A Fast Method to Calculate Hitting Time Distribution for a Random Walk on Connected Undirected Graph

Enzhi Li

Suning R&D Center, Palo Alto, USA

(Dated: August 20, 2019)

With the advent of increasingly larger graphs, we need to find a quick and reliable method to measure the distance between a pair of nodes. There are already algorithms for measuring this distance using the concept of random walk on graphs. However, the implementation of that algorithm is expensive due to the use of Monte Carlo simulations. Here, we propose an alternative measure of the distance between any two nodes that are connected to each other via a path on the graph, a distance which we call the hitting time of random walks on graph. We also give an analytical solution to the hitting time of a random walk on a connected and undirected graph. This analytical solution, which can be conveniently implemented using numerical linear algebra packages, is more time-saving and more accurate than the Monte Carlo simulation results. It is further noted that the hitting times also provide a glimpse of the community structure of a graph.

I. INTRODUCTION

With the advent of the era of social media, graphs have become an increasingly important tool for studying user behavior in such fields as social networks, academic citation, online retails, web analysis, etc. Various graph algorithms have been proposed to attack the problems people encounter during the study of graphs. For example, in order to find the web pages that are of uttermost importance and relevance to a set of keywords, people designed the PageRank algorithm to rank the tens of millions of web pages that are available on line[1]. Nodes on a connected graph may form tightly bound communities, which can be detected using an algorithm that aims to maximize the modularity of each potential community[2–4]. We can equally well study the division of a connected graph into weakly linked communities via the Laplacian matrix, which is derived from a graph's adjacency matrix[5]. It is also of significant interest to find a metric that can measure the distance between any two nodes on a connected graph, and the node2vec methods is there to our resort[6]. All of these algorithms depend on a direct or indirect invocation of a graph's adjacency matrix, an invocation that is understandable due to the fact a graph can be uniquely identified once we have access to its adjacency matrix.

Often times, we also need to explore the influence of a node on another node, or on a specific set of nodes in a graph. PageRank algorithm is a convenient method that can deliver how influential and important a single node is to the graph as a whole, whereas sometimes we also need to know how influential a node is to another specific node, a situation that could arise when we want to know how susceptible a community of nodes is to the presence of a fraudulent user in a social network. For example, we can create a connected graph whose nodes represent the users of an online retailer like Suning.com. If we already know that one user in the graph is a fraudulent user, then we want to know how likely other users that cooccur with the fraudulent user is also fraudulent. Intuitively,

this likelihood depends on the distance between these two users. However, there is no universal definition of this distance, and the definition may vary for each specific case. Nowadays, there are already plenty of algorithms that can measure the distance or similarity between each pair of nodes in a graph, such as the node2vec algorithm[6], and the algorithm for calculating the betweenness of two nodes[7]. These algorithm are essentially considering a random or non-random walk on a graph, and by analyzing the nodes and length of a path connecting the starting node and the target node, we can gain some insight into the distance or similarity between the starting node and the target. Several methods that rely on the concept of random walk on graph have been proposed. One such method starts from a node, say A, and selects a node, say B, as its target, and performs several random walks starting from A and counts how many times this random walker encounters node B within a specified number of steps. This encountering frequency for the random walk provides a measure of the distance between nodes A and B. The larger the frequency, the shorter the distance between A and B. This method of of measuring distances between two nodes, although valid in some sense, has several drawbacks, the most prominent of which is its strong dependence upon such capricious parameters as the maximum number of nodes each random walk can possibly traverse, and the number of random walks to be performed for the frequency to be statistically stable and meaningful. Another weak point of this method is that it is much too time-consuming to perform enough number of random walks to gain a statistically significant result for two nodes that are located afar in a huge graph. The application of this method to a tiny graph is no less troublesome due to the fact that a random walk starting from node A is guaranteed to reach any other node in a connected graph as long as the random walking process lasts long enough, rendering all the nodes have identical distance to a target node. As a result, the differences between these nodes are utterly erased. Considering the time-expensiveness of performing a multitude of random walks on a large graph and the strong dependence of the

final results on the hard-to-choose parameters, we prefer to find an alternative method that can deliver the exact solution to the random walk, thus avoiding this lengthy and tedious process of Monte Carlo simulation from the beginning. People already explored this direction, and mathematical results indicate that performing random walks on a graph is equivalent to solving a discrete Laplacian equation[8]. However, solution of Laplacian equations requires the pre-existence of boundary conditions, which are not always available[9]. For example, in order to quantify the influence of a fraudulent user on the other users that cooccur in the social network, we also need the existence of at least one user that is explicitly labeled as "non-fraudulent" or "innocuous", a label that is not always available.

In order to avoid these conundrums, here we propose a new algorithm that can measure the distance between any two nodes in a graph by giving an exact solution to a random walk problem on undirected graphs. The advantage of this algorithm is that the final result is uniquely obtained by solving a sparse linear system, thus releasing us of the unnecessarily thorny duty of selecting a set of appropriate parameters, and saving us tens of thousands of CPU hours from performing Monte Carlo simulations thanks to the linear algebra packages that are readily available for performing sparse matrix multiplications quickly and conveniently.

The organization of the paper is as follows. In section II, we will show that we can measure the distance between any two nodes A and B located in a graph by performing a random walk starting from A, and see when this random walker hits the target node B for the first time, a time that is called hitting time in a standard textbook on stochastic processes[10]. The hitting time can be estimated using Monte Carlo simulation which is pretty time-consuming, as we already lamented, or can be solved exactly using mathematical techniques that we will elaborate in exquisite details in this paper. After describing the algorithm, we next apply our analytical and numerical methods to small and huge graphs respectively. In section III A, we will exemplify our method by giving an analytical solution to a random walk problem on a simple small graph. We will also cross validate our analytical results by performing Monte Carlo simulations on the same graph, showing that they both yield identical results within machine precision tolerance. For a large graph, analytical solutions only involving mathematical formulae are no longer feasible, and we will resort to highly efficient numerical methods. Section III B is devoted to a numerical solution to the equations derived in section II for huge graphs. We will also perform Monte Carlo simulations on the large graph as benchmarks to demonstrate the rapidity of our method in delivering solutions of higher precision.

II. EXACT SOLUTION TO HITTING TIMES OF RANDOM WALKS ON GRAPH

In this section, we will present an analytical method for finding hitting times of a random walk on an undirected and connected graph G. The connectedness of the graph does not constitute a major restriction to our method due to the availability of efficient algorithms for finding connected components of an undirected graph. The adjacency matrix of this graph is A, which is an $V \times V$ matrix, with matrix elements $A_{ij} = 1$ if there is an edge between node i and node j, otherwise $A_{ij} = 0$. The matrix dimension V is the number of nodes in the graph, and the number of non-zero matrix elements of A gives us the edge number. A is symmetrical if graph G is undirected, or else it is generally non-symmetric. Since we are trying to find the probability of reaching a target node from any other node in the graph, whereas in a directed graph, a node may not be reachable from another node, and that is why we only consider undirected and connected graphs. Thus, the adjacency matrix is always symmetric in our case. Furthermore, our method also applies to weighted graphs, for which $A_{ij} = w > 0$ if the edge connecting node i to j has weight w, and $A_{ij} = 0$ if there is no edge connecting between nodes i and j.

A random walk on a graph is defined as follows:

> **1**. Start from a certain node, and call it our current node, which is denoted as c;
> **2**. Select a node r at random from the neighbors of current node, where node r is chosen with probability proportional to the weight of the edge connecting node c to node r;
> **3**. Designate node r as our updated current node c;
> **4**. Repeat steps $1 - 3$ until we have reached a target node or util the number of steps reaches a pre-specified limit.

For this random walk process, we define its hitting time as the number of steps needed for the random walker to reach a target node for the first time. According to this definition, the hitting time is a random variable that depends on the graph structure, the starting node, and the target node. Here in this section, we will derive a formula for calculating the probability distribution for hitting times. The derivation of this formula requires the knowledge of graph adjacency matrix A, the starting node s, and the target node t. Denote the probability of hitting target t after exactly n steps starting from node s as

$$x_n^{(s)} = P(N_t = n; s), \tag{1}$$

where N_t is a random variable that counts the number of steps used before the random walker hits node t for the first time. Assume that node s has m neighboring nodes, of which at most one is the target node t. We enumerate

these m nodes using indices $i_s = 1, 2, ..., m$. Then the probability $P(N_t = n; s)$ can be recursively represented as

$$P(N_t = n; s) = \sum_{\substack{i_s=1 \\ i_s \neq t}}^{m} \frac{w_{s,i_s}}{W_s} P(N_t = n-1; i_s) \qquad (2)$$

In the above equation, $W_s = \sum_{i_s=1}^{m} w_{s,i_s}$ is the total weight associated with node s, $w_{s,i_s}/W_s$ represents the probability for the random walker to make a transition from node s to one of its neighbors i_s, and $P(N_t = n - 1; i_s)$ is the probability of reaching target node t from node i_s after exactly $n-1$ steps. Since we are calculating the probability of reaching target t for the first time from node s after exactly n steps, the probability of reaching target starting from the target itself is zero for any non-zero number of steps, i.e., $P(N_t = n; t) = 0, \forall n > 0$. Thus, we demand that the random walker in the above recursive equation should not make a transition to the target node t even if t is one of the neighbors of node s, which justifies our notation $i_s \neq t$ in the summation subscript. If we scan all possible starting vertices s, we can obtain a simultaneous system of difference equations for the hitting probabilities $P(N_t = n; i), \forall i \in \mathbb{V}, i \neq t$, where $\mathbb{V}$ is the set of all vertices in graph G. Moreover, if a vertex j has only one single neighbor, and this very neighbor is just our target, then we can directly write out the probability of reaching target t after exactly $n > 0$ number of steps starting node j as $P(N_t = n; j) = \delta_{n,1}$, with $\delta_{n,1}$ being the Kronecker δ function. Since we already know the probability distribution of hitting times for such queer nodes which we call *adherents* to target t, we can ignore those nodes when establishing the simultaneous system of difference equations for $P(N_t = n; s)$.

Definition 1. A node in graph G is called an ***adherent*** to target node t if and only if this node has the target node as its only neighbor.

With these notations, we can write out a system of simultaneous difference equations for the probabilities of hitting target node t for the first time with exactly $n > 0$ steps after starting from different nodes $i \in \mathbb{V}$ as

$$P(N_t = n; i) = \sum_{\substack{j=1 \\ j \neq t}}^{|\mathbb{V}|} B_{ij} P(N_t = n-1; j), \qquad (3)$$

where we have imposed the restriction that the starting node i should not be equal to t, and should not be an *adherent* to target t. The B matrix is called probability transition matrix, the elements of which are $B_{ij} = w_{ij}/\sum_{i'} w_{ii'}, i' \in \{neighbors\ of\ i\}$ for $A_{ij} \neq 0$, and $B_{ij} = 0$ whenever $A_{ij} = 0$. Most of the time, B is a sparse matrix. Note that although in an undirected graph the adjacency matrix A is always symmetric, the probability transition matrix is generally non-symmetric.

Moreover, due to the exclusion of the target node t in the definition of probability transition matrix, the sum of matrix elements for each row in B is not necessarily equal to 1. In fact, for a connected undirected graph, there is at least one row of B whose sum is less than 1. The rule is that $\sum_j B_{ij} = 1$ if the target node t is not a neighbor of node i; otherwise $\sum_j B_{ij} < 1$. Since we have excluded the target node t and all its adherent nodes from the set of starting nodes, the matrix B has a dimension that is smaller than that of matrix A. For an undirected graph, the matrix B is guaranteed to be square due to the fact that a random walker starting from a node that is not target cannot possibly reach an adherent node to the target. The fact that matrix B has rows with sum that are less than 1 means that this matrix is not a Markov matrix, and that all its eigenvalue have a magnitude smaller than 1. As a result, the spectral radius of matrix B is also smaller than 1. We will take advantage of this fact later in this section.

Eq. [3] can be re-written in compact matrix form as

$$\boldsymbol{X}_n = B\boldsymbol{X}_{n-1}, n \geq 2 \qquad (4)$$

where $\boldsymbol{X}_n$ is a column vector, the elements of which are the hitting probabilities $X_n^{(i)} = P(N_t = n; i)$. An iterative solution to Eq. [4] is

$$\boldsymbol{X}_n = B^{n-1}\boldsymbol{X}_1, n \geq 1 \qquad (5)$$

The initial probability vector $\boldsymbol{X}_1$ can be conveniently obtained by the observation that $X_1^{(i)} = 0$ if the target node t is not a neighbor of node i, and that $X_1^{(i)} = w_{it}/w_{ii'}, i' \in \{neighbors\ of\ i\}$ if the target node t is a neighbor of node i. Now that we have known matrix B and the initial probability vector $\boldsymbol{X_1}$, we can calculate all the hitting probabilities for any valid starting node.

Although we can calculate all the hitting probabilities, most of the time, we are more interested in the observable quantities associated with these probability distributions. We can calculate the moments of the probability distribution by invoking their definitions, which are

$$\langle N_t^m \rangle_i = \sum_{n=1}^{\infty} P(N_t = n; i)n^m := \sum_{n=1}^{\infty} X_n^{(i)} n^m \qquad (6)$$

The expectation and variance of the first hitting time starting from any node i can be easily calculated from the first and seconds moments of the hitting probability distribution. At first sight, it seems that we need to know all the hitting probabilities before we can calculate their moments. However, we can exploit the fact the spectral radius of matrix B is less than 1 and directly calculate the moments vector $\langle \boldsymbol{N}_t^m \rangle$ from the recursive Eq. [4]. To accomplish this, we need to first define the characteristic function for the probability density function $f(x)$ as

$$\hat{f}(\omega) = \int_{x \in \mathbb{R}} f(x)e^{i\omega x} dx \qquad (7)$$

For a discrete series like $X_n^{(i)}$, the probability density function is

$$f^{(i)}(x) = \sum_{n=1} X_n^{(i)} \delta(x-n), \tag{8}$$

where $\delta(x-n)$ is the Dirac δ function with the property that for any continuous function $f(x)$, $\int_{x\in\mathbb{R}} f(x)\delta(x-x_0)dx = f(x_0)$. The characteristic function of $f^{(i)}(x)$ is

$$\hat{f}^{(i)}(\omega) = \int_{x\in\mathbb{R}} f^{(i)}(x)e^{\mathrm{i}\omega x}dx \tag{9}$$
$$= \sum_{n=1}^{\infty} X_n^{(i)} e^{\mathrm{i}\omega n}$$

If we further define $z = e^{\mathrm{i}\omega}$, then the characteristic function can be more compactly rewritten as

$$\tilde{f}^{(i)}(z) = \sum_{n=1}^{\infty} X_n^{(i)} z^n \tag{10}$$

We can read off the expectation value and variance of hitting probabilities $X_n^{(i)}$ from the first and second derivatives of $\tilde{f}(z)$ as

$$\langle N_t^{(i)} \rangle = \frac{d}{dz}\left(\tilde{f}^{(i)}(z)\right)\Big|_{z=1} \tag{11}$$
$$\langle N_t^{(i)2} \rangle = \frac{d^2}{dz^2}\left(\tilde{f}^{(i)}(z)\right)\Big|_{z=1} + \frac{d}{dz}\left(\tilde{f}^{(i)}(z)\right)\Big|_{z=1}$$
$$\mathrm{Var}(N_t^{(i)}) = \langle N_t^{(i)2} \rangle - \langle N_t^{(i)} \rangle^2$$

If we consider $N_t^{(i)}$ as the ith component of $\boldsymbol{N}_t$, and $N_t^{(i)2}$ as the ith component of $\boldsymbol{N}_t^2$ which is a component-wise square of vector $\boldsymbol{N}_t$, then the above three relations can be simplified into the form

$$\langle \boldsymbol{N}_t \rangle = \tilde{\boldsymbol{f}}'(1) \tag{12}$$
$$\langle \boldsymbol{N}_t^2 \rangle = \tilde{\boldsymbol{f}}''(1) + \tilde{\boldsymbol{f}}'(1) \tag{13}$$
$$\mathrm{Var}(\boldsymbol{N}_t) = \langle \boldsymbol{N}_t^2 \rangle - \langle \boldsymbol{N}_t \rangle^2 \tag{14}$$

Here, we have defined a vector function $\tilde{\boldsymbol{f}}(z)$ as

$$\tilde{\boldsymbol{f}}(z) = \sum_{n=1}^{\infty} \boldsymbol{X}_n z^n \tag{15}$$

Since the coefficients of $\tilde{f}^{(i)}(z)$ are the hitting probabilities $X_n^{(i)}$, we call it the *generating function* of hitting probabilities. Plug Eq. [4] into the above definition, and we get

$$\tilde{\boldsymbol{f}}(z) = \left(\sum_{n=1}^{\infty} z^n B^{n-1}\right)\boldsymbol{X}_1 \tag{16}$$
$$= z(I-zB)^{-1}\boldsymbol{X}_1$$

The second line of the above equation stems from the fact that $|z| = 1$ and that the spectral radius of matrix B is smaller than 1.

We have already known how to calculate the probability transition matrix B and the initial probability vector $\boldsymbol{X}_1$, we can in principle calculate exactly the generating function from Eq. [16]. However, calculating the inverse of matrix $I - zB$ is no easy task, especially when the graph is huge. Moreover, the fact that the sparsity of matrix B which we should take full advantage of can get lost after matrix inversion compels us to shun the idea of directly inverting matrix $I-zB$ to calculate hitting probability moments. Therefore, we have devised a trick for finding probability moments without resorting to matrix inversion. For this purpose, we rewrite Eq. [16] as

$$(I-zB)\tilde{\boldsymbol{f}}(z) = z\boldsymbol{X}_1 \tag{17}$$

Performing first order derivative of both sides with respect to z and setting $z = 1$ yields

$$-B\tilde{\boldsymbol{f}}(1) + (I-B)\tilde{\boldsymbol{f}}'(1) = \boldsymbol{X}_1 \tag{18}$$

By definition, $\tilde{\boldsymbol{f}}(1) = \begin{pmatrix} 1 & 1 & 1... & 1 \end{pmatrix}^{\mathrm{T}}$, and $\tilde{\boldsymbol{f}}'(1)$ gives us the first moment of $\boldsymbol{X}_n$, which is equal to

$$\tilde{\boldsymbol{f}}'(1) = (I-B)^{-1}(B\tilde{\boldsymbol{f}}(1) + \boldsymbol{X}_1) \tag{19}$$
$$= (I-B)^{-1}\tilde{\boldsymbol{f}}(1)$$

The second line of the above equation is due to the identity that $\tilde{\boldsymbol{f}}(1) = B\tilde{\boldsymbol{f}}(1) + \boldsymbol{X}_1$, which can be easily verified by plugging $z = 1$ into Eq. [16]. We can avoid inverting sparse matrices, an operation that will destroy the sparsity of a matrix, by noting that Eq. [19] can be rewritten as (remember that the spectral radius of B is smaller than 1)

$$\tilde{\boldsymbol{f}}'(1) = \sum_{n=0}^{\infty} B^n \tilde{\boldsymbol{f}}(1) \tag{20}$$

It is noteworthy that the first moments of hitting probabilities starting from each valid vertex are independent of the initial probability vector $\boldsymbol{X}_1$, and depend only on the probability transition matrix B. Taking the second order derivative of Eq. [17] yields

$$\tilde{\boldsymbol{f}}''(1) = 2B(I-B)^{-2}\tilde{\boldsymbol{f}}(1) \tag{21}$$
$$= 2\sum_{n=1}^{\infty} nB^n \tilde{\boldsymbol{f}}(1)$$

We have implemented a Python program for the numerical calculation of mean and variance of hitting times from the knowledge of transition matrix B alone. The Python code is shown below.

```python
import numpy as np
from scipy import *
from scipy.sparse import *

def get_mean_and_variance(transition_matrix):
    row, col = transition_matrix.shape
    assert(row == col)
    dimension = row
    row_indices = range(dimension)
    col_indices = [0] * dimension
    data = [1] * dimension
    ones = csr_matrix((data, (row_indices, col_indices)), shape = (dimension, 1))
    eps = 1.0e-13
    expectation = ones
    variance = csr_matrix(([0] * dimension, (row_indices, col_indices)), shape = (dimension, 1))
    power = ones
    expansion_order = 1000
    counter = 0
    while counter < expansion_order:
        counter += 1
        power = transition_matrix * power
        expectation += power
        variance += counter * power
        error = np.linalg.norm(expectation.todense())
        if error < eps:
            break
    variance = 2.0 * variance
    variance = variance + expectation - expectation.multiply(expectation)
    return expectation, variance
```

Higher order derivatives of Eq. [17] yield higher order moments. Now we have already developed the algorithm for calculating the moments of hitting probabilities using both analytical and numerical methods, next we will illustrate the effectiveness of this algorithm using both small and huge graphs.

The significance of the first order moment lies in that it is a measure of the distance from a starting node to a target node. If we already know that the target node is a fraudulent user in the social network, then we can infer that the nodes with average distance smaller than a threshold value could be considered to be potential fraudulent users. From intuition, we make a claim that the smaller the distance between any two nodes, the more similar they are to each other.

III. RESULTS

A. An analytical calculation of hitting time distribution on a simple graph

In this section, we will show how to calculate the hitting time distribution on a small graph using analytical methods. This graph contains five nodes, which are denoted as $0, 1, 2, 3, 4$, as shown in Fig. [1].
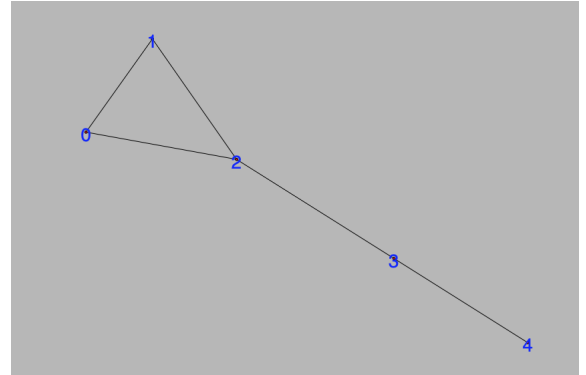

FIG. 1. An undirected graph that is small enough to be solved using analytical formulae. We use node 3 as our target node.

The Adjacency matrix of this graph is

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{22}$$

Our target node is 3, and we want to calculate the probability of hitting the target starting from each vertex for the first time after exactly n steps. Since node 4 is an adherent to target 3, we can directly write out its hitting

probability as

$$P(N_3 = n; 4) = \delta_{n,1} \tag{23}$$

For nodes 0, 1, 2, we can define the probabilities of starting from each node and ending at node 3 after exactly n steps. We encapsulate these probabilities into a column vector as

$$\boldsymbol{X}_n = \begin{pmatrix} P(N_3 = n; 0) \\ P(N_3 = n; 1) \\ P(N_3 = n; 2) \end{pmatrix} \tag{24}$$

The probability transition matrix is

$$B = \begin{pmatrix} 0 & 1/2 & 1/2 \\ 1/2 & 0 & 1/2 \\ 1/3 & 1/3 & 0 \end{pmatrix} \tag{25}$$

The probability vector satisfies this equation

$$\boldsymbol{X}_n = B\boldsymbol{X}_{n-1}, n \geq 2, \tag{26}$$

with initial condition

$$\boldsymbol{X}_1 = \begin{pmatrix} 0 \\ 0 \\ 1/3 \end{pmatrix} \tag{27}$$

The moment generating function for $\boldsymbol{X}_n$ is

$$\tilde{\boldsymbol{f}}(z) = z(I - zB)^{-1}\boldsymbol{X}_1 \tag{28}$$

$$= \frac{z}{3} \frac{1}{1 - \frac{7}{12}z^2 - \frac{z^3}{6}} \begin{pmatrix} \frac{z}{2} + \frac{z^2}{4} \\ \frac{z}{2} + \frac{z^2}{4} \\ 1 - \frac{z^2}{4} \end{pmatrix}$$

We can easily get the first order moments by differentiating the above equation with respect to z at $z = 1$, the results of which are

$$\tilde{\boldsymbol{f}}'(1) = \begin{pmatrix} \langle N_3 \rangle_0 \\ \langle N_3 \rangle_1 \\ \langle N_3 \rangle_2 \end{pmatrix} = \begin{pmatrix} 9 \\ 9 \\ 7 \end{pmatrix} \tag{29}$$

The above result means that the average distance from nodes 0 and 1 to node 3 are equal, both being 9, and the average distance from node 2 to node 3 is 7. We interpret these results as demonstrating that nodes 0 and 1 have equal distance to node 3, whereas node 2 is nearer to node 3 than both 1 and 2. Node 4, being an adherent to target node 3, always has an average distance 1 to the target. These results are consistent with our intuition and signify to us that node 4 is most susceptible to the influence of node 3, node 2 is second most susceptible, and nodes 0 and 1 are least susceptible to its influence.

In order to make a further test of the analytical results, we also perform a Monte Carlo simulation for the random walk on this graph. In the Monte Carlo program, we use node 3 as our target node, and start each random walk from node 0, 1, and 2. Each random walk terminates at

$\langle N_3 \rangle_i$	Exact results	Monte Carlo simulation	Numerical results
$\langle N_3 \rangle_0$	9	9.00323	8.99999999999999
$\langle N_3 \rangle_1$	9	8.96693	8.99999999999999
$\langle N_3 \rangle_2$	7	7.00013	7.000000000000002

TABLE I. Random walk results for the graph shown in Fig. [1]. $\langle N_t \rangle_i$ means the target node is t and the starting node is i. Here, $t = 3$. This table lists the expected number of steps for the random walker to reach target node 3 starting from nodes 0, 1, and 2. We obtain these results using three methods: analytical, Monte Carlo simulation, and numerical. Both Monte Carlo simulation results and numerical computation results are consistent with analytical results, although the numerical results have much higher precision, which justifies our introduction of the numerical algorithm for attacking this problem.

node 3 after some number of steps. By repeating this process thousands of times, we obtain the average number of steps required before the random walker finally reaches the target. For each node in the set $\{0, 1, 2\}$, we perform 10^6 random walks, each of which yields a step number, and then we calculate the mean value of these 10^6 numbers. The Monte Carlo simulation results we get are pretty similar to the analytical results, which are shown together in Table [III A]. We can see that the Monte Carlo simulation results are consistent with our analytical results, with relative errors being approximately 10^{-3}. We do not expect Monte Carlo simulation to give us high precision numerical results, and the final results of Monte Carlo simulations may vary slightly for different random number generators.

Using the algorithm outlined in the previous section, we can equally well compute the average step number starting from each node 0, 1, 2 using numerical methods. We can use either Eq. [19] or Eq. [20] for this purpose, because the probability transition matrix is small enough for the direct inversion of matrix to be feasible. However, Eq. [19] is no longer practical when the graph is large, and thus even for this small graph, we still prefer to use Eq. [20], where we need to calculate the sum of an infinite power series. Due to the quick convergence of this series, we artificially impose a cutoff condition that the summation should terminate if the norm of the summand vector is smaller than a pre-specified error limit ϵ, which we choose to be 10^{-13} here. We run the Python program listed in the previous section on macOS Mojave, and obtain numerical results that are shown together with analytical results and Mont Carlo simulation results in Table [III A]. It is clear that the numerical results obtained using our algorithm have a much higher precision than that of the Monte Carlo simulation results.

B. Numerical computation of hitting time distribution on large graphs

When dealing with large graphs, it is both tedious and impractical to get an analytical formula as Eq. [29]. In-

stead, we will resort to Eq. [20] to find numerical values of hitting times. Another method to find the hitting times is to use Monte Carlo simulation, although we will see that for a large graph, the running time of Monte Carlo simulation is much longer than the numerical method, thus making the Monte Carlo simulation an inferior alternative compared to Eq. [20]. In this section, we apply the numerical method and Monte Carlo simulation method to a connected graph as shown in Fig. [2].
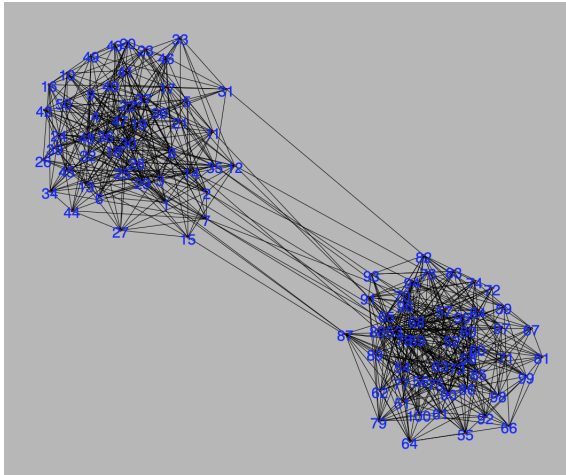

FIG. 2. An undirected graph with 100 edges and 740 edges. This graph contains two communities, and is generated by the rule that each pair of vertices within the same community is connected by an edge with probability $p_{\text{in}} = 0.3$, whereas each pair of vertices from different communities is connected by an edge with probability $p_{\text{out}} = 0.02$.

We will calculate the hitting times from each vertex in the graph to the target node which is chosen to be node 1. In order to visualize the results, we sort the vertices in the graph according to their hitting times to the target node. In Fig. [3], we plot the results from Monte Carlo simulation method and numerical method. We can see from the figure that these two methods give almost the same results, although we know that results from Monte Carlo simulation have a precision that is much lower that obtained from numerical method. Another weak point of Monte Carlo simulation is that it is much more time-consuming than the numerical method. In order to obtain the results shown in Fig. [3], we need to run 10^5 random walks from each vertex in the graph except the target node, and the whole process takes about 157 seconds, whereas in the numerical method, we only need to compute the power series in Eq. [20] up to 6180 terms, and it takes only 3.06 seconds to obtain results with machine precision. Actually, the larger the graph, the more time-saving the numerical method is compared to the Monte Carlo simulation method.

Another feature that is worth mentioning in Fig. [3] is that we can distinguish the two communities in the original graph by looking at the distribution of hitting times
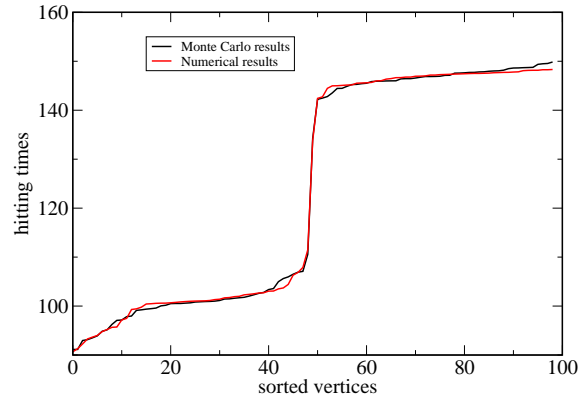

FIG. 3. An undirected graph with 100 edges and 740 edges. This graph contains two communities, and is generated by the rule that each pair of vertices within the same community is connected by an edge with probability $p_{\text{in}} = 0.3$, whereas each pair of vertices from different communities is connected by an edge with probability $p_{\text{out}} = 0.02$.

with respect to vertices. It is clear there is a transition region that connects two plateaus in the hitting time vs. sorted vertices curve. The two plateaus correspond to the two communities in the graph. We can interpret the emergence of these two plateaus by noting that a random walker that starts from within one of the two communities tend to get trapped in a community. Once the random walker gets trapped in a community, the hitting times for each vertex in the community would not change substantially from vertex to vertex, which gives rise to a plateau in the curve. However, as soon as the random walker finds a bridge leading from one community to another, it will make a rapid transition across the two communities, and the hitting times will also experience a significant change. Thus, the calculation of hitting times provides us a tool for community detection as a by-product. However, we should make a caveat that this method of community detection is usable only when the number of communities in the graph is small enough and the communities are clearly separated from each other. Or else, this method of community detection is not as good as the ones compiled in Ref. [5].

IV. CONCLUSION

In this paper, we have derived an analytical formula for calculating the hitting time from a starting vertex to a target vertex in a connected undirected graph. This method relies on the probability transition matrix that can be calculated conveniently from the graph's adjacency matrix. We also propose a quick method for implementing this formula using Python code, without the need to invert a possibly huge sparse matrix. Since hit-

ting time is a core concept in random walk which can directly be simulated using Monte Carlo method, we can obtain an approximate value of the hitting time using Monte Carlo simulation. We tested the validity of our formula by applying the analytical formula and Monte Carlo simulations to undirected connected graphs, and show that these two methods can give similar results within tolerance of error. The advantage of the analytical method over Monte Carlo simulation is that the former is much quicker and more accurate than the latter. Our calculation of the hitting times for vertices in a graph can also give us a glimpse of the community structures of the graph on which we perform random walks, although this method for detecting communities is not as good as other existing algorithms when the communities in graph are numerous and not clearly separated.

[1] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[2] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[3] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[4] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Phys. Rev. E*, 74:036104, Sep 2006.

[5] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[6] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[7] Bart Baesens, Veronique Van Vlasselaer, and Wouter Verbeke. *Fraud analytics using descriptive, predictive, and social network techniques: a guide to data science for fraud detection*. John Wiley & Sons, 2015.

[8] Peter G Doyle and J Laurie Snell. Random walks and electric networks. *arXiv preprint math/0001057*, 2000.

[9] Xiaojin Zhu, Zoubin Ghahramani, and John D Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International conference on Machine learning (ICML-03)*, pages 912–919, 2003.

[10] Sheldon M Ross, John J Kelly, Roger J Sullivan, William James Perry, Donald Mercer, Ruth M Davis, Thomas Dell Washburn, Earl V Sager, Joseph B Boyce, and Vincent L Bristow. *Stochastic processes*, volume 2. Wiley New York, 1996.