Homework II Report

Giulio Zabotto, s279487@studenti.polito.it

December 2020

1 Introduction

A random process is an indexed collection of random variables; it is denoted as $X(t)$ or $\{X(t)\}_{t\in T}$, where T is the index set, as random processes are normally used to model sequence of events over time. Formally,

$$X(t) : \Omega \to \mathcal{X} \quad t \in T.$$

Each random variable of the process has a common sample space Ω and a common state space $\mathcal{X}$. One can associate a random process $X(t)$ with a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, P)$: the set of nodes $\mathcal{V}$ coincides with the state space $\mathcal{X}$, the links in $\mathcal{E}$ coincide with the possible paths from a state to another, while the entries of the normalized weight matrix $P_{ij} = diag(\omega_i)^{-1}W_{ij}$ are the probabilities to evolve from state i to state j, where W is the weight matrix of the graph and $\omega_i = \sum_j W_{ij}$ is the out-degree of a node $i \in \mathcal{X}$.

A Markov chain is a random process. The peculiarity of the Markov chain is that the evolution of this kind of random processes must be independent from the past states, given the current state. This is the so-called *Markov* property

$$\mathbb{P}(X(t+1) = i_{t+1}|X(0) = i_0, ..., X(t) = i_t) = P_{i_t i_{t+1}}.$$

To determine the evolution of a Markov chain, it must be given the probability distribution vector at the initial condition $\overline{\pi}(0) = \mathbb{P}(X(0) = i) = p_i$. Then, it can be shown that we can track the probability distribution vector at any time $t \in T$ is $\overline{\pi}(t)' = p'P^t$.

The index set T may be discrete, $T = \mathbb{N}_0$ or continuous, $T = [0, \infty)$. In a continuous-time Markov chains, an agent moves from a state to another in a random interval of time. The usual distribution to model a interval of time between the random occurrence of two consequential events is the exponential distribution with the parameter λ that is the rate of the occurrence of the events. It is demonstrated that whenever the time intervals between occurrences is an exponential random variable, the number of occurrences is a Poisson process which follows the Poisson distribution with parameter λt where, where λ is the rate of the occurrence, t is the interval to count the number of occurrences in.

In such a setting, the Markov chain events are the changes from a state to another, that are realized whenever a an edge of the graph is crossed. This change of states happens when a link of a node opens one of its *gate* after a random interval of time. The amount of time between each *gate opening* is modeled as an exponential random variable, as we previously said. The a priori information needed to model the gate openings in is the transition rate of each link. Such information is gathered in the so-called *transition rate matrix* Λ; its entries are the rates at which an agent in node i may jump to state j. The graph $\mathcal{G}$ of a continuous-time Markov chain is defined as $\mathcal{G}_\Lambda = (\mathcal{X}, \mathcal{E}, \Lambda)$.

The exponential variable has the *memoryless* property, by which the probability of an arrival in t time is independent from the time already passed. This property is useful when simulating the time between gate opening as we do not have to keep into consideration the past intervals.

Markov chains may be seen from a node perspective as well; in fact, adding all the transition rate regarding one node yields the rate at which the node opens one of its gate to the out-neighbor nodes. Thus, each node can be seen as having an internal clock; each time it ticks, an agent will eventually move to another state of the Markov chain according to the transition probability matrix P whose entries are

$$P_{ij} = \frac{\Lambda_{ij}}{\omega_i} \quad i, j \in \mathcal{X}$$

where $\omega_i = \sum_j \Lambda_{ij}$ is the rate of node's clock.

An equivalent way to interpret a continuous-time Markov chain is to set a global Poisson clock to the whole graph. The rate of occurrence is set equal to the maximum rate among the rates of the individual nodes' clocks, that is the fastest rate of ticking,: $\omega_* = \max \omega_i$. In this perspective, the transition probability matrix $\overline{P}$ has entries

$$\overline{P}_{ij} = \frac{\Lambda_{ij}}{\omega_*}, \quad i \neq j, \quad \overline{P}_{ii} = 1 - \sum_{i \neq j} \overline{P}_{ij}$$

2 Problem I

We are going to simulate a single particle performing a continuous-time random walk on the the graph $\mathcal{G}$ as described by figure 1 and according to Λ transition rate matrix 1.

$$\Lambda = \begin{bmatrix} 0 & 2/5 & 1/5 & 0 & 0 \\ 0 & 0 & 3/4 & 1/4 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/3 & 0 & 2/3 \\ 0 & 1/3 & 0 & 1/3 & 0 \end{bmatrix} \tag{1}$$

where the order of nodes is o, a, b, c, d.

a) What is, according to the simulations, the average time it takes a particle that starts in node a to leave the node and then return to it?

The return time is defined as the time needed for a random walk to start from a given node and coming back. The return time for a random walk starting

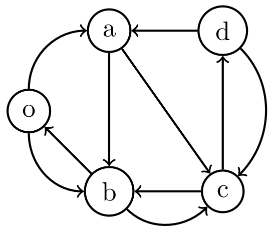

Figure 1: closed network in which particles move according to the transition rate matrix (1)

from node a is defined as $T_a^+ = \inf\{t \geq 0 : X(t) = i, X(s) \neq i, s \in (0,t)\}$ The expected return time is defined as the mean of the return time of every possible random walk starting from a when it first come back: $\mathbb{E}_a[T_a^+]$.

To have an estimate of the return time for the node a, we need to simulate random walks starting from a for m times, then averaging the return time of each run. We run $m = 1000$ experiments, the result of the simulations is

$$\frac{1}{m}\sum_{i=1}^{n} T_{a,i}^+ \approx 6.9750.$$

b) How does the result in a) compare to the theoretical return-time $\mathbb{E}_a[T_a^+]$*?*

It is demonstrated that given a random walk $X(t)$ on a strongly connected graph $\mathcal{G}$, $\overline{\pi}$ its stationary probability vector, then, for every $i \in \mathcal{X}$,

$$\mathbb{E}_i[T_i^+] = \frac{1}{\omega_i \overline{\pi}_i}.$$

The connectivity assumption holds for $\mathcal{G}$ in figure 1, the rate of the node ω_i is the sum of the i-th row of the transition rate matrix (1). What we need to compute is the stationary probability vector $\overline{\pi}$, that is defined as the probability distribution vector $\overline{\pi}$ satisfying the equations

$$L'\overline{\pi} = 0 \quad \mathbb{1}'\overline{\pi} = 1 \tag{2}$$

where L is the Laplacian matrix, $L = diag(\omega) - \Lambda$. The stationary probability vector can be shown that satisfies the relation

$$\overline{P}'\overline{\pi} = \overline{\pi}.$$

To compute $\overline{\pi}$, we can either solve the system of linear equations or compute the eigenvectors of $\overline{P}'$ and select the ones whose eigenvalue is unitary. By the Perron-Frobenius theorem, all the eigenvalues are less or equal to one, and if $\mathcal{G}$ is strongly connected, the dominant eigenvalue, the one that is unitary, is simple. This entails that the eigenvector that we want is the one with the maximum eigenvalue and we know that it is unique. Next, we normalize the eigevector found to satisfy the constraint of being a stochastic vector.

Implementing the latter method, the result yielded is

$$\mathbb{E}_i[T_i^+] = 6.75$$

The relation between the result in a) and the theoretical result is that the latter is the limit of the estimation,

$$\lim_{n\to+\infty} \frac{1}{n}\sum_{i=1}^{n} T_{a,i}^{+} = \mathbb{E}_a[T_a^+]$$

The simulations confirms the results; as n grows, the average return time tends toward the mean.

n	average return time
100	6.5133
1000	6.8742
10000	6.7565

c) What is, according to the simulations, the average time it takes to move from node o to node d?

The time a particle takes to reach a given state is called *hitting time*, that is defined as $T_i := \inf\{t \geq 0 : X(t) = i\}\ i \in \mathcal{X}$. To have an estimate of the expected hitting time $\mathbb{E}_o[T_d]$ [1] we simulated the random walks of particles starting from the node o until they reached node d for $m = 1000$ times. The experiments yielded

$$\frac{1}{m}\sum_{k=1} mT_{d,k} \approx 8.8822.$$

d) How does the result in c) compare to the theoretical hitting-time $\mathbb{E}_o[T_d]$?

Define $\mathcal{S}$ subset of $\mathcal{X}$ whose elements are reachable from every node/state $i \in \mathcal{X}$, it is demonstrated that

$$\mathbb{E}_i[T_\mathcal{S}] = 0 \quad if \quad i \in \mathcal{S}$$
$$\mathbb{E}_i[T_\mathcal{S}] = \frac{1}{\omega_i} + \sum_{j\in\mathcal{S}} P_{ij}\mathbb{E}_i[T_\mathcal{S}] \quad if \quad i \in \mathcal{X}\setminus\mathcal{S}$$

where $P = diag(\omega)^{-1}\Lambda$. Define τ as the vector of the expected hitting times, $\tau_i = \mathbb{E}_i[T_\mathcal{S}]$; the equations can be written as

$$\tau = \frac{1}{\omega} + P\tau$$

where, with abuse of notation, $\frac{1}{\omega}$ denotes a vector whose entries are the reciprocal of the entries of the out-degree vector. The solution τ is yielded by the system of linear equations

$$\tau = \frac{1}{\omega}(I - P)^{-1}$$

[1]The notation $\mathbb{E}_i$ indicates that the random walk starts at node i.

In this case, $\mathcal{S} = \{d\}$; solving the linear system we obtain,

$$\tau_d = \mathbb{E}_o[T_d] \approx 8.7857.$$

As seen in b), this result is the limit of the estimation of the hitting time in point c) as the table below shows the average hitting time getting closer to the theoretical expected hitting time.

n	average hitting time
100	8.0528
1000	9.0915
10000	8.8133

3 Problem II

We are going to simulate the random walks of particles on the same graph $\mathcal{G}$ (figure 1) with the same transition rate matrix Λ (1), but this time the number of particles simulated at once is one hundred. We are going the simulations following two different approaches. We will first assume the particle perspective, by which each particle is an agent that decides to which node will go next. In the second approach we will assume the perspective of the nodes; in this case the nodes decides how many particles access a gate/edge and at which rate.

3.1 Particle Perspective

If 100 particles all start at node a, what is the average time for a particle to return to node a?

The average return time yielded by the simulation is

$$\sum_{i=1}^{100} T^+_{a,i} \approx 6.9234 \tag{3}$$

How does this compare to the answer in Problem 1, why?

Both the results and the formulas that yield the results are the same. This is due to the fact that we assume that there is no interaction between the particles simulated simultaneously, hence simulating one hundred particles at once and simulating one particle for one hundred times is the same.

3.2 Node Perspective

If 100 particles start at node o, and the system is simulated for 60 time units, what is the average number of particles in the different nodes at the end of the simulation?

Define $n_i(t),\ i \in \mathcal{X}$ the number of particle for each state/node of the graph at time t. What we want to know is the average number of particles in each

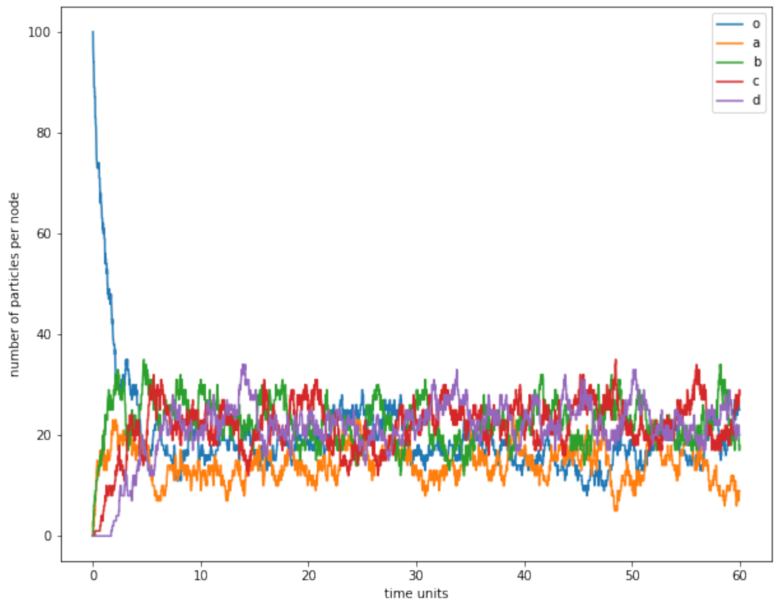

Figure 2: Simulation of the random walks of 100 particles starting from the node o of the graph $\mathcal{G}$ of figure 1

node at $t = 60$ in each node for m simulation:

$$\sum_{k=1}^{m} n_{i,k}(t) \quad i \in \mathcal{X} \tag{4}$$

We experimented with $m = 100$; the vector yielded by the computation is

$$\overline{n} = \begin{bmatrix} 18.58 & 14.45 & 22.14 & 22.7 & 22.13 \end{bmatrix}'.$$

Compare the simulation result in the first point above with the stationary distribution of the continuous-time random walk followed by the single particles.

The probability distribution vector $\overline{\pi}(t)$ describes the probability of finding the a particle in the different states. Since the graph is strongly connected, the following proposition holds

$$\lim_{t \to +\infty} \overline{\pi}(t) = \overline{\pi} \tag{5}$$

where $\overline{\pi}$ is the stationary probability vector as defined by the equations (2). This proposition implies that, given enough time, no matter what the initial probability distribution is, it tends to the stationary probability distribution.

An estimation of $\overline{\pi}(t)$ is the ratio between the number of particles in a node and the total number of particles among, hence, by dividing the average number of particles $\overline{n}$ by the number of particles, we obtain an approximation of the stationary probability vector:

$$\begin{aligned}\overline{n}/100 &\approx \left[\begin{array}{ccccc} 0.19 & 0.14 & 0.22 & 0.23 & 0.22 \end{array}\right]' \\ \overline{\pi} &\approx \left[\begin{array}{ccccc} 0.19 & 0.15 & 0.22 & 0.22 & 0.22 \end{array}\right]'\end{aligned}$$

As we can see by the results of the simulations, $\overline{n}/100$ suits the theoretical stationary distribution vector $\overline{\pi}$ with an error equal to

$$||\overline{n} - \overline{\pi}||_2 = 0.62\%.$$

4 Problem III

We are going to simulate random walks over the network in figure 3 with transition rate matrix (6). The peculiarity of the following simulations is that particles will enter the network through node o at the fixed rate $\lambda = 1$. The particles will then exit the network at the rate of the Poisson clock of the node d. The network is not strongly connected. Node o is a source, node d is a sink.

$$\Lambda = \begin{bmatrix} 0 & 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 2/4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6}$$

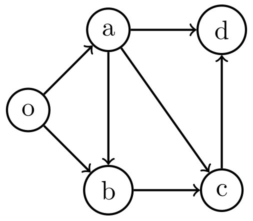

Figure 3: Open network with transition rate matrix defined by (6).

4.1 Proportional rate

In this simulation the rate of the Poisson clocks attached to each node of the graph will be equal to the number of particles in the node. We can see one simulation of this model in figure 4.

What is the largest input rate that the system can handle without blowing up?

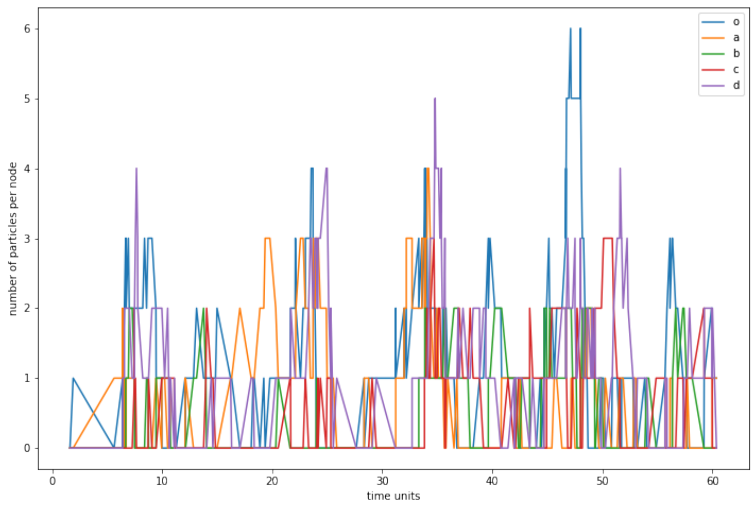

Figure 4: Simulation of the random walks of particles entering at node o at rate $\lambda = 1$. The nodes' clock rates are proportional to the number of particles in the node. The spikes in the graph are due to the fact that the transition from a node to another is instant.

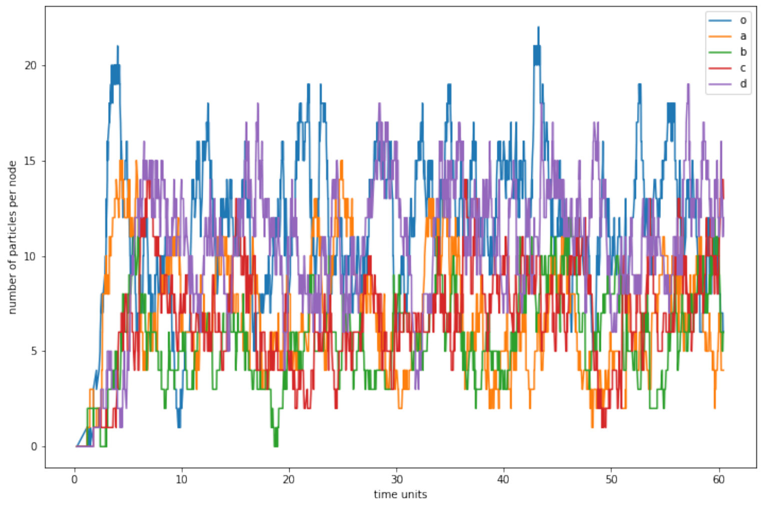

Figure 5: Simulation of the random walks of particles entering at node o at rate $\lambda = 10$. The nodes' clock rates are proportional to the number of particles in the node.

We expect that the there is no limit to the largest input rate that the system can handle; in fact, as the rate of each node is proportional to the number of particles in the node, there can be no node in which particles are stacking. The simulations of figures 5 and figure 6, respectively with input rate 10 and 100, confirms it. Moreover, the simulations show that the number of particles in each node averages around the input rate, which means that, on average, each node passes on a number of particles equal to the one gets in the network.

4.2 Fixed Rate

In this simulation the rate at which each node passes on particles to the others is fixed and equal to one. Figure 7 shows the result of one simulation.

What is the largest input rate that the system can handle without blowing up? Why is this different from the other case?

According to what we have seen in the previous paragraph, the system should handle any input rate that is equal to the rate of the Poisson clocks of the other nodes; any input rate above the unitary threshold should entail accumulation of particles in the source and sink nodes.

This hypothesis holds for input rate $\lambda > 1$ and $\lambda \leq 0.8$ as the table 1 of results show. Define $\overline{n}_i(t) = \frac{1}{m}\sum_{k=1}^{m} n_{i,k}(t)$ where $n_i(t)$ is the number of particle in node i at time t, m is the number of simulations. For $m = 100$,

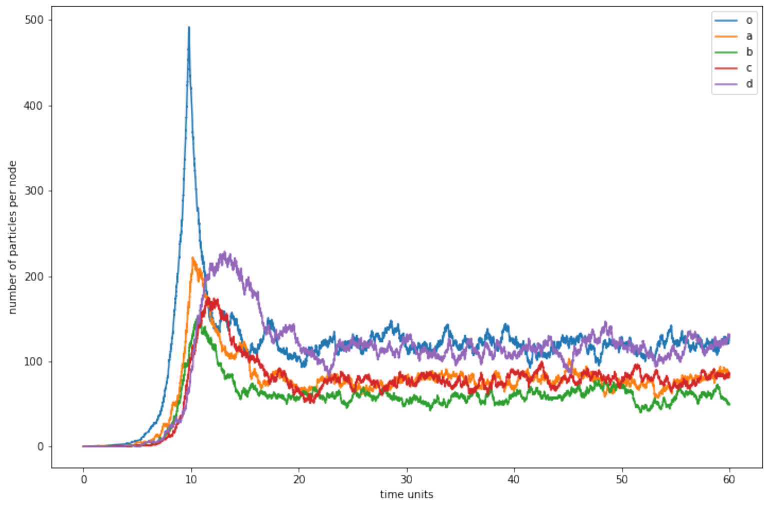

Figure 6: Simulation of the random walks of particles entering at node o at rate $\lambda = 100$. The nodes' clock rates are proportional to the number of particles in the node.

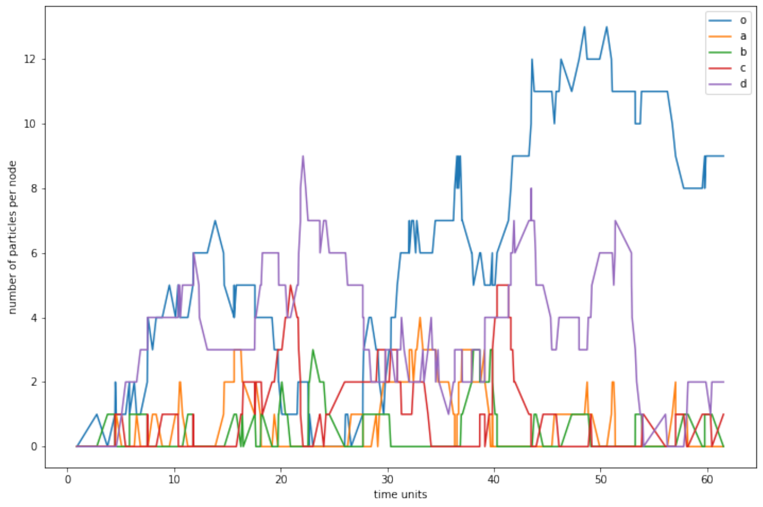

Figure 7: Simulation of the random walks of particles entering at node o at rate $\lambda = 1$. The nodes' clock rates are fixed and equal to 1.

nodes	o	a	b	c	d
$\lambda = 0.8$	5.45 ± 4	1.39 ± 2	0.8 ± 1	1.19 ± 2	3.41 ± 3
$\lambda = 1$	12.2 ± 8	1.75 ± 2	0.92 ± 1	1.67 ± 2	6.39 ± 5
$\lambda = 1.1$	16.59 ± 8	1.6 ± 2	1.13 ± 1	1.57 ± 2	5.93 ± 5

Table 1: Average number of particles and standard errors at t = 60 in 100 simulations

$t = 60$, we have that, on average, for $\lambda = 0.8$, the sink and the source node have the same order of magnitude of the other nodes; on the other hand, for $\lambda = 1.1$ we have that the source node is an order of magnitude higher than the other nodes. Our hypothesis is to be rejected when $\lambda = 1$, in fact in the source node we have on average an accumulation of particles of an order of magnitude higher than the others. Our hypothesis predicts that if the rate of arrival is equal to the rate of exit from a node, the node should have on average a number of particle equal to the rate of arrival, but this happens to be false for the source node.

5 Code listings

5.1 Problem I scripts

Compute the average return time of node a.

```
# np refers to the package numpy
# row-sum of the transition rate
w = np.sum(Lambda, axis=1)

# maximum transition rate
w_star = np.max(w)

# transition probability matrix in the
# system-wide perspective
# Lambda is the transition rate matrix
Q = Lambda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q, axis=1))
Q_cum = np.cumsum(Q, axis=1)

# number of simulations
n_sim = 1000

# array to store the return times of each simulation
return_times = np.zeros(n_sim)

starting_node = a
```

```
for i in range(n_sim):
    node = starting_node
    time = 0
    while True:
        node = np.argwhere(Q_cum[node] >
                           np.random.rand())[0]
        # argwhere retunrs arrays with shape (N, a.ndim)
        node = int(node)
        time = time - np.log(np.random.rand())/w_star
        if node == starting_node:
            return_times[i] = time
            break

average_return_time = np.mean(return_times)
```

Compute the stationary probability distribution

```
# Q is P_bar = Lambda / w_star
values, vectors = np.linalg.eig(Q.T)
# G is strongly connected, 1 eigenvelue is simple
index = np.argmax(values.real)
pi_bar = vectors[:,index].real
pi_bar = pi_bar/np.sum(pi_bar)
```

Compute the average hitting time

```
# array to store the return times of each simulation
hitting_times = np.zeros(n_sim)

starting_node = o
ending_node = d

n_sim = 1000

for i in range(n_sim):
    node = starting_node
    time = 0
    while True:
        node = int(np.argwhere(Q_cum[node] >
                   np.random.rand())[0])
        time = time - np.log(np.random.rand())/w_star
        if node == ending_node:
            hitting_times[i] = time
            break

average_hitting_time = np.mean(hitting_times)
```

Compute the expected hitting time

```
D = np.diag(w) # w is the out-degree vector
P = np.linalg.inv(D) @ Lambda

# Define the set S and the remaining nodes R
S = [d]
R = [node for node in range(len(G)) if node not in S]

# Restrict P to R x R to obtain hat(P)
hatP = P[np.ix_(R, R)]

# solve the linear system to obtain hat(x)
# np.linalg.solve solves a linear matrix equation
# given the coefficient matrix and the dependent
# variable values
hatx = np.linalg.solve((np.identity(len(hatP)) -
                        hatP),1/w[:d])

# define the hitting times to the set S
# hitting time is 0 if the starting node is in S
hitting_s = np.zeros(len(G))
# hitting time is hat(x) for nodes in R
hitting_s[R] = hatx
```

5.2 Problem II scripts

Compute return time under the particle perspective

```
n_particles = 100

starting_node = a

particle_positions = np.ones(n_particles, dtype=np.int8)
particle_times = np.zeros(n_particles)

# this mask will select the nodes that haven't
# come back to node a yet.Once every particle
# will have arrived, the mask array will be all
# flagged as False
mask = np.array([True] * n_particles)

# while any particle hasn't arrived yet
while any(mask) is True:
    # randomly select a particle
    i = np.random.choice(np.arange(n_particles)[mask])
    # get the current position of the particle
    node = particle_positions[i]
    # update the position
    node = np.argwhere(Q_cum[node] > np.random.rand())[0]
    particle_positions[i] = int(node)

    # get the current transition time of the particle
    time = particle_times[i]
    # update the transition time
    time = time - np.log(np.random.rand())/w_star
    particle_times[i] = time

    if node == starting_node:
        mask[i] = False

average_return_time = np.mean(particle_times)
```

Simulate the random walk in the node perspective

```
# row-sum of the transition rate
w = np.sum(Lambda, axis=1)

# maximum transition rate
w_star = np.max(w)

# transition probability matrix
Q = Lambda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q, axis=1))
Q_cum = np.cumsum(Q, axis=1)

n_particles = 100
time_units = 60
rate = n_particles # system-wide rate
n_nodes = len(G)
n_sim = 100

# matrix of the particle distribution among the
# nodes for each instant t
Delta = np.zeros((n_sim, n_nodes, 2*rate*time_units),
                 dtype=np.int8)
times = np.zeros(2*rate*time_units)

for i in range(n_sim):
    # 100 particles start at node o
    Delta[i, o, 0] = 100
    t = 0
    step = 0
    while t < time_units:
        t += -np.log(np.random.rand()) / rate
        step += 1
        times[step] = t
        prob = Delta[i, :, step-1] /
                        np.sum((Delta[i, :, step-1]))
        node = np.random.choice(np.arange(n_nodes),
                                p=prob)
        next_node = int(np.argwhere(Q_cum[node] >
                        np.random.rand())[0])
        Delta[i, next_node, step-1] += 1
        Delta[i, node, step-1] += -1
        Delta[i, :, step] = Delta[i, :, step-1]
```

5.3 Problem III scripts

Simulate the proportional rate system

```
# add_particle function adds a particles to the network
# when the clock of the input rate ticks
def add_particle(Delta, node, step):
    step += 1
    Delta[:, step] = Delta[:, step-1]
    Delta[node, step] += 1
    return Delta, step

# the update function updates the number of particles
# in the nodes every time the system-wide clock ticks.
# the function remove a particles from the node selected
# and adds it to the next node the particle will be sent
# to.
def update(Delta, step, Q_cum):
    n_particles = np.sum(Delta[:, step])
    if n_particles:
        step += 1
        Delta[:, step] = Delta[:, step-1]
        n_nodes = Delta.shape[0]
        prob = Delta[:, step] / n_particles
        node = np.random.choice(np.arange(n_nodes),
                                p=prob)
        Delta[node, step] += -1
        if node != d:
            next_node = int(np.argwhere(Q_cum[node] >
                            np.random.rand())[0])
            Delta[next_node, step] += 1
    return Delta, step
```

```
time_interval = 60
in_rate = 100
sim_steps = in_rate*time_interval*5
Delta = np.zeros((n_nodes, sim_steps), dtype=np.int16)
times = np.zeros(in_rate*time_interval*5)

step = 0
T = -np.log(np.random.rand()) / in_rate
t = T
times[step] = t
while T < time_interval:
    Delta, step = add_particle(Delta, o, step)
    rate = np.sum(Delta[:, step])
    t += -np.log(np.random.rand()) / rate
    T += -np.log(np.random.rand()) / in_rate
    times[step] = t
    if t >= T:
        Delta, step = update(Delta, step, Q_cum)
        times[step] = t
    while t < T:
        Delta, step = update(Delta, step, Q_cum)
        t += -np.log(np.random.rand()) / rate
        times[step] = t
```

Simulate the system at fixed rate

```
def add_particle(Delta, node, step):
    step += 1
    Delta[:, step] = Delta[:, step-1]
    Delta[node, step] += 1
    return Delta, step

def update(Delta, step, active_clocks, Q_cum):
    n  = np.sum(Delta[:, step])
    if n:
        step += 1
        Delta[:, step] = Delta[:, step-1]
        node = np.random.choice(active_clocks)
        Delta[node, step] += -1
        if node != d:
            next_node = int(np.argwhere(Q_cum[node] >
                                np.random.rand())[0])
            Delta[next_node, step] += 1
    return Delta, step

time_interval = 60
in_rate = 1
```

```
sim_steps = time_interval*10

n_sim = 1000
last_pi = np.zeros((n_sim,n_nodes))

for i in range(n_sim):
    Delta = np.zeros((n_nodes, sim_steps),
                      dtype=np.int64)
    times = np.zeros(sim_steps)
    step = 0
    T = -np.log(np.random.rand()) / in_rate
    t = T

    times[step] = t
    while T < time_interval:
        Delta, step = add_particle(Delta, o, step)
        # active clocks = indices of the nodes with at least
        # one particle in
        active_clocks = np.nonzero(Delta[:, step])[0]
        # system-wide rate proportional to the number of
        # active clocks
        rate = active_clocks.shape[0]
        T += -np.log(np.random.rand()) / in_rate
        if rate:
            t += -np.log(np.random.rand()) / rate
            times[step] = t
        else:
            continue
        if t >= T and t < time_interval:
            active_clocks = np.nonzero(Delta[:, step])[0]
            Delta, step = update(Delta, step,
                                 active_clocks, Q_cum)
            times[step] = t
        while t < T:
            active_clocks = np.nonzero(Delta[:, step])[0]
            rate = active_clocks.shape[0]
            if rate:
                Delta, step = update(Delta, step,
                                     active_clocks, Q_cum)
                t += -np.log(np.random.rand()) / rate
                times[step] = t
            else:
                break
```