Matteo Caviglia
Sofia Gaia Formenti

29.09.2023

Planissus Project Report

The project consisted in the development of a life simulation called ‘Planissus’. The main features of the simulation are the planet, composed of water or ground cells, each cell inhabited by individuals of three different species: Erbast (an herbivore-like creature), Vegetob (a plant nourishment of Erbast creatures) and Carviz (a carnivore-like creature, hunter of Erbast). The aim of the project was to implement a visual representation of the planet in a grid-like way and of its ‘inhabitants’, updating at each passing day the display. Furthermore, the user is able to interact with the simulation, pausing, restarting or generating a new world. A useful feature of the program, to understand the mechanisms of the life simulation, is graph tracing, that can be easily obtained by hovering with the cursor on the cell of interest and clicking on it. The user can see real time data of that cell in addition to a histogram graph and a line graph of each creature in the cell.

Introduction

Similar to the Planissus project, other life-simulations have been developed, like the Wa-Tor and Conway’s Game of Life. Our aim with the project was to create a life simulation with detailed creature-interaction and a clear and straightforward display, with the possibility of data visualization both raw and in graph form and we tried to strive for a video-game feel to the program.

Goals, problems and constraints

The main features that we aimed to implement were first of all the map generation and the world initialization, the casual placement of each initialized creature in a valid cell that passed the ground check. This presented the first difficulty: generating a map with a single island surrounded by water cells without the presence of other islands other than the main one.
An important feature was the passing of time, every second the map updates and each creature performs certain tasks. This implementation brought forth a new challenge: individual behavior, how an individual of each species makes decisions that are not casual but are based on its own characteristics and the environment that surrounds it. Furthermore each cell could potentially be inhabited by all three species, bringing a further element into play that was species interaction, how exactly individuals interact with members of the same or of different species.
Finally an objective we had in mind was that of a straightforward user interface, to make it easy to interact with the simulation by the means of a pause, a restart simulation and a generate a new map button and also making it possible to analyze the contents of a single cell and to track graphs that update in real time when the simulation is unpaused.
The aforementioned were the main focus points that we were committed to implement.

Assumptions, methods, procedures

The starting point of the project was the map creation and the grid that would contain each cell information, then the creatures implementation, the interactions and finally the display.

1. The map

To generate the map we developed the `Map_generator()` function, which works by generating a 2D array (a variable we called "mappa") initialized to contain all `'G'` strings with the numpy function. The peripheral cells are set as water cells and through a while loop the function "expands" the water cells from the already existing ones to cells which are not. This lasts until all water cells have been placed, at that point all the remaining cells are set as ground cells. A further function is run to delete single cell islands `Delete_Single_Cell_Islands(mappa)` and finally the function returns the map.
Another 2D numpy array was used in the `Generate_Grid` function, which assigns a value between 0 and 4 to every position based on the presence of creatures in the world. This function has been of great use in the world display.

2. Initialization

Different functions define each creature-type initialization, where the Vegetob initialization only passes as argument the position, whereas Erbast and Carvix creatures also have an energy argument, which is calculated with the numpy random integer generation function.
This initialization provides for the Vegetob a density and returns a creature belonging to the class Vegetob assigned to position, whereas Erbast and Carvix initialization returns a creature of the respective classes assigned to a position with a certain age, social attitude, total lifetime and energy. These three initialization functions are called by a general function, `Initialize_Creature` that passes as arguments the position and the creature type. Finally the initialization of a herd of Erbast and of a pride of Carvix was implemented. For a pride or a herd to form it is sufficient to have a single individual of the species, so every Erbast and every Carvix is a herd/pride on its own.

3. Classes

Various classes were implemented to use as "blueprints" for creating objects with similar attributes and behavior, in this case to address different properties of creatures and creature groups. First of all the `Creature` class, with only the position attribute, the `Vegetob(Creature)` class, with position and density attributes (the former initialized from the constructor class, the Creature, through the super() function). The functions defined for the Vegetob class being the growth, the addition or removal of density, a method to return the value of density and finally a counter for days passed since growing, functional for the probabilistic `grow(self)` function.
The `Erbast(Creature)` class, with methods involving energy (addition, removal, return), position update (this method wasn't specified for Vegetob as it is a static class, whereas Erbast at each passing day can decide to move to a neighbor cell), age addition and age check (that returns

whether the creature has reached it's lifetime or not). Finally the graze method, that checks whether in the same cell as the Erbast a Vegetob is present and if it is, removes density from the Vegetob and adds energy to the Erbast.
Almost identical to the Erbast class, the `Carvix(Creature)` class, except the latter doesn't contain the graze method.
More interesting and complex are the `Herd` and the `Pride` classes.
In the Herd class, apart from the basic general methods, an important implementation was that of the leader, each Herd has a leader which is identified as the member with highest social attitude and energy sum.
The movement function `move_herd(self, world, mappa)` allows for a movement in one of the four neighbor cells (up, down, left, right), the method checks if the direction is allowed (whether its a ground cell in the map) and, if it is, appends it to the possible directions cells.
The direction is chosen through the `evaluate_direction(self, direction)` function and if the direction is not None (in which case the herd remains in its cell), the position of the herd is updated to the new one and the cost of the movement is subtracted. The evaluate direction function works by returning a `weighted_sum` variable that takes into consideration all the factors that make a herd choose a direction rather than another. The factors are: the `dir_weight` (whether the direction is the same as before, the opposite one or neither), `target_density` (the density of Vegetob in the target cell), `vegetob_density` (the density of Vegetob in the direction cell), `danger_level` which assesses the number of predators in the neighbor cells on which the Herd has awareness, combined with the strategy of the Herd (if the strategy is set to 'Escape' the danger level is tripled, if it's set to 'Fight' it is halved), `group_energy` and `energy_expenditure` which are self explanatory.
Two checking method have been implemented: `check_members` that splits the herd if the members are more than a certain number and also checks if a member has zeroed out on energy and, if so, removes it from the herd and also `check_if_empty` that simply returns True or False whether the Herd list is empty or not.
`social_decision` is a method that based on each members' social attitude and energy determines whether or not it will stay in the Herd or separate and create a new one.
The `move_or_stay` method, based again on strategy (foraging, migration or none), assesses the probability for the Herd to move or stay and finally the `Spawn` method that, when a herd member reaches its total lifetime and dies a varying number of children will spawn in that position.
Regarding the `Pride` class, the methods are quite similar to the ones explained before for the Herd class, except for the `hunt` method. This function compares energy levels of the leaders of the Pride and the targeted Herd, if the former is greater, the Pride leader and members perceive an energy gain and a number of Herd members is eliminated based on the number of the Pride members.

4. The `Live_One_Day` function

This function defines what happens at each day of the simulation, it's arguably the pivot of the code and encompasses all the main implementations.
First of all the growth of Vegetob is implemented, every day the existing Vegetob grow and the function `Random_Growth` is called, that randomly spawns new Vegetobs in eligible cells.

Next: the movement dynamic, this is equal for both Herds and Prides and exploits the color to which these are set. When a Pride or a Herd is initialized, the color is set to white, in this case the `social_decision` function is called, which, in turn, calls the `move_or_stay` function (implemented in the classes). If the Herd or the Pride is empty, the color is set to black.
If the Herds' color is set to green the `graze` function is called and then the color is set to black, whereas if the Prides' color is green the `prides_HOS` function is called. The color is set in the `move_or_stay` function of the Herd and Pride classes.
The spawning method for both Herds and Prides simply calls the `Spawn` function and resets the color to white.
Finally the `check_cell` function is called that, in turn, calls the `check_list` function, which refreshes each group list, removing creatures which have died on that day and it ensures that the program works fine.

5. Herd and Pride interaction with themselves

If two Herds or two Prides find themselves in the same cell, the two groups interact. In the case of the Herds the only possible interaction is `herd_union`, the members of the second herd are "transferred" to the first one and the second herd is eliminated. If the number of members in the first herd exceeds a certain value, another herd is created where the remaining members are inserted. Regarding Pride interaction, two things can happen: union or fight. In the first case, the `pride_union` function is called, which works exactly like the herd union function. In the second case the function is `pride_fight`, this function compares the energy levels of the leaders of each pride and eliminates the pride with the weakest leader, assigning to the victorious leader the energy of the defeated one. If both leaders have the same value of energy the two prides eliminate one another.

6. Display

Regarding the display, this was implemented using the tkinter module. On the display the user has the possibility of pausing and starting the simulation, restarting the simulation with the same world or generating a new world (`press_pause_button()`, `restart_sim_button()`, `new_map_button()`). By hovering with the cursor over the map the user is returned with the coordinates of the point of interest (`update_cursor_label()`) and by right-clicking a graph-selection window opens, where the user can choose which type of graph he would like the program to display (`inspectCell()`). The inspect cell function calls the `graph_selection` class, which opens a window in which the user can choose to visualize either the data of the cell, a line graph of one of the three species or a histogram graph where all three species are represented. By pressing play the user can see the development of the cell in the graph, which updates with the world. The visualization of the cell data exploits the `get_vegetob_data`, the `get_erbast_data` and the `get_carvix_data` functions, which are "packed" into the `get_cell_data` function, called by the `visualize_cell_data` class. Finally the various combinations of possible graphs that can be drawn are evaluated and returned by the `press_confirm` function.

Tools and resources

The libraries used are:

- **Numpy**
  Version: 1.24.1

- **Matplotlib**
  Version: 3.6.3

- **Tkinter**
  Version: 8.6

- **Ttkbootstrap**
  Version: 1.10.1

The main inspiration and help for the project was given by the Game of Life and Wa-Tor. The development was based on the v0.95 of the project specification provided.

Testing and results

To make sure our program worked right, we had to do a good amount of backtracking and debugging. Sometimes our initial code didn't quite align with our intended implementations, so we had to backtrack and rewrite some lines of code in a way that everything was coordinated. Also, Debugging was particularly challenging due to the extensive nature of the code and if an error came up we couldn’t exactly spot the mistake.We relied on various print statements in order to isolate the error.

Take, for example, a subtle yet impactful error we encountered that involved a grid transposition. The x-coordinate of the map mistakenly became the y-coordinate of the grid, and vice versa. We only noticed that something was not right because the data we visualized wasn’t always right, while all the rest looked fine. It required an extensive debugging effort just to detect this small but influential mistake.

Possible improvements

Some improvements could be made to the program, in particular the correct deletion of single cell islands, which sometimes is not done properly (near the main island small islands form which aren’t deleted because one of the diagonal cells is neighbor to the main island).
A further implementation could be the possibility for a creature to move not only in the four cross neighbor cells (up, down, left, right), but also in the four diagonal neighbor cells.

A display implementation could be done, with a more graphic and video-game like initialization display.
Finally a further improvement could be to implement the possibility for the user to set the initial data, like the dimension of the map, the number of creatures and other variables.

In conclusion the project development has covered almost all the aims that we had set at the beginning, even exceeding our expectations in terms of the User Interface.