Next Higher Point: two Novel Approaches for Computing Natural Visibility Graphs

Patrick Daniel[1], Vanessa Freitas Silva[2], and Pedro Ribeiro[2]

[1] Faculdade de Ciências, Universidade do Porto, Porto, Portugal,
[2] CRACS-INESC TEC, Faculdade de Ciências, Universidade do Porto, Porto, Portugal

Abstract. With the huge amount of data that has been collected over time, many methods are being developed to allow better understanding and forecasting in several domains. Time series analysis is a powerful tool to achieve this goal. Despite being a well-established area, there are some gaps, and new methods are emerging to overcome these limitations, such as visibility graphs. Visibility graphs allow the analyses of times series as complex networks and make possible the use of more advanced techniques from another well-established area, network science. In this paper, we present two new efficient approaches for computing natural visibility graphs from times series, one for online scenarios in $\mathcal{O}(n \log n)$ and the other for offline scenarios in $\mathcal{O}(nm)$, the latter taking advantage of the number of different values in the time series (m).

Keywords: times series, visibility graphs, online data, offline data

1 Introduction

In different fields, series of data indexed by time are recorded for different purposes, and these series are known as time series. Time series analysis is used to analyse these data, revealing hidden patterns and trends with view to forecasting [10]. Traditional techniques, such as those from statistical linear analysis are commonly employed for this purpose. However, these methods have limitations, particularly in capturing complex relationships in dynamic systems. In response, Network Science offers complementary tools to enhance the analysis of time series and uncover new insights into temporal dependencies [11].
Network science is a field that studies complex networks (or graphs) and these are well known to be able to model real-world scenarios [2]. One popular method for mapping time series to complex networks is the visibility graph [6]. This approach gained interest because network science, with its advanced tools, can uncover hidden properties in the topological structures of resulting networks. For example, periodic series are converted into regular graphs, random series do so into random graphs and fractal series are converted into scale-free networks [6]. Visibility graphs can be obtained using different visibility criteria such as natural visibility, horizontal visibility, and many others [11]. Figure 1 shows a simple illustration for the first criterion.

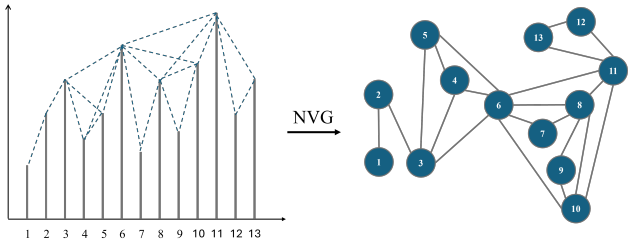


Fig. 1: Illustration of Natural Visibility Graph (NVG) for a toy time series.

In the last decade, lots of research are being conducted in time series analysis through visibility graphs [11]. Unfortunately, the first faced problem during these studies is the time complexity of the algorithms that map time series to visibility graphs, since all known methods are still quadratic in worst-case for the natural visibility graph. Motivated by this, in this paper, we presented a brief overview of existing algorithms and proposed two new efficient methods for computing Natural Visibility Graphs (NVG). One method is designed for online data scenarios, and other for offline scenarios, both using a simple data structure known as Monotonic Stack. We also compared the time complexity of the proposed NVG algorithms with existing ones in the literature and we conducted experiments to analyze the results.

2 Background

2.1 Visibility Graphs

The Visibility Graph (VG) in time series analysis is the resulting graph of the mapping process based on a geometric criterion. This graph is defined as follows:

1. The vertices of the graph are labeled according to the timestamps, with a one-to-one correspondence between data points and graph vertices.
2. There is an edge between two nodes in the graph if, and only if, the corresponding data points "*see*" each other based on a visibility criterion.

The chosen visibility criterion defines the type of VG. In this work, we focus specifically on the Natural Visibility Graph (NVG).
According to the natural visibility criterion, two data points (x_l, y_l) and (x_r, y_r) (assuming $x_l < x_r$ without loss of generality) in the time series are mutually visible if, and only if, every intermediate data point (x_m, y_m) with $x_l < x_m < x_r$ meets the following condition:

$$\frac{y_m - y_l}{x_m - x_l} < \frac{y_r - y_l}{x_r - x_l}. \tag{1}$$

This means that the maximum slope of any line connecting data point (x_l, y_l) and any intermediate data point (x_m, y_m) is less than the slope of the line connecting (x_l, y_l) and (x_r, y_r), as can be observed in Figure 1.
NVG is undirected and always connected, as adjacent data are naively visible to each other. Moreover, NVG doesn't change under affine transformations of the time series, as the visibility criterion is invariant to such transformations [6].

2.2 Natural Visibility Graph Algorithms Review

Since the proposal of the concept by Lacasa *et al.* [6], many algorithms for mapping times series to NVGs have been proposed. In this section, we will briefly list the existing methods and their respective time and space complexities.

Method Name	Auxiliary Space	Offline Scenario (Balanced TS)	Offline Scenario (worst-case)	Online Scenario (Balanced TS)	Online Scenario (worst-case)
Basic [7]	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	—	—
DC [8]	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	—	—
SC [4]	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	—	—
BST [3]	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
LOT [5]	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$[1]	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Table 1: NVG algorithms summary

Given a time series $[(x_1, y_1), (x_2, y_2), .., (x_n, y_n)]$, the naive mapping algorithm for VG enumerates all possible triples of data points $((x_l, y_l), (x_m, y_m), (x_r, y_r))$ with $1 \leq l < m < r \leq n$, checks the visibility criterion and adds the edge (l, r) to the resulting VG, if the visibility criterion is satisfied, leading to a time complexity of $\mathcal{O}(n^3)$. In order to speed up the mapping process, the methods presented in Table 1 have been proposed. These methods approach the problem using different algorithmic strategies (*e.g.* Divide and Conquer (DC) [8], Sort and Conquer (SC) [4] or Backward Iteration (*e.g.* LOT [5])) and data structures (*e.g.* Binary Search Tree (BST) [3]) to take advantage of the time series structures, including the slopes of the lines or even the balance [2].

2.3 Monotonic Stack

Monotonic Stack is a common data structure in computer science. It functions like a standard stack but maintains a key *invariant*: at any time, all the elements in the stack are monotonically increasing (or decreasing). This invariant enables faster solutions in many algorithms, such as the state-of-the-art algorithm for horizontal visibility graphs proposed in [9].

Given a sequence of values $[a_1, a_2, .., a_n]$, we aim to find, for each index $1 \leq i \leq n$, the next element to the left that is greater than a_i. A naive two-nested loop solution takes $\mathcal{O}(n^2)$, but a Monotonic Stack reduces the complexity to $\mathcal{O}(n)$.

Starting with an empty stack, we process the sequence from left to right (a_1 to a_n). For each element a_i, if the stack is empty, no greater element exists to the left of a_i. Otherwise, we check the top of the stack and while the top is smaller than or equal to a_i, we pop the top. If the stack becomes empty, no greater element exists to the left of a_i; otherwise, the element on the top of the stack is the next greater element. Then, we push a_i onto the stack (empty or not).

Since each element is pushed once in the stack and popped at most once, summing up two operations for each element in the worst-case, this results in a linear time, $\mathcal{O}(n)$. The key point of this approach is that, when a new element a_{n+1} is added, the overall complexity is still $\mathcal{O}(n)$, since that an update costs $\mathcal{O}(1)$.

[1] Depends on the used algorithm.

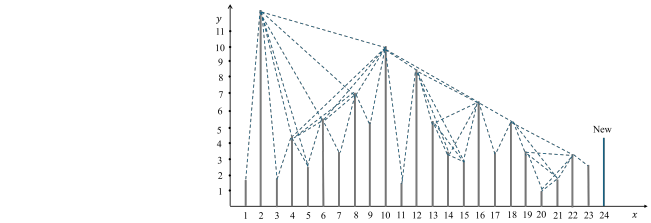
[2] A time series is called balanced when the highest point divides the series into two sub-series of (almost) equal size and the same happens in both sub-series.

3 *Next Higher Point*: two Novel Approaches

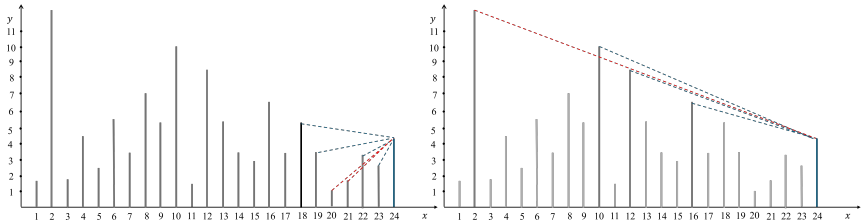
In this section, we present two novel approaches for computing an NVG from a time series, called *Next Higher Point* (NHP) algorithms.
The first, *Online* NHP, is designed for online scenarios (*i.e.*, data streaming), handling new data in a different way, that guarantees an average time of $\mathcal{O}(\log n)$ for each update in balanced series and faster than $\mathcal{O}(n)$ for a generic time series in practice (as show in Section 4), outperforming state-of-the-art approaches.
The second, *Offline* NHP, is designed for offline scenarios and leverages the number of distinct values in the time series. As far as we know, this is the first NVG method with complexity based on the number of distinct values (m), achieving $\mathcal{O}(nm)$ time. This makes our algorithm ideal for real-world cases where few distinct values are usually present [1].

3.1 Approach 1: Online Next Higher Point

Online NHP is designed to work in online fashion. We can assume that we have n existing time series data points and their corresponding NVG is already computed (without loss of generality, as it can be empty if we consider $n = 0$). Suppose a new data point arrives, as illustrated in Figure 2a, and we want to insert it into the NVG and build the corresponding edges. The proposed algorithm involves two main stages, which we describe below.



(a) Arrival of a new data point



(b) First stage (c) Second stage

Fig. 2: Illustration of the Online NHP algorithm. The top figure shows a data update in an existing NVG and the bottom figures depict the first and second stages of Online NHP, respectively. Dashed blue lines represent added edges, and red represent lines those not added.

First Stage: Iterate through the existing (past) data points and check the visibility criterion (adding edges to the graph if Equation 1 is satisfied) while the height of the new data is greater than the iterated data. Figure 2b illustrates this process. After finding the first higher data, go to the second stage.

Second Stage: In this stage, it keeps iterating but in a slightly different fashion. Instead of going naively to the next past data point, we jump to the next point on the left of the currently iterated point that is higher than it (and always checking the visibility criterion to establish the connections if necessary), as there cannot be visibility between two points if there is a point between them that is higher than or equal to the highest of the two. This process is illustrated in Figure 2c.

Online NHP can be efficiently implemented using a Monotonic Stack and an array to store the position of the next greater element (see Algorithm 1). Since this is online, we have an existing time series, a Monotonic Stack, a Next Greater Array, and a graph (which can be empty if $n = 0$). To insert a new data point, these data structures are passed by reference to avoid overhead, and the method modifies them directly without returning anything.

Algorithm 1 Online NHP

Require: A time series $TS \coloneqq (X, Y)$, where $\mathbf{X}$ and $\mathbf{Y}$ are 1-indexed arrays of length N, the Monotonic Stack $\mathbf{S}$, the Next Greater Array $\mathbf{A}$, the graph $\boldsymbol{\mathcal{G}}$ and the new data point (x, y), all passed by reference, except the new data point
Ensure: (empty)

while S is not empty **and** $Y[S.top()] \leq y$ **do** $S.pop()$
if S is not empty **then** $A[N+1] \leftarrow S.top()$
else $A[N+1] \leftarrow -1$ ▷ there is no point on the left that is greater than y
$S.push(N+1)$
$TS.addPoint((x, y))$
$\mathcal{G}.addNode(N+1)$
$l \leftarrow N$
$minSlope \leftarrow +\infty$
while $l \geq 1$ **do**
    $lineSlope \leftarrow slope((X[l], Y[l]), (x, y))$
    if $lineSlope < minSlope$ **then**
        $\mathcal{G}.addEdge(l, N+1)$
        $minSlope \leftarrow lineSlope$
    if $Y[l] \geq y$ **then** $l \leftarrow A[l]$
    else $l \leftarrow l - 1$

Complexity. The use of the Monotonic Stack and the Next Greater Array of past data points, requires an $\mathcal{O}(n)$ space. On the other side, note that for very unbalanced cases (e.g. monotone increasing or decreasing time series), the time complexity reaches its worst-case, $\mathcal{O}(n^2)$, since each of the stages may cost $\mathcal{O}(n)$ time for each new data. But, these cases are not common in practice. However, in balanced cases, after n updates, starting from an empty series, both stages of the algorithm have an overall time complexity of $\mathcal{O}(n \log n)$, as proven next.

Proof (First Stage in a Balanced Time Series). in a balanced time series of length n, the midpoint performs $\frac{n}{2}$ visibility checks, the midpoints of each half perform $\frac{n}{4}$ checks, and this pattern continues for the remaining data points, as shown in Figure 3. Formally, the expected number of visibility checks for each point in a balanced times series of length n is given by:

$$\begin{aligned} E(T_n) &= \frac{1}{n} \cdot \frac{n}{2} + \frac{2}{n} \cdot \frac{n}{4} + \frac{4}{n} \cdot \frac{n}{8} + \ .. \ + \frac{2^h}{n} \cdot \frac{n}{2^{h+1}} \\ &= \sum_{i=0}^{h} \frac{1}{2} = (h+1) \cdot \frac{1}{2} = \frac{1}{2} \cdot \log_2(n+1), \end{aligned} \tag{2}$$

where $n = 1 + 2 + 4 + .. + 2^h$ and h is the number of levels of height in the time series, *i.e.*, the height of the recursion tree of the splitting process (if we consider that, recursively, each data point is dividing into equal parts the sub-series where it is the highest). This implies $n = 2^{h+1} - 1$ (without loss of generality, as n can always be approximated to the predecessor of the nearest greater power of two without changing the asymptotic order of the number of checks), and therefore, $h = \log_2(n+1) - 1$. Thus, each data point performs $\log_2 n$ visibility checks on average in the first stage.

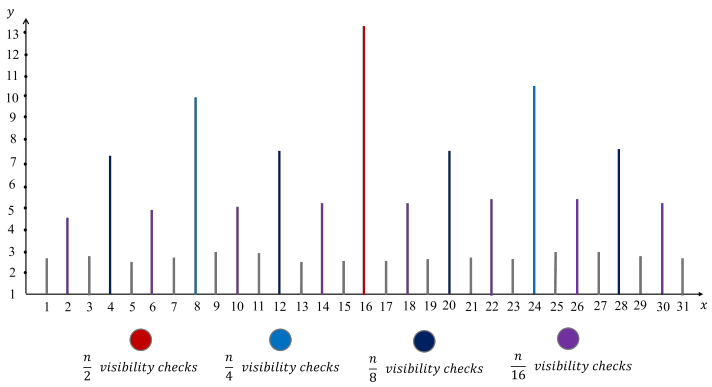


Fig. 3: Visibility checks made by each point in a balanced time series.

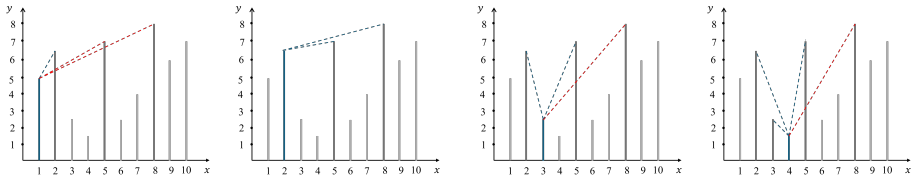
Proof (Second Stage in a Balanced Time Series). since each point splits a range into (almost) equal parts in balanced series, whenever we move from a point r to a higher point l, the range divided by point l is at least twice the range divided by r, as point r belongs to one of the parts divided by l. Thus, starting in any point, we can jump at most $\log n$ times to higher points in the second stage, since jumping more than $\log n$ times implies reaching a point that divides a range greater than n.

Thereby, each update costs $\mathcal{O}(\log n)$ on average (since each data point performs $\log n$ checks per stage) and the overall time complexity for n updates is $\mathcal{O}(n \log n)$, for balanced cases.

3.2 Approach 2: Offline Next Higher Point

The second proposed approach is for offline scenarios. The Offline NHP uses a constructive strategy, building the NVG by connecting the data points to higher points (see Figure 4). Note that:

1. Any point never needs to check visibility with points that are lower than it, as if the visibility criterion is satisfied with a point lower than it, the connection will be established when this lower point is under consideration.
2. Considering a single point, if we focus only on one of its sides (left or right), the data heights satisfying the visibility criterion with it are strictly increasing. This suggests that for each data point we can apply the strategy used in Online NHP (separately) on its both sides, jumping to the next higher point instead of iterating through each point.



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 4: Illustration of the first steps of Offline NHP algorithm in a toy time series.

This algorithm also uses Monotonic Stacks, but uses two arrays instead of one: one to store the position of the next greater point on the left and one for the right. Unlike the first approach, this algorithm first calculates the position of the next greater point for every data point and then tries to establish connections, that's why it is offline. Furthermore, the connections on the left and right side of each data point are handled separately, as can be seen in Algorithm 2. A corner-case arises with points of equal height, handled during the next greater computation (see Algorithm 2, lines 5 - 7). Since we are considering only next greater points, two points of equal height would not be connected. Therefore, we have to deal with this individually. A point can only be connected to at most one point of its height on each side, and we only need to handle this in one of the sides, since the graph is undirected.

Complexity. Like the Online NHP, the space used by Offline NHP is $\mathcal{O}(n)$, because of the two stacks and the two arrays. However, the interesting part of this algorithm is the time complexity, that is $\mathcal{O}(nm)$, where m is the number of distinct values in the time series. The proof of this lies in the fact that, since for each point, we only jump to higher and higher points, we can not jump more than m times (on each side), since with each jump the height increases (i.e. changes to a higher value) and we have only m distinct values, so the total cost is $\mathcal{O}(nm)$. Furthermore, as Offline NHP is very similar to the second stage of Online NHP, the time complexity for a balanced time series is also $\mathcal{O}(n \log n)$ because for each of the n data points we cannot jump more than $\log_2 n$ times on each of its sides, as the range divided by the next higher point is at least twice

of the range divided by the current point (as seen in 3.1). However, Offline NHP may degenerate into $\mathcal{O}(n^2)$ in very unbalanced time series with n distinct values (*e.g.* $m = n$), but it can never be worst than this since the number of distinct values is always bounded by the length of the time series.

Algorithm 2 Offline NHP

Require: A time series (X, Y), where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are 1-indexed arrays of length N
Ensure: The Natural Visibility Graph $\boldsymbol{\mathcal{G}}$

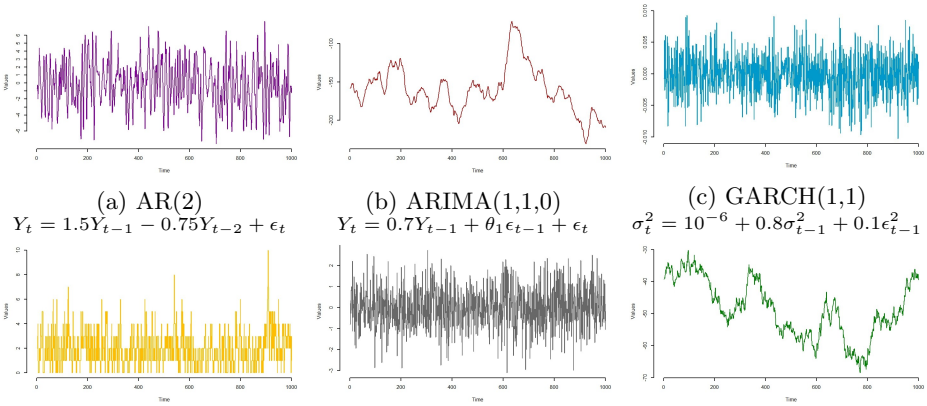
Let S_l and S_r be two empty stacks and A_l and A_r be two empty arrays of size N
for $i \leftarrow 1$ *to* N **do**
    $equalHeightConnections \leftarrow 0$
    while S_l is not empty **and** $Y[S_l.top()] \leq Y[i]$ **do**
        if $Y[S_l.top()] = Y[i]$ **and** $equalHeightConnections = 0$ **then**
            $\mathcal{G}.addEdge(S_l.top(), i)$
            $equalHeightConnections \leftarrow equalHeightConnections + 1$
        $S_l.pop()$
    if S_l is not empty **then** $A_l[i] \leftarrow S_l.top()$
    else $A_l[i] \leftarrow -1$ ▷ there is no point on the left that is greater than $Y[i]$
    $j \leftarrow N - i + 1$
    while S_r is not empty **and** $Y[S_r.top()] \leq Y[j]$ **do** $S_r.pop()$
    if S_r is not empty **then** $A_r[j] \leftarrow S_r.top()$
    else $A_r[j] \leftarrow N + 1$ ▷ there is no point on the right that is greater than $Y[j]$
for $m \leftarrow 1$ *to* N **do**
    $l \leftarrow A_l[m]$
    $minSlope \leftarrow +\infty$
    while $l \geq 1$ **do**
        $lineSlope \leftarrow slope((X[l], Y[l]), (X[m], Y[m]))$
        if $lineSlope < minSlope$ **then**
            $\mathcal{G}.addEdge(l, m)$
            $minSlope \leftarrow lineSlope$
        $l \leftarrow A_l[l]$
    $r \leftarrow A_r[m]$
    $maxSlope \leftarrow -\infty$
    while $r \leq N$ **do**
        $lineSlope \leftarrow slope((X[m], Y[m]), (X[r], Y[r]))$
        if $lineSlope > maxSlope$ **then**
            $\mathcal{G}.addEdge(m, r)$
            $maxSlope \leftarrow lineSlope$
        $r \leftarrow A_r[r]$

4 Experimental Results

4.1 Environment and Datasets

For a fair comparison, all algorithms were implemented in C++11 and all experiments were conducted in the same environment, using an i7-7500U Intel CPU with 16GB of RAM and running Windows 11 Home.

To ensure confidence and understand algorithms performances, we used six synthetic datasets generated from time series models proposed in [12], representing six distinct types of time series (see Figure 5). Each dataset contains 100 time series, each with 10^5 points. We also performed two types of experiments.



(a) AR(2) $Y_t = 1.5Y_{t-1} - 0.75Y_{t-2} + \epsilon_t$

(b) ARIMA(1,1,0) $Y_t = 0.7Y_{t-1} + \theta_1\epsilon_{t-1} + \epsilon_t$

(c) GARCH(1,1) $\sigma_t^2 = 10^{-6} + 0.8\sigma_{t-1}^2 + 0.1\epsilon_{t-1}^2$

(d) INAR(1) $Y_t = 0.5 * Y_{t-1} + \epsilon_t, \epsilon_t \sim Po(1)$

(e) White Noise $\epsilon_t \sim N(0, 1)$

(f) Random walk $Y_t = T_{t-1} + \epsilon_t$

Fig. 5: Illustration of one instance of each simulated time series model (with only 10^3 data points for presentation purposes). Mathematical formulation of the datasets are included. See [12] for more details.

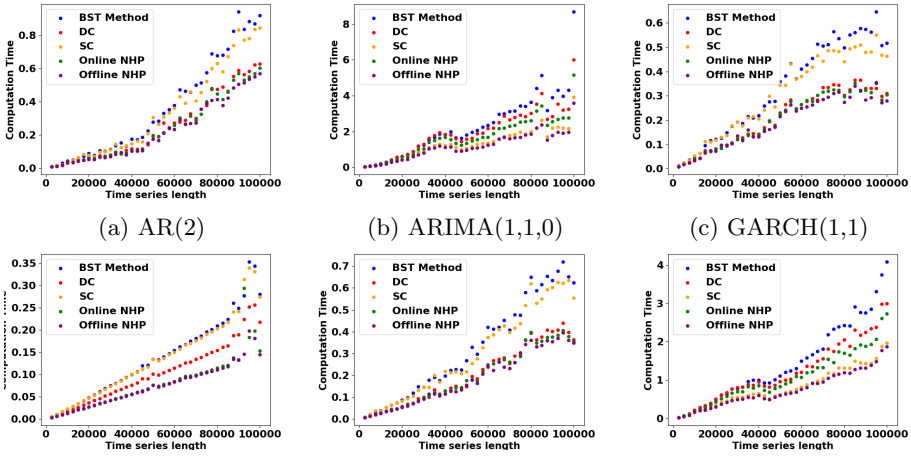
Offline experiment: we assume that we have all data points of the time series beforehand and then we run the algorithm to build the NVG. Each time presented in Section 4.2 is the mean of ten repetitions for the respective size and each size is a multiple of 2500. For this experiment, only DC, SC, BST, and our two approaches were considered. LOT Framework wasn't considered because it uses any of the previous algorithms in its offline mode. Although Online NHP and BST Method are online, they were considered in this comparison, since an offline scenario is just a special case of an online scenario.

Online experiment: we simulate what we expect to happen in some real-life applications and we assume that there is an existing time series and the respective NVG is already computed (as well as auxiliary data structures of each algorithm i.e., the Binary Search Tree for the BST Method and Monotonic Stack and Next Greater Array for Online NHP) and data points arrive one after the other. We measured the time each algorithm takes to perform consecutive updates on the NVG. To fairness, only BST, LOT Framework, and Online NHP were considered in this experiment since they are the only known online algorithms for NVG.

4.2 Results

Offline Experiment As can be seen in Figure 6, our two approaches perform very well in all 6 datasets. The Offline NHP consistently outperforms all state-of-the-art methods regardless of the model, which suggests that the used strategy

is good. As we expected, the most significant difference in terms of computation time, between Offline NHP and state-of-the-art methods lies in INAR model (see Figure 6d), which is a model with integer values, meaning that the number of distinct values is small compared to the other five models.



(a) AR(2) (b) ARIMA(1,1,0) (c) GARCH(1,1)

(d) INAR(1) (e) White Noise (f) Random Walk

Fig. 6: Comparison between algorithms according to different time series models in offline scenario. All times are in seconds.

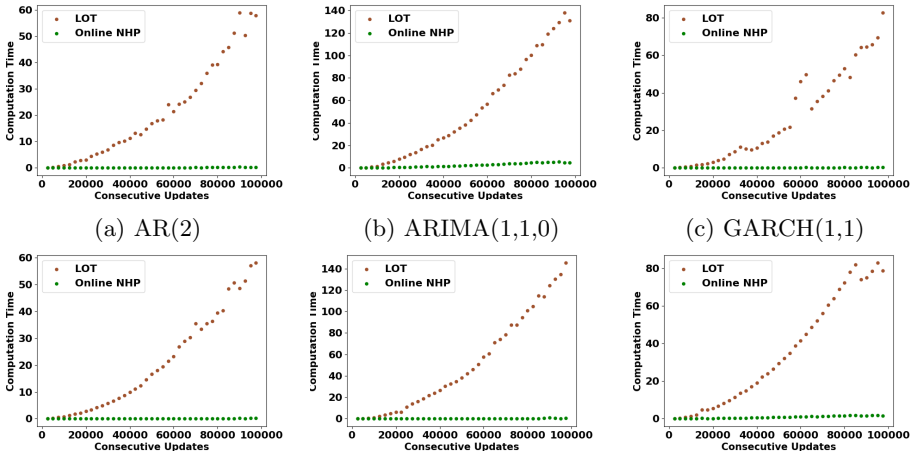
Even Online NHP, which was specially designed for online scenarios, also outperforms state-of-the-art methods in 4 out of 6 datasets that we have. Carefully analyzing the 2 datasets in which Online NHP loses to SC, i.e, ARIMA(1,1,0) (Figure 6b) and Random Walk (Figure 6f), we found out that the observed results are due to the first stage of Online NHP since it iterates naively on the left-hand side until it finds the first higher point, and in these models, it can take longer to find a higher point due to the distance between the peaks of the hill-shaped structures of these models, as can be seen in Figures 5b and 5f. Furthermore, it's clearly evident that our two approaches behave similarly on AR (Figure 6a), GARCH (Figure 6c), INAR (Figure 6d) and White Noise models (Figure 6e), because the structure of these models allows to quickly find a higher point, which consequently makes the Online NHP to easily switch to the second stage and start jumping to higher and higher points as the Offline NHP does.

Online Experiment In this experiment, we started with a time series of length 2500 and performed some consecutive updates. The aim behind this was to understand the algorithm's behavior in data stream scenarios.
The times presented in Table 2 show how BST Method is slow compared to LOT Framework and Online NHP. These results were predictable since BST Method rebuilds the whole NVG for each update, which is quadratic in the worst case, while LOT Framework and Online NHP work in an incremental fashion, only establishing the new connections. From Table 2 we can see that Online NHP is also faster than LOT Framework but the difference between them becomes more

evident in Figure 7. Remember that the LOT Framework always takes linear time for each update (regardless of the time series structure), while Online NHP varies from logarithmic to linear time for each update (according to the time series structure), that's why our proposed method is faster.

	BST Method	LOT Framework	Online NHP
AR(2)	48.422	0.293	0.010
ARIMA(1,1,0)	172.191	0.377	0.055
GARCH(1,1)	34.4885	0.323	0.012
INAR(1)	55.428	0.501	0.014
White Noise	56.840	0.404	0.012
Random Walk	123.517	0.356	0.034

Table 2: Comparison between algorithms according to different time series models in online scenario with 5000 consecutive updates. All times are in seconds.



(a) AR(2) (b) ARIMA(1,1,0) (c) GARCH(1,1)

(d) INAR(1) (e) White Noise (f) Random Walk

Fig. 7: Comparison between LOT Framework and Online NHP for different time series models in online scenario. BST Method is excluded due to its significantly larger time scale (see Table 2). All times are in seconds.

5 Conclusion

In this paper, we have given a brief overview of existing NVG algorithms and presented two new efficient approaches, one for online data and another for offline. We also made an experimental analysis, comparing the proposed approaches with the existing algorithms. The experiments show that both proposed approaches end up outperforming state-of-the-art methods in their respective scenarios.
Online NHP was designed to tackle the NVG construction in data stream scenarios. Theoretically, we proved that it is $\mathcal{O}(n \log n)$ for balanced time series and $\mathcal{O}(n^2)$ in the worst-case. Conducted experiments show that our method is faster than the competing methods.

Offline NHP was designed for offline scenarios. To the best of our knowledge, it's the first proposed method to compute NVG that depends on the number of distinct values in the time series (m), which guarantees $O(nm)$ time but it may degenerate into $\mathcal{O}(n^2)$ when $m = n$. However, the carried out experiments, illustrate that the used strategy makes Offline NHP faster than existing methods. Furthermore, since both methods are based on the same data structures (to store exactly the same information), Monotonic Stack and Next Greater Array, we realized that both can be joined in a single algorithm to get a powerful hybrid algorithm that can compute efficiently NVG in both offline and online scenarios.

References

1. Adhi, M., Hasan, R., Noman, F., Mahmood, S.F., Naqvi, A., Rizv, A.u.H., et al.: Range for normal body temperature in the general population of pakistan. JPMA. The Journal of the Pakistan Medical Association **58**(10), 580 (2008)
2. Costa, L.d.F., Oliveira Jr, O.N., Travieso, G., Rodrigues, F.A., Villas Boas, P.R., Antiqueira, L., Viana, M.P., Correa Rocha, L.E.: Analyzing and modeling real-world phenomena with complex networks: a survey of applications. Advances in Physics **60**(3), 329–412 (2011)
3. Fano Yela, D., Thalmann, F., Nicosia, V., Stowell, D., Sandler, M.: Online visibility graphs: Encoding visibility in a binary search tree. Physical Review Research **2**(2), 023,069 (2020)
4. Ghosh, S., Dutta, A.: An efficient non-recursive algorithm for transforming time series to visibility graph. Physica A: Statistical Mechanics and its Applications **514**, 189–202 (2019)
5. Huang, Y., Deng, Y.: Linear-time online visibility graph transformation algorithm: for both natural and horizontal visibility criteria. arXiv preprint arXiv:2311.12389 (2023)
6. Lacasa, L., Luque, B., Ballesteros, F., Luque, J., Nuno, J.C.: From time series to complex networks: The visibility graph. Proceedings of the National Academy of Sciences **105**(13), 4972–4975 (2008)
7. Lacasa, L., Luque, B., Luque, J., Nuno, J.C.: The visibility graph: A new method for estimating the hurst exponent of fractional brownian motion. Europhysics Letters **86**(3), 30,001 (2009)
8. Lan, X., Mo, H., Chen, S., Liu, Q., Deng, Y.: Fast transformation from time series to visibility graphs. Chaos: An Interdisciplinary Journal of Nonlinear Science **25**(8) (2015)
9. Schmidt, J., Köhne, D.: A simple scalable linear time algorithm for horizontal visibility graphs. Physica A: Statistical Mechanics and its Applications **616**, 128,601 (2023)
10. Shumway, R.H., Stoffer, D.S.: Time Series Analysis and its Applications, 4 edn. 1431-875X. Springer (2017)
11. Silva, V.F., Silva, M.E., Ribeiro, P., Silva, F.: Time series analysis via network science: Concepts and algorithms. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **11**(3), e1404 (2021)
12. Silva, V.F., Silva, M.E., Ribeiro, P., Silva, F.: Novel features for time series analysis: a complex networks approach. Data Mining and Knowledge Discovery **36**(3), 1062–1101 (2022)