Network Dynamics and Learning

Homework 2

2023-2024

Carena Simone, s309521
Carmone Francesco Paolo, s308126
Mazzucco Ludovica, s315093

This document has been produced jointly by the students mentioned above, who have tightly collaborated throughout the whole homework.

Exercise 1

The graph used throughout this exercise is the following one

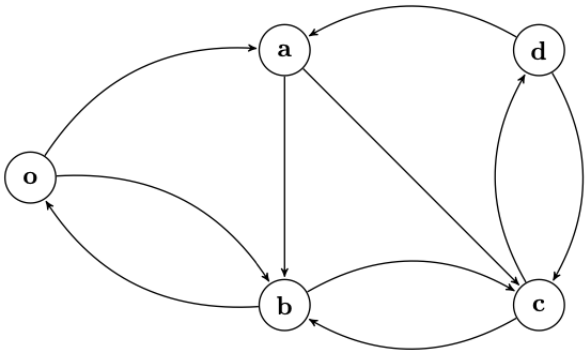


Figure 1: Exercise 1 Graph

which is described by the transition matrix $\mathbf{\Lambda}$

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

Part I

To simulate the random walks over the graph a function `randomWalk` (1) has been implemented. It takes into account three possible scenarios: the first one in which the walk is performed until the particle returns to the starting node, a second one that terminates when the particle reaches a specific node, and the last one that continues until a certain time has elapsed. The algorithm can be described as follows:

1. Wait a random time $t_{\text{next}} = -\log(u)/r_i$ in the current node, where $u \sim \mathcal{U}(0,1)$ and $r_i = (\mathbf{\Lambda 1})_i$
2. Choose the node j where to move the particle, among the successors of the current node i. The next node j is chosen with probability $p = \mathbf{P}_{ij}$, where $\mathbf{D}^{-1}\mathbf{\Lambda}$
3. Based on the task to perform the function assumes three different forms:
   - **Stop at Return** The walk stops when the particle return to the node it started from. Move to the new node j. If the current node j coincides with the starting node terminate the walk, otherwise repeat from point 1
   - **Stop at Target** The walk stops when the particle reaches a specified target node. Move to the new node j. If the current node j coincides with the target node terminate the walk, otherwise repeat from point 1
   - **Stop After a Certain Time** The walks stops after a certain time (marked by the particle's Poisson clock) is elapsed. If the time t elapsed from the start of the walked is greater or equal then a certain specified time terminate the walk, otherwise move to the new node j and repeat from point 1

This is its pseudo-code:

Listing 1: Random Walk

```
visited_nodes = []
waiting_times = []
visited_nodes.append(starting_node)
current_node = starting_node
elapsed_time = 0

Loop forever:
    r = out_degree(current_node)
    t_next = -log(uniform((0,1)))/r
    waiting_times.append(t_next)
    elapsed_time += t_next
    next_node = choose(successors(current_node), p=P[idx(current_node)])
    Switch termination_condtion:
        Case stop_at_return:
            visited_nodes.append(next_node)
            if next_node == starting_node:
                break
            current_node = next_node
        Case stop_at_target:
            visited_nodes.append(next_node)
            if next_node == target_node:
                break
            current_node = next_node
        Case stop_at_elapsed:
            it elapsed_time >= target_time:
                break
            visited_nodes.append(next_node)
            current_node = next_node

return (visited_nodes, waiting_times)
```

Point A

Once the function `randomWalk` (1) is given, to answer this point it is sufficient to enclose it in a "for" loop running multiple times and launching it setting node b as starting node, specifying also the computed random walk has to stop once the particle returns back to it (by setting the boolean parameter *stop_at_return* to *True*) instead of going on with the simulation for a pre-determined period of time. At every iteration, the total travel time of each sample trajectory is collected in a vector in order to average all its elements once the loop ends. Thus, the average return time for the particle starting in b is :

$$T_b = 4.586232216312701 \qquad \text{(Eq. 1)}$$

Point B

For continuous time Markov Chains and in the case of a strongly connected graph as 1, i.e. with a unique invariant distribution, as a consequence of the *Ergodic Theorem* one can theoretically find the expected return time of node i as

$$E[T_i] = \frac{1}{\bar{\pi}_i \omega_i}$$

The effort is just to obtain the invariant distribution vector $\bar{\pi}$, which corresponds to the eigenvector of $\bar{P}$ associated to the eigenvalue 1. The out-degree of node i is indicated with ω_i and it can be directly retrieved by the transition-rate matrix Λ by summing its rows. All in all, the computations lead to the following result:

$$E[T_b^+] = \frac{1}{\bar{\pi}_b \omega_b} = 4.599999999999998$$

This demonstrates what obtained in Eq. 1

Point C

Instead of just simulating random walks ending in the staring node, function `randomWalk` (1) can be also exploited to draw out the time it takes a particle to end up in a given other node in the network. In order to find the average time needed to move a particle from node o to node d, it is possible to reiterate the same steps as in Point A with the only difference that the function is called with the boolean argument *stop_at_node* set to $True$ and it is provided with the node the random walk has to stop in. The average hitting time thus results:

$$T_o(d) = 10.615884285022757 \qquad \text{(Eq. 2)}$$

Point D

For the purpose of theoretically justify the previous result, the following system of equations is exploited:

$$\begin{cases} E_i[T_S] = 0 & i \in S \\ E_i[T_S] = \frac{1}{\omega_i} + \sum_j P_{ij} E_j[T_S] & i \notin S \end{cases}$$

where in our case the set S just contains node d. Thus, it seems computationally more efficient to solve the forementioned systems of equation for each node in the network via matrix form, like it is shown here.

$$\underbrace{\begin{bmatrix} E_o \\ E_a \\ E_b \\ E_c \end{bmatrix}}_{E} = \underbrace{\begin{bmatrix} \frac{1}{\omega_o} \\ \frac{1}{\omega_a} \\ \frac{1}{\omega_b} \\ \frac{1}{\omega_c} \end{bmatrix}}_{W} + \tilde{P} \underbrace{\begin{bmatrix} E_o \\ E_a \\ E_b \\ E_c \end{bmatrix}}_{E}$$

where with $\tilde{P}$ we intended the normalized rate transition matrix Λ without the last row and last column (the one associated to node d), since by definition $E_d[T_d] = 0$. We can get the desired expected hitting time $E_o = E_o[T_d]$ by rearranging the matrix equation above:

$$E - PE = W$$
$$(I_{4x4} - \tilde{P})E = W$$
$$E = (I_{4x4} - \tilde{P})^{-1} W$$
$$E_o[T_d] = E[0,0] = 10.767$$

Clearly, this reflects the result obtained in Eq. 2

Part II, Opinion Dynamics

The evolution or trajectory over time of a linear system is described by its dynamics matrix. In this part, we will focus on the so called *French De-Groot dynamics*, as shown here:

$$x(t) = P^t x(0) \qquad \text{(Eq. 3)}$$

where $P = D^{-1}\Lambda$ is the normalized transition matrix of the previous part.

Point E

The dynamics converge to a consensus state for every initial $x(0)$ because it satisfies the hypotesis of the French-De Groot theorem: the graph G is strongly connected and aperiodic, which assures us that the dynamics (Eq. 3) will converge to $\alpha\mathbf{1}$, where α is a term that depends on the initial condition x_0, and is defined as:

$$\alpha = \pi^T x(0) = \sum_k \pi_k x_k(0), \quad \forall i \qquad \text{(Eq. 4)}$$

We choose randomly 10 initial opinion vectors by sampling a uniform distribution, and for each one of them we ran the dynamics for 100 iterations. Those are the results:

```
0: x(0): [0.218 0.300 0.486 0.163 0.101], x: 0.275946 [1 1 1 1 1]
1: x(0): [0.693 0.759 0.972 0.607 0.498], x: 0.730904 [1 1 1 1 1]
2: x(0): [0.061 0.571 0.424 0.115 0.983], x: 0.407822 [1 1 1 1 1]
3: x(0): [0.176 0.663 0.032 0.596 0.477], x: 0.371147 [1 1 1 1 1]
4: x(0): [0.211 0.223 0.536 0.475 0.675], x: 0.433744 [1 1 1 1 1]
5: x(0): [0.589 0.066 0.316 0.862 0.081], x: 0.404542 [1 1 1 1 1]
6: x(0): [0.136 0.299 0.412 0.152 0.333], x: 0.274713 [1 1 1 1 1]
7: x(0): [0.830 0.445 0.231 0.102 0.237], x: 0.338952 [1 1 1 1 1]
8: x(0): [0.447 0.751 0.351 0.987 0.281], x: 0.576723 [1 1 1 1 1]
9: x(0): [0.992 0.362 0.018 0.254 0.952], x: 0.437807 [1 1 1 1 1]
```

We reach consensus!

Point F

The initial opinon of the origin and destination nodes has been chosen by sampling from a gaussian of zero mean, and variance $\sigma^2 = 2$, while the rest of the nodes have been initialized by sampling from a gaussian of zero mean, and variance $\sigma^2 = 1$. The variance of the consensus value has been calculated in two different ways: analytically and numerically. The former by applying the formula $\text{Var}(\sum \pi_k N_k) = \sum \sigma_k^2 \pi_k^2$ where π_k is the k-th component of the *invariant distribution* (unique for $\mathcal{G}$), and σ_k^2 is the variance of the earlier mentioned gaussians. The latter by applying the definition, i.e. $Var(X = x) = \mathbb{E}([X - \mathbb{E}[X]])^2$, where X contained all the collected consensus values (one for each of the 100 iterations). The results are:

```
pi = [0.16153846 0.18461538 0.26923077 0.23076923 0.15384615]
(Analytical) variance:  0.259349
(Numerical)  variance:  0.265733
```

Point G

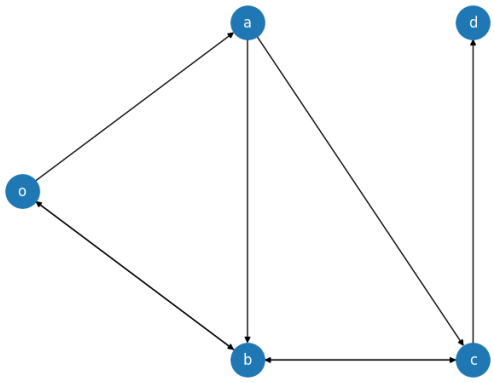


Figure 2: Graph obtained starting from Graph (1) and removing the edges (d, a) and (d, c)

Removing the edges (d, a) and (d, c) made the weight matrix Λ singular, hence there's no way to calculate the normalized weight matrix P. Moreover, the graph is no longer strongly connected: we don't satisfy anymore the hypothesis to apply the French-De Groot theorem. However, by adding a self loop of unitary weight to node d, we are able to obtain a new graph (3) and use the *generalized theorem*, that states that if a graph $\mathcal{G}$ possesses a globally reachable aperiodic

component, then every component of the opinion vector will converge to the dot product between π and the initial condition $x(0)$:

$$\lim_{t \to +\infty} x_i(t) = \pi^T x(0), \quad \forall i \qquad \text{(Eq. 5)}$$

The variable $\pi = P^T \pi$ is the invariant measure centrality of the graph. In this case, the distribution is equal to $\pi = [0\ 0\ 0\ 0\ 1]$, because the globally reachable component is composed just by the node d and we can notice that the opinions of the agents not belonging to the globally reachable component have no influence on the final consensus value. That equivalence holds true because $P^t \to \mathbf{1}\pi^T$ will converge to a matrix where all the rows are the same, and equal to π.

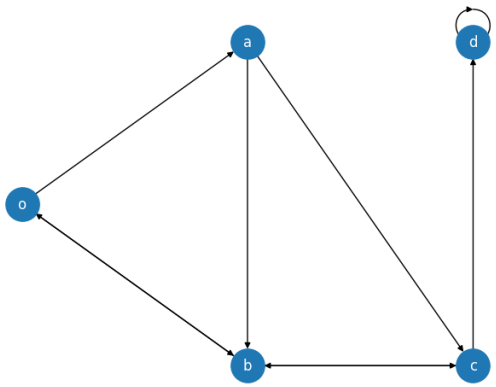


Figure 3: Graph obtained from Graph (2) by adding a self loop to obtain a globally reachable component

Given those initial conditions

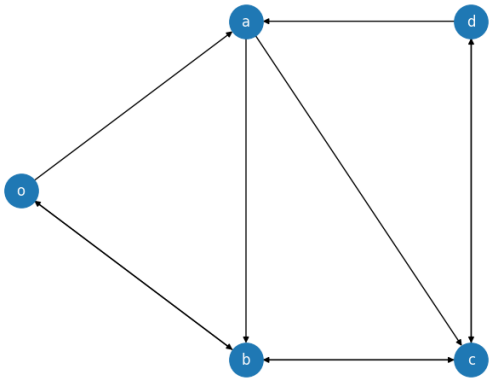
```
0: x(0): [0.32451666 0.8367452  0.6545018  0.3204357  0.61283144]
1: x(0): [0.63576166 0.50488599 0.79559383 0.30238905 0.41662024]
2: x(0): [0.41917593 0.73970436 0.6463774  0.73645559 0.82208617]
3: x(0): [0.78408677 0.81051243 0.49110915 0.96764933 0.86741029]
4: x(0): [0.40973832 0.20560624 0.46864516 0.08323009 0.68121801]
5: x(0): [0.72331821 0.78830216 0.25129057 0.61573844 0.73867747]
6: x(0): [0.39955749 0.09026706 0.29894955 0.63743237 0.54380162]
7: x(0): [0.55544712 0.36424429 0.85029741 0.22633007 0.73094473]
8: x(0): [0.37208915 0.26658816 0.85091167 0.67821425 0.10780403]
9: x(0): [0.83524572 0.45892933 0.53048297 0.476886   0.97716042]
```

The opinion will reach consensus according to Eq. 5:

```
0: x: [0.61283143 0.61283143 0.61283143 0.61283144 0.61283144]
1: x: [0.41662028 0.41662027 0.41662027 0.41662026 0.41662024]
2: x: [0.82208613 0.82208614 0.82208614 0.82208616 0.82208617]
3: x: [0.86741027 0.86741027 0.86741027 0.86741029 0.86741029]
4: x: [0.68121795 0.68121796 0.68121795 0.68121799 0.68121801]
5: x: [0.73867744 0.73867744 0.73867744 0.73867746 0.73867747]
6: x: [0.54380158 0.54380159 0.54380159 0.54380161 0.54380162]
7: x: [0.73094470 0.7309447  0.7309447  0.73094472 0.73094473]
8: x: [0.10780412 0.10780411 0.10780411 0.10780406 0.10780403]
9: x: [0.97716035 0.97716036 0.97716036 0.9771604  0.97716042]
```

Point H

These are the results obtained by running the French De-Groot dynamics (Eq. 3) on this graph:




```
0: x: [0.1402396  0.1402396  0.13450618 0.14979529 0.12877277]
1: x: [0.86336953 0.86336953 0.82858967 0.92133595 0.79380982]
2: x: [0.43307406 0.43307406 0.63118205 0.10289408 0.82929003]
3: x: [0.30368936 0.30368936 0.42016000 0.10957164 0.53663063]
4: x: [0.46391683 0.46391683 0.26167891 0.80098003 0.05944099]
5: x: [0.51237218 0.51237218 0.55794614 0.43641557 0.60352010]
6: x: [0.49679289 0.49679289 0.28637962 0.84748166 0.07596635]
7: x: [0.69754093 0.69754093 0.67393929 0.73687700 0.65033765]
8: x: [0.76482601 0.76482601 0.83413517 0.64931075 0.90344432]
9: x: [0.36711999 0.36711999 0.22599409 0.60232981 0.08486819]
```

starting from these initial opinions:

```
0: x(0): [0.74640191 0.13493035 0.93957376 0.14979529 0.12877277]
1: x(0): [0.00441832 0.22140661 0.23684450 0.92133595 0.79380982]
2: x(0): [0.09228566 0.54922039 0.62565038 0.10289408 0.82929003]
3: x(0): [0.86021949 0.08963199 0.96637501 0.10957164 0.53663063]
4: x(0): [0.65366834 0.34196435 0.71636516 0.80098003 0.05944099]
5: x(0): [0.24390016 0.09762964 0.35045534 0.43641557 0.60352010]
6: x(0): [0.44694957 0.65346370 0.63373851 0.84748166 0.07596635]
7: x(0): [0.07615568 0.85409446 0.20533760 0.73687700 0.65033765]
8: x(0): [0.16720877 0.33609519 0.36266600 0.64931075 0.90344432]
9: x(0): [0.04766658 0.96689239 0.45030419 0.60232981 0.08486819]
```

We can observe that the dynamic diverges. That is because we can't apply the French-De Groot model since the graph isn't strongly connected, and we can't apply the generalized result (Eq. 5) either because the globally reachable component isn't aperiodic (4).

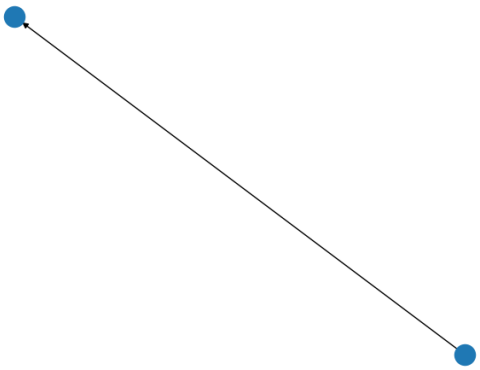


Figure 4: The condensed graph has two components, (`'c', 'd'`) and (`'b', 'a', 'o'`)

It's interesting to notice that the components made out of the nodes (`'o', 'a', 'b'`) almost reaches a consensus, and that happens because that component is aperiodic.

Exercise 2

For this task we consider the same network as before (1). The analysis of the system takes into account two distinct perspectives: the particle's perspective and the node's perspective. To simulate the former, we use the same previously mentioned `randomWalk` function (1), while to simulate the latter, we adopt a global Poisson clock instead of individual clocks for each node. The function `randomWalk_gloablClock` then becomes the following:

1. Initialize the system by considering the starting node to contain n particles
2. Wait a random time $t_{\text{next}} = -\log(u)/r$ in the current node, where $u \sim \mathcal{U}(0,1)$ and $r = \omega^* n$, with $\omega^* = \max\{\mathbf{\Lambda 1}\}_i$
3. If the specified maximum simulation time has elapsed terminate the simulation, otherwise proceed to the next step
4. Once the clock has ticked, choose the node i to activate with probability $p \propto n_i$, where n_i is the number of particle in node i
5. Choose the node j where to move the particle with probability $\bar{P}_{ij}$. The $\bar{P}$ matrix is used, since having a global Poisson clock overestimates the activation frequency, and thus using the $\bar{P}$ matrix takes into account the probability of node i not actually activating (this is represented by having a self-loop in each node that expresses the probability of each particle to remain in the current node when it activates)
6. Repeat from point 2

Its pseudo code is

Listing 2: Random Walk with a global Poisson clock

```
elapsed_time = 0
waiting_times = []
nodes_particles = {node:[] for node in nodes}
node_particles[starting_node] = n
node_particles[!starting_node] = 0
r = w_star*n

Loop Forever:
    t_next = -log(uniform((0,1)))/r
    elapsed_time += t_next
    waiting_times.append(t_next)
    if elapsed_time >= simulation_time:
        return (waiting_times, node-particles)
    p = node_particles[:][end]/n
    activated_node = choose(nodes, p)
    next_node = choose(successors(activated_nodes),p=Pbar[idx(current_node)])
    node_particles[activated_node].append(node_particles[activated_node]-1)
    node_particles[next_node].append(node_particles[next_node]+1)
    node_particle[other nodes] = node_particles[other_nodes][end]
```

Point A

Assuming the independence of random walks among particles, determining the average return time to node b for a set of 100 particles is analogous to simulating a single particle's journey and repeating the simulation a hundred times. This approach mirrors the methodology employed in Problem 1, utilizing the `randomWalk` function (1). The outcome is the following, and it is coherent with the one in Eq. 1:

$$T_b = 4.600789463264558$$

Point B

As outlined in the introductory section of this problem, a distinct methodology is necessary for calculating the average number of particles in various nodes from a node-centric perspective. This approach involves considering a global Poisson clock, which ticks at the rate of the node with the maximum out-degree. We make use of the function `randomWalk_globalClock` (2) passing as arguments the starting node (o), the total number of particles wandering (100), the time interval (60 time units) and lastly ω_* (1), collecting then the resulting lists for each node containing the number of particles actually present in each node at each time step and finally computing the mean on them. The output is proposed here:

```
average number of particles in o:  24.911427186035237
average number of particles in a:  14.982544043963149
average number of particles in b:  27.084855341845806
average number of particles in c:  16.635526103119442
average number of particles in d:  16.385647325036366
```

The graph below shows the evolution of particles number inside each node over the whole time interval

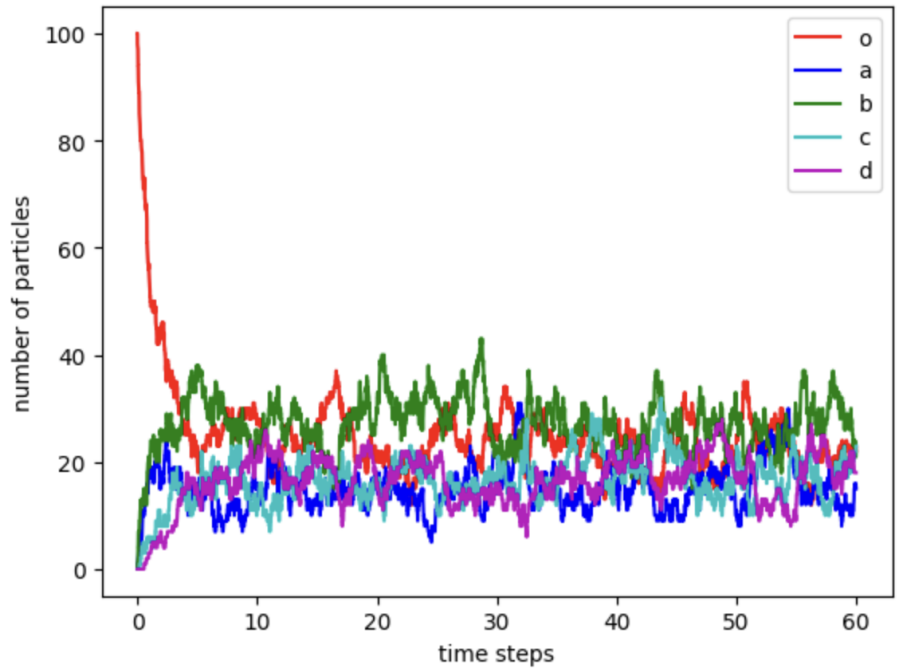


Figure 5: Evolution in time of the number of particles in each node

A theorem states that given a strongly connected graph and any initial probability distribution $\bar{\pi}(0)$, then $\lim_{t\to\infty} \bar{\pi}(t) = \bar{\pi}$. We verified the result by computing $\bar{\pi}(t)$ by saving the distribution of the particles after 60 timesteps and normalizing it, and then compared it with the network's graph stationary distribution.

```
pi_60 =   [  0.22        0.15        0.22        0.23        0.18      ]
pi_bar =  [  0.2173913   0.14906832  0.26086957  0.1863354   0.1863354 ]
```

Had the simulation lasted longer, the results would have been even more accurate.

Exercise 3

The graph used throughout this task is the following one:

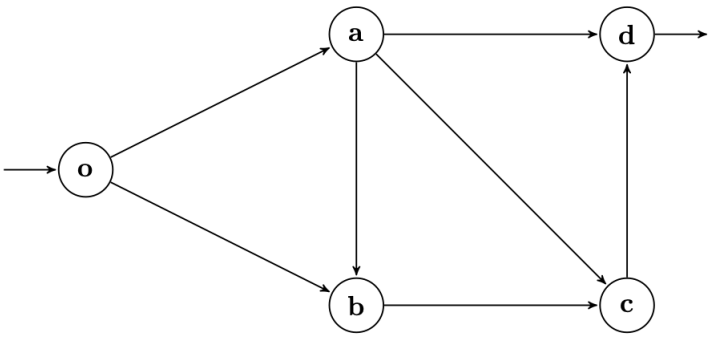


Figure 6: Exercise 1 Open Graph

which is described by the transition matrix $\mathbf{\Lambda}$

$$\mathbf{\Lambda} = \begin{pmatrix} 0 & 3/4 & 3/4 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 2/4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The case under study is an *open network*: the particles will enter the system at node o according to a Poisson process with input rate λ, and leave the system from node d. The system is studied considering two different scenarios: the first one in which the nodes have a varying proportional rate $\omega_i n_i$, where n_i is the number of particles currently in node i, and the second one in which the nodes have it fixed to ω_i.

Point A

To simulate the first scenario, the one in which nodes Poisson clock ticks at proportional rate, we have to implement yet another function, `randomWalk_globalClock_prop` (3), in which a global clock ticking at rate $\omega_* N_{\text{particles}}$ is considered to manage nodes transitions, exploiting $\bar{P}$ for transition probabilities. The algorithm underlying its functioning can be described by the following pseudo-code:

Listing 3: Random Walk on a Open Network with Global Clock

```
initialize:
nparticles=0
timer=0
times=[]
nodes_distr={ n:[] for n in G.nodes}
nodes_rate=wstar

loop forever:
    nodes_rate = nparticles * wstar
    choose which clock ticks first (input clock or nodes clock) given the
        probabilities w.r.t. their rates
    compute next clock tick
    add to timer the calculated time step
    if time elapsed:
        return (times, nodes_distr)
    append to times the current clock tick
```

```
16
17      if input clock tick first:
18          update nparticles
19          update nodes_distr adding one particle to node o and leaving others
                unchanged
20      else :
21          if network is empty :
22              nothing to move, update lists in nodes_distr appending their last
                    element
23              continue to next iteration
24          choose proportionally the node from which moving the particle
25          if chosen node is d and it has at least one particle inside:
26              decrease by one nparticles
27              update correspondingly d list in nodes_distr
28          else:
29              choose target node to move the particle to according to Pbar
30              if the target node has at least one particle inside:
31                  apply the transition and update lists in nodes_distr accordingly
```

Once the output dictionary is retrieved, it is straightforward to plot the variation in time of the number of particles in the network and in the nodes:

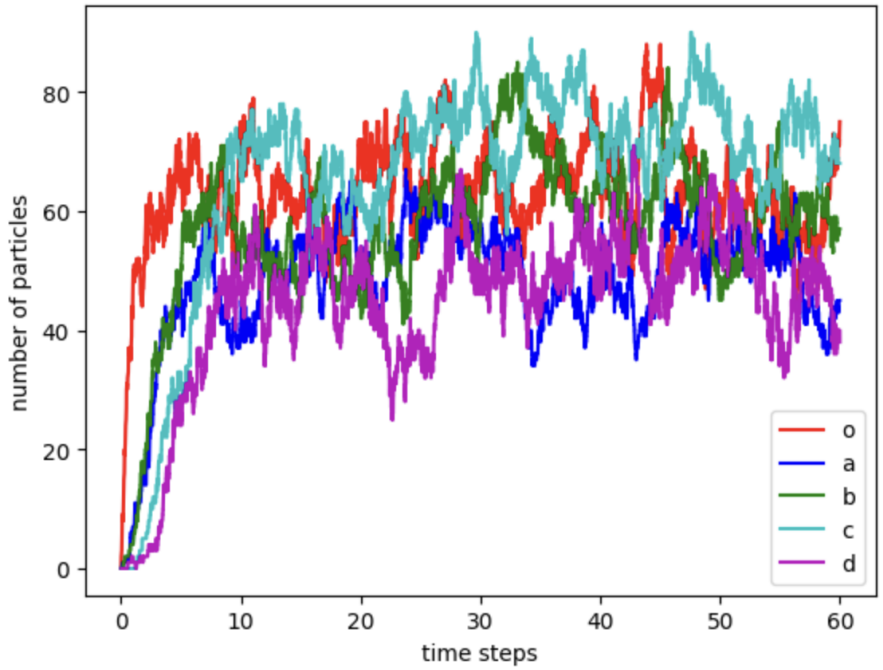


Figure 7: Evolution in time of the number of particles in each node with $\lambda = 100$

If the input rate λ is increased to an exaggerated value, say 1000, in order to inspect the changes in particles behaviour, then the plot will look like this:

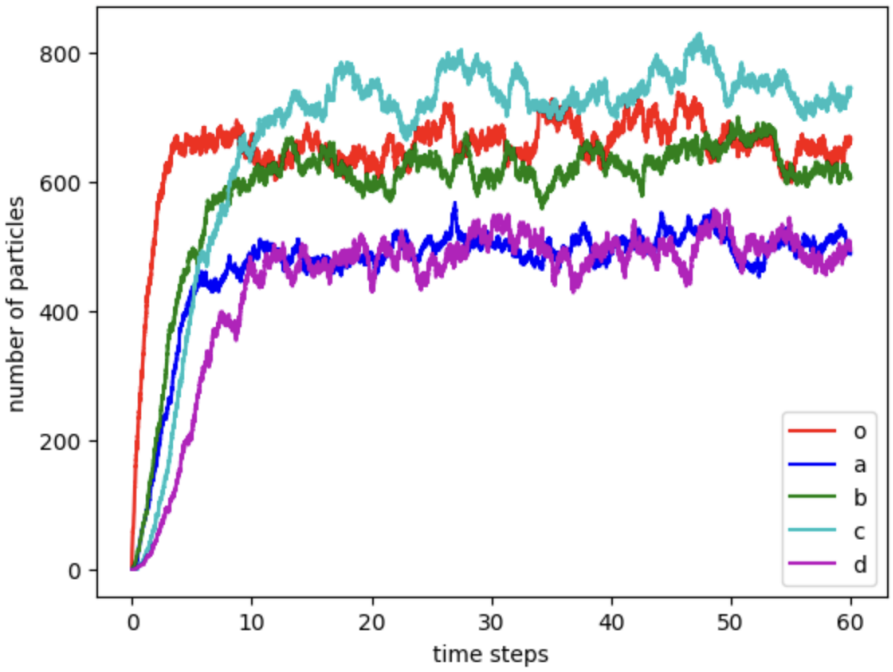


Figure 8: Evolution in time of the number of particles in each node $\lambda = 1000$

As the latter figure suggests, there is no substantial variation with the previous one with an input rate far too slower. This is the consequence of the fact that the nodes global clock is proportional to the total number of particles in there and so the more the entering particles the more the ones exiting, leading to a sort of network homeostasis.

Point B

In what this second scenario is concerned, that is considering the nodes transition rate fixed, the function `randomWalk_localClocks_fix` (4) is exploited. Unlike the previous case, here the chosen approach makes use of local clocks instead of a unique global one and each time every single clock (included the input one) is simulated, the one ticking first is selected for the corresponding transition. The pseudo-code of the function clearly explains the details.

Listing 4: Random Walk on a Open Network with Local Clocks

```
initialize:
nparticles=0
timer=0
times=[]
nodes_distr={ n:[] for n in G.nodes}

loop forever:
    tnexts=[]
    for each node rate:
        simulate next tick
        append to tnexts
    simulate next input clock tick
    append it to tnexts
    choose which clock ticks first in tnexts
    add to timer the calculated time step
    if time elapsed:
        return (times, nodes_distr)
    append to times the current clock tick

    if input clock tick first:
        update nparticles
```

```
        update nodes_distr adding one particle to node o and leaving others
            unchanged
    elif d clock ticks first and d has at least one particle:
        decrease by 1 nparticles
    else :
        if network is empty :
            nothing to move, update lists in nodes_distr appending their last
                element
            continue to next iteration
        else:
            choose target node to move the particle to according to P
            if the target node has at least one particle inside:
                apply the transition and update lists in nodes_distr accordingly
```

Even in this case different plots are reported to make a comparison with respect to the effect on the system of the increasing input rate:

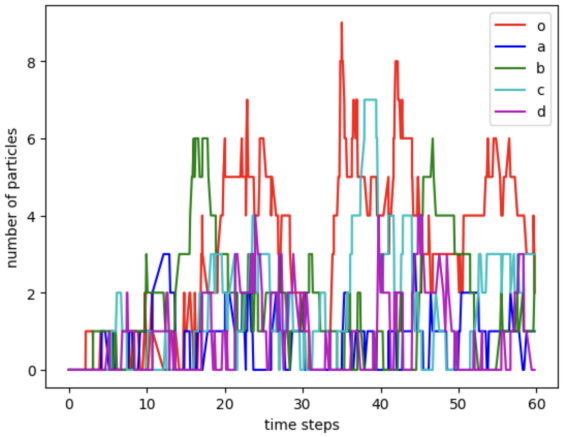


Figure 9: Evolution in time of the number of particles in each node $\lambda = 1$

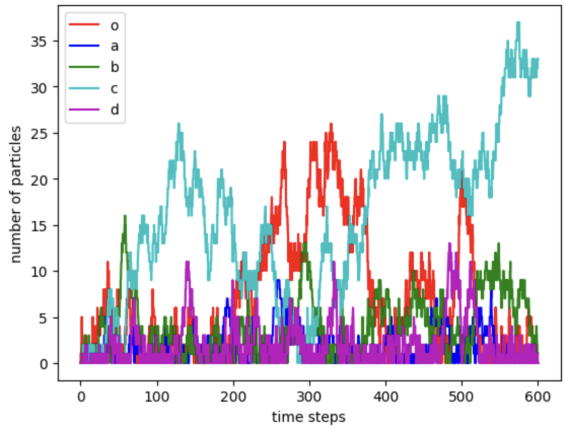


Figure 10: Evolution in time of the number of particles in each node $\lambda = 1.33$

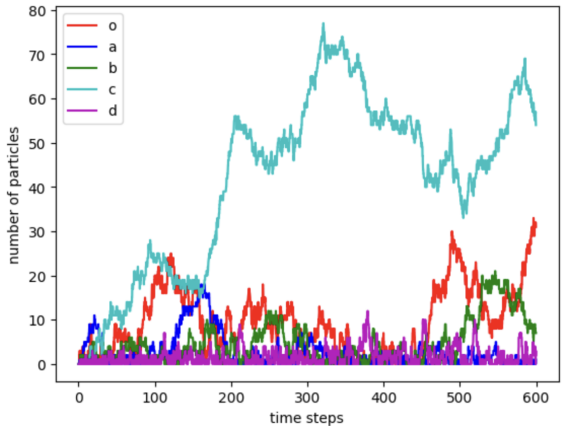


Figure 11: Evolution in time of the number of particles in each node $\lambda = 1.5$

As the figures highlight, with $\lambda = 1$ the system is stable, with $\lambda = 1.33$ node c starts to diverge and lastly with $\lambda = 1.5$ even node o begins to grow in the number of particles.
In order to explain theoretically the underlying reason of this behaviour, we can think at this network as the union of five $M/M/1$ queues, each corresponding to a node, with an aggregate input rate λ_i and a service rate μ_i represented by the i-th node transition rate ω_i. For every node i the following formula holds:

$$\lambda_i = \eta_i + \sum_{j=1}^{N} \bar{P}_{ji}\lambda_j$$

where η_i is the exogenous in-flow in node i if present; in our case this is indeed λ entering the network in o. What we desire to do is to solve the system composed by equations like the one above to draw out the aggregate input rate for each node. All in all, the system we refer to is the following one:

$$\begin{cases} \lambda_o = \eta \\ \lambda_a = 0.5\lambda_o \\ \lambda_b = 0.25\lambda_a + 0.5\lambda_o \\ \lambda_c = 0.25\lambda_a + \lambda_b \\ \lambda_d = 0.5\lambda_a + \lambda_c \end{cases}$$

Solving this for each λ_i we get:

$$\begin{cases} \lambda_o = \eta \\ \lambda_a = 0.5\eta \\ \lambda_b = 0.625\eta \\ \lambda_c = 0.75\eta \\ \lambda_d = \eta \end{cases}$$

At this point, we have to verify for each node the stability conditions with respect to η which is our exogenous input rate, this means proving that $\rho_i = \frac{\lambda_i}{\omega_i} < 1$. Hence, all those inequalities have to hold:

$$\begin{cases} \eta < 1.5 \\ \eta < 2 \\ \eta < 1.6 \\ \eta < 1.33 \\ \eta < 2 \end{cases}$$

Thus, the most restrictive of them is the one corresponding to node c, therefore in order to not let the system diverge $\eta < 1.33$. Finally, Figure 11 underlines that setting $\lambda = 1.5$ breaks even the first inequality condition corresponding to node o, leading to instability.