Planisuss - Final exam project - A. Y 2022/23 Computer programming, Algorithm and Data str., Mod. 1 Bachelor degree in Artificial Intelligence

Alessandro Ghiotto 513944

1 Goal of the project

The final exam project consists in the design and implementation of a simulation of a fictitious world called "Planisuss". The simulation is intended to provide data to be interactively visualized using the library matplotlib. The main objective of the first part (the simulation of the Planisuss world) revolves around OOP and the use of inheritance. Instead, the second part focuses on the use of an external library (that is matplotlib). The main constraints that I have faced during the elaboration of this projects concern the computational complexity of the model, I had to choose a smaller grid and a fewer number of days of the simulation than the ones indicated in the guidelines of the project.

2 README

Download the files *Ghiotto_Alessandro_513944_planisuss_constants.py* and *Ghiotto_Alessandro_513944_planisuss_main.py*, then open the second one in IDLE or in another IDE, such as Spyder, then simply run the code (it is preferable to open the image in full screen).

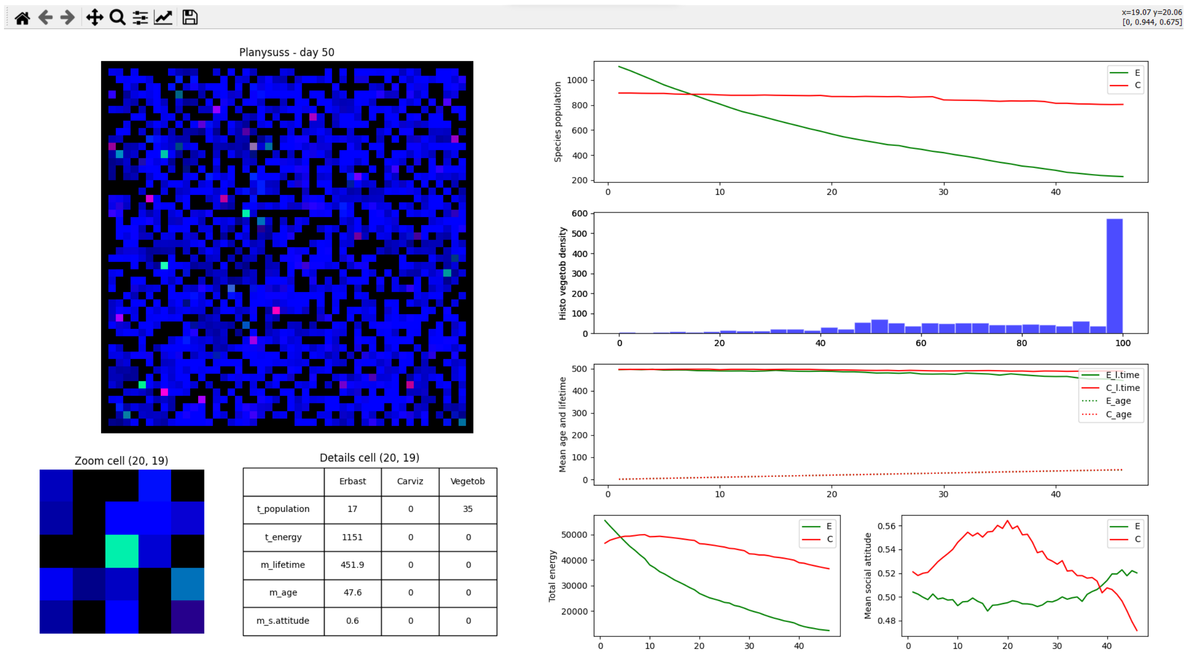
For modifying all the constants present in the simulation just edit the file *Ghiotto_Alessandro_513944_planisuss_constants.py.* Here we can find all the lower and upper bounds for the values of the stats of Erbast and Carviz, constants for the initialization of the code like the number of rows and columns of the grid, and other ones used for the decision function presents in the code.
key_press_event interactivity with the simulation (keyboard):

- **"escape"**: close the simulation;

- **“spacebar”**: play/pause the simulation;
- **“r”**: visualize only the Carviz on the grid;
- **“g”**: visualize only the Erbast on the grid;
- **“b”**: visualize only the Vegetob on the grid;
- **“a”**: visualize the complete grid;
- **“u”**: update manually to the next day (only while pause is True).

button_press_event interactivity with the simulation (mouse):

- **“click”** in the plot in which is visualized the world (the grid): select the cell in which zooming in and the cell chosen to be analysed in the table. You can click either in the complete grid, or in the zoomed one.



	Erbast	Carviz	Vegetob
t_population	17	0	35
t_energy	1151	0	0
m_lifetime	451.9	0	0
m_age	47.6	0	0
m_s.attitude	0.6	0	0

Figure 1: Simulation visualization

3 Main structure

Simulation

Starting from the "biggest" object of mine implementation we have the Simulation object, which has the following attributes:

- **world**: the actual world "Planisuss";
- useful variables for the methods, such as **pause**, **flag**, **zoom_row** and **zoom_col**;
- elements that constitute the figure, such as **GridSpec** and **axes** objects;
- the **FuncAnimation** object (the one that do the most of the work concerning the animation) and the **cids**, in which we connect the interactive functions.

World

Then we have the World object, the object which constitute the environment of the Planisuss, here we store the grid and the dictionaries that store the herds and prides:

- **grid**: a 3-dimensional matrix 3 x NR x NC, in the first layer we have the density of Vegetob or -100 (if there is water), in the second the number of Erbast in each cell, in the third the number of Carviz in each cell;
- **herds** and **prides**: two dictionaries that have as keys a tuple (i, j), which represent a cell, and as values a **herd** o **pride** object;
- **NR** and **NC**: number of rows and columns;
- **ground_cells** and **n_ground_cells**: a list of the cells in which we don't have the water, and the len of this list;
- lists of data for the visualization of the plots, such as **time_data**, **erbast_population_data**, **vegetob_density_data** and **erbast_energy_data**.

In particular we will focus in the **herds** and **prides** dictionaries, the data structures that contains the objects **herd** and **pride**. I have chosen this data structure since is fast for searching, and we can simply eliminates cells that became uninhabited and add new one. With this implementation we can simply iterate over the useful cells (the ones that are populated), without wasting time on empty cells, because in this dictionaries we have a pair key-value only for the cells that actually contains a herd or pride.

In the **grid** I have simply collected the number of animals of each specie in each cell. In particular for the Vegetob I have chosen to put -100 for the water and a number in [0, 100] for the ground, so that when I visualize the grid the blue channel is split in half (black for the water and blue/light blue for the Vegetob density).

Group: Herd and Pride

The herds and prides objects are collected in the dictionaries **herds** and **prides**, and are used for containing the list of animals present in the cell, and some useful data for the computation of the methods and the plots. Herd and Pride are defined as child class of the Group class, and they have the following attributes:

- **population**: a list of **erbast** objects (if we are in a herd) or **carviz** objects (if we are in a pride), that inhabit the cell in which is present the herd/pride;
- collections of useful data: **total_energy**, **total_lifetime**, **total_age** and **total_social_attitude**.

I have chosen to create the superclass **Group** so that I can write here methods that are commons to both **Herd** and **Pride**, thus taking advantage of the inheritance property I don't have to write multiple times the same code. For example the __init__ method is the same, so I have implemanted it only in the Group class.

Specie: Erbast and Caviz

The **erbast** and **carviz** objects are collected in the corresponding herd and pride via the list attribute **population**, they constitute the actual animals that populate the Planisuss world. Each of this object have the following attributes:

- **energy**: represents the strength of the individual, a int in [0, MAX_ENERGY];
- **lifetime**: duration of the life of the Carviz in days, a int in [1, MAX_LIFE];
- **age**: number of days from birth, a int in [0, self.lifetime];
- **social_attitude**: measures the likelihood of an individual to join to a group, a float in [0, 1];
- **moved**: a bool, True if the animal moved today, false otherwise.

I have chosen to create the superclass **Specie** so that I can write here methods that are commons to both **Erbast** and **Carviz**, exactly like I have done for Herd and Pride.

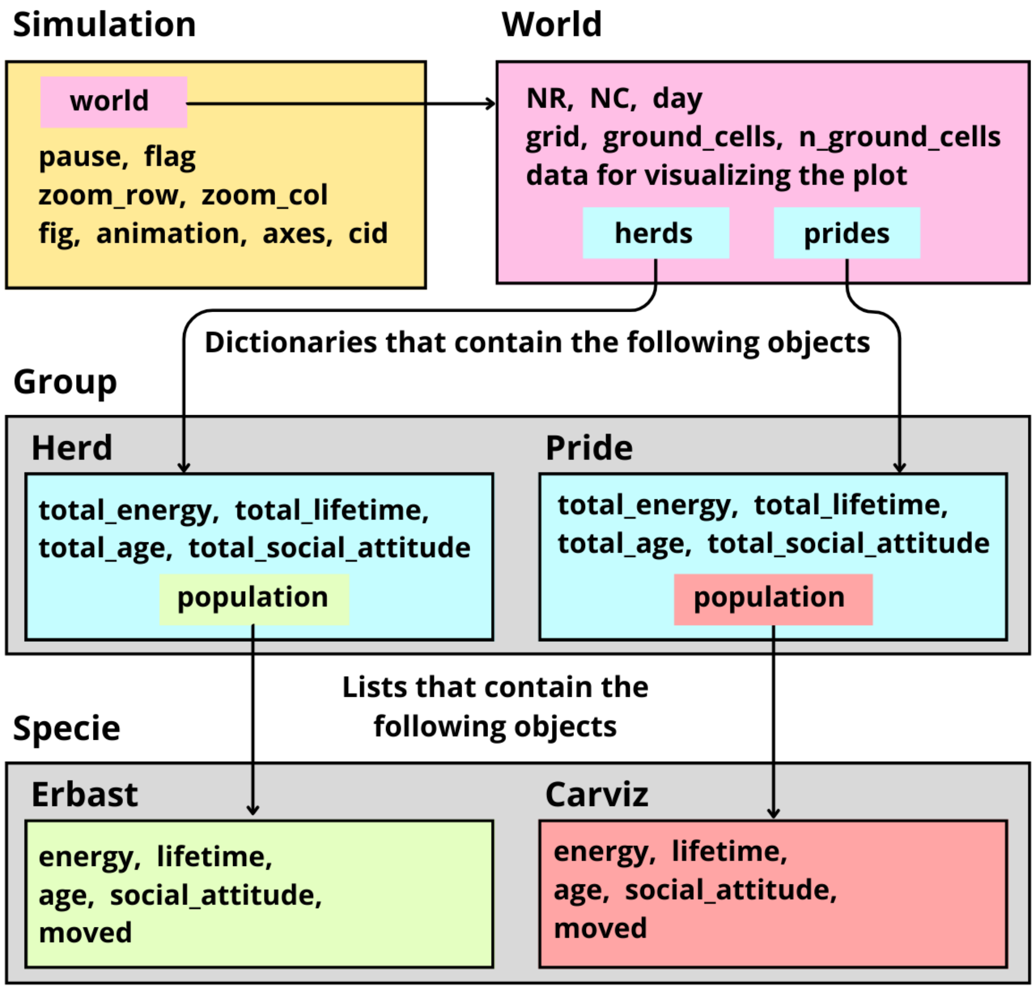

Simulation
world
pause, flag
zoom_row, zoom_col
fig, animation, axes, cid
World
NR, NC, day
grid, ground_cells, n_ground_cells
data for visualizing the plot
herds
prides
Dictionaries that contain the following objects
Group
Herd
total_energy, total_lifetime,
total_age, total_social_attitude
population
Pride
total_energy, total_lifetime,
total_age, total_social_attitude
population
Lists that contain the following objects
Specie
Erbast
energy, lifetime,
age, social_attitude,
moved
Carviz
energy, lifetime,
age, social_attitude,
moved

Figure 2: Main structure

4 Visualization

The main work is done by the FuncAnimation object:

self.ani = FuncAnimation(self.fig, self.update, frames=const.NUMDAYS, ...)

Here I specify the figure **self.fig**, the update function **self.update** and the number of frames **const.NUMDAYS**.

Figure

In the figure we have 3 **GridSpec** objects, the first that works as background (**gs_background**), the second that constitute the left part of the figure (**gs1**) and the third that constitute the right part of the figure (**gs2**). Each GridSpec object is divide in several parts, in each subplot we locate an **ax** object, that will host our plots.
gs1 is devided in 3 parts, in which we have the visualization of the Planisuss world, a subset of the total grid (useful for seeing more closely a certain part of the grid and for selecting more easily the cell in which we are interested) and a table, in which we visualise the details of the cell that we have clicked.

- **Grid**: the visualization of the whole Planisuss world, at each day of the simulation;
- **Zoomed grid**: a subset of the total grid (useful for seeing more closely a certain part of the grid and for selecting more easily the cell in which we are interested);
- **Table**: collect some useful data regarding the selected cell (tot_population, tot_energy, mean_lifetime, mean_age and mean_social_attitude of all the Erbast and Carviz in the cell, and the density of the Vegetob).

gs2 is divided in 5 parts and in each of them we visualize different plots, regarding the attributes of the animals that populate the Planisuss, starting from the top we have:

- **Species population**: plot of the total number of Erbast and Carviz, w.r.t time;
- **Histo vegetob density**: histogram that represent the number of cells of Vegetob, classified depending on the density in each cell;
- **Mean age and lifetime**: plot of the average energy of all the Erbast and Carviz, w.r.t. time;
- **Total energy**: plot of the sum of the energy of all the Erbast and Carviz, w.r.t. time;
- **Mean social attitude**: plot of the average energy of all the Erbast and Carviz, w.r.t. time.

Update function

The **update** function is execute at each frame of our animation, is stopped when **pause** is True and return all the axes object that we modify.
Is formed by 3 main functions:

- **world.a_day_on_planysuss()**: update the Planysuss world, we will se more in details about this method in the following section;
- **create_plots()**: update all the plots that are visualized;
- **display()**: display the new figure.

create_plots() is outside the method display() since the first is executed only when the animation go to the next frame. Instead display() is recalled also after a **onKey** or **onClick** interaction, since we update the display also when, for example, I select a cell to zoom in and see the details; or also when I want to see only the Erbast the display have to be updated directly after the click (not in the next day, after the update of the Planisuss, because I want to see the same instant that I was seeing before).

5 A day on Planisuss

Function that update the Planisuss world, is called in the update function of the FuncAnimation object and contains each method that constitute a part of the day. Most of this methods works by doing an iteration over the **herds** and **prides** vocabularies (for cell, herd in self.herds.items()), and applying a method to this objects, that do an iteration over the animals in the **population** list and apply a specific function.

__init__() of the World object

Initialize a random grid with a probability of 1/3 for spawning water and 2/3 for spawning Vegetob (with the corresponding density), while I create the grid I save also the cells in which I have the ground. Then we create a random number of herds and prides with a random number of Erbast and Carviz with random stats. Each of this number is generated randomly with the **random** library, between a min and a max, which are specified in the file *planisuss_constants.py*. The groups are spawned in a random cell, between the list of cells **ground_cells**.

growing()

We update the first layer of the grid (the one with the Vegetob density) by increasing each ground_cell by **const.GROWING**.

overwhelming()

Eliminate the animals situated in a cell that is surrounded by cells with 100 of Vegetob density.

movement()

We apply the **movement()** method to every herd and pride, that works by following the following steps:

- choosing the **best_cell** in the neighbourhood (the one with more nutrients);
- The group decide if they want to move, depending on the values of nutrients in the current cell w.r.t. the nutrients in the best_cell;
- Each animal will decide if they want to follow the group or not, the higher the social_attitude the more they will want to stay with the group, the lower the energy the more they will want to stay in the current cell (for not wasting energy);
- update the **moved** attribute of each animal, useful since only the Erbast that didn't move will graze;
- save the result in a list of tuples (group, cell), in which we save the group with the corresponding population and the cell in which is situated.

After the movement phase our dictionaries **herds** and **prides** have a slightly different structure, the values (instead of being herd and pride object) are lists of groups, since it is possible that two groups are in the same cell (after the movement).

unify_groups()

Now we need to bring back the initial structure of the dictionaries **herds** and **prides**, for the herds is quite simple, since we just unify the herds (sum the population lists). Instead the prides can choose to fight or to join with another pride. In the end only one pride will remain standing. Until we have only one pride, we sort the list of prides by increasing number of Carviz, then we choose the first two prides, they decide if they want to join or fight and then we go to the next iteration.

grazing()

For each Herd the Erbast that didn't moved today will graze, starting from the ones with less energy. 1 point of Energy is assigned for 1 point of Vegetob density.

hunting()

The pride identifies the stronger Erbast in the cell and combat with him. The Erbast is always took down and the Energy of the prey is shared by the pride individuals, increasing their energy value.

spawning()

We apply the spawning method to each herd and pride. Since it is the same for herds and prides, it is only defined in the super class Group. Here we update the age, decrease the energy (every ten days), and when the age reach the lifetime we spawn the offsprings.

remove_empty_groups()

We check if there is a group with 0 animals (it could happen for example after the hunting phase) and we eliminate it. Is useful to avoid dividing by zero for example when I calculate the mean age for the plot data.

update_stats()

Update the attributes total_energy, total_lifetime, total_age and total_social_attitude of each group and we update the attributes of the world object that works as data for the plot (such as time_data, erbast_population_data, carviz_population_data). This data are list of values, that represent a specific attributes at each time of the simulation.

neighbourhood(self, cell_row, cell_col, radius)

A function that gives the list of the cells neighbours of my cell with coordinates (cell_row, cell_col), within the distance radius. Is an auxiliare method that is useful for example during the movement phase.

6 Tools and resources

As IDE I always used Spyder 5.4.4, but the code can be run also in the IDLE without any problem. The following are the libraries and file that I have imported:
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import random

import Ghiotto_Alessandro_513944_planisuss_constants as const
from functools import reduce
from IPython import get_ipython
Further information such as the versions of the packages can be found at the file *Ghiotto_Alessandro_513944_requirements.txt*.
I have took all my inspirations from the file *2022_23_planisuss_v0.95.pdf* provided by the professor, following it quite strictly and adding almost nothing new. For the creation of the code I looked mainly to the matplotlib specification, in particular the pages regarding FuncAnimation for the animation, GridSpec for creating a complex layout, plots and histogram. While for the visualization of the grid of the Planysuss world and the functions for the onKey and onClick interactivity I have looked only at the useful resources given by the professor.

7 Conclusion

After several testing I think that the major problems are the decision functions of each phase of the day, that create a not very balanced simulation between the species, for example we arrive in a state in which we have an high difference between the total population of Erbast and Carviz. We usually find two situation, the first in which a pride doesn't encounter a herd and die of starvation, and the second in which a pride make contact with an herd and exterminate every Erbast. But overall I would say that the simulation create a reasonable dataset to be interactively visualized using matplotlib.

An other problem of my solution is that I don't have implemented the visualization of the trajectories, in my initial version of the project I have also that each herd and pride have the attribute visited_cells, in which I store every cell previously visited by the group. But I had to remove it since updating this attribute takes too much times, and increase with each day that pass.

The last problem that would be my most important objective for future improvements is the overall performance of the simulation, that is too slow for visualizing changes through a long period of time.

Thanks for the attention
Alessandro Ghiotto