Planisuss

Report of the final exam project
Computer programming, algorithms and Data structures -Mod 1
Artificial Intelligence 2022-2023
Yusuf Mehmet COLAK

1. Introduction

The goal of the project is to code in Python a simulated world called “Planisuss” and then use different libraries to understand what happens in the simulated word. Only two main support libraries were allowed: NumPy and Matplotlib.

Matplotlib was used to visualize the information of the word, while NumPy was mainly used in building the arrays that were needed to plot the main simulation grid.

The world is composed of three main entities type:

- Herbast → organized in Herds

- Carviz → organized in Prides

- Vegetob

The herbast are the “herbivore” while carviz are the “carnivores.” The vegetobs are the vegetation.

Planisuss is a grid-like word, each cell in the grid can contain multiple objects, and the colour of each cell depends on its content. In addition to that there are two types of cells: ground, and water. Ground cells can contain any type of living creature, while water cells cannot contain any creature.

More detailed characteristics on the world itself can be learned in the project specification that should come alongside this report. From now on this report will focus more on the development challenges and strategies that were devised to build a functional simulation.

Classes

Design and Implementation

The `Cell` Class

In the development of Planisuss, the `Cell` class serves as a fundamental building block of the environment. Each instance of `Cell` represents a discrete unit within the simulated world, containing key properties that define its state and the entities it holds.

Attributes of `Cell`

- ***VEGETOP***: This attribute holds information about the vegetation present in the cell. It could represent the amount or type of vegetation.

- ***ERBAST***: This attribute represents a specific type of entity within the cell, a herbivore.

- ***CARVIZ***: This attribute represents another type of entity within the cell, a carnivore.

- ***TYPE***: This attribute indicates whether the cell is land (`1`) or water (`0`), helping to define the terrain of the simulated world.

Design Decision: Using Arrays Instead of Classes for Entities

In Planisuss, a conscious decision was made to avoid creating separate classes for each type of entity, such as `Erbast` and `Carviz`. Instead, these entities are represented by their properties stored in arrays. Each index in these arrays corresponds to a specific property, which simplifies the overall structure and avoids the complexity of multiple classes.

Benefits of This Approach

1. **Simplicity**: By storing properties in arrays, the design remains straightforward. There is no need to manage multiple class definitions and their relationships.

2. **Efficiency**: Accessing and manipulating entity properties through arrays can be more efficient, especially when dealing with large numbers of entities.

3. **Flexibility**: This approach allows for easy addition or modification of properties without altering the class structure. New properties can be appended to the array, and existing ones can be updated directly.

Example of Entity Properties in Arrays

Suppose we have entities `Erbast` and `Carviz`, each with properties like health, age, and energy. Instead of creating classes, these properties are stored in arrays:

 # Properties of Erbast

erbast_properties = [energy, socialability, age, lifetime]

 # Properties of Carviz

carviz_properties =[energy, socialability, age, lifetime]

Discussion on the Design Choice

This design choice emphasizes the balance between complexity and functionality. While traditional object-oriented design might suggest creating a class for each type of entity, the array-based approach used in Planisuss provides a leaner and more flexible solution. It allows for rapid prototyping and easier adjustments as the simulation evolves.

Trade-offs

- **Lack of Encapsulation**: One downside of this approach is the lack of encapsulation that classes provide. Each property is exposed directly, which can lead to potential misuse or errors.

- **Reduced Readability**: Arrays with indices might be less readable compared to named properties within classes. Documentation and clear conventions are necessary to mitigate this.

The grow_vegetop Method

In the Cell class, the grow_vegetop method is designed to simulate the growth of vegetation within a cell. This method plays a crucial role in maintaining and updating the state of the simulated environment, ensuring that vegetation levels are managed properly over time.

1. **Growth Condition**:

- The method first checks if the cell is of type 1, which indicates that it is ground. Vegetation can only grow on ground cells, not on water cells (type 0).

2. **Vegetation Growth**:
   - If the cell is ground, the vegetation (VEGETOP) is increased by 1 unit.
   - To ensure that the vegetation level does not exceed a certain maximum threshold, the min function is used. This keeps VEGETOP capped at 100.

The grow_carviz Method: The grow_carviz method effectively manages the lifecycle and energy levels of CARVIZ entities within the Planisuss simulation. By incrementing age, reducing energy periodically, and checking for end-of-life conditions, the method ensures that the simulation reflects realistic aging and survival processes. This contributes to the overall realism and functionality of the simulated world.

Attributes of CARVIZ

- **Energy (CARVIZ[0])**: Represents the energy level of the CARVIZ. Energy is crucial for the entity's survival and activities.
- **Sociability (CARVIZ[1])**: Represents how social the CARVIZ is. This might affect interactions with other entities, although it is not used in the grow_carviz method.
- **Age (CARVIZ[2])**: Represents the current age of the CARVIZ.
- **Lifetime (CARVIZ[3])**: Represents the total lifespan of the CARVIZ. When the age equals the lifetime, the entity is considered to have reached the end of its life.

Functionality

1. **Age Increment**:
   - The method starts by checking if CARVIZ exists in the cell. If it does, the age of CARVIZ (CARVIZ[2]) is incremented by 1, simulating the passage of time for the entity.
2. **Energy Reduction**:
   - Every time the age of CARVIZ is a multiple of 10, the energy (CARVIZ[0]) is reduced by 1. This could represent periodic energy expenditures such as feeding or other activities that require energy.
3. **End of Life Check**:
   - The method checks two conditions to determine if the CARVIZ should be neutralized (removed from the simulation):
     - If the age (CARVIZ[2]) equals the lifetime (CARVIZ[3]), indicating the CARVIZ has reached the end of its natural lifespan.
     - If the energy (CARVIZ[0]) is less than or equal to 0, indicating the CARVIZ has run out of energy and cannot survive.

The grow_erbast Method:

The grow_erbast method follows the exact same functionality and logic as the grow_carviz method. It manages the age and energy levels of ERBAST entities within a cell, ensuring they age over time,

expend energy periodically, and are neutralized when they reach the end of their lifespan or run out of energy.

Code Duplication and Design Considerations

One apparent downside of using simple arrays to implement class attributes is the repetition of code. For instance, both grow_erbast and grow_carviz methods contain similar logic. If ERBAST and CARVIZ were distinct classes, we could have created a general Animal class with shared properties and methods, and then derived specific classes for ERBAST and CARVIZ from it. This approach would eliminate code duplication and promote better code organization and reusability.

The eat Method

The eat method in the Cell class simulates the feeding behavior of the ERBAST entity, ensuring it consumes available vegetation to gain energy.

Logic and Functionality

1. **Check Conditions**:
   - The method first checks if vegetation (VEGETOP) is available and if ERBAST is present in the cell. Feeding only occurs if both conditions are met.
2. **Consume Vegetation**:
   - If the conditions are satisfied, vegetation is reduced to simulate consumption. This ensures ERBAST gets the necessary food while keeping vegetation levels realistic.
3. **Replenish Energy**:
   - The ERBAST‘s energy is increased by consuming vegetation, but it’s capped at a maximum value (30) to reflect natural limits.
4. **Lifecycle Management**:
   - The method checks if ERBAST has reached its maximum age or has no energy left. If so, the neutralize function is called to remove ERBAST from the simulation, representing death due to old age or starvation.

Importance

- **Ecosystem Balance**: Models interaction between herbivores and plants, maintaining ecological balance.
- **Survival Dynamics**: Ensures ERBAST entities need food to survive, creating dynamic resource management.
- **Population Control**: Manages population by removing entities that starve or reach the end of their lifespan.

The fight Method

The fight method simulates a conflict between the ERBAST and CARVIZ entities within a cell. This interaction models predatory behavior, energy dynamics, and includes an element of randomness to reflect the unpredictability of such encounters.

Logic and Functionality

1. **Initial Conflict Check**:
   - The method begins by checking if both ERBAST and CARVIZ are present in the cell. If either is absent, the conflict does not occur.
2. **Energy Comparison**:
   - If CARVIZ has lower energy than ERBAST, both entities lose energy, simulating the struggle. Specifically, ERBAST loses 2 energy points, and CARVIZ loses 5 energy points. This represents the predator using more energy in the fight.
3. **Neutralization Check**:
   - After the energy loss, the method checks if either entity's energy has fallen to zero or below. If so, the neutralize function is called to remove the entity from the simulation, indicating death due to exhaustion.
4. **Equal Energy Levels**:
   - If ERBAST and CARVIZ have the same energy level, the fight does not occur, and the method returns early. This simulates a standoff where neither entity gains the upper hand.
5. **Random Escape Chance**:
   - If CARVIZ has higher energy than ERBAST, there is a chance for ERBAST to escape. Using a random number generator, the method introduces a 23% chance (1 - 0.77) that ERBAST fails to escape and gets caught by CARVIZ. If ERBAST fails to escape, it is neutralized, and CARVIZ gains the remaining energy of ERBAST, up to a maximum of 40 energy points.

Initialization of the World

Key Components

1. **Constants**:
   - KATSAYI = 10: This constant is used as a multiplier to scale certain parameters, ensuring that they have appropriate magnitudes within the simulation.
2. **Function Definition**:
   - initilazie_world(row_number, col_numbers): This function initializes a world grid with specified rows and columns.

Steps and Logic

1. **Initialize Cells List**:
   - cells = []: An empty list to store rows of cells.

2. **Type Choices**:
   - type_choices = [0, 1]: Defines possible types for cells, where 1 represents ground and 0 represents water.
3. **Grid Creation**:
   - A nested loop iterates through each cell position in the grid. The outer loop runs through the rows (row_number), and the inner loop runs through the columns (col_numbers).
4. **Border Cell Check**:
   - is_border_cell: A boolean variable that checks if the current cell is on the grid's border. Border cells are more likely to be water.
5. **Probability Distribution**:
   - probabilities = [1, 0] if is_border_cell else [0.2, 0.8]: Sets the probability distribution for choosing the cell type. Border cells are always water ([1, 0]), while non-border cells have a higher probability of being ground ([0.2, 0.8]).
6. **Cell Type Selection**:
   - cell_type = np.random.choice(type_choices, p=probabilities): Randomly selects the cell type based on the defined probabilities.
7. **Cell Initialization**:
   - If the cell is water (cell_type == 0), it is initialized with None values for VEGETOP, ERBAST, and CARVIZ, and TYPE set to 0.
   - If the cell is ground (cell_type == 1), it further randomizes the content:
     - **Animal Selection**: Randomly chooses between different animal configurations (2, 3, or 4).
     - Depending on the chosen animal type:
       - **Type 2**: Initializes a cell with a random VEGETOP, no ERBAST, and a CARVIZ with randomized properties.
       - **Type 4**: Initializes a cell with a random VEGETOP, both ERBAST and CARVIZ with randomized properties.
       - **Type 3**: Initializes a cell with a random VEGETOP, an ERBAST with randomized properties, and no CARVIZ.
8. **Append Rows**:
   - Each initialized row is appended to the cells list, which represents the entire grid.

The initilazie_world function effectively creates a diverse and randomized grid of cells, each with unique properties. By incorporating random choices and varying parameters, the function ensures that the simulation world is varied and dynamic, reflecting realistic environmental conditions and interactions. This setup forms the foundation for further simulation processes within Planisuss.

Shuffling

The shuffle_type_1_cells function enhances the diversity and randomness of the simulation by redistributing ground cells after the initial world setup. This additional layer of randomness can help create more varied and unpredictable simulation scenarios, making the simulation more robust and reflective of real-world variability.

The update_world Function

The update_world function is designed to simulate the progression of the world over a specified number of days. This function iterates through each cell in the grid and updates the state of the world by invoking various methods that simulate growth, feeding, and interactions among entities.

Broad Explanation of the main_plot Function with Herd and Pride Implementation

The main_plot function processes the current state of a grid of cells in the simulated world. It collects various data points, such as vegetation levels, creature positions, and the status of entities (ERBAST and CARVIZ). The function also handles merging and movement of creatures, accounting for their interactions and updating the grid accordingly.

Function Structure and Logic

1. **Initial Setup and Error Handling**:
   - The function begins by checking if the cells input is None and raises an error if so.
2. **Grid Initialization**:
   - Three grids are initialized: grid for general data, vegetop_grid for vegetation data, and creature_grid for creature positions.
   - Counters and lists are initialized to track dead creatures, their coordinates, and positions.
3. **Preparation of creature_grid**:
   - A grid (creature_grid) filled with zeros is created to represent the positions and status of creatures within the grid.
4. **Processing Each Cell**:
   - The function iterates over each cell in the grid.
   - For each cell, it checks the type and contents (whether it contains ERBAST or CARVIZ).
5. **Handling Water Cells**:
   - Water cells are marked with a -1 in the creature_grid.
6. **Handling ERBAST**:
   - If the cell contains an ERBAST, its energy and vegetation levels are recorded.
   - The function also updates the creature_grid with a unique identifier for ERBAST.
   - Coordinates and positions of ERBAST are stored for later processing.
   - **Herd Creation**:
     - If there are adjacent ERBAST entities and certain conditions are met (e.g., no adjacent vegetation, low vegetation level), the function attempts to merge them into a herd.
     - **Conditions**: Both ERBAST entities must have sociability set to 1 to form a herd.
     - **Merging Process**:
       - **Sum Energies**: The energies of the two ERBAST entities are summed.
       - **Average Age**: The ages of the two ERBAST entities are averaged.
       - **Maximum Lifetime**: The lifetime is set to the maximum of the two entities' lifetimes.
       - **Update Grid**: The merged entity is updated in the grid, and the old entity is neutralized.
7. **Handling CARVIZ**:

- If the cell contains a CARVIZ, its energy and vegetation levels are recorded.
- The function also updates the creature_grid with a unique identifier for CARVIZ.
- Coordinates and positions of CARVIZ are stored for later processing.
- **Pride Creation**:
  - If there are adjacent CARVIZ entities and certain conditions are met (e.g., no adjacent ERBAST), the function attempts to merge them into a pride.
  - **Conditions**: Both CARVIZ entities must have sociability set to 1 to form a pride.
  - **Merging Process**:
    - **Sum Energies**: The energies of the two CARVIZ entities are summed.
    - **Average Age**: The ages of the two CARVIZ entities are averaged.
    - **Maximum Lifetime**: The lifetime is set to the maximum of the two entities' lifetimes.
    - **Update Grid**: The merged entity is updated in the grid, and the old entity is neutralized.

8. **Handling Empty Cells**:
   - Cells that are empty or contain only vegetation are marked appropriately in the creature_grid.
9. **Updating the Grid and Counting Entities**:
   - The function updates the main grid with processed data.
   - Counters for dead and active ERBAST and CARVIZ are updated based on their energy levels.
10. **Output**:

- The function prints summary statistics about the state of the grid, including the number of dead and active creatures and the energy and vegetation levels in the grid.

Key Points of Interest

- **Herd and Pride Creation**: The function includes logic to merge ERBAST and CARVIZ entities if they meet certain conditions, simulating social behaviors and group dynamics. This merging involves summing their energies, averaging their ages, and taking the maximum lifetime.
- **Randomness and Interactions**: The function checks for adjacent cells and environmental factors, introducing an element of randomness and interaction between entities.
- **Grid Representation**: By maintaining separate grids for vegetation, creatures, and general data, the function provides a detailed and layered view of the simulation's state.

Conclusion

The main_plot function is a comprehensive tool for analyzing and updating the state of a simulated world. It processes each cell in the grid, manages interactions and movements of entities, and provides detailed output about the current state of the ecosystem. This function is crucial for visualizing and understanding the dynamics of the simulation over time.

Explanation of Graphs and Their Functionality

The `main_plot` function generates visualizations to represent the state of the simulated world, while the `animation` function animates the progression of this state over several days. The following section explains the graphs and their functionalities:

Colormap Definition

```python
colors = [
    (0, 'blue'),        # Water
    (0.1, 'black'),    # Ground
    (0.2, 'yellow'),   # Start of Carviz range
    (0.6, 'orange'),
    (0.8, 'lightgreen'), # End of Carviz range
    (1.0, 'darkgreen')  # End of Erbast range
]

cmap = LinearSegmentedColormap.from_list('creature_colors', colors, N=301)
```

- **Colormap Definition**: Custom colors are assigned to different cell types and creature energy levels for clear visual differentiation.

Plot Setup

```python
fig, axs = plt.subplots(2, 2, figsize=(15, 12))
```

- **Subplots Creation**: Four subplots are created in a 2x2 grid layout, with a specified figure size.

First Graph - Creature Energy Levels

```python
energy_im = axs[0, 0].imshow(grid, cmap='hot')
fig.colorbar(energy_im, ax=axs[0, 0], fraction=0.046, pad=0.04)
axs[0, 0].set_title('Creature Energy Levels')
```

- **Functionality**: This graph visualizes the energy levels of creatures across the grid.
- **Colormap**: Uses a 'hot' colormap to represent different energy levels.
- **Colorbar**: Provides a reference scale for energy levels.

Second Graph - VEGETOP Density

```python
dens_veg_im = axs[0, 1].imshow(vegetop_grid, cmap='Greens')
fig.colorbar(dens_veg_im, ax=axs[0, 1], fraction=0.046, pad=0.04)
axs[0, 1].set_title('VEGETOP Density in Each Cell')
```
- **Functionality**: This graph shows the density of vegetation (`VEGETOP`) in each cell.
- **Colormap**: Uses 'Greens' to represent vegetation density.
- **Colorbar**: Provides a reference for vegetation density levels.

Third Graph - Number of Dead and Alive Creatures

```python
categories = ['ERBAST', 'CARVIZ']
alive_counts = [erbast_counter, carviz_counter]
dead_counts = [dead_erbast, dead_carviz]
index = range(len(categories))
bar_width = 0.35

bars_alive = axs[1, 0].bar(index, alive_counts, bar_width, label='Alive', color='green')
bars_dead = axs[1, 0].bar([p + bar_width for p in index], dead_counts, bar_width, label='Dead',
color='red')

axs[1, 0].set_title('Counts of Alive and Dead Creatures')
axs[1, 0].set_xticks([p + bar_width / 2 for p in index])
axs[1, 0].set_xticklabels(categories)
axs[1, 0].legend()
```
- **Functionality**: This bar chart compares the counts of alive and dead creatures for both `ERBAST` and `CARVIZ`.
- **Bar Chart**: Green bars represent alive creatures, and red bars represent dead creatures.
- **Legend**: Differentiates between alive and dead counts.

Fourth Graph - Placement of the Creatures

```python
creature_im = axs[1, 1].imshow(creature_grid, cmap=cmap, vmin=-1, vmax=300)
cbar = fig.colorbar(creature_im, ax=axs[1, 1], fraction=0.046, pad=0.04)

axs[1, 1].set_title('Creature Locations')
cbar.set_ticks([-1, 20, 100, 250])
cbar.set_ticklabels(['Water', 'Ground', 'Carviz (0-100)', 'Erbast (100-300)'])
```

```
- **Functionality**: This graph visualizes the locations of creatures within the grid.
- **Colormap**: Custom colormap differentiates water, ground, and creatures.
- **Colorbar**: Provides reference points for different entities.

Final Adjustments and Display

```python
plt.tight_layout()
plt.suptitle('Platinus')
manager = plt.get_current_fig_manager()
manager.resize(2100, 1200)
plt.show()
```
- **Layout and Title**: Adjusts the layout for better spacing and sets a title for the entire figure.
- **Window Resize**: Resizes the window for better visibility.
- **Display**: Shows the final plot.

`animation` Function

```
- **Interactive Mode**: Enables interactive mode for real-time updates.
- **Day Counter**: Iterates over a specified number of days, updating the world and plotting the state.
- **Pause**: Adds a pause to simulate time progression.
- **Final Display**: Shows and then closes the final plot after the animation loop completes.

**Brief**

The `main_plot` function generates comprehensive visualizations of the simulation's state, including energy levels, vegetation density, creature counts, and locations. The `animation` function animates these visualizations over time, providing a dynamic view of how the simulated world evolves. These tools are essential for analyzing and understanding the behavior and interactions within the Planisuss simulation.

## Conclusion

The Planisuss project successfully demonstrates the creation and simulation of a complex, dynamic world using Python. By leveraging various libraries, we have modeled an ecosystem with detailed interactions among entities, such as `ERBAST` and `CARVIZ`, and their environment. The key

components of this project include the initialization of a diverse world grid, simulation of daily activities, and visualization of the ecosystem's state.

***Initialization and Diversity***:
- The `initilazie_world` function sets up a world grid with cells of different types and properties, ensuring a varied and realistic starting point. The use of randomness in cell type selection and entity attributes further enhances the diversity of the simulation.

*****Entity Management*****:
- The `grow_carviz` and `grow_erbast` methods simulate the aging and energy management of the entities, reflecting realistic lifecycle processes. The `eat` method models the feeding behavior of herbivores, ensuring their survival depends on available resources. The `fight` method introduces predator-prey dynamics, with energy levels influencing the outcomes of conflicts.

*****Herd and Pride Formation*****:
- The simulation incorporates social behaviors by merging entities into herds or prides when conditions are met. This involves summing energies, averaging ages, and taking the maximum lifetime, simulating the benefits of social structures in the animal kingdom.

*****Visualization and Animation*****:
- The `main_plot` function provides comprehensive visualizations of the simulation's state, including creature energy levels, vegetation density, and the counts of alive and dead creatures. Custom colormaps and multiple subplots offer a detailed and intuitive view of the ecosystem.
- The `animation` function brings the simulation to life, showing the evolution of the world over time. It allows real-time observation of changes and interactions within the grid, enhancing our understanding of the dynamic processes at play.

*****Overall Impact*****:
- This project showcases the power of Python in modeling complex systems and simulating real-world phenomena. By combining robust data structures, detailed simulations, and effective visualizations.

The Planisuss project underscores the importance of detailed modeling and visualization in understanding complex systems. The methods and approaches used here can be applied to various fields, from ecology to resource management, offering a versatile platform for studying interactions within any simulated environment. This project is my First year university project assigned by Prof. Ferrari from University of Milan. It serves as a introduction to computer programming and implementations of fundamental programming approachs.