Complex Systems

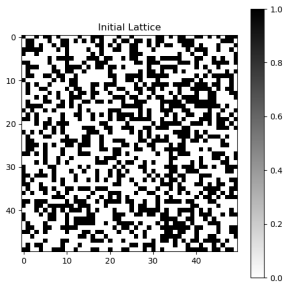
CS2024/problem_6.pdf

Result

Task 1

Listing 1: Plot of the lattice LxL

```python

import numpy as np
import matplotlib.pyplot as plt
import random
import copy

# Helper function to display lattice
def plot_lattice(lattice, title="Lattice"):
    plt.figure(figsize=(6, 6))
    plt.imshow(lattice, cmap='gray_r', origin='upper')
    plt.title(title)
    plt.colorbar()
    plt.savefig(f"{title}.png")
    plt.show()


def generate_lattice(L, p):
    lattice = np.random.choice([1, 0], size=(L, L), p=[p, 1 - p])
    return lattice

# Parameters
L = 50  # Lattice size
p = 0.4  # Probability of a dog (1)

# Generate and plot lattice
lattice = generate_lattice(L, p)
plot_lattice(lattice, "Initial Lattice")

# Save lattice
np.savetxt("lattice.txt", lattice, fmt='%d')
```



Task 2

Listing 2: Flea on the very left dog

```
import numpy as np
import matplotlib.pyplot as plt
import random
import copy

# Helper function to generate a lattice


def generate_lattice(L, p, immunization_rate=0.0):
    # Generate lattice with dogs (1) and empty spaces (0)
    lattice = np.random.choice([1, 0], size=(L, L), p=[p, 1 - p])

    # Apply immunization by changing a fraction of dogs to immune (3)
    total_dogs = np.sum(lattice == 1)
    num_immunized = int(immunization_rate * total_dogs)
    immunized_positions = random.sample(
        list(zip(*np.where(lattice == 1))), num_immunized)

    for pos in immunized_positions:
        lattice[pos] = 3  # Mark as immunized (3)

    return lattice

# Function to find the first dog in the first row or subsequent rows


def find_first_dog(lattice):
    # Iterate over rows in case the first row is empty
    for i in range(lattice.shape[0]):
        for j in range(lattice.shape[1]):
            if lattice[i, j] == 1:  # Found an infected dog
                return (i, j)
    return None

# Flea simulation with infection spread, including immunization


def simulate_flea_updated(lattice, t):
    # Make a copy to avoid modifying the original
    lattice = copy.deepcopy(lattice)
    flea_pos = find_first_dog(lattice)

    if flea_pos is None:
        print("No dog found in the lattice.")
        return lattice

    x, y = flea_pos
    lattice[x, y] = 2  # Mark the initial position as infected

    for _ in range(t):
        neighbors = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]
        valid_moves = [(nx, ny) for nx, ny in neighbors if 0 <= nx < lattice.
            shape[0]
                       and 0 <= ny < lattice.shape[1] and lattice[nx, ny] ==
                           1]
```

```
54
55          # Ensure the flea does not infect immunized dogs (marked as 3)
56          valid_moves = [
57              pos for pos in valid_moves if lattice[pos[0], pos[1]] != 3]
58
59          if not valid_moves:  # If no valid moves, stop the simulation early
60              break
61
62          x, y = random.choice(valid_moves)
63          lattice[x, y] = 2  # Mark as infected
64
65      return lattice
66
67  # Function to calculate epidemic fraction
68
69
70  def updated_epidemic_fraction(lattice, t, num_runs, immunization_rate):
71      fractions = []
72      for _ in range(num_runs):
73          # Generate a new lattice with given immunization rate for each run
74          lattice_with_immunization = generate_lattice(
75              lattice.shape[0], 0.7, immunization_rate)  # Adjust p and
                    immunization
76          infected_lattice = simulate_flea_updated(lattice_with_immunization, t)
77          denom = np.sum(infected_lattice > 0)
78          fraction = np.sum(infected_lattice == 2) / denom if denom > 0 else 0
79          fractions.append(fraction)
80      return fractions
81
82  # Visualization helper function
83
84
85  def plot_lattice(lattice, title="Lattice"):
86      plt.figure(figsize=(6, 6))
87      plt.imshow(lattice, cmap='viridis', origin='upper')
88      plt.title(title)
89      plt.colorbar(label="Cell State")
90      plt.show()
91
92
93  # Parameters
94  L_large = 100  # Larger lattice size
95  p_high = 0.7  # Higher probability of dogs
96  t_long = 1000  # Longer simulation time
97  num_runs = 50  # Number of simulation runs
98  # Testing different levels of immunization
99  immunization_rates = [0.1, 0.3, 0.5]
100
101 # Run the epidemic fraction analysis for different immunization rates
102 all_fractions = {}
103
104 for immunization_rate in immunization_rates:
105     fractions_improved = updated_epidemic_fraction(generate_lattice(
106         L_large, p_high), t_long, num_runs, immunization_rate)
107     all_fractions[immunization_rate] = fractions_improved
108
109 # Plot the updated epidemic size analysis
110 plt.figure(figsize=(8, 5))
111 for immunization_rate, fractions in all_fractions.items():
```

```
    plt.plot(range(num_runs), fractions, marker='o', linestyle='-',
                label=f'Immunization Rate {immunization_rate}')

plt.xlabel("Simulation Run")
plt.ylabel("Fraction of Infected Nodes")
plt.title("Epidemic Size Over Time (With Immunization)")
plt.legend()
plt.savefig("epidemic_size_with_immunization.png")
plt.show()

# Parameters for flea simulation
t = 100  # Number of jumps

# Create a lattice and run flea simulation
lattice = generate_lattice(L_large, p_high)
infected_lattice = simulate_flea_updated(lattice, t)
plot_lattice(infected_lattice, "Lattice After Flea Movements")

# Save infected lattice to file
np.savetxt("infected_lattice.txt", infected_lattice, fmt='%d')
```

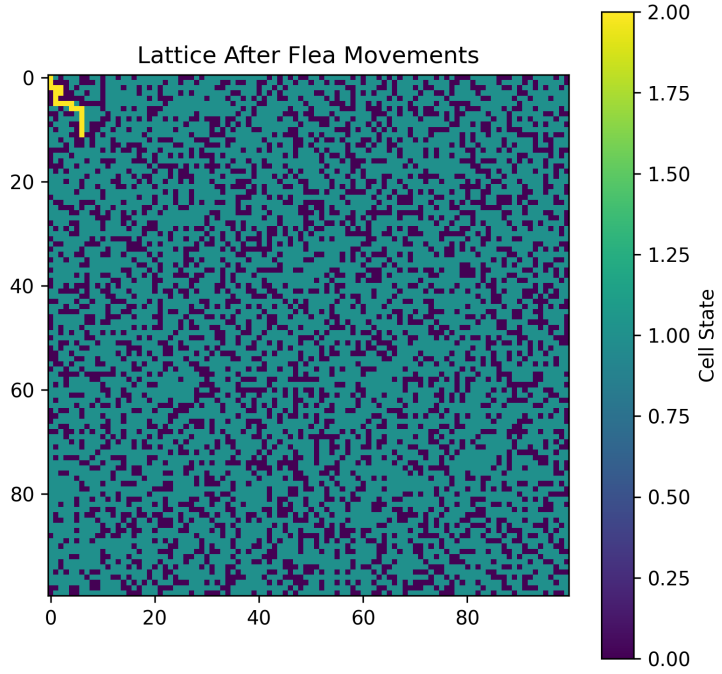


Figure 1: Lattice After Flea Movements

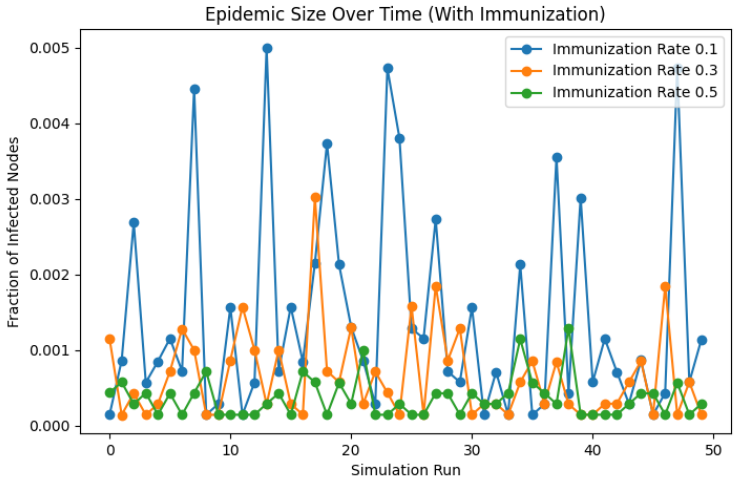


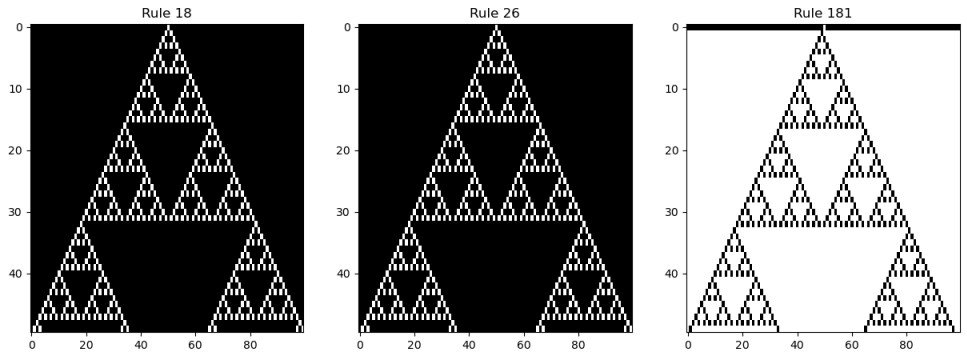
Figure 2: Epidemic Size with Immunization

Task 3

Wolfram Cellular Automata

Listing 3: Flea on the very left dog

```

def wolfram_ca(rule, steps, size):
rule_bin = np.array([int(x) for x in np.binary_repr(rule, width=8)])
ca = np.zeros((steps, size), dtype=int)
ca[0, size // 2] = 1  # Single seed

for i in range(1, steps):
    for j in range(size):
        left = ca[i-1, (j-1) % size]
        center = ca[i-1, j]
        right = ca[i-1, (j+1) % size]
        idx = 7 - (left * 4 + center * 2 + right)
        ca[i, j] = rule_bin[idx]

return ca

# Parameters
steps, size = 50, 100
rules = [18, 26, 181]

plt.figure(figsize=(15, 5))
for i, rule in enumerate(rules):
ca = wolfram_ca(rule, steps, size)
plt.subplot(1, len(rules), i+1)
plt.imshow(ca, cmap='gray', aspect='auto')
plt.title(f"Rule {rule}")

plt.savefig('ca_wolfram.png')
plt.show()
```



Task 5

Listing 4: Game of Life

```python
from scipy.signal import convolve2d
import numpy as np
import matplotlib.pyplot as plt

# Game of Life step function


def game_of_life_step(grid):
    kernel = np.array([[1, 1, 1], [1, 0, 1], [1, 1, 1]])
    neighbors = convolve2d(grid, kernel, mode='same', boundary='wrap')
    return ((neighbors == 3) | ((grid == 1) & (neighbors == 2))).astype(int)

# Plot lattice helper function


def plot_lattice(grid, title="Grid"):
    plt.figure(figsize=(6, 6))
    plt.imshow(grid, cmap="gray_r", origin="upper")
    plt.title(title)
    plt.colorbar(label="Cell State")
    plt.show()


# Initialize grid with a glider pattern
L = 50
grid = np.zeros((L, L), dtype=int)

# Add a glider pattern
glider = np.array([[0, 1, 0],
                   [0, 0, 1],
                   [1, 1, 1]])

# Place the glider in the grid
grid[1:4, 1:4] = glider

# Run simulation
for step in range(10):
    plot_lattice(grid, f"Game of Life - Step {step}")
    grid = game_of_life_step(grid)
```

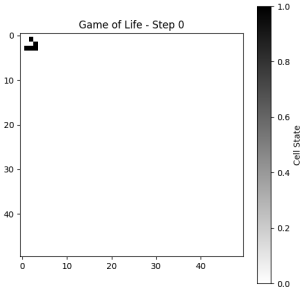


Figure 3: Step 0

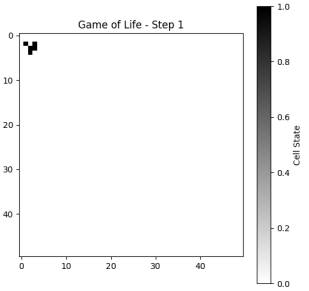


Figure 4: Step 1

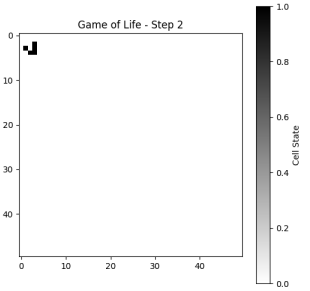


Figure 5: Step 2

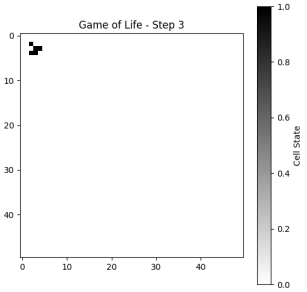


Figure 6: Step 3

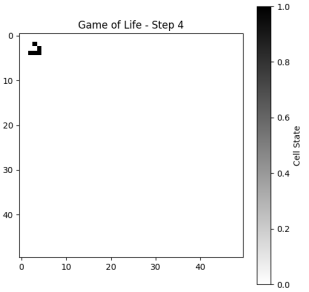


Figure 7: Step 4

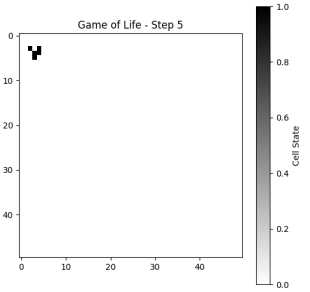


Figure 8: Step 5

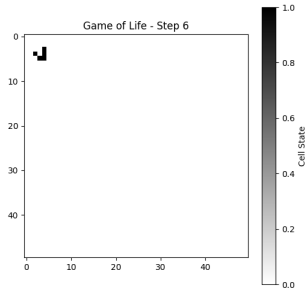


Figure 9: Step 6

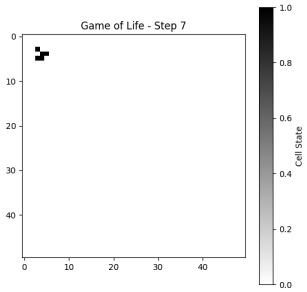


Figure 10: Step 7

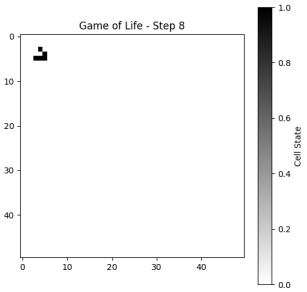


Figure 11: Step 8

Figure 12: Game of Life Steps from 0 to 8