CoNVEY'S GAME OF LIFE

PROJECT REPORT

Presented by

SARDAR Hassan Aftab

Roll no : 14873

1. Abstract

The Conway's Game of Life is a zero-player game that showcases the behavior of cells on a grid. Through a simple set of rules, complex and often surprising patterns emerge. This report presents an implementation of the game in Python, emphasizing its core features, functionality, and expected behaviors. It discusses how the program adheres to the original rules, its visualization methods, and areas for potential improvement.

2. Introduction

The Game of Life, developed by John Horton Conway in 1970, is a classic example of cellular automata. Each cell in the grid exists in one of two states: alive or dead. The evolution of the grid is determined by interactions between cells, defined by specific rules. This project implements Conway's

Game of Life using Python and provides a dynamic visualization to observe its behavior over time.

3. Core Features of Conway’s Game of Life

3.1 Grid Representation

The game operates on a two-dimensional grid where each cell can be alive (1) or dead (0). In the implementation, this is represented using a 2D NumPy array, making it computationally efficient to update and visualize.

3.2 Initial State

The grid is initialized with random states, simulating an environment where cells are distributed randomly between life and death.

3.3 Rule Set

Each cell transitions to the next state based on the following rules:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.

2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

3.4 Neighbor Calculation

Each cell has eight neighbors, and the calculation considers edge wrapping to create a toroidal grid. This ensures no boundary constraints, allowing the simulation to flow smoothly across edges.

3.5 Dynamic Visualization

The simulation is animated using Matplotlib, updating the grid in real time. Alive cells are displayed in white, and dead cells are shown in black, providing a clear view of the evolving patterns.

4. Implementation Details

4.1 Code Design

The program follows a modular design:

- The count_neighbors function calculates the number of live neighbors for each cell.
- The update_gridg function applies the Game of Life rules to generate the next state of the grid.
- The animation is handled using Matplotlib’s FuncAnimation, updating the grid frame by frame.

4.2 Libraries Used

- **NumPy**: For efficient array manipulation and grid representation.
- **Matplotlib:** For visualization and real-time animation of the simulation.

4.3 Grid Size and Performance

The grid size is set to 50x50 for a balance between computational efficiency and visual clarity. Larger grids may slow down the simulation but allow for observing more intricate patterns.

5. Observations and Expected Behaviors

The Game of Life can produce several interesting behaviors depending on the initial configuration, including:

- **Static Structures:** Patterns that remain unchanged (e.g., blocks, beehives).
- **Oscillators**: Patterns that repeat after a fixed number of generations (e.g., blinkers, toads).
- **Spaceships:** Patterns that move across the grid (e.g., gliders).

In this random initialization, users may observe these behaviors emerge naturally as the simulation progresses.

6. Areas for Improvement

1. **Custom Initialization:** Allowing users to define initial configurations instead of relying on random states.
2. **Enhanced Visualization:** Adding colors or shapes to differentiate between states.

3. **Optimized Performance:** Using algorithms like Hashlife for faster computations on large grids.
4. **User Interaction:** Enabling users to pause and modify the grid during the simulation.

Code

The following Python code was used for this project:

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Initialize grid with random 0s (dead) and 1s (alive)
GRID_SIZE = 50
grid = np.random.choice([0, 1], size=(GRID_SIZE,
GRID_SIZE))

# Count the number of alive neighbors for a cell
def count_neighbors(x, y):
    neighbors = [
        (x-1, y-1), (x-1, y), (x-1, y+1),
        (x, y-1),           (x, y+1),
        (x+1, y-1), (x+1, y), (x+1, y+1)
    ]
    return sum(grid[i % GRID_SIZE, j % GRID_SIZE] for i, j
in neighbors)

# Update grid based on Conway's rules
def update_grid():
    new_grid = grid.copy()
    for x in range(GRID_SIZE):
        for y in range(GRID_SIZE):
            alive_neighbors = count_neighbors(x, y)
            if grid[x, y] == 1:  # Alive cell
                if alive_neighbors < 2 or alive_neighbors > 3:
                    new_grid[x, y] = 0  # Dies
            else:  # Dead cell
                if alive_neighbors == 3:
                    new_grid[x, y] = 1  # Becomes alive
    return new_grid

# Animation function
def animate(frame):
    global grid
    grid = update_grid()
    mat.set_data(grid)
    return [mat]

# Set up plot
```

```
fig, ax = plt.subplots()
mat = ax.matshow(grid, cmap="binary")
ax.axis("off")

# Run animation
ani = FuncAnimation(fig, animate, interval=200 ,
blit=True)

plt.show()
```

7. Conclusion

Conway’s Game of Life demonstrates how simple rules can give rise to complex behaviors. The Python implementation successfully captures these dynamics, offering an engaging visualization of the system. Future enhancements could further enrich the user experience and extend its applicability.