

# **Νευρο-ασαφής Έλεγχος**

9ο Εξάμηνο 2023 – 2024

Assignment 4 – Solutions

Ζαρίφης Στέλιος – el20435

Email: [el20435@mail.ntua.gr](mailto:el20435@mail.ntua.gr)

# Contents

<b>1</b>	<b>Markov Chain</b>	<b>3</b>
1.1	Αναδρομικές Κλάσεις και Περιοδικότητα . . . . .	3
1.2	Εξέλιξη στη Μαρκοβιανή Αλυσίδα . . . . .	9
1.3	Ποσοστό Χρόνου για Κάθε Κατάσταση . . . . .	11
<b>2</b>	<b>Robot and Maze</b>	<b>13</b>
2.1	Μέσος Χρόνος Απορρόφησης . . . . .	13
2.2	Προσομοίωση Στοχαστικής δυναμικής . . . . .	13
<b>3</b>	<b>Parking Problem</b>	<b>16</b>
<b>4</b>	<b>Controlled Random Walk</b>	<b>18</b>

# 1 Markov Chain

## 1.1 Αναδρομικές Κλάσεις και Περιοδικότητα

Αναδρομική κλάση σε Μαρκοβιανή αλυσίδα είναι το σύνολο των states που επικοινωνούν όλα μεταξύ τους. Η περίοδος ενός state είναι ο μέγιστος κοινός διαιρέτης του set όλων των δυνατών βημάτων επιστροφής στο ίδιο το state. Αν είναι 1, τότε το state είναι aperiodic. Ανάλογα με τις τιμές των  $\alpha, \beta, \gamma$ , μηδενίζονται ορισμένες ακμές, οπότε σχηματίζονται διάφορες κλάσεις. Εξετάζουμε όλες τις δυνατές διατάξεις για  $(\alpha, \beta, \gamma) \in [0, 1] \times [0, 1] \times [0, 1]$  (αφού είναι πιθανότητες):

- **Case 1:**  $\alpha = 0, \beta = 0, \gamma = 0$ :

Αναδρομική κλάση:  $R = 1, 2, 3, 4$

Περιοδικότητα:

- $S_i = i, i = 1, 2, 3, 4$
- $r_{i,j}(n) = 1, i = 1, 2, 3, 4, j = i \bmod 4 + 1$
- period:  $d = 4$

Transient state: 5

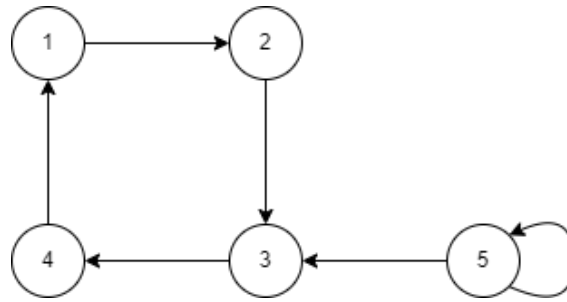


Figure 1:  $\alpha = 0, \beta = 0, \gamma = 0$

- **Case 2:**  $\alpha = 0, \beta = 0, \gamma \in (0, 1)$

Αναδρομική κλάση:  $R = 1, 2, 3, 4, 5$

Περιοδικότητα: periodic

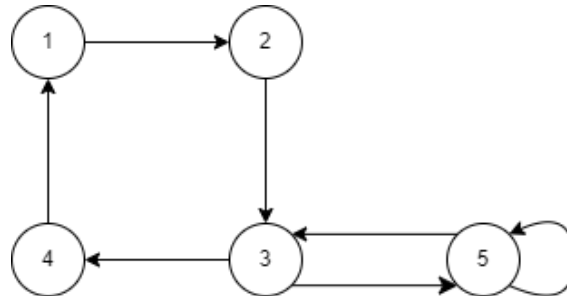


Figure 2:  $\alpha = 0, \beta = 0, \gamma \in (0, 1)$

- **Case 3:**  $\alpha = 0, \beta = 0, \gamma = 1$   
 Αναδρομική κλάση:  $R = 3, 5$   
 Περιοδικότητα: aperiodic  
 Transient states: 1, 2, 4

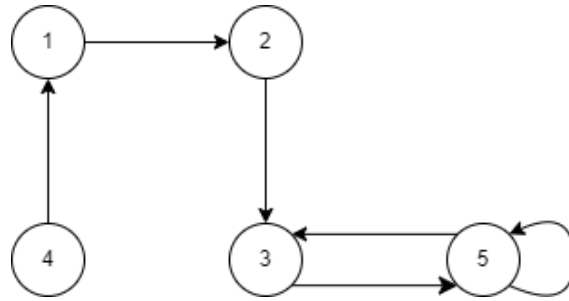


Figure 3:  $\alpha = 0, \beta = 0, \gamma = 1$

- **Case 4:**  $\alpha = 0, \beta = 1, \gamma = 0$   
 Αναδρομική κλάση:  $R = 1, 4$   
 Περιοδικότητα:  
 -  $S_1 = 1, S_2 = 4$   
 -  $r_{i,j}(n) = 1, i = 1, 2, j = i \bmod 2 + 1$   
 - period:  $d = 2$

Transient states: 2, 3, 5

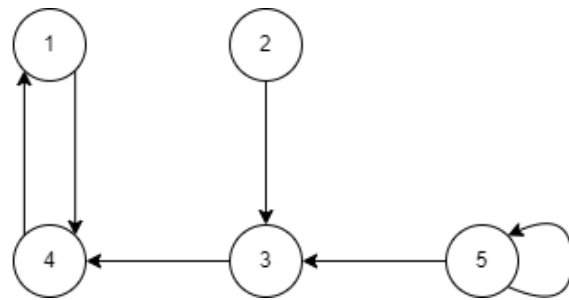


Figure 4:  $\alpha = 0, \beta = 1, \gamma = 0$

- **Case 5:**  $\alpha = 0, \beta = 1, \gamma \in (0, 1)$

Αναδρομική κλάση:  $R = 1, 4$

Περιοδικότητα:

- $S_1 = 1, S_2 = 4$
- $r_{i,j}(n) = 1, i = 1, 2, j = i \bmod 2 + 1$
- period:  $d = 2$

Transient states: 2, 3, 5

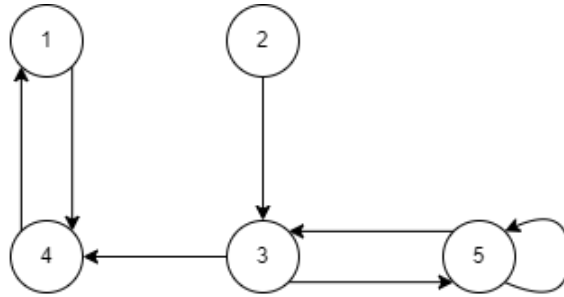


Figure 5:  $\alpha = 0, \beta = 1, \gamma \in (0, 1)$

- **Case 6:**  $\alpha = 0, \beta = 1, \gamma = 1$

Αναδρομική κλάση:  $R_1 = 1, 4$

Περιοδικότητα:

- $S_1 = 1, S_2 = 4$
- $r_{i,j}(n) = 1, i = 1, 2, j = i \bmod 2 + 1$
- period:  $d = 2$

Αναδρομική κλάση:  $R_2 = 3, 5$

Περιοδικότητα: aperiodic Transient state: 2

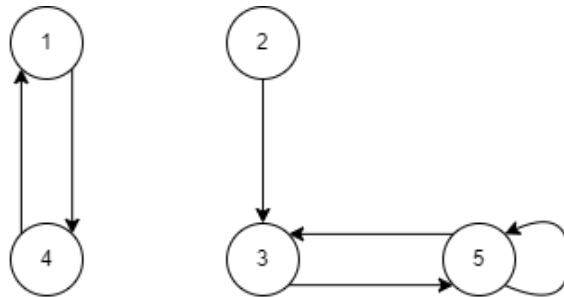


Figure 6:  $\alpha = 0, \beta = 1, \gamma = 1$

- **Case 7:**  $\alpha = 1, \beta = 0, \gamma = 0$

Αναδρομική κλάση:  $R = 1, 3, 4$

Περιοδικότητα:

- $S_1 = 1, S_2 = 3, S_3 = 4$
- $r_{i,j}(n) = 1, i = 1, 2, j = i \bmod 3 + 1$
- period:  $d = 3$

Transient states: 2, 5

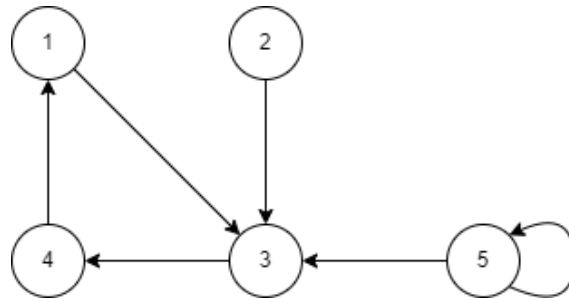


Figure 7:  $\alpha = 1, \beta = 0, \gamma = 0$

- **Case 8:**  $\alpha = 1, \beta = 0, \gamma \in (0, 1)$

Αναδρομική κλάση:  $R = 1, 3, 4, 5$

Περιοδικότητα: aperiodic

Transient state: 2

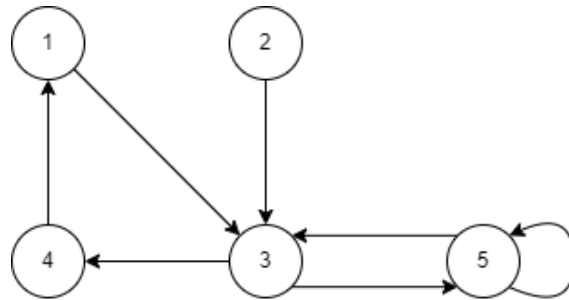


Figure 8:  $\alpha = 1, \beta = 0, \gamma \in (0, 1)$

- **Case 9:**  $\alpha = 1, \beta = 0, \gamma = 1$   
 Αναδρομική κλάση:  $R = 3, 5$   
 Περιοδικότητα: aperiodic  
 Transient states: 1, 2, 4

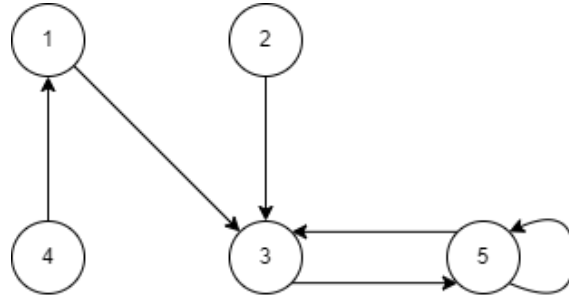


Figure 9:  $\alpha = 1, \beta = 0, \gamma = 1$

- **Case 10:**  $\alpha \neq 0, \beta \neq 0, \gamma = 0, a + b = 1$   
 Αναδρομική κλάση:  $R = 1, 3, 4$   
 Περιοδικότητα: aperiodic  
 Transient states: 2, 5

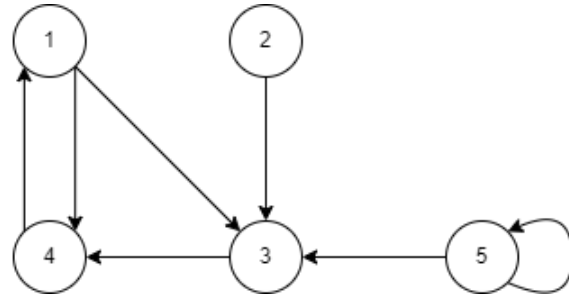


Figure 10:  $\alpha \neq 0, \beta \neq 0, \gamma = 0, a + b = 1$

- **Case 11:**  $\alpha \neq 0, \beta \neq 0, \gamma \in (0, 1), a + b = 1$   
 Αναδρομική κλάση:  $R = 1, 3, 4, 5$   
 Περιοδικότητα: aperiodic  
 Transient state: 2

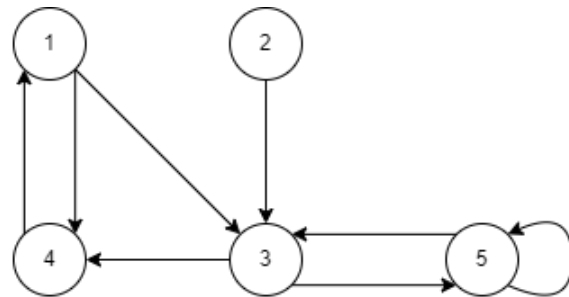


Figure 11:  $\alpha \neq 0, \beta \neq 0, \gamma \in (0, 1), a + b = 1$

- **Case 12:**  $\alpha \neq 0, \beta \neq 0, \gamma = 1, a + b = 1$   
 Αναδρομική κλάση:  $R = 3, 5$   
 Περιοδικότητα: aperiodic Transient states: 1, 2, 4

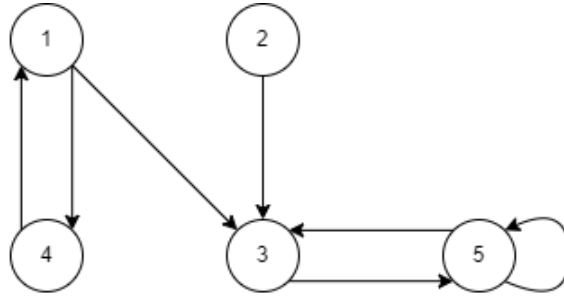


Figure 12:  $\alpha \neq 0, \beta \neq 0, \gamma = 1, a + b = 1$

- **Case 13:**  $\alpha \neq 0, \beta \neq 0, \gamma = 0, a + b < 1$   
 Αναδρομική κλάση:  $R = 1, 2, 3, 4$   
 Περιοδικότητα: aperiodic Transient state: 5

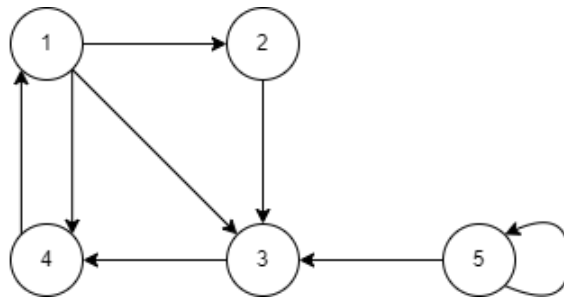


Figure 13:  $\alpha \neq 0, \beta \neq 0, \gamma = 0, a + b < 1$

- **Case 14:**  $\alpha \neq 0, \beta \neq 0, \gamma \in (0, 1), a + b < 1$   
 Αναδρομική κλάση:  $R = 1, 2, 3, 4, 5$   
 Περιοδικότητα: aperiodic

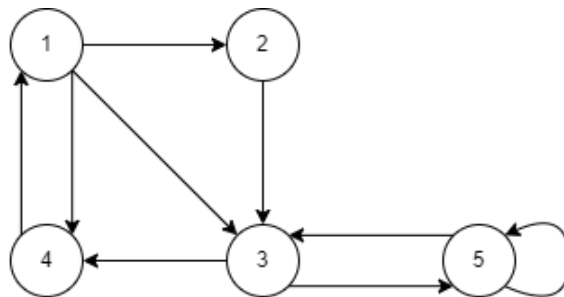


Figure 14:  $\alpha \neq 0, \beta \neq 0, \gamma \in (0, 1), a + b < 1$

- **Case 15:**  $\alpha \neq 0, \beta \neq 0, \gamma = 1, a + b < 1$   
 Αναδρομική κλάση:  $R = 3, 5$   
 Περιοδικότητα: aperiodic Transient states: 1, 2, 4

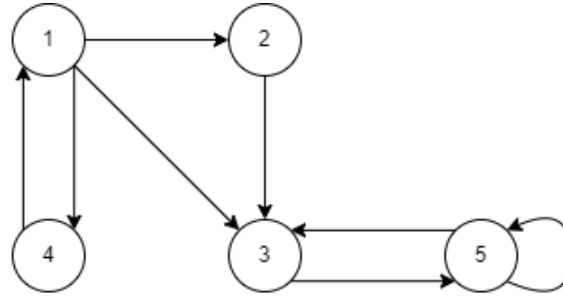


Figure 15:  $\alpha \neq 0, \beta \neq 0, \gamma = 1, a + b < 1$

## 1.2 Εξέλιξη στη Μαρκοβιανή Αλυσίδα

Χρησιμοποιούμε python για να εξετάσουμε πώς εξελίσσεται το σύστημα. Ορίζουμε τη συνάρτηση `markov_chain_probability` η οποία για εκτελεί συγκεκριμένο πλήθος βημάτων, ανανεώνοντας κάθε φορά την κατανομή πιθανότητας στα states. Για τα ζητούμενα  $\alpha, \beta, \gamma$ , υπολογίζουμε τις κατανομές για όλες τις αρχικές συνθήκες, στα ζητούμενα βήματα:

```

1 import numpy as np
2
3 def markov_chain_probability(transition_matrix, initial_distribution,
4   n_steps):
5     current_distribution = initial_distribution
6     for _ in range(n_steps):
7         current_distribution = np.dot(current_distribution,
8   transition_matrix)
9
10    return current_distribution
11
12 for initial_distribution in [[1, 0, 0, 0, 0],
13   [0, 1, 0, 0, 0],
14   [0, 0, 1, 0, 0],
15   [0, 0, 0, 1, 0],
16   [0, 0, 0, 0, 1]]:
17     for a, b, c in [(0, 0.1, 0), (0.1, 0.1, 0.1)]:
18         print(f"Starting from state {np.argmax(initial_distribution) + 1}
19   with a = {a}, b = {b}, c = {c}:")
20         for n_steps in [1000, 1001, 1002, 1003]:
21             transition_matrix = np.array([
22                 [0, 1 - a - b, a, b, 0],
23                 [0, 0, 1, 0, 0],
24                 [0, 0, 0, 1 - c, c],
25                 [1, 0, 0, 0, 0],
26                 [0, 0, 0.5, 0, 0.5]
27             ])

```

```

25         # Calculate the probability distribution after n_steps
26         result_distribution = markov_chain_probability(
transition_matrix, initial_distribution, n_steps)
27         # Print the result
28         print(f"Probability distribution after {n_steps} time steps:
{result_distribution}")
29

```

Το αποτέλεσμα φαίνεται εδώ. Μπορούμε να παρατηρήσουμε περιοδικότητα, για παράδειγμα στο πρώτο πείραμα ή ότι το σύστημα έφτασε στο steady state, για παράδειγμα στο δεύτερο πείραμα:

```

1  =====
2  Starting from state 1 with a = 0, b = 0.1, c = 0:
3  Distribution after 1000 time steps: [0.526 0.      0.474 0.      0.    ]
4  Distribution after 1001 time steps: [0.      0.474 0.      0.526 0.    ]
5  Distribution after 1002 time steps: [0.526 0.      0.474 0.      0.    ]
6  Distribution after 1003 time steps: [0.      0.474 0.      0.526 0.    ]
7  -----
8  Starting from state 1 with a = 0.1, b = 0.1, c = 0.1:
9  Distribution after 1000 time steps: [0.25 0.2  0.25 0.25 0.05]
10 Distribution after 1001 time steps: [0.25 0.2  0.25 0.25 0.05]
11 Distribution after 1002 time steps: [0.25 0.2  0.25 0.25 0.05]
12 Distribution after 1003 time steps: [0.25 0.2  0.25 0.25 0.05]
13 =====
14 Starting from state 2 with a = 0, b = 0.1, c = 0:
15 Distribution after 1000 time steps: [0.      0.474 0.      0.526 0.    ]
16 Distribution after 1001 time steps: [0.526 0.      0.474 0.      0.    ]
17 Distribution after 1002 time steps: [0.      0.474 0.      0.526 0.    ]
18 Distribution after 1003 time steps: [0.526 0.      0.474 0.      0.    ]
19 -----
20 Starting from state 2 with a = 0.1, b = 0.1, c = 0.1:
21 Distribution after 1000 time steps: [0.25 0.2  0.25 0.25 0.05]
22 Distribution after 1001 time steps: [0.25 0.2  0.25 0.25 0.05]
23 Distribution after 1002 time steps: [0.25 0.2  0.25 0.25 0.05]
24 Distribution after 1003 time steps: [0.25 0.2  0.25 0.25 0.05]
25 =====
26 Starting from state 3 with a = 0, b = 0.1, c = 0:
27 Distribution after 1000 time steps: [0.526 0.      0.474 0.      0.    ]
28 Distribution after 1001 time steps: [0.      0.474 0.      0.526 0.    ]
29 Distribution after 1002 time steps: [0.526 0.      0.474 0.      0.    ]
30 Distribution after 1003 time steps: [0.      0.474 0.      0.526 0.    ]
31 -----
32 Starting from state 3 with a = 0.1, b = 0.1, c = 0.1:
33 Distribution after 1000 time steps: [0.25 0.2  0.25 0.25 0.05]
34 Distribution after 1001 time steps: [0.25 0.2  0.25 0.25 0.05]
35 Distribution after 1002 time steps: [0.25 0.2  0.25 0.25 0.05]
36 Distribution after 1003 time steps: [0.25 0.2  0.25 0.25 0.05]
37 =====
38 Starting from state 4 with a = 0, b = 0.1, c = 0:
39 Distribution after 1000 time steps: [0.      0.474 0.      0.526 0.    ]

```

```

40 Distribution after 1001 time steps: [0.526 0.      0.474 0.      0.    ]
41 Distribution after 1002 time steps: [0.      0.474 0.      0.526 0.    ]
42 Distribution after 1003 time steps: [0.526 0.      0.474 0.      0.    ]
43 -----
44 Starting from state 4 with a = 0.1, b = 0.1, c = 0.1:
45 Distribution after 1000 time steps: [0.25 0.2  0.25 0.25 0.05]
46 Distribution after 1001 time steps: [0.25 0.2  0.25 0.25 0.05]
47 Distribution after 1002 time steps: [0.25 0.2  0.25 0.25 0.05]
48 Distribution after 1003 time steps: [0.25 0.2  0.25 0.25 0.05]
49 =====
50 Starting from state 5 with a = 0, b = 0.1, c = 0:
51 Distribution after 1000 time steps: [1.754e-001 3.158e-001 1.579e-001
    3.509e-001 9.333e-302]
52 Distribution after 1001 time steps: [3.509e-001 1.579e-001 3.158e-001
    1.754e-001 4.666e-302]
53 Distribution after 1002 time steps: [1.754e-001 3.158e-001 1.579e-001
    3.509e-001 2.333e-302]
54 Distribution after 1003 time steps: [3.509e-001 1.579e-001 3.158e-001
    1.754e-001 1.167e-302]
55 -----
56 Starting from state 5 with a = 0.1, b = 0.1, c = 0.1:
57 Distribution after 1000 time steps: [0.25 0.2  0.25 0.25 0.05]
58 Distribution after 1001 time steps: [0.25 0.2  0.25 0.25 0.05]
59 Distribution after 1002 time steps: [0.25 0.2  0.25 0.25 0.05]
60 Distribution after 1003 time steps: [0.25 0.2  0.25 0.25 0.05]

```

### 1.3 Ποσοστό Χρόνου για Κάθε Κατάσταση

Πάλι με Python, εκτελούμε μια προσομοίωση 10000 βημάτων για αρχική κατάσταση 1 και  $a = 0.1, \beta = 0.1, \gamma = 0.1$ . Παρατηρούμε ότι οι τιμές των ποσοστών συγκλίνουν στα κανονικοποιημένα μέτρα των αντίστοιχων ιδιοδιανυσμάτων!

```

1 import numpy as np
2 from scipy.sparse.linalg import eig
3
4 def simulate_markov_chain(transition_matrix, initial_state, n_steps):
5     """Simulates a Markov chain for a given number of steps."""
6     current_state = initial_state
7     state_counts = np.zeros(len(initial_state))
8     for _ in range(n_steps):
9         current_state = np.dot(current_state, transition_matrix)
10        state_counts += current_state
11    return state_counts / n_steps
12
13 # Define the transition matrix
14 a = b = c = 0.1
15 transition_matrix = np.array([
16     [0, 1 - a - b, a, b, 0],
17     [0, 0, 1, 0, 0],

```

```

18     [0, 0, 0, 1 - c, c],
19     [1, 0, 0, 0, 0],
20     [0, 0, 0.5, 0, 0.5]
21 ])
22
23 # Define the initial state
24 initial_state = np.array([1, 0, 0, 0, 0]) # Starting from state 1
25
26 # Number of time steps
27 n_steps = 10000
28
29 # Simulate the Markov chain
30 state_percentages = simulate_markov_chain(transition_matrix,
31     initial_state, n_steps)
32
33 # Print the result
34 print(f"Percentage of time in each state over the first {n_steps} time
35     steps:")
36 print(state_percentages)
37
38 # Print the raw eigenvalue and eigenvector
39 print("Eigenvectors:")
40 print(np.abs(vec.T))
41
42 # Print the normalized eigenvector representing the stationary
43     distribution
44 print("Normalized stationary distribution:")
45 print(normalized_vec.T)

```

Το αποτέλεσμα φαίνεται εδώ:

Percentage of time in each state over the first 10000 time steps:  
 [0.24996792 0.20003433 0.25001236 0.24999292 0.04999247]

Eigenvectors:  
 [[0.52128604 0.41702883 0.52128604 0.52128604 0.10425721]]

Normalized Eigenvectors:  
 [[0.25 0.2 0.25 0.25 0.05]]

## 2 Robot and Maze

### 2.1 Μέσος Χρόνος Απορρόφησης

Μπορούμε να δούμε το λαβύρινθο ως μια μαρκοβιανή αλυσίδα η οποία έχει μεταβάσεις από κάθε state προς τα γειτονικά states (γείτονες ενός state ορίζουμε τα states στα οποία επιτρέπεται να μεταβεί το ρομπότ) με πιθανότητες  $1/\#neighbors$ . Για τα 14 states, προκύπτει ο πίνακας μεταβάσεων  $P$ , όπου  $P_{ij}$  είναι η πιθανότητα μετάβασης στο state  $j$  από το state  $i$ :

$$P = \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/3 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/3 & 0 & 1/3 & 0 & 1/3 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 1/2 & 0 \end{pmatrix}$$

Γνωρίζουμε ότι ο αναμενόμενος χρόνος απορρόφησης για το state  $i$  είναι

$$\begin{cases} \mu_i = 0, \text{ for all recurrent states } i \\ \mu_i = 1 + \sum_{j=1}^m P_{ij}\mu_j, \text{ for all transient states } i \end{cases}$$

Όλα τα  $\mu_i$  συγκροτούν το διάνυσμα  $\vec{\mu}$  και οι άνω εξισώσεις παράγουν το εξής σύστημα:

$$\mu_i = 1 + \bar{P}_i \cdot \vec{\mu} \Rightarrow \vec{\mu} = \mathbf{1} + \bar{P} \vec{\mu} \Rightarrow (\mathbf{I} - \bar{P}) \vec{\mu} = \mathbf{1}$$

Όπου ο πίνακας  $\bar{P}$  προκύπτει από τον  $P$  αφού αφαιρέσουμε τη γραμμή και στήλη που αντιστοιχεί στην απορροφητική κατάσταση. Λύνοντας το σύστημα, προκύπτει ότι ο μέσος χρόνος απορρόφησης είναι  $\sim 37.8$

### 2.2 Προσομοίωση Στοχαστικής δυναμικής

Με το παρακάτω python script προσομοιώνουμε 100000 πειράματα για την τροχιά του ρομπότ. Ορίζουμε τα όρια του χάρτη και τα 2 εμπόδια με τη συνάρτηση `is_valid_state`. Οι δυνατές κινήσεις του ρομπότ σε κάθε state δίνονται από τη συνάρτηση `get_neighbors`. Ακόμα, η συνάρτηση `simulate_robot` προσομοιώνει την κίνηση του ρομπότ έως ότου φτάσει στο goal state και επιστρέφει το πλήθος βημάτων που έκανε. Η συνάρτηση `main` εκτελεί 1000 πειράματα και επιστρέφει το μέσο όρο των χρόνων απορρόφησης και το ιστόγραμμα των χρόνων αυτών.

```
1 import random
2 import matplotlib.pyplot as plt
3
4 def is_valid_state(state):
5     return 0 <= state[0] < 4 and 0 <= state[1] < 4 and state not in [(1,
6     1), (2, 2)]
```

```

6
7 def get_neighbors(state):
8     neighbors = [(state[0] - 1, state[1]), (state[0] + 1, state[1]),
9                 (state[0], state[1] - 1), (state[0], state[1] + 1)]
10    return [neighbor for neighbor in neighbors if is_valid_state(neighbor
11    )]
12
13 def simulate_robot():
14     current_state = (0, 0)
15     goal_state = (2, 3)
16     steps = 0
17
18     while current_state != goal_state:
19         neighbors = get_neighbors(current_state)
20         current_state = random.choice(neighbors)
21         steps += 1
22
23     return steps
24
25 def main(num_simulations):
26     total_steps = 0
27     arrival_times = []
28
29     for _ in range(num_simulations):
30         steps = simulate_robot()
31         total_steps += steps
32         arrival_times.append(steps)
33
34     average_steps = total_steps / num_simulations
35     print(f"Average number of steps to reach the goal in {num_simulations
36     } simulations: {average_steps:.2f}")
37
38     # Plotting the histogram
39     plt.hist(arrival_times, bins=range(min(arrival_times), max(
40     arrival_times) + 1), align='left', rwidth=0.8)
41     plt.title('Histogram of Arrival Times to Reach the Goal')
42     plt.xlabel('Number of Steps')
43     plt.ylabel('Frequency')
44     plt.show()
45
46 if __name__ == "__main__":
47     num_simulations = 100000
48     main(num_simulations)

```

Το αποτέλεσμα φαίνεται εδώ. Παρατηρούμε ότι ο αναμενόμενος χρόνος απορρόφησης που υπολογίσαμε θεωρητικά ταιριάζει με αυτόν που βρίσκουμε πειραματικά!

Average number of steps to reach the goal in 100000 simulations: 37.93

Παραθέτουμε και το ιστόγραμμα των χρόνων απορρόφησης που προέκυψε από τα πειράματα:

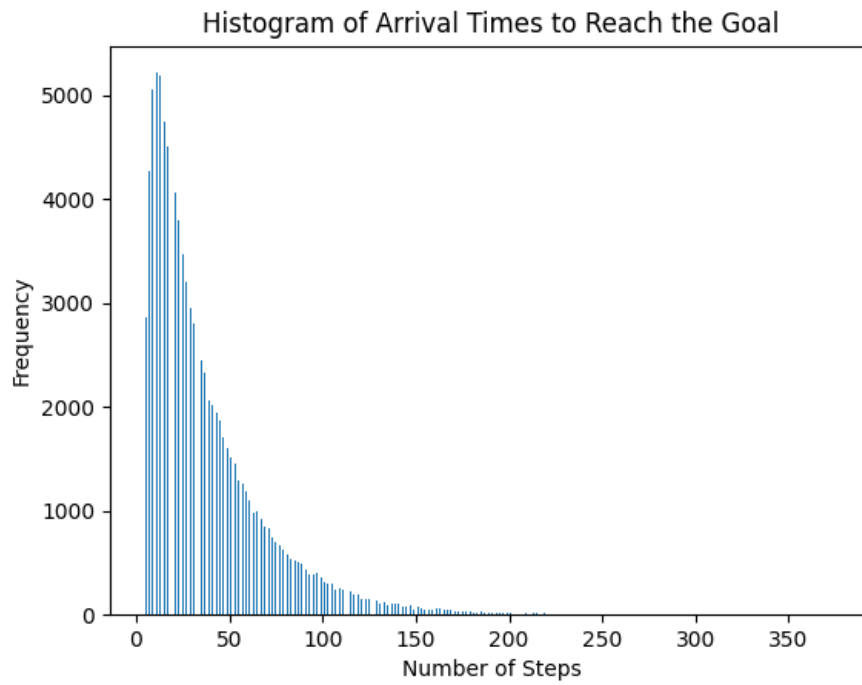


Figure 16: Absorption Times Histogram

### 3 Parking Problem

Ακολουθώντας την πορεία σκέψης στο textbook [1] (pp. 41-44), θα αντιμετωπίσουμε το πρόβλημα με Δυναμικό Προγραμματισμό. Έχουμε  $N$  στάδια, ένα για κάθε θέση parking και ένα τελικό state  $t$  που συμβολίζει το γεγονός ότι σταθμεύσαμε. Κάθε στάδιο  $k = 0, \dots, N - 1$  έχει δύο καταστάσεις  $(k, F)$  και  $(k, \bar{F})$ , που αντιστοιχούν στο αν η θέση  $k$  είναι ελεύθερη ή κατειλημμένη. Η απόφαση στην κατάσταση  $(k, F)$  είναι να σταθμεύσουμε ή να συνεχίσουμε.

Για τον αλγόριθμο Δυναμικού Προγραμματισμού (DP), συμβολίζουμε:

- $J_k^*(F)$  : Το βέλτιστο κόστος μετάβασης κατά την άφιξη σε θέση  $k$  που είναι ελεύθερη ( $F$ : Free).
- $J_k^*(\bar{F})$  : Το βέλτιστο κόστος μετάβασης κατά την άφιξη σε θέση  $k$  που είναι κατειλημμένη ( $\bar{F}$ : Not Free).
- $J_N^*(t) = C$  : Το κόστος μετάβασης κατά την άφιξη στο γκαράζ.
- $J_k^*(t) = 0$  : Το τελικό κόστος μετάβασης.

Ο αλγόριθμος DP για  $k = 0, \dots, N - 1$  έχει τη μορφή

$$J_k^*(F) = \begin{cases} \min [c(k), p(k+1)J_{k+1}^*(F) + (1 - p(k+1))J_{k+1}^*(\bar{F})] & \text{if } k < N - 1, \\ \min [c(N - 1), C] & \text{if } k = N - 1, \end{cases}$$
$$J_k^*(\bar{F}) = \begin{cases} p(k+1)J_{k+1}^*(F) + (1 - p(k+1))J_{k+1}^*(\bar{F}) & \text{if } k < N - 1, \\ C & \text{if } k = N - 1, \end{cases}$$

Ισοδύναμα, αφού η συνιστώσα ( $F$  ή  $\bar{F}$ ) είναι μη ελέγξιμη, ορίζουμε το βέλτιστο αναμενόμενο κόστος μετά την άφιξη στη θέση  $k$ , πριν από την επαλήθευση αν είναι ελεύθερη ή κατειλημμένη.

$$\hat{J}_k = p(k)J_k^*(F) + (1 - p(k))J_k^*(\bar{F}), \quad k = 0, \dots, N - 1$$

Από τον προηγούμενο αλγόριθμο DP, έχουμε:

$$\hat{J}_{N-1} = p(N-1) \min [c(N-1), C] + (1 - p(N-1))C$$
$$\hat{J}_k = p(k) \min [c(k), \hat{J}_{k+1}] + (1 - p(k))\hat{J}_{k+1}, \quad k = 0, \dots, N - 2$$

Από αυτό, μπορούμε να λάβουμε τη βέλτιστη πολιτική στάθμευσης, η οποία είναι η στάθμευση στη θέση  $k = 0, \dots, N - 1$  εάν είναι ελεύθερη και  $c(k) \leq \hat{J}_{k+1}$ . Με το κάτωθι python script υπολογίζουμε το  $\hat{J}_k$  για την περίπτωση  $p(k) = 0.05, c(k) = N - k, C = 100, N = 200$ . Βλέπουμε πως η βέλτιστη πολιτική είναι να ταξιδέψει κανείς ως τη θέση 165 (ξεκινώντας από τη 200-οστή που είναι η πιο μακρινή από τον προορισμό) και στη συνέχεια να σταθμεύσει στην πρώτη διαθέσιμη θέση.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Parameters
5 N = 200
6 C = 100
7 p = 0.05
8 c = np.arange(N, 0, -1)
9
```

```

10 # Initialize J and J_hat
11 J_F = np.zeros(N)
12 J_notF = np.zeros(N)
13 J_hat = np.zeros(N)
14
15 # Terminal conditions
16 J_F[-1] = min(c[-1], C)
17 J_notF[-1] = C
18 J_hat[-1] = p * J_F[-1] + (1 - p) * J_notF[-1]
19
20 # DP algorithm
21 for k in range(N-2, -1, -1):
22     J_F[k] = min(c[k], J_hat[k+1])
23     J_notF[k] = J_hat[k+1]
24     J_hat[k] = p * J_F[k] + (1 - p) * J_notF[k]
25
26 # Plotting
27 plt.figure(figsize=(8, 6))
28 plt.plot(J_hat[::-1], label='$\hat{J}_k$')
29 plt.xlabel('Parking Space')
30 plt.ylabel('Optimal Expected Cost-to-Go')
31 plt.title('Optimal Expected Cost-to-Go vs. Parking Space')
32 plt.legend()
33 plt.grid(True)
34 plt.show()
35

```

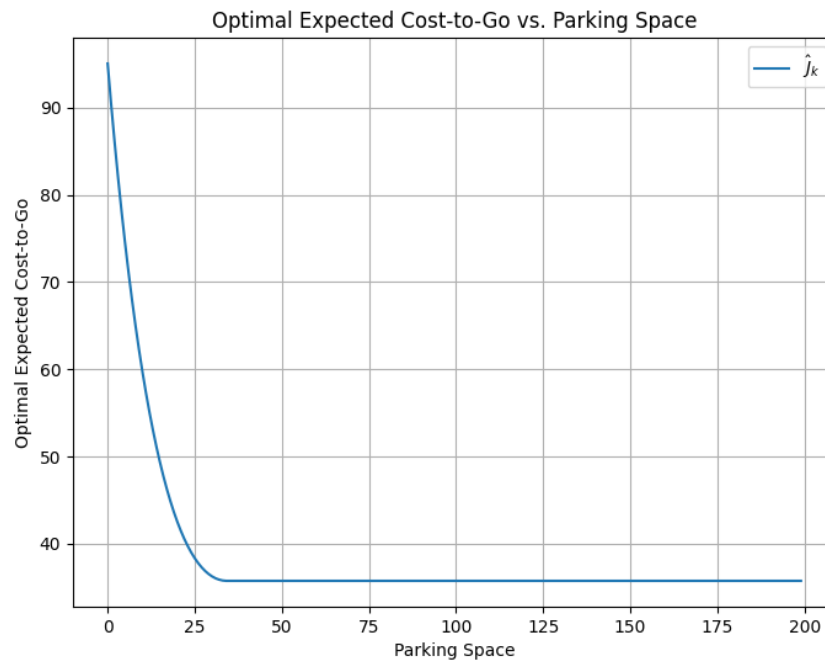


Figure 17: Cost of moving to the next position, at each position

## 4 Controlled Random Walk

Έστω μια controlled Markov chain με καταστάσεις  $\{1, \dots, 10\}$  και έλεγχο  $\{+1, -1\}$ . Οι πιθανότητες μετάβασης ορίζονται ως εξής:

- Για  $u = +1$ , η κατάσταση μετακινείται κατά μία θέση προς τα δεξιά με πιθανότητα 0.5 και παραμένει η ίδια με πιθανότητα 0.5.
- Για  $u = -1$ , η κατάσταση μετακινείται κατά μία θέση προς τα αριστερά με πιθανότητα 0.5 και παραμένει η ίδια με πιθανότητα 0.5.

Η συνάρτηση κόστους  $g(x)$  δίνεται από τον ακόλουθο πίνακα:

$x$	1	2	3	4	5	6	7	8	9	10
$g(x)$	1	2	3	4	5	4	2	0	1	2

Θέλουμε η πολιτική να ελαχιστοποιεί το κόστος σε άπειρο ορίζοντα, για διάφορες τιμές του συντελεστή  $a$ :

$$J = \sum_{k=0}^{\infty} a^k g(x_k)$$

Το πρόβλημα επιλύεται με τη χρήση του αλγορίθμου Value Iteration, ο οποίος ενημερώνει επαναληπτικά τη συνάρτηση αξίας  $V$  μέχρι να συγκλίνει. Η βέλτιστη πολιτική υπολογίζεται στη συνέχεια επιλέγοντας την ενέργεια που μεγιστοποιεί την αναμενόμενη ανταμοιβή σε κάθε κατάσταση.

**Value Iteration in Reinforcement Learning** Ο αλγόριθμος Value Iteration (VI) χρησιμοποιείται για την επίλυση RL προβλημάτων όπου έχουμε πλήρη γνώση όλων των χαρακτηριστικών της Markov Decision Process (MDP). Επαναληπτικά βελτιώνει την εκτίμηση της αξίας (value) να βρισκόμαστε στο εκάστοτε state, λαμβάνοντας υπόψη το άμεσο reward και τα μελλοντικά αναμενόμενα rewards με τη λήψη διαφόρων αποφάσεων. Κάποια στιγμή θα υπάρξει σύγκλιση με αποτέλεσμα μια optimal policy που περιλαμβάνει απεικονίσεις  $state \rightarrow action$  τις οποίες αν ακολουθήσει ο agent θα έχει λάβει τις καλύτερες αποφάσεις στο environment. Ο VI είναι αλγόριθμος Dynamic Programming (DP), όπου ισχύει η αρχή της βελτιστότητας: Η optimal solution σε μικρά subproblems οδηγεί σε optimal solution στο συνολικό μεγαλύτερο πρόβλημα. Η Bellman equation χρησιμοποιείται για να οδηγήσει τη διαδικασία της επαναληπτικής ανανέωσης των εκτιμήσεων value για κάθε state. Πηγή: [Introduction to Value Iteration by Carl Bettosi \[2\]](#). Το python script για το πρόβλημα είναι το εξής:

```
1 import numpy as np
2
3 # Define the reward function g(x)
4 g = np.array([1, 2, 3, 4, 5, 4, 2, 0, 1, 2])
5
6 # Define the transition probabilities for u = +1 and u = -1
7 P_plus = 0.5 * np.eye(10) + 0.5 * np.eye(10, k=1)
8 P_minus = 0.5 * np.eye(10) + 0.5 * np.eye(10, k=-1)
9 P_plus[-1, -1] = 1
10 P_minus[0, 0] = 1
11
12 # Initialize the value function
13 V = np.zeros(10)
14
```

```

15 # Define the discount factor
16 a = 0.9 # Change this to try different values of a
17 for a in [0.1, 0.3, 0.5, 0.7, 0.9]:
18     print("=====")
19     print(f"For a = {a}")
20     # Value Iteration
21     while True:
22         V_new = np.max([g + a * P_plus @ V, g + a * P_minus @ V], axis=0)
23         if np.allclose(V, V_new):
24             break
25         V = V_new
26
27     # Compute the optimal policy
28     policy = np.argmax([g + a * P_plus @ V, g + a * P_minus @ V], axis=0)
29
30     print("Optimal Value Function:", V)
31     print("Optimal Policy:", policy)

```

Το αποτέλεσμα φαίνεται εδώ:

```

=====
For a = 0.1
Optimal Value Function:
[1.17283816 2.28393303 3.39473634 4.49999995 5.49999995
 4.49999995 2.34210477 0.12326822 1.16959042 2.222222 ]
Optimal Policy: [0 0 0 0 0 1 1 1 0 0]
=====
For a = 0.3
Optimal Value Function:
[1.73418084 3.16039692 4.57562182 5.92856299 6.92856299
 5.92856299 3.39915123 0.5998433 1.68066852 2.85713911]
Optimal Policy: [0 0 0 0 0 1 1 1 0 0]
=====
For a = 0.5
Optimal Value Function:
[2.98146187 4.94442483 6.83331371 8.49998038 9.49998038
 8.49998038 5.49998038 1.83331372 2.66665795 3.99999128]
Optimal Policy: [0 0 0 0 0 1 1 1 0 0]
=====
For a = 0.7
Optimal Value Function:
[ 6.79703029  9.76607899 12.42288373 14.49980681 15.49980681
 14.49980681 10.88442219  5.86075355  5.12811926  6.6665808 ]
Optimal Policy: [0 0 0 0 0 1 1 1 0 0]
=====
For a = 0.9
Optimal Value Function:
[32.8152669 37.88589049 41.8610971 44.49746074 45.49746074
 44.49746074 40.04291528 32.76192355 28.62293031 27.05466312]
Optimal Policy: [0 0 0 0 0 1 1 1 1 1]

```

## References

- [1] Dimitri P. Bertsekas. Reinforcement learning and optimal control. Draft textbook, 2019. Massachusetts Institute of Technology.
- [2] Carl Bettosi. Introduction to value iteration. *Towards Data Science*, 2022. [Introduction to Value Iteration by Carl Bettosi](#).