POLITECNICO DI TORINO

NETWORK DYNAMICS AND LEARNING

2020 / 2021

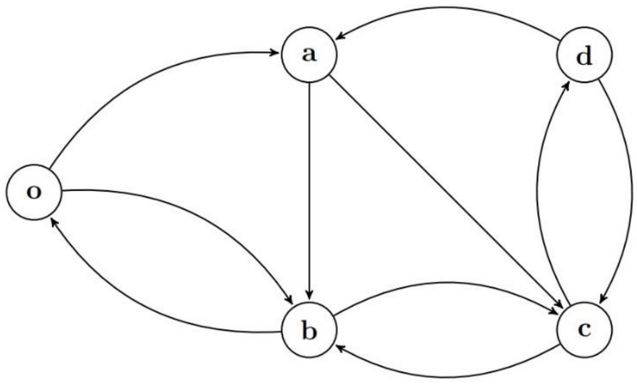
ALBERTO MARIA FALLETTA

S277971

HOMEWORK II

Problem 1

The first part of the assignment consists in studying a single particle performing a continuous-time random walk in the network described by the following graph and with transition rate matrix Λ.



$$\Lambda = \begin{array}{c} \\ \\ \end{array}\begin{array}{ccccc|c} o & a & b & c & d & \\ \end{array}$$

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

A)

To simulate a particle moving around in the network in continuous time according to the transition rate matrix Λ the approach considering each node i equipped with its own independent Poisson clock with rate $\omega_i = \sum_j \Lambda_{ij}$ has been preferred, as shown in the code in Appendix I. According to this approach when the clock of a given node clicks, the particle has no choice but to jump to a neighbor node j with probability $P_{ij} = \Lambda_{ij}/\omega_i$, with $\omega = \Lambda\mathbb{1}$ that is Λ row-sum vector.

The empirical return time for a particle starting from node A is $\mathbb{E}_a[T_a^+] = 6.75196 \pm 0.01416$ s and has been computed averaging over 100.000 simulations of single random walks.

An important remark to be made is that to simulate the Poisson clock associated to node i in which the particle was, the time t_{next} (with $t_{next} = -\ln(u)/\omega_i$ and $u \sim U(0,1)$ drawn from a uniformly distributed random variable) between two clicks of that Poisson clock has been simulated instead, since this time is an independent random variable with exponential distribution of rate ω_i.

Finally, once the Poisson clock associated to the node containing the particle clicks, it is then selected as neighbor node to jump to the one associated to the column index for which the cumulative sums of rows matrix, in the row of current node, has the smallest value among all the values greater than a random number uniformly distributed between 0 and 1.

B)

To compute the theoretical return-time $\mathbb{E}_a[T_a^+]$ is useful to remember the theorem[1] according to which if G_Λ is strongly connected, as in our case, (being G_Λ the graph associated to the continuous-time Markov chain with finite state space $\mathcal{X}$ and transition rates $\Lambda_{ij}\ for\ i \neq j\ \in\ \mathcal{X}$. With Λ nonnegative square matrix with outside-diagonal entries equal to the transition rates and diagonal entries $\Lambda_{ii} = 0$) then:

[1] Como, Fagnani, *Lecture notes on Network Dynamics*: theorem 7.2, page 118

- there exists a unique Laplace-invariant probability distribution π_bar
- the expected return times satisfy $\mathbb{E}_a[T_a^+] = 1/(\omega_a \cdot \pi\_bar_a)$

Laplace-invariant probability distribution π_bar has been computed, by means of the code in Appendix II, normalizing the eigenvector associated to the leading eigenvalue of the matrix P_bar defined as:

$$P\_bar_{ij} = \frac{\Lambda_{ij}}{\omega^*}, i \neq j. \;\; P_{bar_{ii}} = 1 - \sum_{i \neq j} P\_bar_{ij}$$

Therefore, remembering node order is (O, A, B, C, D), the obtained results are:

$\pi_{bar} =$ [0.18518519 0.14814815 0.22222222 0.22222222 0.22222222]
$\omega = \Lambda\mathbb{1} =$ [0.6 1. 1. 1. 0.66666667]
$\mathbb{E}_a[T_a^+] = 1/(\omega_a \cdot \pi_{bar_a}) = 1/(1 \cdot 0.14814815) \approx 6.75$

We can, in fact, see that the simulation result and the theoretical one are consistent with each other since the first one tends to the second one as the number of simulations, over which to average, increases.

C)

To empirically compute, on average, the time it takes to move from node O to node D, namely the hitting-time of D starting from O, the code in Appendix III has been used. This code is very similar to the one used to compute the expected return time, except this time the starting node is O and the ending condition is if D has been reached, instead of the previous ending condition according to which a node would have to be the same as the starting one. Apart from these two differences the code works exactly as the one already analyzed in point A.

The empirical hitting time of D for a particle starting from node O is $\mathbb{E}_O[T_D] = 8.77194 \pm 0.01257$ s and has been computed averaging over 100.000 simulations of single random walks.

D)

A further implication of the theorem mentioned above, in point B, is that in case G_Λ is strongly connected the expected hitting times $\tau_i^S = \mathbb{E}_i[T_S],\ i \in \mathcal{X}$ are the unique solutions of:

$$\begin{cases} \tau_s^S = 0 \; if \; s \in S \\ \tau_i^S = \dfrac{1}{\omega_i} + \displaystyle\sum_{j \in \mathcal{X}} P_{ij}\, \tau_j^S \; if \; i \in \mathcal{X} \backslash S \end{cases}$$

The system can therefore be solved to find $\tau_O^D = \mathbb{E}_O[T_D]$ keeping in mind that:

- S in our case contains only node D
- $P_{ij} = \Lambda_{ij}/\omega_i$, where $\omega = \Lambda \mathbb{1}$
- To keep a simpler notation: $\tau_i^S = \tau_i$

$$\begin{cases} \tau_O = \frac{1}{\omega_O} + \frac{\Lambda_{OA}}{\omega_O}\tau_A + \frac{\Lambda_{OB}}{\omega_O}\tau_B = \frac{1}{3/5} + \frac{2/5}{3/5}\tau_A + \frac{1/5}{3/5}\tau_B \quad (1) \\ \tau_A = \frac{1}{\omega_A} + \frac{\Lambda_{AB}}{\omega_A}\tau_B + \frac{\Lambda_{AC}}{\omega_A}\tau_C = 1 + \frac{3}{4}\tau_B + \frac{1}{4}\tau_C \quad (2) \\ \tau_B = \frac{1}{\omega_B} + \frac{\Lambda_{BO}}{\omega_B}\tau_O + \frac{\Lambda_{BC}}{\omega_B}\tau_C = 1 + \frac{1}{2}\tau_O + \frac{1}{2}\tau_C \quad (3) \\ \tau_C = \frac{1}{\omega_C} + \frac{\Lambda_{CB}}{\omega_C}\tau_B = 1 + \frac{1}{3}\tau_B + \frac{2}{3}\tau_D = 1 + \frac{1}{3}\tau_B \quad (4) \\ \tau_D = 0 \quad (5) \end{cases}$$

Substituting (4) in (3):

$$\tau_B = 1 + \frac{1}{2}\tau_O + \frac{1}{2}\tau_C = 1 + \frac{1}{2}\tau_O + \frac{1}{2}\left(1 + \frac{1}{3}\tau_B\right) \rightarrow \tau_B = \frac{6}{10}\tau_O + \frac{18}{10}$$

Substituting (3) and (4) in (2):

$$\tau_A = 1 + \frac{3}{4}\tau_B + \frac{1}{4}\tau_C = 1 + \frac{3}{4}\left(\frac{6}{10}\tau_O + \frac{18}{10}\right) + \frac{1}{4}\left[1 + \frac{1}{3}\left(\frac{6}{10}\tau_O + \frac{18}{10}\right)\right] \rightarrow \tau_A = \frac{1}{2}\tau_O + \frac{33}{12}$$

Finally substituting (2) and (3) in (1):

$$\tau_O = \frac{5}{3} + \frac{2}{3}\left(\frac{1}{2}\tau_O + \frac{33}{12}\right) + \frac{1}{3}\left(\frac{6}{10}\tau_O + \frac{18}{10}\right) \rightarrow \tau_O = \frac{123}{14} \approx 8.78571$$

We can clearly see how the empirical results tend to the theoretical results as the number of iterations over which to average increases.

Problem 2

In this exercise we will once again consider the network described in Problem 1. However, we will now simulate many particles moving around in the network in continuous time. Each particle in the network will move around just as the single particle moved around in the previous problem: the time it will stay in a node is exponentially distributed, and on average it will stay $1/\omega_i$ time-units in a node i before moving to one of its out-neighbors.

A) Particle Perspective

To simulate particle perspective a single, as shown in code in Appendix IV, a system-wide Poisson clock with rate equal to the number of particles in the network is used. At every click of the system-wide clock we randomly select a particle but this time, it is not mandatory for the particle to move since the Poisson clock used is global. To enforce this behavior, we therefore use matrix P_bar instead of matrix P. This means, that if a particle in node O is selected, there is a probability of $1 - \omega_O$ that it will stay in node O, and a probability of ω_O that it will leave the node.

The result over five simulations shows it takes roughly $0.497365 \pm 0.314095\ s$ for the first particle to arrive while it takes 26.107276 ± 3.974694 s for the last, but most importantly it shows that it is consistent with the previous problem's calculations since the average converges to the theoretical 6.75 s as the number of simulations increases.

B) Node Perspective

To simulate node perspective, as shown in code in Appendix V, once again a system-wide Poisson clock with rate equal to the number of particles in the network is used. At each click a node, from which to move a particle, is selected randomly, and proportionally to the number of particles in the different nodes. Then a particle from the selected node will move according to the transition probability matrix P_bar, once again allowing the particle to stay in the node.

From this experiment, consisting in averaging the results obtained over 10 simulation, we can notice that the distribution of particles in the network tends to the Laplace-invariant probability distribution π_bar as the number of simulations increases. We have in fact:

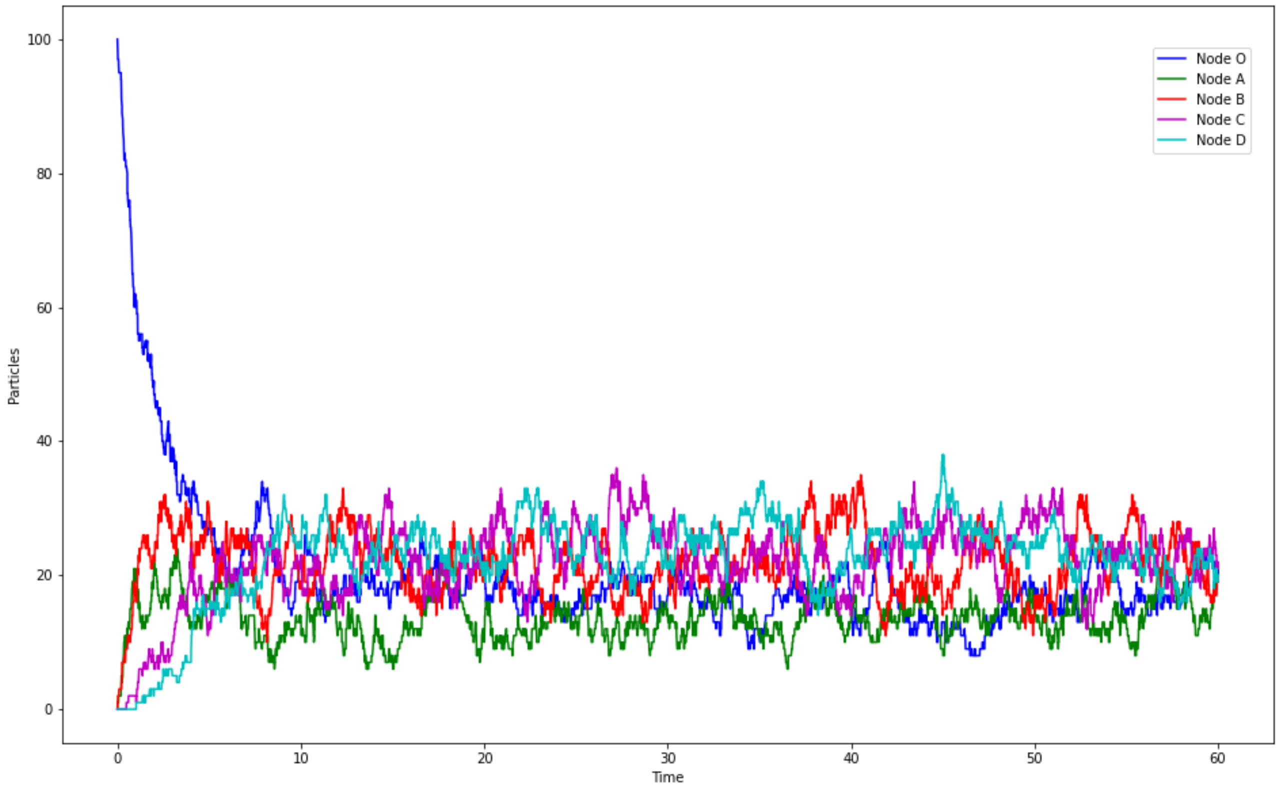
$\text{Particle distribution averaged over 10 simulations} = [0.175, 0.16, 0.232, 0.215, 0.218]$

$Theoretically\_computed\_\pi_{bar} =$ [0.185 0.148 0.222 0.222 0.222]

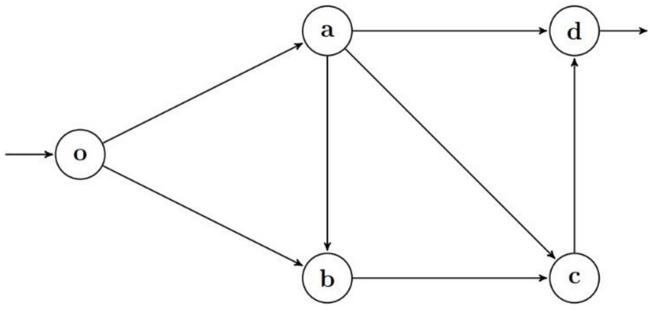
This result does not actually come as a surprise since, recalling once again the theorem[2] mentioned above, we have that if G_Λ is strongly connected then for every initial probability distribution $P(X(0) = i) = \pi_{bar\,i}(0)\ for\ i \in \mathcal{X}$, the time-t marginal probability distribution $\pi_{bar\,i}(t) = P(X(t) = i)$ satisfies: $\lim_{t\to+\infty} \pi_{bar}(t) = \pi_{bar}$

[2] Como, Fagnani, *Lecture notes on Network Dynamics*: theorem 7.2, page 118

Finally, it is possible to Illustrate the simulation above with a plot showing the number of particles in each node during the simulation time:



Problem 3



$$
\Lambda_{\text{open}} = \begin{array}{c} \\ \end{array}\left(\begin{array}{ccccc} 0 & 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 2/4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right)\begin{array}{c} o \\ a \\ b \\ c \\ d \end{array}
$$

(columns: $o \quad a \quad b \quad c \quad d$)

In this final exercise we study how different particles affect each other when moving in a network in continuous time. We consider the open network above, with transition rate matrix Λ_{open}.

Particles enter the system in node O according to a Poisson process with rate 1 and each node will then pass along a particle according to a given rate. We will simulate two different scenarios, differing by the rate at which nodes will pass along particles: in the first case the rate will be proportional (that is each node will pass along particles according to a Poisson process with rate equal to the number of particles in the node), while in the second case each node will instead pass along particles with a fixed rate of 1. Note that since node D does not have a node to send its particles to, when the Poisson clock clicks for this node, we simply decrease the number of particles in the node by one, however, to avoid working with a singular matrix, a 1 will be set as entry $\Lambda_{open\ DC}$ (the results do not change since the click for D will only decrease the particles in D and will not send any particle to C).

Furthermore, both point A and B have been modelled as a system with two Poisson clocks, the first one managing particle entrance in the network with initial rate 1 (rate that will be modified to experiment and study the behavior of the network), while the second, global, managing the movement of particles through the network. In this second global Poisson clock lays the biggest difference between the two scenarios.

A) Proportional Rate

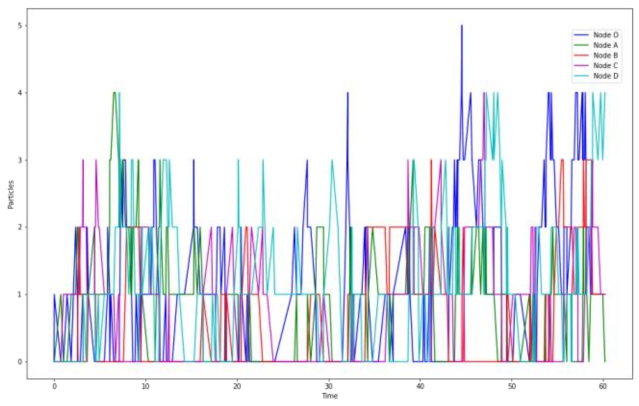
This first scenario has been modelled in two different implementations, leading to the same result but equally interesting, whose code can be found in Appendix VI and VII.

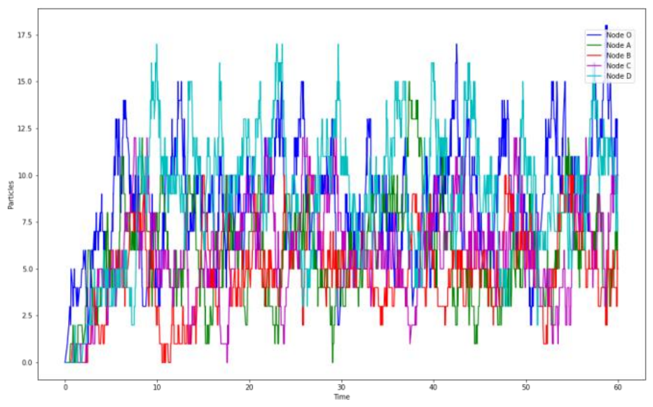
Being in the Proportional Rate setting, differently from the second scenario, the rate of the global clock depends on the number of particles in the network.

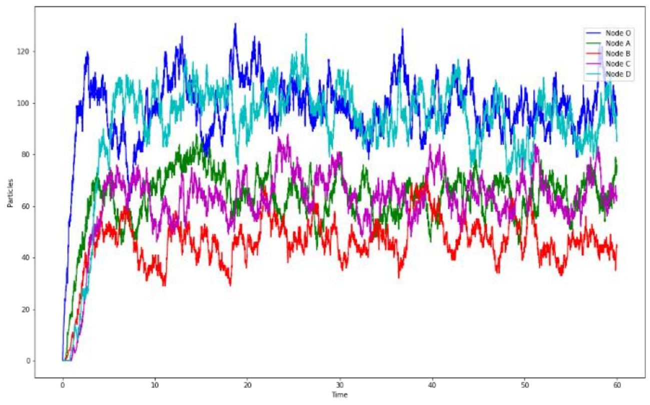
In the first implementation, at the beginning of each iteration (except for the first one since a particle has to enter the network that would otherwise be empty), both the time for the input clock to click and the time for the global clock to click are simulated and compared: if the global clock's click time is smaller that means a particle has to decide whether to jump or stay in its current node, a node is therefore chosen with probability correlated to the number of particles in the given node and finally a random particle among the ones in the chosen node is selected and moved according to P_bar (always allowing for the particle to stay in the node) except if the clicking clock is related to node D, in that case a random particle is removed from D. If the input clock's click time is smaller instead, a particle enters the network in O.

The second implementation, on the other hand, simulates all the clicks of the input clock for 60 units of time in advance and saves the time values in which a particle must enter in the network in a list. During the actual simulation only the global clock click time is simulated, also in this case a node is then chosen with probability correlated to the number of particles in the given node and finally a random particle among the ones in the chosen node is selected and moved according to P_bar (always allowing for the particle to stay in the node) except if the clicking clock is related to node D, in that case a random particle is removed from D, but differently from the previous implementation, particles enter the network in the precomputed time values.

As we can see from the following plots, respectively of input rate 1, 10 and 100, no value of input rate makes the network 'blow up' and that is because with proportional rate of the global clock, the more the particles in the network, the bigger the rate and the faster is the network in disposing the particles out of D.





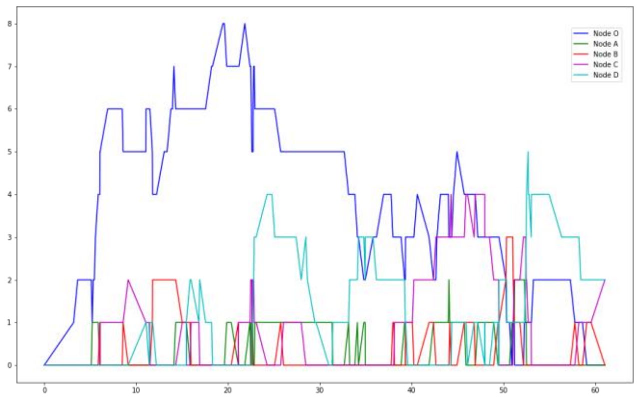


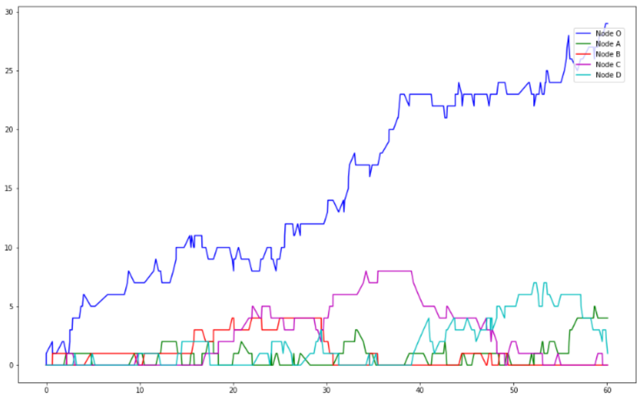
B) Fixed Rate

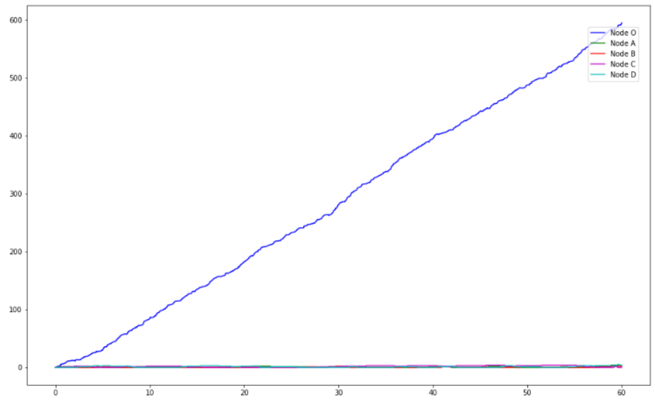
The code for this second scenario can instead be found in Appendix VIII.

Being in the Fixed Rate setting, differently from the second scenario, the rate of the global clock only depends on the number of nodes holding at least one particle.

In the simulation, at the beginning of each iteration (except for the first one since a particle has to enter the network that would otherwise be empty), both the time for the input clock to click and the time for the global clock to click are simulated and compared: if the global clock's click time is smaller that means a particle has to decide whether to jump or stay in its current node, a node is therefore chosen with uniform probability among the nodes holding at least one particle and finally a random particle among the ones in the chosen node is selected and moved according to P_bar (always allowing for the particle to stay in the node) except if the clicking clock is related to node D, in that case a random particle is removed from D. If the input clock's click time is smaller instead, a particle enters the network in O.







As we can see from the previous plots, respectively of input rate 0.5, 1 and 10, already with low values of input rate such as the starting value of 1, the network ‘blows up’. That is because differently from the proportional rate, the clock is unable of accelerating and is therefore not able to dispose the particles out of D fast enough.

Appendix

I)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

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]]

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

w = np.sum(Lambda, axis=1)
D = np.diag(w)
P = np.linalg.inv(D) @ Lambda

n_iter = 100000  # number of times to repeat the simulation
n_steps = 50  # automatic arrest of the simulation over 50 steps
starting_node = 'a'
total_time = 0
cnt = 0

for j in range(n_iter):
  pos = np.zeros(n_steps, dtype=int)  # storing the sequence of visited nodes
  pos[0] = node_dict[starting_node]  # starting node 'a' associated with 1
  transition_times = np.zeros(n_steps)  # storing times nodes have been visited
  # First Poisson clock click simulation
  t_next = -np.log(np.random.rand())/w[node_dict[starting_node]]
  P_cum = np.cumsum(P, axis=1)

  for i in range(1,n_steps):  # Actual simulation of a given particle
    # jump to a new node
    pos[i] = np.argwhere(P_cum[pos[i-1]] > np.random.rand())[0]
    transition_times[i] = transition_times[i-1] + t_next  # updates the time

    if pos[i] == pos[0]:  # check if the new node is the starting one
      total_steps = i+1
      simulation_time = transition_times[i]
      break
    # if not simulates next click and will jump to a new one in the next iter
```

```
        t_next = -np.log(np.random.rand())/w[pos[i]]

    total_time += simulation_time
    cnt += 1

print("Average return time: ", round(total_time/cnt, 4), 's')
```

II)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

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]]

w = np.sum(Lambda, axis=1)
w_star = np.max(w)
P_BAR = Lambda/w_star
P_BAR = P_BAR + np.diag(np.ones(len(w))-np.sum(P_BAR,axis=1))

values,vectors = np.linalg.eig(P_BAR.T)
index = np.argmax(values.real)  # index of the leading eigenvalue
pi_bar = vectors[:,index].real  # eigenvector associated to the leading eigenvalue
pi_bar = pi_bar/np.sum(pi_bar)  # normalization

print("pi_bar=", pi_bar)
print("Ea[Ta+] = 1/(wa + πa) = ", round(1/(w[1]*pi_bar[1]), 4))
```

III)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

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]]

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

w = np.sum(Lambda, axis=1)
D = np.diag(w)
P = np.linalg.inv(D) @ Lambda

n_iter = 100000  # number of times to repeat the simulation
n_steps = 50  # automatic arrest over 50 steps
starting_node = 'o'
total_time = 0
```

```
cnt = 0

for j in range(n_iter):
  pos = np.zeros(n_steps, dtype=int)  # vector storing sequence of visited nodes
  pos[0] = node_dict[starting_node]  # starting node 'a' associated with 1
  transition_times = np.zeros(n_steps)  # storing times nodes have been visited
  # First Poisson clock click
  t_next = -np.log(np.random.rand())/w[node_dict[starting_node]]
  P_cum = np.cumsum(P, axis=1)

  for i in range(1,n_steps):
    pos[i] = np.argwhere(P_cum[pos[i-1]] > np.random.rand())[0]
    transition_times[i] = transition_times[i-1] + t_next
    t_next = -np.log(np.random.rand())/w[pos[i]]
    if pos[i] == 4:
      total_steps = i+1
      simulation_time = transition_times[i]
      break

  total_time += simulation_time
  cnt += 1

print("Average hitting time of 'd' from 'o': ", round(total_time/cnt, 4), 's')
```

IV)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

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]]

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

w = np.sum(Lambda, axis=1)
P_BAR = Lambda/w_star
P_BAR = P_BAR + np.diag(np.ones(len(w))-np.sum(P_BAR,axis=1))

n_particles = 100  # number of particles
global_poisson_clock_rate = n_particles
n_steps = 50  # automatic arrest over fixed number of steps
starting_node = 'a'
pos = np.zeros((n_particles, n_steps), dtype=int)  # matrix of visited nodes for particl
# keeps track of where the sequence ends in pos array for each particle
advancement_indexes = np.ones(n_particles, dtype=int)
# if a given particle has exited A to be considered return
particle_has_exited_first = np.zeros(n_particles, dtype=int)
travel_times = np.zeros(n_particles)  # vector storing particles travel times
# final travel times stored only when particle returns to A
final_travel_times = np.zeros(n_particles)
returned_particles = np.zeros(n_particles)  # storing if a particle returned to A or not

# Setting starting node 'a' that is associated with 1, for every particle
for particle in range(0, n_particles):
  pos[particle][0] = node_dict[starting_node]
```

```
P_BAR_cum = np.cumsum(P_BAR, axis=1)

while returned_particles.sum() < n_particles:
  """
  This loop cycles until there are particles that are not yet returned at least once
  1- simulates the click of the clock
  2- chooses a random particle
  3- if the particle has not returned to A it may jump or not according to P_BAR
  4- when particle jumps the flag "particle_has_exited_first" is set to True
  5- advancement_indexes[particle] points at the next spot in the sequence
  6- the waiting time for the click is then added to all the particles
  7- when the new node is equal to the starting node the waiting time for the click is
added and the
    total waiting time is stored in final_travel_times[particle]
  """
  t_next = -np.log(np.random.rand())/global_poisson_clock_rate  # Global Clock click
  particle = random.choice([i for i in range(0, n_particles)])  # Random particle choice

  if returned_particles[particle] == 0:

pos[particle][advancement_indexes[particle]]=np.argwhere(P_BAR_cum[pos[particle][advancem
ent_indexes[particle]-1]]>np.random.rand())[0]  # move to new node
    particle_has_exited_first[particle] = 1

    # returned particle is flagged and travel time is saved in final_travel_times array
    if pos[particle][advancement_indexes[particle]] == pos[particle][0] and
particle_has_exited_first[particle] == 1:
      travel_times[particle] += t_next
      returned_particles[particle] = 1
      final_travel_times[particle] = travel_times[particle]
      print(f"Particle {particle} Returned\tTrajectory:
{pos[particle][:advancement_indexes[particle]+1]}\t\ttime:
{round(travel_times[particle],4)}")

    advancement_indexes[particle] += 1

  for particle in range(0, n_particles):
    travel_times[particle] += t_next  # adding time to every particle

print()
print("FASTEST PARTICLE = ", min(final_travel_times))
print("SLOWEST PARTICLE = ", max(final_travel_times))
print("AVERAGE = ", sum(final_travel_times) / len(final_travel_times))
```

V)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

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]]

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}
```

```
w = np.sum(Lambda, axis=1)
w_star = np.max(w)
P_BAR = Lambda/w_star
P_BAR = P_BAR + np.diag(np.ones(len(w))-np.sum(P_BAR,axis=1))

# Simulating and averaging over n_iter iterations
n_iter = 10
node_0_count = 0
node_1_count = 0
node_2_count = 0
node_3_count = 0
node_4_count = 0

for i in range(0, n_iter):
  n_particles = 100  # number of particles
  global_poisson_clock_rate = n_particles
  simulation_time_units = 60
  current_nodes = [0 for i in range(0, n_particles)]  # storing node for each particle
  elapsed_time = 0

  # For visualization of last iteration
  plot_time = [elapsed_time]
  particles_in_0 = [n_particles]
  particles_in_1 = [0]
  particles_in_2 = [0]
  particles_in_3 = [0]
  particles_in_4 = [0]

  P_BAR_cum = np.cumsum(P_BAR, axis=1)

  while elapsed_time < simulation_time_units:
    t_next = -np.log(np.random.rand())/global_poisson_clock_rate  # Global Clock click

    # probabilities based on particles in each node
    node_0_prob = current_nodes.count(0)/n_particles
    node_1_prob = current_nodes.count(1)/n_particles
    node_2_prob = current_nodes.count(2)/n_particles
    node_3_prob = current_nodes.count(3)/n_particles
    node_4_prob = current_nodes.count(4)/n_particles

    # node choice and choice of a particle in choosen node
    choosen_node = choice([0, 1, 2, 3, 4], 1, p=[node_0_prob, node_1_prob, node_2_prob,
node_3_prob, node_4_prob])[0]  # since choice returns a list
    particles_in_choosen_node = []
    # extraction of indices associated to particles in the choosen node
    for index in range(0, len(current_nodes)):
      if current_nodes[index] == choosen_node:
        particles_in_choosen_node.append(index)
    # Random particle choice in chosen node
    particle = random.choice(particles_in_choosen_node)

    # Sending the particle to a new node according to P
    current_nodes[particle] = np.argwhere(P_BAR_cum[current_nodes[particle]] >
np.random.rand())[0][0]  # move to new node

    # Updating elapsed time
    elapsed_time += t_next

    # For visualization purposes
    plot_time.append(elapsed_time)
    particles_in_0.append(current_nodes.count(0))
    particles_in_1.append(current_nodes.count(1))
    particles_in_2.append(current_nodes.count(2))
    particles_in_3.append(current_nodes.count(3))
    particles_in_4.append(current_nodes.count(4))

  node_0_count += current_nodes.count(0)
  node_1_count += current_nodes.count(1)
  node_2_count += current_nodes.count(2)
  node_3_count += current_nodes.count(3)
  node_4_count += current_nodes.count(4)

print(f"Average particle distribution: [{node_0_count/(n_iter*n_particles)},
{node_1_count/(n_iter*n_particles)}, {node_2_count/(n_iter*n_particles)},
{node_3_count/(n_iter*n_particles)}, {node_4_count/(n_iter*n_particles)}]")

# PI_BAR
values,vectors = np.linalg.eig(P_BAR.T)
index = np.argmax(values.real)  # index of the leading eigenvalue
pi_bar = vectors[:,index].real  # eigenvector associated to the leading eigenvalue
pi_bar = pi_bar/np.sum(pi_bar)  # normalization
print("pi_bar:", pi_bar)

fig_1, ax = plt.subplots(figsize=(16, 10))
ax.plot(plot_time, particles_in_0, c='b', linestyle='-', label='Node O')
ax.plot(plot_time, particles_in_1, c='g', linestyle='-', label='Node A')
ax.plot(plot_time, particles_in_2, c='r', linestyle='-', label='Node B')
ax.plot(plot_time, particles_in_3, c='m', linestyle='-', label='Node C')
ax.plot(plot_time, particles_in_4, c='c', linestyle='-', label='Node D')
ax.legend(loc=(0.9, 0.8))  # position: 90% of the length, 80% of height
ax.set_xlabel('Time')
ax.set_ylabel('Particles')
plt.show()
```

VI)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

Lambda = [
[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, 1, 0]]  # modified to avoid singular matrix

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

w = np.sum(Lambda, axis=1)
w_star = np.max(w)
P_BAR = Lambda/w_star
P_BAR = P_BAR + np.diag(np.ones(len(w))-np.sum(P_BAR,axis=1))

node_0_count = 0
node_1_count = 0
node_2_count = 0
node_3_count = 0
node_4_count = 0

input_poisson_clock_rate = 1
simulation_time_units = 60
elapsed_time = 0
current_nodes = []  # particles in the network, index is the particle, value is the node
```

```
# For visualization of last iteration
plot_time = [elapsed_time]
particles_in_0 = [0]
particles_in_1 = [0]
particles_in_2 = [0]
particles_in_3 = [0]
particles_in_4 = [0]

P_BAR_cum = np.cumsum(P_BAR, axis=1)

while elapsed_time < simulation_time_units:

  if len(current_nodes) == 0:  # if no particle in the network, one has to enter first
    # Input Poisson Clock click
    input_click_waiting_time = -np.log(np.random.rand())/input_poisson_clock_rate
    current_nodes.append(0)
  else:
    # Input Poisson Clock click
    input_click_waiting_time = -np.log(np.random.rand())/input_poisson_clock_rate
    # Global Poisson Clock click
    global_click_waiting_time = -np.log(np.random.rand())/len(current_nodes)

    if global_click_waiting_time < input_click_waiting_time:  # then one particle moves

      t_next = global_click_waiting_time

      # probabilities based on particles in each node
      node_0_prob = current_nodes.count(0)/len(current_nodes)
      node_1_prob = current_nodes.count(1)/len(current_nodes)
      node_2_prob = current_nodes.count(2)/len(current_nodes)
      node_3_prob = current_nodes.count(3)/len(current_nodes)
      node_4_prob = current_nodes.count(4)/len(current_nodes)

      # node choice and choice of a particle in chosen node
      choosen_node = choice([0, 1, 2, 3, 4], 1, p=[node_0_prob, node_1_prob, node_2_prob,
node_3_prob, node_4_prob])[0]  # since choice returns a list
      particles_in_choosen_node = []
      for index in range(0, len(current_nodes)):
      # extraction of indeces associated to particles in the chosen node
        if current_nodes[index] == choosen_node:
          particles_in_choosen_node.append(index)
      particle = random.choice(particles_in_choosen_node)  # Random particle choice

      # Sending the particle to a new node according to P
      if choosen_node == 4:
        del current_nodes[particle]
      else:
        current_nodes[particle] = np.argwhere(P_BAR_cum[current_nodes[particle]] >
np.random.rand())[0][0]  # move to new node

    else:  # Then one particle enters
      t_next = input_click_waiting_time
      current_nodes.append(0)

  # Updating elapsed time
  elapsed_time += t_next

  # For visualization purposes
  plot_time.append(elapsed_time)
  particles_in_0.append(current_nodes.count(0))
  particles_in_1.append(current_nodes.count(1))
  particles_in_2.append(current_nodes.count(2))
  particles_in_3.append(current_nodes.count(3))
  particles_in_4.append(current_nodes.count(4))

node_0_count += current_nodes.count(0)
node_1_count += current_nodes.count(1)
node_2_count += current_nodes.count(2)
node_3_count += current_nodes.count(3)
node_4_count += current_nodes.count(4)

fig_1, ax = plt.subplots(figsize=(16, 10))
ax.plot(plot_time, particles_in_0, c='b', linestyle='-', label='Node O')
ax.plot(plot_time, particles_in_1, c='g', linestyle='-', label='Node A')
ax.plot(plot_time, particles_in_2, c='r', linestyle='-', label='Node B')
ax.plot(plot_time, particles_in_3, c='m', linestyle='-', label='Node C')
ax.plot(plot_time, particles_in_4, c='c', linestyle='-', label='Node D')
ax.legend(loc=(0.9, 0.8))  # position: 90% of the length, 50% of height
ax.set_xlabel('Time')
ax.set_ylabel('Particles')
plt.show()
```

VII)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

Lambda = [
[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, 1, 0]]  # modified to avoid singular matrix

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

w = np.sum(Lambda, axis=1)
w_star = np.max(w)
P_BAR = Lambda/w_star
P_BAR = P_BAR + np.diag(np.ones(len(w))-np.sum(P_BAR,axis=1))

input_poisson_clock_rate = 1
input_times = []  # list of times when to enter a particle
input_index = 0  # index that tracks next time to make a particle entering
simulation_time_units = 60
elapsed_time = 0
current_nodes = []  # particles in the network, index is the particle, value is the node

# For visualization of last iteration
plot_time = [elapsed_time]
particles_in_0 = [0]
particles_in_1 = [0]
particles_in_2 = [0]
particles_in_3 = [0]
particles_in_4 = [0]

P_BAR_cum = np.cumsum(P_BAR, axis=1)

while elapsed_time < simulation_time_units:  # computation of input times
  # Input Poisson Clock click
  input_click_waiting_time = -np.log(np.random.rand())/input_poisson_clock_rate
  elapsed_time += input_click_waiting_time
  input_times.append(elapsed_time)

elapsed_time = 0

while elapsed_time < simulation_time_units:
  if len(current_nodes) == 0:  # if no particle in the network, one has to enter first
    elapsed_time += input_times[input_index]  # Updating elapsed time
    input_index += 1
    current_nodes.append(0)

    # For visualization purposes
    plot_time.append(elapsed_time)
    particles_in_0.append(current_nodes.count(0))
    particles_in_1.append(current_nodes.count(1))
    particles_in_2.append(current_nodes.count(2))
    particles_in_3.append(current_nodes.count(3))
    particles_in_4.append(current_nodes.count(4))

  else:
    # Global Poisson Clock click
    global_click_waiting_time = -np.log(np.random.rand())/len(current_nodes)

    while elapsed_time + global_click_waiting_time <= input_times[input_index] and
len(current_nodes) != 0:
      elapsed_time += global_click_waiting_time

      # probabilities based on particles in each node
      node_0_prob = current_nodes.count(0)/len(current_nodes)
      node_1_prob = current_nodes.count(1)/len(current_nodes)
      node_2_prob = current_nodes.count(2)/len(current_nodes)
      node_3_prob = current_nodes.count(3)/len(current_nodes)
      node_4_prob = current_nodes.count(4)/len(current_nodes)

      # node choice and choice of a particle in choosen node
      choosen_node = choice([0, 1, 2, 3, 4], 1, p=[node_0_prob, node_1_prob, node_2_prob,
node_3_prob, node_4_prob])[0]  # since choice returns a list
      particles_in_choosen_node = []
      for index in range(0, len(current_nodes)):
        # extraction of indices associated to particles in the chosen node
        if current_nodes[index] == choosen_node:
          particles_in_choosen_node.append(index)
      particle = random.choice(particles_in_choosen_node)  # Random particle choice

      # Sending the particle to a new node according to P
      if choosen_node == 4:
        del current_nodes[particle]
      else:
        current_nodes[particle] = np.argwhere(P_BAR_cum[current_nodes[particle]] >
np.random.rand())[0][0]  # move to new node

      if len(current_nodes) != 0:
        # Global Poisson Clock click
        global_click_waiting_time = -np.log(np.random.rand())/len(current_nodes)

      # For visualization purposes
      plot_time.append(elapsed_time)
      particles_in_0.append(current_nodes.count(0))
      particles_in_1.append(current_nodes.count(1))
      particles_in_2.append(current_nodes.count(2))
      particles_in_3.append(current_nodes.count(3))
      particles_in_4.append(current_nodes.count(4))

    elapsed_time = input_times[input_index]  # now is the turn for a particle to enter
    input_index += 1
    current_nodes.append(0)

    # For visualization purposes
    plot_time.append(elapsed_time)
    particles_in_0.append(current_nodes.count(0))
    particles_in_1.append(current_nodes.count(1))
    particles_in_2.append(current_nodes.count(2))
    particles_in_3.append(current_nodes.count(3))
    particles_in_4.append(current_nodes.count(4))

fig_1, ax = plt.subplots(figsize=(16, 10))
ax.plot(plot_time, particles_in_0, c='b', linestyle='-', label='Node O')
ax.plot(plot_time, particles_in_1, c='g', linestyle='-', label='Node A')
ax.plot(plot_time, particles_in_2, c='r', linestyle='-', label='Node B')
ax.plot(plot_time, particles_in_3, c='m', linestyle='-', label='Node C')
ax.plot(plot_time, particles_in_4, c='c', linestyle='-', label='Node D')
ax.legend(loc=(0.9, 0.8))  # position: 90% of the length, 50% of height
ax.set_xlabel('Time')
ax.set_ylabel('Particles')
plt.show()
```

VIII)

```
import networkx as nx
import numpy as np
from numpy.random import choice, rand
import matplotlib.pyplot as plt
import sys
import random

Lambda = [
[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, 1, 0]]  # modified to avoid singular matrix

node_dict = {'o':0, 'a':1, 'b':2, 'c':3, 'd':4}

w = np.sum(Lambda, axis=1)
w_star = np.max(w)
P_BAR = Lambda/w_star
P_BAR = P_BAR + np.diag(np.ones(len(w))-np.sum(P_BAR,axis=1))

node_0_count = 0
node_1_count = 0
node_2_count = 0
node_3_count = 0
node_4_count = 0

input_poisson_clock_rate = 1
simulation_time_units = 60
elapsed_time = 0
current_nodes = []  # index is the particle, value is the node

# For visualization of last iteration
plot_time = [elapsed_time]
particles_in_0 = [0]
particles_in_1 = [0]
particles_in_2 = [0]
particles_in_3 = [0]
```

```
particles_in_4 = [0]

P_BAR_cum = np.cumsum(P_BAR, axis=1)

while elapsed_time < simulation_time_units:

  if len(current_nodes) == 0:  # if no particle in the network, one must enter first
    # Input Poisson Clock click
    input_click_waiting_time = -np.log(np.random.rand())/input_poisson_clock_rate
    current_nodes.append(0)
  else:
    # Input Poisson Clock click
    input_click_waiting_time = -np.log(np.random.rand())/input_poisson_clock_rate
    # Global Poisson Clock click with rate number of nodes with at least a particle
    global_click_waiting_time = -np.log(np.random.rand())/len(set(current_nodes))

    if input_click_waiting_time < global_click_waiting_time:
      t_next = input_click_waiting_time
      current_nodes.append(0)
    else:
      t_next = global_click_waiting_time

      if len(current_nodes) > 0:  # If there is at least particle in the network

        # node choice and choice of a particle in chosen node
        choosen_node = choice([0, 1, 2, 3, 4], 1)[0]  # since choice returns a list

        # if there is a particle it moves, else nothing happens
        if current_nodes.count(choosen_node) > 0:
          particles_in_choosen_node = []
          # extraction of indices associated to particles in the chosen node
          for index in range(0, len(current_nodes)):
            if current_nodes[index] == choosen_node:
              particles_in_choosen_node.append(index)

          # Random particle choice in chosen node
          particle = random.choice(particles_in_choosen_node)

          # Sending the particle to a new node according to P
          if choosen_node == 4:
            del current_nodes[particle]
          else:
            current_nodes[particle] = np.argwhere(P_BAR_cum[current_nodes[particle]] >
np.random.rand())[0][0]  # move to new node

  # Updating elapsed time
  elapsed_time += t_next

  # For visualization purposes
  plot_time.append(elapsed_time)
  particles_in_0.append(current_nodes.count(0))
  particles_in_1.append(current_nodes.count(1))
  particles_in_2.append(current_nodes.count(2))
  particles_in_3.append(current_nodes.count(3))
  particles_in_4.append(current_nodes.count(4))

node_0_count += current_nodes.count(0)
node_1_count += current_nodes.count(1)
node_2_count += current_nodes.count(2)
node_3_count += current_nodes.count(3)
node_4_count += current_nodes.count(4)

fig_1, ax = plt.subplots(figsize=(16, 10))
ax.plot(plot_time, particles_in_0, c='b', linestyle='-', label='Node 0')
ax.plot(plot_time, particles_in_1, c='g', linestyle='-', label='Node A')
ax.plot(plot_time, particles_in_2, c='r', linestyle='-', label='Node B')
ax.plot(plot_time, particles_in_3, c='m', linestyle='-', label='Node C')
ax.plot(plot_time, particles_in_4, c='c', linestyle='-', label='Node D')
ax.legend(loc=(0.9, 0.8))  # position: 90% of the length, 50% of height
plt.show()
```