Network Dynamics and Learning
Homework 2 report

Tommaso Monopoli
s278727

Please refer to the appendix A at the end of this document for all the Python codes used to solve the problems.

Problem 1

The first part of this assignment consists in studying a single particle performing a continuous-time random walk in the network described by the graph in fig. 1, according to the following transition rate matrix.

$$\Lambda = \begin{array}{c} \begin{matrix} o & a & b & c & d \end{matrix} \\ \begin{pmatrix} 0 & 2/5 & 1/5 & 0 & 0 \\ 0 & 0 & 3/4 & 1/4 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/3 & 0 & 2/3 \\ 0 & 1/3 & 0 & 1/3 & 0 \end{pmatrix} \end{array} \begin{matrix} o \\ a \\ b \\ c \\ d \end{matrix} \tag{1}$$

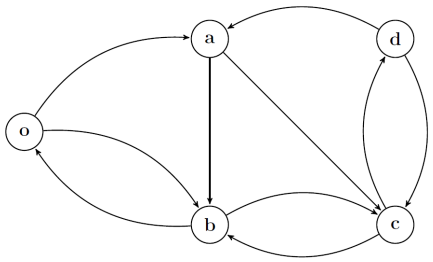

Figure 1: Closed network in which particles move according to the transition rate matrix (1)

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 particle to be simulated performs a continuous-time random walk in the network in fig 1, according to the transition matrix Λ defined in (1) and starting from node a. As we know from theory, this random walk can be modeled as a continuous-time Markov chain $X(t)$ with transition matrix Λ and initial condition $X(0) = a$.
There are several equivalent ways to define a continuous-time Markov chain. For setting up the simulations, I find it useful to refer to the one involving a single, system-wide rate-ω_* Poisson clock, where $\omega_* = \max_i \omega_i = 1$ and $\omega = \Lambda \mathbb{1} = (3/5, 1, 1, 1, 2/3)$. Every time this Poisson clock ticks, the particle jumps to another node (or stays in the node it is on), with transition probabilities given by the row of $\overline{P}$ associated to the node in which the particle is currently on. $\overline{P}$ is defined as:

$$\begin{cases} \overline{P}_{ij} = \frac{\Lambda_{ij}}{\omega_*} & \text{if } i \neq j \\ \overline{P}_{ii} = 1 - \sum_{j \neq i} \overline{P}_{ij} \end{cases}$$

Therefore:

$$\overline{P} = \begin{array}{c} \begin{matrix} o & a & b & c & d \end{matrix} \\ \begin{pmatrix} 2/5 & 2/5 & 1/5 & 0 & 0 \\ 0 & 0 & 3/4 & 1/4 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/3 & 0 & 2/3 \\ 0 & 1/3 & 0 & 1/3 & 1/3 \end{pmatrix} \end{array} \begin{matrix} o \\ a \\ b \\ c \\ d \end{matrix} \tag{2}$$

This is equivalent to say that when the particle is in node i it will stay $1/\omega_i$ time units there before moving to one of the out-neighbors of i. The next node the particle will visit is based on the probability matrix $P = \text{diag}(\omega)^{-1}\Lambda$.

The simulation stops when the particle has left node a and has returned to it. The output of a single simulation is the sequence of nodes visited by the particle during its random walk and the sequence of time instants in which the particle jumped to another node. Clearly, the last entry of this time sequence is the return time to a, $T_a^+ = \inf\{t > 0 \mid X(t) = i \text{ and } X(s) \neq i \text{ for some } s \in (0,t)\}$

The required *"average time it takes a particle that starts in node a to leave the node and then return to it"* is an estimation of the expected return time to a, $\mathbb{E}_a[T_a^+]$. To compute this estimate, we can perform a certain number n of simulations, thus obtaining a sample of n return times to a, $(T_a^{+(1)}, \ldots, T_a^{+(n)})$, and then compute the sample mean $\overline{T_a^+} = \frac{1}{n}\sum_{i=1}^{n} T_a^{+(i)}$

For a sufficiently large n, $\overline{T_a^+}$ well approximates $\mathbb{E}_a[T_a^+]$, and therefore is a good estimate for it.

After performing $n = 1000$ simulations, retrieving a sample of 1000 return times to a and averaging them, the resulting sample mean is $\overline{\mathbf{T_a^+}} \approx \mathbf{6.559}$ time units.

b) *How does the result in a) compare to the theoretical return time $\mathbb{E}_a[T_a^+]$? (Include a description of how this is computed.)*

As stated before, for sufficiently large values of n, $\overline{T_a^+}$ well approximates $\mathbb{E}_a[T_a^+]$. I verify it by computing $\mathbb{E}_a[T_a^+]$ analytically. This can be done simply by using Kac's Formula in continuous time:

$$\mathbb{E}_a[T_a^+] = \frac{1}{\omega_a \overline{\pi}_a}$$

where $\omega_a = 1$ and $\overline{\pi}$ is the stationary probability distribution (Laplace-invariant probability distribution of the graph G_Λ having Λ as weight matrix: $L'\overline{\pi} = (\text{diag}(\omega) - \Lambda)'\overline{\pi} = 0$ and $\mathbb{1}'\overline{\pi} = 1$; $\overline{\pi}$ is unique, since G_Λ is strongly connected).

We can prove that $L'\overline{\pi} = 0 \Leftrightarrow \overline{\pi} = \overline{P}'\overline{\pi}$, so we can compute $\overline{\pi}_a$ analytically by solving the system of linear equations:

$$\begin{cases} \overline{\pi} = \overline{P}'\overline{\pi} \\ \mathbb{1}'\overline{\pi} = 1 \end{cases} \Leftrightarrow \begin{cases} \overline{\pi}_o = \frac{2}{5}\overline{\pi}_o + \frac{1}{3}\overline{\pi}_b \\ \overline{\pi}_a = \frac{2}{5}\overline{\pi}_o + \frac{1}{3}\overline{\pi}_d \\ \overline{\pi}_b = \frac{1}{5}\overline{\pi}_o + \frac{3}{4}\overline{\pi}_a + \frac{1}{3}\overline{\pi}_c \\ \overline{\pi}_c = \frac{1}{4}\overline{\pi}_a + \frac{1}{2}\overline{\pi}_b + \frac{1}{3}\overline{\pi}_d \\ \overline{\pi}_d = \frac{2}{3}\overline{\pi}_o + \frac{1}{3}\overline{\pi}_o \\ \overline{\pi}_o + \overline{\pi}_a + \overline{\pi}_b + \overline{\pi}_c + \overline{\pi}_d = 1 \end{cases} \Leftrightarrow \cdots \Leftrightarrow \begin{cases} \overline{\pi}_o = \frac{5}{27} = 0.1852 \\ \overline{\pi}_a = \frac{4}{27} = 0.1481 \\ \overline{\pi}_b = \frac{2}{9} = 0.2222 \\ \overline{\pi}_c = \frac{2}{9} = 0.2222 \\ \overline{\pi}_d = \frac{2}{9} = 0.2222 \end{cases} \tag{3}$$

Therefore, $\mathbb{E}_a[T_a^+] = 27/4 = \mathbf{6.75}$ time units, a value very close to the previously obtained average return time to a. One may argue that further increasing n will make the average return time to a even closer to the expected return time to a.

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

Now a simulation consists in a particle performing a random walk starting from node o. The simulation stops when the particle hits node d.

The output of a single simulation is again the sequence of nodes visited by the particle during its random walk and the sequence of time instants in which the particle jumped to another node. Clearly, the last entry of this time sequence is the hitting time from o to d, $T_d = \inf\{t \geq 0 \mid X(t) = d\}$, with $X(0) = o$
The required *"average time it takes a particle to move from node o to node d"* is an estimation of the expected hitting time from o to d, $\mathbb{E}_o[T_d]$. As before, to compute this estimate, we can perform a certain number n of simulations, thus obtaining a sample of n hitting times from o to d, $(T_d^{(1)}, \dots, T_d^{(n)})$, and then compute the sample mean $\overline{T_d} = \frac{1}{n} \sum_{i=1}^{n} T_d^{(i)}$
For a sufficiently large n, $\overline{T_d}$ well approximates $\mathbb{E}_o[T_d]$, and therefore is a good estimate for it.

After performing $n = 1000$ simulations, retrieving a sample of 1000 hitting times from o to d and averaging them, the resulting sample mean is $\overline{\mathbf{T_d}} \approx \mathbf{8.579}$ time units.

d) *How does the result in c) compare to the theoretical hitting time $\mathbb{E}_o[T_d]$? (Include a description of how this is computed.)*

As stated before, for sufficiently large values of n, $\overline{T_d}$ well approximates $\mathbb{E}_o[T_d]$.
I verify it by computing $\mathbb{E}_o[T_d]$ analytically. This can be done simply by solving the system:

$$\begin{cases} \mathbb{E}_d[T_d] = 0 \\ \mathbb{E}_i[T_d] = \frac{1}{\omega_i} + \sum_j P_{ij}\mathbb{E}_j[T_d] \quad \text{if } i \neq d \end{cases} \Leftrightarrow \begin{cases} \mathbb{E}_o[T_d] = \frac{5}{3} + \frac{2}{3}\mathbb{E}_a[T_d] + \frac{1}{3}\mathbb{E}_b[T_d] \\ \mathbb{E}_a[T_d] = 1 + \frac{3}{4}\mathbb{E}_b[T_d] + \frac{1}{4}\mathbb{E}_c[T_d] \\ \mathbb{E}_b[T_d] = 1 + \frac{1}{2}\mathbb{E}_o[T_d] + \frac{1}{2}\mathbb{E}_c[T_d] \\ \mathbb{E}_c[T_d] = 1 + \frac{1}{3}\mathbb{E}_b[T_d] + \frac{2}{3}\mathbb{E}_d[T_d] \\ \mathbb{E}_d[T_d] = 0 \end{cases} \Leftrightarrow \cdots \Leftrightarrow \begin{cases} \mathbb{E}_o[T_d] = \frac{123}{14} \\ \mathbb{E}_a[T_d] = \frac{50}{7} \\ \mathbb{E}_b[T_d] = \frac{99}{14} \\ \mathbb{E}_c[T_d] = \frac{47}{14} \\ \mathbb{E}_d[T_d] = 0 \end{cases}$$

Therefore, $\mathbb{E}_o[T_d] = 123/14 \approx \mathbf{8.786}$ time units, a value very close to the previously obtained average hitting time from o to d. One may argue that further increasing n will make the average hitting time from o to d even closer to the expected hitting time from o to d.

Problem 2

The second part of this assignment consists in studying 100 particles simultaneously performing continuous-time random walks in the same network of problem 1. Each of the 100 particles in the network will move around just as the single particle moved around in problem 1: the time each of them will stay in a node is exponentially distributed, and on average they will stay $1/\omega_i$ time-units in a node i before moving to one of their out-neighbors. The next node they will visit is based on the probability matrix $P = \text{diag}(\omega)^{-1}\Lambda$.

The system needs to be simulated from two different perspectives:

- **Particle perspective**, i.e. "follow the particles": we are interested in the sequence of nodes visited by each of the 100 particles, keeping track also of the time instants at which each particle jumps. To simulate the particle perspective I prefer to implement a single, system-wide Poisson clock (as in problem 1) with rate $100\omega_* = 100$: every time this Poisson clock ticks, we choose at random (uniformly) a particle to move, among the 100; this particle is then moved to a neighbor node (which can be itself) according to the transition probability matrix $\overline{P}$.
- **Node perspective**, i.e. "observe the nodes": we are interested in keeping track of the number of particles sojourning on each node over time. To simulate the node perspective I implement again a single, system-wide Poisson clock with rate $100\omega_* = 100$. Then, at each tick we randomly, and proportionally to the number of particles in the different nodes, select a node from which we should move a particle. If at least one particle is present on that node, then a particle from the selected node will move according to $\overline{P}$ (else, nothing happens). By doing so, each node will pass along particles at a rate proportional to the number of particles in it: if at time t the number of particles in node i is $n_i(t)$, it will pass along particles at a rate of $n_i(t)\omega_i$.

Both the solutions are equivalent to attaching a Poisson clock of rate $\omega_* = 1$ to each particle, and moving it according to $\overline{P}$ every time its clock ticks.

a) Particle Perspective

a.i) If 100 particles all start in node a, what is the average time for a particle to return to node a?

The required *"average time for a particle to return to node a"* is an estimation of the expected minimum return time to a, i.e. the expected minimum time needed for at least one particle among the 100 of a simulation to return to a after having left it. Mathematically, we have to estimate $\mathbb{E}_a[\tau_a^+]$ where $\tau_a^+ = \min(T_{a,1}^+, \ldots, T_{a,100}^+)$ and $T_{a,i}^+$ is the return time to a of the i-th particle of the simulation.

A simulation consists in 100 particles all starting from node a and performing a random walk, as described above. The simulation stops when a particle has left node a and has returned to it. The output of a single simulation is the time instant in which a particle has returned to a.

To compute the required estimate, we can perform a certain number n of simulations, thus obtaining a sample of n minimum return times to a, $(\tau_a^{+(1)}, \ldots, \tau_a^{+(100)})$ and then compute the sample mean $\overline{\tau_a^+} = \frac{1}{n}\sum_{i=1}^{n} \tau_a^{+(i)}$.
For a sufficiently large n, $\overline{\tau_a^+}$ well approximates $\mathbb{E}_a[\tau_a^+]$, and therefore is a good estimate of it.

After performing $n = 1000$ simulations, retrieving a sample of 1000 minimum return times to a and averaging them, the resulting sample mean is $\overline{\tau_{\mathbf{a}}^+} \approx \mathbf{0.679}$ time units.

a.ii) How does this compare to the answer in Problem 1, and why?)

$\overline{\tau_a^+} \approx 0.679 < \overline{T_a^+} \approx 6.559$. This result makes sense, as we are selecting the minimum return time to a among 100 particles, which will be less than or equal to all the return times to a achieved by the 100 particles, as $\min(a, b) \leq a$ and $\min(a, b) \leq b$, where $a, b \in \mathbb{R}$. Maybe it can be proved that, given $X_1, \ldots, X_n$ i.i.d. random variables with mean $\mathbb{E}[X_i] = m$, it is $\mathbb{E}[\min(X_1, \ldots, X_n)] \leq m$.

b) Node Perspective

b.i) If 100 particles start in 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?

The required *"average number of particles in the different nodes at the end of the simulation"* is an estimation of the expected distribution of the 100 particles over the nodes of the network at $t = 60$. Mathematically, we have to estimate $(\mathbb{E}[n_o(60)], \mathbb{E}[n_a(60)], \ldots, \mathbb{E}[n_d(60)])$.

A simulation consists in 100 particles all starting from node o and performing a random walk. The simulation stops at $t = 60$. The output of a single simulation is the sequence of the different distributions of the 100 particles in the network over time, along with the time instants in which the distribution changes (because one of the particles jumped to another node).

Therefore, we can simulate the system with 100 particles starting from node o for $t = 60$ time units, and retrieve the final distribution of particles over the nodes at the end of the simulation.

To compute the required estimate, we can perform a certain number n of simulations, thus obtaining a sample of n particle distributions at $t = 60$, $(d^{(1)}, \ldots, d^{(100)})$, and then compute the sample mean $\overline{d} = \frac{1}{n}\sum_{i=1}^{n} d^{(i)}$.

After performing $n = 200$ simulations, the resulting average distribution at $t = 60$ is $(18.56, 14.69, 22.43, 22.17, 22.16)$; therefore on average at $t = 60$ we will find on the five nodes o, a, b, c, d 19, 15, 22, 22, 22 particles respectively.

b.ii) Illustrate the simulation above with a plot showing the number of particles in each node during the simulation time.

The plot in fig. 2 refers to the first simulation, among the 200 performed.

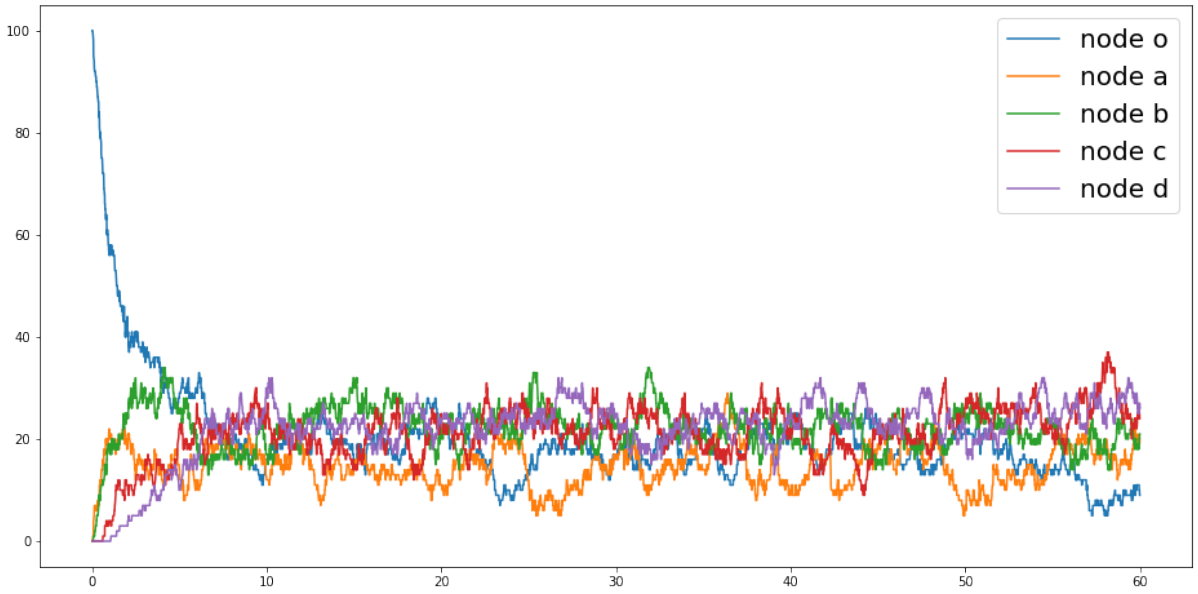

Figure 2: Distribution of particles in each node over the simulation time. x-axis: t, y-axis: $n_i(t)$ (this convention for the axes is followed in all the plots of this report)

We can see that after little time, the numbers of particles in each node begin to oscillate around the average values found above.

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

We can notice that $\overline{d}$ computed before is approximately $100\overline{\pi}$ (where $\overline{\pi}$ was computed in (3)).

This behavior was expected: since G_Λ is strongly connected, $\overline{\pi}_i$ tells us the time spent on average in node i by a particle performing a random walk on the network (this is an application of the ergodic theorem in continuous-time), or equivalently the probability that the particle is in node i, at a sufficiently large time instant ($\overline{\pi}(t) \to \overline{\pi}$ as $t \to \infty$). If the particles in the network are instead n, we can interpret $\overline{\pi}_i$ as the fraction of particles which are expected to be in node i, at a sufficiently large time instant. Therefore, if $n = 100$, we expect to find a number of particles in node i given by $100\overline{\pi}_i$, if the simulation lasts long enough.

Problem 3

In this part we study how different particles affect each other when moving around in a network in continuous time. We consider the open network of fig. 3, with transition rate matrix Λ_{open} according to (4).

$$\Lambda_{\text{open}} = \begin{array}{c} \begin{matrix} o & a & b & c & d \end{matrix} \\ \begin{pmatrix} 0 & 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 2/4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} o \\ a \\ b \\ c \\ d \end{matrix} \tag{4}$$

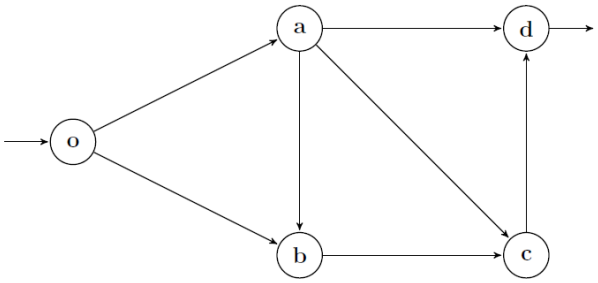

Figure 3: Open network in which particles move according to the transition matrix (4)

For this system, particles will enter the open network at node o according to a Poisson process with rate $r_{\text{input}} = 1$, which we will call *input rate*. Each node will then pass along a particle according to a given rate. Particles which are in node d can exit the open network on their next jump. We will simulate two different scenarios that differ by which rate the nodes will pass along particles:

- **proportional rate**: each node i will pass along particles according to a Poisson process with rate $r_i = n_i(t)$, i.e. equal to the number of particles in that node.
- **fixed rate**: each node i will pass along particles with a fixed rate $r_i = 1$.

Here I describe some important implementation details.
I found it easier to think of this open network as a closed network by adding two auxiliary nodes, o' and d', and two additional directed edges (o', o) and (d, d'). The sequence of time instants at which a particle enters the open network in o ("entrance times") is computed before starting the simulation; the number of particles N which will enter the open network during the 60 time units of the simulation is equal to the length of the sequence of entrance times. The simulation starts with N particles in node o'; these particles will enter one at a time in the open network (through node o) at each entrance time. At this point:

- for proportional rate, I implement a system-wide rate-N Poisson clock. Every time this clock ticks, one of the seven nodes of the closed network is chosen at random; each node has a probability of being chosen proportional to the nodes which sojourn on it in that time instant (so empty nodes are not chosen). If the chosen node is o' or d', nothing happens; if it is o, a, b or c then a particle will leave that node and jump to another node of the open network according to the probabilities in Λ_{open}; if that node is d then a particle will leave d and jump to d'. In this way each node of the open network will pass along particles at a rate $n_i(t)$. This solution is equivalent to implementing a rate-$n_i(t)$ Poisson clock for each node of the open network.
- for fixed rate, I implement a system-wide rate-5 Poisson clock. Every time this clock ticks, one of the five nodes of the open network is chosen. If the chosen node has at least one particle on it, a particle will leave that node and jump to a neighboring node, according to the probabilities in Λ_{open}. In this way each node of the open network will pass along particles at a fixed rate 1. This solution is equivalent to implementing a rate-1 Poisson clock for each node of the open network.

Please refer to the appendix A for the code.

a) Proportional rate

a.i) Simulate the system for 60 time units and plot the evolution of the number of particles in each node over time.

The output of the simulation is the sequence of the different distributions of particles in the open network over time, along with the time instants in which the distribution changes (because either a particle in the open network jumped to another node of the open network, or a new particle entered the open network, or a particle exited the open network).

The requested plot is in fig. 4

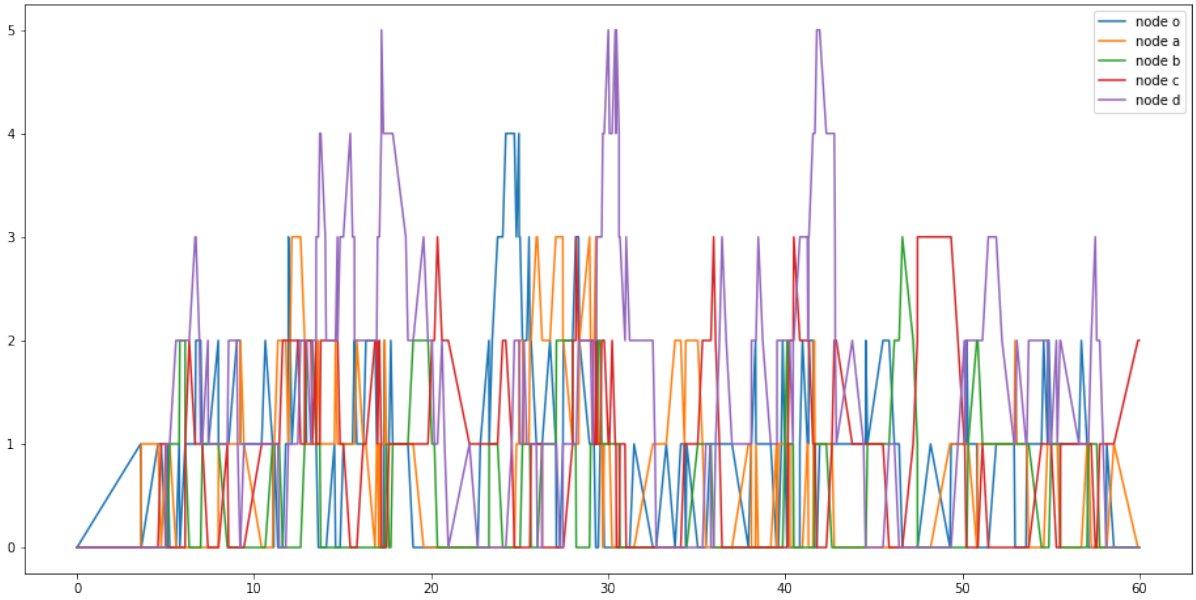

Figure 4: Evolution of the distribution of particles on each node over time, $r_{\text{input}} = 1$

There is not a noticeable accumulation of particles in the nodes over time. The reason for this is argued in the answer to the next point.

a.ii) What is the largest input rate that the system can handle without blowing up?

I assume that the system "blows up" when there is at least a node i of the open network which accumulates an infinite number of particles as $t \to \infty$.

I performed the simulation again with input rates $r_{\text{input}} = 10$ (fig. 5) and $r_{\text{input}} = 100$ (fig. 6).

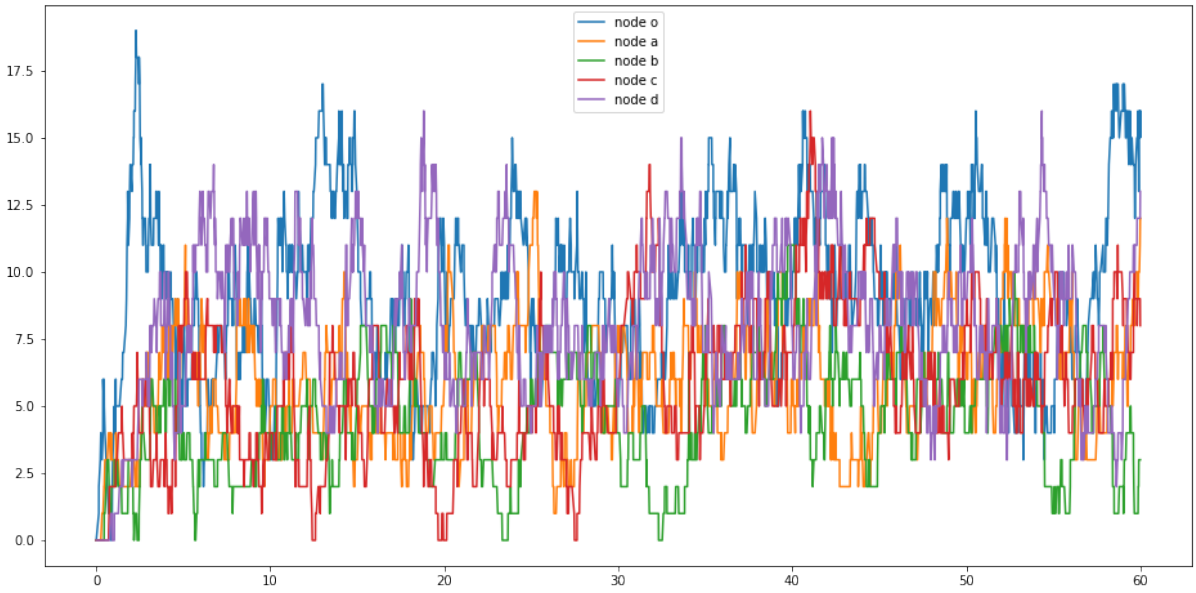

Figure 5: Evolution of the distribution of particles on each node over time, $r_{\text{input}} = 10$

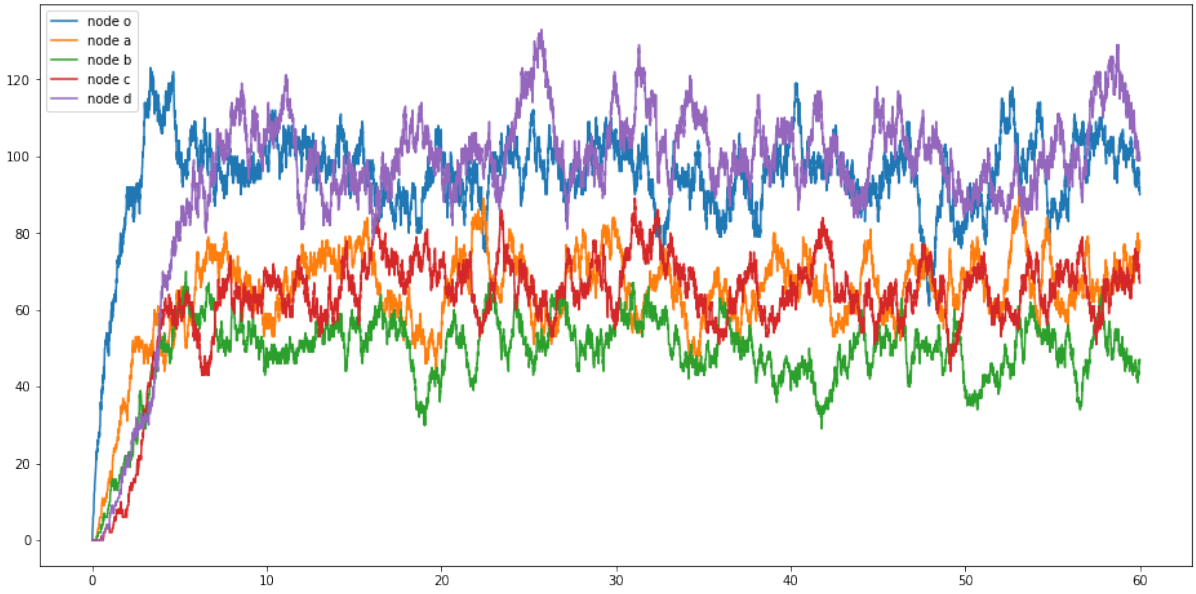

Figure 6: Evolution of the distribution of particles on each node over time, $r_{\text{input}} = 100$

Again, as the simulation goes on there is not a single node which increasingly accumulates particles over time: $n_i(t)$ is bounded, in the sense that $\forall t_1\ \exists t_2 > t_1$ s.t. $n_i(t_2) < n_i(t_1)$, whatever the input rate. This is because the rate of the Poisson clock of each node is equal to the number of particles in it: the more particles sojourn on node i, the higher will be r_i, so the faster particles will leave node i. Therefore, we argue that the minimum input rate for which the system blows up is $r_{\text{input}} = +\infty$ (i.e., the system will never blow up).

b) Fixed rate

b.i) Simulate the system for 60 time units and plot the evolution of the number of particles in each node over time.

The requested plot is in fig. 7

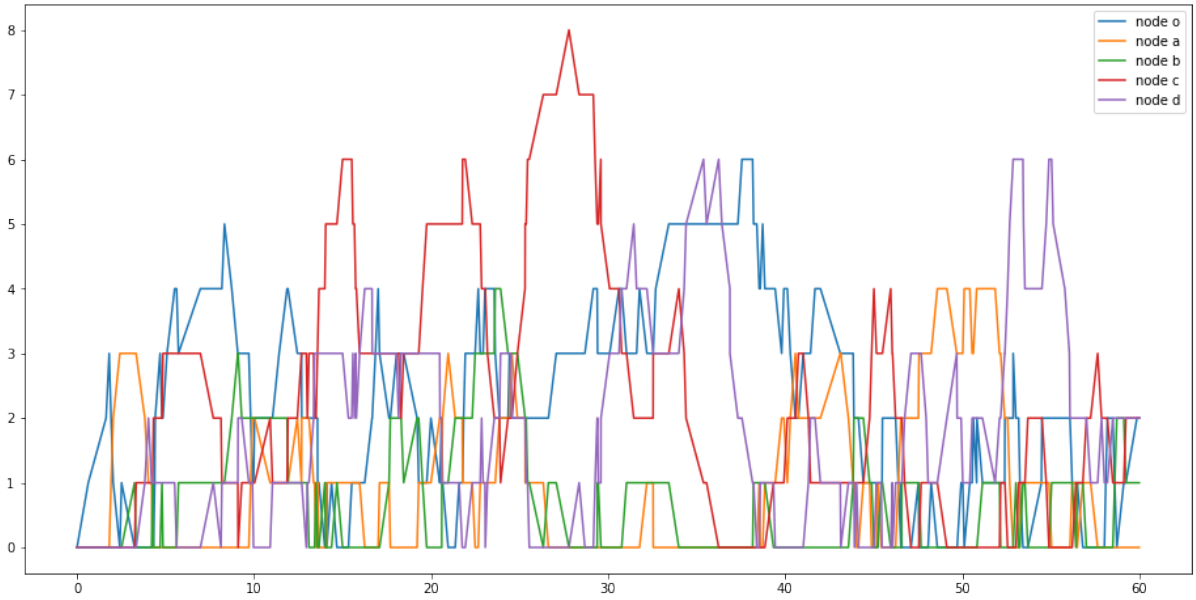

Figure 7: Evolution of the distribution of particles on each node over time, $r_{\text{input}} = 1$

There is not a noticeable accumulation of particles in the nodes over time. The reason for this is argued in the answer to the next point.

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

I performed the simulation again with input rates $r_{\text{input}} = 1.1$ (fig. 8) and $r_{\text{input}} = 1.2$ (fig. 9).

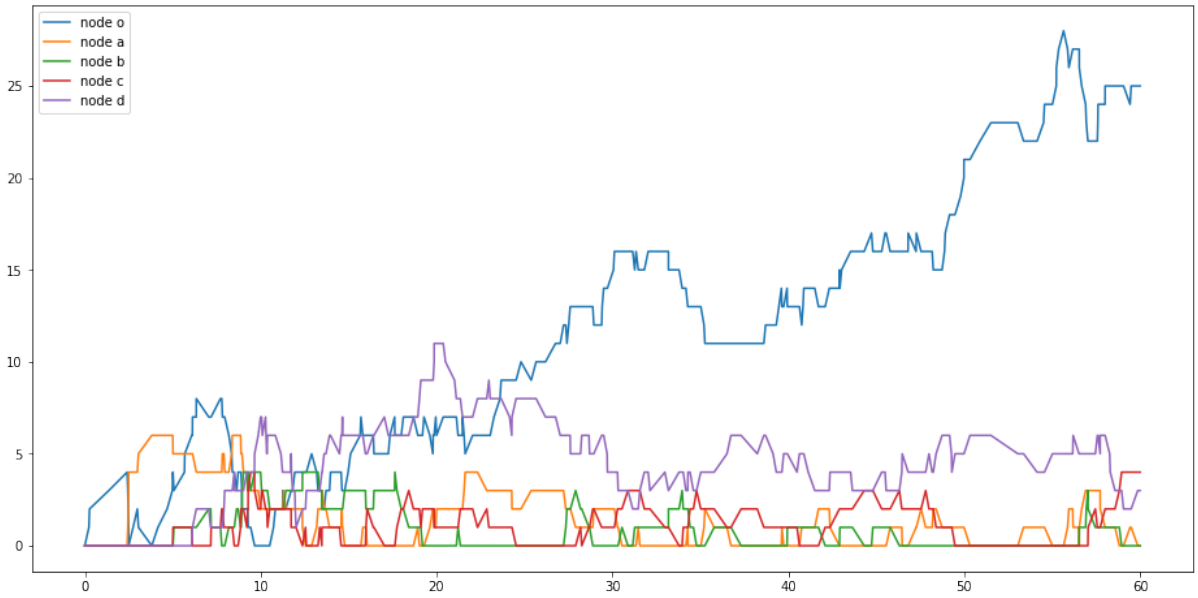

Figure 8: Evolution of the distribution of particles on each node over time, $r_{\text{input}} = 1.1$

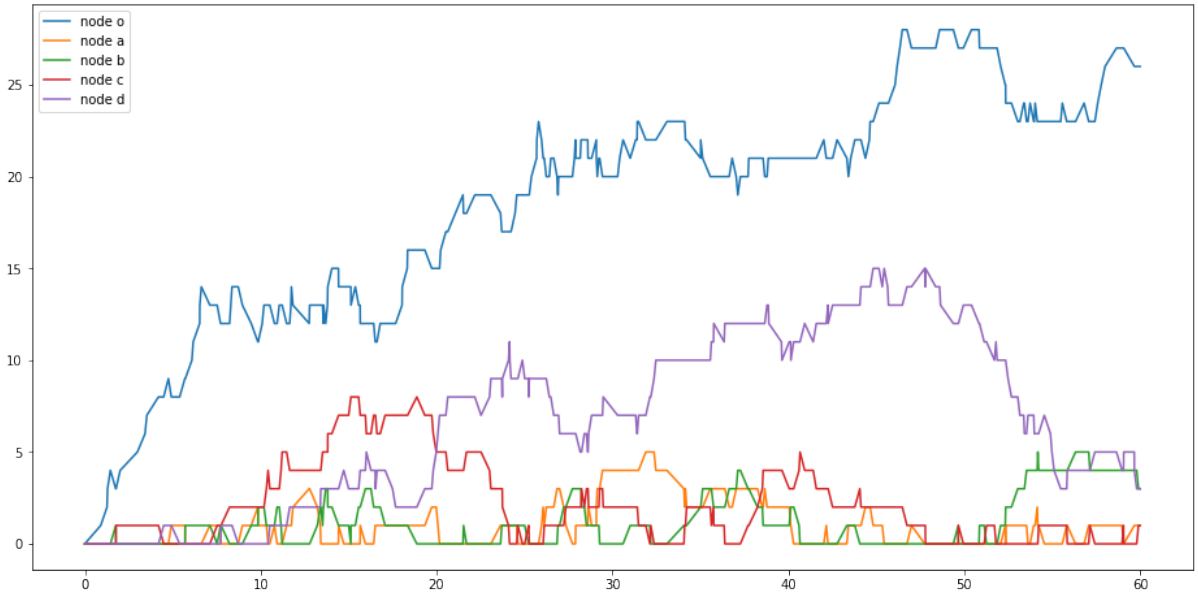

Figure 9: Evolution of the distribution of particles on each node over time, $r_{\text{input}} = 1.2$

This time, in opposition to the case of proportional rate r_i, as the simulation goes on there is evidence that node o accumulates an increasing number of particles over time. It seems that if $r_{\text{input}} > 1$ then $\lim_{t\to\infty} n_o(t) = +\infty$. This is because particles enter in o faster than they jump away from it, as $r_{\text{input}} > r_o = 1$. Therefore, we argue that the system blows up for an input rate $r_{\text{input}} > 1$.

A Python code

In this section is reported (starting from the next page) the Python code used to solve the problems. *N.B.:* the matrix called `Q` in the code corresponds to the matrix $\overline{P}$ in this report.

Problem 1

The first part of this assignment consists in studying a single particle performing a continuous-time random walk in the network described by the graph in Fig. 1 (see text) and with the following transition rate matrix (see Lambda in the code below).

Your task is to simulate the particle moving around in the network in continuous time according to the transition rate matrix given.

[ ]:
```
##### Setup
import numpy as np
from numpy.random import choice

# Define the mapping between 0,1,2,3,4 and o,a,b,c,d
mapping = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

# Create Lambda
Lambda = [
    [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]]

# Define Q, the transition matrix associated to the jump chain associated to the␣
 ↪CTMC
w = np.sum(Lambda, axis=1)
w_star = np.max(w)
Q = Lambda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
print("Q matrix:\n", Q)

# Define the random walk simulation function
def random_walk(Q, xi, n_steps=100, until_first_return = False,␣
 ↪until_S_hit=False, S=[0]):
    '''
    This function simulates a random walk of a particle starting in node xi of a␣
 ↪graph with associated transition matrix Q

    Parameters:
        until_first_return: if True, the random walk stops when the particle has␣
 ↪stepped out of xi at least once and then returns to xi
        until_S_hit: if True, the random walk stops when the particle hits a␣
 ↪node contained in S
        n_steps, if both until_first_return and until_S_hit are false, the␣
 ↪random walk stops when the particle has jumped n_steps times

    Returns:
```

```
        node_seq: sequence of nodes hit by the particle during its random walk
        transition_times: time instants in which the particle has jumped
    '''
    n_nodes = len(Q)
    node_seq = [xi]    # node_seq will keep trace of the visited states; start from node xi
    transition_times = [0]    # transition_times will store the time instants at which jumps/transitions happen; at t=0 the random agent is in node xi
    t_next = -np.log(np.random.rand())/w_star    # the random time to wait for the next transition is drawn from a rate-w_star exponential distribution
    x_start = xi    # starting node
    transition_time = 0    # t=0

    if until_first_return:
        while True:
            # Random jump:
            # the next state to visit will be extracted according to the probabilities
            # stored in the row of Q corresponding to the current state, node_seq[-1]
            old_xi = xi
            xi = choice(range(n_nodes), size=1, p=Q[xi])[0]
            if xi != old_xi:
                node_seq.append(xi)
                transition_time = transition_time + t_next    # store the time instant of the current transition
                transition_times.append(transition_time)
                t_next = -np.log(np.random.rand())/w_star
            else:    # do not jump, stay in the node: in this case, we still have to wait to jump
                t_next = t_next + -np.log(np.random.rand())/w_star
            if xi == x_start and transition_time!=0:    # stopping criterion
                return node_seq, transition_times

    elif until_S_hit:
        if x_start in S:
            return node_seq, transition_times
        while True:
            # Random jump:
            # the next state to visit will be extracted according to the probabilities
            # stored in the row of Q corresponding to the current state, node_seq[-1]
            old_xi = xi
            xi = choice(range(n_nodes), size=1, p=Q[xi])[0]
            if xi != old_xi:
```

```
                node_seq.append(xi)
                transition_time = transition_time + t_next    # store the time␣
↪instant of the current transition
                transition_times.append(transition_time)
                if xi in S:    # stopping criterion: the particle has hit S
                    return node_seq, transition_times
                else:    # continue the simulation
                    t_next = -np.log(np.random.rand())/w_star
            else:    # in this case, we still have to wait to jump
                t_next = t_next + -np.log(np.random.rand())/w_star

    else:
        for _ in range(n_steps):
            # Random jump:
            # the next state to visit will be extracted according to the␣
↪probabilities
            # stored in the row of Q corresponding to the current state,␣
↪node_seq[-1]
            old_xi = xi
            xi = choice(range(n_nodes), size=1, p=Q[xi])[0]
            if xi != old_xi:
                node_seq.append(xi)
                transition_time = transition_time + t_next    # store the time␣
↪instant of the current transition
                transition_times.append(transition_time)
                t_next = -np.log(np.random.rand())/w_star
            else:    # in this case, we still have to wait to jump
                t_next = t_next + -np.log(np.random.rand())/w_star
        return node_seq, transition_times
```

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?

[ ]:
```
def estimate_avg_return_times(Q, n_rnd_walks=100):
    ''' This function estimates the average return times of the len(Q) = 5 nodes␣
↪of the network '''
    n_nodes = len(Q)
    return_times = np.zeros(n_nodes)

    for node in range(n_nodes):    # for each node of G
        return_times_sample = []
        for _ in range(n_rnd_walks):
            _, transition_times = random_walk(Q, node, until_first_return = True)
            return_times_sample.append(transition_times[-1])
        return_times[node] = np.array(return_times_sample).mean()
    return return_times
```

```
# Compute the average return times to each of the five nodes
# (the simulation is performed 1000 times, so the average is computed over 1000 samples)
avg_return_times = estimate_avg_return_times(Q, n_rnd_walks=1000)
print("Return times:", avg_return_times)

# Get the average return time of node a
avg_return_time_a = avg_return_times[mapping['a']]
print("Return time of node a:", avg_return_time_a)
```

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

[ ]:
```
# Find pi bar, the (unique) stationary probability vector
values,vectors = np.linalg.eig(Q.T)
index = np.argmax(values.real)
pi_bar = vectors[:,index].real
pi_bar = pi_bar/np.sum(pi_bar)
print("pi_bar =", pi_bar)

# Compute expected return times (using Kac's formula in continuous time)
expected_return_times = 1/(pi_bar * w)
print("Expected return times:", expected_return_times)

# Get the expected return time of node a
expected_return_time_a = expected_return_times[mapping['a']]
print("Return time of node a:", expected_return_time_a)
```

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

[ ]:
```
def estimate_avg_hitting_times(Q, S, n_rnd_walks=100):
    ''' This function estimates the average hitting times of the len(Q) = 5 nodes of the network related to the set of nodes S '''
    n_nodes = len(Q)
    hitting_times = np.zeros(n_nodes)

    for node in range(n_nodes):    # for each node of G
        hitting_times_sample = []
        for _ in range(n_rnd_walks):
            _, transition_times = random_walk(Q, node, until_S_hit=True, S=S)
            hitting_times_sample.append(transition_times[-1])
        hitting_times[node] = np.array(hitting_times_sample).mean()
    return hitting_times

S = [mapping['d']]
avg_hitting_times = estimate_avg_hitting_times(Q, S, n_rnd_walks=1000)
print("Average hitting times related to node d:", avg_hitting_times)
```

```
avg_hitting_time_o = avg_hitting_times[mapping['o']]
print("Average hitting time of node o related to node d", avg_hitting_time_o)
```

d) How does the result in c) compare to the theoretical hitting-time $\mathbb{E}_o[T_d]$? (Describe also how this is computed.)

[ ]:
```
# Define P, the transition matrix whose i-th row contains the transition␣
 ↪probabilities from node i to its neighbors
# (no self-loops added, as in Q)
w = np.sum(Lambda, axis=1)    # node-clock rates
P = Lambda/(w.reshape(-1,1))    # normalization

# Compute expected hitting times related to d
A = np.eye(len(Q))-P    # (I-P) matrix of coefficients
A[4] = np.array([0,0,0,0,1])
b = 1/w    # (1/w_o, 1/w_a, ..., 1/w_d) vector of constant terms
b[4] = 0
expected_hitting_times = np.linalg.solve(A,b)
print("Expected hitting times related to node d:", expected_hitting_times)

# Get the expected hitting time of node o related to d
expected_hitting_time_o = expected_hitting_times[mapping['o']]
print("Expected hitting time of node o related to node d:",␣
 ↪expected_hitting_time_o)
```

Problem 2

In this part we will again consider the network of the previous exercise, with weights according to transition matrix Lambda (see Lambda in the code below). However, now we will simulate many particles moving around in the network in continuous time.

[ ]:
```
##### Setup
import numpy as np
from numpy.random import choice
from copy import deepcopy
import matplotlib.pyplot as plt

# Define the mapping between 0,1,2,3,4 and o,a,b,c,d
mapping = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}
reverse_mapping = {0:'o', 1:'a', 2:'b', 3:'c', 4:'d'}

# Create Lambda
Lambda = [
    [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]]

# Define Q, the transition matrix associated to the jump chain associated to the␣
 ↪CTMC
w = np.sum(Lambda, axis=1)
w_star = np.max(w)
Q = Lambda/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
print("Q matrix:\n", Q)
```

a) Particle perspective

a.i) If 100 particles all start in node a, what is the average time for a particle to return to node a?

N.B.: I interpreted this request as: "what is the average minimum time for at least one particle to leave and return to node a? I.e., what is the average minimum return time to node a, among the 100 particles?

[ ]:
```
def return_time_100_particles(xi):
    '''
    This function simulates 100 particles doing a random walk, starting in node␣
 ↪xi of a graph with associated transition matrix Q
    The algorithm stops when at least one particle has left and then has␣
 ↪returned to xi.

    Returns:
```

```
        time: time instant at which one of the particles has returned to xi, after having left it
    '''
    n_nodes = len(Q)
    time = 0    # global clock time
    rate = 100 * w_star
    current_state = [ xi for _ in range(100) ]    # a list with 100 values; the i-th value is the node in which the i-th particle is, before the next jump
    t_next = -np.log(np.random.rand())/rate    # the random time to wait for the next transition is drawn from a rate-(100*w_star) exponential distribution

    while True:
        # Choose at random (uniformly) one of the 100 particles
        moving_particle = choice(range(100), size=1)[0]
        # Random jump:
        # the next state that moving_particle will visit will be extracted according to the probabilities
        # stored in the row of Q corresponding to moving_particle's current state, current_state[moving_particle]
        mov_part_curr_state = current_state[moving_particle]    # the last node that the moving particle was in
        new_state = choice(range(n_nodes), size=1, p=Q[mov_part_curr_state])[0]
        if new_state != mov_part_curr_state:    # jump
            current_state[moving_particle] = new_state
            time = time + t_next    # store the time instant of the current transition
            t_next = -np.log(np.random.rand())/rate
        else:    # do not jump, stay in the node: in this case, we still have to wait to jump
            t_next = t_next + -np.log(np.random.rand())/rate
        if new_state == xi and mov_part_curr_state != xi:    # stopping criterion
            return time    # a particle (in particular moving_particle) has returned to xi: the algorithm stops
```

[ ]:
```
# Compute the average minimum return time to node a, among the 100 particles
# (the simulation is performed 1000 times, so the average is computed over 1000 samples)
n_rnd_walks = 1000
return_times_sample = np.array([return_time_100_particles(mapping['a']) for _ in range(n_rnd_walks)])
avg_return_time = return_times_sample.mean()

print("Average time for the first particle to return to a:", avg_return_time)
```

b) Node perspective

b.i) If 100 particles start in 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?

[ ]:
```
def simulate_100_particles_60_time_units(xi):
    '''
    This function simulates 100 particles doing a random walk for 60 time units, starting in node xi of a graph with associated transition matrix Q

    Returns:
        node_distribution: a list of quintuples;
                           node_distribution[i] is the distribution of the 100 particles on the len(Q)=5 nodes,
                           in the time interval ( transition_times[i], transition_times[i+1] )
        transition_times: a list of values;
                          transition_times[i], for i!=-1, is the time instant in which the i-th transition happens
                          transition_times[-1] is t=60.
    '''
    n_nodes = len(Q)
    time = 0    # global clock time
    time_limit = 60    # time units for which the simulation lasts
    transition_times = [0]    # this will store the time instants in which jumps are taken (i.e. in which the global clock ticks)
    rate = 100 * w_star
    t_next = -np.log(np.random.rand())/rate    # the random time to wait for the next transition is drawn from a rate-(100*w_star) exponential distribution

    node_distribution = [ [100 if i==xi else 0 for i in range(n_nodes)] ]    # a list which will contain lists, each containing five values (the n. of particles in each of the five nodes)
                                                                             # one list is appended when a particle is moved, to record the new distribution of particles
    while time + t_next <= time_limit:
        # Select a node at random, proportionally to the number of particles in the different nodes
        current_node_distribution = node_distribution[-1]
        probs = [current_node_distribution[i] / 100 for i in range(n_nodes)]
        node = choice(range(n_nodes), size=1, p=probs)[0]
        # Random jump:
        # move a particle from node node to a neighboring node, with probabilities given by the row of Q corresponding to node node
        new_node = choice(range(n_nodes), size=1, p=Q[node])[0]
        new_node_distribution = deepcopy(current_node_distribution)
```

```
        new_node_distribution[node] -= 1    # NB: it doesn't matter if node and␣
↪new_node coincide
        new_node_distribution[new_node] += 1
        node_distribution.append(new_node_distribution)

        time = time + t_next    # time instant of the jump just occurred
        transition_times.append(time)

        t_next = -np.log(np.random.rand())/rate    # the random time to wait for␣
↪the next transition

    else:
        # t=60 seconds have passed; the simulation stops
        # Append the final distribution to node_distribution and 60 to␣
↪transition_times
        # (this is done to correctly plot the n. of particles in each node␣
↪during the simulation)
        node_distribution.append(node_distribution[-1])
        transition_times.append(60)

    return (node_distribution, transition_times)
```

[ ]:
```
# Compute the average final (t=60) distribution of the 100 particles over the 5␣
↪nodes of the network
# (the simulation is performed 200 times, so the average is computed over 200␣
↪samples)
n_rnd_walks = 200

distribution_transitions_sample =␣
↪[simulate_100_particles_60_time_units(mapping['o']) for _ in␣
↪range(n_rnd_walks)]    # 200 couples
distribution_sample = [i[0] for i in distribution_transitions_sample]    # a␣
↪list containing 200 lists; the i-th inner list contains node_distribution of␣
↪the i-th simulation
transition_times_sample = [i[1] for i in distribution_transitions_sample]    # a␣
↪list containing 200 lists; the i-th inner list contains transition_times of␣
↪the i-th simulation

final_distribution_sample = []    # this list will contain only the␣
↪distributions at t=60 of the 200 simulations
for d in distribution_sample:
    final_d = d[-1]
    final_distribution_sample.append(final_d)
avg_final_distribution = np.array(final_distribution_sample).mean(axis=0)
```

```
print("Average final distribution of particles over the nodes",␣
 ↪avg_final_distribution)
```

Average final distribution of particles over the nodes [18.558 14.695 22.425 22.167 22.155] $\approx 100\overline{\pi}$

b.ii) Illustrate the simulation above with a plot showing the number of particles in each node during the simulation time.

N.B.: I focus on just one of the 200 simulations performed previously, for instance the first one

[ ]:
```
# Take node_distribution and transition_times of the first of the 200 simulations
node_distribution = np.array(distribution_sample[0])
transition_times = deepcopy(transition_times_sample[0])

fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
for node in range(len(Q)):
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend(prop={'size': 20})
```

b.iii) 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 stationary distribution pi bar has been already computed in problem 1;
# here I just repeat that code

# Find pi bar, the (unique) stationary probability vector
values,vectors = np.linalg.eig(Q.T)
index = np.argmax(values.real)
pi_bar = vectors[:,index].real
pi_bar = pi_bar/np.sum(pi_bar)
print("pi_bar =", pi_bar)
```

As noticed previously, it seems that the average final distribution is approximately $100\overline{\pi}$. This result has been justified in the report.

Problem 3

In this part we study how different particles affect each other when moving around in a network in continuous time. We consider the open network of Fig. 2 (see text), with transition rate matrix Lambda_open (see Lambda_open in the code below).

[ ]:
```
##### Setup
import numpy as np
from numpy.random import choice
from copy import deepcopy
import matplotlib.pyplot as plt

# Define the mapping between 0,1,2,3,4 and o,a,b,c,d
mapping = {"o'":0, 'o':1, 'a':2, 'b':3, 'c':4, 'd':5, "d'":6}
reverse_mapping = {0:"o'", 1:'o', 2:'a', 3:'b', 4:'c', 5:'d', 6:"d'"}

# Create Lambda_open
Lambda_open = np.array([
    [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]])

# Add two auxiliary nodes to the open network, to make it "closed": o' and d'
# This is done by adding two rows and columns to Lambda_open: those relative to␣
 ↪o' and d'
# In other words, convert Lambda_open to Lambda_closed
Lambda_closed = deepcopy(Lambda_open)
col_o_prime = np.array([0, 0, 0, 0, 0]).reshape(-1,1)
col_d_prime = np.array([0, 0, 0, 0, 1]).reshape(-1,1)
Lambda_closed = np.hstack((col_o_prime, Lambda_closed, col_d_prime))
row_o_prime = [1, 0, 0, 0, 0, 0, 0]    # self-loop: particles in o' stay in o'␣
 ↪(a separate clock regulates the entrance of the particles in the open network)
row_d_prime = [0, 0, 0, 0, 0, 0, 1]    # self-loop: particles in d' stay in d'
Lambda_closed = np.vstack((row_o_prime, Lambda_closed, row_d_prime))

# Define Q, the transition matrix associated to the jump chain associated to the␣
 ↪CTMC on the closed network
w = np.sum(Lambda_closed, axis=1)
w_star = np.max(w)
Q = Lambda_closed/w_star
Q = Q + np.diag(np.ones(len(w))-np.sum(Q,axis=1))
print("Q matrix:\n", Q)
```

a) Proportional rate

```
[ ]: def simulate_open_network_proportional_rate(input_rate=1, time_limit=60):
        '''
        This function simulates particles entering in the open network (= moving␣
    ↪from o' to o)
        every time a (input_rate)-rate Poisson clock ticks; each of these particles␣
    ↪then performs
        a random walk on the open network, until it exits it (= moves from d to d').
        Each node of the open network passes along particles at each tick of a␣
    ↪Poisson clock
        with rate proportional to the n. of particles on it.

        The simulation stops after t=60 time units have passed.

        Returns:
            node_distribution: a list of 7-tuples;
                               node_distribution[i] is the distribution of the 100␣
    ↪particles on the len(Q)=7 nodes
                               of the closed network, in the time interval (␣
    ↪transition_times[i], transition_times[i+1] )
            transition_times: a list of values;
                              transition_times[i], for i!=-1, is the time instant in␣
    ↪which the i-th transition happens
                              transition_times[-1] is t=60.
        '''

        # Compute at random the time instants in which a new particle enters in node␣
    ↪'o',
        # and therefore also the total number of particles which will enter during␣
    ↪the simulation
        enter_times = []
        time = 0
        enter_t_next = -np.log(np.random.rand())/input_rate
        while time + enter_t_next <= time_limit:
            enter_times.append(time + enter_t_next)
            time = time + enter_t_next
            enter_t_next = -np.log(np.random.rand())/input_rate
        n_particles = len(enter_times)

        # Simulation
        n_nodes = len(Q) # =7
        time = 0    # global clock time
        transition_times = [0]    # this will store the time instants in which jumps␣
    ↪are taken (i.e. in which the global clock ticks)
        rate = n_particles * w_star
        t_next = -np.log(np.random.rand())/rate    # the random time to wait for the␣
    ↪next transition is drawn from a rate-(100*w_star) exponential distribution
```

```
   node_distribution = [ [n_particles,0,0,0,0,0,0] ]    # a list which will␣
↪contain lists, each containing seven values (the n. of particles in each of␣
↪the seven nodes)

   while time + t_next <= time_limit:
       #current_node_distribution = deepcopy(node_distribution[-1])


       while len(enter_times) > 0 and time + t_next >= enter_times[0]:
           # The global clock has ticked after one (or more) particle have␣
↪entered:
           # first let that (those) particle enter in node o, then eventually␣
↪move a particle in the open network

           # A particle enters the open network in node o
           current_node_distribution = deepcopy(node_distribution[-1])
           current_node_distribution[0] -= 1
           current_node_distribution[1] += 1

           node_distribution.append(current_node_distribution)
           transition_times.append(enter_times[0])

           del enter_times[0]    # discard the enter time encountered
           # in whatever case, move a particle in the open network

       # Choose one of the 7 nodes at random (with probabilities proportional␣
↪to the distribution of particles in the nodes);
       # the chosen node will pass along a particle
       current_node_distribution = deepcopy(node_distribution[-1])
       probs = [current_node_distribution[i] / n_particles for i in range(7)]
       node = choice(range(7), size=1, p=probs)[0]

       if current_node_distribution[node] > 0 and node!=0 and node!=6:
       # Random jump
       # if node node has at least a particle sojourning on it, and if it is␣
↪not o' or d',
       # then move a particle from node node to a neighboring node, with␣
↪probabilities
       # given by the row of Q corresponding to node node
           new_node = choice(range(7), size=1, p=Q[node])[0]
           new_node_distribution = deepcopy(current_node_distribution)
           new_node_distribution[node] -= 1
           new_node_distribution[new_node] += 1
           node_distribution.append(new_node_distribution)
```

```
            time = time + t_next    # time instant of the jump just occurred
            transition_times.append(time)

            t_next = -np.log(np.random.rand())/rate    # the random time to wait␣
↪for the next transition
        else:
            # otherwise, if node node has no particles sojourning on it, or if␣
↪it is o' or d',
            # don't take any jumps and just update the time
            time = time + t_next
            t_next = -np.log(np.random.rand())/rate    # the random time to wait␣
↪for the next transition


    else:
        # no more transitions happen, but what about entrances?
        # there may be one or even more particles waiting to enter the open␣
↪network
        while len(enter_times) > 0:
            time = enter_times.pop(0)
            transition_times.append(time)
            # A particle enters the open network in node o
            current_node_distribution = deepcopy(node_distribution[-1])
            current_node_distribution[0] -= 1
            current_node_distribution[1] += 1
            node_distribution.append(current_node_distribution)

        # Append the final distribution to node_distribution and 60 to␣
↪transition_times
        # (this is done to correctly plot the n. of particles in each node␣
↪during the simulation)
        node_distribution.append(node_distribution[-1])
        transition_times.append(time_limit)

    return (node_distribution, transition_times)
```

[ ]:
```
# Perform the simulation with entrance rate = 1
node_distribution, transition_times =␣
 ↪simulate_open_network_proportional_rate(input_rate=1)
node_distribution = np.array(node_distribution)

# Plot the evolution of the n. of particles in each node over time
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
for node in range(1,6):
```

```
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend()



# Perform the simulation with entrance rate = 10
node_distribution, transition_times =␣
 ↪simulate_open_network_proportional_rate(input_rate=10)
node_distribution = np.array(node_distribution)

# Plot the evolution of the n. of particles in each node over time
fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
for node in range(1,6):
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend()



# Perform the simulation with entrance rate = 100
node_distribution, transition_times =␣
 ↪simulate_open_network_proportional_rate(input_rate=100)
node_distribution = np.array(node_distribution)

# Plot the evolution of the n. of particles in each node over time
fig = plt.figure(3, figsize=(16,8))
ax = plt.subplot()
for node in range(1,6):
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend()
```

b) Fixed rate

```
[ ]: def simulate_open_network_fixed_rate(input_rate=1, time_limit=60):
    '''
    This function simulates particles entering in the open network (= moving␣
 ↪from o' to o)
    every time a (input_rate)-rate Poisson clock ticks; each of these particles␣
 ↪then performs
    a random walk on the open network, until it exits it (= moves from d to d').
```

```
    Each node of the open network passes along particles, if any, at each tick␣
↪of a Poisson clock
    with a fixed rate = 1, associated to that node.

    The simulation stops after t=60 time units have passed.

    Returns:
        node_distribution: a list of 7-tuples;
                           node_distribution[i] is the distribution of the 100␣
↪particles on the len(Q)=7 nodes
                           of the closed network, in the time interval (␣
↪transition_times[i], transition_times[i+1] )
        transition_times: a list of values;
                          transition_times[i], for i!=-1, is the time instant in␣
↪which the i-th transition happens
                          transition_times[-1] is t=60.
    '''

    # Compute at random the time instants in which a new particle enters in node␣
↪'o',
    # and therefore also the total number of particles which will enter during␣
↪the simulation
    enter_times = []
    time = 0
    enter_t_next = -np.log(np.random.rand())/input_rate
    while time + enter_t_next <= time_limit:
        enter_times.append(time + enter_t_next)
        time = time + enter_t_next
        enter_t_next = -np.log(np.random.rand())/input_rate
    n_particles = len(enter_times)

    # Simulation
    n_nodes = len(Q) # =7
    time = 0    # global clock time
    transition_times = [0]    # this will store the time instants in which jumps␣
↪are taken (i.e. in which the global clock ticks)
    rate = 5 * w_star
    t_next = -np.log(np.random.rand())/rate    # the random time to wait for the␣
↪next transition is drawn from a rate-(100*w_star) exponential distribution

    node_distribution = [ [n_particles,0,0,0,0,0,0] ]    # a list which will␣
↪contain lists, each containing seven values (the n. of particles in each of␣
↪the seven nodes)

    while time + t_next <= time_limit:
        #current_node_distribution = deepcopy(node_distribution[-1])
```

```
        while len(enter_times) > 0 and time + t_next >= enter_times[0]:
            # The global clock has ticked after one (or more) particle have␣
↪entered:
            # first let that (those) particle enter in node o, then eventually␣
↪move a particle in the open network

            # A particle enters the open network in node o
            current_node_distribution = deepcopy(node_distribution[-1])
            current_node_distribution[0] -= 1
            current_node_distribution[1] += 1

            #print(current_node_distribution)
            node_distribution.append(current_node_distribution)
            transition_times.append(enter_times[0])

            del enter_times[0]    # discard the enter time encountered
            # in whatever case, move a particle in the open network

        # Choose one of the 5 nodes of the open network at random (uniformly);
        # the chosen node will pass along a particle, if it has one
        node = choice(range(1,6), size=1)[0]
        current_node_distribution = deepcopy(node_distribution[-1])

        if current_node_distribution[node] > 0:
        # Random jump
        # if node node has at least a particle sojourning on it, then move a␣
↪particle from node node to a neighboring node,
        # with probabilities given by the row of Q corresponding to node node
            new_node = choice(range(7), size=1, p=Q[node])[0]
            new_node_distribution = deepcopy(node_distribution[-1])
            new_node_distribution[node] -= 1    # NB: it doesn't matter if node␣
↪and new_node coincide
            new_node_distribution[new_node] += 1
            node_distribution.append(new_node_distribution)

            time = time + t_next    # time instant of the jump just occurred
            transition_times.append(time)

            t_next = -np.log(np.random.rand())/rate    # the random time to wait␣
↪for the next transition
        else:
            # otherwise, if node node has no particles sojourning on it, don't␣
↪take any jumps and just update the time
            time = time + t_next
```

```
            t_next = -np.log(np.random.rand())/rate    # the random time to wait␣
↪for the next transition

    else:
        # no more transitions happen, but what about entrances?
        # there may be one or even more particles waiting to enter the open␣
↪network
        while len(enter_times) > 0:
            time = enter_times.pop(0)
            transition_times.append(time)
            # A particle enters the open network in node o
            current_node_distribution = deepcopy(node_distribution[-1])
            current_node_distribution[0] -= 1
            current_node_distribution[1] += 1
            node_distribution.append(current_node_distribution)

        # Append the final distribution to node_distribution and 60 to␣
↪transition_times
        # (this is done to correctly plot the n. of particles in each node␣
↪during the simulation)
        node_distribution.append(node_distribution[-1])
        transition_times.append(time_limit)

    return (node_distribution, transition_times)
```

[ ]:
```
# Perform the simulation with entrance rate = 1
node_distribution, transition_times =␣
 ↪simulate_open_network_fixed_rate(input_rate=1, time_limit=60)
node_distribution = np.array(node_distribution)

# Plot the evolution of the n. of particles in each node over time
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
for node in range(1,6):
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend()



# Perform the simulation with entrance rate = 2
node_distribution, transition_times =␣
 ↪simulate_open_network_fixed_rate(input_rate=1.1, time_limit=60)
node_distribution = np.array(node_distribution)
```

```
# Plot the evolution of the n. of particles in each node over time
fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
for node in range(1,6):
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend()



# Perform the simulation with entrance rate = 5
node_distribution, transition_times =␣
 ↪simulate_open_network_fixed_rate(input_rate=1.2, time_limit=60)
node_distribution = np.array(node_distribution)

# Plot the evolution of the n. of particles in each node over time
fig = plt.figure(3, figsize=(16,8))
ax = plt.subplot()
for node in range(1,6):
    ax.plot(transition_times, node_distribution[:,node], label=f'node␣
 ↪{reverse_mapping[node]}')    # label=f'node {reverse_mapping[node]}'

ax.legend()
```