Network Dynamics, and Learning (Homework 2)

Hesam Khanjani 310141
Shaghayegh Abedi 309894
Nastaran Ahmadi Bonakdar 310073
Mohammadreza Mashhadigholamali 308499
Ali Samimi Fard 308956

December 15, 2023

Contributions to the Project

Question one is solved by Shaghayegh Abedi and Nastaran Ahmadi Bonakdar.

Question two is solved by Hesam Khanjani and Mohammadreza Mashhadigholamali with the help of Ali Samimi Fard.

Question three is solved by Mohammadreza Mashhadigholamali and Ali Samimi Fard with the help of Hesam Khanjani.

The report was written in LaTeX by Hesam Khanjani, Mohammadreza Mashhadigholamali, and Ali Samimi Fard.

Continuous Time Markov Chains

In Continuous Time Markov Chains (CTMC), time flows in a continuum ($t \geq 0$) rather than being discrete ($t = 0, 1, \ldots$). The evolution of a state variable x inside a discrete state space $\mathcal{X}$ with a graph structure is still described by the random process. This graph, $G = (\mathcal{X}, \Lambda)$, has nodes $\mathcal{X}$ and a weight matrix Λ that indicates potential node/state transitions.

These transitions occur at random times determined by the ticking of a clock known as the Poisson clock. The time interval that separates any two consecutive ticks of a Poisson clock can be thought of as an independent random variable with an exponential distribution and a predetermined rate. This fact will help to simulate continuous time Markov chains.

Note: We represent this time interval by t_{next}. This is necessary to simulate a Poisson clock with rate r. The computation of t_{next} is as follows:

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

where u is a uniformly distributed random variable and $u \in U(0, 1)$.

Being memoryless is the main characteristic of the exponential distribution:

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

Modeling Continuous Time Markov Chains

Continuous Time Markov Chains (CTMCs) can be modeled in two similar ways.

The First Strategy:

1. You specify a distinct **global** Poisson clock with a suitable rate $\omega^* = \max_i(\omega_i)$ where $\omega_i = \sum_j \Lambda_{ij}$.
2. When **the global clock ticks** while you are at node i, you either jump to a neighbor j with probability $Q_{ij} = \frac{\Lambda_{ij}}{\omega^*}$, where $i \neq j$, or you stay in the same node (no transition) with probability $Q_{ii} = 1 - \sum_{i \neq j} Q_{ij}$.

This method uses a global clock to "discretize" the continuous time, with the jumps being described by the matrix Q. The matrix Q is referred to as the Continuous Time Markov Chain's (**CTMC**) **jump chain** for this reason. Observe that for the nodes i maximizing ω, $Q_{ii} = 0$; this value increases as $\frac{\omega_i}{\omega}$ decreases.

The Second Strategy:

1. Every node i is outfitted with a Poisson clock of rate $\omega_i = \sum_j \Lambda_{ij}$.
2. At node i, you jump to a neighbor j with probability $P_{ij} = \frac{\Lambda_{ij}}{\omega_i}$ when **that node's clock ticks**.

We must add a self-loop $\Lambda_{ii} > 0$ for nodes i such that $\omega_i = 0$ (i.e., once the process is in i, it remains in i forever); otherwise, the matrix P is not well-defined.

The following is the definition of the probability distribution $\bar{\pi}(t)$ of the Continuous Time Markov Chain (**CTMC**) $X(t)$ with transition rate matrix Λ.

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

It changes in accordance with the formula:

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

where $L = diag(w) - \Lambda$, with $w = \Lambda_1$.

The eigenvector of L' corresponding to eigenvalue 0 is thus the invariant probability vectors. Additionally, the left dominant eigenvector of Q, which is the row-stochastic matrix defined as Q, can be shown to be $\bar{\pi}$.

$$Q_{ij} = \frac{\Lambda_{ij}}{\omega^*}, \quad i \neq j$$

$$Q_{ij} = 1 - \sum_{i \neq j} Q_{ij}$$

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

Problem 1

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

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

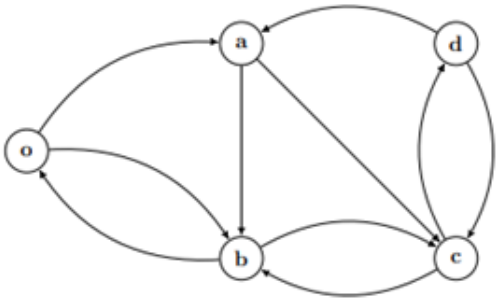

Figure 1: Closed network in which particles move according to the transition rate matrix(1)

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

Solution

To simulate a continuous random walk is the goal of this exercise. It is our task to replicate these network movements in the provided graph format, using the provided transition rate matrix as a guide. Using Λ as our transition matrix, we display the graph in accordance with the inputs. We use the previously stated second method.

The transition rate matrix is to be calculated as follows based on our transition matrix. For every node ω_i, it is acquired in this manner:

$$\forall i, j \in \mathcal{X}, \quad \omega_i = \sum \Lambda_{ij}$$

The conditional probabilities of moving between nodes i and j, called Q, and remaining at that node in the subsequent step are $1 - Q$. It is necessary for us to use a single-position clock in a node that has the same transmission rate as ω_i.

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

By identifying the leading eigenvector of the matrix Q', we can compute the invariant probability vector $\bar{\pi}$ of the Continuous Time Markov Chain (**CTMC**). The direction of this vector corresponds to the probability of being at node i. Moving from the position clock with rate ω_i for each node i in this network is our main objective in this simulation.

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

     # Set a fixed seed for random number generation and adjust numpy print settings
     np.random.seed(42)
     np.set_printoptions(precision=3, suppress=True)

     # Define the matrix representing transition probabilities
     transition_matrix = np.array([
         [0, 2 / 5, 1 / 5, 0, 0],
         [0, 0, 3 / 4, 1 / 4, 0],
         [1 / 2, 0, 0, 1 / 3, 0],
         [0, 0, 1 / 3, 0, 2 / 3],
         [0, 1 / 3, 0, 1 / 3, 0]
     ])

     # Create and populate a directed graph
     graph = nx.DiGraph()
     nodes = ['o', 'a', 'b', 'c', 'd']
     edges = [
         ('o', 'a', 2 / 5), ('o', 'b', 1 / 5), ('a', 'b', 3 / 4), ('a', 'c', 1 / 4),
         ('b', 'o', 1 / 2), ('b', 'c', 1 / 2), ('c', 'b', 1 / 3), ('c', 'd', 2 / 3),
         ('d', 'a', 1 / 3), ('d', 'c', 1 / 3)
     ]
     graph.add_nodes_from(nodes)
     graph.add_weighted_edges_from([(u, v, round(w, 2)) for u, v, w in edges])

     # Visualize the graph
     # Circular layout for the positions of the nodes
     positions = nx.circular_layout(graph)
     nx.draw(graph, pos=positions, with_labels=True, node_color='lightgray')
     edge_labels = nx.get_edge_attributes(graph, 'weight')
     nx.draw_networkx_edge_labels(graph, positions, edge_labels=edge_labels, label_pos=0.2, rotate=False)
     plt.show()
```

```
# Calculate the number of nodes in the graph
num_nodes = graph.number_of_nodes()

# Sum of weights in the transition matrix
weight_sum = np.sum(transition_matrix, axis=1)

# Compute the maximum weight
max_weight = np.max(weight_sum)

# Construct the normalized transition probability matrix
D_matrix = np.diag(weight_sum)
P_matrix = np.linalg.inv(D_matrix).dot(transition_matrix)

# Construct the Q matrix for further computations
Q_matrix = transition_matrix / max_weight
Q_matrix += np.diag(np.ones(num_nodes) - np.sum(Q_matrix, axis=1))

# Display various matrices and their properties
print("Number of nodes:", num_nodes)
print("Weight Vector:\n", weight_sum)
print("Diagonal Matrix D:\n", D_matrix)
print("Normalized Transition Matrix P:\n", P_matrix)
print("Matrix Q:\n", Q_matrix)

# Calculate the dominant eigenvector for the invariant distribution
eigenvalues, eigenvectors = np.linalg.eig(Q_matrix.T)
dominant_index = np.argmax(eigenvalues.real)
invariant_distribution = eigenvectors[:, dominant_index].real
invariant_distribution /= np.sum(invariant_distribution)
print("Invariant Distribution (pi_bar):\n", invariant_distribution)
```

```python
import numpy as np
import matplotlib.pyplot as plt

# Define the graph with weighted edges
edges = {'o': {'a': 3/4, 'b': 3/4},
         'a': {'b': 1/4, 'c': 1/4, 'd': 2/4},
         'b': {'c': 1},
         'c': {'d': 1},
         'd': {"d'": 0},
         "o'": {'o': 0}}

# Define node positions
pos = {'o': (0, 0), 'a': (1, 2), 'b': (1, -2), 'c': (2, 2), 'd': (2, -2), "d'": (3, -2), "o'": (-1, 0)}

# Draw nodes
for node, position in pos.items():
    plt.scatter(*position, label=node, s=800, color='lightgray')

# Draw edges with arrows and labels
for start_node, edges_dict in edges.items():
    for end_node, weight in edges_dict.items():
        plt.annotate("", xy=pos[end_node], xytext=pos[start_node], arrowprops=dict(arrowstyle="->", connectionstyle="arc3"), alpha=0.5, color='black')
        plt.text((pos[start_node][0] + pos[end_node][0]) / 2, (pos[start_node][1] + pos[end_node][1]) / 2, f"{weight:.2f}", fontsize=10, ha='center', va='center')

# Draw labels for nodes
for node, position in pos.items():
    plt.text(position[0], position[1], node, fontsize=12, color='black')

# Set plot title and turn off axis
plt.title("Directed Graph")
plt.axis('off')

# Show the plot
plt.show()
```

Problem 1: a

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

In this code segment, we define a function named `simulationAvgTime` which executes a random walk simulation within a network framework. The primary goal of this function is to compute the average time it takes for a particle to leave a specified node (indexed as 2, corresponding to node 'b') and then return to it. To ensure statistical robustness, the simulation is conducted 10,000 times, as indicated by the simulation's parameter. Each iteration within the simulation records the elapsed time it takes for the particle to travel from node 'b' back to itself. This process is governed by the transition probabilities specified in the `transition_probabilities` array.

The random walk procedure in each iteration involves calculating the time until the next transition, modeled using an exponential distribution. This distribution represents a Poisson clock mechanism, where the rate parameter is determined by the maximum transition rate (`max_transition_rate`) calculated from the transition probabilities. This Poisson clock is a key feature of the simulation, encapsulating the random and memoryless nature of the transition intervals.

The function dynamically updates the particle's position during each simulation iteration. It does so by generating a random number and comparing it against the cumulative transition probabilities in the `cumulative_Q` matrix. The simulation continues until the particle, after having left its starting node, returns to it.

Once the simulation cycles through all iterations, the function calculates and returns the mean of these recorded times. This mean represents the average time required for a particle to complete a round trip, starting and ending at node 'b', within this network model.

```python
# Problem1 a
import numpy as np

# Set the random seed for reproducibility
np.random.seed(123)

# Transition probability matrix
transition_probabilities = np.array([
    [0, 0.4, 0.2, 0, 0],
    [0, 0, 0.75, 0.25, 0],
    [0.5, 0, 0, 0.33, 0],
    [0, 0, 0.33, 0, 0.67],
    [0, 0.33, 0, 0.33, 0]
])

# Maximum transition rate for the Poisson process
summed_probabilities = transition_probabilities.sum(axis=1)
max_transition_rate = summed_probabilities.max()

# Matrix Q and its cumulative sum
matrix_Q = transition_probabilities / max_transition_rate + np.diag(1 - summed_probabilities / max_transition_rate)
cumulative_Q = np.cumsum(matrix_Q, axis=1)

def simulationAvgTime(start, target, simulations=10000):
    total_return_time = 0

    for _ in range(simulations):
        node_position = start
        elapsed_time = 0
        has_departed_start = False

        # Loop until returning to the target node after departure
        while not (node_position == target and has_departed_start):
            elapsed_time += np.random.exponential(1 / max_transition_rate)
            random_threshold = np.random.rand()
            next_node = np.searchsorted(cumulative_Q[node_position], random_threshold)

            if node_position != start:
                has_departed_start = True

            node_position = next_node

        total_return_time += elapsed_time

    return total_return_time / simulations

# Run simulation
avg_return_time = simulationAvgTime(start=2, target=2)
print(f"Average return time to node 'b': {avg_return_time:.2f} units")
```

```
Average return time to node 'b': 4.67 units
```

Figure 2: Code part 1a

Problem 1: b

How does the result in a) compare to the theoretical return-time $E_b[T_b]$? (Include a description of how this is computed.)

In our analysis, we compute the expected hitting times: $\hat{x} = (E_i[T_S])_{i \in R}$ for a set S and for all nodes $i \in R = V\S$. These expected hitting times represent the average time required for a random walk to reach the set S from any node in R. The calculation of $\hat{x}$ involves solving a system of linear equations formulated as:

$$\hat{x} = \frac{1}{\hat{w}} + \hat{P} \cdot \hat{x}$$

Here, $\hat{P}$ is derived from the normalized adjacency matrix P of the graph by excluding the rows and columns corresponding to the nodes in S. This process effectively isolates the part of the network outside S, allowing us to focus on transitions among those nodes. The equation can be rearranged and expressed more explicitly as:

$$\hat{x} = (I - \hat{P})^{-1} \cdot \frac{1}{\hat{w}}$$

Remark: It's crucial to note that the matrix $(I - \hat{P})$ is invertible only if the nodes in $V \backslash S$ have at least one transition leading to S. If this matrix is not invertible, it implies

that a random walk starting from nodes in $V \backslash S$ will never reach S, making the hitting times theoretically infinite.

In the context of our Markov chain model, we apply this approach to calculate the theoretical return time from node 'b' to itself, denoted as $E_b[T_b]$. This is achieved by first determining the expected time to leave 'b' (given by $1/w_b$, where w_b is the total transition rate out of 'b') and then adding the expected time to hit 'b' again from the other nodes, computed as $\hat{x}$ weighted by the transition probabilities from 'b' to these nodes.

The theoretical return time, according to our calculation, provides a benchmark against which we compare the results from our simulation. This comparison is quantified as the 'error simulation', which measures the deviation of the simulated average return time (obtained in part a) from the theoretical value calculated here. This error reflects the accuracy and reliability of our simulation model in approximating complex stochastic processes described by Markov chains.

Theoretical Return Time: 4.6000000000000005
Error Simulation: 0.07288474618606244

```python
# Problem 1-b
# Define the target node and create the set of the target node
target_node = 2
target_node_set = [target_node]

# Define the set of non-target nodes
non_target_nodes = list(set(range(num_nodes)) - set(target_node_set))

# Create a restricted transition matrix for non-target nodes
restricted_transition_matrix = P_matrix[np.ix_(non_target_nodes, non_target_nodes)]
restricted_weights = weight_sum[np.ix_(non_target_nodes)]

# Solve the linear system to find the hitting times for non-target nodes
restricted_hitting_times = np.linalg.solve(np.identity(num_nodes - len(target_node_set)) - restricted_transition_matrix,
                                           np.ones(num_nodes - len(target_node_set)) / restricted_weights)

# Initialize the array for hitting times with zeros
all_hitting_times = np.zeros(num_nodes)

# Assign the calculated hitting times to non-target nodes
all_hitting_times[non_target_nodes] = restricted_hitting_times

print('Expected hitting times to node b: ', all_hitting_times)

# Calculate the theoretical expected return time to the target node
theoretical_return_time = 1 / weight_sum[target_node] + np.dot(P_matrix[target_node, :], all_hitting_times)

# Compute the difference between the simulated and theoretical return times
Error_simulation = abs(avg_return_time - theoretical_return_time)

print("******\nProblem 1 - B : \n")
print('Theoretical Return Time: ', theoretical_return_time)
print("Error_Simulation:", Error_simulation)
print("\n*****")
```

```
Expected hitting times to node b:  [3.  2.  0.  4.  4.5]
******
Problem 1 - B :

Theoretical Return Time:  4.6000000000000005
Error_Simulation: 0.07288474618606244

*****
```

Figure 3: Code part 1b

Problem 1: c

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

We are going to obtain an average time from node o to node d. The only distinction between our approach and the previous one is the nodes we used (origin=1 (o) and

destination=4 (d)).

Average Return Time from node o to node d: 10.759209863745694

```
[ ]  # Problem1-C
     print("******\nProblem 1 - C : \n")
     average_return_time = simulationAvgTime(start=0, target=4, simulations=10000)
     print("Average return time from node o to node d:",average_return_time)
```

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

Average return time from node o to node d: 10.759209863745694
```

Figure 4: Code part 1c

Problem 1: d

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

Here, calculating the hitting time and obtaining the error simulation are our goals. Using the formula $\frac{1}{\omega_o \overline{\pi}_d}$, we obtain the hitting time. Every calculation step is the same as in question part "b," with the exception of the start and finish nodes (o to d). The simulation error in this step is 0.007456802920971839.

```
# Problem 1-d
target_node_d = 4  # Target node is 'd' represented by index 4
target_set_d = [target_node_d]

# Determine the set of nodes excluding the target node 'd'
non_target_nodes_d = list(set(range(num_nodes)) - set(target_set_d))

# Restrict the transition matrix P to the set of non-target nodes and obtain a modified matrix
modified_transition_matrix_d = P_matrix[np.ix_(non_target_nodes_d, non_target_nodes_d)]
modified_weights_d = weight_sum[np.ix_(non_target_nodes_d)]

# Solve the linear system to find the hitting times for the non-target nodes
modified_hitting_times_d = np.linalg.solve(np.identity(num_nodes - len(target_set_d)) - modified_transition_matrix_d,
                                           np.ones(num_nodes - len(target_set_d)) / modified_weights_d)

# Initialize an array to store the hitting times for all nodes with zeros
all_hitting_times_d = np.zeros(num_nodes)
# Assign the calculated hitting times for the non-target nodes
all_hitting_times_d[non_target_nodes_d] = modified_hitting_times_d

# Expected hitting time from node 'o' to 'd'
expected_hitting_time_od = all_hitting_times_d[0]

print("******\nProblem 1 - D : \n")
print("Hitting time from 'o' to 'd': {} time units".format(expected_hitting_time_od))
# Compare the expected hitting time from 'o' to 'd' with the average return time calculated in part c
print("Error in simulation", abs(average_return_time - expected_hitting_time_od))
print("\n*****")
```

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

Hitting time from 'o' to 'd': 10.766666666666666 time units
Error in simulation 0.007456802920971839

*****
```

Figure 5: Code part 1d

Problem 1: e

Interpret the matrix Λ as the weight matrix of a graph $G = (V, \epsilon, \Lambda)$, and simulate the French-DeGroot dynamics on G with an arbitrary initial condition $x(0)$. Does the

dynamics converge to a consensus state for every initial condition $x(0)$?

In this part, the evolution of a dynamic system towards consensus is explored over a span of 60 iterations. The system is initialized with an arbitrary initial condition $x(0)$, and its state at each iteration is updated using matrix multiplication. Eigenvalues and eigenvectors of the transposed matrix $\hat{P}$ are then calculated. We identify the eigenvector corresponding to an eigenvalue close to 1, normalize it to represent a probability distribution, and print the final state of the system after 60 iterations. Additionally, we compute the consensus value by multiplying the normalized eigenvector with the initial condition. We then plot the results.

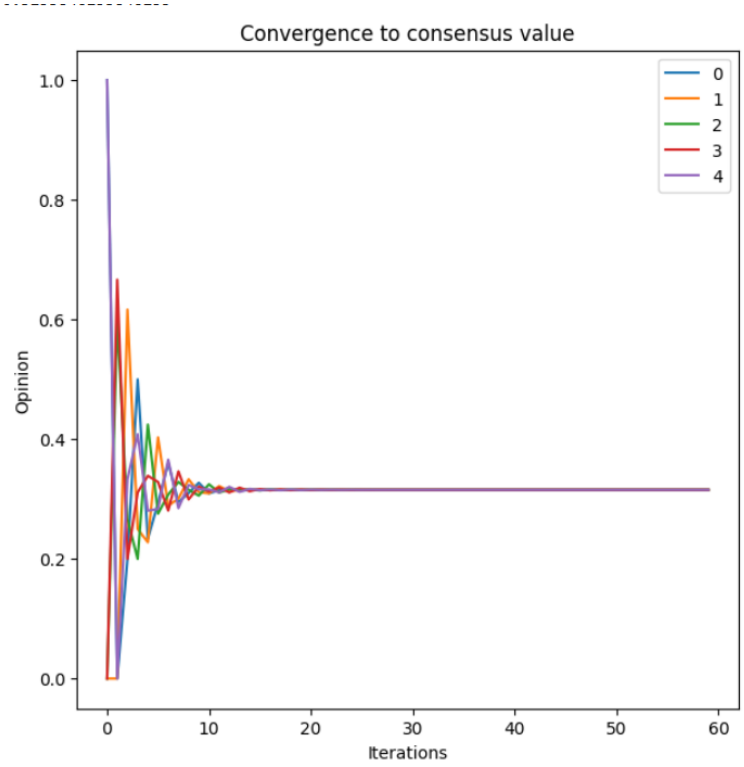

Figure 6: Convergence to consensus value

```python
itrs = 60
x = np.zeros((len(graph.nodes), itrs))
x[:, 0] = np.array([1, 0, 0, 0, 1]) # initial condition

# evolve the states
for t in range(1, itrs):
    x[:, t] = P_matrix @ x[:, t - 1]

values, vectors = np.linalg.eig(P_matrix.T)
# selecting the eigenvalue 1
for index in np.where(np.isclose(values, 1))[0]:
    pi = vectors[:, index].real / np.sum(vectors[:, index].real)

print("consensus state:")
print(x[:, itrs - 1])
print("consensus value:")
print(pi @ (x[:, 0])) # multiplying pi by the initial condition

# plotting
fig = plt.figure(1, figsize=(7, 7))
ax = plt.subplot(111)
ax.set_title("Convergence to consensus value")
plt.xlabel('Iterations')
plt.ylabel('Opinion')
for node in range(0, graph.number_of_nodes()):
    ax.plot(x[node, :], label=node)
ax.legend()
plt.show()
```

```
consensus state:
[0.315 0.315 0.315 0.315 0.315]
consensus value:
0.3153846153846155
```

Figure 7: Code part 1e

Problem 1: 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 $i \in V$ is given by $x_i(0) = \epsilon_i$, where $\{\epsilon_i\}_{i \in V}$ are i.i.d. random variables with variance σ^2. Compute analytically the variance of the consensus value.

In this part, we analyze the consensus convergence of a dynamic system. Initially, edges ('d', 'a') and ('d', 'c') are removed from the graph, and a self-loop with a weighted edge is added to node 'd'. The visual representation of the modified graph is illustrated using NetworkX. Subsequently, the matrix $\hat{P}$ is computed based on the adjusted graph structure, where the diagonal entries are inverses of the sum of weights in each row. Then, an arbitrary initial condition vector $\mathbf{x}$ is generated and printed. We then iterate the system through 100 steps using matrix multiplication with $\hat{P}$ to reach an asymptotic state. Additionally, numerical simulations are performed 300 times, each time with a random initial condition, to evaluate the variance of the consensus state after 400 iterations. The results provide insights into the system's convergence behavior and the impact of varying initial conditions on the consensus state variance.

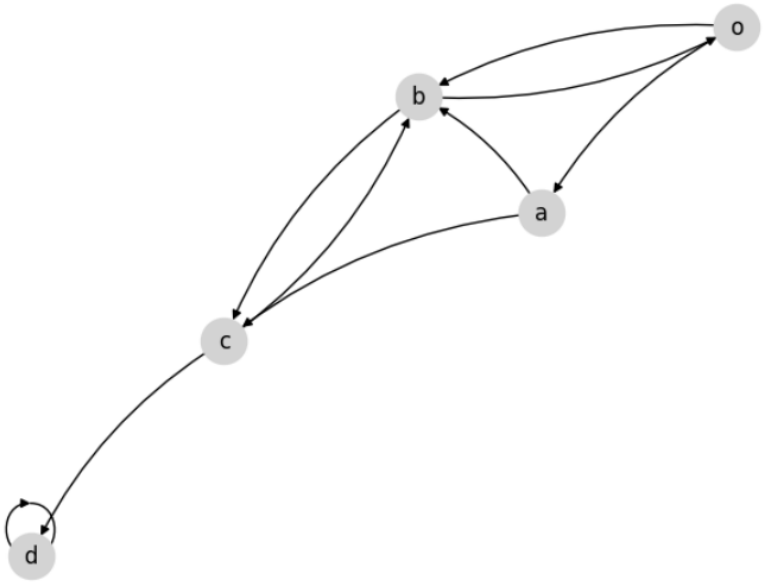

Figure 8: plot

```
graph.remove_edges_from([('d', 'a'), ('d', 'c')]) # remove edges
graph.add_weighted_edges_from([('d', 'd', float("{0:.2f}".format(1 / 1000000)))]) # self loop to d
nx.draw(graph, with_labels=True, node_size=600, font_size=12, node_color='lightgray',
        connectionstyle='arc3, rad = 0.15')

# calculating P
W = nx.adjacency_matrix(graph).toarray()
W[4][4] = 0.0000001  # self loop weight lowest possible
D_matrix = np.diag(np.sum(W, axis=1))
P_matrix = np.linalg.inv(D_matrix) @ W

x = np.random.rand(5) # initial condition
print("Arbitary initial condition x(0):")
print(x)

for i in range(0, 100):
    x = P_matrix @ x
print("The asymptotic state x:")
print(x)

# numerical simulations
alpha = np.zeros(300)
for i in range(300):
    x = np.random.rand(5)
    for n in range(0, 400):
        x = P_matrix @ x
    alpha[i] = (1 / 2 - np.mean(x)) * (1 / 2 - np.mean(x))

print("consensus state's variance:")
print(np.mean(alpha))
```

```
Arbitary initial condition x(0):
[0.557 0.875 0.117 0.202 0.633]
The asymptotic state x:
[0.633 0.633 0.633 0.633 0.633]
consensus state's variance:
0.07410085554907042
```

Figure 9: Code part 1g

Problem 1: h

Consider the graph (V, ϵ, Λ), and remove the edges (c, b) and (d, a). Analyze the French-DeGroot dynamics on the new graph. In particular, describe how the asymptotic behavior of the dynamics varies in terms of the initial condition $x(0)$, and motivate your answer.

In this part, we simulate the evolution of opinions in a directed graph G over 60 iterations. The transition matrix $\hat{P}$ is reconstructed from the new graph's adjacency matrix, considering the inverse of the diagonal degree matrix. The system's initial condition is set randomly, and the opinions of nodes are evolved over iterations using matrix multiplication with $\hat{P}$. Then we printed final opinions of each node, and we plotted the convergence of opinions over iterations. The plot depicts the evolution of opinions for each node, providing insights into the convergence dynamics and consensus values of the system.

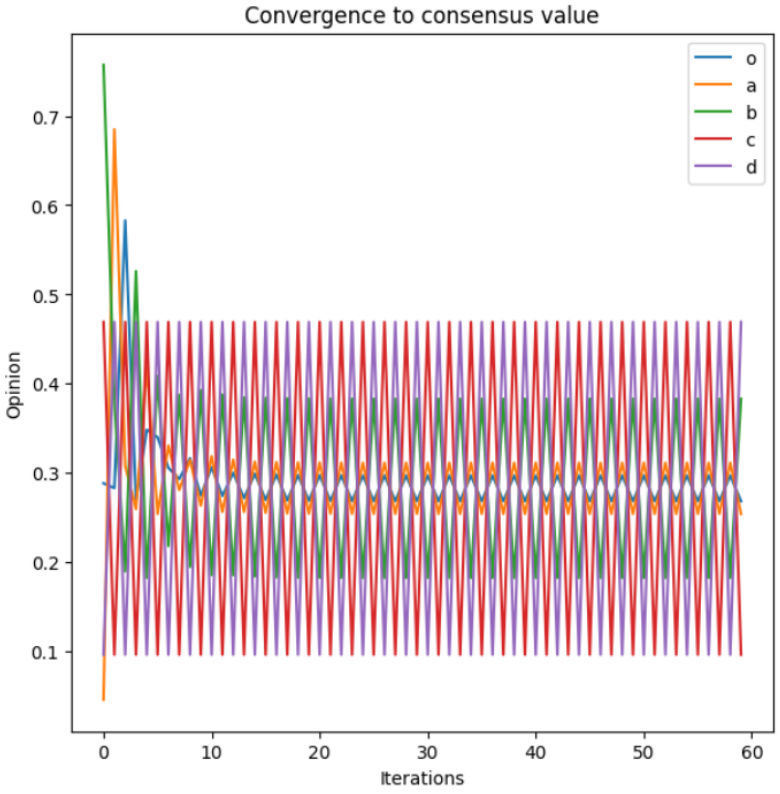

Figure 10: Convergence to consensus value

```python
itrs = 60
graph = nx.DiGraph()
graph.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))),
])

# reconstructing P
W = nx.adjacency_matrix(graph).toarray()
degrees = np.sum(W, axis=1)
D_matrix = np.diag(degrees)
P_matrix = np.linalg.inv(D_matrix) @ W

x = np.zeros((5, itrs))
x[:, 0] = np.random.rand(5) # initial condition
print('initial condition x(0):')
print(x[:, 0])

# evolving the states
for t in range(1, itrs):
    x[:, t] = P_matrix @ x[:, t - 1]

print("Final opinions:")
print(x[:, itrs - 1])

fig = plt.figure(figsize=(7, 7), dpi=100)
ax = plt.subplot(111)
node_names = ['o', 'a', 'b', 'c', 'd']

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

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

```
initial condition x(0):
[0.288 0.046 0.757 0.469 0.096]
Final opinions:
[0.268 0.254 0.383 0.096 0.469]
```

Figure 11: Code part 1h

Problem 2

Problem 2: a

In our exploration of the particle perspective, our focus is on determining the average time it takes for a particle to return to its starting node. This involves simulating the movement of 100 particles within the network, and specifically, measuring the time it takes for each particle to complete a round-trip from node 'b' to 'b'. We adopt a strategy involving a Poisson Clock mechanism with rates ω_i assigned to each node i within the discrete space.

To execute this simulation, we recognize that the independence and identical dis-

tribution of all particles allow us to treat the scenario as a random walk, where one particle's journey serves as a representative for the entire ensemble. Instead of conducting the simulation 1000 times, as in a previous problem, we opt for a more granular approach. We run the simulation 1000 times, but uniquely for each node, effectively simulating the movement of 100 particles in parallel.

In this section, our simulation framework remains consistent with a random walk implementation, where we traverse from node 2 to node 2 as an illustrative example. The key distinction lies in the multiplication factor: we scale the number of particles by 1000 to account for the individualized simulation for each node. This approach allows us to gain insights into the average return time for particles starting from node 'b'.

This nuanced simulation methodology ensures a detailed exploration of the particle perspective, unraveling the intricacies of individual trajectories within the network.
Average return time: 4.621099931422537
Error simulation: 0.021099931422536855

Problem 2: b

In this part, we're looking at 100 particles moving around for 60 time units, but this time, we're not doing a random walk like before. Now, we're more interested in how the nodes move, where they start, and where they end up. Instead of simulating 1000 particles, we're working with 100 particles this time. The way we simulate the random walk for these 100 particles is similar to doing a single particle's random walk but repeating it 100 times. When calculating the average, we change our approach from the second part of the initial problem. Rather than running a thousand simulations for each particle, we run the simulation a hundred times. The average we get is 4.62, quite close to the theoretical answer from the second part of the initial problem, which was 4.60.
Particles per node at final step: {'o': 23.98, 'a': 15.21, 'b': 25.88, 'c': 17.72, 'd': 17.22}
Average number of particles in every node $\bar{\pi}$: [21.739 14.907 26.087 18.634 18.634]

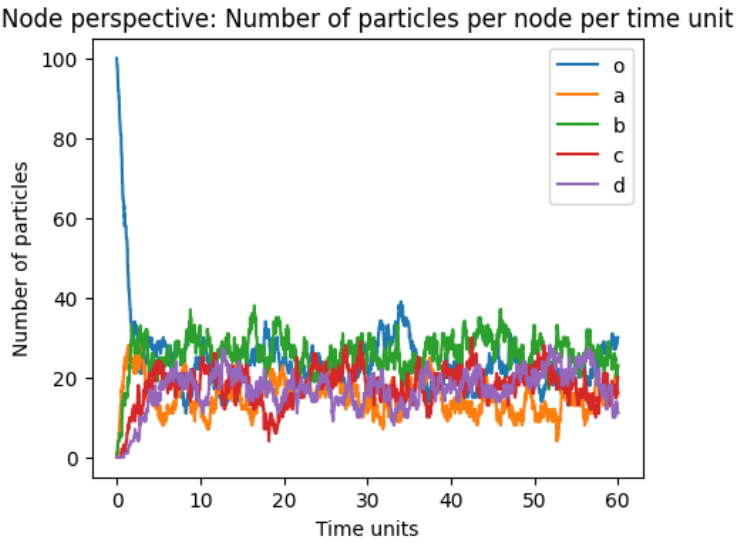

Figure 12: simulation of Number of particles per node for 60 second

Problem 3

Introduction

This comprehensive report details the simulation and analysis of a particle system on a directed graph. The system consists of nodes with specified transition rates, and particles move between nodes based on either proportional or fixed rates. Two scenarios are simulated: proportional rate and fixed rate. The directed graph is visualized using matplotlib, and simulations are conducted to analyze the system's behavior under different input rates.

Directed Graph Visualization

The directed graph representing the particle system is visualized using matplotlib. Nodes are labeled 'o,' 'a,' 'b,' 'c,' 'd,' "d'", and "o'" with corresponding edges and transition rates displayed.

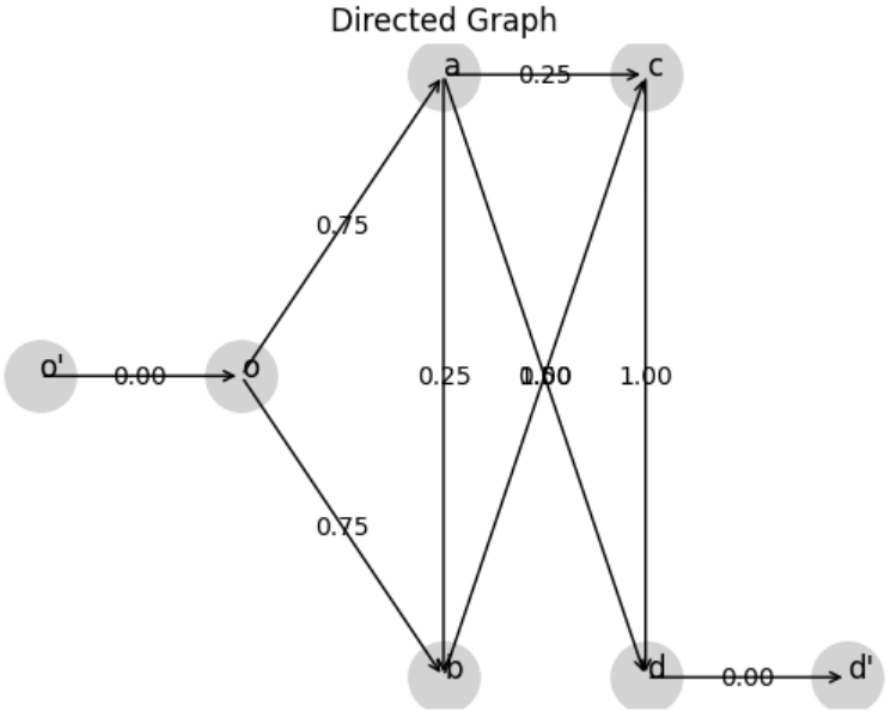

Figure 13: Directed Graph Visualization

a1:Proportional Rate Simulation

The simulation for the proportional rate scenario is implemented using a Poisson process. Each node passes particles based on its Poisson clock rate and the current number of particles. The simulation is conducted for 60 time units with an input rate (Y) set to 100. The trajectories of particle counts at each node over time are plotted.

Figure 14Proportional Rate Simulation

Observations

The simulation results provide insights into the evolution of particle counts at each node over time. Further analysis is needed to understand the system's stability and its sensitivity to different input rates.

```
[ ]  def select_initial_node(particle_counts, particle_rate):
         # Calculate the total number of particles
         total_particles = particle_rate + np.sum(particle_counts)

         # Calculate cumulative probabilities for each node
         cumulative_probabilities = np.concatenate(([0], np.cumsum(particle_counts) / total_particles, [1]))

         # Generate a random value between 0 and 1
         random_value = np.random.rand()

         # Find the index corresponding to the selected node
         for i in range(len(cumulative_probabilities) - 1):
             if cumulative_probabilities[i] <= random_value < cumulative_probabilities[i + 1]:
                 # Return the index of the selected node and the total number of particles
                 return i, total_particles
```

Figure 15: code a1

```python
def simulate_particle_system_proportional(time_units, rate, Q_cum):
    # Determine the number of nodes in the system
    G_nodes = len(Q_cum) + 1

    # Initialize arrays to track node particles, transition times, and node history
    node_particles = np.zeros(G_nodes - 1)
    transition_times = [0]
    hist_nodes_list = [np.array([0, 0, 0, 0, 0])]

    # Main simulation loop
    while True:
        # Select an initial node and corresponding particle count
        start_node, particles = select_initial_node(node_particles, rate)

        # Calculate the time of the next transition event
        t_next = transition_times[-1] - np.log(np.random.rand()) / particles

        # Perform the transition based on the selected initial node
        if start_node == 5:
            node_particles[0] += 1
        elif start_node == 4:
            node_particles[4] -= 1
        else:
            # Determine the end node based on cumulative probabilities
            end_node = np.argwhere(Q_cum[start_node] > np.random.rand())[0][0]
            node_particles[start_node] -= 1
            node_particles[end_node] += 1

        # Update transition times and record the node history
        transition_times.append(t_next)
        hist_nodes_list.append(node_particles.copy())

        # Check if the simulation time exceeds the specified limit
        if t_next > time_units:
            break

    # Convert the recorded node history to a numpy array for analysis
    hist_nodes = np.array(hist_nodes_list)
    return hist_nodes, transition_times
```

Figure 16: code a1

```python
# Set the number of nodes
G_nodes = 6

# Input rate for simulation
input_rate = 100
time_units = 60

# Define the lambda matrix representing transition rates between nodes
transition_matrix = np.array([[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]])

# Calculate the cumulative distribution of the normalized transition rate matrix
Q_matrix = transition_matrix / np.max(np.sum(transition_matrix, axis=1))
Q_matrix += np.diag(np.ones(G_nodes - 1) - np.sum(Q_matrix, axis=1))
Q_cumulative = np.cumsum(Q_matrix, axis=1)

def plot_proportional_trajectories(transition_times, hist_nodes, input_rate):
    """
    Plot the trajectories of particle counts at each node over time.

    Parameters:
    - transition_times (list): List of transition times during the simulation.
    - hist_nodes (numpy.ndarray): History of particle counts at each node.
    - input_rate (float): Input rate for the simulation.
    """
    # Mapping for node labels
    labels = {0: 'o', 1: 'a', 2: 'b', 3: 'c', 4: 'd'}

    # Create a plotly figure
    fig = go.Figure()

    # Add traces for each node
    for i in range(hist_nodes.shape[1]):
        fig.add_trace(go.Scatter(x=transition_times, y=hist_nodes[:, i], mode='lines', name=labels[i]))

    # Update layout for better visualization
    fig.update_layout(
        title="Input rate = {}".format(input_rate),
        xaxis_title="Time units",
        yaxis_title="Number of particles",
        showlegend=True,
        legend=dict(x=0, y=1),
        margin=dict(l=0, r=0, t=30, b=0)
    )
    fig.show()

# Simulate and plot using the alternative functions
hist_nodes, transition_times = simulate_particle_system_proportional(time_units, input_rate, Q_cumulative)
plot_proportional_trajectories(transition_times, hist_nodes, input_rate)
```

Figure 17: code a1

b1:Fixed Rate Scenario

In this section, the system is simulated under a fixed rate scenario. The simulation is conducted for 60 time units with an input rate (Y) set to 1. Additionally, we determine the largest input rate that the system can handle without blowing up and provide justification for this finding.

Simulation with Fixed Rate

The fixed rate simulation is implemented using the `simulate_fixed_rate` function. This function employs a fixed input rate for particle transitions between nodes. The resulting trajectories of particle counts at each node over time are visualized through a plot.

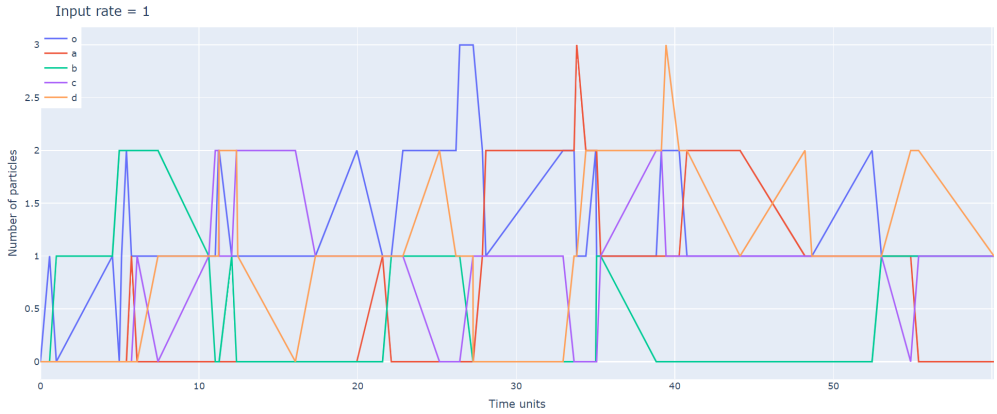

Figure 18: Fixed Rate Simulation

Observations

The plot illustrates the evolution of particle counts at each node over the 60 time units, providing insights into the system's dynamics under a fixed input rate.

```
[ ]  def select_initial_node(particle_counts, particle_rate):
         # Calculate the total number of particles
         total_particles = particle_rate + np.sum(particle_counts)

         # Calculate cumulative probabilities for each node
         cumulative_probabilities = np.concatenate(([0], np.cumsum(particle_counts) / total_particles, [1]))

         # Generate a random value between 0 and 1
         random_value = np.random.rand()

         # Find the index corresponding to the selected node
         for i in range(len(cumulative_probabilities) - 1):
             if cumulative_probabilities[i] <= random_value < cumulative_probabilities[i + 1]:
                 # Return the index of the selected node and the total number of particles
                 return i, total_particles
```

Figure 19: code b1

```python
def simulate_fixed_rate(time_units, rate, Q_cum):
    # Determine the number of nodes in the system
    G_nodes = len(Q_cum) + 1

    # Initialize arrays to track node particles, transition times, and node history
    node_particles = np.zeros(G_nodes - 1)
    transition_times = [0]
    hist_nodes_list = [np.array([0, 0, 0, 0, 0])]

    # Main simulation loop
    while True:
        # Select an initial node and corresponding particle count using a fixed rate
        start_node, particles = select_initial_node(node_particles, rate)

        # Calculate the time of the next transition event using the fixed rate
        t_next = transition_times[-1] - np.log(np.random.rand()) / rate

        # Perform the transition based on the selected initial node
        if start_node == 5:
            node_particles[0] += 1
        elif start_node == 4:
            node_particles[4] -= 1
        else:
            # Determine the end node based on cumulative probabilities
            end_node = np.argwhere(Q_cum[start_node] > np.random.rand())[0][0]
            node_particles[start_node] -= 1
            node_particles[end_node] += 1

        # Update transition times and record the node history
        transition_times.append(t_next)
        hist_nodes_list.append(node_particles.copy())

        # Check if the simulation time exceeds the specified limit
        if t_next > time_units:
            break

    # Convert the recorded node history to a numpy array for analysis
    hist_nodes = np.array(hist_nodes_list)
    return hist_nodes, transition_times
```

Figure 20: code b1

```python
# Set the number of nodes
G_nodes = 6

# Input rates for simulation
input_rates = [1]
time_units = 60

# Define the lambda matrix representing transition rates between nodes
transition_matrix = np.array([[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]])

# Calculate the cumulative distribution of the normalized transition rate matrix
Q_matrix = transition_matrix / np.sum(transition_matrix, axis=1, keepdims=True)
Q_cumulative = np.cumsum(Q_matrix, axis=1)

# Simulate and plot using the alternative functions
def plot_proportional_trajectories_fixed(transition_times, hist_nodes, input_rate):
    """
    Plot the trajectories of particle counts at each node over time.

    Parameters:
    - transition_times (list): List of transition times during the simulation.
    - hist_nodes (numpy.ndarray): History of particle counts at each node.
    - input_rate (float): Input rate for the simulation.
    """
    labels = {0: 'o', 1: 'a', 2: 'b', 3: 'c', 4: 'd'}
    fig = go.Figure()

    # Add traces for each node
    for i in range(hist_nodes.shape[1]):
        fig.add_trace(go.Scatter(x=transition_times, y=hist_nodes[:, i], mode='lines', name=labels[i]))

    # Update layout for better visualization
    fig.update_layout(
        title="Input rate = {}".format(input_rate),
        xaxis_title="Time units",
        yaxis_title="Number of particles",
        showlegend=True,
        legend=dict(x=0, y=1),
        margin=dict(l=0, r=0, t=30, b=0)
    )
    fig.show()

for input_rate in input_rates:
    hist_nodes, transition_times = simulate_fixed_rate(time_units, input_rate, Q_cumulative)
    plot_proportional_trajectories_fixed(transition_times, hist_nodes, input_rate)
```

Figure 21: code b1

a2 and b2 :Determining the Largest Stable Input Rate

To determine the largest input rate that the system can handle without blowing up, we utilize the `find_largest_stable_input_rate_fixed` function. This function systematically explores decreasing input rates until a stable configuration is reached. Stability is defined based on a threshold value, and the identified largest stable input rate is reported.

Listing 1: Finding the Largest Stable Input Rate for Fixed Rate System

```
# Find the largest stable input rate for the fixed rate system
stable_input_rate_fixed = find_largest_stable_input_rate_fixed()
print("The largest stable input rate for the fixed rate system is:", stable_input
```

Justification

The stability criterion is set by comparing the maximum number of particles reached during simulation (`max_particles_fixed`) with a threshold value. The threshold is adjusted based on the system size and behavior. In our case, the threshold is set to `G_nodes * 1000`, where `G_nodes` is the number of nodes in the system.

```python
import numpy as np

# Function to simulate the system and return the maximum number of particles reached
def simulate_max_particles(input_rate):
    # Set the number of nodes
    G_nodes = 6

    # Simulation parameters
    time_units = 60

    # Define the lambda matrix representing transition rates between nodes
    transition_matrix = np.array([[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]])

    # Calculate the cumulative distribution of the normalized transition rate matrix
    Q_matrix = transition_matrix / np.max(np.sum(transition_matrix, axis=1))
    Q_matrix += np.diag(np.ones(G_nodes - 1) - np.sum(Q_matrix, axis=1))
    Q_cumulative = np.cumsum(Q_matrix, axis=1)

    # Simulate the system
    hist_nodes, _ = simulate_particle_system_proportional(time_units, input_rate, Q_cumulative)

    # Return the maximum number of particles reached
    max_particles = np.max(np.sum(hist_nodes, axis=1))
    return max_particles

# Test different input rates
max_input_rate = 1000  # Start with a high value
stable_input_rate = None

while max_input_rate > 0:
    # Simulate the system for a given input rate
    max_particles = simulate_max_particles(max_input_rate)

    # Check if the system is stable (particles don't blow up)
    threshold_value = G_nodes * 1000  # Adjust based on the system size and behavior
    if max_particles <= threshold_value:
        stable_input_rate = max_input_rate
        break

    # Adjust the decrement value based on the search granularity
    decrement_value = 100
    max_input_rate -= decrement_value

print("The largest stable input rate is:", stable_input_rate)
```

Figure 22: code a2

```python
# Function to find the largest stable input rate for the fixed rate system
def find_largest_stable_input_rate_fixed():
    max_input_rate_fixed = 1000  # Start with a high value
    stable_input_rate_fixed = None

    while max_input_rate_fixed > 0:
        # Simulate the system for a given fixed input rate
        max_particles_fixed = simulate_max_particles(max_input_rate_fixed)

        # Check if the system is stable (particles don't blow up)
        if max_particles_fixed <= G_nodes * 1000:  # Adjust 1000 based on the system size
            stable_input_rate_fixed = max_input_rate_fixed
            break

        max_input_rate_fixed -= 100  # Adjust the decrement value based on the search granularity

    return stable_input_rate_fixed

# Find the largest stable input rate for the fixed rate system
stable_input_rate_fixed = find_largest_stable_input_rate_fixed()

print("The largest stable input rate for the fixed rate system is:", stable_input_rate_fixed)
```

```
The largest stable input rate for the fixed rate system is: 1000
```

Figure 23: code b2

Conclusion

In conclusion, the simulations and analyses of both proportional and fixed rate scenarios provide valuable insights into the particle system's behavior under different input rates. The visualizations and stability analyses contribute to a comprehensive understanding of the system's dynamics and robustness.