Unbiased randomization of weighted bipartite networks with exact degree and strength sequences

G. Glaviano and S. Micciché
Dipartimento di Fisica e Chimica - Emilio Segrè, Università degli Studi di Palermo, Viale delle Scienze, Ed. 18, 90128 Palermo, Italy

(Received 10 March 2026; accepted 1 July 2026; published 27 July 2026)

In the context of complex networks, designing a null model that preserves certain properties of the original network is a fundamental problem. While for binary bipartite networks, the problem has been solved by the curveball algorithm, which preserves the degree of each node exactly, for weighted bipartite networks, finding a null model that strictly preserves some features of the original network is still an open challenge. In this work, we present a microcanonical algorithm that preserves exactly the degree and the strength of each node in a weighted bipartite network. The algorithm is based on an edge-swap procedure that combines three different moves: weight shuffling, simple edge swap, and bridge edge swap. The algorithm has been built to guarantee that the underlying transition matrix of the Monte Carlo Markov chain (MCMC) is symmetric, which is a crucial property to obtain an unbiased sample of random graphs. We validate our algorithm empirically, showing through a χ-square test that for small graphs, the algorithm produces an unbiased sample of graphs. Finally, we also provide a heuristic method to estimate the mixing time of the MCMC.

DOI: 10.1103/rn1f-zsgb

I. INTRODUCTION

In network science, one of the central challenges is to distinguish structural properties of a network from features arising from randomness. In this regard, considerable efforts have been made to find a null hypothesis that satisfies a certain set of constraints. This problem can be addressed in two ways: through the microcanonical approach, which preserves exactly the required constraints, generating a sample of random graphs, or through a canonical approach, which preserves the constraints only on average [1,2]. One of the earliest works adopting a microcanonical approach was carried out by Maslov and Sneppen [3], who formalized an edge-swap algorithm to randomize a binary network while preserving the in- and out-degrees of each node. Another successful microcanonical method is the curveball algorithm [4,5], which is designed to randomize binary bipartite networks.

In this paper, we propose a new microcanonical method to sample random weighted bipartite graphs with integer weights under the hard constraint of preserving both the degree and the strength sequences. We call this algorithm BiWES (bipartite weighted edge swap). From a microcanonical perspective, some methods have already been proposed to randomize weighted graphs, although under different sets of constraints [6–8]. The closest works to our approach are the canonical method proposed in Ref. [9], which formalizes a null hypothesis that preserves, on average, the degree and the strength of each node, and the information filtering methodology proposed in Ref. [10], both developed for weighted unipartite networks.

BiWES works by randomly performing three moves: *weight shuffling*, *simple edge swap*, and *bridge edge swap*. Each move, when considered individually, preserves the degree and strength of each involved node, and all three are required to fully explore the configuration space. *Weight shuffling* modifies the weights of selected links while preserving the network topology. *Simple edge swap* alters the network topology while preserving the edge-weight distribution. Finally, *bridge edge swap* modifies both the topology and the weights simultaneously. These three moves are combined in a Monte Carlo Markov Chain that allows for switching among graphs until complete randomization is reached. We show that, under the proposed move-selection scheme, the underlying transition matrix that generates the Monte Carlo Markov Chain is symmetric, which is a crucial requirement for obtaining an unbiased sample when using any edge-swap-based algorithm. This is done by considering small networks for which it is computationally feasible to enumerate the entire configuration space, and we perform simulations showing that each graph is sampled with equal probability. We also provide an empirical estimate of the number of moves required to randomize the network.

The manuscript is organized as follows: in Sec. II, we introduce the three moves that enable the randomization process. In Sec. III, we describe how the three moves are combined to construct the BiWES. In Sec. IV we empirically demonstrate that the algorithm produces an unbiased sample. We draw our conclusions in Sec. V. The code to enumerate the configuration space and to randomize a bipartite weighted network is available as a Python package [11].

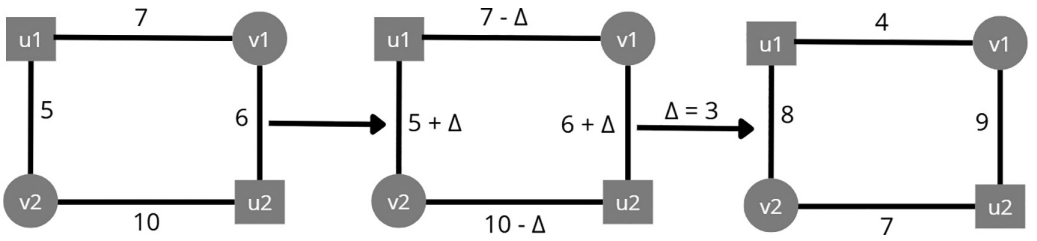

FIG. 1. Example of weight shuffling.

II. RANDOMIZATION MOVES

This paper introduces an algorithm to randomize a weighted bipartite network with integer weights under the following constraints: the degree and the strength must be preserved simultaneously. In other words, we want to preserve both the degree sequence and the strength sequence of the two sets of nodes simultaneously. We approach this problem using a generalized edge-swap algorithm, and we identified three different moves that allow us to shuffle a bipartite network while maintaining these constraints. In this section, we introduce the three moves, analyzing their properties.

A. Weight shuffling

Weight shuffling (WS) is a move that changes the edge weight distribution, but preserves the network topology and the strength of each node. This type of move was first formalized in 1998 in Ref. [12], to randomize a contingency table. Weight shuffling requires 4 nodes, two from the set A, and two from the set B: u_1, u_2, v_1, v_2, arranged in a square. Specifically, the network must contain four links: $e_{11} = (u_1, v_1)$, $e_{22} = (u_2, v_2)$, $e_{12} = (u_1, v_2)$, and $e_{21} = (u_2, v_1)$. Without losing generality, considering the graphical representation of the square Fig. 1, we will say that the links e_{11} and e_{22} are parallel, and also the two links e_{12} and e_{21} are parallel. The idea of this move is the following: If a pair of parallel links transfers a fraction Δ of their weight to the other pair of parallel links, then we will obtain a new graph with shuffled weights, Fig. 1. This move always preserves the strength of each node involved, but we also want to preserve the degree of each node, or even the meaning of the graph, by avoiding negative weights. Considering that we subtract Δ from some weight links, it is necessary to force Δ within a certain range to ensure that every link involved can reach 1 as a minimum possible value. To understand the constraints on Δ, it is useful to write how this move changes the biadjacency matrix.

Let A^t be the current biadjacency matrix, where w^t_{uv} is the weight of the link between node $u \in A$ and node $v \in B$ in the current state. The new matrix A^{t+1} is obtained by changing the four elements corresponding to the edges of the square:

$$A^{t+1}_{u_1v_1} = w^t_{u_1v_1} - \Delta, \quad A^{t+1}_{u_2v_2} = w^t_{u_2v_2} - \Delta,$$
$$A^{t+1}_{u_1v_2} = w^t_{u_1v_2} + \Delta, \quad A^{t+1}_{u_2v_1} = w^t_{u_2v_1} + \Delta. \tag{1}$$

To ensure that the new weights are positive integers, we must derive bounds for Δ. From the first two equations, we get

$$w^t_{u_1v_1} - \Delta \geqslant 1 \Rightarrow \Delta \leqslant w^t_{u_1v_1} - 1,$$
$$w^t_{u_2v_2} - \Delta \geqslant 1 \Rightarrow \Delta \leqslant w^t_{u_2v_2} - 1.$$

This gives us the upper bound: $\Delta \leqslant \min(w^t_{u_1v_1}, w^t_{u_2v_2}) - 1 = b$.

From the other two equations, we get the lower bound:

$$w^t_{u_1v_2} + \Delta \geqslant 1 \Rightarrow \Delta \geqslant 1 - w^t_{u_1v_2},$$
$$w^t_{u_2v_1} + \Delta \geqslant 1 \Rightarrow \Delta \geqslant 1 - w^t_{u_2v_1}.$$

This gives us the lower bound: $\Delta \geqslant 1 - \min(w^t_{u_1v_2}, w^t_{u_2v_1}) = a$. Therefore, Δ must be an integer in the interval $[a, b]$.

If we select a random integer Δ belonging to that interval and then apply the changes to the biadjacency matrix, then we are effectively performing a random selection of one configuration out of all possible configurations that satisfy our initial constraints on the degree and strength of the nodes. It is important to note that, after applying the move, it is always possible to "invert" it, selecting the same square and choosing the appropriate Δ.

B. Simple edge swap

While WS generates a new realization of the network that preserves its topology, the move we refer to as a *simple edge swap* (SES) allows the network topology itself to be modified. This move is inspired by the classic Maslov-Sneppen algorithm [3], which was originally introduced to randomize binary networks under the sole constraint of preserving the in and out degree sequences.

Consider two links from the network, $e_{11} = (u_1, v_1)$ and $e_{22} = (u_2, v_2)$. If the two links have the same weight w and there are no existing links between the nodes (u_1, v_2) and (u_2, v_1), then it is possible to perform an edge swap. The edge swap consists of deleting the original links and creating two new links, $e_{12} = (u_1, v_2)$ and $e_21 = (u_2, v_1)$, as illustrated in Fig. 2. To preserve the strength sequence, the two original links must have the same weight w as e_{11} and e_{22}. This constraint implies that, after this move, not only are the degree and strength sequences preserved, but also the edge weight distribution.

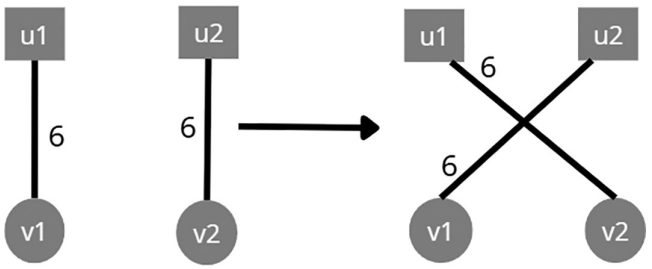

FIG. 2. Example of simple edge swap.

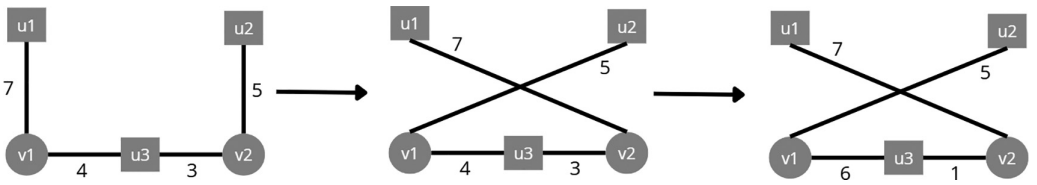

FIG. 3. Example of bridge edge swap.

The two moves described so far are complementary. In fact, WS changes the edge weight distribution but preserves the topology, whereas the SES changes the topology but preserves the edge weight distribution. To construct an algorithm capable of exploring the entire space of graphs with fixed degree and strength sequences, both moves are necessary.

It is important to note that the move can be inverted. In fact, after applying a SES move, the algorithm can go back to the previous state, selecting as two links for the next swap e_{12} and e_{21}, obtaining the two original links.

C. Bridge edge swap

Even though the two previous moves can generate networks with different topologies and edge weight distributions, it is necessary to introduce a third move. There are certain configurations where neither of the previous moves can be applied, despite the existence of other valid networks that satisfy the constraints. In the 5-node network shown in Fig. 3, there are no squares, meaning that weight shuffling cannot be applied. At the same time, there are no pairs of links with identical weights, which makes a simple edge swap impossible. Nevertheless, as shown below, there exists another configuration that cannot be reached using only the two previous moves.

The core concept of this move, which we call *bridge edge swap* (BES), is to perform an edge swap while preserving the strength sequence, even when the two selected edges have different weights. This is possible only if there exists a fifth node that acts as a bridge between the two edges.

Bridge edge swap is defined as follows:

(1) *Selection of edges*. Select two edges $e_{11} = (u_1, v_1)$ and $e_{22} = (u_2, v_2)$ with weight $w_{11} \neq w_{22}$. To apply the move, it is necessary to have a bridge node u_3 such that the two links e_{31} and e_{32} exist.

(2) *Edge swap*. Remove the edges e_{11} and e_{22} and create the edges $e_{12} = (u_1, v_2)$ and $e_{21}(u_2, v_1)$, assigning weights w_{11} and w_{22}, respectively. This operation preserves the degree sequence but induces a strength imbalance on nodes v_1 and v_2, central panel in Fig. 3.

(3) *Strength compensation*. Let us define $\gamma = |w_{11} - w_{22}|$. If $w_{11} > w_{22}$, then node v_2 has an excess of γ units of strength and node v_1 has a deficit of the same amount (and vice versa otherwise). The imbalance is compensated by decreasing (or increasing) the weight of edge e_{32} by γ and increasing (or decreasing) the weight of e_{31} by γ.

(4) *Validity condition*. The move is accepted only if, after the weight adjustment, all the weights remain greater than 0, ensuring that all weights remain positive and that the degree sequence is preserved. In particular, if $w_{11} > w_{22}$, then it is necessary that $w_{32} \geqslant \gamma + 1$. Instead, if $w_{11} < w_{22}$, then it is necessary that $w_{31} \geqslant \gamma + 1$.

The fifth node, which is not present in the SES, is crucial here, because otherwise it would be impossible to adjust the weight for nodes v_1 and v_2. Therefore, a chain of five nodes is the simplest structure that allows an edge swap between links with different weights while preserving the strength. It is worth noting that if $w_{11} = w_{22}$, then the weight compensation γ would be equal to 0, and the move would become identical to SES. Such a condition would break the symmetry of the transition matrix generated by the BiWES algorithm. For this reason, the condition $w_{11} \neq w_{22}$ is explicitly enforced in the first step of the procedure. Also, in this case, it is important to note that after applying the move, it is always possible to find a move that inverts it. In fact, after the move is applied, the five-node structure is preserved, and it is still possible to apply a BES move by removing the links e_{12} and e_{21} and recreating e_{11} and e_{22}. From the tests conducted in Sec. IV, these three moves are sufficient to fully explore the configuration space. For this reason, we did not search for additional or more complex moves involving other sub-network structures.

D. Pairwise disjointness property of the move set

In this section, we want to address the issues associated with implementing a randomization algorithm as an MCMC.

As we will show, one of the key requirements for obtaining unbiased sampling is that the transition matrix of the Markov chain be symmetric. In this section, we introduce a property of the three moves that will be crucial in proving that the transition matrix induced by our algorithm is symmetric. In fact, we will prove that the sets of reachable networks, after applying a single move, are pairwise disjoint.

Let us consider a network G where any of the three moves can be applied. With just one move, we can reach a limited number of networks. If we apply a WS to G, then we can reach a certain set of networks $\mathcal{G}_w$; if we apply a SES move, then we can reach a graph in the set $\mathcal{G}_s$; finally, if we apply a BES move, then we can reach a graph in the set $\mathcal{G}_b$. We want to prove that these three sets are pairwise disjoint:

$$\mathcal{G}_w \cap \mathcal{G}_s = \varnothing \quad \text{and} \quad \mathcal{G}_w \cap \mathcal{G}_b = \varnothing \quad \text{and} \quad \mathcal{G}_s \cap \mathcal{G}_b = \varnothing.$$

This property follows directly from the definition of the three moves. WS modifies only the weights of four links while preserving the network topology. SES, instead, alters the topology of the network. Therefore, the set $\mathcal{G}_w$ is disjoint from both $\mathcal{G}_s$ and $\mathcal{G}_b$. The two edge swap moves do similar things to the graph, but in a different way. Both of them swap two edges, but simple edge swaps do not alter edge weights.

Conversely, a BES changes the weight of the two links used as a "bridge." For this reason, we can conclude that $\mathcal{G}_s$ and $\mathcal{G}_b$ are also disjoint sets.

From this property, it follows that, if a move of a given type transforms a graph G into G^*, then the inverse transition from G^* back to G can only be obtained by applying a move of the same type.

III. BIPARTITE WEIGHTED EDGE SWAP ALGORITHM

In this section, we will describe how these moves can be employed to design an algorithm that produces a randomized version of a bipartite network. We will refer to the algorithm as BiWES. The problem of obtaining a random graph using an edge swap algorithm can be formalized with a Monte Carlo Markov Chain (MCMC) [13]. Therefore, it is essential to clarify the properties that the Markov chain must have to ensure that the final algorithm produces an unbiased sample of random graphs. One of these properties is the symmetry of the transition matrix. For this reason, in this section, we explain exactly how the algorithm decides the next move, showing that for every move, the probability P_{ij} of transitioning from state i to state j is equal to the probability P_{ji} of transitioning from state j to state i.

A. Monte Carlo Markov chain properties to perform an unbiased sample

We can formally define $\mathcal{G} = \{G_1, G_2, \ldots, G_{|\mathcal{G}|}\}$ as the set of all bipartite graphs with a given strength and degree sequence. $|\mathcal{G}|$ is the number of possible configurations. Each graph $G_i \in \mathcal{G}$ can be considered a state of a MCMC process. Every time we apply one of the three previously defined moves to the graph G_a, we obtain a new graph G_b that still belongs to the set $\mathcal{G}$. From this perspective, we can randomize a graph by applying a single move. Clearly, this approach is not sufficient, because for large graphs a single move allows us to reach only a subset of $\mathcal{G}$. Since we are working within an MCMC framework, it is crucial to understand which properties the Markov chain must satisfy to produce an unbiased sample of random graphs. This topic is well discussed in Ref. [13], but for the sake of completeness, we briefly recall the key points here.

Following the lines of [13], there are two necessary (but not sufficient) conditions that are required to perform an unbiased uniform sampling of a state from an MCMC:

(1) The underlying Markov chain must be *irreducible.*

(2) The underlying Markov chain must be *aperiodic.*

If the underlying Markov chain is aperiodic and irreducible, then it is called an ergodic Markov chain.

The aperiodicity is the easiest condition to obtain in the MCMC algorithm. A sufficient condition to guarantee the aperiodicity is that all the states are aperiodic [14,15]. This means that for each state i, there is a probability $P_{ii} > 0$ of remaining in the same state. This is an easy condition to add to any MCMC algorithm because it is sufficient to add an acceptance probability to every single move [15].

A Markov chain is said to be *irreducible* if, for any pair of states i and j, the probability of reaching state j from state i in a finite number of steps is greater than zero. In other words, if we imagine the Markov chain as a directed graph, then it must be a strongly connected graph. So, we want to be sure that by applying the three moves defined in the previous section several times, we can reach every graph in the set $\mathcal{G}$. This condition is related not only to the way one can construct an edge swapping algorithm, but most importantly, to the randomization moves. As we saw in the section dedicated to BES, we are sure that if we remove this move from the final algorithm, then there are some configurations that the algorithm would never visit. So far, we do not have any mathematical proof that our algorithm produces an irreducible Markov chain. But as we will show below, all the tests that we conducted on small networks, when we can actually enumerate all the possible configurations given our constraints, gave us positive results, thus suggesting that indeed the algorithm can visit all the possible states.

As we have already mentioned, if the two previous conditions are satisfied, then the Markov chain is said to be *ergodic*. Ergodicity is a sufficient condition to guarantee the existence and the uniqueness of a *stationary distribution*

$$\Pi = \{\pi_1, \pi_2, \ldots, \pi_{|\mathcal{G}|}\}$$

associated with the Markov chain. The stationary distribution represents the long-term behavior of the chain: after a sufficiently large number of steps t^*, the probability of being in the state i converges to π_i, regardless of the initial state. In the context of edge swap algorithms, we aim to produce an *unbiased sampling*. In other words, we want to simulate a uniform sampling from the set $\mathcal{G}$, meaning that each graph should be sampled with the same probability. This implies that every edge swap algorithm must have a stationary distribution such that

$$\Pi_i = \frac{1}{|\mathcal{G}|}.$$

If the underlying Markov chain is ergodic, then a sufficient condition to achieve this property is that the transition matrix of the process has to be symmetric, $P_{ij} = P_{ji}$ [13]. In the context of edge-swapping algorithms, we do not know the transition matrix explicitly; we simply apply moves to randomize the network. As we will show later, it is not trivial to obtain a symmetric transition matrix, and even small implementation details can change the stationary distribution.

There is one last important condition that we have to discuss to guarantee an unbiased sample. It is possible to construct an ergodic Markov chain with a symmetric transition matrix, but if only a single move is applied, then the resulting network, although it is a randomized version of the original one, will be extremely similar to the original one. This is a well-known problem in the context of MCMC: even though we know that a stationary distribution exists for the Markov chain, we do not know how many steps t^* are required to sample from the distribution Π. In an MCMC procedure, there are typically two phases: a *burn-in* phase and a *stationary* phase [16]. The number of steps required to exit the burn-in is commonly referred to as the *mixing time*. In the context of the edge-swap algorithm, the mixing time is the number of moves needed to obtain a fully randomized version of the network. Estimating the mixing time analytically remains an open problem, both in the context of edge-swap algorithms and in the broader context of MCMC. For our algorithm, in Sec. IV B, we will propose a method to empirically

estimate the number of necessary moves to ensure unbiased sampling.

B. How the algorithm picks a move

1. WS move implementation

Given a square in the network, weight shuffling modifies the weights of all its links, while preserving the strength of each involved node. Therefore, performing this move requires selecting a random square in the network. At first glance, it may seem reasonable to maintain a list of all squares in the network, randomly select one of them, and apply the move. However, although this approach appears theoretically simple, it turns out to be computationally inefficient. After any edge swap move, the topology of the network changes: some squares may be broken, and some other new squares may be created. As a result, the algorithm should update the square list after every edge swap move. This update cannot be done in a computationally constant time. Since large graphs require a huge number of moves to be properly randomized, it is essential that each move be extremely fast.

For this reason, we decided to adopt another strategy to find a random square. We employ a technique known as path sampling [17]. The idea is to start from a randomly chosen node, explore its neighbors step by step, and stop when the desired structure is found. The algorithm that we used to select a random square is the following:

(1) *Select the first node.* Select a random node a and determine the set of its neighbors $\mathcal{N}_a$.

(2) *Select two random neighbors.* Select two random neighbors of a from the set $\mathcal{N}_a$, and call them b and c. These nodes have their own neighbor sets $\mathcal{N}_b$ and $\mathcal{N}_c$. If node a has fewer than two neighbors, then the algorithm does not return any square.

(3) *Find the fourth node.* Randomly select one node between b and c; for example, assume b is chosen. From the set $\mathcal{N}_b$, select a node different from a and call it d. Check whether d also belongs to $\mathcal{N}_c$. If this condition is satisfied, then a square has been found; otherwise, the algorithm does not return any square.

Using an adjacency list as a way to represent the graph, this algorithm has a computational cost of O(1). However, it does not guarantee that a square is found at every call. When no square is identified, the algorithm simply remains in the same graph for one more step.

We also want to emphasize that this selection process is biased; it does not select any square in the graph with the same probability. This is not a problem for the symmetry of the transition matrix. What is crucial is that the probability of each square is the same before and after the move. This is always true, considering that WS does not change the network topology.

2. SES move implementation

To perform a simple edge-swap move, two links with the same weight are required. Therefore, after each move, we can define and update a data structure that associates each unique weight with the links that have that weight:

$$\mathcal{E} = \begin{cases} w_1 : \{(i_1, j_1), (i_2, j_2), \dots\}, \\ w_2 : \{(i_3, j_3), (i_4, j_4), \dots\}, \\ \vdots \\ w_k : \{(i_m, j_m), \dots\}. \end{cases}$$

We implemented the simple edge-swap algorithm as follows:

(1) *Select a random weight.* A random weight w is selected among all available weights.

(2) *Select two random links.* Considering all the links with weight w, select two random links $e_{11} = (u_1, v_1)$ and $e_{22} = (u_2, v_2)$.

(3) *Perform the swap.* Delete the original links e_{11} and e_{22}, and create two new links $e_{12} = (u_1, v_2)$ and $e_{21} = (u_2, v_1)$. The swap can be performed only if the two new links do not already exist. Otherwise, the algorithm does not perform any move and remains on the same graph for an additional step.

However, not all weight classes contain the same number of edges. Consequently, when selecting the weight uniformly from the set of all unique weights, links belonging to less populated weight classes will have a higher probability of being chosen. In other words, this procedure introduces a bias in the link selection, since edges from rare weight classes are overrepresented in the sampling process. This selection bias is not a major issue. In fact, as long as the probability of selecting the same link in the next step remains the same, the transition matrix will stay symmetric. This is always true using this link selection criteria.

This can be proven as follows. Suppose the algorithm is currently in the graph G_i and selects an SES as the next move. The probability of selecting a specific pair of links to perform the swap is

$$P_{ij} = p_1 \cdot p_2,$$

where p_1 is the probability of selecting a random weight w, and p_2 is the probability of selecting a random pair of links (u_1, v_1) and (u_2, v_2) with weight w. Since the weight is chosen uniformly from the set of distinct weights, we have

$$p_1 = \frac{1}{W},$$

where W is the number of unique weights in the network. Moreover, if λ_w denotes the number of links with weight w, then the probability of selecting a specific pair of links with that weight is

$$p_2 = \frac{2}{\lambda_w(\lambda_w - 1)}.$$

After applying the SES move, the algorithm deletes the two selected links and creates two new links (u_1, v_2) and (u_2, v_1), resulting in a new graph G_j. As shown in Sec. II D, the only way to return from G_j to G_i is by performing another SES move. In particular, this requires selecting the same weight w and the two links created by the previous swap, namely (u_1, v_2) and (u_2, v_1). Since the SES move does not alter either the set of unique weights or the number of links associated with each weight, both W and λ_w remain unchanged. As

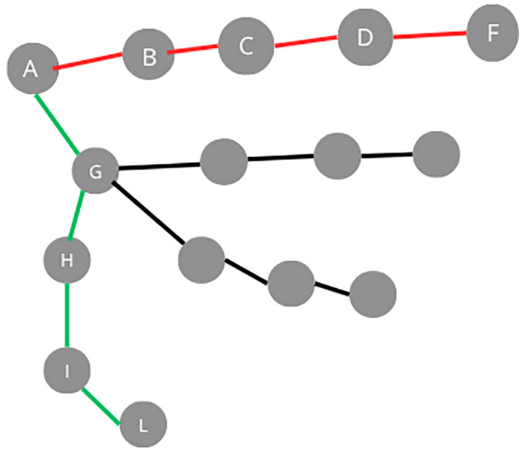

FIG. 4. Example of a random walk to randomly find a five-node structure.

a result, the reverse transition probability satisfies $P_{ji} = P_{ij}$, which proves that the transition matrix is symmetric also for SES.

3. BES move implementation

To perform a BES move, it is required to find a five-node structure that consists of a chain of five nodes. Also in this case, to randomly select this kind of structure, we use path sampling with the following algorithm.

(1) *Select the first node.* Select a random node A.

(2) *Perform a random walk.* One of the neighbors of A is selected at random and labeled B. Then, one of the neighbors of B is selected at random. The random walk continues in this way until a chain of five nodes is reached.

(3) *Check for dead ends.* If, during the random walk, a node with only one neighbor is selected (except for the first and the last nodes), then the random walk reaches a dead end. In this case, the process is stopped, and the algorithm proceeds to the next step while keeping the current graph unchanged.

(4) *Validate the structure.* If the algorithm successfully identifies a chain of five nodes, then it checks whether the chain satisfies all the constraints required to perform the BES moves described in Sec. II C. If the chain is valid, then the move is applied; otherwise, the algorithm proceeds to the next step while keeping the current graph unchanged.

This algorithm does not select the structure uniformly from the network. To show this, we can compute the probability S_{AF} of obtaining the structure corresponding to the red path in Fig. 4, and compare it with the probability S_{AL} of obtaining the structure corresponding to the green path. An important property of the search algorithm described above is that, to find any five-node structure, the first randomly selected node must be one of the two endpoints of the chain. Therefore, in the example shown, to obtain the red path, the algorithm must select either node A or node F as the starting node. Consequently, to compute S_{AF}, it is sufficient to evaluate two probabilities: $S_{A\to F}$, the probability that, starting from node A, the random walk follows the red path and reaches F; and $S_{F\to A}$, the probability that, starting from node F, the random walk reaches node A along the same path. Finally, S_{AF} can be written as

$$S_{AF} = \frac{1}{15}[S_{A\to F} + S_{F\to A}]. \tag{2}$$

The constant $\frac{1}{15}$ is the probability of randomly selecting a specific node. Since the algorithm does not allow the random walk to step backwards, once the algorithm selects the node F, it is forced to follow the red path until the node A, so $S_{F\to A} = 1$. However, if A is the first node selected, then the probability of moving to B is $\frac{1}{2}$. Once node B is selected, the algorithm is forced to reach the node F, therefore $S_{A\to F} = \frac{1}{2}$, and finally using Eq. (2) we find $S_{AF} = \frac{1}{10}$. The probability of obtaining the green path is instead given by

$$S_{AL} = \frac{1}{15}[S_{A\to L} + S_{L\to A}]. \tag{3}$$

The probability $S_{A\to L}$ is $\frac{1}{2} \times \frac{1}{3}$, because starting from A the probability of reaching G is $\frac{1}{2}$, from G there is a probability of $\frac{1}{3}$ of choosing H as the next step. Instead, starting from the node L, the probability of following the green path until A is reached is $S_{L\to A} = \frac{1}{3}$. Using Eq. (3) we find $S_{AL} = \frac{1}{30}$. This proves that this selection algorithm is biased.

As mentioned in Secs. III B 1 and III B 2, this bias in the move selection is not a significant issue as long as the transition matrix remains symmetric. Let us therefore prove that our matrix is indeed symmetric. Let us assume that the algorithm is currently in the state represented by the graph G_i, and that BES is selected as the next move. Suppose that a valid structure that connects the node A to the node F is found, so in the next step, the algorithm will reach the state G_j. The probability of reaching this state is

$$P_{ij} = S^i_{AF}, \tag{4}$$

where S^i_{AF} denotes the probability of choosing a five-node structure that has the nodes A and F as endpoints in the graph G_i. As discussed in Sec. II D, returning from state G_j to the state G_i is only possible by applying another BES move. As shown in Fig. 3, after applying the move, the five-node topological structure remains unchanged, in the sense that all the nodes are still connected in a chain with the same two nodes as endpoints. Therefore, starting from the state G_j, the inverse move leading back to G_i can only occur if a BES move is applied and the same five-node structure is selected. So, we can write the probability P_{ji} as

$$P_{ji} = S^j_{AF}, \tag{5}$$

where S^j_{AF} is the probability of choosing a five-node structure with nodes A and F as endpoints for graph G_j. So we only need to prove that $S^i_{AF} = S^j_{AF}$. Given that the number of neighbors of each node is the same before and after the move, the previous equality is satisfied. Consequently, the transition matrix is symmetric.

ALGORITHM 1. BiWES.

Require: Initial Graph G, Number of steps K
Ensure: Randomized Graph G
data_structures ← PRECOMPUTEDATASTRUCTURE(G) ▷ Initialize all the data structures
for $k \leftarrow 1$ **to** K **do**
  $r \leftarrow$ random number in [0, 1)
  Move Selection:
  if $r < 1/3$ **then** ▷ **Attempt Weight Shuffling (WS)**
    $S \leftarrow$ FINDRANDOMSQUARE(G) ▷ Returns a square or ∅
    if $S \neq \emptyset$ **then**
      $G \leftarrow$ APPLYWEIGHTSHUFFLING(G, S, data_structures)
      data_structures ← UPDATE_DATA_STRUCTURESDATA_STRUCTURES
    end if
    ▷ If $S = \emptyset$, G remains unchanged
  els if $r < 2/3$ **then** ▷ **Attempt Simple Edge Swap (SES)**
    $w \leftarrow$ select a random weight from the list of all weights
    $e_{11}, e_{22} \leftarrow$ Select two uniform random edges with weight w
    if ISSWAPFEASIBLE(e_{11}, e_{22}) **then**
      $G \leftarrow$ APPLYSIMPLEEDGESWAP(G, e_{11}, e_{22}, data_structures)
      data_structures ← UPDATE_DATA_STRUCTURESDATA_STRUCTURES
    end if
    ▷ If the swap is not feasible, G remains unchanged
  else ▷ **Attempt bridge edge swap (BES)**
    $C \leftarrow$ FINDFIVENODECHAIN(G) ▷ Returns chain or ∅
    if $C \neq \emptyset$ **and** CANPERFORMBES(C) **then**
      $G \leftarrow$ APPLYBRIDGEEDGESWAPG, C, data_structures
      data_structures ←UPDATE_DATA_STRUCTURESDATA_STRUCTURES
    end if
    ▷ If search failed or move invalid, G remains unchanged
  end if
end for
return G

C. Algorithm implementation

The previous sections introduced the three algorithms used to select a specific move. Now we will describe the algorithm that takes a graph G as input, randomly repeats these three moves K times, and finally outputs the randomized graph G_r.

At each step, the algorithm has a probability $\frac{1}{3}$ of selecting any one of the three moves. Once the move is chosen, the algorithm will look for a valid structure, as described in the previous sections. If a valid structure is found, then the algorithm applies the move, and a new network configuration is obtained; otherwise, the algorithm proceeds to the next step, keeping the graph unchanged.

The overall procedure is described in the pseudo-code reported in Algorithm 1. In this algorithm, we make extensive use of the variable *data_structures* and of the function *update_data_structures*, which updates them. To ensure that the computational cost for each move time does not increase with the network size, it is necessary to simultaneously maintain multiple representations of the current network and update them after every single move. For the interested reader, we described in the Appendix all the algorithms and data structures we used to optimize the code.

IV. ALGORITHM TESTS

We have shown that the algorithm induces a symmetric transition matrix. Moreover, since at each step there is a nonzero probability of remaining in the same state ($P_{ii} \neq 0$), the corresponding Markov chain is aperiodic. However, a formal proof of irreducibility is still lacking; in particular, it is not guaranteed that the three proposed moves are sufficient to generate every graph in the set $\mathcal{G}$.

In the following, we provide numerical evidence suggesting that the Markov chain is irreducible and that the algorithm performs an unbiased sample of graphs. For sufficiently small networks, it is possible to explicitly enumerate the entire set $\mathcal{G}$ within reasonable computational time using an exact algorithm, available at [11]. For these restricted cases, corresponding to specific degree and strength sequences, we verify that all graphs in $\mathcal{G}$ are visited by the chain. Furthermore, we observe that, for sufficiently large values of K, each graph is sampled with equal probability $\frac{1}{|\mathcal{G}|}$.

A. Empirical evidence for unbiased sampling

An unbiased sample process can be thought of as an urn model, where the set $\mathcal{G}$ represents the urn, and each graph

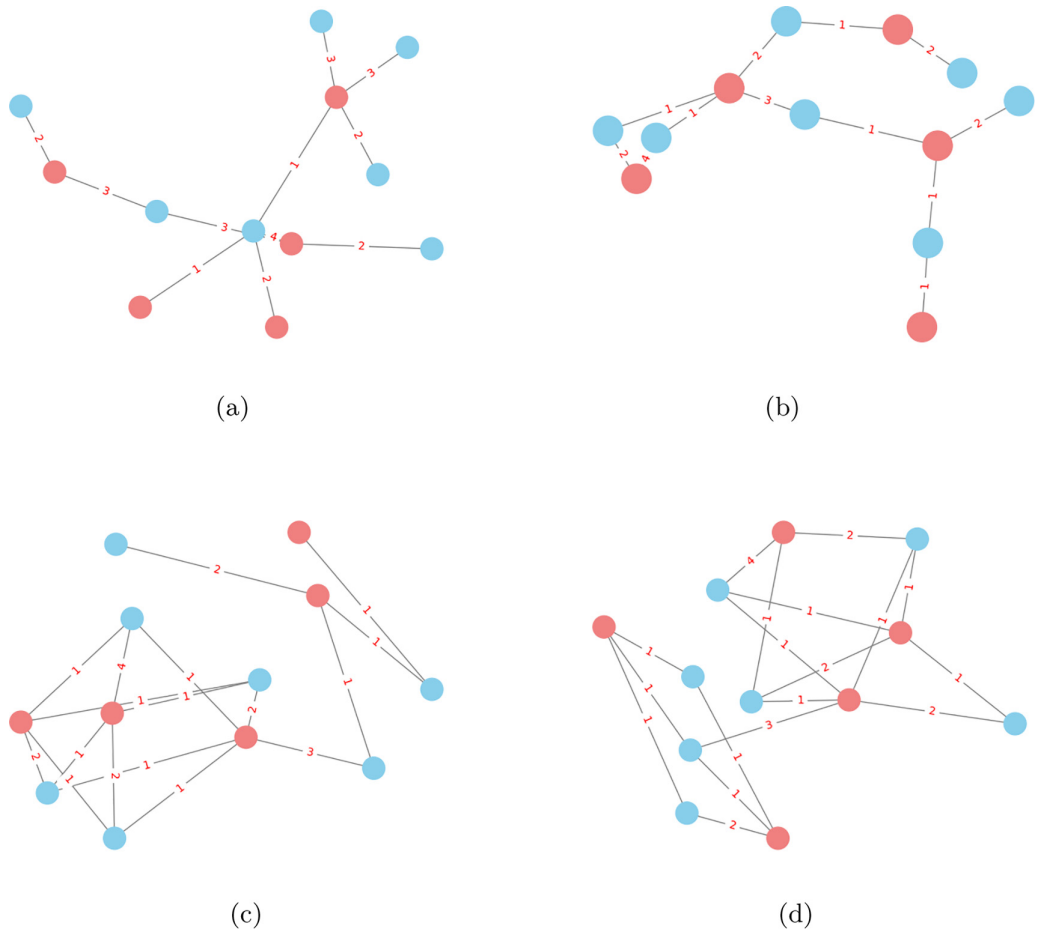

FIG. 5. Starting graphs used as a test to show empirically that the sampling is unbiased. (a) 11 links $|G| = 427$. (b) 12 links $|G| = 1904$. (c) 17 links $|G| = 50722$. (d) 18 links $|G| = 216652$.

corresponds to an element that can be drawn with equal probability $p = \frac{1}{|\mathcal{G}|}$. The process of drawing Z elements from this urn with replacement follows a uniform distribution. Under this null hypothesis, when performing a random sample of Z graphs, the expected number of times each graph is sampled, E_i, is given by

$$E_i = pZ = \frac{Z}{|\mathcal{G}|}.$$

Therefore, if we perform an actual sample of Z graphs using BiWES, then we can count how many times each graph G_i is drawn, creating a counting vector O, where the entry O_i represents the number of times the graph G_i has been drawn. By comparing the expected counts E with the observed counts O, using the χ-square test, we can check whether the sample obtained using the BiWES is unbiased.

In Fig. 5 we show four example networks with 12 nodes with different link densities. For each network, we computed the degree and strength sequences, as well as the full configuration space consistent with those sequences. Although the graphs are relatively small, the number of possible configurations, $|\mathcal{G}|$, grows rapidly as the number of links increases, ranging from 427 to over 200 000. For the first two cases [Figs. 5(a) and 5(b)], we performed the test by sampling $Z = 10^6$ graphs, each obtained by applying 1000 moves to the original graph. Due to the larger size of the configuration space, for the graph in Fig. 5(c) we performed $Z = 2 \times 10^6$ simulations, while for graph in Fig. 5(d) we performed $Z = 10^7$ simulations. In all cases, the p-value associated with the χ-square test is greater than 0.3, providing empirical support that the sampling procedure is unbiased. This type of test can only be performed on small graphs, since the number of possible configurations increases very rapidly with the number of nodes and links, making the enumeration problem computationally infeasible within a reasonable time.

B. Empirical estimation of the mixing time

Estimating the mixing time of an MCMC Markov chain is a nontrivial task. In the context of edge swapping algorithms, the mixing time is typically assessed by monitoring the evolution of specific network characteristics over time, such as the number of triangles or squares [18], or, in the case of directed networks, the number of feed-forward loops [19]. For binary bipartite networks, Neal proposed a method based on computing the distance between the networks during the randomization process [20]. We propose a new heuristic method based on the convergence of the empirical probability distribution of edge weights.

Let us consider a probability matrix $\mathcal{P}^{\text{True}}$ of size $n_a \times n_b$, where the entry $\mathcal{P}^{\text{True}}_{ij}$ represents the probability that nodes i and j are connected by a link with weight greater than or equal

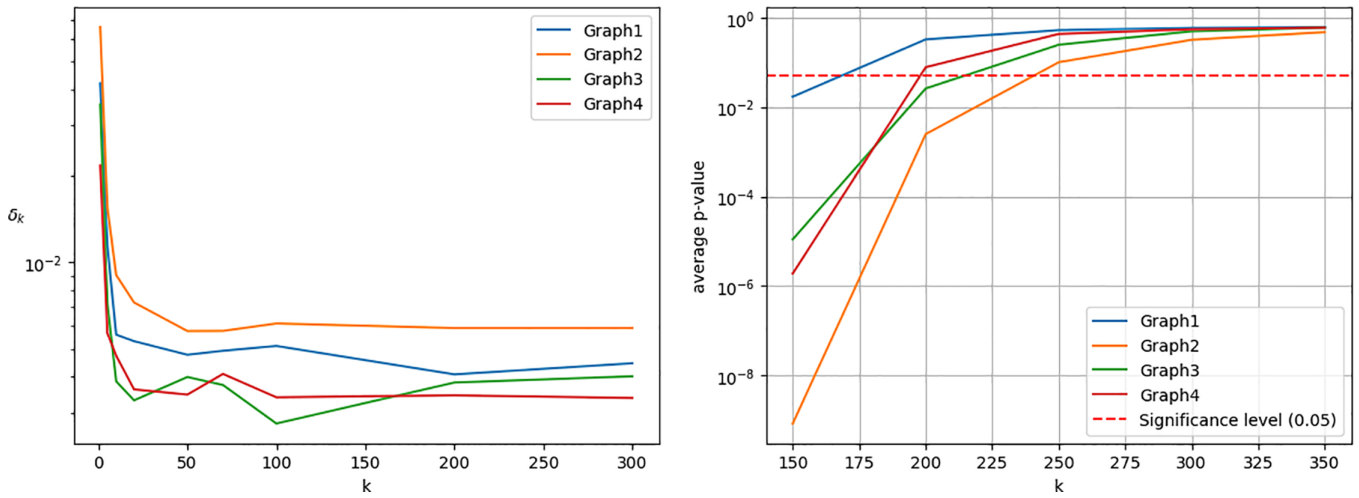

FIG. 6. Mixing time analysis for edge-swapping MCMC algorithms: left panel shows the decay of the mean relative error of the empirical probability matrix as a function of swaps per edge, while the right panel reports Neal's stopping rule based on the stabilization of graph-distance distributions.

to the observed weight $w_{ij}^{\rm obs}$. One possible way to estimate this matrix is to generate $N_{\rm run}$ randomized versions of the network and count how many times the randomized weight satisfies $w_{ij}^{(r)} \geqslant w_{ij}^{\rm obs}$. We decided to study the evolution of the matrix $\mathcal{P}$ for estimating the mixing time for a practical purpose that we did not address in this work, namely, the statistical validation of links through a microcanonical graph ensemble. Indeed, the entry $\mathcal{P}_{ij}$ represents the p-value associated with the link having observed weight $w_{ij}^{\rm obs}$. Therefore, we consider the study of the mixing time through the matrix $\mathcal{P}$ as the basis for future work.

Starting from the original network G, after t randomization moves we sample a single network G^{t_1} and, for each entry, check whether $w_{ij}^{(r)} \geqslant w_{ij}^{\rm obs}$. After this check, G^{t_1} is taken as the new input, and the chain is allowed to evolve for another t steps before the next sample is collected. This procedure is known as *thinning* in the context of MCMC methods [21].

This process leads to the construction of a counting matrix C. An estimate of the matrix $\mathcal{P}^{\rm True}$ is then obtained by dividing the counting matrix by $N_{\rm run}$:

$$\mathcal{P} = \frac{1}{N_{\rm run}} C. \tag{6}$$

The estimation of $\mathcal{P}$ may depend on the thinning step t. If t is too small, then the samples may be biased because the Markov chain has not yet passed the burn-in phase. Once the burn-in phase has been reached, further increasing the thinning step does not lead to significant changes in the estimated probability matrix $\mathcal{P}$.

With this in mind, we conducted experiments on a set of real networks, each taken from [22], to empirically investigate the typical mixing time of our algorithm. For each real graph G with L links, we fixed the number of simulations to $N_{\rm run} = 10^6$ and computed multiple realizations of the probability matrix $\mathcal{P}$. We denote by $\mathcal{P}^k$ a single realization, where k indicates that, before generating each randomized version of the graph, we applied $L \cdot k$ moves. In all our tests, $k \in [1, 500]$.

The matrix $\mathcal{P}^{500}$ is expected to be the closest approximation to the true matrix $\mathcal{P}^{\rm True}$, even though, from a theoretical standpoint, we have no guarantee that $500L$ moves are sufficient to randomize the network. Nevertheless, we can compare $\mathcal{P}_{ij}^{500}$ with the matrices obtained for smaller values of k. Specifically, for each k, we can compute the average relative error as follows:

$$\delta_k = \frac{1}{|\Omega|} \sum_{(i,j)\in\Omega} \frac{\left|\mathcal{P}_{ij}^{k} - \mathcal{P}_{ij}^{500}\right|}{\mathcal{P}_{ij}^{500}}, \tag{7}$$

where $|\Omega|$ denotes the cardinality of the set $\Omega = \{(i, j) : w_{ij} > 0 \ \wedge \ \mathcal{P}_{ij}^{500} > 10^{-4}\}$. We restrict our analysis to probabilities larger than 10^{-4}, since, given that $N_{\rm run} = 10^6$, considering smaller probabilities may lead to large relative errors in the probability estimates.

The left panel of Fig. 6 shows the behavior of the mean relative error δ_k as a function of the parameter k, for four different real-world networks [23–26]. All four curves exhibit a monotonically decreasing behavior of δ_k as the number of moves used to randomize the network increases. This decay continues until a plateau is reached for all curves. We identify the onset of this plateau as the critical mixing time, denoted by k_c. Beyond this point, allowing the MCMC to further evolve does not provide any additional advantage in the estimation of $\mathcal{P}$. From Fig. 6, we infer that $50 < k_c < 70$. Consequently, this test suggests that the number of moves required to obtain an unbiased sample using BiWES lies in the range $50L < t < 70L$.

We compared our approach for estimating the mixing time with the stopping rule proposed by Ref. [20]. Here, we briefly summarize Neal's method. Let A be the biadjacency matrix of the initial graph. We create an ensemble of V identical copies of A and let them evolve independently. After t swaps, we measure the distance between the original matrix A and each randomized matrix $A_i^{(t)}$ in the ensemble. This yields a vector D_t of length V, which represents the distance distribution from the original matrix after t swaps. Subsequently, we perform Δt additional swaps on all matrices and again compute the distances from the original matrix, obtaining a new vector $D_{t+\Delta t}$. The method iteratively compares the distributions D_t and $D_{t+\Delta t}$ using the two-sample Kolmogorov-Smirnov

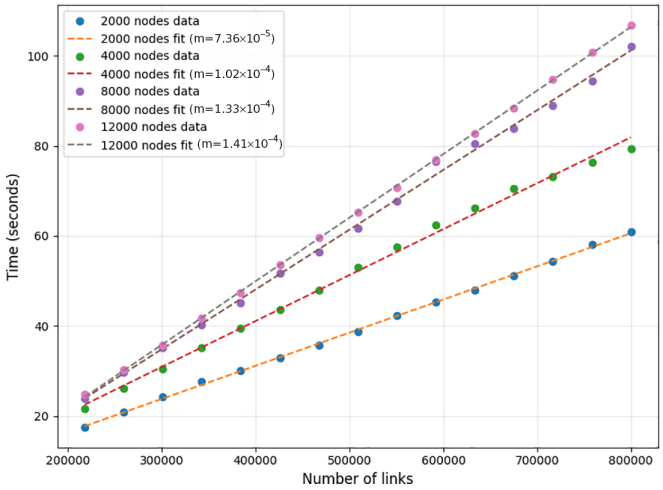

FIG. 7. Runtime of the BiWES algorithm as a function of the number of links for different network sizes.

test, stopping when the resulting p-value exceeds the 0.05 threshold, which indicates that the distance distribution has stabilized. For each graph tested, we set $V = 500$, computed each p-value 1000 times, and used the average as a reference for the test. To measure the distance between graphs, we used the normalized Manhattan distance, which is the same metric that Neal uses in his paper.

We applied Neal's test to the same four real-world networks we used to perform our test based on the matrix $\mathcal{P}$. The results are summarized in the right panel of Fig. 6. The mixing time estimated by this test is higher. In fact, the average p-value appears to stabilize with a number of swaps around 200–300 times the number of links. Nevertheless, our tests show that, if this algorithm is used for statistical validation, then a less conservative approach using a number of swaps equal to 50 or 70 times the number of links is still reliable, considering that no significant differences are observed in the computed p-values as the number of swaps increases.

Once the mixing time has been estimated, it is important to evaluate the computational cost of the algorithm in terms of runtime. In Appendix, we show how the algorithm is implemented so that each move has constant computational cost. In this section, we demonstrate that the number of moves required to randomize a network scales linearly with the number of links. Therefore, the expected computational cost of the BiWES algorithm is $O(L)$. We empirically test this hypothesis using the following setup: we generate random weighted bipartite networks, increasing the number of nodes from 2,000 up to 12,000, and increasing the number of links from 10 000 up to 800 000. For each generated network, we apply BiWES for $50 \times L$ moves, measuring the required runtime. In Fig. 7, we report the results. As expected, the runtime increases linearly with the number of links. Moreover, the slope (angular coefficient) increases as the number of nodes becomes larger. This effect is due to the fact that, although each swap is performed in constant time, increasing the number of nodes increases the constant factor associated with each operation. From this plot, it can be seen that although a single realization can be obtained in a reasonable amount of time even for graphs with 10 000 nodes, generating an ensemble of millions of graphs would be too computationally expensive for this microcanonical algorithm. It is nonetheless important to emphasize that, when multiple realizations are required, the algorithm can be easily parallelized across multiple cores, since each realization is independent. The simulation in Fig. 7 was performed using an Intel Core i7-14700K processor.

V. CONCLUSIONS

In this work, we propose a new microcanonical algorithm to randomize a bipartite weighted network, preserving the degree and the strength of each node. Compared to other edge-swap algorithms, this method features three different moves: WS, SES, and BES, which are applied randomly to the network.

We formalized this problem as an MCMC and described the properties to obtain an unbiased sample of graphs, namely ergodicity and the symmetry of the transition matrix. In Sec. III B, we proved that BiWES produces a symmetric transition matrix; However, at the present stage, a rigorous mathematical proof of ergodicity is still lacking. This theoretical limitation represents the main issue of the algorithm. Indeed, if irreducibility were not satisfied, then the algorithm would produce biased graph sampling. Although this theoretical limitation remains, we have shown that, at least for small graphs where it is possible to fully enumerate the state space, the algorithm is always able to successfully explore the entire space of graphs with a fixed degree and strength sequence. While this test, performed in Sec. IV A, cannot replace a formal proof of ergodicity of the algorithm, we would like to emphasize that no combination of degree and strength sequences has been found that exhibits fragmentation of the state space. The irreducibility of BiWES remains an open theoretical problem, and it would be interesting to investigate in future work whether this algorithm does or does not generate an irreducible Markov chain.

We also derived, using two different methods, the number of moves required to achieve unbiased sampling. The stopping rule suggested by Ref. [20] indicates that the number of moves required is in the range of 200 to 300 times the number of links. However, our tests based on the convergence of the p-value suggest that there are no substantial changes in the p-value estimates after 50–70 times the number of links, showing that, at least for future statistical validation purposes, it is possible to use this less conservative range.

A slow mixing time leading to high computational costs is undoubtedly the main limitation of all microcanonical null models on graphs. This slow convergence to the stationary distribution can result in prohibitive computational costs when the size and the number of links in a graph become too large, and in such cases, the use of canonical models based on the maximum entropy principle is the only computationally feasible solution. However, for small graphs, microcanonical algorithms such as BiWES are preferred. finite-size effects and statistical fluctuations inherent to canonical models are large. In these cases, the estimation of some observables may be too affected by fluctuations.

The importance of the methodology we illustrated here is twofold. On one side, we provide a microcanonical methodology for shuffling a bipartite network while preserving degree

and strength simultaneously. This is the first such algorithm, to our knowledge. On the other side, this algorithm could be used anytime one needs to formulate realistic null hypotheses on bipartite networks.

This is the case when one is interested in validating the links of a bipartite network. For this purpose, BiWES can provide a null hypothesis for a computationally based statistical validation of the links. In Sec. IV B, we started to develop a framework for statistical validation, which will be further investigated in future work.

Other examples are given by the search of communities in bipartite networks. In this case, BiWES might be used to create a generalized modularity [27] that takes into account both degree and link weights.

ACKNOWLEDGMENTS

G.G. acknowledges financial support in this work by the European Union Next Generation EU through the project GRINS (Growing Resilient, Inclusive and Sustainable), funded within the framework of the Piano Nazionale di Ripresa e Resilienza (PNRR), Mission 4 ‘Istruzione e Ricerca’, Component 2 ‘Dalla Ricerca all’Impresa’, Investment 1.3 (PE0000018).

DATA AVAILABILITY

The data used in this paper consist of four real-world networks, all publicly available online [22]. The code used to reproduce all results is available both as a Python package and through the associated GitHub repository [11]. All the algorithms discussed in this paper are available as a Python library [11].

APPENDIX: TECHNICAL DETAILS

In this Appendix, we describe the implementation details of the BiWES, providing a clear overview of the data structures and algorithms employed. As will be discussed, we use multiple data structures to represent the same network. This design choice allows each elementary move to be performed in constant time, at the cost of increased memory usage. Since the additional memory overhead is not prohibitive, we consider this approach to be the most effective way to ensure that the algorithm remains practical and efficient when applied to real-world networks. One of the most commonly used data structures to represent a network is the adjacency matrix. In our case, since we are dealing with an undirected weighted bipartite graph, we employ a biadjacency matrix. This data structure allows us to check the weight of a given link in $O(1)$ time, which is useful for all three types of moves. Moreover, when the weight of any link changes, updating the biadjacency matrix can also be done in constant computational time.

During the execution of the algorithm, applying a weight shuffling move or a complex edge swap requires a random walk within the network. After randomly selecting the first node, one must inspect its nearest neighbors, randomly choose one of them, and continue this process until the required structure is reached, as described in the preceding sections. Given a node in the set A, finding all its nearest neighbors requires $O(n_b)$ computational time when using the biadjacency matrix,

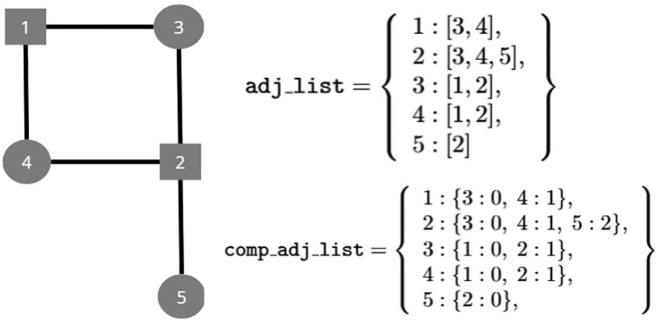

FIG. 8. Example of adjacency list and complementary adjacency list of a graph.

making an algorithm based solely on this data structure extremely slow.

Typically, when frequent access to the set of nearest neighbors of a node is required, a more convenient data structure is the adjacency list, implemented as a hash map, commonly called a dictionary in Python. Therefore, at the beginning of the simulation, we precompute a dictionary that is updated during the simulation. The dictionary maps each node i to a Python list containing the indices of all its neighbors:

$$\texttt{adj\_list} = \begin{Bmatrix} 1 : [a_1, a_2, a_3, \dots], \\ 2 : [b_1, b_2, \dots], \\ 3 : [c_1, c_2,], \\ \vdots \end{Bmatrix}.$$

This adjacency list allows us to perform two operations in constant time: retrieving the list of all neighbors of a node and randomly selecting one neighbor from that list. However, after an SES or CES move, two new links are created, and two are deleted, so the sets of nearest neighbors for the four nodes change, and the dictionary must be updated accordingly. Adding a new link (i, j) is computationally straightforward with this data structure because accessing the neighbors of i or j requires constant time, and appending j to the list of neighbors of i also requires constant time using the `append` method. The problem arises when removing a link from the network, since removing j from the list of neighbors of i requires $O(\text{degree}[i])$ computational time, which can again lead to inefficiency. To solve this problem, we decided to use a well-known technique called *swap-and-pop*, which requires another Python dictionary (or more generally another hash map) extremely similar to the adjacency list that we called *complementary adjacency list*. Specifically, the primary key of this dictionary is the index of a node, denoted by i. However, the value associated with i is not a simple list of neighbors, but rather a nested dictionary. In this inner dictionary, the keys correspond to the neighbors j of node i, while the associated values store the index position currently occupied by j in the adjacency list of i. In essence, the standard adjacency list answers the question: *“Who is the k*th *neighbor of node i?,”* whereas the complementary adjacency list addresses the inverse query: *“At which index k can neighbor j be found within the adjacency list of node i?”* To better understand these two complementary dictionaries, an example is shown in Fig. 8. The complementary adjacency matrix allows us to remove a node from the list of neighbors, because it guarantees fast

access to the index in which the node to remove j is stored, allowing the *swap-and-pop*.

To perform a simple edge swap, it is necessary to identify two links sharing the same weight. For this reason, it is convenient to store an additional dictionary that associates each distinct weight with the set of links having that weight:

$$\texttt{weight\_dictionary} = \begin{Bmatrix} w_1 : [(a,b),\ (b,c),\ \dots], \\ w_2 : [(e,f),\ (g,h),\ \dots], \\ \vdots \end{Bmatrix}.$$

Whenever the weight of a link is modified, this dictionary must be updated by adding or removing the corresponding link. This update operation poses the same challenge encountered when maintaining the adjacency list. To address this issue, we again rely on a complementary dictionary that enables the use of the *swap-and-pop* technique, ensuring constant-time updates.

To achieve a significant speed-up, every function that composes BiWES is compiled using Numba [28].

[1] J. Park and M. E. J. Newman, Statistical mechanics of networks, Phys. Rev. E **70**, 066117 (2004).

[2] G. Cimini, T. Squartini, F. Saracco, D. Garlaschelli, A. Gabrielli, and G. Caldarelli, The statistical physics of real-world networks, Nat. Rev. Phys. **1**, 58 (2019).

[3] S. Maslov and K. Sneppen, Specificity and stability in topology of protein networks, Science **296**, 910 (2002).

[4] G. Strona, D. Nappo, F. Boccacci, S. Fattorini, and J. San-Miguel-Ayanz, A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals, Nat. Commun. **5**, 4114 (2014).

[5] K. Godard and Z. P. Neal, Fastball: A fast algorithm to randomly sample bipartite graphs with fixed degree sequences, J. Complex Networks **10**, cnac049 (2022).

[6] T. Opsahl, V. Colizza, P. Panzarasa, and J. J. Ramasco, Prominence and control: The weighted rich-club effect, Phys. Rev. Lett. **101**, 168702 (2008).

[7] G. Ansmann and K. Lehnertz, Constrained randomization of weighted networks, Phys. Rev. E **84**, 026103 (2011).

[8] V. Zlatic, G. Bianconi, A. Díaz-Guilera, D. Garlaschelli, F. Rao, and G. Caldarelli, On the rich-club effect in dense and weighted networks, Eur. Phys. J. B **67**, 271 (2009).

[9] R. Mastrandrea, T. Squartini, G. Fagiolo, and D. Garlaschelli, Enhanced reconstruction of weighted networks from strengths and degrees, New J. Phys. **16**, 043022 (2014).

[10] R. Marcaccioli and G. Livan, A Pólya urn approach to information filtering in complex networks, Nat. Commun. **10**, 745 (2019).

[11] https://github.com/Glaviano/biwes.

[12] P. Diaconis and B. Sturmfels, Algebraic algorithms for sampling from conditional distributions, Ann. Stat. **26**, 363 (1998).

[13] C. J. Carstens and K. J. Horadam, Switching edges to randomize networks: What goes wrong and how to fix it, J. Complex Networks **5**, 337 (2017).

[14] A. Papoulis, *Random Variables and Stochastic Processes* (McGraw Hill, New York, NY, 1965).

[15] D. A. Levin and Y. Peres, *Markov Chains and Mixing Times* (American Mathematical Society, Providence, RI, 2017), Vol. 107.

[16] B. Walsh, Markov chain Monte Carlo and Gibbs sampling, Lecture Notes EEB **581**, 3 (2004).

[17] M. Jha, C. Seshadhri, and A. Pinar, Path sampling: A fast and provable method for estimating 4-vertex subgraph counts, in *Proceedings of the 24th International Conference on World Wide Web* (Association for Computing Machinery, New York, 2015), pp. 495–505.

[18] L. Tabourier, C. Roth, and J.-P. Cointet, Generating constrained random graphs using multiple edge switches, J. Exp. Algorithmics **16**, 1 (2011).

[19] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon, On the uniform generation of random graphs with prescribed degree sequences, arXiv:cond-mat/0312028.

[20] Z. P. Neal, A stopping rule for randomly sampling bipartite networks with fixed degree sequences, Social Networks **80**, 59 (2025).

[21] W. A. Link and M. J. Eaton, On thinning of chains in MCMC, Methods Ecol. Evol. **3**, 112 (2012).

[22] https://icon.colorado.edu/networks.

[23] D. W. Inouye and G. H. Pyke, Pollination biology in the Snowy Mountains of Australia: Comparisons with montane Colorado, USA, Aust. J. Ecol. **13**, 191 (1988).

[24] M. Kato, T. Kakutani, T. Inoue, and T. Itino, Insect-flower relationship in the primary beech forest of Ashu, Kyoto: An overview of the flowering phenology and the seasonal pattern of insect visits, Contrib. Biol. Lab. Kyoto Univ. **27**, 309 (1990).

[25] A. F. Motten, Pollination ecology of the spring wildflower community of a temperate deciduous forest, Ecol. Monogr. **56**, 21 (1986).

[26] M. Schleuning, N. Blüthgen, M. Flörchinger, J. Braun, H. M. Schaefer, and K. Böhning-Gaese, Specialization and interaction strength in a tropical plant—Frugivore network differ among forest strata, Ecology **92**, 26 (2011).

[27] M. J. Barber, Modularity and community detection in bipartite networks, Phys. Rev. E **76**, 066102 (2007).

[28] S. K. Lam, A. Pitrou, and S. Seibert, Numba: A LLVM-based Python JIT compiler, in *Proceedings of the 2nd Workshop on the LLVM Compiler Infrastructure in HPC* (Association for Computing Machinery, New York, 2015), pp. 1–6.