the most out of hardware concurrency. What about single-windowed analyses? Internal parallelization would be the only foothold we could rely on in that case, remembering the fact that internal multi-threading is used in approximate searches only.

In order to always make the most out of the machine's capabilities, a third level of parallelization has been introduced. If no changes had been made, one thread only would have received tasks to take care of, while the others would have died right after their creation. By introducing this feature, single-windowed exact searches involve threads operating on the same sub-window, equally dividing up work by means of distributed bookings: a single booker is now made out of a group of threads, rather than a single one. This operation comes at a cost we need to take into account: the distributed approach intrinsically prevents workers from having a complete view over the dataset, reducing the overall precision. Although partitioning exact searches may sound risky the benefits outweigh the costs, as we will better understand investigating the Anomaly Detection field further.

5.2 Contextualization of GPPDA in state-of-the-art Anomaly Detection algorithms

Although the algorithm has been extended to meet every possible need, it excels by a large margin when few specific sub-fields of the Pattern Discovery spectrum are tackled. As it is well known amongst computer engineers, there exists no silver bullet when it comes to solving diverse problems, even if they belong to the same research field: GPPDA stands its ground in the analysis of signals that have a tendency to show consecutive repetitions of patterns (sub-signals) in a big data environment, where the data sampling frequency is very high and potentially unstable.

Consecutiveness in time series tackles two specific fields GPPDA is able to monopolize in terms of results accuracy and execution speed, proposing itself as the state-of-the-art algorithm for such analyses:

- **Anomaly Detection**:
  It is performed on time series showing one single pattern that repeats throughout the entire signal. The search is exact and single-windowed, as it is important to detect subtle anomalies in small sub-signals that repeat with a well-known frequency.
  Example: ECG records

- **Consecutive Pattern Discovery**:
  It is performed on time series showing rather irregular data. The search is approximate and multi-windowed, as the goal is to detect consecutive repetitions of patterns of unknown length that rarely occur in an exact manner.
  Example: Stock data

Since Consecutive Pattern Discovery is an unexplored research field, we will focus our attention on Anomaly Detection. We will compare GPPDA to the world-class Pattern Discovery algorithms and provide commentary on results, speed and scalability, saving considerations about Consecutive Pattern Discovery in financial data for our conclusions in an ambitious perspective.

Almost all of the algorithms chosen for comparison belong to the Matrix Profile project, owned the University of California, Riverside. The project's algorithms have received great attention from the world of science, being described as the fastest, easiest to use, most accurate tools to perform Pattern Discovery and Anomaly Detection. The project started in 2016 with the presentation of UCR's first algorithm at the IEEE International Conference on Data Mining: [1]. Eamonn Keogh, Computer Science professor at UCR, described SCAMP (and its successor [2]) as "out of date" in a quick email chat with myself: this statement best outlines the dynamic nature of the Pattern Discovery field, remembering us that the best algorithm today may become the second best tomorrow.

As of 2021, the "fastest algorithm on the planet" when it comes to performing single-windowed analyses is [3], according to [4].

Before diving straight into the testing section, we will provide a theorical definition for the Matrix Profile (together with a visual definition) in order to have a glance at what is behind these algorithms:

We call the Matrix Profile $P(mT_s)$ *of a signal* $T(nT_s)$ *the signal resulting from the computation, for each* $x, q \in \mathbb{Z}$ *such that both* $T(nT_s)\,|\,n = x$ *and* $T(nT_s)\,|\,n = x + q$ *exist, of the lowest Euclidean Distance between the sub-signal* $T(nT_s)\,|\,x \leq n < x + q$ *and every possible sub-signal* $T(nT_s)\,|\,x' \leq n < x' + q$, $x' \geq x + q \vee x' \leq x - q$, *with* q *being the fixed, pre-chosen length of the sub-signals (subsequences) to be analyzed.*

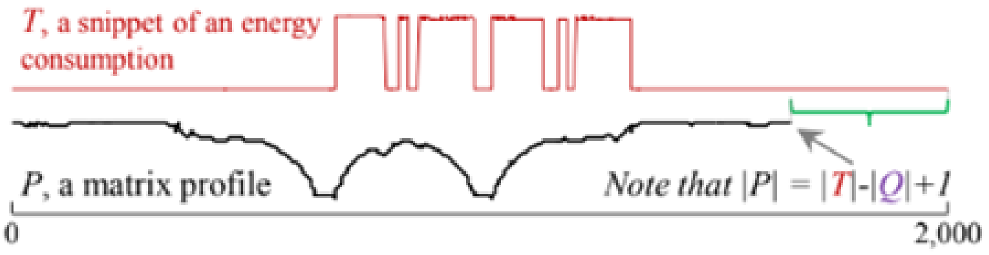


Figure 5.2: A Matrix Profile visualized (2)

Source: [1]

6. Experimental Results

6.1 Study cases

As we have previously ascertained in the "Data Source Choices" section, financial data is an optimal starting point for our research as it presents itself as the most diversified source of information as far as pattern occurrence is concerned. The diversification property of financial data has brought us to create a General Purpose Pattern Discovery Algorithm that paves the way for multiple research areas to be further explored, two of which being Anomaly Detection and Consecutive Pattern Discovery. As it was clarified in the "Contextualization of GPPDA in state-of-the-art Anomaly Detection algorithms" section, the Consecutive Pattern Discovery of financial data is an unexplored field that cannot be framed into any existing research work, but can and will be investigated in future works as it appears promising both in academic research and business contexts.

The proposed study cases provide material backed by the scientific community and include data belonging to both the fields of statistics and biomedical sciences. These domains capture our interest for they incorporate multiple examples of consecutive repetitions of patterns, that can be exploited to guarantee the accurate detection of anomalies in extremely small execution times. The latter statement is valid for datasets including billions (or even trillions) data points, for which manual analyses are not possible unless considerable human resources are invested.

The following is a list describing the datasets Anomaly Detection experiments were run on, with information regarding their details, granularity, missing values and presence of noise, in order for the reader to get a detailed overview of this thesis's study cases and their properties.

- **PhysioNet's ECG dataset** ([5, 9])
  Description: 48 30-minute ECG recordings performed on 47 subjects aged 23 to 89
  Purpose: Testing GPPDA's Anomaly Detection results
  Anomalies: Present in most recordings
  Frequency: 360 Hz
  Missing values: No
  Noise: Present in some recordings

- **Numenta's NAB v1.1** ([11, 12, 13])
  Description: 8 real-world timeseries data snippets including online advertisement clicking rates, the number of NYC taxi passengers, real time traffic data from the Twin Cities Metro area in Minnesota and Twitter mentions
  Purpose: Testing GPPDA's Anomaly Detection results
  Anomalies: Present in most recordings
  Frequency: Varies
  Missing values: No
  Noise: Absent

- **4 randomly generated time series**
  Description: Every time series is made up of 2^{19}, 2^{20}, 2^{22}, 2^{26} values ranging from 0 to 1000, each one being different from the previous
  Purpose: Testing GPPDA's Anomaly Detection speed and scalability

6.2 Design of the experiments

All of the experiments described in the above section were run in the same conditions. Below is a table showing the details of the experiments and the machine they were run on, accompanied by further information concerning each experiment's physical and logical context (more specific details will follow in the "Results" section).

Tested Algorithms (SoTA)	SCAMP (GPU), GPU-STOMP, SCAMP (CPU), SCRIMP++ ([10])
SoTA method	Matrix Profile
Tested Algorithm (Proposed)	GPPDA
Proposed method	Consecutive Pattern Discovery
Test Date	Varies (Spring 2021)
Dataset	Varies (see the "Study cases" section for details)
Dataset Width n	Varies (from 2^{11} to 2^{26})
Minimum Window Width ω_m	Varies (from 30 to 360)
Maximum Window Width ω_M	Varies (from 30 to 360)
Number of Windows λ	1
Machine Model	Acer Nitro 5 AN515-54-72-DX
CPU	Intel Core i7-9750H @ 2.60 GHz
GPU	NVIDIA GeForce GTX 1650
RAM	16 GB DDR4 2 667 MT/s
Threads used (CPU algorithms)	Varies (12 for GPPDA and SCAMP (CPU), 1 for SCRIMP++)

CUDA cores used (GPU algorithms)	896
OS	Windows 10 Home v20H2, 64 bits
C++ version	C++ 17
CUDA version	11.2
MATLAB version	R2021a

Table 6.1: Design of the experiments - SoTA Algorithms and GPPDA

We can clearly see that all of the SoTA algorithms make use of the same method, aiming to retrieve a time series' Matrix Profile, which is defined as follows:

We call the Matrix Profile $P(mT_s)$ of a signal $T(nT_s)$ the signal resulting from the computation, for each $x, q \in \mathbb{Z}$ such that both $T(nT_s) \,|\, n = x$ and $T(nT_s) \,|\, n = x + q$ exist, of the lowest Euclidean Distance between the sub-signal $T(nT_s) \,|\, x \leq n < x + q$ and every possible sub-signal $T(nT_s) \,|\, x' \leq n < x' + q$, $x' \geq x + q \vee x' \leq x - q$, with q being the fixed, pre-chosen length of the sub-signals (subsequences) to be analyzed.

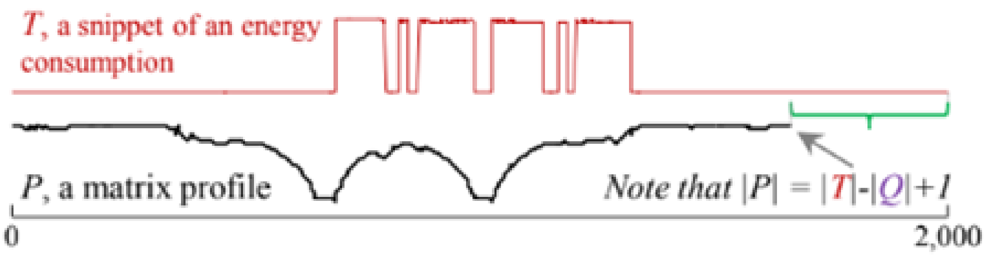


Figure 6.1: A Matrix Profile visualized (3)

Source: [1]

On the opposite hand, GPPDA proposes a new approach that aims to retrieve every Consecutive Pattern present throughout the signal, which is best described by the following definition:

We call "Consecutive Pattern" in time series analysis a pattern occurring, with a certain degree of approximation, twice in a consecutive manner.

The degree of the approximation is obviously zero in the above study cases, as the anomalies to be spotted may be very subtle. Precision is our top priority in the Anomaly Detection field: fortunately, GPPDA's analysis is configurable in such a way that we can successfully achieve our goal, by means of an exact and single-windowed search. As we will better comprehend in the "Results" section, the window width to be chosen varies alongside the dataset's nature both for GPPDA and the Matrix Profile algorithms.

The tests performed on PhysioNet's ECG dataset and Numenta's NAB v1.1 will show the goodness of the GPPDA's results over the SoTA algorithms, both in terms of precision and recall. Performing experiments on the 4 randomly generated datasets will, instead, provide useful information about speed and scalability, finalizing the outline of the algorithms' performances.

All the experiments are fully reproducible: GPPDA is available on [14] upon direct request.

6.3 Results

GPPDA and SCAMP (the fastest Matrix Profile algorithm on the planet[4]) were both tested on a publicly available ECG dataset provided by PhysioNet and referenced in [6]. The dataset consists of an heterogeneous collection of 48 ECG recordings (1,806 seconds at 360 Hz each) performed on 25 male subjects aged 32 to 89 and 22 female subjects aged 23 to 89. Every recording is composed of two time series (MLII and Vx, commonly referred to as ECG's higher and lower signals respectively), adding up to a total of 62.4 million data points divided across 96 different files: each recording was downloaded by [5], converted into two ASCII files and analyzed using both GPPDA and SCAMP.

Since each patient's heart rate ranged differently during the recordings, it would have not been a good decision to choose a single window width to configure all the analyses with. Remembering the fact that it must be possible to compare two algorithms regardless of the window width that has been chosen, the parameter was selected in such a way that both GPPDA and SCAMP would be able to compare at least a heartbeat to another completely. In order to do so, the window width ω was calculated as follows:

$$\omega = \lceil f_s \cdot \frac{1}{2f_{mp}} \rceil, \tag{6.1}$$

where f_s is the time series' sample frequency and f_{mp} is the minimum frequency for the main pattern to repeat. Let us consider $f_s = 360\,Hz$ and $f_{mp} = 1.15\,Hz$, meaning that the patient's minimum heart rate during the recording was $60f_{mp} = 69bpm$: the resulting window width will be $\omega = 157$. This configuration, along with the storage window counterpart being equal to 2, mathematically guarantees (for GPPDA and, by definition, for SCAMP) the coverage of every area of the dataset.

The method of the analysis consists of computing every sub-signal's nearest neighbor (let it be the result of a GPPDA's partial search with "knn" set to "1" or a SCAMP's full search) and displaying it in a bidimensional space: the points that stand out the most from the others indicate an anomaly in terms of absence of a similar sub-signal across the neighborhood.

The following are some of the results obtained both from GPPDA in the above configuration and SCAMP with the corresponding setup

(patient 100: –window=157 –profile_type=1NN;

patient 234: –window=131 –profile_type=1NN

– see [7, 8] for details).

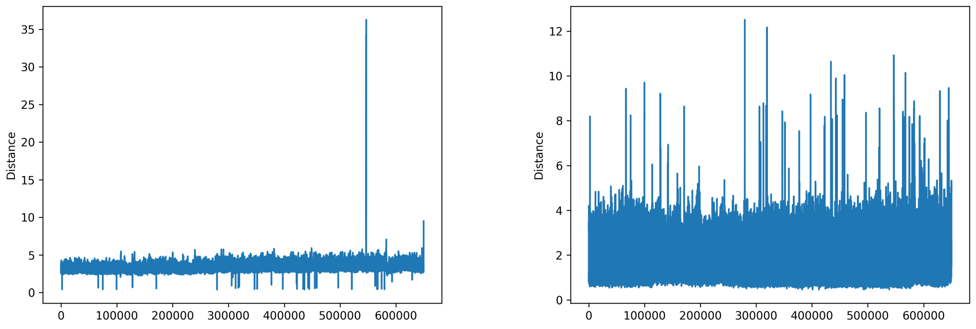


Figure 6.2: Patient 100, MLII (GPPDA on the left, SCAMP on the right)

It is evident that SCAMP was not able to outline a meaningful solution with the same configuration used for GPPDA. The anomaly is not able to stand out in SCAMP because the UCR's algorithm decontextualizes sub-signals, comparing them to every other sub-signal present throughout the dataset and resulting in closer nearest neighbors for potential anomalies. If we repeat the experiment after doubling SCAMP's window width (GPPDA's results would not change), we will see some improvements:

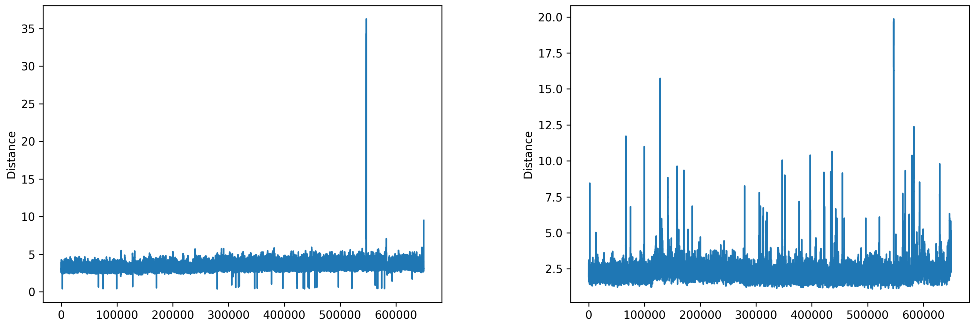


Figure 6.3: Patient 100, MLII (GPPDA on the left, SCAMP - double window - on the right)

Both the algorithms are now able to detect the only anomaly present in this recording, a Premature Ventricular Contraction (PVC) at minute 25:13, as pointed out in [9]. Nevertheless, the Matrix Profile obtained with SCAMP still displays unclear results, making us wonder whether there is an optimal value for the window width such that results are the clearest. We must consider that not having a guideline for choosing the correct window width induces uncertainty in a potential user.

It is worth noticing how not only performing Matrix Profiles in the Anomaly Detection field results in chaotic outputs, but it also prevents the algorithm from detecting multiple, similar anomalies (we will better see this with patient 234).

Below is the anomaly detected by the algorithms:

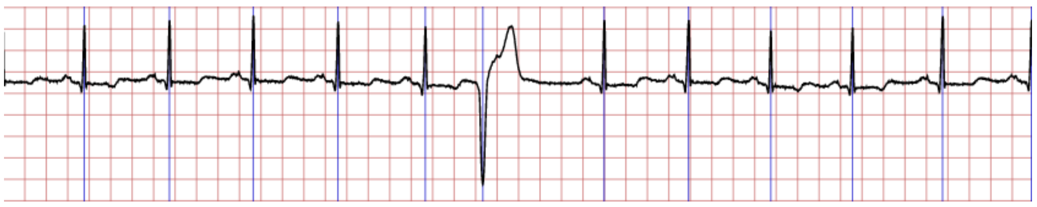


Figure 6.4: PVC in patient 100

Let us keep our experiments going.

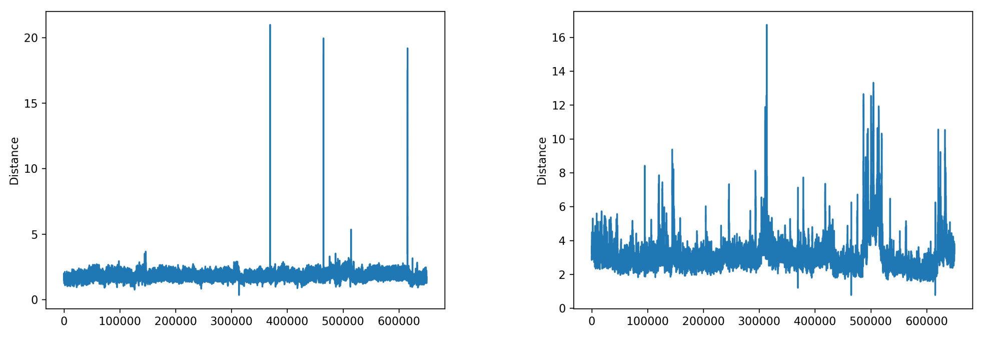


Figure 6.5: Patient 234, V1 (GPPDA on the left, SCAMP on the right)

Two PVCs were to be found in this recording, one at minute 17:02 and the other at minute 21:26. Only GPPDA pointed them out correctly, even though SCAMP was configured with twice the original window width: the first two vertical lines in the left chart correspond to the anomalies, while the third one is just noise. At a first glance, it may seem that both GPPDA and SCAMP detected the anomalies, but if we pay attention to the approximate timestamp resulting from the temporal collocations of the PVCs we find out that what SCAMP displays has nothing to do with the actual discords.

Below are the two anomalies detected by GPPDA: does anything look alike?

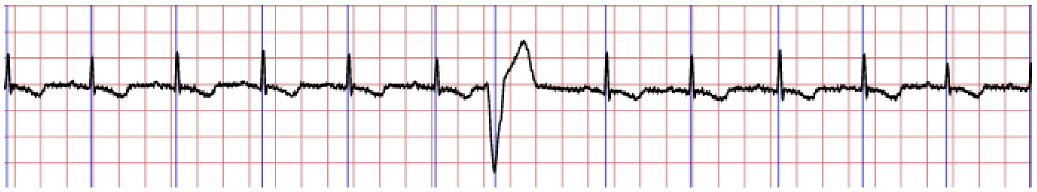


Figure 6.6: PVC in patient 234

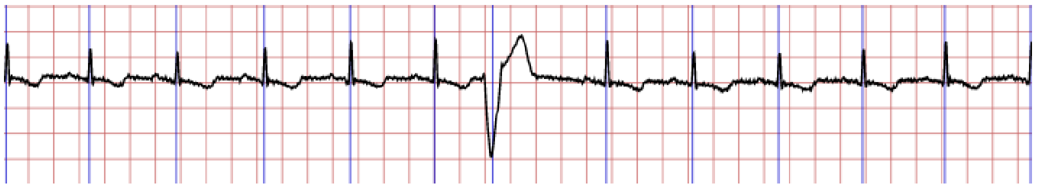


Figure 6.7: PVC in patient 234 (2)

The above illustrations would be enough to proclaim GPPDA as the absolute winner, but we are not satisfied until we make our algorithm win from every possible perspective. After taking a look at the usual test details table we will examine the speed and scalability properties of GPPDA, comparing them to those of SCAMP (GPU), GPU-STOMP and SCAMP (CPU).

In order to obtain the fairest results from our comparison, a script was used to generate a random time series of length 2^{26} made up of values ranging from 0 to 1000, each one being different from the previous (this constraint guarantees GPPDA operates in a worst-case environment). Tests were executed on four incremental subsets (having lengths 2^{19}, 2^{20}, 2^{22} and 2^{26}) belonging to the newly generated signal, providing a truthful indicator of the algorithms' behavior in terms of speed and scalability.

Table 6.2: Speed and Scalability test details for SoTA algorithms

Test Date	Varies (Spring 2021)
Dataset	Randomly generated signal
Dataset Width n	Varies (2^{19}, 2^{20}, 2^{22} and 2^{26})
Minimum Window Width ω_m	256
Maximum Window Width ω_M	256
Number of Windows λ	1
OS	Windows 10 Home v20H2, 64 bits
Threads used (CPU algorithms)	Varies (12 for GPPDA and SCAMP (CPU), 1 for SCRIMP++)
CUDA cores used (GPU algorithms)	896
CPU	Intel Core i7-9750H @ 2.60 GHz
GPU	NVIDIA GeForce GTX 1650
RAM	16 GB DDR4 2 667 MT/s
C++ version	C++ 17
CUDA version	11.2
MATLAB version	R2021a

The following table and charts show the algorithms' execution times for each of the generated datasets (estimated times are followed by an asterisk):

Algorithm	**Language**	$\mathbf{2^{19}}$	$\mathbf{2^{20}}$	$\mathbf{2^{22}}$	$\mathbf{2^{26}}$
GPPDA	C++	21s	**42s**	**3m4s**	**50m8s**
SCAMP (GPU)	C++, CUDA	**13s**	52s	13m50s	2d11h*
GPU-STOMP	C++, CUDA	16s	1m4s	17m14s	3d2h*
SCAMP (CPU)	C++	3m4s	12m32s	3h21m*	35d16h*
SCRIMP++	MATLAB	12m59s	52m49s	14h35m*	166d18h*
STOMP	-	-	-	-	-
STAMP	-	-	-	-	-

Table 6.3: Speed and Scalability test results for SoTA algorithms

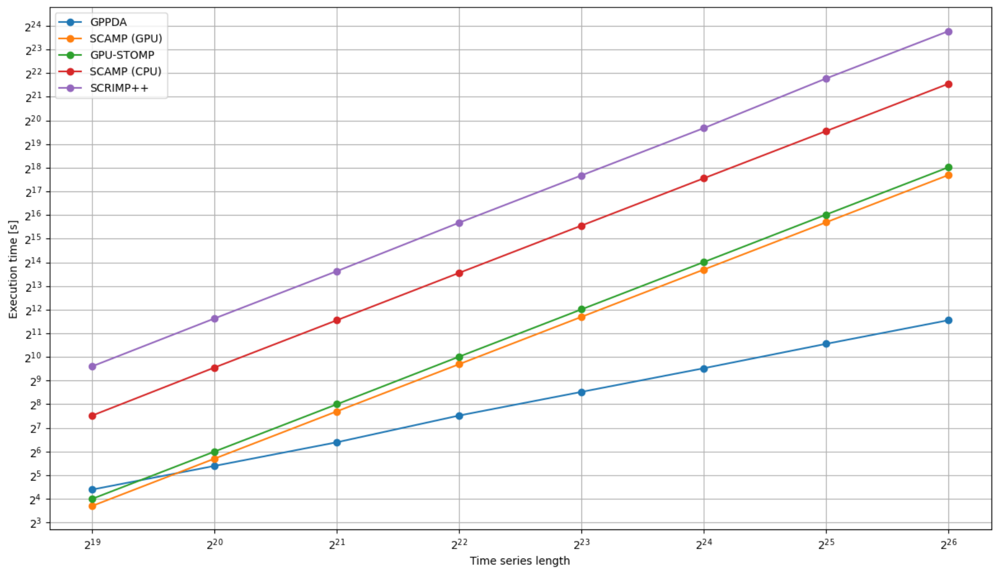


Figure 6.8: Speed and Scalability test results for SoTA algorithms

Before getting into the commentary, it is worth pointing out that SCRIMP++ is a single-threaded, anytime algorithm provided directly by E. Keogh as a MATLAB script that aims to replace STAMP and STOMP entirely, the latter being unavailable to the scientific community as of 2021.

As we can clearly see from the logarithmic chart above, doubling the size of the dataset results in doubled execution times for GPPDA, while it results in quadrupled execution times for the UCR's algorithms. This happens because GPPDA's worst-case complexity is linear with respect to n, differently from what happens for SCAMP, STOMP and SCRIMP++ whose nature implies a quadratic complexity. It is worth clarifying that the average window width configuration for GPPDA in the ECG test was much lower than 256: the purpose of this higher value is to have it easily memorized for the reader and to show the intersection between GPPDA's line and SCAMP's happening somewhere between 2^{19} and 2^{20} data points.

As the last statement suggests, the intersection would not have been visible had the window width been lower, meaning that ω influences the execution times. If we had set $\omega = 512$ we would have seen GPPDA's blue line translating two blocks upwards on the y-axis (worst-case execution times would have been quadrupled), shifting the intersection somewhere between 2^{21} and 2^{22} data points. Although this is true in theory, the chances of needing to analyze smaller datasets with bigger windows decrease significantly in practice, as it would require little effort for a human to perform a manual search of signals showing a few hundred repetitions, rather than downloading code from the internet and making it work on their machine.

Since what has been stated above is easily arguable, we will bring the attention to a wider range of applications.

Numenta has made available NAB (Numenta Anomaly Benchmark) v1.1, a novel benchmark for evaluating algorithms for anomaly detection in streaming, real-time applications, providing a vast array of real-world and artificial timeseries data snippets.

GPPDA was tested alongside SCAMP on 8 datasets divided as follows, with ω being twice the minimum size allowed for GPPDA and execution times floating far below 1 second for both the algorithms:

- 2 real datasets that include online advertisement clicking rates, only one showing anomalies (around 2^{11} data points);
- 1 real dataset that includes the numbers of NYC taxi passengers, showing five anomalies occurring during the NYC marathon, Thanksgiving, Christmas, New Year's day, and a snow storm (around 2^{13} data points);
- 3 real datasets that include real time traffic data from the Twin Cities Metro area in Minnesota (metrics being occupancy and speed) showing several anomalies (around 2^{11} data points);
- 2 real datasets that include Twitter mentions of large publicly-traded companies such as Google and Amazon (around 2^{14} data points);

The goal of this analysis is to provide a larger and heterogeneous accuracy benchmark for GPPDA, approaching diverse contexts that aim to stress the recurring nature of pattern data (pattern repetition frequency, data points per pattern and degree of anomaly) and ignoring execution times.

The following pages present, for each analyzed dataset, a chart quadruple depicting the results obtained with GPPDA (in the upper left corner), the results obtained with SCAMP (in the upper right corner) and the original time series (in the lower right and left corners), as well as a brief commentary addressing some of the reader's possible questions.

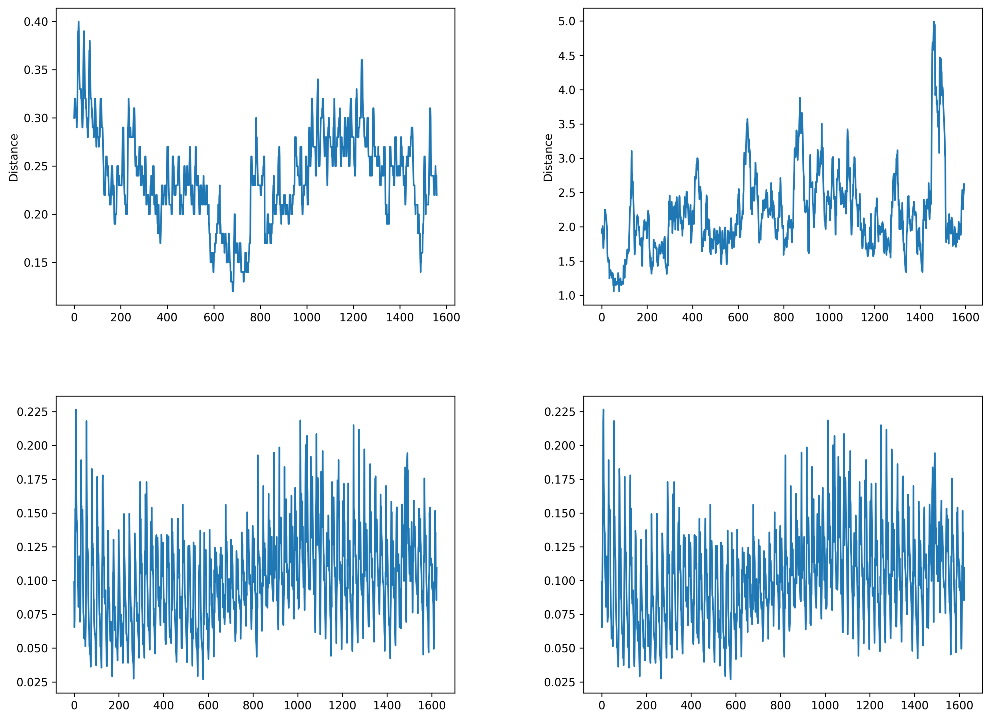


Figure 6.9: Clicking rates (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 30$

File name: realAdExchange/exchange-2_cpc_results.csv

Commentary: No anomalies were present in this dataset. We will make use of an index $\Gamma = \frac{R_M - R_{avg}}{R_{avg}}$ to indicate the protrusion of the biggest anomaly, where R_M and R_{avg} are the maximum and average Euclidean Distances computed by an algorithm: the higher Γ, the higher the extent of the biggest computed anomaly. Be careful: this value must not be interpreted as an overall performance score for the algorithm.

$\boldsymbol{\Gamma_{\mathbf{GPPDA}}} = 0.68$

$\boldsymbol{\Gamma_{\mathbf{SCAMP}}} = 1.28$

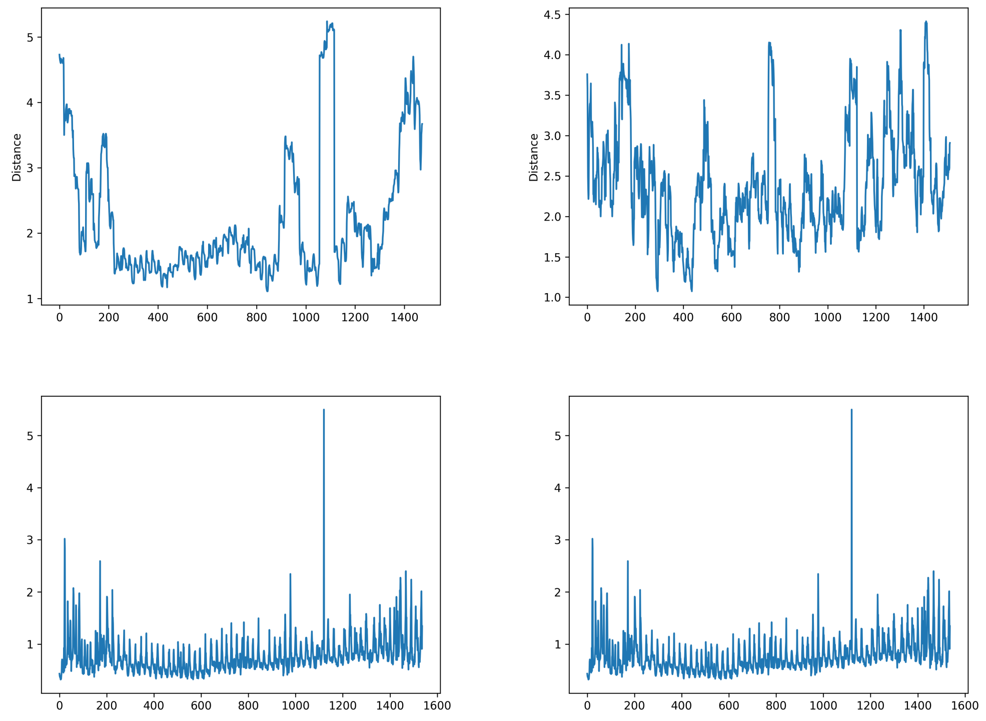


Figure 6.10: Clicking rates 2 (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 30$

File name: realAdExchange/exchange-3_cpm_results.csv

Commentary: A bunch of anomalies were present in this dataset, all of which were correctly detected by GPPDA only.

$\boldsymbol{\Gamma_{\mathrm{GPPDA}}} = 1.36$

$\boldsymbol{\Gamma_{\mathrm{SCAMP}}} = 0.85$

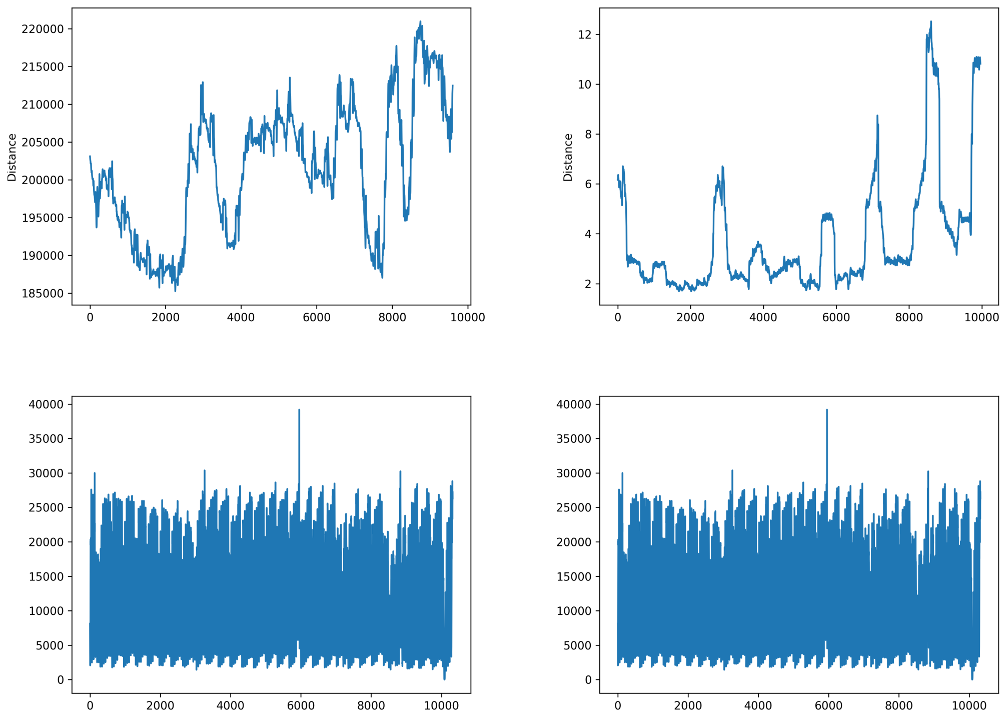


Figure 6.11: NYC taxi (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 360$

File name: realKnownCause/nyc_taxi.csv

Commentary: This dataset shows a high variability in consecutive patterns, that is the reason why GPPDA's results show anomalies to a more conspicuous extent. The ability to show changes in trends is another point of force for GPPDA. Peaks in anomalies are consistent in both GPPDA and SCAMP nonetheless.

$\boldsymbol{\Gamma}_{\mathbf{GPPDA}} = 0.10$

$\boldsymbol{\Gamma}_{\mathbf{SCAMP}} = 2.37$

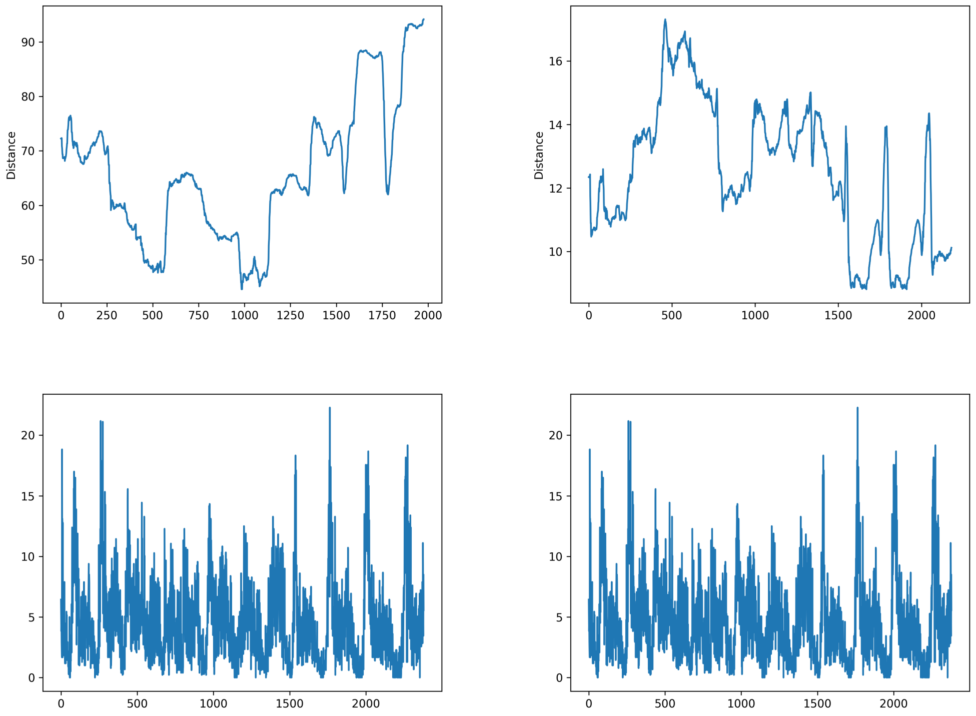


Figure 6.12: Traffic (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 200$

File name: realTraffic/occupancy_6005.csv

Commentary: Considerations are omitted for this dataset as they would be of speculative nature. The reader is invited to draw their own conclusions.

$\boldsymbol{\Gamma}_{\mathbf{GPPDA}} = 0.43$

$\boldsymbol{\Gamma}_{\mathbf{SCAMP}} = 0.39$

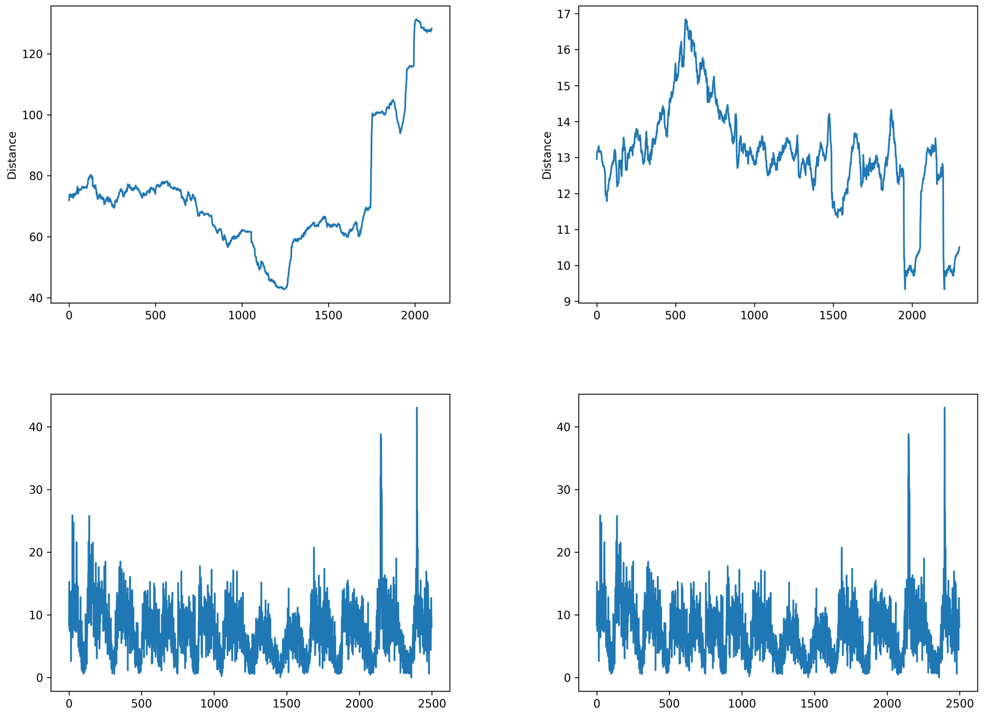


Figure 6.13: Traffic 2 (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 200$

File name: realTraffic/occupancy_t4013.csv

Commentary: As we have previously had the opportunity to conclude for the ECG dataset, SCAMP fails to detect similar anomalies.

$\mathbf{\Gamma_{GPPDA}} = 0.79$

$\mathbf{\Gamma_{SCAMP}} = 0.29$

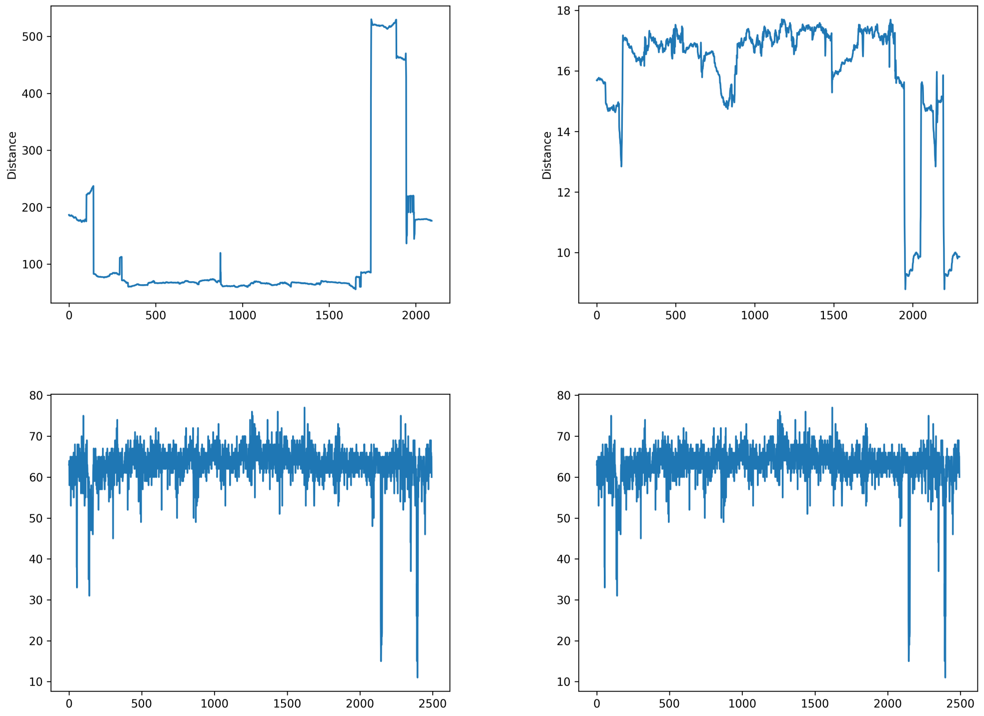


Figure 6.14: Traffic 3 (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 200$

File name: realTraffic/speed_t4013.csv

Commentary: The same condition that brings SCAMP to fail occurs once again.

$\boldsymbol{\Gamma_{\mathrm{GPPDA}}} = 3.15$

$\boldsymbol{\Gamma_{\mathrm{SCAMP}}} = 0.11$

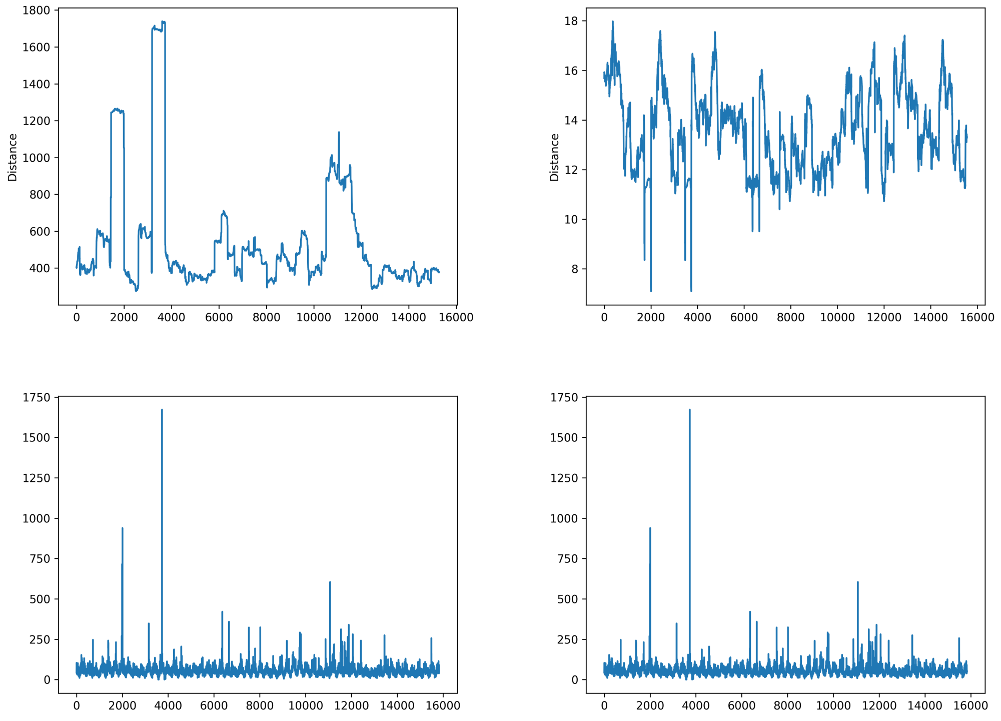


Figure 6.15: Tweets (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 275$

File name: realTweets/Twitter_volume_AMZN.csv

Commentary: Another fail for SCAMP.

$\boldsymbol{\Gamma}_{\mathbf{GPPDA}} = 2.22$

$\boldsymbol{\Gamma}_{\mathbf{SCAMP}} = 0.32$

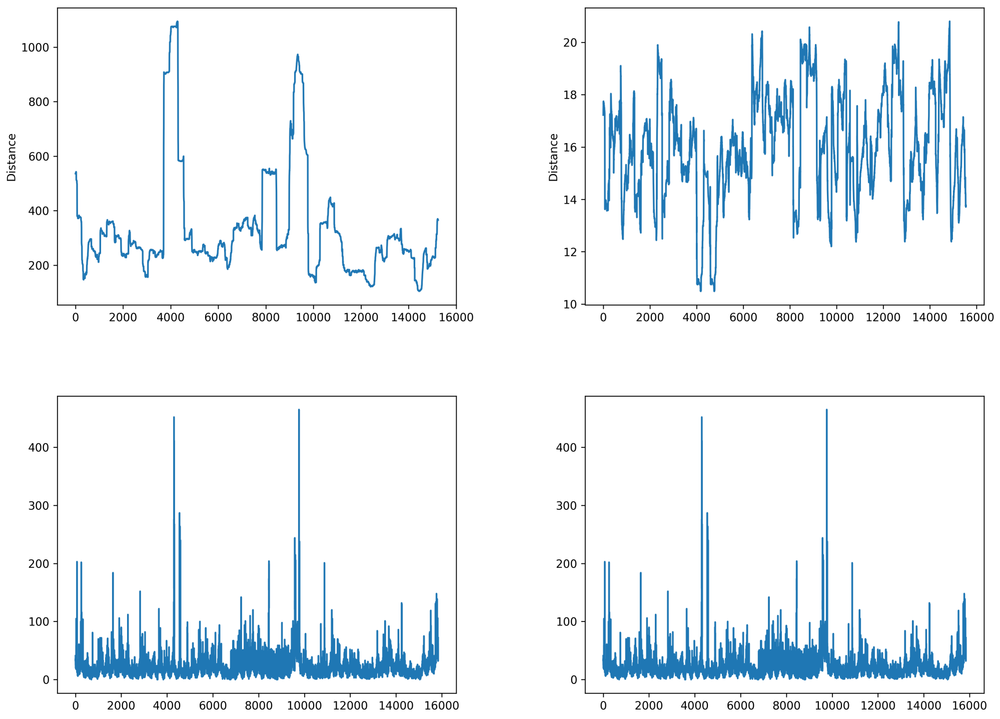


Figure 6.16: Tweets 2 (GPPDA on the left, SCAMP on the right)

$\boldsymbol{\omega} = 300$

File name: realTweets/Twitter_volume_GOOG.csv

Commentary: Another fail for SCAMP.

$\boldsymbol{\Gamma}_{\mathbf{GPPDA}} = 2.27$

$\boldsymbol{\Gamma}_{\mathbf{SCAMP}} = 0.30$

7. Conclusions and future work

We examined the state-of-the-art algorithms for Data Science challenges that require Pattern Discovery techniques to be dealt with, highlighting their pitfalls in terms of time complexity and results meaningfulness for time series analysis. We questioned the validity of a global approach in comparing n-long subsequences with one another, illustrating the inability of Matrix-Profile-based algorithms to compensate the scalability costs of this approach with meaningful results.

We introduced CPDA, a novel algorithm that targets the discovery of patterns in a consecutive manner, freeing similarity searches both from the "brute-force curse" and the limiting quadratic attributes that have been characterizing state-of-the-art Pattern Discovery Algorithms for years. We then extended CPDA to GPPDA, a General Purpose version of the Consecutive Pattern Discovery Algorithm that can be configured to address multiple Pattern-Discovery-related problems, converging our efforts to provide a solid Anomaly Detection tool. We demonstrated how the Consecutive approach provides advantages in Anomaly Detection time series analyses both in terms of scalability and relevance of results for recurring anomalies.

GPPDA paves the way for further research: the sequential nature of the algorithm, together with its cutting-edge performance, could guarantee the real-time detection of anomalies in continuous streams of data. This would allow immediate actions to be taken, automatically or manually, after a sensor (or any device that outputs data in a time series format) detects a data flow diverging from its normal behavior.

Although the real-time Anomaly Detection is a promising research path per se, more ambitious projects can be undertaken: the prediction path is as interesting as high are the stakes of being able to make short-term forecasts in a profitable context such as the financial markets. The possibility of classifying patterns (just like a PVC is a known, classified anomaly in a person's heartbeat) would allow for Artificial Intelligence techniques to be employed in the prediction of patterns showing consecutively, including the possibility for them to repeat multiple times over. As stated many times throughout this thesis work, financial markets (and, in particular, the XAU-USD market) appear to be a literal gold mine of possibilities.

Ultimately, the prediction prospect can also be visualized from the Anomaly Detection's point of view: many Anomaly-Detection-based Failure Prediction Algorithms exist in literature, with possibilities growing as our imagination draws additional scenarios where Anomaly Detection comes purposefully in help for prediction problems.

References

[1] STAMP: C. M. Yeh et al., "Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View That Includes Motifs, Discords and Shapelets," 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 2016, pp. 1317-1322, doi: 10.1109/ICDM.2016.0179.
Accessed 30 Sep 2021

[2] STOMP: Y. Zhu et al., "Matrix Profile II: Exploiting a Novel Algorithm and GPUs to Break the One Hundred Million Barrier for Time Series Motifs and Joins," 2016 IEEE 16th International Conference on Data Mining (ICDM), Barcelona, Spain, 2016, pp. 739-748, doi: 10.1109/ICDM.2016.0085.
Accessed 30 Sep 2021

[3] SCAMP: Zimmerman, Zachary & Kamgar, Kaveh & Shakibay Senobari, Nader & Crites, Brian & Funning, Gareth & Brisk, Philip & Keogh, Eamonn. (2019). Matrix Profile XIV: Scaling Time Series Motif Discovery with GPUs to Break a Quintillion Pairwise Comparisons a Day and Beyond. 74-86. 10.1145/3357223.3362721.
Accessed 30 Sep 2021

[4] https://www.cs.ucr.edu/~eamonn/MatrixProfile.html
Accessed 30 Sep 2021

[5] https://physionet.org/content/mitdb/1.0.0/
Accessed 30 Sep 2021

[6] Chuah, Mooi Choo & Fu, Fen. (2007). ECG Anomaly Detection via

Time Series Analysis. 123-135. 10.1007/978-3-540-74767-3_14.
Accessed 30 Sep 2021

[7] https://scamp-docs.readthedocs.io/en/latest/cli.html
Accessed 30 Sep 2021

[8] https://scamp-docs.readthedocs.io/en/latest/profiles.html
Accessed 30 Sep 2021

[9] https://archive.physionet.org/physiobank/database/html/mitdbdir/records.htm
Accessed 30 Sep 2021

[10] SCRIMP++: Y. Zhu, C. M. Yeh, Z. Zimmerman, K. Kamgar and E. Keogh, "Matrix Profile XI: SCRIMP++: Time Series Motif Discovery at Interactive Speeds," 2018 IEEE International Conference on Data Mining (ICDM), 2018, pp. 837-846, doi: 10.1109/ICDM.2018.00099.
Accessed 30 Sep 2021

[11] https://numenta.com
Accessed 30 Sep 2021

[12] https://github.com/numenta/NAB
Accessed 30 Sep 2021

[13] Ahmad, Subutai & Lavin, Alexander & Purdy, Scott & Agha, Zuha. (2017). Unsupervised real-time anomaly detection for streaming data. Neurocomputing. 262. 10.1016/j.neucom.2017.04.070.
Accessed 30 Sep 2021

[14] https://github.com/sttata/GPPDA
Accessed 30 Sep 2021

[15] Fu-lai Chung, Tak-chung Fu, R. Luk and V. Ng, ”Evolutionary time series segmentation for stock data mining,” 2002 IEEE International Conference on Data Mining, 2002. Proceedings., 2002, pp. 83-90, doi: 10.1109/ICDM.2002.1183889.
Accessed 30 Sep 2021