Online visibility graphs: Encoding visibility in a binary search tree

Delia Fano Yela,[1,*] Florian Thalmann,[1] Vincenzo Nicosia,[2] Dan Stowell,[1] and Mark Sandler[1]

[1]*Centre for Digital Music, School of Electronic Engineering and Computer Science, Queen Mary University of London, Mile End Road, London, E14NS, United Kingdom*

[2]*School of Mathematical Sciences, Queen Mary University of London, Mile End Road, London, E14NS, United Kingdom*

(Received 17 July 2019; accepted 6 February 2020; published 23 April 2020)

A visibility algorithm maps time series into complex networks following a simple criterion, and the resulting visibility graphs have recently proven to be a powerful tool for time series analysis. However, their direct computation is time-consuming and rigid, motivating the development of more efficient algorithms. Here we introduce a description of a method to compute visibility graphs online, which is highly efficient, compatible with batchwise progressive updates, and capable of assimilating new data without having to recompute the graph from scratch. We use a binary search tree to encode and store visibility relations, which can be decoded at a later stage into a visibility graph. The proposed encoder/decoder approach offers an online computation solution at no additional computational cost and makes it possible to use visibility graphs for large-scale time series analysis and for applications where online data assimilation is required.

DOI: 10.1103/PhysRevResearch.2.023069

I. INTRODUCTION

In the last decade, several methods to map time series into graphs have been proposed, under the hypothesis that appropriate graph representations can preserve the original time series information while providing alternatives to deal with nonlinearity and multiscale issues typical of complex signals [1–3]. This line of research represents a bridge between nonlinear signal analysis and complex network theory, and has been successfully applied to extract meaningful information from a variety of different systems in physics [4,5], finance [6–8], engineering [9], and neuroscience [10,11].

The most notable algorithms to construct a graph from an ordered sequence of data points are based on either correlation [12–14], recurrence [15–17], dependence [18,19], or visibility [20]. However, the visibility algorithms proposed by Lacasa *et al.* [20,21] are among the most popular, as they provide a deterministic and nonparametric symbolization of a time series preserving full information of its linear and nonlinear correlations. Visibility algorithms can also effectively deal with nonstationary signals and are in general computationally efficient. For this reason they have found numerous applications in diverse fields including image processing [22,23], number theory [24], finance [7,25], and neuroscience [26].

The straightforward computation of visibility graphs presents a worst-case time complexity quadratic in the length of the series. Even though such complexity should not be an issue for medium-sized series (10^4–10^5 points), it remains inefficient for longer ones. Therefore, faster algorithms have been proposed employing a "divide and conquer" (DC) approach, reducing the average-case time complexity to $O(n \log n)$ [27].

Both the direct and the DC method are offline algorithms, as they require all data points in the time series to be available before the graph is constructed. Consequently, the integration of new data points normally requires recomputing the visibility graph from scratch. This is indeed a major shortcoming, which limits the real-world applications of visibility graphs.

In this paper we introduce an online algorithm to compute visibility graphs. The proposed algorithm employs an encoder/decoder approach and is based on the representation of a time series (or of any ordered sequence of data points) through an appropriately constructed binary search tree. The binary search tree associated with a time series can be efficiently updated every time a new chunk of data becomes available, by merging the data with the binary search tree associated with the new data. The resulting data structure contains full information about the time series and can subsequently be decoded to obtain the corresponding visibility graph when required. The flexibility introduced by the encoder/decoder approach comes at no significant computational cost, as the proposed method has the same time complexity as the current fastest visibility algorithm (DC).

II. VISIBILITY GRAPHS

A visibility graph is obtained from an ordered sequence of values by associating each datum to a node and connecting two nodes with an edge if the corresponding data points are visible from each other. A point a is visible from the point b if one can draw a straight line from a to b without passing underneath any intermediate points. In this paper we will consider visibility as a symmetric relation, so that the resulting visibility graphs are undirected.

*Corresponding author: d.fanoyela@qmul.ac.uk

Published by the American Physical Society under the terms of the Creative Commons Attribution 4.0 International license. Further distribution of this work must maintain attribution to the author(s) and the published article's title, journal citation, and DOI.

The natural visibility criterion (NV) allows the visibility line between a and b to take any slope, whereas the horizontal visibility criterion (HV) is restricted to horizontal lines, as shown in Fig. 1(f). More precisely, given a time series

$$y = f(t)$$

of length n, two points (t_a, y_a) and (t_b, y_b) are said to be *naturally visible* if every intermediate point (t_c, y_c), such that $t_a < t_c < t_b$, fulfills the following simple geometrical criterion:

$$y_c < y_a + (y_b - y_a)\frac{t_c - t_a}{t_b - t_a}.$$

This natural visibility criterion will therefore establish the connections between nodes in the resulting natural visibility graph (NVg).

One can analogously map a time series into a horizontal visibility graph (HVg) where two points (t_a, y_a) and (t_b, y_b) are said to be *horizontally visible* if

$$\forall c : t_a < t_c < t_b \Longrightarrow y_a > y_c \text{ and } y_b > y_c.$$

From the definition of visibility it immediately follows that, for a set visibility criterion, the visibility graph associated with a given time series is unique. Moreover, any two subsequent data points of the time series are always connected by an edge, thus visibility graphs are connected and Hamiltonian [21]. In addition, visibility graphs are also locally invariant to rescaling on both horizontal and vertical axes (i.e., the first point on either side of a node i remains visible from i no matter how far apart they are), and invariant to vertical and horizontal translations (i.e., only the relative values of point determine visibility relations).

In Fig. 1(f) we show both the natural and horizontal visibility criteria at work on an arbitrary time series. Notice that horizontal visibility is a more stringent criterion than natural visibility, meaning that if two points are horizontally visible, then they are also trivially visible when using the natural visibility criterion. Consequently, the horizontal visibility graph of a time series is always a subgraph of the natural visibility graph associated with the same time series.

III. STATE OF THE ART

A straightforward approach to compute visibility graphs consists in checking whether any of the points of the time series is visible or not from every other point. This corresponds to evaluating the visibility criteria for every pair of points in the time series. Since we consider visibility as a symmetric relation, the total number of checks needed to obtain a visibility graph of a time series of n data points is equal to $n(n-1)/2$, corresponding to a $O(n^2)$ time complexity.

In the case of HV, one can take a step further and safely assume that no point after a value larger than the current value t_a will be horizontally visible from t_a. This observation effectively reduces the time complexity of the construction to $O(n \log n)$, and, in the case of noisy (stochastic or chaotic) signals, it can be proved that this algorithm has an average-case time complexity $O(n)$ [21]. Nevertheless, all pairs of

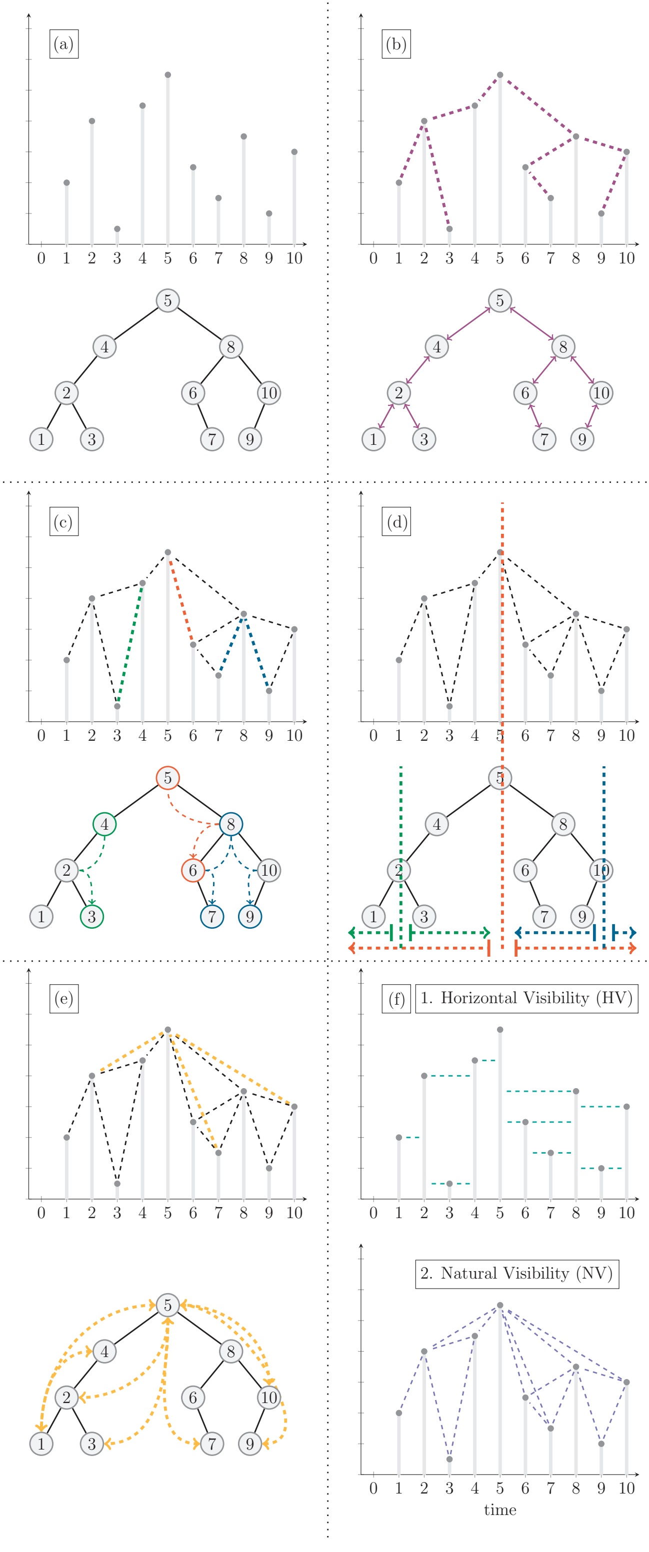

FIG. 1. Representation of the different steps of the proposed algorithm for visibility graphs computation. (a) The sample time series and its correspondent maximum binary search tree. (b) The connections deduced by the first connectivity rule. The second (c) and third connectivity rules (d). (e) The remaining checks needed to ascertain natural visibility. (f) The horizontal and natural visibility graph associated with the original time series.

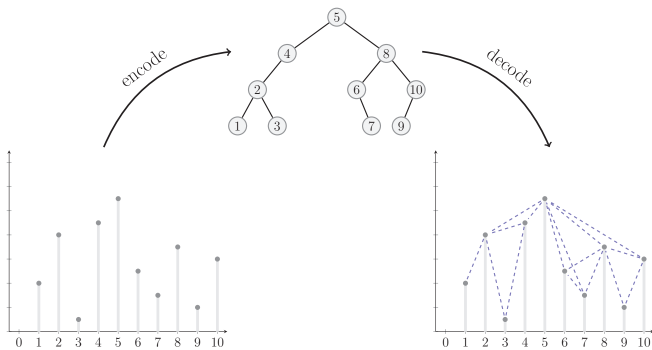

FIG. 2. Illustration of the encode/decode approach of the proposed method to calculate visibility graphs.

points need to be checked in the case of NV. From now on, this simple approach will be referred to as the basic method for both natural and horizontal visibility computation [28].

As an improved alternative for visibility computation, Lan *et al.* presented a "divide and conquer" (DC) approach [27]. This algorithm reduces the average-case time complexity of the construction of the natural visibility graph to $O(n \log n)$, and it significantly reduces computation time for most balanced time series.

The basic idea behind the DC algorithm is related to the horizontal visibility optimization mentioned above. Once the maximum value M of the time series is known, one can safely assume that the points on the right of M will not be naturally visible from the points on the left of M (the point M is effectively acting as a wall between the two sides of the time series). The same argument is then applied recursively to the two halves of the time series separated by M, where the local maxima subsequently found at each level are connected with an edge to the maxima at the level immediately above them. From now on, this improved method will be referred to as "divide and conquer" (or DC for short).

Both the basic method and DC are offline approaches, meaning that they require all the points of the time series to be accessible at the beginning of the computation. This rigid requirement limits the applicability of visibility graphs, especially in fields like telecommunications or finance, where there is a constant incoming flow of new data to be processed and assimilated. Moreover, in such big data scenarios, one tends to favor an initial overall high level analysis that will reveal the need for further processing. This work flow would benefit from dynamic algorithms unlike the ones presented above.

IV. PROPOSED METHOD: BINARY SEARCH TREE FOR VISIBILITY ENCODING

Here we present an encoding/decoding approach to compute visibility graphs. In the proposed method, the necessary visibility information is first encoded into an appropriately constructed binary search tree, which can then be successively decoded into a visibility graph when needed, as illustrated in Fig. 2.

As discussed in the following section, the binary search tree can be updated when new data arrive, and so the visibility can be decoded without having to reprocess old data points, thus allowing us to compute and update visibility graphs online. The proposed method maintains the same time complexity as the state-of-the-art natural visibility graphs computation, but it represents a substantial improvement for the computation of horizontal visibility graphs.

A. Encoding—Maximum binary search tree

The construction of a maximum binary search tree is fairly straightforward, and its corresponding pseudocode is shown in Algorithm 1. The first step is to sort the given time series in descending order of values, while storing the original position of each value in the time series. From now on, we will refer to the original positions as indices (i.e., t) and to the values of the times series simply as values [i.e., $y(t)$]. For repeated values in the sequence, the first encountered index will come first after sorting (stable sort).

Once we have a list of values sorted in descending order, together with the corresponding indices, we follow the standard procedure to build a binary search tree based on the indices. Every entry in the index list becomes a node, and each node has a left and right child, as shown in the data structure proposed in Algorithm 1 (i.e., *Node*). The first node of the binary tree (the one with no *parent*) is called *root*. In our case, the root is the index of the datum corresponding to the maximum value in the time series, which is also the first entry in the index list.

The next index, corresponding to the point with the second-largest value, is then added to the tree. If its index is smaller than the index of the root, it becomes the left child of root, whereas if its index is larger than that of the root, it becomes the right child of root (see function *add* in Algorithm 1). The procedure continues by comparing following index to the root; if it is smaller, the index recursively travels through the left subtree of root, while if it is larger it will travel through the right subtree. Each index continues descending the tree until an empty spot is found, as in the standard procedure to populate a binary search tree, until all the data points have been considered (see function *build_tree* in Algorithm 1).

In the case of the sample time series in Fig. 1(a), the maximum is in position 5 and therefore becomes the *root* of the binary tree. The point whose value is immediately smaller than the maximum is in position 4 (less than 5), so it becomes the left child of the root. The third point in the list is in position 2 and travels down the tree on the leftmost branch (smaller than both 5 and 4). The right branch of the tree is populated by the fourth point (in position 8), whose index is larger than the *root*. In Fig. 1(b) one may appreciate the correspondence between the time series and its associated binary tree structure. The time complexity of the procedure needed to encode the time series into the maximum binary search tree is $O(S + T)$ where $O(S)$ is the time complexity of sorting the series and $O(T)$ is the time complexity of the algorithm to construct the binary search tree. Sorting by comparison is known to be $O(n \log n)$ (e.g., by using either MergeSort of QuickSort), while constructing a binary search tree costs on average $O(n \log n)$. Hence the overall average-case time complexity of the encoding step is $O(n \log n)$.

Algorithm 1. Pseudocode of the algorithm used to build a maximum binary search tree.

```
Node {
  index : float    # x, input, argument
  value : float    # f(x), output
  left  : Node     # left child subtree
  right : Node     # right child subtree
    }

def buildTree(values : {float}, indexes: {float}):

    root ← Node()

    sorted_values = sort_descending(values)
    sorted_indexes= indexes[getIndex(sorted_values)]

    for (i, v) in (sorted_indexes, sorted_values):
        root.add(Node(index = i, value = v))

    return root


def add(self : {Node}, node : {Node}):

    if  self is empty :
            self.index  = node.index
            self.value  = node.value
    else:
        if node.index < self.index:
            self.left.add(node)
        else:
            self.right.add(node)
```

B. Decoding—Connectivity rules

The structure of the maximum binary search tree encodes sufficient information about the time series to allow an efficient construction of the corresponding horizontal visibility graph. The decoding procedure is based on the following connectivity rules, also illustrated in Fig. 1:

(1) All the nodes connected by an edge in the maximum binary search tree are visible to each other and therefore connected in the visibility graph [Fig. 1(b)]

(2) Each node of the maximum binary search tree sees all the nodes in the leftmost branch of the subtree rooted at its right child, as well as all the nodes in the rightmost branch of the subtree rooted at its left child [Fig. 1(c)]

(3) The nodes of the left subtree of a node i are not visible from the nodes of the right subtree of node i [Fig. 1(d)].

Note that, if there are no adjacent repeated values, the HVg is fully determined by these connectivity rules. In particular, when checking the connectivity rules, we simply skip a node if it has the same value as the current node. One can think of adjacent points with equal values as an interconnected “super node,” which takes the smallest index value when “seen” from the left and the largest index value when “seen” from the right or from above.

Since the horizontal visibility decoding will always be fully determined by the three connectivity rules above, its time complexity is the sum of the time complexity of the rules. Essentially, each rule can be reduced to a series of look-ups in a binary search tree, and each look-up operation has time complexity $O(\log n)$ in a balanced tree. These connectivity rules are applied to every node in the tree, and so the overall time complexity of decoding a HVg is $O(n \log n)$. This represents a major improvement over the state-of-the-art algorithms, which can ramp up to $O(n^2)$ in the worst-case scenario.

The construction of the NVg, instead, requires the creation of some connections that are not captured by the three connectivity rules above. Hence, in this case we need to perform additional visibility checks, as shown in Fig. 1(e). In particular, for each node i we must check the natural visibility criterion with each node in the subtree rooted at the right child of i and with each node in the subtree rooted at the left child of i. These additional checks do not modify the average-case time complexity [which remains $O(n \log n)$], but the worst-case scenario still depends on the actual structure of the time series and yields a worst-case time complexity $O(n^2)$ for monotonically increasing or decreasing time series.

C. Time complexity

In order to determine the time complexity of the proposed method, we will follow the standard procedure by considering the worst-case and average-case scenarios. In both NV and HV scenarios, the time complexity of the encoding stage is determined by the time complexity of the sorting algorithm used, which in general is $O(n \log n)$, and of the construction of the binary search tree, which is $O(n \log n)$. So in both cases encoding into a binary search tree costs $O(n \log n)$.

Decoding into a HVg is done via the three rules [illustrated in Figs. 1(b) to 1(d)], which require only a visit of the binary search tree [with time complexity $O(n)$]. Hence, the overall time complexity of encoding and decoding into a horizontal visibility graph is $O(n \log n)$.

The worst case for decoding into a NVg occurs with monotonically increasing, monotonically decreasing, or constant series, whose corresponding binary search trees take the form of a single branch. In this case, the second and third connectivity rules are trivial, leaving only the first rule and the additional natural visibility checks. More precisely, if the tree is a single branch, we need to check the natural visibility among $(n-1)(n-2)/2$ pairs of nodes, while the visibility of the remaining $(n-1)$ pairs of nodes is determined by the first connectivity rule. Even though this requires $(n-1)$ checks less than the basic implementation [which requires $n(n-1)/2$], the time complexity is still $O(n^2)$ for the worst-case scenario.

For the average case we assume the maximum binary search tree to be balanced. This means that the connectivity rules of the decoder will significantly reduce the overall number of visibility checks. If we consider a perfectly balanced binary tree as shown in Fig. 3, the inner left branch of the right subtree and the inner right branch of the left subtree of a node are visible to the parent node. These are represented in green in Fig. 3, where the root is the parent node. This means that the visibility between the root and all the rest of the nodes is unknown and needs to be checked.

Therefore we can deduce that the number of remaining visibility checks for the root in a balanced tree of height $h_{\max}$ is equal to $2^{h_{\text{root}}+1} - 1 - 2h_{\text{root}}$, where $2^{h_{\text{root}}+1} - 1$ is the total number of nodes below the root while $2h_{\text{root}}$ is the number of nodes whose visibility can be deduced by the three decoding rules (the green nodes in Fig. 3). Notice that the height of the root h_{root} corresponds to the maximum height of the balanced tree $h_{\max}$. The same reasoning applies to all other nodes. More

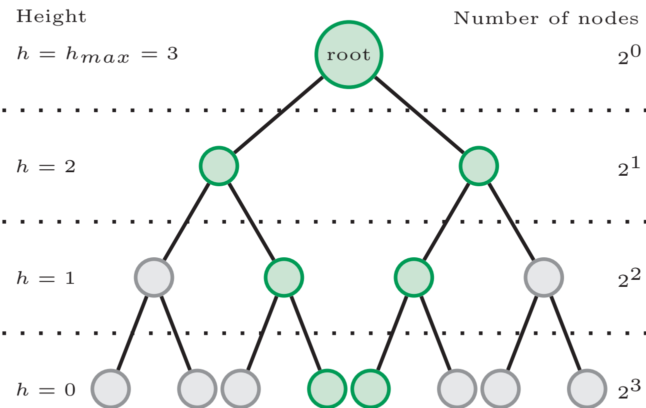

FIG. 3. Representation of a perfectly balanced tree of height 4. The nodes in green are visible to the root and this visibility can be deduced by the proposed decoder (i.e., the connectivity rules). The number of nodes at each height in a balanced tree can always be expressed in base 2.

precisely, for a node at height h, there will be $(2^{h+1} - 1 - 2h)$ remaining visibility checks to be performed.

In order to calculate the total number of remaining visibility checks, one needs to multiply the individual expression above by the number of nodes at that height $2^{h_{\max}-h}$ and sum across all heights where the checks are needed (all except the last two). Therefore, one can express the total number of remaining natural visibility checks in a perfectly balanced binary tree as follows:

$$\sum_{h=2}^{h_{\max}} 2^{h_{\max}-h}[2^{h+1} - (2h+1)]$$
$$= 2^{h_{\max}}\left[2(h_{\max}-1) - \sum_{h=2}^{h_{\max}} h2^{1-h} - \sum_{h=2}^{h_{\max}} 2^{-h}\right]$$

Since the maximum height of a balanced tree with n nodes is $h_{\max} = \log_2(n)$, the total number of operation is dominated by the first term of the expression above,

$$2^{h_{\max}}2(h_{\max}-1) = 2n[\log_2(n) - 1],$$

while the remaining terms will only introduce logarithmic corrections. In conclusion, the time complexity of the decoding for NVg is on average $O(n \log n)$.

The proposed method has the same average-case time complexity than the DC algorithm, thus improving on the original basic algorithm for both horizontal and natural visibility graphs. In the Numerical Experiment section below we will see that in practice our algorithm outcompetes the basic algorithm and performs as well as the DC approach, with the additional property of allowing for online assimilation of new data points.

V. ONLINE VISIBILITY GRAPHS: MERGING BINARY TREES

Every time a node is added to an existing binary search tree it "travels" down the tree, going left if smaller and right if larger, until it finds an empty space (see the pseudocode of the *add* function in Algorithm 1). Therefore when a node is added to an existing binary tree there is no need to recalculate the tree structure from scratch. Due to the fact that the proposed encoder is a binary search tree, it is possible to efficiently update it online.

In many applications, streaming data become available in small batches. There may also be the need to merge two or more sets of already-processed data. Thus, given a time series and its correspondent binary search tree, we consider how to integrate a new batch of data points into the tree without recomputing it from scratch. One could process the points of the newly available batch of data individually and include them in the existing tree structure by comparing both values and indices. However, other than being a time-consuming approach for large numbers of points, processing points individually fails to include useful information about both the new batch and the current tree structure. For instance, in the case of data coming in real time, e.g., financial time series or meteorological data, the newly available data points will all come later in time than the data points we already had stored. Consequently, all the corresponding nodes of the new batch to be added will have larger indices than the nodes of the current tree structure, and in particular larger indices than the current root. This means that all nodes of the new batch will populate only the right subtree of the current root. If the nodes were treated individually, this valuable information would be overlooked, resulting in an inefficient algorithm.

We propose to compute the binary search tree of the new nodes and to subsequently *merge* it with the previous tree structure. In this way, if all the new nodes indices are larger than the current root, one can use this information to substantially reduce the number of comparisons to be performed, and could potentially merge all the new data performing just one comparison. The proposed merge approach covers both

Algorithm 2. Pseudocode of the proposed algorithm to merge two binary trees defined by their root (class *Node*). The input is a list of roots to be merged.

```
def merge(input:{Node}):

    if input is empty: return null

    r    ← min_index(maxima_value(input))
    pool ← input \ {r}

    pool.append(r.left, r.right)

    for n in input \ {r} :

        for c in [n.left, n.right] :

            if sign(n.index - r.index)
                ≠ sign(c.index - r.index):

                pool.append(c)
                n.remove(c)

    return Node(
        index = r.index,
        value = r.value,
        left  =
          merge({p | p ∈ pool, p.index < r.index }),
        right =
          merge({p | p ∈ pool, p.index > r.index }))
```

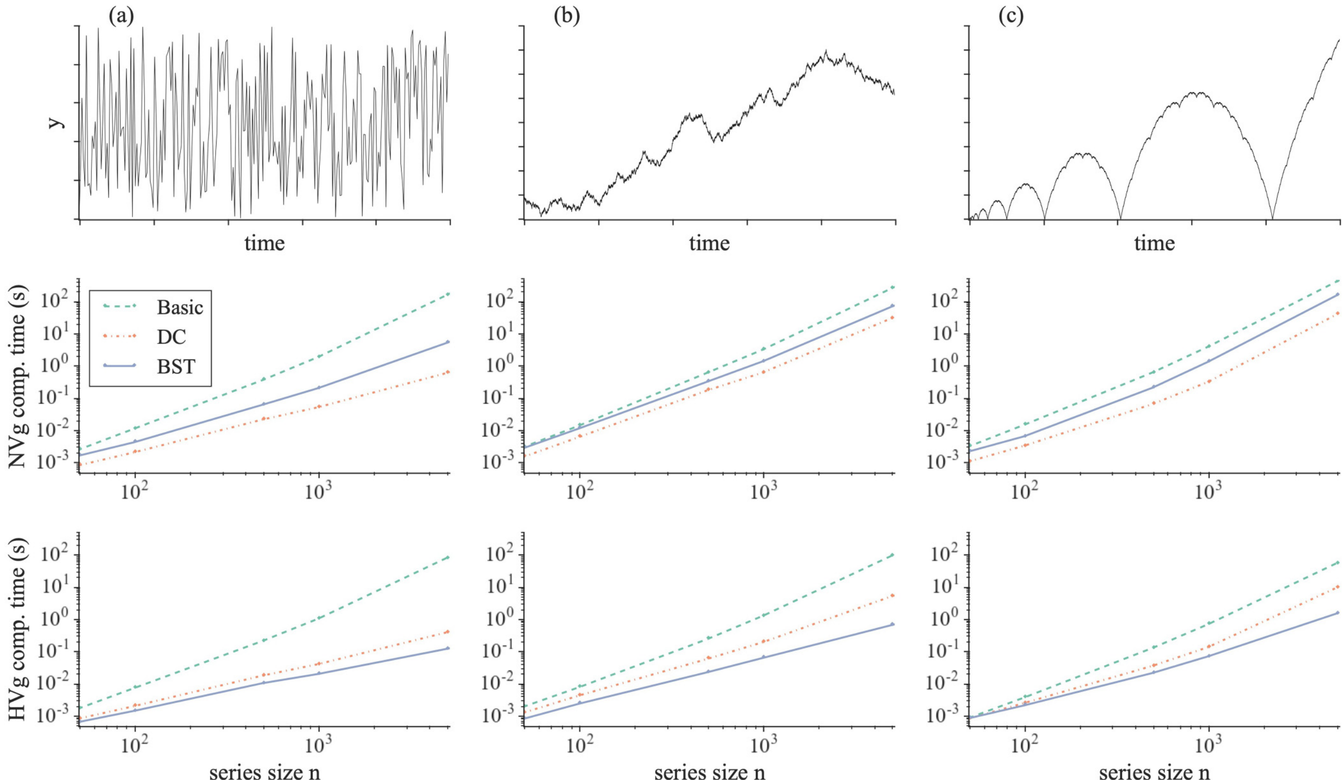

FIG. 4. Computation time of the natural and horizontal visibility graph (NVg, second row; HVg, third row) of different time series [examples on first row of a random (a), random walk (b), and Conway (c) series] using the current visibility algorithms: Basic, Divide and Conquer (DC), and the proposed Binary Search Tree (BST) method. Each point at every series size is the mean of the computation time for 10 series of that size.

append and insert operations, i.e., either the addition of points coming later in time or the assimilation of newly available information at a higher temporal resolution.

In order to merge two trees, we compare them recursively level by level, increasing depth at every step. The proposed *merge* function is outlined in Algorithm 2, and takes as input the list of trees (class *Node*) to be merged. The comparison happens in two steps: first, the node values at the currently considered levels are compared to determine which node will occupy that location in the resulting tree; second, the node indices are compared to determine on which side the remaining nodes will be placed.

Following the construction of the proposed binary search tree, the node with a larger value will be chosen and the rest of the nodes will travel left if their indices are smaller than the chosen one and right otherwise. The nodes to be compared are the children of the chosen node and the not-yet-chosen nodes from previous levels, starting by comparing the two roots of the trees to be merged.

Usually, the children of the nodes that travel down are ignored during comparison. However, when new data are to be inserted into the existing series, the child of the node traveling down may have an index corresponding to the other branch of the resulting tree. In this case, the connection between the node and that child will be broken thereafter. For further detail please refer to the code freely available online [29].

VI. NUMERICAL EXPERIMENTS

In this section we present empirical results in order to show how the proposed visibility algorithm compares to the state of the art in synthetic and real-world time series. All the code used to run the following experiments is implemented in Python 2.7 and freely available online [29]. The machine used in the simulations is an early 2015 MacBook Pro Retina with a 2.9 GHz Intel Core i5 processor and 16 GB of RAM.

To put the presented algorithm into context, in Fig. 4 we report the computation time needed by current visibility algorithms on different synthetic time series of increasing length. Since the actual efficiency of each algorithm depends to some extent on the nature of the original time series, we chose to use uniform random noise [which has no structure and on average produces almost-balanced binary search trees; Fig. 4(a)], a Conway series [which has a quite rich structure and corresponds to a quite unbalanced tree; Fig. 4(c)], and a random walk series [which represents the more realistic scenario of a signal with both structure and noise; Fig. 4(b)].

Following Ref. [27], we define a recursively generated Conway series of size n as $a(t) - \frac{t}{2}$, where

$$a(t) = a[a(t-1)] + a[t - a(t-1)] \quad \forall t \in [2, n] \qquad (1)$$

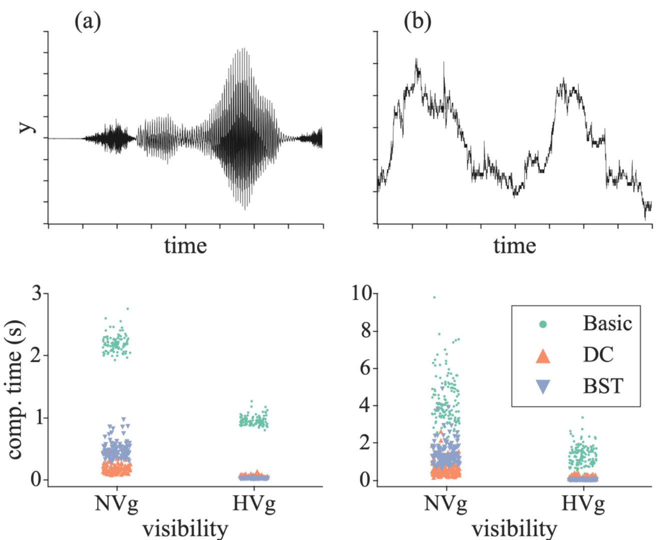

FIG. 5. Current and proposed visibility algorithms computation time for 100 speech (a) and finance (b) time series of 1000 points. The speech time series are sampled from the training TIMIT data set [30]. The finance time series corresponds to the 2013 quarterly data used in Ref. [8].

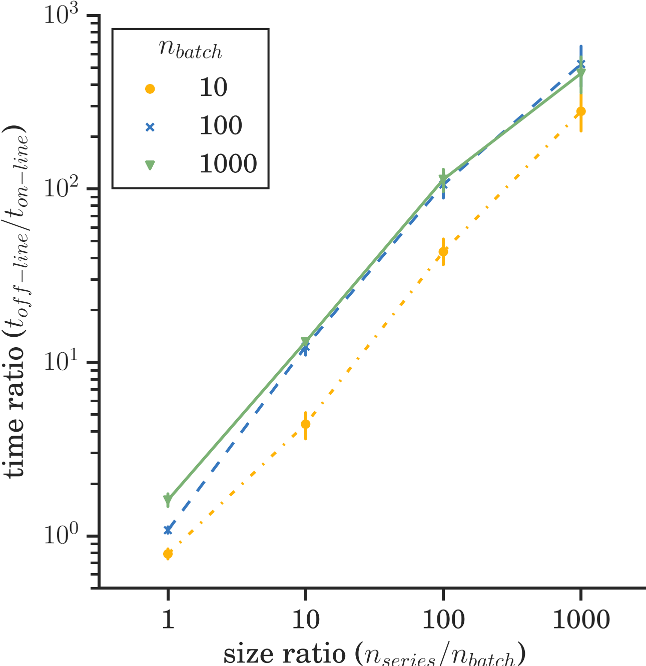

FIG. 6. Given a random time series (size n_{series}) and a batch of new random points (size n_{batch}) to be added to it, the advantage, in terms of computation time, of the proposed online approach versus the offline alternative. For the proposed method we measure the time it takes to build the maxima tree of the new points and to merge it with the existing tree (i.e. t_{online}). For the offline alternative computation we measure the time it takes to build the whole new maxima binary tree from scratch including the new points to the time series (i.e. t_{offline}). The time ratio is the logarithm of $t_{\text{offline}}/t_{\text{online}}$ and indicates how much quicker the proposed method is. The size ratio is $n_{\text{series}}/n_{\text{batch}}$, i.e., how much longer the existing time series is compared to the batch to be added. Both append and insert scenarios are represented here, with 10 random series for each case in every size configuration. Each point of the graph is therefore the mean of the combined 20 samples, while the error bars indicate the associated standard deviation.

and $a(1) = a(2) = 1$. The random walk time series $w(t)$ is generated by the discrete map

$$w(t) = w(t-1) + \epsilon, \tag{2}$$

where ϵ is a Bernoulli distributed variable such that $P(\epsilon = 1) = P(\epsilon = -1) = \frac{1}{2}$.

In the case of uniform random noise we observe the largest gap in computation time between the basic algorithm and the more efficient ones. Indeed, this case corresponds to the average case where both algorithms (DC and the proposed one) significantly reduce the number of operations with respect to the basic algorithm. Notice that these differences are more pronounced in the computation of the horizontal visibility graph, where the proposed algorithm based on the maxima binary search tree outperforms the other ones.

In Fig. 5 we present the results for two real-world data sets: samples of speech [English language; Fig. 5(a)] [30] and prices of stocks in a financial market [Fig. 5(b)] [8]. We sampled 100 time series (first 1000 points) from the TIMIT acoustic-phonetic continuous speech corpus (630 American English speakers reading 10 phonetically rich sentences recorded at 16 kHz) [30], as well as 100 time series from the daily prices of U.S. stocks traded in 2013 used in Ref. [8]. Figure 5 is particularly interesting as it clearly shows a correlation between time computation and the time series structure (please note the different scale for time computation). Even though the time computation may differ, the computation time for both DC and the proposed method seem to vary very little between data types, in stark difference with the relatively high spread observed for the basic algorithm.

The computation time for horizontal visibility remains stable in both DC and the proposed method, and could potentially be considered independent of the data type, up to a scaling factor. This behavior was expected as the proposed method is fully defined by the aforementioned connectivity rules and has average-case time complexity $O(n \log n)$. On the other hand, Fig. 5 suggests that the efficiency of the computation of natural visibility graphs is subject to wider fluctuations. The position of the maximum in the time series affects the efficiency of both the DC and the proposed method, as it will determine the number of additional visibility checks needed to obtain the natural visibility graph.

The English speech time series considered here will typically have its maximum somewhere towards the middle section of the signal (since we rarely tend to raise our voice at the end of our speech). Therefore the proposed method will most probably produce an almost balanced binary search tree for the speech time series, yielding a time complexity of $O(n \log n)$. For this reason, one may observe a wider gap in computation time between the basic method and the faster alternatives for the speech data in Fig. 5(a) than for the financial time series in Fig. 5(b).

In terms of computation time, the proposed method and the DC one are closely related. They are both quicker than the basic implementation in both natural and horizontal visibility,

and they both present similar trends for increasing time series size (Fig. 4). However, the proposed algorithm has proven to consistently be the quickest option for horizontal visibility graph computation. On the other hand, the DC algorithm in general does perform better than the proposed method for natural visibility computation. Even though at this point both DC and the proposed method seem equally good options for fast visibility computation, the presented algorithm has the additional property of allowing online assimilation of new data, which is something not easily achievable in either the DC or basic approach.

The most straightforward way to asses the online functionality of the proposed method is to compare it with the equivalent offline approach. In our case, it directly relates to the search binary tree construction. Given a batch of n_{batch} new points to be added to the n_{series} long time series, in the offline visibility analysis approach, the new batch is simply added to the time series itself, and then the search binary tree must be recomputed from scratch. In the proposed online approach, the next batch is encoded into its own binary tree that is then merged to the existing one using the procedure detailed in Algorithm 2. Note that the decoding step remains the same for the online and offline approach, and so the comparison will essentially be between computing a binary search tree from scratch (offline) and merging two trees into a single binary search tree (online).

Figure 6 shows how much quicker the computation of the online method (time $t_{\text{on-line}}$ to build binary tree of new data and merge to existing tree) is in comparison to the computation time of the offline approach (time t_{offline} to build binary tree from scratch), for different time series and batch sizes. In particular, the online approach is always better if the new batch to be added is equal or longer than the existing time series, especially for large time series.

VII. CONCLUSION

The proposed visibility algorithm based on an encoder/decoder approach is, to the authors' knowledge, the first efficient online algorithm to compute visibility graphs. The analysis and the numerical experiments shown in the paper confirm that the proposed algorithm represents a substantial improvement over the state-of-the art for horizontal visibility computation, and is on par with the most efficient natural visibility algorithm (i.e., DC) available. Moreover, the procedure to assimilate new data by means of merging the corresponding binary search tree encoding into the existing tree allows for efficient online computation of visibility graphs and represents a substantial speed-up with respect to the existing offline algorithms. This novel online capability broadens the applications for visibility graphs at no additional computational cost.

[1] R. V. Donner, M. M. Small, J. F. Donges, N. Marwan, Y. Zou, R. Xiang, and J. Kurths, Recurrence-based time series analysis by means of complex network methods, Intl. J. Bifurcation Chaos **21**, 1019 (2011).

[2] A. M. Nuñez, L. Lacasa, J. P. Gomez, and B. Luque, Visibility algorithms: A short review, *New Frontiers in Graph Theory*, edited by Y. Zhang (IntechOpen, 2012), Chap. 6, p. 120.

[3] T. Tanizawa, T. Nakamura, F. Taya, and M. Small, Constructing directed networks from multivariate time series using linear modeling technique, Physica A **512**, 437 (2018).

[4] A. H. Shirazi, G. R. Jafari, J. Davoudi, J. Peinke, M. R. R. Tabar, and M. Sahimi, Mapping stochastic processes onto complex networks, J. Stat. Mech.: Theory Exp. (2009) P07046.

[5] L. Lacasa and R. Toral, Description of stochastic and chaotic series using visibility graphs, Phys. Rev. E **82**, 036120 (2010).

[6] P. Fiedor, Networks in financial markets based on the mutual information rate, Phys. Rev. E **89**, 052801 (2014).

[7] L. Lacasa, V. Nicosia, and V. Latora, Network structure of multivariate time series, Sci. Rep. **5**, 15508 (2015).

[8] N. Musumeci, V. Nicosia, T. Aste, T. D. Matteo, and V. Latora, The multiplex dependency structure of financial markets, Complexity **2017**, 9586064 (2017).

[9] I. G. Torre, B. Luque, L. Lacasa, J. Luque, and A. Hernández-Fernández, Emergence of linguistic laws in human voice, Sci. Rep. **7**, 43862 (2017).

[10] E. Bullmore and O. Sporns, Complex brain networks: Graph theoretical analysis of structural and functional systems, Nat. Rev. Neurosci. **10**, 186 (2009).

[11] E. Bullmore and O. Sporns, The economy of brain network organization, Nat. Rev. Neurosci. **13**, 336 (2012).

[12] R. Mantegna, Hierarchical structure in financial markets, Eur. Phys. J. B **11**, 193 (1999).

[13] G. Bonanno, F. Lillo, and R. Mantegna, High-frequency cross-correlation in a set of stocks, Quantitative Finance **1**, 96 (2001).

[14] Y. Yang and H. Yang, Complex network-based time series analysis, Physica A **387**, 1381 (2008).

[15] R. V. Donner, Y. Zou, J. F. Donges, N. Marwan, and J. Kurths, Recurrence networks: A novel paradigm for nonlinear time series analysis, New J. Phys. **12**, 033025 (2010).

[16] J. F. Donges, R. V. Donner, and J. Kurths, Testing time series irreversibility using complex network methods, Europhys. Lett. **102**, 10004 (2013).

[17] J. H. Feldhoff, R. V. Donner, J. F. Donges, N. Marwan, and J. Kurths, Geometric signature of complex synchronisation scenarios, Europhys. Lett. **102**, 30007 (2013).

[18] D. Marinazzo, M. Pellicoro, and S. Stramaglia, Kernel Method for Nonlinear Granger Causality, Phys. Rev. Lett. **100**, 144103 (2008).

[19] W. Liao, J. Ding, D. Marinazzo, Q. Xu, Z. Wang, C. Yuan, Z. Zhang, G. Lu, and H. Chen, Small-world directed networks in the human brain: Multivariate Granger causality analysis of resting-state FMRI, NeuroImage **54**, 2683 (2011).

[20] L. Lacasa, B. Luque, F. Ballesteros, J. Luque, and J. C. Nuno, From time series to complex networks: The visibility graph, Proc. Natl. Acad. Sci. USA **105**, 4972 (2008).

[21] B. Luque, L. Lacasa, F. Ballesteros, and J. Luque, Horizontal visibility graphs: Exact results for random time series, Phys. Rev. E **80**, 046103 (2009).

[22] L. Lacasa and J. Iacovacci, Visibility graphs of random scalar fields and spatial data, Phys. Rev. E **96**, 012318 (2017).

[23] J. Iacovacci and L. Lacasa, Visibility graphs for image processing, IEEE Trans. Pattern Anal. Mach. Intell. **42**, 974 (2020).

[24] L. Lacasa, B. Luque, I. Gómez, and O. Miramontes, On a dynamical approach to some prime number sequences, Entropy **20**, 131 (2018).

[25] R. Flanagan and L. Lacasa, Irreversibility of financial time series: A graph-theoretical approach, Phys. Lett. A **380**, 1689 (2016).

[26] S. Sannino, S. Stramaglia, L. Lacasa, and D. Marinazzo, Visibility graphs for FMRI data: Multiplex temporal graphs and their modulations across resting-state networks, Netw. Neurosci. **1**, 208 (2017).

[27] X. Lan, H. Mo, S. Chen, Q. Liu, and Y. Deng, Fast transformation from time series to visibility graphs, Chaos **25**, 083105 (2015).

[28] The original Fortran 90 implementations of basic algorithms to construct visibility graphs can be found at http://www.maths.qmul.ac.uk/~lacasa/Software.html.

[29] Available at https://github.com/delialia/bst.

[30] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett, TIMIT Acoustic-Phonetic Continuous Speech Corpus, LDC93S1, Web Download, Linguistic Data Consortium, Philadelphia, 1993.