Homework 2

Network Dynamics, and Learning

01TXLSM

Fatemeh Ahmadvand

s301384

2022-2023

Continuous Time Markov Chains: In CTMC, the time is not discrete ($t = 0, 1, \ldots$) but it flows in a continuum ($t \geq 0$). The random process still describes the evolution of a state variable x inside a discrete state space $\mathscr{X}$ with a graph structure. We are given a graph $G = (\mathscr{X}, \Lambda)$ with nodes $\mathscr{X}$ and weight matrix Λ describing possible transitions between nodes/states.

Transitions now happen at random time instants that are decided by the tick of a so called **Poisson clock**. A Poisson clock is characterized by the property that the time elapsed between any two of its consecutive ticks is an independent random variable with exponential distribution with a specified rate. to simulate continuous time Markov chains the following fact will be useful.

Note 1: To simulate a Poisson clock with rate r, one must simulate the time between two consecutive ticks, which we denote by t_{next}. We can compute t_{next} as

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

where u is a random variable with uniform distribution, $u \in \mathscr{U}(0, 1)$.

The key property of the exponential distribution is that it is memoryless:

$$\mathbb{P}(X \geq t + s \mid X \geq t) = \frac{\mathbb{P}(X \geq t + s)}{\mathbb{P}(X \geq t)} = \frac{e^{-r(t+s)}}{e^{-rt}} = e^{-rs} = \mathbb{P}(X \geq s).$$

Modeling Continuous time Markov chains

There are two equivalent ways of modeling CTMCs.

1st approach:

1. you define a unique **global** Poisson clock with an appropriate rate $\omega^* = \max_i(\omega_i)$ where $\omega_i = \sum_j \Lambda_{ij}$

2. when you are at node i and **the global clock ticks**, either you jump to a neighbor j with probability $Q_{ij} = \frac{\Lambda_{ij}}{\omega_*}$, $i \neq j$ or you stay in the same node (no transition) with probability $Q_{ii} = 1 - \sum_{i \neq j} Q_{ij}$.

In this approach, the continuous time is "discretized" using a global clock, while the matrix Q describes the jumps. For this reason the matrix Q is called **jump chain** of the CTMC.

Notice that $Q_{ii} = 0$ for the nodes i maximizing ω, and it is larger as ω_i/ω is small.

2nd approach:

- each node i is equipped with its own Poisson clock with rate $\omega_i = \sum_j \Lambda_{ij}$.
- when you are at node i and **the clock of that node ticks**, you jump to a neighbor j with probability $P_{ij} = \frac{\Lambda_{ij}}{\omega_i}$.

For nodes i such that $\omega_i = 0$ (which means that once the process in i it remains i forever), we need to add a selfloop $\Lambda_{ii} > 0$, otherwise the matrix P is not well defined.

The probability distribution $\bar{\pi}(t)$ of the CTMC $X(t)$ with transition rate matrix Λ is defined as:

$$\bar{\pi}_i(t) = \mathbb{P}(X(t) = i), \quad i \in \mathscr{X}\,.$$

It evolves according to the equation

$$\frac{d}{dt}\bar{\pi}(t) = -L'\bar{\pi}(t)$$

where $L = diag(w) - \Lambda$, with $w = \Lambda\mathbf{1}$.

Thus, the invariant probability vectors are eigenvector of L' corresponding to eigenvalue 0. It can also be proven that $\bar{\pi}$ is the left dominant eigenvector of Q, where Q is the row-stochastic matrix defined as

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

with $\omega = \Lambda\mathbf{1}$ and $\omega_* = \max_i \omega_i$.

Problem1: The first part of this assignment consists in studying a single particle performing a continuous-time random walk in the network described by the graph in Fig. 1 and with the following transition rate matrix:

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

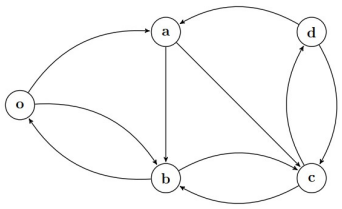


Figure 1: Closed network in which particles move according to the transition rate matrix (1).

Your task is to simulate the particle moving around in the network in continuous time according to the transition rate matrix (1).

Solution. In this exercise, we want to simulate a continuous random walk. In the given network in the form of a graph, our task is to simulate these movements in this network based on the given transition rate matrix. For this, we display the graph itself according to the inputs and Λ is our transition matrix. We apply the second approach mentioned above.

Now for our simulation, we need to define metrics and vectors. According to our transition matrix, we want to calculate the transition rate matrix as follows. That is how it is obtained for each node ω_i:

$$\forall i, j \in \mathcal{X}, \omega_i = \sum_j \Lambda_{ij} \quad (1)$$

In the next step, the conditional probabilities of moving between nodes i and j called Q and staying at that node are $1 - Q$. We have a single-position clock that we need to use in a node with the same rate as ω_i, which is our transmission rate.

Matrix P is defined as a conditional probability matrix that contains movement between nodes i and j, provided that this node when our position clock is ticked at node i. In fact, P_{ij} is calculated as follows:

$$P_{ij} = \frac{\Lambda_{ij}}{\omega_i} \quad (2)$$

we Compute the invariant probability vector $\bar{\pi}$ of the CTMC by determining the leading eigenvector of the matrix Q'. This vector is to go from element i to node i. In general, our goal in this simulation is to move from the position clock with rate ω_i for each node i in this network.

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

  In this part, We define random walk simulation function which is finally supposed to give us an average time for the random walk performed for each node it passes through. The simulation starts from node a, walks into the network then returns to node a. Return time for node a is done using the **simulationAvgTime** function, by setting origin=1, and destination=1. In this function, we set the number of steps to 10000. Then we define the array_return_time keeps the trace of the visited states. After that, we iterate over simulations that start from state a. By transition_times variable, we store the time instant of the current transition. According to the t_{next} formula, we calculate the Poison clock of node a.

  Average return time: 6.645609572207408

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

  The expected hitting times $\hat{x} = (\mathbb{E}_i[T_S])_{i \in R}$ for the setthe S and for all nodes $i \in R = \mathcal{V} \ S$ can be computed by solving the system of equations

  $$\hat{x} = \mathbf{1} + \hat{P}\hat{x},$$

  where $\hat{P}$ is obtained from P (the normalized weight matrix of the graph) by removing the rows and columns corresponding to the nodes in the set S. More explicitly, the expected hitting times can be expressed as

  $$\hat{x} = (I - \hat{P})^{-1}\mathbf{1}$$

  Remark: note that $(I - \hat{P})$ is invertible only if $V \ S$ has at least a link pointing to S. Indeed, if $(I - \hat{P})$ is not invertible. the random walk starting from nodes in $V \ S$ cannot hit nodes in S, and the hitting times diverge.

  Thus, in this section, according to the metrics we defined in part a, we calculate the theoretical return-time from node a to node a by the following formula

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

  Eventually, error simulation shows the difference between the theoretical return-time (6.74) and the average return time(6.64) (part a).

  Expected return time: 6.749999999999998

  Error simulation 0.10439042779259022

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

  We will get an average time going from node o to node d. We used the same method as above, the only difference is that the nodes are different. And here we have a hitting time $E_o[T_d]$ and an error simulation compared to the time that was supposed to be taken. (origin=1 (o) and destination=4 (d))

  Average return time: 8.1396675878699 s

- *d) How does the result in c) compare to the theoretical hitting-time $E_o[T_d]$? (Describe also how this is computed.)*

  Here we want to calculate the hitting time and get the error simulation. We get the hitting time with the formula $\frac{1}{\omega_o \bar{\pi}_d}$. All calculation steps are the same as question part 'b', but the only difference is in the beginning and end nodes o to d. In this step error simulation is 0.64.

  Hitting time: 8.785714285714285 time units

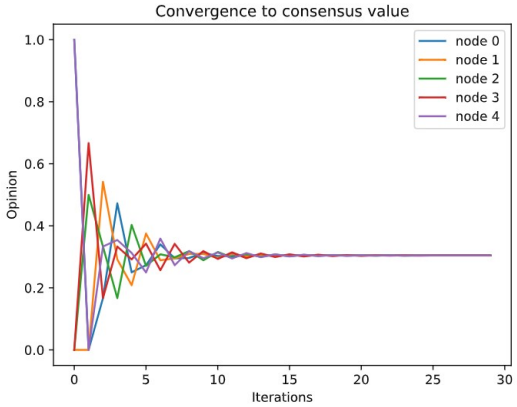
  Error simulation 0.6460466978443851

- *e) Interpret the matrix Λ as the weight matrix of a graph $G = (v, E, \Lambda)$, and simulate the French-DeGroot dynamics on G with an arbitrary initial condition x(0). Do the dynamics converge to a consensus state for every initial condition x(0)? Motivate your answer.*

  This is the most important result of the linear averaging model. It asserts that consensus is the asymptotic outcome of the model under certain connectivity assumptions. The consensus value is the weighted average of the agent$i$'s initial opinion $x_i(0)$ coincides with its invariant distribution π_i. This remarkable fact gives the invariant distribution centrality π of the graph G significance also in a dynamic context. Considering that the given graph is strongly connected and the condensation, Therefore, it has only one sink and the sink is repeated aperiodically and accumulates at the point $x(t)$. The consensus state represents the only sink of the graph.

  The consensus state is: [0.304 0.304 0.304 0.304 0.304]

  The consensus value is: 0.30434782608695654



- *f) Assume that the initial state of the dynamics for each node $i \in \nu$ is given by $x_i(0) = \xi_i$, where $\xi_{ii\in\nu}$ are i.i.d random variables with variance σ^2. Compute the variance of the consensus value, and compare your results with numerical simulations.*

  According to the calculation of the consensus state in part (F), in this part, we calculate the consensus state variance, and then we also calculate the expected variance value.

  The the variance of the consensus state is: 0.021

  Expected variance: 0.017032464563328763

  The difference between these two variance values indicates the same amount of error mentioned in the previous question.

- *g) Remove the edges (d, a) and (d, c). Describe and motivate the asymptotic behavior of the dynamics. If the dynamics converges to a consensus state, how is the consensus value related to the initial condition x(0)? Assume that the initial state of the dynamics for each node $i \in \nu$ is given by $x_i(0) = \xi_i$, where $\xi_{ii\in V}$* are i.i.d random variables with variance σ^2. Compute the variance of the consensus value. Motivate your answer.

  We want to count the number of edges from node d to a and from node d to c. To calculate it, we changed the construction of our graph, we removed the edges d to a and d to c. So we have to calculate P again. For calculating the asymptotic state x, the state that we are in, as we expect, it gives us a constant number for output.

  arbitrary initial condition x(0): [0.764 0.723 0.04 0.124 0.794]

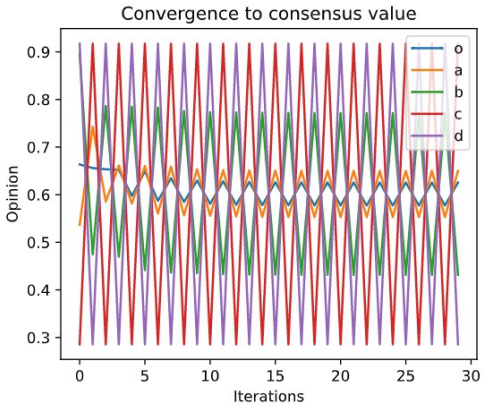
  The asymptotic state x is: [0.794 0.794 0.794 0.794 0.794]

  And finally, we calculate the variance of the consensus state according to the initial condition we have.

  The variance of the consensus state is: 0.078

- *h) Consider the graph (ν, E, Λ), and remove the edges (c, b) and (d, a). Analyze the French-DeGroot dynamics on the new graph. In particular, describe and motivate the asymptotic behavior of the dynamics in terms of the initial condition x(0).*

  We remove different edges to calculate the arbitrary initial condition x(0). This time, the c to b and d to a edges are to be removed. By removing these edges, the initial condition x(0) changes. To calculate it, we reconstruct our graph according to the edges we removed, so we have to calculate P again, and finally, we calculate the final option.



Problem2: In this part, our goal is the simulation both aspects of the question.

a) Particle perspective:

- *If 100 particles all start in node a, what is the average time for a particle to return to node a?*
- *How does this compare to the answer in Problem 1, why?*

  In the first part of the particle's perspective, we want to calculate how much Average Time we will have if we have 100 particles that move from node a and reach node a again at the end. To simulate this issue, we use Poisson Clock with an ω_i for each node i in this discrete space. It should be noted that because "all particles are independent and have the same distribution", our simulation is therefore a random walk in which one particle is repeated 100 times. Instead of simulating 1000 times as in the first problem of part A, To calculate the average time, we can do the same simulation 1000 times, but for each node separately, in other words, we have 100 particles that we have to do for each one separately. In this section, we still use the simulated random walk function, as an example, we move from node 1 to node 1, with the difference that we multiply the number of particles by 1000.

  Average return time: 6.764747790576242 s

  Error simulation 0.014747790576243425

b) Node perspective:

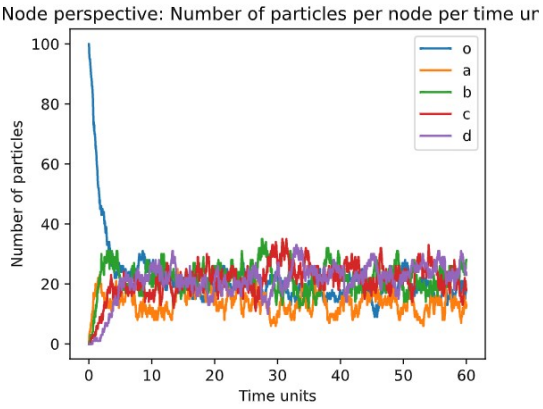
- *If 100 particles start in node o, and the system is simulated for 60-time units, what is the average number of particles in the different nodes at the end of the simulation?*
- *Illustrate the simulation above with a plot showing the number of particles in each node during the simulation time.*
- *Compare the simulation result in the first point above with the stationary distribution of the continuous-time random walk followed by the single particles*

  In this section, since the number of particles is 100 and the time unit is assumed to be 60, we calculate the transaction, this calculation is no longer based on a random walk, because our perspective in this section is the nodes that start from which index and in which index to the end. The important thing is that the π values obtained in this section are very close to the values obtained in the first question, with the difference that in the first section we have simulated 1000 particles, but in this section, we have simulated 100 particles.

  The simulation of random walk for 100 particles is exactly like the simulation of random walk, 100 times for one particle, but to calculate the average, instead of doing the simulation for each particle a thousand times and then taking the average (we did this in the second part of the first problem). We can only run the simulation 100 times. The average obtained for 100 particles is T=6.76, which shows how close they are to the theoretical answer we get from the second part of the first problem, which is equal to T=6.75.

  Particles per node at final step: o:21.62, a:14.04, b:21.87, c:21.22, d:21.24

  Average number of particles in every node $\bar{\pi}$:[18.519 14.815 22.222 22.222 22.222]



Problem3: In this part, we study how different particles affect each other when moving around in a network in continuous time. We consider the open network of Figure 2, with transition rate matrix Λ_{open} according to (2).

$$\Lambda_{\text{open}} = \begin{array}{c} \begin{matrix} o & a & b & c & d \end{matrix} \\ \begin{pmatrix} 0 & 3/4 & 3/8 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 2/4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{matrix} o \\ a \\ b \\ c \\ d \end{matrix} \quad (2)$$

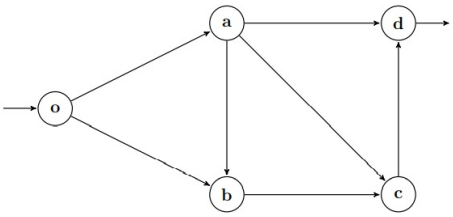


Figure 2: Open network

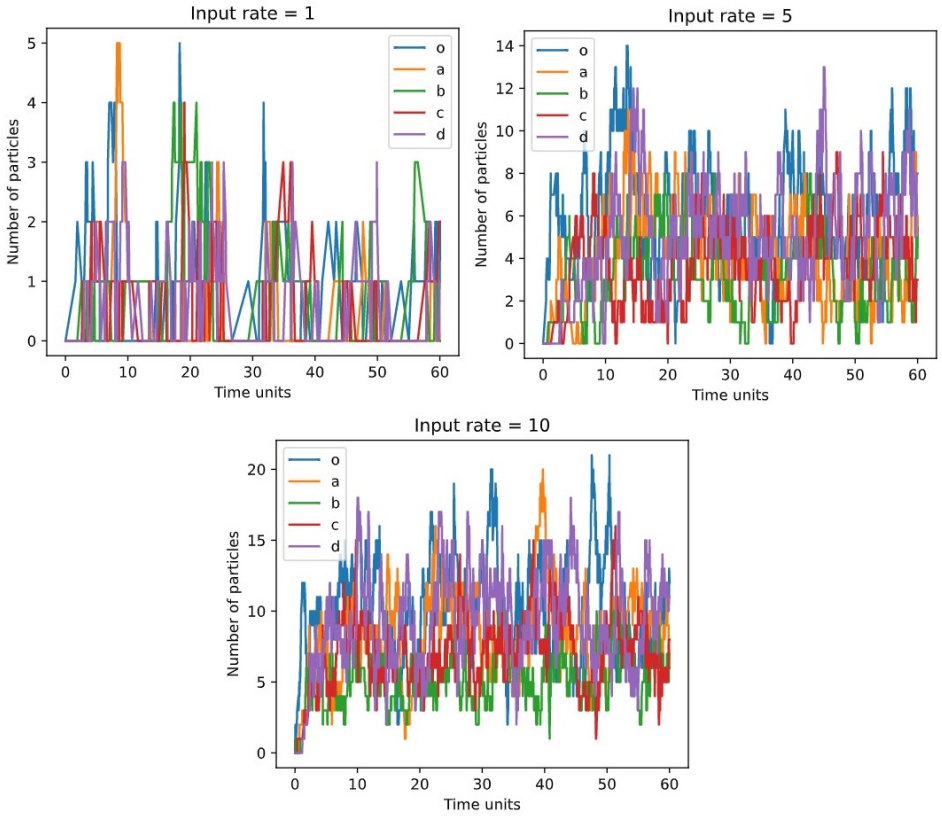
a) Proportional rate:

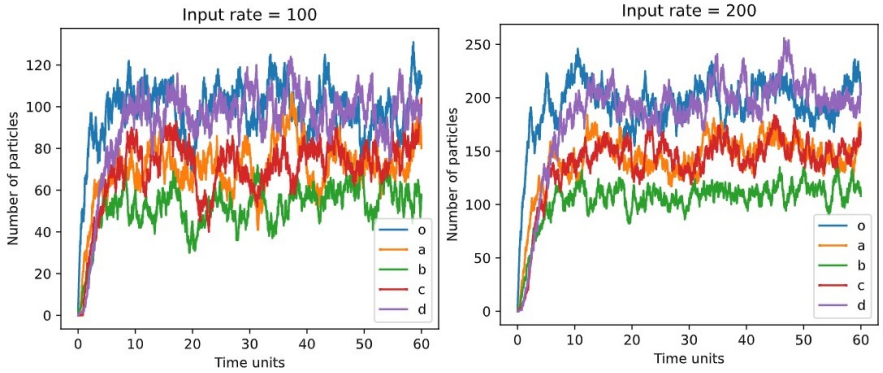
- *Simulate the system for 60-time units and plot the evolution of the number of particles in each node over time.*
- *What is the largest input rate that the system can handle without blowing up?*

  In this problem, we want to consider a system in which a series of particles are entered from node 'o' by processing with = 1, and these particles are transferred to any other node. This problem is similar to problem 2 in the node perspective section. We must note that node D does not have any node to transfer the particles to, and every time this position clock is set, and node D is supposed to transfer these particles, we can assume that to a hypothetical node D' It is transferring these particles. In the first part of the work, we must redefine all the metrics and vectors that we defined for problem 1. As you can see, we first drew the graph and calculated the matrix of .

In the first part of this problem, we have a function to get the starting node, here we want to work with a proportional rate, in which we specify that the position clock rate should be equal to the number of particles we have in each node. When this happens, the number of particles in each node naturally increases and the position clock rate also increases. We perform this simulation for time-unit = 60 and finally, we are going to display the progress of the number of particles in each node and the maximum amount that we can give to the rate input to the system on the graph.

For this purpose, we defined a get starting node, to start the work and simulation with the function simulate_proportional_rate, and finally, with the function plot_proportional_trajectories, we display the performed calculations. In this problem, we have considered the input rate as 1, 5, 10, 100, and 200, and we draw the desired plot for each input. Based on that, the number of particles we have increases our fluctuation rate. Fortunately, up to the number of input rate = 200, the system did not encounter any bugs and was passable, and more numbers than this require more calculation time. It should be kept in mind that as the number of particles increases, the rate also increases.





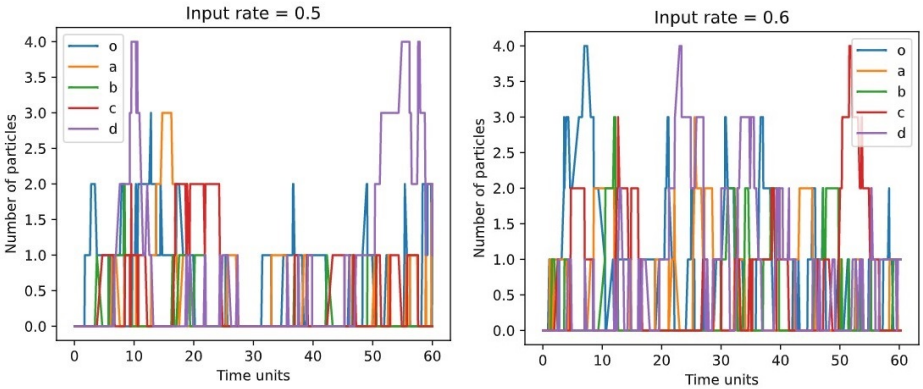
b) Fixed rate:

- *Simulate the system for 60-time units and plot the evolution of number of particles in each node over time.*
- *What is the largest input rate that the system can handle without blowing up? Why is this different from the other case?*

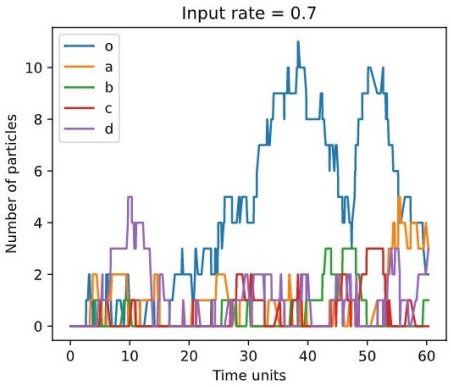
  We want to use the fixed rate in the second part of this problem. In this case, the position clock rate for each node is constant and equal to 1. In this problem, as in the previous part, we perform the simulation with time-unit = 60 and show how the number of particles in each node increases and the maximum amount of input rate that we can apply. Like the previous part, we have a get starting node, and here we have a function called simulate_fixed_rate, which we want to perform the simulation based on a fixed rate for the position clock, and finally, we display it.

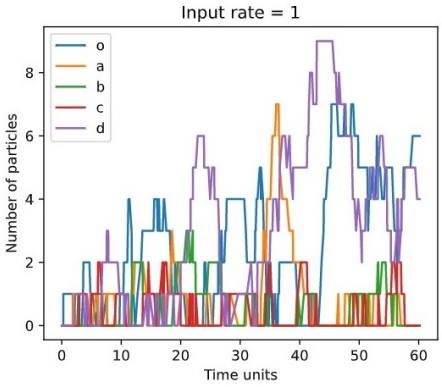
  In this problem, we have considered the input rate values of 0.5, 0.6, 0.7, 1, and 2, and the point is that at rates of 1 and above, this system encounters a bug and does not work, so to speak, it becomes unstable. The highest in our opinion, rate that this system can run is 1, but it is also unstable in this case. The best rates that can be considered for this purpose are 0.6 and 0.7, which we have shown in the graph.

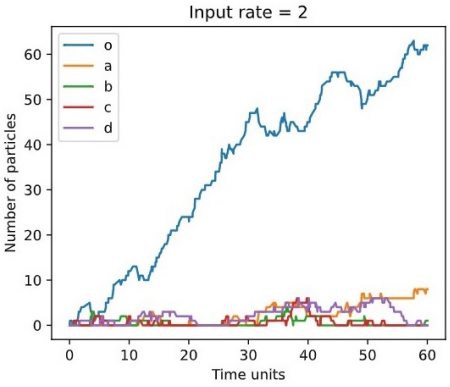
  At the input rate = 0.5, which is the lowest value that we have considered, the particles in the nodes did not have any problems. Also, we have not had any problem with the input rate = 0.6. At input rate = 0.7, the system is almost without problems, but at input rate = 1, the particles have encountered problems and we can see an unstable state in the diagram, and at input_rate = 2, the state of instability for node 'o' is quite clear. What is clear is that in fixed rates, no matter how much the number.


Input rate = 0.5
Number of particles
Time units
o
a
b
c
d

Input rate = 0.6
Number of particles
Time units
o
a
b
c
d


Input rate = 0.7
Number of particles
Time units
o
a
b
c
d


Input rate = 1
Number of particles
Time units
o
a
b
c
d

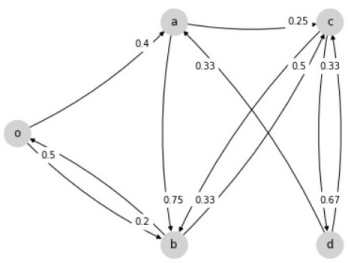

Input rate = 2
Number of particles
Time units
o
a
b
c
d

In [1]:
```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

Exercise 1

In [2]:
```
np.random.seed(seed=42)
np.set_printoptions(precision=3, suppress=True)
# lambda : Transition Matrix
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],
]

G = nx.DiGraph()
G.add_nodes_from(['o', 'a', 'b', 'c', 'd'])
G.add_weighted_edges_from([('o', 'a', float("{0:.2f}".format(2 / 5))), ('o', 'b', float("{0:.2f}".format(1 / 5))),
                           ('a', 'b', float("{0:.2f}".format(3 / 4))), ('a', 'c', float("{0:.2f}".format(1 / 4))),
                           ('b', 'o', float("{0:.2f}".format(1 / 2))), ('b', 'c', float("{0:.2f}".format(1 / 2))),
                           ('c', 'b', float("{0:.2f}".format(1 / 3))), ('c', 'd', float("{0:.2f}".format(2 / 3))),
                           ('d', 'a', float("{0:.2f}".format(1 / 3))), ('d', 'c', float("{0:.2f}".format(1 / 3)))])
pos = {'o': (0, 0), 'a': (1, 2), 'b': (1, -2), 'c': (2, 2), 'd': (2, -2)}
nx.draw(G, pos=pos, with_labels=True, node_size=800, font_size=12, node_color='lightgray',
        connectionstyle='arc3, rad = 0.1')
labels = {e: G.edges[e]['weight'] for e in G.edges}
val = nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, label_pos=0.20, rotate=False,
                                   horizontalalignment='center')

n_nodes = len(G.nodes)

w = np.sum(Lambda, axis=1)
w_star = np.max(w)

# contruct the P matrix (instead of Q) and clock rates w
D = np.diag(w)
P = np.linalg.inv(D) @ Lambda
P_cum = np.cumsum(P, axis=1)
# compute the off-diagonal part of Q
Q = Lambda / w_star
# add the diagonal part
Q = Q + np.diag(np.ones(len(w)) - np.sum(Q, axis=1))

Q_cum = np.cumsum(Q, axis=1)

print("Number of nodes: ", n_nodes)
print("\Vector w:\n", w)
print("\nMatrix D:\n", D)
print("\nMatrix P:\n", P)
# print("\nMatrix P_cum:\n", P_cum)
print("\nMatrix Q:\n", Q)
# print("\nMatrix Q_cum:\n", Q_cum)

# compute dominant eigenvector
# Compute invariant distribution pi_bar
values, vectors = np.linalg.eig(Q.T)
index = np.argmax(values.real)
pi_bar = vectors[:, index].real
pi_bar = pi_bar / np.sum(pi_bar)
print("pi_bar=", pi_bar)
```

```
Matrix D:
 [[0.6  0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.  ]
 [0.   0.   1.   0.   0.  ]
 [0.   0.   0.   1.   0.  ]
 [0.   0.   0.   0.   0.667]]

Matrix P:
 [[0.    0.667 0.333 0.    0.   ]
 [0.    0.    0.75  0.25  0.   ]
 [0.5   0.    0.    0.5   0.   ]
 [0.    0.    0.333 0.    0.667]
 [0.    0.5   0.    0.5   0.   ]]
Matrix Q:
 [[0.4   0.4   0.2   0.    0.   ]
 [0.    0.    0.75  0.25  0.   ]
 [0.5   0.    0.    0.5   0.   ]
 [0.    0.    0.333 0.    0.667]
 [0.    0.333 0.    0.333 0.333]]
pi_bar= [0.185 0.148 0.222 0.222 0.222]
```



In [3]:
```
# set the number of steps in the n_simulations=10000
def simulateRandomWalk(origin, destination, n_simulations=10000):
    # problem 1 _ a:
    # t_next = -np.log(np.random.rand())/w[1] #Poison clock of node 'a'

    # array_return_time will keep trace of the visited states
    array_return_time = np.zeros(n_simulations, dtype=float)

    for j in range(n_simulations):  # iterate over simulations
        pos = []
        # we start from state 'a'
        pos.append(origin)  # node 1 -> 'a'
        transition_times = []
        # store the time instant of the current transition
        transition_times.append(0)
        node_counter = 0
        # the random time to wait for the next transition
        # is drawn according to its distribution, discussed in Note1
        # NOTE: in the formula for t_next we use the rate of the clock of
        # the current state, in this case w_star.
        t_next = -np.log(np.random.rand()) / w_star

        while True:
            node_counter += 1
            # append index
            # the next state to visit will be extracted according to the probabilities
            # stored in the row of Q_cum corresponding to the current state.
            pos.append(np.argwhere(Q_cum[pos[node_counter - 1]] > np.random.rand())[0][0])
            # store the time instant of the current transition
            transition_times.append(transition_times[node_counter - 1] + t_next)

            # the particle is in node 'a' after exiting from it
            if pos[node_counter] == destination:
                break
            # compute the waiting time to the next transition
            # NOTE: we use the rate w_star of the clock of the current position
            t_next = -np.log(np.random.rand()) / w_star

        array_return_time[j] = transition_times[-1]

    avg_return_time = np.mean(array_return_time)
    print("Average return time: {} s".format(avg_return_time))
    return avg_return_time

print("******\nProblem 1 - A : \n")
simulationAvgTime = simulateRandomWalk(origin=1, destination=1, n_simulations=10000)
print("\n*****")
```

```
******
Problem 1 - A :

Average return time: 6.645609572207408 s

*****
```

In [4]:
```
# Problem1 b
# define the set S and the remaining nodes R
node = 1
S = [node]

# define the remaining nodes R
R = [node for node in range(n_nodes) if node not in S]

# restrict P to R x R and obtain hat(P)
hatP = P[np.ix_(R, R)]
hatw = w[np.ix_(R)]

# hat(x) is the solution of the linear system
# np.linalg.solve solves a linear matrix equation given
# the coefficient matrix and the dependent variable values
hatx = np.linalg.solve((np.identity(n_nodes - len(S)) - hatP), (np.ones(n_nodes - len(S)) / hatw))

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

print('Expected hitting times on a: ', hitting_s)
expected_theoric = 1 / w[1] + np.dot(P[1, :], hitting_s)

print("******\nProblem 1 - B : \n")
print('Expected return time: ', expected_theoric)
print("Error simulation", abs(simulationAvgTime - expected_theoric))
print("\n*****")
```

```
Expected hitting times on a:  [3.571 0.    5.714 5.857 4.429]
******
Problem 1 - B :

Expected return time:  6.749999999999998
Error simulation 0.10439042779259022
*****
```

In [5]:
```
print("******\nProblem 1 - C : \n")
simulationAvgTime = simulateRandomWalk(origin=0, destination=4, n_simulations=1000)
print("\n*****")
```

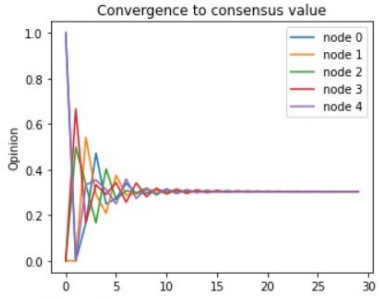
```
******
Problem 1 - C :

Average return time: 8.1396675878699 s

*****
```

In [6]:
```
node = 4  # node d
S = [node]
R = [node for node in range(n_nodes) if node not in S]

hatP = P[np.ix_(R, R)]
hatw = w[np.ix_(R)]

hatx = np.linalg.solve((np.identity(n_nodes - len(S)) - hatP), (np.ones(n_nodes - len(S)) / hatw))

hitting_s = np.zeros(n_nodes)
hitting_s[R] = hatx

expected_od = hitting_s[0]

print("******\nProblem 1 - D : \n")
print("Hitting time: {} time units ".format(expected_od))
print("Error simulation", abs(simulationAvgTime - expected_od))
print("\n*****")
```

```
******
Problem 1 - D :

Hitting time: 8.785714285714285 time units
Error simulation 0.6460466978443851

*****
```

In [7]:
```
# Problem1 e
# number of iterations
n_iter = 30

# ksave the evolution of the consensus
x = np.zeros((len(G.nodes), n_iter))

# set initial condition (1,0,0,0,1)
x[:, 0] = np.array([1, 0, 0, 0, 1])
# evolve the states
for t in range(1, n_iter):
    x[:, t] = P @ x[:, t - 1]

x[:, n_iter - 1]

print("******\nProblem 1 - E : \n")
print("The consensus state is:", x[:, n_iter - 1])

values, vectors = 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(values[i], 1)]:
    pi = vectors[:, index].real  # -> eigenvectors are complex but pi is real, so we convert it to real
    pi = pi / np.sum(pi)
# found pi, we have to multiply by the initial condition
x0 = x[:, 0]

print("The consensus value is:", pi @ x0)
print("\n*****")

# plot the record of the consensus vector to have a better look of the convergence
fig = plt.figure(1, figsize=(5, 4))
ax = plt.subplot(111)
ax.legend()
ax.set_title("Convergence to consensus value")
plt.xlabel('Iterations')
plt.ylabel('Opinion')
plt.savefig("Ex1_E.svg")
plt.show()
```

```
******
Problem 1 - E :

The consensus state is: [0.304 0.304 0.304 0.304 0.304]
The consensus value is: 0.30434782608695654

*****
```

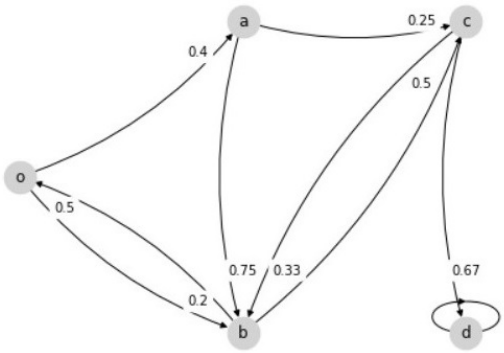


In [8]:
```
# Problem1 f
alfa_err = np.zeros(200)

# with numerical simulations
for i in range(200):

    # since rand returns random values in [0,1], mu = 1/2
    x = np.random.rand(5)
    for n in range(500):
        x = P @ x
    alfa_err[i] = (1 / 2 - np.mean(x)) * (1 / 2 - np.mean(x))

print("******\nProblem 1 - F : \n")
print("The variance of the consensus state is:", np.mean(alfa_err), "\n")
values, vectors = np.linalg.eig(Q.T)
index = np.argmax(values.real)
pi = vectors[:, index].real
pi = pi / np.sum(pi)
# uniform distribution, so the variance is 1/12
sigma = 1 / 12
print("Expected variance:", np.sum(np.square(pi)) * 1 / 12)
print("\n*****")
```

```
******
Problem 1 - F :

The variance of the consensus state is: 0.021506366564129152

Expected variance: 0.017032464563328763

*****
```

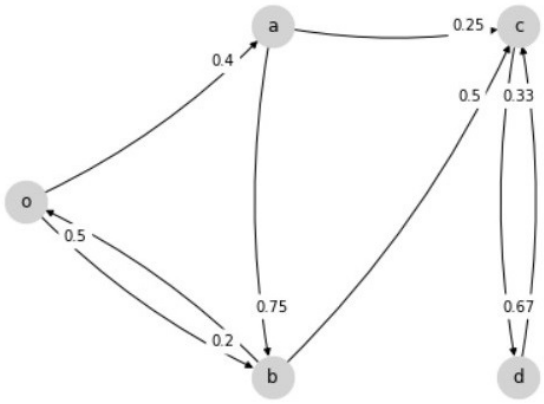
In [9]:
```
# Problem1 g
# remove edges
G.remove_edges_from([('d', 'a'), ('d', 'c')])
# add self loop to d
G.add_weighted_edges_from([('d', 'd', float("{0:.2f}".format(1 / 10000)))])
pos = {'o': (0, 0), 'a': (1, 2), 'b': (1, -2), 'c': (2, 2), 'd': (2, -2)}
nx.draw(G, pos=pos, with_labels=True, node_size=600, font_size=12, node_color='lightgray',
        connectionstyle='arc3, rad = 0.15')
labels = {e: G.edges[e]['weight'] for e in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, label_pos=0.20, rotate=False, horizontalalignment='center')

# calculate P
W = nx.adjacency_matrix(G)
W = W.toarray()
W[4][4] = 0.00001  # self loop weight lowest possible
degrees = np.sum(W, axis=1)
D = np.diag(degrees)
P = np.linalg.inv(D) @ W

n_iter = 70

# arbitrary initial condition
x = np.random.rand(5)
print("******\nProblem 1 - G : \n")
print('Arbitary initial condition x(0):', x)
for i in range(n_iter):
    x = P @ x
print("The asymptotic state x is:", x)

# with numerical simulations
for i in range(200):

    # since rand returns random values in [0,1], mu = 1/2
    x = np.random.rand(5)
    var = np.var(x)
    for n in range(500):
        x = P @ x
    alfa_err[i] = (1 / 2 - np.mean(x)) * (1 / 2 - np.mean(x))

print("The variance of the consensus state is:", np.mean(alfa_err), "\n")
print("\n*****")
```

```
******
Problem 1 - G :

Arbitary initial condition x(0): [0.764 0.723 0.04  0.124 0.794]
The asymptotic state x is: [0.794 0.794 0.794 0.794 0.794]
The variance of the consensus state is: 0.07852765531998336


*****
```



In [10]:
```
# Problem1 h
# reconstruct the new graph
G = nx.DiGraph()
G.add_weighted_edges_from([('o', 'a', float("{0:.2f}".format(2 / 5))), ('o', 'b', float("{0:.2f}".format(1 / 5))),
                           ('a', 'b', float("{0:.2f}".format(3 / 4))), ('a', 'c', float("{0:.2f}".format(1 / 4))),
                           ('b', 'o', float("{0:.2f}".format(1 / 2))), ('b', 'c', float("{0:.2f}".format(1 / 2))),
                           ('c', 'd', float("{0:.2f}".format(2 / 3))), ('d', 'c', float("{0:.2f}".format(1 / 3)))])

pos = {'o': (0, 0), 'a': (1, 2), 'b': (1, -2), 'c': (2, 2), 'd': (2, -2)}
nx.draw(G, pos=pos, with_labels=True, node_size=800, font_size=12, node_color='lightgray',
        connectionstyle='arc3, rad = 0.1')
labels = {e: G.edges[e]['weight'] for e in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, label_pos=0.20, rotate=False, horizontalalignment='center')

# reconstruct P
W = nx.adjacency_matrix(G)
W = W.toarray()
degrees = np.sum(W, axis=1)
D = np.diag(degrees)
P = np.linalg.inv(D) @ W

n_iter = 30

x = np.zeros((5, n_iter))
# arbitrary initial condition
x[:, 0] = np.random.rand(5)

print("******\nProblem 1 - H : \n")
print('Arbitary initial condition x(0):', x[:, 0])
```

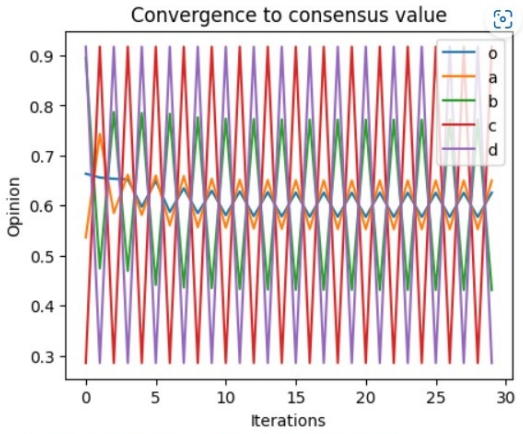
```
******
Problem 1 - H :

Arbitary initial condition x(0): [0.664 0.536 0.895 0.286 0.917]
```



In [11]:
```
# evolve the states
for t in range(1, n_iter):
    x[:, t] = P @ x[:, t - 1]

print("Final opinions:", x[:, n_iter - 1])
print("\n*****")
```

```
Final opinions: [0.626 0.65  0.431 0.917 0.286]

*****
```

In [12]:
```
fig = plt.figure(figsize=(5, 4), dpi=100)
ax = plt.subplot(111)
node_names = ['o', 'a', 'b', 'c', 'd']

for node in range(5):
    op = x[node, :]
    ax.plot(range(n_iter), op, label=f'{node_names[node]}')

ax.legend()
ax.set_title("Convergence to consensus value")
plt.xlabel('Iterations')
plt.ylabel('Opinion')
plt.savefig("Ex1_H.svg")
plt.show()
```



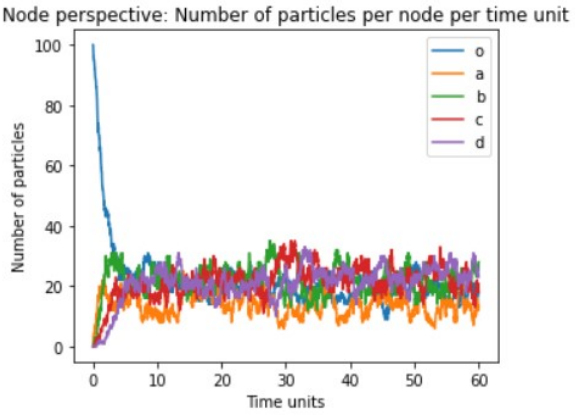
In [13]:
```
print("\n**********\nProblem2\n**********\n")
print("******\nProblem 2 - A : \n")
particles = np.array(range(100)).astype(np.int32)
simulationAvgTime = simulateRandomWalk(origin=1, destination=1, n_simulations=len(particles) * 1000)
print(simulationAvgTime)
print("Error simulation", abs(simulationAvgTime - expected_theoric))
print("\n*****")

# stating with 100 particles in node 'o'
n_particles = 100  # Equal to the rate
time_units = 60
hist_nodes = np.array([[100, 0, 0, 0, 0]], dtype=float)

n_nodes = np.zeros(len(G.nodes), dtype=float)
n_nodes[0] = n_particles
P_nodes = n_nodes / n_particles
P_nodes_cum = np.cumsum(P_nodes)

transition_times = []
transition_times.append(0)

t_next = -np.log(np.random.rand()) / n_particles
i = 0

timeSeq = []
timeSeq.append(0)

while transition_times[i] < time_units:
    i += 1

    transition_times.append(transition_times[i - 1] + t_next)
    timeSeq.append(transition_times[-1])

    start_index = np.argwhere(P_nodes_cum > np.random.rand())[0][0]
    dest_index = np.argwhere(Q_cum[start_index] >= np.random.rand())[0][0]
```

```
    n_nodes[dest_index] += 1
    n_nodes[start_index] -= 1

    hist_nodes = np.concatenate((hist_nodes, [n_nodes]), axis=0)

    P_nodes = n_nodes / n_particles
    P_nodes_cum = np.cumsum(P_nodes)

    t_next = -np.log(np.random.rand()) / n_particles

avg_particles = {}
for nodes, avg_value in zip(G.nodes, np.average(hist_nodes, axis=0)):
    avg_particles[nodes] = round(avg_value, 2)
```

```
**********
Problem2
**********

******
Problem 2 - A :

Average return time: 6.764747790576242 s
6.764747790576242
Error simulation 0.014747790576243425

*****
```

In [14]:
```
print("******\nProblem 2 - B : \n")
print("Particles per node at final step: ", avg_particles)
print("Average number of particles in every node pi_bar: ", 100 * pi_bar)
print("\n*****")
fig, ax = plt.subplots(figsize=(5, 4))
labels = dict(enumerate(G.nodes))

for i in range(len(G)):
    ax.plot(timeSeq, hist_nodes[:, i], label=labels[i])

ax.legend()
ax.set_title("Node perspective: Number of particles per node per time unit")
plt.xlabel('Time units')
plt.ylabel('Number of particles')
plt.savefig("Ex2_B.svg")
plt.show()
```

```
******
Problem 2 - B :

Particles per node at final step:  {'o': 21.62, 'a': 14.04, 'b': 21.87, 'c': 21.22, 'd': 21.24}
Average number of particles in every node pi_bar:  [18.519 14.815 22.222 22.222 22.222]

*****
```



In [15]:
```
print("\n**********\nProblem3\n**********\n")
# draw graph
G = nx.DiGraph()
G.add_nodes_from(['o', 'a', 'b', 'c', 'd', "d'"])
G.add_weighted_edges_from([('o', 'a', float("{0:.2f}".format(2 / 3))), ('o', 'b', float("{0:.2f}".format(1 / 3))),
                           ('a', 'b', float("{0:.2f}".format(1 / 4))), ('a', 'c', float("{0:.2f}".format(1 / 4))),
                           ('a', 'd', float("{0:.2f}".format(2 / 4))), ('b', 'c', 1), ('c', 'd', 1), ('d', "d'", 0)])
pos = nx.spring_layout(G)
pos = {'o': (0, 0), 'a': (1, 2), 'b': (1, -2), 'c': (2, 2), 'd': (2, -2), "d'": (3, -2)}
nx.draw(G, pos=pos, with_labels=True, node_size=800, font_size=12, node_color='lightgray',
        connectionstyle='arc3, rad = 0.1')
labels = {e: G.edges[e]['weight'] for e in G.edges}
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, label_pos=0.20, rotate=False, horizontalalignment='center')
```

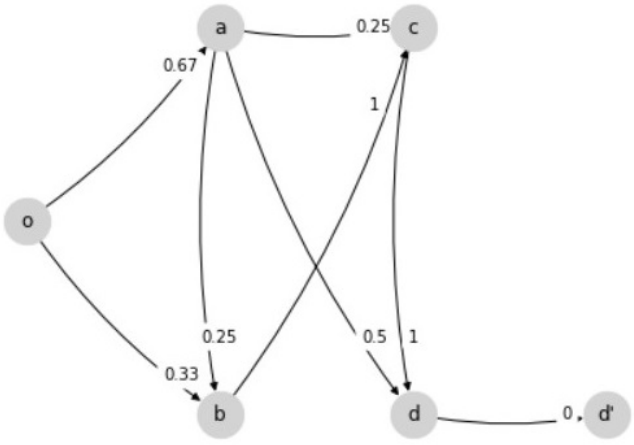
```
lambda_matrix = [[0, 3 / 4, 3 / 8, 0, 0],
                 [0, 0, 1 / 4, 1 / 4, 2 / 4],
                 [0, 0, 0, 1, 0],
                 [0, 0, 0, 0, 1],
                 [0, 0, 0, 0, 0],
                 ]
w = np.sum(lambda_matrix, axis=1)

G_nodes = len(G.nodes)

w_star = np.max(w)
Q = lambda_matrix / w_star
Q = Q + np.diag(np.ones(len(w)) - np.sum(Q, axis=1))
Q_cum = np.cumsum(Q, axis=1)

print("Number of nodes: ", n_nodes)
print("\nVector w:\n", w)
print("\nMatrix D:\n", D)
print("\nMatrix P:\n", P)
print("\nMatrix P_cum:\n", P_cum)
print("\nMatrix Q:\n", Q)
print("\nMatrix Q_cum:\n", Q_cum)
```

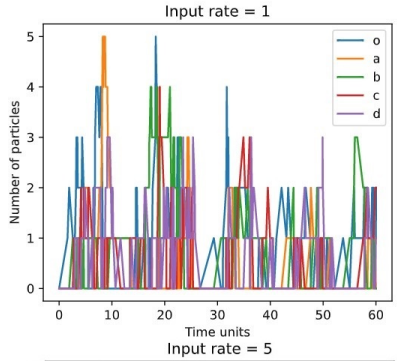
```
def get_starting_node(node_particles, rate):
    particles = rate

    # 1 dummy variable for entry in node 'o'
    # Particles in nodes
    n_nodes = np.zeros(G_nodes)
    n_nodes[5] = rate

    particles += np.sum(node_particles)

    for i in range(G_nodes - 1):
        n_nodes[i] = node_particles[i]

    particles_cum = np.cumsum(n_nodes) / particles
    start_node = np.argwhere(particles_cum > np.random.rand())[0][0]
    # print(start_node)

    return start_node, particles
```

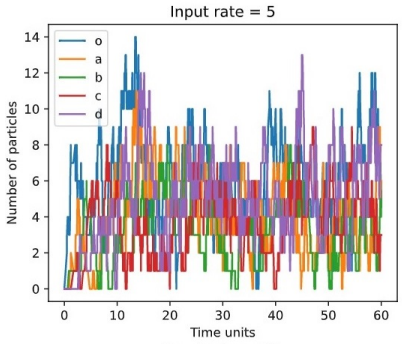
```
def simulate_proportional_rate(time_units, rate):
    # particles in node
    node_particles = np.zeros(G_nodes - 1)

    transition_times = []
    transition_times.append(0)

    hist_nodes = np.array([[0, 0, 0, 0, 0]])

    while True:

        start_node, particles = get_starting_node(node_particles, rate)
        t_next = transition_times[-1] - np.log(np.random.rand()) / particles

        # check dummy variable
        if start_node == 5:
            node_particles[0] += 1
        elif start_node == 4:
            node_particles[4] -= 1

        else:
            end_node = np.argwhere(Q_cum[start_node] > np.random.rand())[0][0]
            node_particles[start_node] -= 1
            node_particles[end_node] += 1

        transition_times.append(t_next)

        hist_nodes = np.concatenate((hist_nodes, [node_particles]), axis=0)

        if t_next > time_units:
            break

    return hist_nodes, transition_times
```

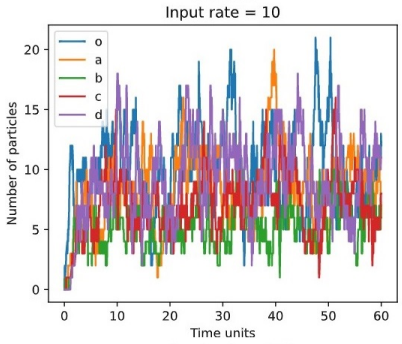
```
def plot_proportional_trajectories(transition_times, hist_nodes, input_rate):
    fig, ax = plt.subplots(figsize=(5, 4), dpi=100)
    labels = {0: 'o', 1: 'a', 2: 'b', 3: 'c', 4: 'd'}
    for i in range(G_nodes - 1):  # only 5 nodes, excluding d'
        ax.plot(transition_times, hist_nodes[:, i], label=labels[i])

    ax.legend()
    plt.xlabel("Time units")
    plt.ylabel("Number of particles")
    plt.title("Input rate = {}".format(input_rate))
    plt.savefig("ParticlesProportionalRate" + str(input_rate) + ".svg")
    plt.show()
    plt.close()
input_rates = [1, 5, 10, 100, 200]
time_units = 60
for input_rate in input_rates:
    hist_nodes, transition_times = simulate_proportional_rate(time_units, input_rate)
    plot_proportional_trajectories(transition_times, hist_nodes, input_rate)
```

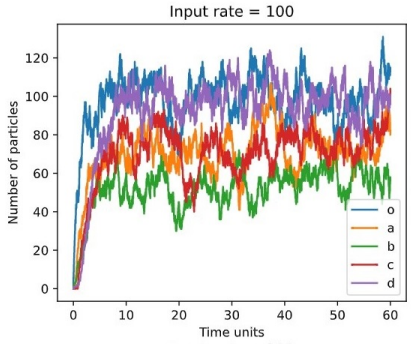
```
Problem3
**********

Number of nodes:  [18. 13. 28. 18. 23.]

Vector w:
 [1.125 1.    1.    1.    0.   ]

Matrix D:
 [[0.6  0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.  ]
 [0.   0.   1.   0.   0.  ]
 [0.   0.   0.   0.67 0.  ]
 [0.   0.   0.   0.   0.33]]

Matrix P:
 [[0.    0.667 0.333 0.    0.   ]
 [0.    0.    0.75  0.25  0.   ]
 [0.5   0.    0.    0.5   0.   ]
 [0.    0.    0.    0.    1.   ]
 [0.    0.    0.    1.    0.   ]]

Matrix P_cum:
 [[0.    0.667 1.    1.    1.   ]
 [0.    0.    0.75  1.    1.   ]
 [0.5   0.5   0.5   1.    1.   ]
 [0.    0.    0.333 0.333 1.   ]
 [0.    0.5   0.5   1.    1.   ]]

Matrix Q:
 [[0.    0.667 0.333 0.    0.   ]
 [0.    0.111 0.222 0.222 0.444]
 [0.    0.    0.111 0.889 0.   ]
 [0.    0.    0.    0.111 0.889]
 [0.    0.    0.    0.    1.   ]]
Matrix Q_cum:
 [[0.    0.667 1.    1.    1.   ]
 [0.    0.111 0.333 0.556 1.   ]
 [0.    0.    0.111 1.    1.   ]
 [0.    0.    0.    0.111 1.   ]
 [0.    0.    0.    0.    1.   ]]
```

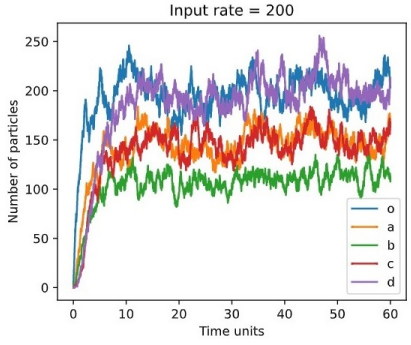



Input rate = 1
Number of particles
Time units
o
a
b
c
d

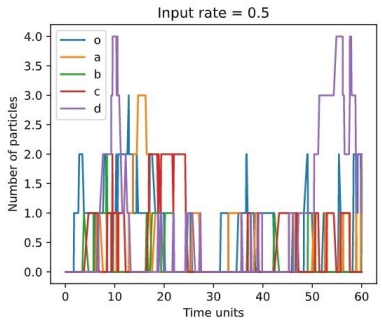

Input rate = 5
Number of particles
Time units
o
a
b
c
d

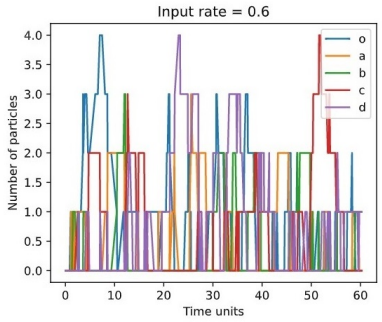

Input rate = 10
Number of particles
Time units
o
a
b
c
d

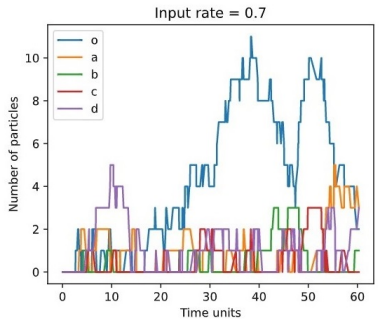

Input rate = 100
Number of particles
Time units
o
a
b
c
d

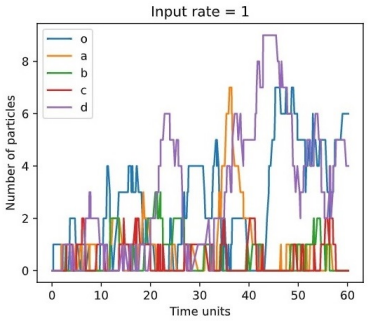

Input rate = 200
Number of particles
Time units
o
a
b
c
d

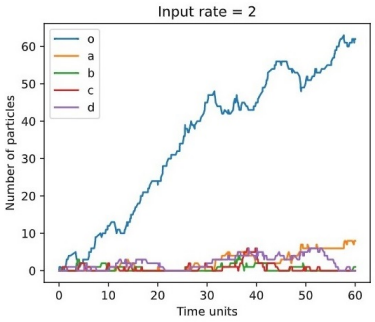
In [16]:
```python
# problem 3 : B
def get_starting_node(rate):
    total_rate = rate

    # 1 dummy variable for entry in node 'o'
    n_nodes = np.zeros(G_nodes)
    n_nodes[5] = rate

    for i in range(G_nodes - 1):
        n_nodes[i] = 1
        total_rate += 1

    particles_cum = np.cumsum(n_nodes) / total_rate
    start_node = np.argwhere(particles_cum > np.random.rand())[0][0]

    return start_node, total_rate


def simulate_fixed_rate(time_units, rate):
    node_particles = np.zeros(G_nodes - 1)

    transition_times = []
    transition_times.append(0)

    hist_nodes = np.array([[0, 0, 0, 0, 0]])

    while True:

        start_node, total_rate = get_starting_node(rate)
        t_next = transition_times[-1] - np.log(np.random.rand()) / total_rate

        # check dummy variable
        if start_node == 5:
            node_particles[0] += 1

        elif node_particles[start_node] == 0:
            # Do nothing if I don't have any particle
            # Check after if start_nod == 5 but before start_node == 4.
            pass
        elif start_node == 4:
            node_particles[4] -= 1

        else:
            end_node = np.argwhere(Q_cum[start_node] > np.random.rand())[0][0]
            node_particles[start_node] -= 1
            node_particles[end_node] += 1

        transition_times.append(t_next)
        hist_nodes = np.concatenate((hist_nodes, [node_particles]), axis=0)

        if t_next > time_units:
            break

    return hist_nodes, transition_times


def plot_fixed_rate_trajectories(transition_times, hist_nodes, input_rate):
    fig, ax = plt.subplots(figsize=(5, 4), dpi=100)
    labels = {0: 'o', 1: 'a', 2: 'b', 3: 'c', 4: 'd'}
    for i in range(G_nodes - 1):  # only 5 nodes, excluding d'
        ax.plot(transition_times, hist_nodes[:, i], label=labels[i])

    ax.legend()
    plt.xlabel("Time units")
    plt.ylabel("Number of particles")
    plt.title("Input rate = {}".format(input_rate))
    plt.savefig("ParticlesFixedRate" + str(input_rate) + ".svg")
    plt.show()
    plt.close()
input_rates = [0.5, 0.6, 0.7, 1, 2]
time_units = 60
for input_rate in input_rates:
    hist_nodes, transition_times = simulate_fixed_rate(time_units, input_rate)
    plot_fixed_rate_trajectories(transition_times, hist_nodes, input_rate)
```


Input rate = 0.5
Number of particles
Time units
o
a
b
c
d


Input rate = 0.6
Number of particles
Time units
o
a
b
c
d


Input rate = 0.7
Number of particles
Time units
o
a
b
c
d


Input rate = 1
Number of particles
Time units
o
a
b
c
d


Input rate = 2
Number of particles
Time units
o
a
b
c
d