# Homework 2

Vincenzo Marciano' (282004)

December 12, 2021

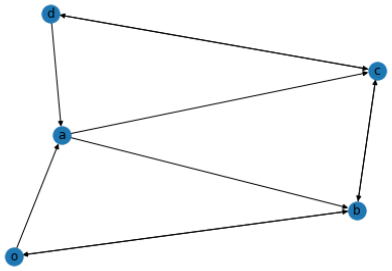
**Abstract**

In the second Homework, we are supposed to analyze how to simulate the continuous-time Markov chains (CTMC).

The exercises are made in collaboration with Alessandro Pecora (290369) and Andrea Ferretti (289677).

## Exercise 1

A single particle and its trajectory are give. The latter is defined by a random walk in a graph $\mathcal{G}$ and a certain transition matrix $\Lambda$, both displayed in Figure 1.




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

Figure 1: Graph $\mathcal{G}$ and transistion matrix $\Lambda$

An important notion to take in consideration is the fact we're operating in the domain of *Random Walks*, with particular attention to *continuous time applications*

### 1.a

To address the first requested task, i.e. computing experimentally the average time it takes a particle to start in node $a$ and to return in it, we actually need 3 different ingredients:

- **Poisson Clock of the imputed node**, a Poisson process with rate $r$ that determines the jump-time by exploiting its ticks. We call $t_{next}$ the time between two consecutive ticks and we define it as:

$$t_{next} = -\frac{\ln(u)}{r}$$

$$u \in \mathcal{U}(0,1), \text{a random variable with uniform distribution}$$

- $\omega$ that is a weight vector containing all the rates belonging to Poissons clocks. In particular:

$$\omega_i = \sum_j \Lambda_{ij}$$

- $\bar{\mathbf{P}}$, a matrix containing the conditional probabilities referred to the need of staying in the same node $i$ or jumping from $i$ to $j$ with the *global* Poisson clock ticking:

$$\bar{P}_{ii} = 1 - \sum_{i \neq j} \bar{P}_{ij}$$

$$\bar{P}_{ij} = \frac{\Lambda_{ij}}{\omega^*}$$

Here the global clock rate coincides with $\omega^* = max_i(\omega_i)$.

Setting an arbitrarily large number of simulations (e.g. 10000) and building a function that runs through a pipeline which operates with the previous described items, the process ends up in an **average return time** of **6.744 time units**. The key idea here was to extract a random number in [0,1], compare it with the *(i-1)-th position* of the cumulative $\bar{P}$ and pick the first smallest state for which the cumulative sum is greater than the random number. Then the process is repeated, by filling the vector of time instances until we run out of available simulations. Each time interval is provided by the function displayed in Listing 1.

```
def particle_routing_time(start_node,end_node,w_start,P_bar):
    start_node_idx=nodeToIdx[start_node]
    end_node_idx=nodeToIdx[end_node]
    #set the number of steps of simulation
    n_steps=1000
    #pos keep the trace of the visited nodes
    pos=np.zeros(n_steps,dtype=int)

    pos[0]=start_node_idx #start from state 0 and node a in position 1

    transition_times=np.zeros(n_steps) #store the time for esch transition
    t_next= -np.log(np.random.rand())/w_star
    Q_cum=np.cumsum(Q,axis=1) #cumulative matrix of row of Q

    for i in range(1,n_steps):
        pos[i]=np.argwhere(Q_cum[pos[i-1]]>np.random.rand())[0]

        #store the time instant of the current transition
        transition_times[i]=transition_times[i-1]+t_next
        #recopute the waiting time of the next transition
        t_next=-np.log(np.random.rand()/w_star)

        #If the state come back to the original state stop
        if pos[i]==end_node_idx:
            #print("The state comes back to the origin in: ",transition_times[i])
            return transition_times[i]
```

Listing 1: Function to compute the particle routing time in a random walk from "start_node" to "end_node"

However, this is not the only available approach: one can use the *local clock rate* ($\omega_i$) combined with the *stochastic matrix* $P$, which instead will be exploited in the next tasks.

## 1.b

This new task wants to assess the comparison between the already computed average time and the theoretical one. For this purpose we must add one more tool to use for our convenience: it's the case of **stationary probability vector** $\bar{\pi}$, in which the *i-th* element is represented by the probability in being at that specific node.

Here **Kac's formula** for computing return time of node $a$ in these context comes in handy, after satisfying the effective strongly connection of our graph $\mathcal{G}$:

$$E_a[T_a^+] = \frac{1}{w_a \bar{\pi}_a}$$

Where $\bar{\pi} = \bar{P}'\bar{\pi}$. The following code in Listing 2 provides a better explanation of the entire process, in which the key idea of computing $\bar{\pi}$ was to solve the previous equation by exploiting the eigendecomposition of the already computed matrix $\bar{P}$:

```
    def return_time(start_node):

    eig_vals,eig_vecs=np.linalg.eig(P_bar.T)
    pi_bar=eig_vecs[:,np.argmax(eig_vals.real)].real
    pi_bar=pi_bar/np.sum(pi_bar) #normalization
    theo_ret_time=1/(w[nodeToIdx[start_node]]*pi_bar[nodeToIdx[start_node]])

    return theo_ret_time
```

By applying the aforementioned function, the outcome is a **theoretical return time** equal to **6.750 time units**, pretty close to the experimental one computed in the first task.

## 1.c

The following two tasks will focus on the *hitting time*, i.e. the time spent by a particle to travel from node $o$ to node $d$. A new algebraic object comes in our disposal: the **stochastic matrix P**, a matrix that has the conditional probabilities of jumping from $i$ to $j$, but with respect to *local* Poisson clock rate, i.e. the one which refers to node $i$:

$$P_{ij} = \frac{\Lambda_{ij}}{\omega_i}$$

In particular, the **average hitting time** for this **o-d journey** is requested: to compute it we use the cumulative approach of the first task. Therefore, we'll create a list of time entries exploiting the code snippet in Listing 1 from node $o$ to node $d$, then we average along this computed vector. As a final result we obtain **7.387**.

Eventually, a further study among the network is run to understand the number of simulations needed to achieve a final **convergence**. In our case, the system converges after 24 iterations:

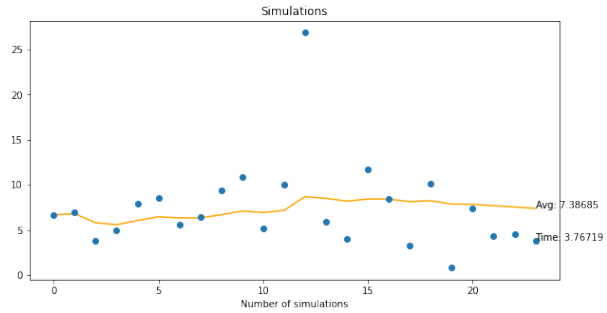



Figure 2: Average time of hitting time and number of simulations to achieve convergence.

## 1.d

Let's define a subset of stubborn nodes $\mathcal{S} \subseteq \mathcal{V}$ in which the particle would be eventually trapped by the system, and let's indicate with $s \in \mathcal{S}$ the s-th node belonging to this set. An *hitting probability* $\Gamma_{i,s}$ is the probability of a random walk $X(t)$ to "hit" (i.e. arrive in) $\mathcal{S}$ in $s$.

Now, let's call $\mathcal{R}$ the complementary set of nodes with respect to $\mathcal{S}$, so:

$$\mathcal{R} = \mathcal{V} \setminus \mathcal{S}$$

Its functionality is quite trivial: it represents the network space in which the walk occurrs without being stopped at all and it could be used for our convenience to reduce our stochastic $P$ dimensionality.

To finally compute the **theoretical hitting time** we use the recursive formula for the return times extended to the (o,d) pair. Mathematically speaking we'll have:

$$E_o[T_d] = \frac{1}{w^*} + \sum_j P_{o,j} E_j[T_d]$$

$$E_d[T_d] = 0$$

```python
def hitting_time(start_node,end_node):
    start_idx=nodeToIdx[start_node]
    R = [list(G.nodes()).index(node) for node in list(G.nodes()) if node != end_node]
    # calculating the expected hitting times for the other nodes
    # the recursive formula can be solved through a linear matrix equation
    degrees = np.sum(A,axis=1)
    D = np.diag(degrees)
    P = np.linalg.inv(D) @ A
    P_hat = P[np.ix_(R, R)]

    exp_hitting_times_R = np.linalg.solve((np.identity(P_hat.shape[start_idx]) - P_hat
    ), np.ones(P_hat.shape[start_idx])/w_star)

    exp_hitting_times = np.zeros(len(list(G.nodes())))
    exp_hitting_times[R] = exp_hitting_times_R
    exp_od_time = exp_hitting_times[start_idx]

    return exp_od_time
```

Listing 2: Function used to compute the hitting time between the pair o-d of nodes. A further implementation detects whenever the start and the end nodes coincide and apply the *return_time* function instead

The idea behind this application, and in particular $\hat{P}$, is to shrinking the domain in which we work: setting $\mathcal{S}$ having only $d$ as stubborn node, the resulting $\hat{P}$ will have $dim_P - dim_S = 4$.

Let's for a moment name $\tau_o^d$ the *expected theoretical hitting time*. From the previous equation we obtain:

$$\tau_o^d = \frac{1}{\omega^*} + P_{o,j}\tau_j^d$$

But we know for sure that if $j \in \mathcal{S}$ the hitting time would be 0, since it results in being trapped by the stubborn set. Therefore, our pre-computed $\hat{P}$ helps us to smooth our equation:

$$\tau_o^d = \frac{1}{\omega^*} + \hat{P}_{o,j}\tau_j^d, \ \mathbf{j} \in \mathcal{R}$$

$$\downarrow$$

$$\tau_o^d = (I - \hat{P})^{-1}\frac{1}{\omega^*}$$

The computations are then solved through Listing 3, obtaining a final value of **7.357**.

## 1.e

The most general way to study the *French-De Groot dynamics* is the following: if our given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \Lambda)$ owns a gloablly reachable and aperiodic component $\mathcal{W}$, then the **consensus** is reached. This means that only the opinions in the sink are the valid ones, i.e. the opinions of agent not belonging to $\mathcal{W}$ have no influence on the final state of the consensus:

$$\lim_{t \to +\infty} x_i(t) = \alpha\mathbf{1} = \pi' x(0), \forall i$$

In our the case, graph $\mathcal{G}$ is both **strongly connected** and **aperiodic**, so that the networks fully complies with the given conditions.

Eventually, we can divide the task into two parts, but first let's highlight what are the main ingredients here:

- an arbitrary, yet simple **initial condition** $x(0) = (1, 0, 0, 0, 1)$.
- a **stochastic matrix P** computed as:

$$P = D^{-1}\Lambda = diag(\omega_i)^{-1}\Lambda$$

### Averaging Dynamics simulation

The basic idea behind the simulation process of the dynamics is to update each node opinion until we reach the final iteration or at least an equally distributed opinion among each node. The updating process will then comply with the following equation:

$$x(t) = P^t x(0)$$

In our case, our system run for 200 simulations, ending up with a shared **consensus** of **0.3043** after $\simeq$ 80 iterations, as Figure 3. More details are displayed in the following code, which exploits the previous definition of each state $x(t)$ and uses it to compute the final opinion:

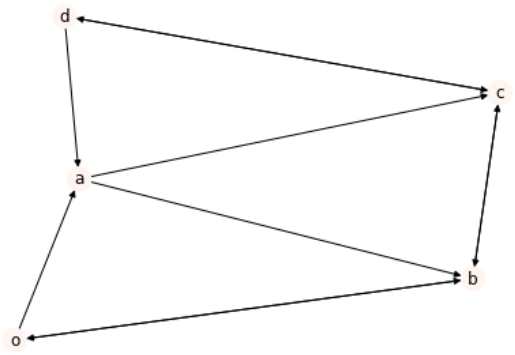



Figure 3: Resulting graph of the average opinion that reaches consensus. The opinion-value is determined by the colour of the nodes, clearly light and equal for all of them.

```
def deGroot_dynamics(init_state,G,W,n_iter=200,printEvery=10):

  # Construct P
  degrees = np.sum(W,axis=1)
  D = np.diag(degrees)
  P = np.linalg.inv(D) @ W

  # keep track of the trajectory
  x = np.zeros((n_iter,len(G.nodes())))

  # set initial condition
  x[0]=init_state

  print("Starting opinion:",x[0])
  nx.draw(G,pos,node_color=np.round(x[0], decimals=2)*100,with_labels=True, cmap=plt.
    cm.Reds)
  plt.show()

  # evolve the states
  for t in range(1,n_iter):
      x[t] = P @ x[t-1]
      if t%(printEvery)==0:
        print("Opinion after",t,"iteration:",x[t])
        nx.draw(G,pos,node_color=np.round(x[t], decimals=2)*100,with_labels=True, cmap
    =plt.cm.Reds)
        plt.show()

  print("Ending opinion:",x[-1])
  nx.draw(G,pos,node_color=np.round(x[-1], decimals=2)*100,with_labels=True, cmap=plt.
    cm.Reds)
  plt.show()
  return x
```

### Reaching Theoretical Consensus

After some algebraic manipulations, however, we can ensure an important equivalence for what concerns the consensus. Let's take a closer look:

$$\lim_{t \to +\infty} x_i(t) = \bar{x}, \forall i$$

$$\downarrow$$

$$\begin{cases} x(t) = P^t x(0) \\ \pi' P^t = \pi' \\ \pi' \mathbf{1} \bar{x} = \bar{x} \end{cases}$$

$$\pi' x(0) = \pi' P^t x(0) \to \bar{x} = \pi' x(0)$$

Where $\pi$ is the *invariant distribution*, computed on the eigendecomposition of $P^t$ matrix. Now we can use this equivalence to compute the **theoretical consensus**, thanks also to the following snippet of code, with a final result of **0.3043** as we expected.

```
def consensus_value(G,W,states):
  #Note: states contains the opinions vectors of each iteration of degroot_dynamics
  # Construct P to compute eigenvalue and eigenvector
  degrees = np.sum(W,axis=1)
  D = np.diag(degrees)
  P = np.linalg.inv(D) @ W
  w,v = np.linalg.eig(P.T)

  # selects the eigenvalue 1 and save the eigenvector
  for index in [i for i in range(len(G)) if np.isclose(w[i],1)]:
      pi = v[:,index].real  # convert eigenvector to real
      pi = pi/np.sum(pi)

  print("Theoretical convergence opinion:",pi.T@states[0])
```

## 1.f

Let's now assume that each entry of our initial state array is given by:

$$x_i(0) = \zeta_i$$

where $\zeta_i$ is an i.i.d. random variable, in particulare we operate with a *uniform distribution* with variance 1/12. The variance of the consensus value can be computed as follow:

$$\sigma^2_{consensus} = \sigma^2 \sum_i \pi_i^2 < \sigma^2$$

One can expect that the variance of the consensus value is less than the variance of the node state, i.e *crowd is wiser than the single*.

With these premises, we can run our French-DeGroot dynamics to evaluate the consensus, and after that achieve 4 different results:

- Numerical variance of node states, computed as the average between the variances of each node state at each simulation (**0.0638**);
- Theoretical variance of node states, the real variance value of the random variable given to each state for each node (1/12=**0.0833**);
- Numerical varince of consensus value, the estimate for the variance of the consensus during the numerical simulation (**0.0185**);
- Theoretical variance of consensus value, the one referred to the initial variance formula, i.e. $\sigma^2_{consensus} = \sigma^2 \sum_i \pi_i^2$ (**0.0178**)

As we expected, the *wisdom of crowd* is then satisfied. The two functions which allows us to reach our goal are listed in the following code:

```
    def theo_var_cons_value(G,W,var):
      # Construct P
      degrees = np.sum(W,axis=1)
      D = np.diag(degrees)
      P = np.linalg.inv(D) @ W
      w,v = np.linalg.eig(P.T)

      # selects the eigenvalue 1 and print the eigenvector
      for index in [i for i in range(len(G)) if np.isclose(w[i],1)]:
          pi = v[:,index].real  # -> eigenvectors are complex but pi is real, so we
    convert it to real
          pi = pi/np.sum(pi)

      return np.sum(pi**2)*var

    n_sim=200
    n_states=500
    W = np.array(A) # -> return type is scipy.sparse.csr_matrix

    degrees = np.sum(W,axis=1)
    D = np.diag(degrees)
    P = np.linalg.inv(D) @ W

    # start with random initial states and run the dynamics
    alfa_err = np.zeros(n_sim)
    vars=[]
    for i in range(n_sim):
    # rand returns random values in [0,1], thus \mu = 1/2
        x = np.random.rand(len(G.nodes()))
        vars.append(np.var(x))
        for n in range(n_states):
            x = P @ x
        alfa_err[i] = (1/2 - np.mean(x))*(1/2 - np.mean(x))

    print("Numerical variance of the node states:", np.mean(vars))
    print("Theoretical variance of the node states:", 1/12,"\n")
    print("Numerical variance of the consensus state:", np.mean(alfa_err))
    print("Theoretical variance of the conensus state",theo_var_cons_value(G,W,1/12))
```

## 1.g

Removing the two proposed edges from our graph will result in altering the internal structure of it. In particular we don't have a strongly connected graph anymore, yet it maintains the aperiodicity, as Figure 4. Moreover, to avoid any dimension-related problem in solving matrices operations, a self loop of infinitesimal weight is added to the modified $\Lambda$.

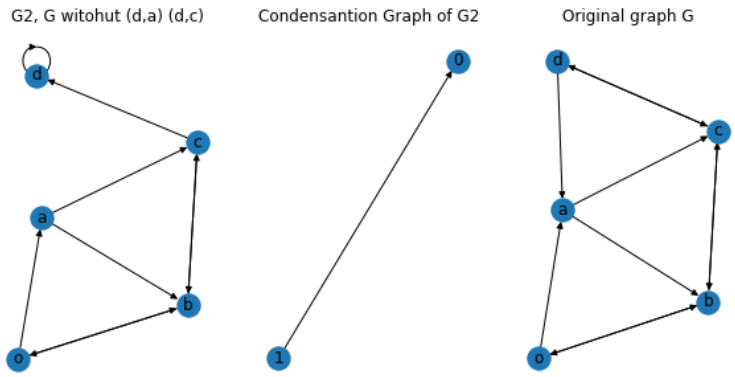



Figure 4: New updated graph

Actually, what's happening here is that we have a very strong component in $d$ that "manipulates" and bias the overall network opinion to be equal to its own (this is recognizable also in social behaviours). Two experiments are conducted, with tho different states: what we expect is to achieve as final consensus the **starting opinion of node d**. Let's see if we can confirm this:

- $x^{(1)}(0) = [1, 1, 1, 1, \mathbf{2}] \rightarrow$ consensus = **2** achieved by $\simeq$40 simulations;
- $x^{(2)}(0) = [1, 5, 2, 3, \mathbf{0}] \rightarrow$ consensus = **0** achieved by $\simeq$40 simulations.

Another point of this task is to try with the new structure a normal-distributed initial state $x_i(0) = \zeta_i$ with respective mean $\mu = 0.5$ and variance $\sigma^2 = 10$. The procedure will be the same, however the behaviour experienced could affects differently the network: note that we are working with a very strong node, so as a result we can't expect anymore a *wisdom of crowd* since the crowd's opinion is compresses entirely in the most powerful node $d$ in terms of influence: instead to achieve a smaller variance with the final consensus, we'll get a comparable (even equal) variance with respect to the own belonging to node $d$. Running again our already compiled function we obtain:

- Numerical variance of node states (**8.275**);
- Theoretical variance of node states,(**10**);
- Numerical varince of consensus value, (**8.942**);
- Theoretical variance of consensus value, (**10**)

### 1.h

With the last point of this exercise we are going to simulate a particular behaviour related to another update of our network. We remove the edges (c,b) and (d,a) and we run our simulations from the arbitrary initial condition.

Three experiments are runned and we obtain a strange behaviour of each node:

- *node d and c* exchange thei opinions between each other;
- *node o and a* will converge to a "local" consensus;
- *node b* will not converge at all (at least it will have a similar value to the consensus achieved by o and a)

## Exercise 2

In this exercise the network considered is the same as before. Instead of studying the walk of a single particle, we are now interested in studying the behaviour of many particles moving around in the network in continuous time.

The system will be simulated from two different perspectives: the *particle perspective* and the *node perspective*.

### 2.a - Particle perspective

Simulating the system from a particle perspective is exactly as in Problem 1, but now the focus is on more than one single particle.

- For this simulation, we will use the approach involving the use of a Poisson clock with rate $w_i$ for every node i in our state space, $\forall i \in X$ .

  The average return time obtained is **6.66 s**.

- Since all the particles are independent and identically distributed, simulating a random walk with 100 particles is equal to simulating a random walk with one particle 100 times. So this simulation is equal to the one performed in Problem 1 expect for the number of simulations. Here the number of simulations is equal to the number of particles (100).

### 2.b - Node perspective

To simulate the system from the node perspective we observe the particles from the node, which means that the focus is on the number of particles in a node.

For this perspective we use a system-wide Poisson clock with rate 100 and at each tick we randomly, and proportionally to the number of particles in different nodes, select a node from which we should move a particle. Then a particle from the selected node will move according to the transition probability matrix.

- It will be shown in the next answer that the average number of particles in the different nodes at the end of the simulation is 20.
- *Illustrate the simulation above with a plot showing the number of particles in each node during the simulation time*:

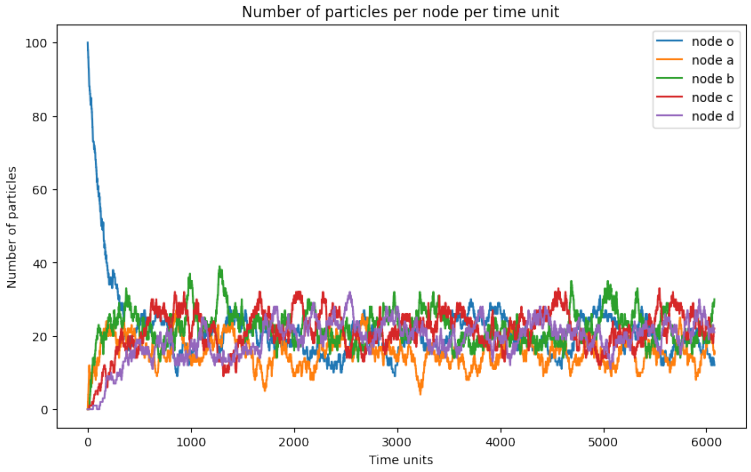



Figure 5: Plot showing the number of particles in each node during the simulation time.

- The estimated result of this simulation should be equal to the invariant probability vector $\pi$ multiply by 100, because the invariant probability vector contains the probabilities that a single particle end up in each node.

  `Particles per node at final step: [16 14 17 28 25] Average number of particles in every node (π * 100): [18.51851852 14.81481481 22.22222222 22.22222222 22.22222222]`

## Exercise 3

In this exercise we consider an open network (6) in which we study how different particles affect each other when moving around in continuous time. The transition rate matrix $\Lambda_{open}$ is given.

For this system, particles will enter the system at node o according to a Poisson process with rate $\lambda = 1$. Each node will then pass along a particle according to a given rate, similar to what you did in Problem 2 with the "node perspective". Since node d does not have a node to send its particles to, when the Poisson clock ticks for this node we could equivalently think of another node d' connected to node d, such that at every tick of the Poisson clock of d, it sends a particle to node d'.

The goal is to simulate two different scenarios that differ by what rate the nodes will pass along particles: **proportional rate** and **fixed rate**.

### 3.a - Proportional rate

The **rate of the Poisson clock of each node is equal to the number of particles in it.**

-

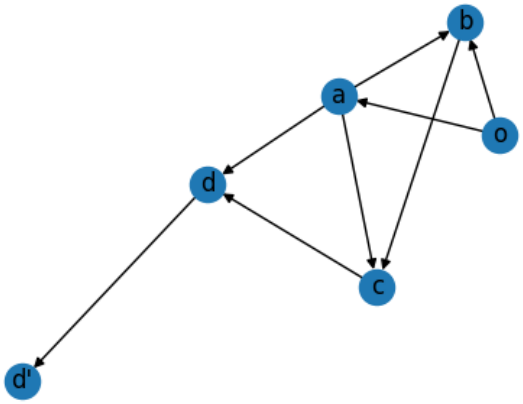



Figure 6: Open network we are considering.

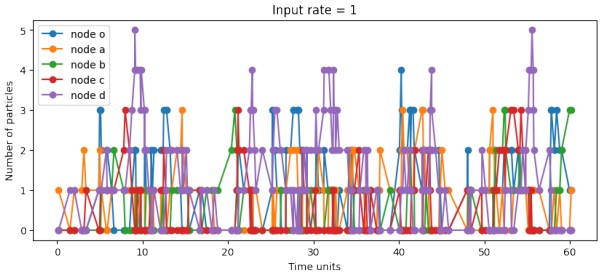



Figure 7: Plot showing the evolution of the number of particles in each node over time.

- In this scenario we have two clock: one for the entrance of the particles with rate l=1 and one global clock with rate proportional to the number of particles in the system. We simulate the system till we reach 60 time units. At every step we compare the cumulative transition time of the entrance clock and the one of the global clock: we let ticks the clock with the lower transition time. For the simulation we set 2 different input rates: 1 and 10. We can see the system does not blow up, in particular, particles do not accumulate in node 'o' and they move around the graph. This is due to the proportional rate of the global clock, the more particles are in the graph, the more the rate is higher. So each node is able to pass along its particles to other nodes.

## 3.b. - Fixed rate

The **rate of the Poisson clock of each node is fixed, and equal to one.**

-

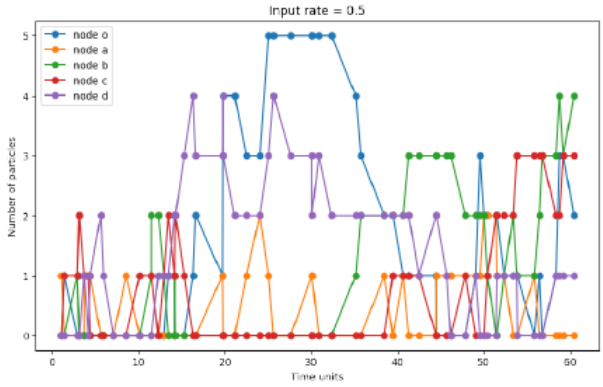



Figure 8: Plot showing the evolution of the number of particles in each node over time.

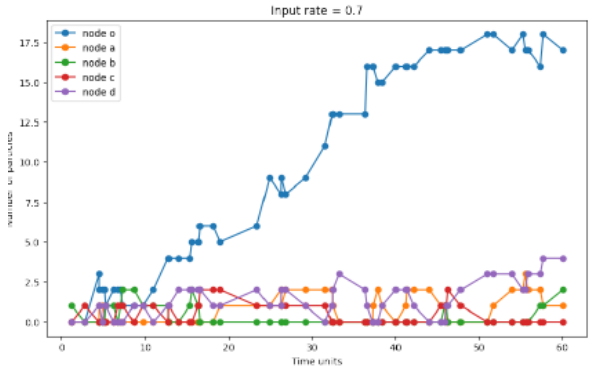



Figure 9: Plot showing the evolution of the number of particles in each node over time.

- The difference between this simulation and the past one is the rate of the global clock: in this case it is fixed and equals to 1 (wstar: the value of the maximum value from the vector w) We simulate the rate case with 4 different input rates: 0.5, 0.7, 1, 2. In this scenario we can clearly see that particles accumulate in node 'o' causing the blow-up. From the plot we can identify the crucial input rate: 0.7.