Efficient algorithm to compute mutually connected components in interdependent networks

S. Hwang,[1,2] S. Choi,[2] Deokjae Lee,[2] and B. Kahng[2,*]
[1]*Institute for Theoretical Physics, University of Cologne, 50937 Köln, Germany*
[2]*CCSS and CTP, Department of Physics and Astronomy, Seoul National University, Seoul 151-747, Korea*
(Received 4 September 2014; revised manuscript received 24 December 2014; published 23 February 2015)

Mutually connected components (MCCs) play an important role as a measure of resilience in the study of interdependent networks. Despite their importance, an efficient algorithm to obtain the statistics of all MCCs during the removal of links has thus far been absent. Here, using a well-known fully dynamic graph algorithm, we propose an efficient algorithm to accomplish this task. We show that the time complexity of this algorithm is approximately $O(N^{1.2})$ for random graphs, which is more efficient than $O(N^2)$ of the brute-force algorithm. We confirm the correctness of our algorithm by comparing the behavior of the order parameter as links are removed with existing results for three types of double-layer multiplex networks. We anticipate that this algorithm will be used for simulations of large-size systems that have been previously inaccessible.

DOI: 10.1103/PhysRevE.91.022814 PACS number(s): 89.75.Hc, 64.60.ah, 05.10.−a

I. INTRODUCTION

Networks are ubiquitous in our world and many of these interact with one another [1–4]. One striking instance of a strong internetwork correlation was the power outage in 2003 in Italy [5], during which the power grid network and a computer network strongly interacted with each other. A failure in one network thus led to another failure in the other network and this process continued back and forth. Such avalanche processes can continue until no additional node can fail. This avalanche of failures and its devastating consequences triggered efforts to assess the resilience of interdependent network structures against external forces [5–14].

As a natural measure of resilience of such interdependent networks, the size of a mutually connected component (MCC) per system size has served as an order parameter of the percolation transition [5,15–18]. Here the MCC means that a node belonging to an MCC is connected to all other nodes directly or indirectly in the same MCC in each layer network, called the A-layer network and the B-layer network, respectively. Note that each node in the A-layer network has a one-to-one correspondence with its counterpart node in the B-layer network. However, each layer network has its own set of link connections between nodes and these are independent of those of the other layer network. Although MCCs have been proven to be an excellent measure of network resilience, obtaining results for large-size systems has been computationally difficult because of the absence of an efficient algorithm. This problem was partially solved by a recently proposed algorithm in which a newly designed data structure was used to keep track of the size of a giant MCC efficiently during removal processes of nodes [19]. However, one still needs to resort to the brute-force algorithm when other physical quantities such as the size distribution of the MCCs are requested.

Here we introduce another efficient algorithm that keeps track of not only the size of a giant MCC but also the sizes of all other MCCs and thus the size distribution of MCCs can be traced during removal processes. In particular, our algorithm is designed to proceed as links are deleted. Thus, the percolation transition of the size of a giant MCC can be traced in terms of the actual number of removed nodes. To design it, we utilized a fully dynamic graph algorithm widely used in the computer science community, the one introduced in [20], which is called the HDT algorithm hereafter.

For each layer network, once a component that is not a tree is changed into a spanning tree, its connection profile is saved in a special type of data structure called an Euler tour (ET) tree [21,22], because ET trees can be efficiently managed to merge or split spanning trees. However, to maintain the spanning trees efficiently when link deletions occur, information of redundant paths between nodes needs to be organized properly. The HDT algorithm is a way to maintain such information. It guarantees amortized $O(\log^2 N)$ time for a link deletion or creation when the ET tree is used for the data structure of the spanning trees. The details of the ET tree are presented in Appendix A.

II. ALGORITHM

We first briefly introduce the prerequisites needed to explain our dynamic graph algorithm. To query and update the connection profile of networks, we maintain a data structure for each layer in the form of ET trees. Those ET trees constitute a dynamic forest (DF) denoted by $\mathcal{F}$, which performs the following four operations efficiently.

(1) `Connected`$(v, w, \mathcal{F})$ determines whether or not the nodes v and w are in the same component.

(2) `Size` $(v, \mathcal{F})$ returns the size n of the component (tree) that contains the node v.

(3) `Insert` $(e, \mathcal{F})$ adds a link e to $\mathcal{F}$.

(4) `Delete` $(e, \mathcal{F})$ removes a link e from $\mathcal{F}$.

All of these operations can be performed within $O(\log^2 N)$ computing time using the HDT algorithm.

Our algorithm consists of two main parts: identifying MCCs of a given multiplex network and evolving the MCCs as links are removed one by one. Each update utilizes the previous information of the details of MCCs. Throughout these processes, we trace the evolution of MCCs as a function of the number of links removed.

*bkahng@snu.ac.kr

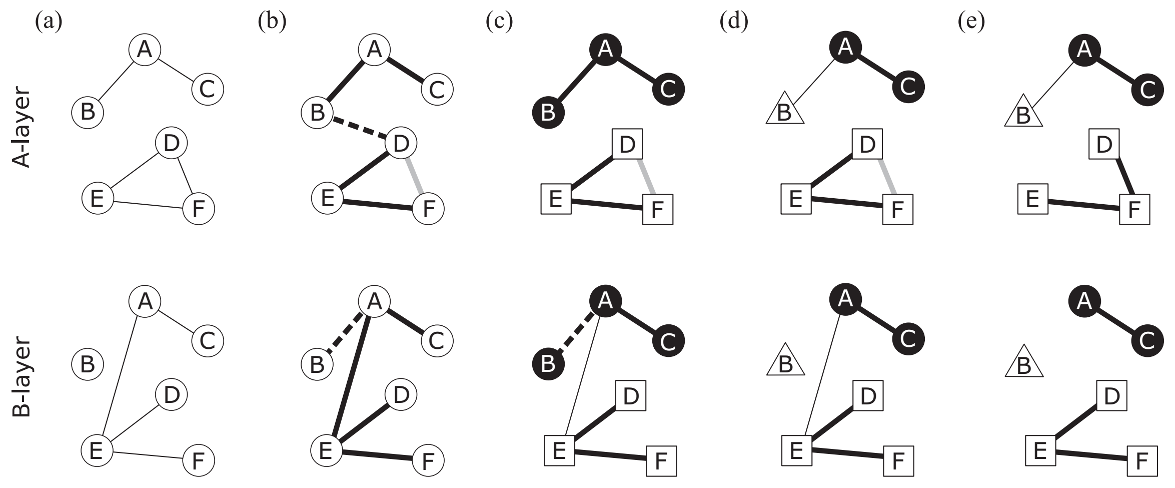

FIG. 1. (a) Initial configurations of A-layer (upper) and B-layer (lower) networks. (b) First, each connected component is maintained in a spanning tree form. Link D-F (gray line) in the A layer is treated as a redundant link. Second, *ad hoc* links (dashed lines) B-D in the A layer and A-B in the B layer are added between two nodes through random selection from each component to connect the networks. Then there is only one MCC and all links including the *ad hoc* links are active (thick lines). (c) An *ad hoc* link B-D is deleted in the A layer. This deletion splits the A network into two components. Subsequently, link A-E in the B layer becomes inactive (thin line) and we identify two MCCs {A,B,C} and {D,E,F}. (d) The other *ad hoc* link A-B in the B layer is deleted. Subsequently, link A-B in the A layer becomes inactive (thin line) and the component {A,B,C} is split into two components {A,C} and {B}. At this stage, there are no remaining *ad hoc* links and the MCCs (represented by different node symbols) of the networks in (a) have been retained with identification of active and inactive links. (e) Now we delete links in the original networks one by one in the same manner. Here we show two examples of link deletion that do not cause a cascade of inactivations: (i) link E-D deletion in the A layer and (ii) link A-E deletion in the B layer. For case (i) the redundant link D-F is recovered and maintains the spanning tree. For case (ii), because the link is inactive, nothing occurs.

To be specific, each link is categorized as active or inactive. Here an active link is the one that belongs to an MCC. The rest of the occupied links are regarded as inactive. For example, the thick solid and dashed lines in Fig. 1 represent active links, whereas the thin lines represent inactive ones. We remark that, even if two nodes (v,w) are connected by a link e in layer A and through a certain pathway in layer B, link e can be inactive when the pathway in layer B contains one or more inactive links. However, once a link is deemed to be inactive, it remains inactive permanently throughout link removal processes. Using this simple fact, we design the algorithm to identify MCCs.

In a double-layer multiplex network with N nodes on each layer, let $\mathcal{L}_A$ and $\mathcal{L}_B$ denote the sets of links present on layers A and B, respectively. The DFs in each network are denoted by $\mathcal{F}_A$ and $\mathcal{F}_B$, respectively. Each $\mathcal{F}_X$ (where X represents either A or B) stores the structure of MCCs of layer X containing connection information of active links.

The first part of the algorithm proceeds as follows.

(i) For a given initial configuration of each layer network [see Fig. 1(a)], a spanning tree is extracted randomly from each component, based on which $\mathcal{F}_X$ $(X = A,B)$ is constructed. By using `Connect`$(v,w,\mathcal{F})$, the connection profile of each network is obtained.

(ii) To identify MCCs, some *ad hoc* links are added between disconnected trees, which means adding *ad hoc* links to $\mathcal{F}_X$.[1] Let $\mathcal{D}_X$ denote the set of all *ad hoc* links in layer X. Then the set of active links can be denoted by $\mathcal{A}_X = \mathcal{L}_X \cup \mathcal{D}_X$ and the set of inactive links becomes $\mathcal{I}_X = \emptyset$ [Fig. 1(b)].

(iii) Choose a link at random from the set of *ad hoc* links $\mathcal{D}_A$ and remove it [Fig. 1(c)]. If $e \in \mathcal{I}_A$, then e is removed from $\mathcal{I}_A$. This case does not occur at the beginning, but it can occur during the iterative processes. If $e \notin \mathcal{I}_A$, execute `Delete`$(e = (v,w),\mathcal{F}_A)$. If no other pathway connecting v and w exists, this component would be split into two. The connection between v and w can be checked via `Connect`$(v,w,\mathcal{F}_A)$ after deleting e.

(iv) As shown in Fig. 1(d), the above division process in one layer may trigger some active links in the other layer into becoming inactive. For each of those inactivated links e, execute `Delete`$(e,\mathcal{F}_X)$ and add it to $\mathcal{I}_X$. This deletion in turn can trigger some other links in $\mathcal{A}_Y$ into becoming inactive; then we repeat the above processes iteratively, where Y represents the counterpart layer of X. The details can be found in Appendix B.

(v) Repeat steps (iii) and (iv) until $\mathcal{D}_A = \emptyset$ and $\mathcal{D}_B = \emptyset$.

After the above first part is completed, all MCCs for a given multiplex network are identified and their structural information such as the sizes of each MCC can also be obtained from $\mathcal{F}_X$ $(X = \{A,B\})$. Next we take the following step to determine the evolution of the MCCs as links are actually removed. This second part of the algorithm can be accomplished by taking steps similar to (iii) and (iv).

[1]The standard ways are visiting each node of a network in a depth-first order or a breadth-first order. Connected components are identified naturally in the visiting and *ad hoc* links between them can be created easily. One simple way may be to pick up a node, from which links are added to the nodes that are not connected to it. However, such a way is inefficient for simulation because a large number of *ad hoc* links are needed.

(vi) Repeat steps (iii) and (iv) on $\mathcal{L}_A$ and $\mathcal{L}_B$ instead of $\mathcal{D}_A$ and $\mathcal{D}_B$. This process is repeated until the number of removed links reaches the value one wants [Fig. 1(e)]. The order of link removals depends on the problem given.

Step (vi) contains the process of removals of active links in the original networks. Thus, we can trace the control parameter. Each updating of the second part builds on the DFs that store the MCC structures obtained in the previous iteration. Therefore, by performing step (vi), we can easily garner the MCCs as a function of the number of remaining links.

III. ASSESSMENT

We check the correctness of our algorithm for three types of double-layer multiplex networks, including (i) random graphs proposed by Erdős and Rényi [23] and (ii) scale-free random graphs introduced in [24], in which the degree of a node in one layer is statistically the same as the one of the corresponding node in the other layer. Thus, degrees of nodes with the same node index on each layer are assortatively correlated [25]. In addition, we also consider (iii) two-dimensional regular lattices. For each type of network, the size of a giant MCC and the number of MCCs are measured as a function of the mean degree. Here the mean degree is given as $k = 2L/N$, where L is the number of links remaining in either $\mathcal{L}_A$ or $\mathcal{L}_B$ at each iteration step. Actually, those numbers are the same. To compare the results in a consistent manner, all networks are initiated by mean degree $k = 4$.

We first examine the size P_∞ of a giant MCC normalized by the system size N in Fig. 2(a). For Erdős-Rényi (ER) graphs, the order parameter exhibits a jump of $P_\infty^{\text{jump}} \approx 0.51$ at $k_c \approx 2.46$, values that are in agreement with the result in [5,15]. For scale-free networks, the jump sizes diminish as the degree exponent γ decreases to 3. For $\gamma \leqslant 3$, the transition becomes continuous and no jump is obtained. This result is also consistent with the previous result in [25] obtained using the conventional algorithm even though nodes are deleted there.

We also perform similar simulations for two-dimensional double-layer regular lattices. We obtain the transition point $k_c \approx 2.29$ in Fig. 2(b), corresponding to the occupation probability $p_c \approx 0.57$ in the conventional scheme. This transition point is between $p_c = 0.5$ for the bond percolation transition and $p_c \approx 0.593$ for the site percolation transition in a two-dimensional monolayer network. Through the numerical results obtained thus far, we have confirmed that our algorithm successfully reproduces the previous results using the conventional algorithms.

We also examine the number N_C of MCCs divided by the system size $s = N_C/N$. One may expect that this quantity s is small when a giant MCC exists in a large mean degree region; however, it is large when most of the links are deleted in a small mean degree region. The behavior of s is shown in Fig. 3. For ER and scale-free networks, we find that s exhibits a behavior similar to that of P_∞ in Fig. 2 but in an upside-down manner. However, for the two-dimensional case, it looks somewhat different. The examination of the behavior of s vs k in such large-size systems would not be possible unless our algorithm is applied.

Next we consider the time complexity of the algorithm. Step (v) forces steps (iii) and (iv) to be repeated $O(N)$ times

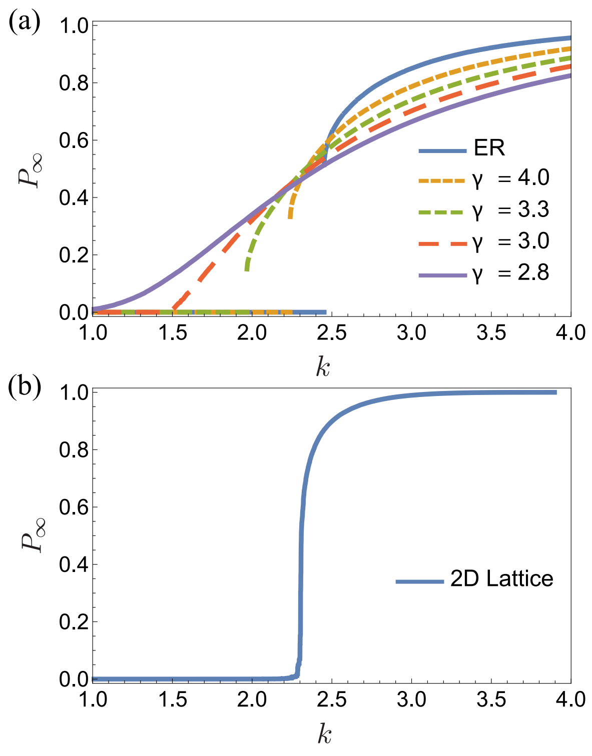

FIG. 2. (Color online) Plot of P_∞ (the size of a giant MCC divided by N) vs the mean degree $k = 2L/N$, where L is the number of remaining links in the system. The system size $N = 10^6$ and an initial mean degree $k_0 = 4$ are taken. As links are removed randomly one by one from each layer, P_∞ exhibits various discontinuous or continuous transitions depending on the underlying networks.

and for each (iii) and (iv) the `Delete` operation has to be executed at least once. These steps request the computing time $O(N \log^2 N)$. However, when one link is deleted, inactivation of a multiple number of links can follow, which may induce cascading of deletions of links back and forth between the two layers. However, it is also possible that the deletion of one link may not induce any further inactivation process. Such complicated processes make it difficult to extract time complexity in a closed form. Thus, we resort to numerical methods to estimate time complexity.

We measure the total computing time T to keep track of the MCCs from $k = 4$ to 1 for the three types of underlying networks. In Fig. 4 we plot T vs system size N. Each data point is obtained by averaging over ten different samples. For ER and scale-free networks, it is likely that the degree exponent γ does not affect time complexity for $k \geqslant 3$, but the constant factor can vary. We find in Fig. 4 that the time complexity depends on the system size as $O(N^{1.2})$ for those complex networks. For a two-dimensional regular lattice, we find that the time complexity is estimated as $T \sim O(N^{1.3})$. This numerical difference may be caused by the nature of the avalanche failures depending on the underlying network structures. Apart from the power-law behavior, one may expect logarithmic corrections affected by the four operations of the DF data structure, but this is not clearly conclusive in Fig. 4.

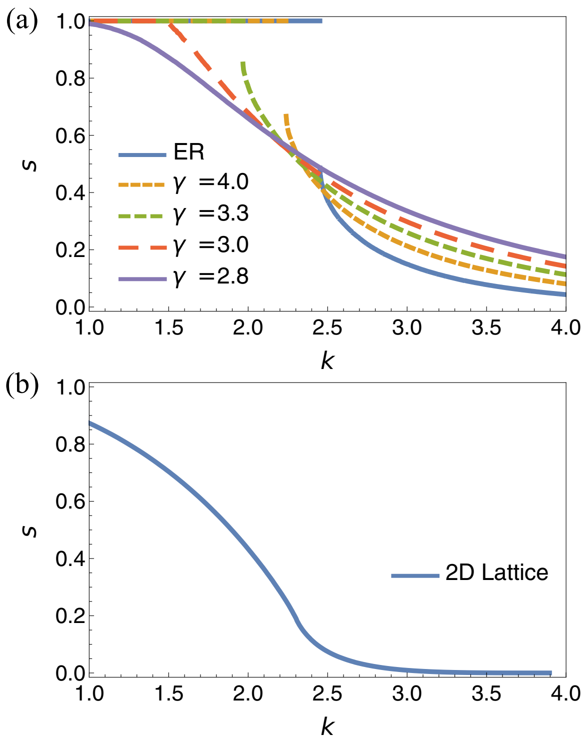

FIG. 3. (Color online) (a) Plot of s (the number of MCCs divided by the system size N) vs the mean degree k under the same conditions as those in Fig. 2. Here s exhibits behavior similar to m but in an upside-down manner for complex networks. (b) However, s exhibits a behavior somewhat different from m for the two-dimensional lattices.

IV. SUMMARY

We have introduced an efficient algorithm that keeps track of the MCCs in an interdependent multiplex network. Our algorithm maintains the full structural information of MCCs during deletions of links and thus it enables one to extract various interesting physical quantities such as the sizes of a giant MCC as well as other MCCs. A similar algorithm was introduced in [19], in which nodes, however, instead of links are deleted. In this case the algorithm can be simpler and a multiple number of links can be simultaneously deleted. Accordingly, the computing time is reduced as $O(N \log N)$. Moreover, the algorithm was designed to trace only the largest cluster. In contrast, our algorithm provides other useful information on structural features of the MCCs. Therefore, we anticipate that our algorithm can facilitate further studies in various directions.

Finally, we remark that the HDT algorithm utilized here can be applied to other problems such as temporal network models [26] in which links can be added or deleted.

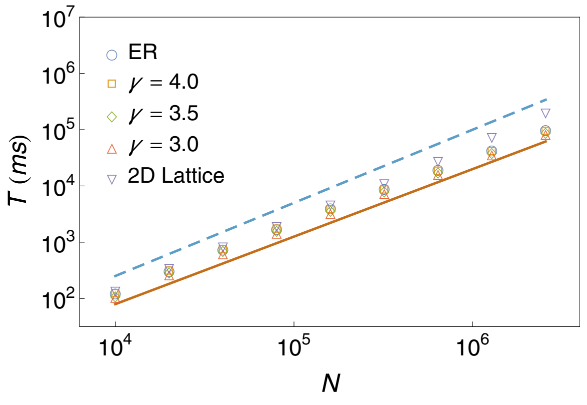

FIG. 4. (Color online) Plot of total computing time T vs system size N for different underlying network structures. Each point is obtained by taking an average over ten samples. The solid and dashed line are guidelines with slopes of 1.2 and 1.3, respectively.

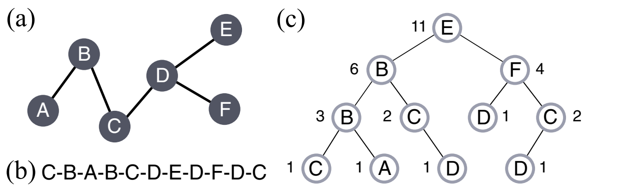

FIG. 5. (Color online) (a) The tree we want to represent. (b) Euler tour sequence from node C of the tree. (c) Sequence stored in a balanced tree. The number next to each node of (c) is the size of the subtree from the node.

ACKNOWLEDGMENTS

This work was supported by National Research Foundation of Korea Grants No. 2010-0015066 and No. 2014-069005, an SNU research grant (B.K.), and the Deutsche Forschungsgemeinschaft within SFB 680 (S.H.).

APPENDIX A: THE ET TREE AND HDT ALGORITHM

The Euler tour tree is a scheme to represent a dynamical tree efficiently. For a spanning tree of size n, for example, the tree in Fig. 5(a), an Euler tour of the tree is a sequence of nodes recorded in a depth-first walk on the tree from an arbitrarily chosen root. It has length (the number of links) $2n - 2$, as shown in Fig. 5(b). It starts from an arbitrary node and ends at the same node. This cycle can be represented as a sequence of $2n - 1$ node indices. Each sequence is then stored in a self-balanced tree consisting of $2n - 1$ nodes [Fig. 5(c)] and its ordering is preserved. Each node of the tree carries the index of the node; thus the leftmost and rightmost nodes of each tree carry the same index. We refer to the trees built this way as Euler tour trees [Fig. 5(c)]. It is noteworthy that a node having degree k in the spanning tree appears k times in the Euler tour tree with one exception being the starting node, the root [node **E** in Fig. 5(c)], which appears $k + 1$ times.

Note that, in self-balanced ET trees, the hierarchical steps from the root to any terminal node are almost the same. Thus the length is $O(\log N)$ and the time complexity is determined accordingly. This property is preserved even in the process of merging and splitting of the spanning trees. There are several widely used algorithms to maintain the balance (e.g., the AVL tree or the red-black tree [27]) and any of them can be used.

One useful piece of information would be the size of the connected component to which a given node belongs. For this, we augment each node of the tree to keep track of the number of its descendants. For example, 11, the augmentation of node E in Fig. 5(c), indicates the number of descendants. Whenever a node is given, identifying the root and hence finding the size of the component can be obtained in $O(\log N)$ steps.

After constructing such an ET tree, we run the four principal operations to the data structure $\mathcal{F}$. Let $\mathcal{E}$ denote a set of links in the network and $\mathcal{S}$ the set of links that constitute the forest. Usually, $\mathcal{E}$ is the set of all links in the network, but in our algorithm it is the set of all *active* links.

(1) `Connected` $(v,w,\ \mathcal{F})$ determines whether v and w are in the same component.

(a) Let r_v and r_w be the roots of the Euler trees containing v and w, respectively.

(b) Return true if $r_v = r_w$; return false otherwise.

(2) `Size` $(v,\ \mathcal{F})$ returns the size n of the component that contains v.

(a) Find the root of the Euler tree containing v.

(b) The root will contain the number s of its decedents, which would be $s = 2n - 2$. Thus return $n = s/2 + 1$.

(3) `Insert` $(e = (v,w),\ \mathcal{F})$ adds a link e to the DF.

(a) Add e to $\mathcal{E}$.

(b) If `Connected` (v,w), do nothing. Otherwise, connect the two Euler trees adjacent to e.

(4) `Delete` $(e = (v,w),\ \mathcal{F})$ removes a link e from the DF.

(a) Remove e from $\mathcal{E}$.

(b) If $e \notin \mathcal{S}$, do nothing. If $e \in \mathcal{S}$, remove e from $\mathcal{S}$. This will split a tree into two pieces. Determine whether there exists a link $e' \in \mathcal{E}$ that can replace e, i.e., connect the two again. If so, add e' to $\mathcal{S}$.

It is clear that `Connected`, `Size`, and `Insert` each require $O(\log N)$ steps. In contrast, in `Delete`, finding e' efficiently is nontrivial. To achieve this, the HDT algorithm assigns an integer, called a level, for each link and maintains a spanning forest for each level. The levels are updated during the search process of e' so that further calls of `Delete` can find their e' more efficiently, which sets the amortized costs of `Insert` and `Delete` to $O(\log^2 N)$. For details we refer the reader to [20].

We, however, found that a brute-force searching for e' without the HDT algorithm is actually faster for our problem. We implemented two versions of our MCC algorithm. One uses brute-force searching and the other uses the HDT algorithm. They exhibit the same time complexity, but the HDT algorithm has a larger prefactor in our empirical tests. Moreover, the HDT algorithm needs additional information and consumes more memory. Thus, we used ET trees without the HDT algorithm in the assessment of our MCC algorithm. However, the HDT algorithm might be needed when the network size becomes much larger than the sizes used in our tests because it theoretically guarantees the $O(\log^2 N)$ time complexity, while we cannot provide a precise time complexity for the brute-force searching.

APPENDIX B: SUCCESSIVE REMOVAL OF LINK e FROM $\mathcal{D}_A$

(1) If $e \in \mathcal{I}_A$, remove e from $\mathcal{I}_A$.

(2) Otherwise, perform `Delete` $(e = (u,v), \mathcal{F}_A)$. This might split an MCC into two. Check whether this occurs using `Connect` $(u,v,\mathcal{F}_A)$.

(a) If `Delete` does not split any connected component, nothing more needs to be done.

(b) If it does, some links in $\mathcal{A}_B$ have to be changed to inactive, as one can see from Fig. 1.

(i) All links in $\mathcal{A}_B$ that connect the two split components will become inactive. To find them, one can scan each node in the smaller component exhaustively and determine whether any of its outgoing links connect it to a node belonging to the larger component.

(ii) For each link e of these, perform `Delete` $(e,\mathcal{F}_B)$ and add it to $\mathcal{I}_B$.

(iii) Of course, this in turn can trigger some links of $\mathcal{A}_A$ into becoming inactive and again we perform `Delete` for each.

(iv) This recursive process has to be performed until no more inactive links are generated.

[1] A.-L. Barabasi and R. Albert, Science **286**, 509 (1999).

[2] A. Barrat, M. Barthelemy, and A. Vespignani, *Dynamical Processes on Complex Networks* (Cambridge University Press, Cambridge, 2008), Vol. 1.

[3] R. Pastor-Satorras and A. Vespignani, *Evolution and Structure of the Internet: A Statistical Physics Approach* (Cambridge University Press, Cambridge, 2007).

[4] S. N. Dorogovtsev and J. F. Mendes, *Evolution of Networks: From Biological Nets to the Internet and WWW* (Oxford University Press, Oxford, 2013).

[5] S. V. Buldyrev, R. Parshani, G. Paul, H. E. Stanley, and S. Havlin, Nature (London) **464**, 1025 (2010).

[6] R. Parshani, S. V. Buldyrev, and S. Havlin, Phys. Rev. Lett. **105**, 048701 (2010).

[7] R. Parshani, S. V. Buldyrev, and S. Havlin, Proc. Natl. Acad. Sci. U.S.A. **108**, 1007 (2011).

[8] J. Gao, S. V. Buldyrev, S. Havlin, and H. E. Stanley, Phys. Rev. Lett. **107**, 195701 (2011).

[9] Y. Hu, B. Ksherim, R. Cohen, and S. Havlin, Phys. Rev. E **84**, 066116 (2011).

[10] X. Huang, J. Gao, S. V. Buldyrev, S. Havlin, and H. E. Stanley, Phys. Rev. E **83**, 065101 (2011).

[11] W. Li, A. Bashan, S. V. Buldyrev, H. E. Stanley, and S. Havlin, Phys. Rev. Lett. **108**, 228702 (2012).

[12] J. Gao, S. V. Buldyrev, H. E. Stanley, and S. Havlin, Nat. Phys. **8**, 40 (2012).

[13] D. Zhou, J. Gao, H. E. Stanley, and S. Havlin, Phys. Rev. E **87**, 052812 (2013).

[14] S.-W. Son, P. Grassberger, and M. Paczuski, Phys. Rev. Lett. **107**, 195702 (2011).

[15] S.-W. Son, G. Bizhani, C. Christensen, P. Grassberger, and M. Paczuski, Europhys. Lett. **97**, 16006 (2012).

[16] G. Bianconi, S. N. Dorogovtsev, and J. F. F. Mendes, Phys. Rev. E **91**, 012804 (2015).

[17] B. Min, S. D. Yi, K.-M. Lee, and K.-I. Goh, Phys. Rev. E **89**, 042811 (2014).

[18] S. Watanabe and Y. Kabashima, Phys. Rev. E **89**, 012808 (2014).

[19] C. M. Schneider, N. A. M. Araújo, and H. J. Herrmann, Phys. Rev. E **87**, 043302 (2013).
[20] J. Holm, K. de Lichtenberg, and M. Thorup, J. ACM **48**, 723 (2001).
[21] M. R. Henzinger and V. King, in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing* (ACM, New York, 1995), pp. 519–527.
[22] M. R. Henzinger and V. King, J. ACM **46**, 502 (1999).
[23] P. Erdös and A. Rényi, Pub. Math. (Debrecen) **6**, 290 (1959).
[24] K.-I. Goh, B. Kahng, and D. Kim, Phys. Rev. Lett. **87**, 278701 (2001).
[25] S. V. Buldyrev, N. W. Shere, and G. A. Cwilich, Phys. Rev. E **83**, 016112 (2011).
[26] P. Holme and J. Saramäki, Phys. Rep. **519**, 97 (2012).
[27] Edited by T. H. Cormen, *Introduction to Algorithms*, 3rd ed. (MIT Press, Cambridge, 2009).