Physica A 514 (2019) 189–202

Contents lists available at ScienceDirect

Physica A

journal homepage: www.elsevier.com/locate/physa

An efficient non-recursive algorithm for transforming time series to visibility graph

Saptorshi Ghosh [a], Amlan Dutta [b,*]

[a] Department of Physics, Indian Institute of Science Education and Research, Bhopal 462066, India
[b] Department of Metallurgical and Materials Engineering, Indian Institute of Technology Kharagpur, 721302, India

HIGHLIGHTS

- A novel sort-and-conquer algorithm is proposed to transform a time series into visibility graph.
- Being non-recursive, this algorithm does not use stack frames in memory.
- Speed of execution is either comparable or better than the existing divide-and-conquer algorithm.
- Memory efficiency is significantly superior to the recursive method.

ARTICLE INFO

Article history:
Received 5 June 2018
Received in revised form 3 September 2018
Available online 22 September 2018

Keywords:
Time series analysis
Complex network
Visibility graph algorithm
Iteration and recursion

ABSTRACT

In recent years, transforming a time series into visibility network has emerged as a powerful tool of data analysis, with applications in many pure and applied domains of statistical physics and non-linear dynamics. The algorithms available for this transform are either very slow or consume copious amount of memory resorting to recursive calls. Here we propose an efficient non-recursive algorithm for constructing natural visibility graph from time series data. In comparison to the recursive method, the new algorithm offers safer and more optimized use of memory space without sacrificing its speed. Performance of this algorithm is tested with a variety of synthetic and experimental time series data-sets.

1. Introduction

Time series is one of the most common forms of data obtained from physical or computational experiments. Techniques of time series analysis have been immensely helpful in multiple domains including economics [1], geology [2,3], meteorology [4,5], material science [6,7], medical science [8,9], etc. The major objectives of time series analysis are twofold. At one hand, it explores the structure, features and patterns of the time series data itself using statistical techniques like regression methods, spectral analysis and stochastic modeling [10]. At the other hand, there is an aspect of system identification, which employs tools like principal component analysis [11], independent component analysis [12] and factor analysis [13], for investigating the underlying dynamics of a system. In the last few years, a novel scheme of time series analysis has gained wide recognition among the researchers. It involves transforming a time series into an equivalent complex network by means of some suitable algorithm. It has been observed that several important features of the time series are reflected in the statistical and topological features of the derived network. Moreover, a detailed analysis of this network can reveal some key aspects of the dynamics, which are difficult to identify by direct analysis of the time series data. The close duality between time series and network has led to the development of multiple transformation schemes. These schemes are based

* Corresponding author.
E-mail address: amlan.dutta@metal.iitkgp.ac.in (A. Dutta).

https://doi.org/10.1016/j.physa.2018.09.053
0378-4371/

on different mathematical structures like phase space [14], variance ratios of fluctuation pattern [15], recurrence plot [16], correlation matrix [17] and visibility link [18].

Out of these, the visibility graph algorithm developed by Lacasa et al. [18] has been the most widely used method for time series to network transform. On account of its simplicity and intuitive structure, it has found applications in many domains. For example, this method can be used to quantify the extent of synchronization in coupled dynamic systems [19]. Mutua et al. [20] have developed a method of analyzing chaotic time series by constructing chain of visibility graphlets. Recently, Rodríguez [21] has demonstrated how the Hurst exponent of a fractional time series depends upon the degree distribution of its visibility network. Beside such fundamental studies, time series to network transform is equally prolific in many applied domains as well. For instance, Wang et al. [22] and Zhuang et al. [23] have used the visibility graphs to analyze financial time series. Similarly, Elsner et al. [24] used this technique on the time series of hurricane counts. This method has also been employed to obtain a metric for identification of congestive heart failure from the heart rate data-set [25]. Interestingly, Lacasa and Iacovacci have recently demonstrated that it is possible to extend the fundamental idea of visibility link to scalar fields of higher dimension, and employ it for useful purpose like cancer detection through image analysis [26]. Apart from the natural visibility graph, the horizontal visibility graph, which is essentially a sub-graph of the natural graph, has also been used in the fields like nuclear physics [27], optics [28] and thermoacoustics [29]. Nevertheless, despite its ease of application, the original visibility graph algorithm suffers from a poor $O(N^3)$ time complexity, which can at best be reduced to a quadratic trend with some refactoring. In order to eliminate this bottleneck, Lan et al. [30] presented an elegant divide-and-conquer (DC) algorithm, which removes all the deterministic and unnecessary iterations. Although this method can offer a much improved performance of $O(N \log N)$ under favorable conditions, like most of the typical DC algorithms, its core functionality makes extensive use of recursive calls. For a well balanced time series, the DC algorithm has a recursion depth of the order of $\sim\log_2 N$. However, this scenario changes in the case of a highly unbalanced graph, where the recursion depth approaches $\sim N$. As a result, the implementation runs the risk of causing stack overflow even for a moderately large data-set. Moreover, a programming language or software tool may also place an upper limit on the maximum number of allowed recursion calls. These aspects render the DC implementation prohibitively difficult for transforming unbalanced time series with limited computational resources.

In this paper, we present a novel sort-and-conquer (SC) approach for converting a time series into visibility graph. This algorithm is entirely iterative in nature, which eliminates the problems related to recursion stack. We demonstrate that for a balanced time series data, the SC algorithm outperforms its DC counterpart in terms of the speed of execution, whereas for an unbalanced series, their speeds are almost comparable. We also show that due to its non-recursive nature, the memory consumption of the SC method may be orders of magnitude smaller than that of the recursive algorithm. The rest of the paper is organized as follows. In Section 2, we provide a brief overview of the existing algorithms for constructing visibility graphs. Section 3 presents the outline and working principle of the proposed non-recursive SC algorithm. Section 4 specifies the details of implementation and features of the benchmarking time series data-sets. The comparative performances of DC and SC algorithms in the contexts of execution speed and memory consumption have been analyzed in Section 5. Section 6 presents the concluding remarks.

2. Visibility graphs

A time series data $D = \{d_i\}_{i=1}^{N}$ is a set of N data elements (d_i), where each element is an ordered pair of two numbers expressed as, $d_i \equiv (t_i, x_i)$. According to the concept of visibility graph algorithm, the time series D can be transformed into an equivalent graph, G_N, containing N number of vertices such that each data point in D maps to a single and unique vertex in G_N. If we express the existence of an edge between vertices i and j as a Boolean (True/False) function $E(i, j)$, the natural visibility algorithm [18] proposes the following form for an undirected visibility graph,

$$E(i,j) = \neg\exists d_k : x_k > x_i + (x_j - x_k)\frac{t_k - t_i}{t_j - t_i} \forall t_i < t_k < t_j. \tag{1}$$

This idea is graphically elucidated in Fig. 1, which schematically shows the magnitudes of a few data elements of a time series. Any two vertices, say i and j, are connected by an edge if the corresponding data amplitudes x_i and x_j can 'see' each other along a rectilinear path without any obstruction by an intermediate data element. From Eq. (1), it can be observed that if the data-set undergoes one or more of the following operations: $\{t_i\} = \{t_i + \tau\}$ (horizontal translation), $\{x_i\} = \{x_i + \chi\}$ (vertical translation), $\{t_i\} = \{\varepsilon t_i\}$ (horizontal scaling), $\{x_i\} = \{\gamma x_i\}$ (vertical scaling) and $\{x_i\} = \{x_i + \alpha t_i + \beta\}$ (linear superposition), the resulting network is still isomorphic to the original one [18].

A straightforward implementation of Eq. (1) yields a highly inefficient algorithm. This can be realized from the observation that for a graph with N vertices, evaluation of $E(i, j)$ involves $N(N-1)/2$ inspections. Furthermore, for each of these (i, j) pairs, there are at the most $(j - i - 1)$ intermediate data points which are needed to be checked. Consequently, the total number of elementary operations vary as $\sim n^3$, which diverges rapidly. However, one may notice that Eq. (1) can be restructured as,

$$E(i,j) = \neg\exists d_k : m_{ik} < m_{ij} \forall t_i < t_k < t_j, \tag{2}$$

where the slope of a rectilinear path is given by $m_{pq} = \frac{x_q - x_p}{t_q - t_p}$. This is essentially equivalent to,

$$E(i,j) = m_{ij} > \sup\{m_{ik} : t_i < t_k < t_j\}. \tag{3}$$

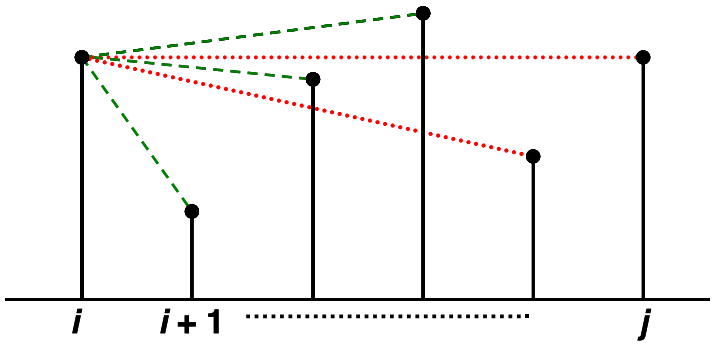

Fig. 1. Concept of visibility link. The green dashed lines represent the rectilinear visibility paths between data elements. The red dotted lines indicate those paths, which have been obstructed by intermediate data elements.

Therefore, as pointed out in Ref. [30], a better strategy is to keep track of the maximum slope obtained during iterative evaluation of visibility links for a given vertex. If the next slope is larger than the current maximum, the maximum slope is updated and a new edge is added to the network.

Although the $O(N^2)$ time complexity of this methods offers a substantial improvement over the original $O(N^3)$ approach, it still diverges rapidly and becomes unfeasible for large data-sets. This problem is solved by the divide-and-conquer algorithm [29], which is based on the fact that for any time series, the data element with the largest amplitude partitions the whole series into two disjoint parts, such that all the elements of one part are essentially opaque to all elements of the other (Fig. 2). This immediately rules out the possibilities of a large number of visibility edges. Thus, in the first step, the DC algorithm selects the largest data element and checks its visibility from all other data vertices. In the next step, the data elements on the left and right sides of this largest element are considered as two separate time series and the same operation is repeated for both of them in a recursive manner. For a well balanced data-set, each level of recursion-call splits the time series into two parts of approximately equal lengths. Under such circumstances, the total number of operations involved in the DC method may asymptotically approach the trend of $\sim N \log_2 N$.

As pointed out in the previous section, dependence on recursion is a major drawback of the DC algorithm. A few programming languages or some of their implementations have non-existent or poor support for recursive calls. However, the most significant side effect of recursion is the fact that each call of a recursive function occupies a frame in the execution stack [31]. Apart from the returning address to the calling function, the stack frame also holds function parameters and copies of temporary variables. Therefore, even if the original data-set occupies only moderate amount of memory space, the total memory allocation for a recursive program can be orders of magnitude larger than this. From a practical standpoint, this may also lead to severe bottleneck in terms of execution time if the available volatile memory space is entirely exhausted and any further allocation points towards the secondary storage with slow access. In addition, programming languages like Python and MATLAB usually fix the maximum number of allowed recursion-calls to prevent segmentation fault, which is often violated if the time series is intrinsically unbalanced. Even though the earlier $O(N^2)$ algorithm presents a poor time complexity, its memory usage is much lower than that of the DC technique. Clearly, we face a dilemma of space–time tradeoff, which may render the implementation of DC method difficult if the computational resources are limited or the data-set is large and unbalanced.

3. Sort-and-Conquer algorithm

Here we present a novel sort-and-conquer method for deriving the visibility network from a time series. It is iterative and hence, free from the problem of stack overflow. Moreover, the memory requirement is only limited by the size of the resulting data structure describing the network. The basic working principle underlying the SC method is as follows.

3.1. Working principle

Let us consider three data elements, d_i, d_j and d_k, of a time series under the following conditions:
(1) $t_i < t_j < t_k$ and $x_i < x_j < x_k$,
(2) $E(i, j) = True$ and $E(i, k) = True$,
(3) there does not exist any data element d_m such that, $t_j < t_m < t_k$, $x_m > x_j$ and $E(i, m) = True$.

If all the conditions given above are satisfied, then $E(i, p) = False\ \forall d_p : t_j < t_p < t_k$. The truth of this statement is easily established by contradiction. Let us assume that under the given conditions, there is a data element, d_p, between d_j and d_k (*i.e.*, $t_j < t_p < t_k$) for which $E(i, p) = True$. From condition (2), we can conclude that $x_p < x_k$, otherwise, it would have rendered $E(i, k) = False$ by blocking the visibility between vertices i and k. Now, since $x_p > x_k$, we have only two possibilities; $x_p \leq x_j$ or $x_j < x_p > x_k$. For $x_p \leq x_j$, $E(i, p) = False$ by virtue of Eq. (1). Therefore, for $E(i, p) = True$ to hold, the only remaining

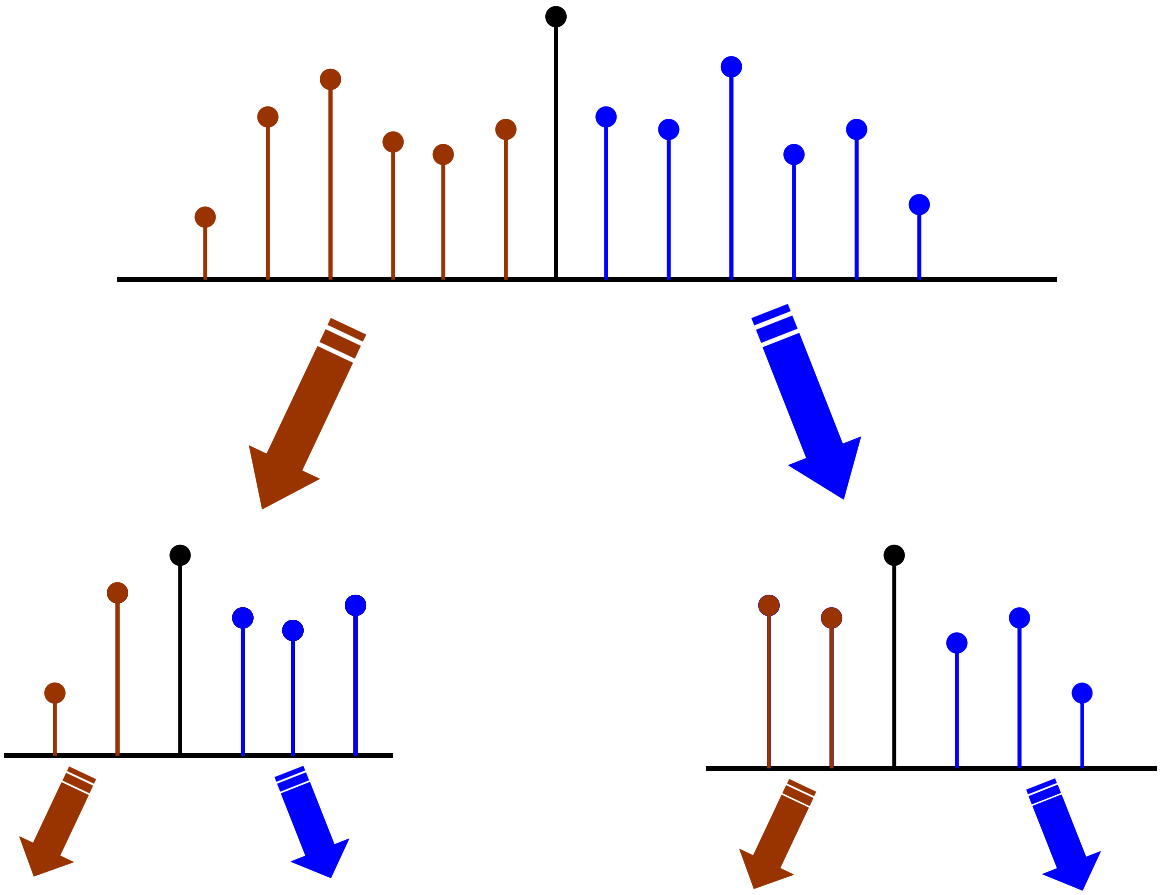

Fig. 2. Scheme of the recursive divide-and-conquer algorithm.

condition is $x_j < x_p < x_k$, but it already contradicts condition (3), which forbids the existence of such a data element. Thus, $E(i, p) = False$ for any arbitrary data element d_p between d_j and d_k. Similarly, if condition (1) is altered to,

(4) $t_i > t_j > t_k$ and $x_i < x_j < x_k$, and condition (3) to,

(5) there does not exist any data element d_m such that, $t_j > t_m > t_k$, $x_m > x_j$ and $E(i, m) = True$,

then a similar flow of logic implies that under conditions (4), (2) and (5), $E(i, p) = False\ \forall d_p: t_j > t_p > t_k$.

In essence, the condition-set (1, 2, 3) depicts a scenario, where the data elements d_i, d_j and d_k are positioned in ascending order of position (*i.e.*, time) and magnitude, and d_i is visibly linked to d_j and d_k. If an algorithm can be devised, which guarantees such a situation, it eliminates the need of examining the visibility links from d_i to any other data point between d_j and d_k. In the same way, the condition-set (4, 2, 5) implies a situation where the data elements d_i, d_j and d_k are positioned in descending order of position and time, with d_i linked to both d_j and d_k. As this configuration exhibits mutual exclusion and qualitative inversion with respect to the condition set (1, 2, 3), the symmetry of relational arguments suggests that d_i cannot be linked to a data element positioned between d_j and d_k. The SC algorithm makes an optimized and efficient use of this principal without resorting to recursive calls.

3.2. Scheme of implementation

The SC algorithm consists of two primary stages. The first stage involves sorting of the data elements according to their magnitudes (x_i values). In the next stage, we iterate over the data elements in the order of their descending sequence. Accordingly, we initiate the iterations with the largest element and the subsequent iterations consider the second largest element, followed by the third largest and so on. In each iteration, the current status of the visibility graph is read to obtain the list of vertices connected to the current vertex or data element under consideration. From this list, a pair of vertices are chosen such that they enclose the narrowest interval (in terms of t_i values) containing the current vertex. Once such an interval is obtained, the connectivities of the current data element are tested with only those elements, which lie within this local interval and the network is updated accordingly.

Fig. 3 provides a demonstration of how the implementation of the SC algorithm is expected to perform. The left panel of Fig. 3(a) shows a sample time series data with seven data elements. On the top of each data element, its amplitude-wise rank

is shown. Accordingly, we can see that d_6 and d_1 are the elements with the largest and smallest magnitudes, respectively. Now, this series is sorted in descending order of magnitude and the sorted array is shown in the right panel of Fig. 3(a). In this particular example, the magnitudes of the elements vary as, $d_6 > d_7 > d_2 > d_4 > d_3 > d_5 > d_1$.

Next, we enter the second phase of the method, which is displayed in Fig. 3(b). We start iterations over the vertices in the order given by the sorted data-set observed in Fig. 3(a). In the subsequent description, the particular data element over which an iteration is being carried out is referred to as the current element. As the time series has seven data elements in total, the visibility graph needs seven iterations for its construction.

Step 1 — Being largest in magnitude, d_6 is the current data element. As the visibility graph does not have any edge at this point of time, the local bracketing interval for d_6 is the entire data-set. Hence, the visibility of d_6 is tested from all the data elements of the time series. In this case, d_6 is found to be visible from d_2, d_4, d_5 and d_7. Accordingly, edges E (2, 6), E (4, 6), E (5, 6) and E (7, 6) are added to the graph.

Step 2 — As the next largest element is d_7, we look for the vertices to which it has already been connected. At this step, it is only connected to d_6 on its left, while there is no other element on its right. Furthermore, as d_6 is also adjacent to d_7, there is no bracketing interval and hence, there is no further addition of visibility link involving d_7.

Step 3—d_2 is the next largest element, and we check for all the edges connected to it. So far, it has only one connection, E (2, 6), which lies on its right, whereas there is no connection to its left. Thus, the bracketing interval for d_2 is from d_1 to d_5, *i.e.*, excluding the connected node-6. Visibilities of all the elements in this interval are inspected from d_2 and the graph is updated.

Step 4 — The fourth largest element is d_4. In the previous steps, it had been linked to d_6 on its right and d_2 on its left. Hence, the bracketing interval for node-4 comprises of all the elements between these connected nodes 2 and 6. In this specific example, d_4 is found to have only one link on each of its sides. Nevertheless, for a general case, an element may have multiple links on a side. In such a case, the connected element nearest to the current node must be considered for the purpose of determining the bracketing interval. Similarly, if there is no existing connection on any side of the current node, the limit of the local bracketing interval extends up to the extreme end of the data-set on that side. We have already witnessed such instances in steps 1 and 2 above. In the present demonstration, the bracketing interval comprises of d_3 and d_5 only. As both the elements are adjacent to d_4, visibility is established and E (3, 4) and E (5, 4) are added to the network.

Steps 5, 6, 7 — d_3 becomes the current node. As it is already linked to its neighbors d_2 and d_4, the bracketing interval is void and there is no need to test its visibility from any other element. In the same way, d_5 and d_1 are also saturated and no further inspection of visibility is necessary. Consequently, the visibility graph remains unchanged during these iterations.

Algorithm 1 presents the pseudocode for implementation of the SC method. The first phase of the algorithm, which involves magnitude-wise sorting of the data-set is accomplished in line 2 of the pseudocode. It generates an array '*s_index*' containing the indices of the data elements in a sequence which would sort them in descending order of magnitude. In the example described above, the array 's_index' is pointed out in the right panel of Fig. 3(a). Therefore, on the ith iteration of the outer loop (line 3), the variable '*cur_node*' contains the index of the ith largest data element. Sorting of the time series can be done in $O(N \log N)$ time with any suitable sorting method [32].

It is worthwhile to point out that line 5 of the pseudocode introduces a feedback mechanism, where the connectivities of the current node are retrieved from the visibility graph. It is a key aspect, which distinguishes the SC method from the other algorithms. This is because in all of the previous methods, edges are only added to the graph but the graph itself is never read back during the course of iterations. In contrast, the SC algorithm not only adds edges to the visibility network, but also retrieves and utilizes the updated topology in every iteration. The variables '*left*' and '*right*' initialized in the lines 6 and 7 are used to specify the lower and upper bounds of the local bracketing interval for the current data element. The inner loop (lines 8–14) ensures that the index number held by '*left*' points to the data element which is nearest and connected to the current element on its left side. Similarly, the variable '*right*' refers to the element which is closest and connected to the current data element on its right side.

At this juncture, it is interesting to observe the fundamental principle of the SC algorithm at work. Let us assume that at any particular iteration, we have a current data element with index i (see Fig. 4 for a schematic representation). r_1, r_2, r_3, … are the indices of elements which have already been found to be connected to d_i and are placed on its right side. r_1 represents the element which is closest to d_i. Then r_2 is the next closest element, followed by r_3, and so on. Similarly, l_1, l_2, l_3, … represent the connected elements on left side of d_i, with l_1 being closest to d_i. Since the outer loop of the algorithm iterates over the data elements in descending order of their magnitudes, we find for any triplet $(i, r_k, r_k{+}1)$, condition-set (1, 2, 3) are satisfied (*c.f.* Section 3.1) and hence, a visibility link can never exist between d_i and any other element in the interval $(r_k, r_k{+}1)$. In the same way, we can realize that condition-set (4, 2, 5) are valid for the any triplet $(i, l_k, l_k{+}1)$ and the only interval in which the visibilities are required to be tested is (l_1, r_1). The function of the inner loop in Algorithm 1 is to evaluate these limits l_1 and r_1. As the local bracketing interval should not include these vertices with which the visibility links have already been established, they are excluded from this interval in lines 15 and 16. Finally, line 17 invokes a module 'visibility', which checks for the visibilities of the current element from all other data elements within the bracketing interval and adds edges to the network accordingly. For the sake of convenience, pseudocode of this module is provided in the Appendix A.

Algorithm 1 Pseudocode of the sort-and-conquer algorithm.

Algorithm 1: The sort and conquer method

Input: Time series data-set $D\{t, x\}$, where t and x are arrays, each of length N. ($t[k]$, $x[k]$) represents the data element corresponding to index k.

Output: Visibility graph G.

```
1.  Initialize an undirected graph G with N vertices
2.  Sort the array x in descending order and store the corresponding indices in an
    array s_index
3.  for i = 1 to N:
4.  |  cur_node ← s_index[i]
5.  |     Array connected_node ← indices of data elements presently connected
    |  to the element cur_node in G
6.  |  left ← 1
7.  |  right ← N
8.  |  for each j in connected_node:
9.  |  |  if j < cur_node:
10. |  |     left ← sup{left, j}
11. |  |  else:
12. |  |     right ← inf{right, j}
13. |  |  end if
14. |  end for
15. |  left ← left + 1
16. |  right ← right − 1
17. |  G ← visibility(D, left, right, cur_node, G)
18. end for
```

4. Testing and validation

Time and memory complexities of visibility graph methods may vary significantly depending upon the structure of time series. Hence, it is necessary to test the efficiency and performance of such algorithm against a broad variety of data-sets. In the present study, we have used two major categories of time series data to investigate how much computational resources are consumed in terms of execution time and allocated memory. In the first category, we use a set of synthetic time series which are expected to impose different amounts of computational load. Under the second category, a few experimentally obtained time series belonging to different types of systems and dynamics are studied. Aiming to compare the relative performances of the DC and SC algorithms, both the approaches have been applied to produce visibility graphs from each time series. Brief descriptions of the benchmarking data-sets used in this study are given hereunder.

4.1. Synthetic time series

White noise—A white noise is simulated simply by generating random numbers, which are uniformly distributed over a given range. As the visibility network is invariant with respect to affine transformation of the time series, the exact values of the lower and upper bounds of the data-set becomes irrelevant. Lacasa et al. [18] showed that the degree distribution of a visibility graph constructed from such white noise exhibits an exponential tail due to the occasional presence of hubs, which are the nodes with very large number of visibility links. However, on account of uniform distribution of magnitudes along the time series, it is statistically stationary and well balanced from the perspective of the DC algorithm. Fig. 5(a) shows the typical waveform of a white noise.

Conway series—This series is given by $x(t) = a(t) - t/2$ for the positive integral values of t, where

$$a(t) = \left\{ \begin{array}{l} 1, \textit{for } t = 1, 2 \\ a(a(n-1)) + a(n - a(n-1)), \textit{ for } t > 2 \end{array} \right\}. \tag{4}$$

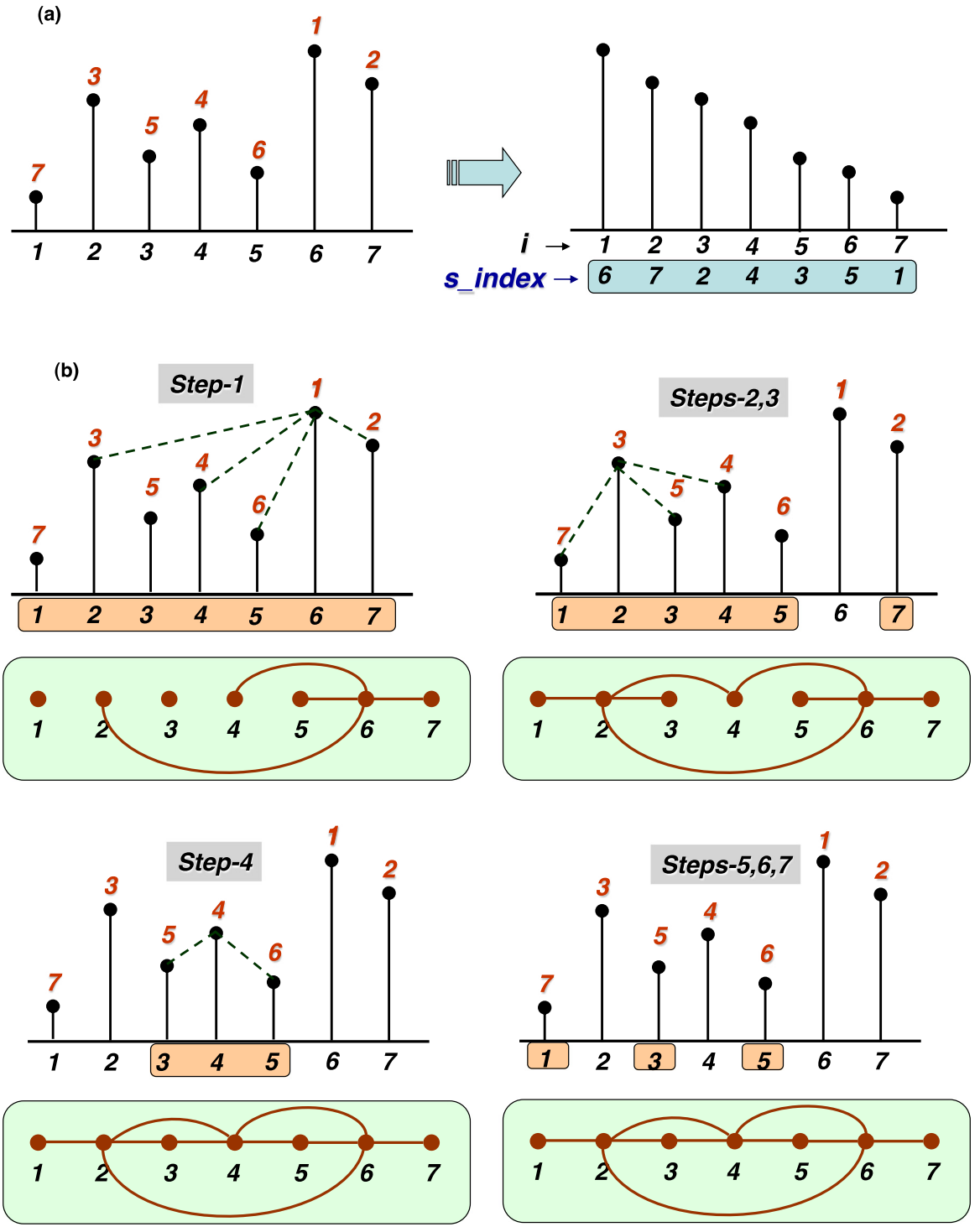

Fig. 3. Demonstration of the sort-and-conquer algorithm. (a) The sort-stage involves sorting of the time series in descending order of magnitude. (b) The conquer-stage determines a local bracketing interval (shown as highlighted indices) for each current node and adds visibility links to the network. The dynamically evolving network is also shown here.

This recursively generated sequence, which is shown in Fig. 5(b), exhibits a fractal structure. Moreover, $x(t) = 0$ whenever t is a positive integral power of 2. It is an unbalanced and non-stationary time series. When transformed to a visibility graph, the Conway series is found to produce scale-free network, which does not exhibit the small-world effect [18].

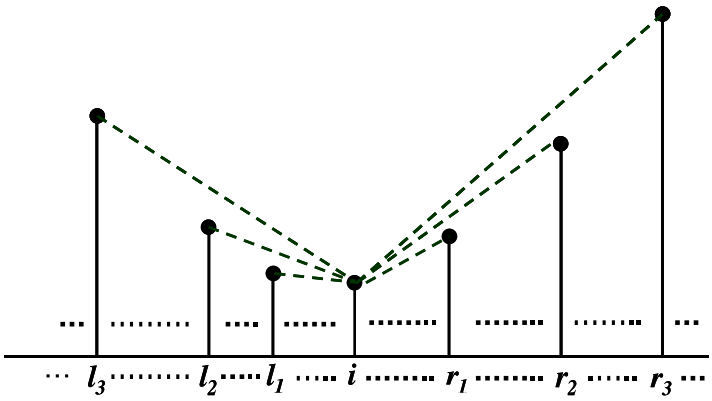

Fig. 4. Schematic representation showing the visibility edges of the ith data-element at the beginning of the ith iteration.

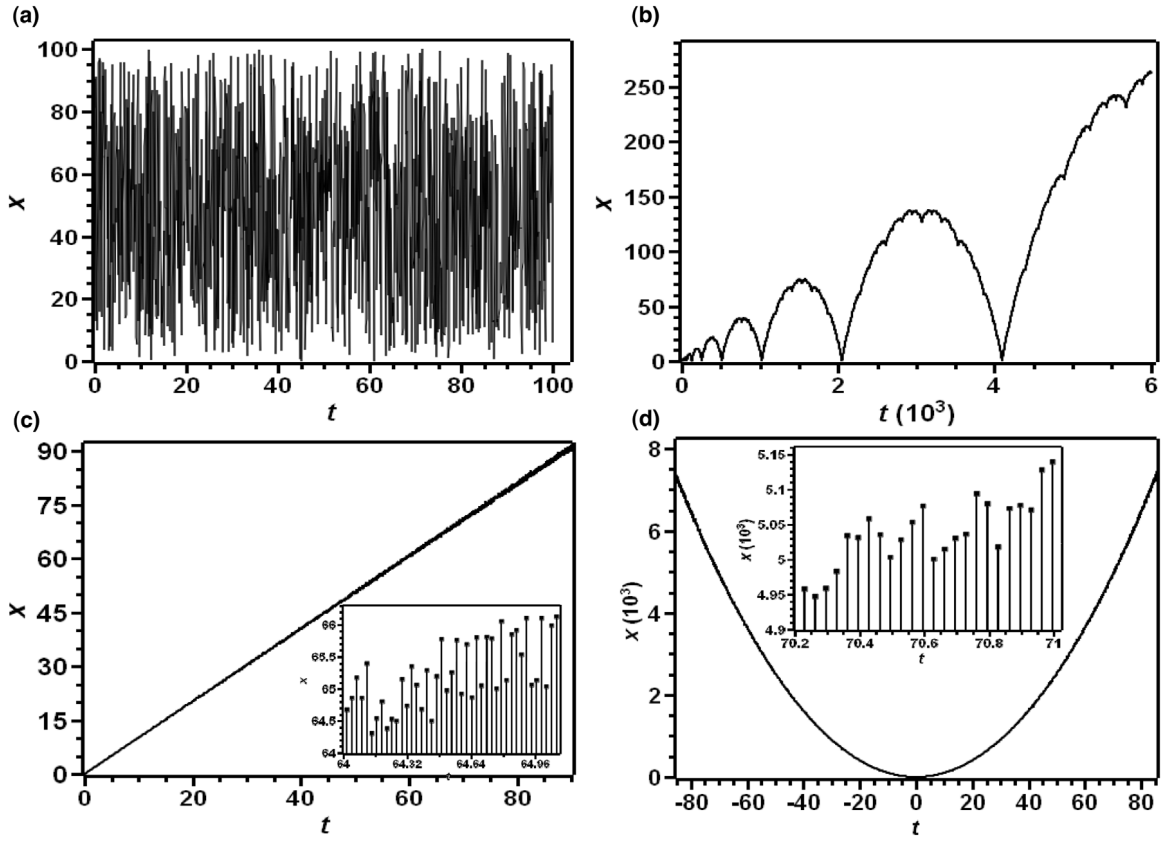

Fig. 5. Representative plots of the synthetic time series showing (a) white-noise, (b) Conway series, (c) noisy linear series and (d) noisy parabolic series.

Noisy linear series—This series is synthesized by superposing a weighted random noise on a linear trend. It is given by, $x(t) = t\,(1 + \eta \mathrm{U(t)})$, where U(t) is a random noise uniformly distributed in (0, 1) and η is the weight given to these fluctuations. Fig. 5(c) displays a noisy linear waveform with $\eta = 0.02$. We can observe that this series closely resembles the monotonically increasing time series (*c.f.* Fig. 4(a) in Ref. [30]) and therefore, is highly unbalanced. The inset of Fig. 5(c) provides a closer view of a small part of the time series and highlights the significance of the added fluctuations. We observe that the visibility is severely restricted on account of the superimposed noise on an otherwise linear trend. This significantly suppresses the mean degree of connectivity as well as the probability of a given data element to become a hub.

Noisy parabolic series—In this series (Fig. 5(d)), weighted fluctuations are added to a parabola, thereby making it similar to the unbalanced, bowl-like time series discussed by Lan et al. (*c.f.* Fig. 4(c) in Ref. [30]). The noisy parabolic data-set, which is

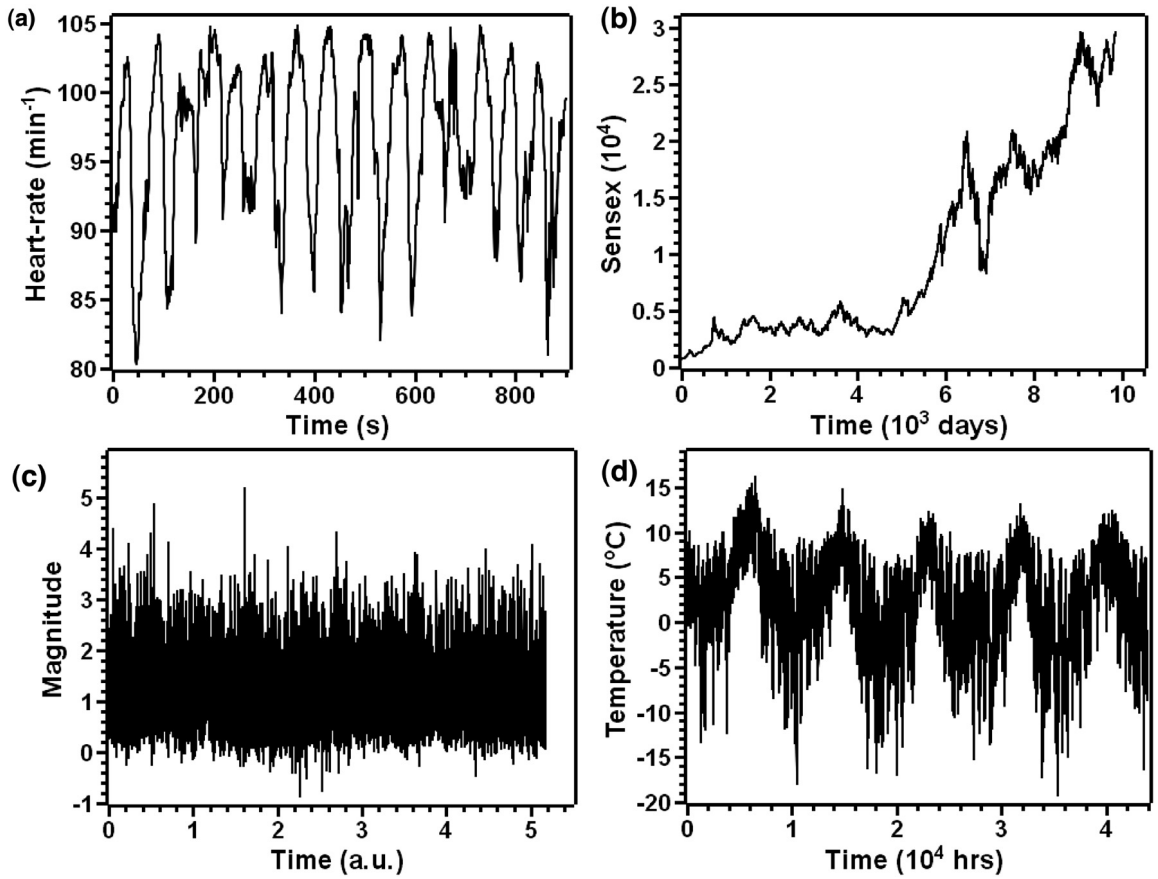

Fig. 6. Plots of the experimental time series showing (a) heart-rate, (b) BSE Sensex, (c) magnitude of seismic events and (d) dew point.

obtained as $x(t) = t^2(1 + \eta \mathrm{U(t)})$, is different from the noisy linear series in the context of computational load. This is because unlike the previous case, random fluctuations do not restrict the occurrence of hubs, for many elements on $t < 0$ can still see a large number of elements on $t > 0$. Apart from testing visibility, reading and modifying a graph structure also take finite amount of time. Hence, the noisy parabolic series is expected to be more computationally intensive than its linear counterpart.

4.2. Experimental time series

Heart-rate—Fig. 6(a) presents the heart-rate time series for a healthy subject [33]. For this data-set, measurements were taken at 0.5 s intervals and there are 1800 data elements in the series. This series is available at http://ecg.mit.edu/time-series in text format.

Stock exchange—Fig. 6(b) shows the thirty-component stock exchange index, commonly known as BSE-30 or S&P BSE SENSEX of the Bombay Stock Exchange, India. The figure displays 6487 daily closing values for all operational days from April 1, 1990 to March 31, 2017.

Seismic events—Fig. 6(c) displays the magnitudes of 20 783 earthquake events recorded from January 1, 2016 to May 31, 2017. This data-set, which has been obtained from the Southern California Earthquake Data Centre (http://scedc.caltech.edu), corresponds to a geographical region spanning from lat. 30° to 39° and long. -124° to −111°.

Dew point—This time series (Fig. 6(d)) consists of dew point measured at 1 h interval at the Afstapahraun weather station in Iceland. This data-set of size 43 925 covers a time-span from November 26, 2002 (18:00 h.) to January 11, 2008 (20:00 h). The raw data is available at https://datamarket.com/.

All the computational results reported for these benchmarking tests have been obtained by implementing the DC and SC algorithms on a desktop computer with Intel Core i5-4570 CPU clocked at 3.2 GHz and 4 GB RAM. Programming and execution is done with MATLAB under the Ubuntu (14.04 LTS) operating system. A visibility graph is stored as an adjacency matrix represented in the sparse format of MATLAB. As a precautionary measure, MATLAB allows a maximum recursion depth of 500 only. Hence, this safety setting is deliberately overridden while benchmarking with large unbalanced data-sets.

5. Results and discussion

Fig. 7 presents the execution time of the DC and SC algorithms for all the four synthetic data-sets described in Section 4.1. For each type of time series and a given data size, a set of ten runs are performed and their mean result is shown in the plot. We clearly observe that the trend of execution time differs from one type of series to another. Being approximately balanced, the white noise is capable of being transformed to a visibility graph at a much faster pace as compared to the other time series studied here. Fig. 7(a) further reveals that the non-recursive method is faster than the recursive DC algorithm for such balanced series. In particular, this difference gradually grows with the increment in data size. A detailed profiling reveals that the largest fraction of the total execution time is actually consumed by the process of adding edges to the network by updating the sparse adjacency matrix in each iteration. However, as the number of times this operation is performed is exactly same in DC and SC methods, it suggests that the comparatively slower performance of the DC algorithm is likely due to the overhead caused by repeated access of the stack frames. Section S1 of the supplementary document offers a comparison of performances of the two algorithms for even larger data size and a detailed discussion of the time complexities.

For the Conway series, the execution time *vs*. data size plots follow a peculiar trend, which consists of alternate plateau-like regions and steep upturns (Fig. 7(b)). The structural pattern of the Conway series underlies this interesting effect. Fig. 5(b) shows that this time series is made of a sequence of fractal humps with each hump being larger than the previous one. Moreover, as the zeros are located at integral powers of 2, the widths of the humps also increase as 2, 4, 8, 16, etc. Structure of the Conway series suggests that the data elements at the upper part of the humps have much lower degrees of connectivity due to restricted visibility. On the other hand, most of the hubs are located near the zeros as elements on the left side of a zero have wide visibility of elements on its right. As a result, whenever the data size increases to include a new zero, a steep upturn in the execution time is noticed as the program has to spend increasingly more time on adding a large number of new links to the visibility graph. In contrast, when the series terminates at the upper part of a hump, the execution time increases slowly and produces a plateau-like appearance. Unlike the balanced white noise, most of the time is consumed in testing for the visibility, while only a small fraction is used for adding edges. This explains why the performances of the two algorithms are almost similar, though SC slightly outperforms DC at larger data size.

Fig. 7(c), which corresponds to the noisy linear time series, is an extreme example of the unbalanced time series. Here the visibility is often tested over wide bracketing intervals, whereas the average number of edges per node is very small due to restricted visibilities. Consequently, the speeds of execution are almost indistinguishable for the two algorithms over the entire range of data size explored here. It is interesting to note that while the monotonic and bowl-like series are equally unbalanced from the perspective of the recursive algorithm [30], we clearly find that the noisy parabolic data exhibits a noticeably superior performance for the SC method as evident from Fig. 7(d). This is because unlike the noisy linear series (Fig. 7(c)), the noisy parabolic series has a large number of hubs due to enhanced visibility, which leads to addition of numerous edges in most of the iterations. As the addition of an edge is already found to be the slowest of all the elementary operations, the difference between the speeds of SC and DC method becomes apparent in Fig. 7(d). Here it is worthwhile to point out that for a well balanced time series, the DC and SC methods offer the advantage of $O(N \log_2 N)$. However, for a highly unbalanced data-set, the DC method has to undergo the same number of operations as the $O(N^2)$ algorithm (*c.f.* Eq. (3)). In the case of SC algorithm, the number of mathematical operations is almost same as that of the DC method for an extremely unbalanced time series (e.g., Fig. 7(c)), which makes the SC algorithm follow the $\sim N^2$ complexity as well. In practice, we may expect a data-set of interest to be neither perfectly balanced, nor completely unbalanced and hence, the execution time would be somewhere between the two extreme complexities of $O(N^2)$ and $O(N \log_2 N)$.

Having observed the comparative performances in terms of execution time, it is pertinent to analyze the memory consumption of the two algorithms. As memory profiling is a considerably slower process than time profiling, we have estimated this metric for smaller lengths of time series than those used in Fig. 7. Fig. 8 shows the memory allocated to the program for different types of time series data-sets. As expected, the recursive DC algorithm has a much larger memory requirement as compared to the non-recursive SC implementation. While the memory consumption of SC algorithm is primarily dictated by the number of visibility links present in the network, the memory requirement of the DC method also depends upon the maximum recursion depth for a given time series. This depth varies as $\sim \log_2 N$ for a well balanced series, but may elevate to $\sim N$ for a highly unbalanced data-set. This is why, memory allocated to the recursive program is typically smaller for the white noise than that for the unbalanced data-sets. Even out of the three unbalanced time series, the memory requirement tends to be lowest for the noisy linear data and highest for the noisy parabolic series. This is due to much fewer visibility links present in the former case in contrast to the latter data-set, where the graph is densely populated with edges. Furthermore, the analysis of memory requirement also explains why the range of data-size studied for the unbalanced time series is much smaller than that for the balanced white noise in Fig. 7, As we increase the number of data elements of an unbalanced time series, the memory consumption of the DC algorithm diverges rapidly and begins to use the secondary storage as part of the virtual memory. This causes an abrupt increase in read/write delay and it becomes inappropriate to benchmark its speed performance on the same footing as that for a smaller series which only accesses the significantly faster volatile memory.

Finally, we investigate the relative efficiencies of the algorithms in constructing visibility network from the experimental data-sets detailed in Section 4.2. This is presented in Table 1, which shows the execution time and memory allocation for the four time series under consideration. We find that for all the four data-sets, the execution time of the SC method is either same or somewhat smaller than the DC implementation. Moreover, the memory usage of DC method is two or three orders

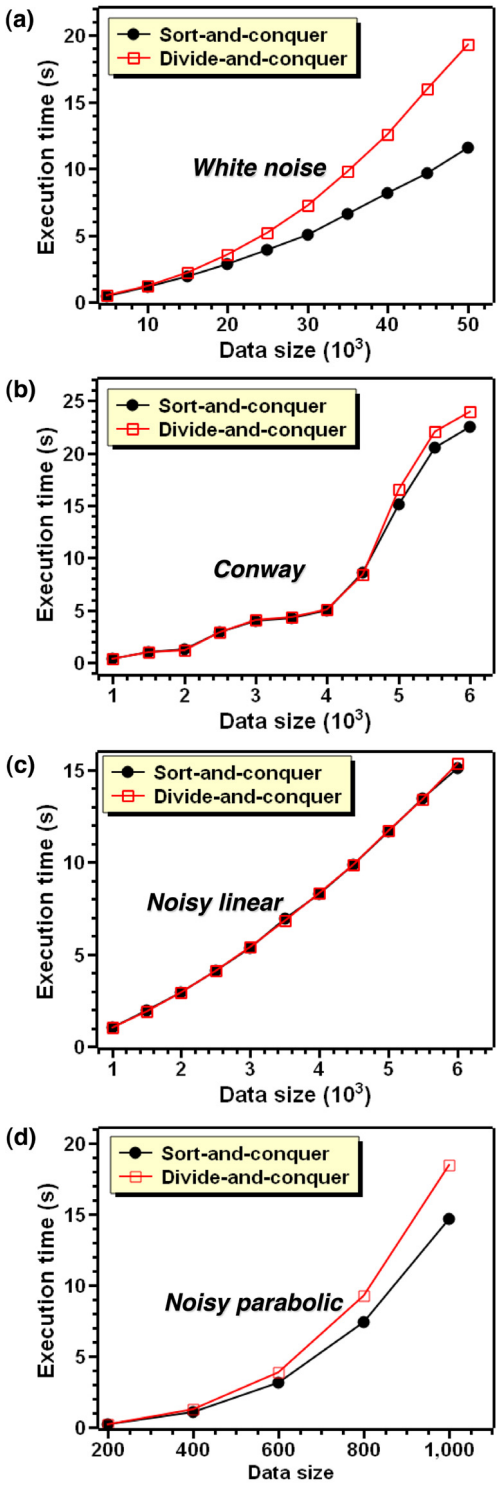

Fig. 7. Variations of execution time plotted as function of data size for the (a) white-noise, (b) Conway series, (c) noisy linear series and (d) noisy parabolic series. Filled circles and open rectangular symbols correspond to the SC and DC implementations, respectively.

of magnitude larger than that of its non-recursive counterpart. The table also presents the expected times of execution for the white noise time series of comparable lengths. We find that for each case, the experimental data-set takes longer than the white noise series of same length. We note that even in case of the seismic time series, the execution durations are about 23% and 37% larger for the DC and SC implementations, respectively, in comparison to the white noise. This shows

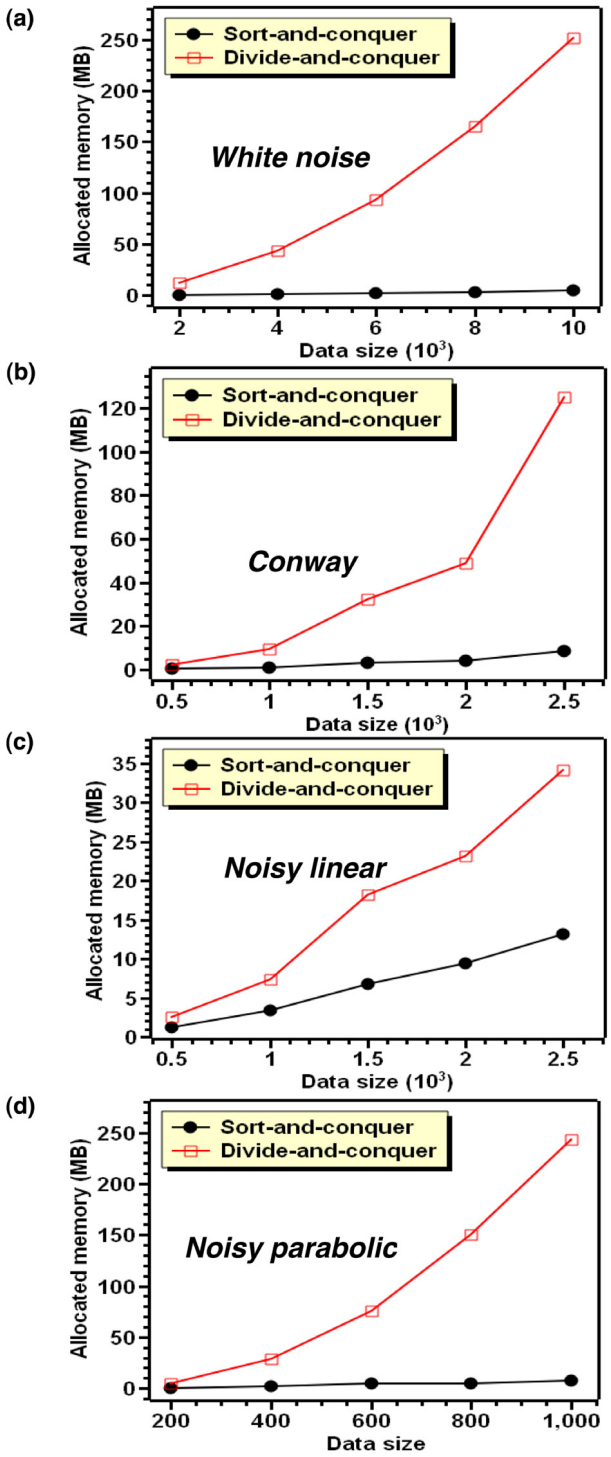

Fig. 8. Plots of allocated memory vs. data size for the (a) heart-rate, (b) BSE Sensex, (c) seismic events and (d) dew point time series.

that there is a degree of imbalance in this time series, even though it visually appears to be akin to a white random series. The supplementary material presents a similar analysis for the time-series obtained from a simple Hénon-map. It shows that for different dynamical parameters corresponding to the chaotic and non-chaotic behavior of the map, the speed and memory performances may vary drastically. Therefore, such profiling may also provide some structural information about a time-series and motivate further analysis of the statistical and topological features of the resulting visibility network.

Table 1
Speed and memory performances of the DC and SC algorithms with respect to the experimental time series data. Each entry given in the brackets presents the execution time for an equivalent random noise of the same length.

Data	Size	Execution time (s)		Allocated memory (MB)	
		DC	SC	DC	SC
Heart rate	1800	0.41 (0.14)	0.40 (0.14)	24.06	1.46
Sensex	6487	10.63 (0.64)	10.1 (0.62)	705.55	27.17
Seismic	20783	4.44 (3.6)	3.97 (2.89)	1210.5	8.8
Dew point	43925	54.88 (14.62)	48.86 (8.55)	10608.5	55.86

6. Conclusions

In a nutshell, the sort-and-conquer algorithm proposed in this study offers several advantages over the existing methods employed to transform a time series data to visibility network. Being non-recursive, it can be used efficiently even when the data-set is large and computational resources are scarce. Therefore, unlike the previously developed techniques, it does not force a space–time trade-off and entirely eliminates the risk of stack overflow for any arbitrary time series. Even in the context of speed of execution, the SC method is either equally efficient or faster than the DC algorithm. Moreover, as the memory requirement diverges very slowly with increase in data size, a programmer is not particularly required to perform extreme fine-tuning for optimization of the memory usage. In view of the rapidly expanding scope of application of the visibility graph method, we hope that the proposed sort-and-conquer algorithm will offer optimum use of computing time and memory space to a broad variety of research themes.

Appendix A

Here we present the module 'visibility', which is called in line 17 of Algorithm 1. It evaluates the presence of visibility links between the current node and all the data elements belonging to its local interval.

In lines 1 and 2, '$-Inf$' and '$+Inf$' represent the smallest and the largest signed values, respectively, which can be unambiguously stored in the allocated bit fields for the variables. In the reported benchmarking tests, the initial blank graph G is represented as a sparse adjacency matrix of zeros. During iterations, lines 7 and 14 of Algorithm 2 are implemented by substituting the default value of 0 by the constant 1 at appropriate locations of the adjacency matrix, thereby representing addition of edges. By simply replacing this constant by the variable 'slope' or its inverse tangent value, we can obtain a more generic form of the visibility graph [34], which includes an additional attribute for each of the edges. Our scheme of implementation ensures that even for such generalized parametric visibility graphs, the relative performances of DC and SC algorithms remain the same.

Algorithm 2 Pseudocode of the '*visibility*' module.

Algorithm 2: module *visibility*(D, *left*, *right*, i, G)

Input: Time series data-set $D\{t, x\}$, where t and x are arrays, each of length N. i is the index of the current node. *left* and *right* are the lower and upper limits, respectively, of the local bracketing interval. G is the current status of the visibility graph.

Output: Updated status of the visibility graph G.

```
1.   max_slope ← -Inf
2.   min_slope ← +Inf
3.   for j = i + 1 to right:
4.   |  slope = (x[j] - x[i])/(t[j] - t[i])
5.   |    if slope >max_slope:
6.   |      max_slope ← slope
7.   |      Add edge (i, j) to G
8.   |    end if
9.   end for
10.  for j = left to i – 1 in reverse:
11.  |  slope = (x[i] - x[j])/(t[i] - t[j])
12.  |    if slope<min_slope:
13.  |      min_slope ← slope
14.  |      Add edge (i, j) to G
15.  |    end if
16.  end for
```

Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.physa.2018.09.053.

References

[1] W.R. Bell, S.H. Holan, T.S. McElroy (Eds.), Economic Time Series: Modeling and Seasonality, CRC Press, Boca Raton, 2012.
[2] G. Ciminoa, et al., Chem. Geol. 161 (1999) 253–270.
[3] W. Gossel, R. Laehne, Hydrol. Earth Syst. Sci. Discuss. 10 (2013) 12793–12827.
[4] I. Auer, et al., Int. J. Climatol. 27 (2007) 17–46.
[5] A. Longobardi, P. Villani, Int. J. Climatol. 30 (2010) 1538–1546.
[6] D. Ninos, et al., IEEE Geosci. Remote Sens. Lett. 1 (2004) 162–165.
[7] D. Kugiumtzis, A. Kehagias, E.C. Aifantis, H. Neuhäuser, Phys. Rev. E 70 (2004) 036110.
[8] A. Anguera, J.M. Barreiro, J.A. Lara, D. Lizcano, Comput. Struct. Biotechnol. J. 14 (2016) 185–199.
[9] A. Onisko, M.J. Druzdzel, R.M. Austin, J. Pathol. Inform. 7 (2016) 50.
[10] H. Madsen, Time Series Analysis, Chapman & Hall/CRC, Boca Raton, 2008.
[11] M.H. Tan, J.K. Hammond, Mech. Syst. Signal Process 21 (2007) 1576–1600.
[12] Z. Wang, et al., Biomed. Signal Process Control 10 (2014) 250–259.
[13] A. Darvish, K. Najarian, D.H. Jeong, W. Ribarsky, Proc. IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (2005), pp. 1-6.
[14] Z. Gao, N. Jin, Chaos 20 (2010) 019902.
[15] P. Li, B. Wang, Physica A 378 (2007) 519–526.
[16] R.V. Donner, Int. J. Bifurcation Chaos 21 (2011) 1019–1046.
[17] Y. Yang, H. Yang, Physica A 387 (2008) 1381–1386.
[18] L. Lacasa, B. Luque, F. Ballesteros, J. Luque, J.C. Nuño, Proc. Natl. Acad. Sci. USA 105 (2008) 4972–4975.
[19] M. Ahmadlou, H. Adeli, Physica D 241 (2012) 326–332.
[20] S. Mutua, C. Gu, H. Yang, Chaos 26 (2016) 053107.
[21] M.A. Rodríguez, Phys. Rev. E 95 (2017) 062309.
[22] N. Wang, D. Li, Q. Wang, Physica A 391 (2012) 6543–6555.
[23] E. Zhuang, M. Small, G. Feng, Physica A 410 (2014) 483–495.
[24] J.B. Elsner, T.H. Jagger, E.A. Fogarty, Geophys. Res. Lett. 36 (2009) L16702.
[25] A. Bhaduri, S. Bhaduri, D. Ghosh, Physica A 482 (2017) 786–795.
[26] L. Lacasa, J. Iacovacci, Phys. Rev. E 96 (2017) 012318.
[27] P. Mali, S.K. Manna, A. Mukhopadhyay, P.K. Haldar, G. Singh, Physica A 493 (2018) 253–266.
[28] A. Aragoneses, L. Carpi, N. Tarasov, D.V. Churkin, M.C. Torrent, C. Masoller, S.K. Turitsyn, Phys. Rev. Lett. 116 (2016) 033902.
[29] S. Murayama, H. Kinugawa, I.T. Tokuda, H. Gotoda, Phys. Rev. E 97 (2018) 022223.
[30] X. Lan, H. Mo, S. Chen, Q. Liu, Y. Deng, Chaos 25 (2015) 083105.
[31] E.L. Leiss, A Programmer's Companion to Algorithm Analysis, Chapman & Hall/CRC, Boca Raton, 2007.
[32] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., MIT Press, London, 2009.
[33] A.L. Goldberger, D.R. Rigney, in: L. Glass, P. Hunter, A. McCulloch (Eds.), Theory of Heart: Biomechanics, Biophysics, and Nonlinear Dynamics of Cardiac Function, Springer-Verlag, New York, 1991, pp. 583–605.
[34] I.V. Bezsudnov, A.A. Snarskii, Physica A 414 (2014) 53–60.