PLANISUSS

Report of the Final exam project — A.Y. 2022/23

Computer programming

Marco Maria Mosconi 517462

INTRODUCTION

The strategy I decided to adopt was the one of simplifying as much as possible the project by deconstructing it and by keeping the functions as short as I could, creating many different files and organizing them in several folders.

The file where everything converges is *app*, where I created the *setup()* and the *main()* functions. There I called the five phases plus the *visualizing* file which takes the data for the plotting, unified in the folder *phases*. Each one of the phases iterates through the herds, the prides, or the cells, and calls other functions:

- if they are functions in common between erbasts and carvizes (like it happens in the *movement* phase) I put them inside a file of the folder *animals*,
- if they are specific for one of the species there is their folder, *vegetobs, erbasts,* or *carvizes*.

There is also:

- a *map* file, where I built the grid for simulation,
- a *keygenerator* file, that I used for the keys of the dictionaries in the setup of the herds and the prides,
- an *interact* file, where I wrote the interactive part of the project,
- a *parameters* file, where I built the class with all the parameters and their getters.

1. CELLS

The world is a grid of *Numcells x Numcells* cells. Each of them is randomly generated as type *Ground* or *Water* and with an x and y coordinate, hence for example the cell *c_0_0* will be of type *Water* (path: *cells/setup*).

The cells are stored in a dictionary *cells*, which has key equal to its coordinates (e.g., *c_0_0*) and as value the corresponding object of the class *Cells* (path: *cells/cell*). This is useful to identify the ground cells, since they are the only ones which can be populated by the three species.

2. SPECIES

2.1. VEGETOB

Vegetob is a vegetable species, the easiest one to implement since it spawns in every ground cell and cannot move, growing day by day of the fixed quantity *Growing.*

The vegetobs are stored in the dictionary *vegetobs* (path: *vegetobs/setup*), which has key equal to the cell the live in and as value the corresponding object of the class *Vegetobs* (path: *vegetobs/vegetob*).

This class has:

- As attributes:

    - *cell:* the cell the vegetob belongs to
    - *density:* an initial density which is randomly generated between 0 and *Max_density*.
- As methods:
    - the setter and the getter of density.
    - *grows()*: it makes the density increase without exceeding the maximum threshold.
    - *graze()*: since the Vegetob is the nutrient of the herbivore species, the Erbast, it decreases the density of the vegetob based on the number of erbasts in the cell and returns the value to be distributed among them. I decide to use the method *ceil()*, since *Growing* (and as consequence the density) can be a float value, but I wanted to return an integer one to simplify the *graze()* method of the herd, the social group of the erbasts.

2.2. **ANIMAL**

The two other species are the Erbast, the herbivore one which eat Vegetob, and the Carviz, the carnivore one predating Erbast. Erbast and Carviz have many characteristics in common, since they both move to find better living conditions, they can group together forming respectively a herd or a pride, and they have the same properties, hence I decided to make the abstract class *Animal* (path: *animals/animal*).

This class has:

- As attributes:
    - *cell:* the cell the animal belongs to.
    - *lifetime*: if it is None it is randomly generated between 0 and *Max_Lifetime*. It can be predefined, hence not None, if the animal is generated when its parent dies because its age reaches the lifetime value.
    - *age*: always set to 0. When it reaches the lifetime value it causes the death of the animal.
    - *energy*: if it is None it is randomly generated between 0 and *Max_Energy*. It can be predefined if the animal is generated when its parent dies because its age reaches the lifetime value. If the energy value reaches 0, the animal dies.
    - *socialAttitude*: float value between 0 and 1 randomly generated, apart from the cases like the lifetime and the energy where it is predefined.
    - *moved*: it is set to False and I used it to keep track of whether or not the animal has moved and consequentially:
        - in the case of the Erbast, if it can eat (the Erbast which moved cannot eat),
        - in the case of both species it avoids the issue of an animal moving twice: if an animal living in the cell *c_1_1* moves to the cell *c_2_1*, going on with the iteration among all cells, when I arrive to the *c_2_1* one, the animal which previously went there will have *moved* set to True and it will stand still.
- As methods:
    - The setters of the randomly generated properties and the getter of social attitude and energy.
    - *grows()*: it increases *age*, which makes energy decrease by *Aging* if it reaches a value multiple of 10, and sets the flag *isAlive* to False in the two cases for which the animal can die, which will be useful to generate the offspring and remove the dead animals from their social group.
    - *moves()*: if decreases the energy and sets *moved* to True
    - *willMove()*: the first “if” is necessary for the previously explained case of the *moved* attribute. I cared about the cases when the animal moves, so if it follows its social group’s decision of moving or if it decides to move despite its social group standing still:

- *moves* makes the animal follow its social group's decision of moving if it is set to True. This happens when its social attitude is greater or equal than *Min_Social_Attitude* and a random integer between 0 and *Max_Energy* is lower or equal than its energy, hence animals with bigger energy value will have greater chances of moving.
- *isStill* makes the animal move even if its social group doesn't if its social attitude is lower than *Min_Social_Attitude* and a random integer between 0 and *Max_Energy* is lower or equal than its energy.

I decided not to fix a minimum energy value below which animals don't move to avoid cases of entire social groups with low energy which couldn't and didn't move until all the animals forming them died.

2.3. **ERBAST**

The class *Erbast* (path: *erbasts/erbast*) is a subclass of the class *Animal*. There are no attributes added but just the method *grazes()*, which increments by 1 the energy of the erbast if it didn't move, otherwise it sets *moved* back to False so that the next day the erbast will be able to leave the cell.

2.4. **CARVIZ**

The class *Carviz* (path: *carvizes/carviz*) is a subclass of the class *Animal*. Like in the *Erbast* class, there are no attributes added but just one method, *eats()*, which takes *indEnergy* (individual energy), a part of the energy of the defeated erbast, as parameter and adds it to the carviz energy. The difference with the grazing of the Erbast species is that the erbasts always increment their energy by 1, while carvizes kill the erbast and divide all its energy among themselves, hence I could not make one single method for both species inside the class *Animal*.

3. **SOCIAL GROUPS**

Vegetobs and carvizes can group together forming a herd or a pride if they are in the same cell. The strategy I used was to initialize a herd and a pride for each ground cell of the world, even though they don't have any animal inside of them. In this way the advantage is that, even if it is the social group (herd/pride) which decides whether to move or not, in any case it stands still and the animals inside of it, if moving, are removed and added in target social group.

Since herds and prides has many characteristics in common, I decided to create the abstract class *Group* (path: *animals/group*).

This class has:

- As attributes:
  - *cell*: the cell it belongs to.
  - *animals*: the dictionary which contains the animals present in the social group.
  - *prevCells* and *currInCells*: two lists which are used for the memory of the social groups in order not to let them move in the cells where they have been the previous day.
- As methods:
  - Some getters which were useful for different purposes:
    - *getAvSocialAttitude()*: used specifically in the *struggle* phase.
    - *getTotalEnergy()*: used in the *hunt* and *fight* phases. .
  - *addAnimal()* and *removeAnimal()*: used in the *movement* phase, as explained before, and in the *struggle* phase when a pride joins another one.

- *kill()*: used in the *growing* phase in case the animals are overwhelmed by the Vegetob and terminated.
- *spawns()*: it is the complete *spawning* phase which is then iterated through all the herds and prides. For every animal in the group, it calls *grows*, which I explained before, and if the animal dies and there is enough space in the group (using *Max_Herd* or *Max_Pride*), it generates two children. The children's properties are defined such that:
  - *age* is set to 0.
  - *energy*: their sum is equal to the energy of the parent.
  - for the other properties their average is equal to the corresponding property of the parent.
- *move()*: given a target group and based on *Move_Probability*, it the group moves its cell is put in the *currInCells* list and every animal inside the group will or will not move based on the method *willMove* explained before.

3.1. **HERD**

The class *Herd* (path: *erbasts*/*herd*) is a subclass of the class *Group*. I only added the method *graze()*, which iterates through the sorted dictionary of the herd, makes the erbasts graze and decreases the social attitude by 1 if there is not enough food.

For the setup I created the dictionary *herds*, iterated through all the ground cells, added to the dictionary a random string as key and as value the herd. Then, based on *Erbast_Probability*, an erbast is eventually generated and added to the dictionary of the herd (path: *erbasts/setup*).

3.2. **PRIDE**

The class *Pride* (path: *carvizes/pride*) is a subclass of the class *Group.* I had to add more methods than I did with the class *Herd* because of the *struggle* phase, which was focused almost only on the Carviz species, like the *grazing* phase was with the Erbast species, with the difference that the *struggle* phase was much longer than the *grazing* one.

- The *eat()* method, as I explained before with the *eats()* method of the *Carviz* class, determines an individual energy which is given to the carvizes and then distributes the remaining energy starting from the ones with less energy.
- The *join()* method takes place when two prides decide to join in the *struggle* phase moving all the carvizes of a pride inside the other one.
- The *failHunt()* methods takes place in the *hunt* part of the *struggle* phase, decreasing the social attitude and the energy.
- The *winFight()* method takes place in the *hunt* and in the *fight* parts increasing the *socialAttitude*.

As I did for the erbasts, I made the setup by creating the dictionary *prides*, iterating through all the ground cells, adding to the dictionary a random string as key and as value the pride. A carviz was then eventually added to the pride based on *Carviz_Probability* (path: *carvizes*/*setup*).

4. **PHASES**

As I told in the introduction, the five phases contained in *phases* call the functions and the methods of the classes written in the files of the other folders.

4.1. **GROWING**

The function *growing()* calls:

- *grow()* (path: *vegetobs/grow*), which iterates though the vegetobs making their density increase.

- *neighbors()* (path: *animals/neighbors*), which iterates through herds and prides, uses *findPrey()*, indicating if the social group is surrounded by cells having the maximum Vegetob density, and eventually kills all the animals inside of it.
    - The *findPrey()* function (path: *animals/findPrey*) is also used later to identity the target cells where the herds and prides needed to move, but the useful part for this phase is where I initialize *density*, *neighbors,* and *killGroup* set to False, I iterate through the cells next to the current one, excluding the current one and the ones of type *Water*, I add their density to *density*, setting the flag *killGroup* to True is the total *density* value is the maximum.

4.2. **MOVEMENT**

In the *movement* phase I decided to make the iteration through the herds and through the prides in different ways.

For both cases it is based on the parameter *Neighborhood*, checking all the cells in its range, hence if for example the current cell is *c_1_1* and *Neighborhood* is set to 1, then the social group can move just in one cell next to it, but if *Neighborhood* is set to 2 the target cell could be *c_1_3* and the social group will be able to move directly there, jumping the cell *c_1_2*.

Erbast:

- For each herd, if it is inhabited, based on the Vegetob density the function finds the target cell with *findPrey()*, where also the cells from which the erbasts arrived the day before are passed, it identifies the herd inside the *herds* dictionary with that cell as attribute and moves the herd there. After the iteration is finished, another one is started so that the cells of the herds which moved in the current herd (*currInCells*) are moved to the list of the previous cells (prevCells) and are not considered for the movement the day after.

Carviz:

- The function iterates through the dictionary *prides*, it finds the target cell based on the Erbast energy, but instead of identifying the target pride inside the dictionary it creates another pride, moves the current pride individuals there, and adds it to *prides*. This difference is due to the fact that I needed to maintain all the prides separated for the *struggle* phase when they have the possibility of joining or fighting, so I could not unify directly all the carvizes like I did with the erbasts.

4.3. **GRAZING**

The function g*razing()* simply iterates through the herds and if they are inhabited it takes the available energy of the vegetob and give it to the herd to be distributed.

4.4. **STRUGGLE**

For the *struggle* phase by iterating on the dictionary *cells*, the function finds the prides which belong to the same cell and put them in the list *joiningPrides*, removing the empty ones if they are several, and then until only one remains it takes the first two prides of the list and based on the average social attitude of their carvizes they decide whether to join or not, fighting in case of negative response.

- In the *fight()* function (path: *carvizes/fight*) a random number is drawn and each pride has a winning probability proportional to the sum of the Energy on its components, with the social attitude of the ones of the winning pride which is increased.

When only one pride remains, it hunts.

- In the *hunt()* function (path: *carvizes/hunt*) the herd belonging to the same cell as the pride is identified and the erbast with the most energy is taken. The last blood scheme takes place, hence a random number is drawn and the probability of success depends on the value of the cumulative energy of the pride and the energy of the prey, and this scheme is repeated on and on until the erbast is took down or all the carvizes die, and every fruitless assault costs some energy to the pride and decreases the carvizes social attitude.
  When the pride takes down the erbast their social attitude increases and the energy is shared.

At the end, like I did with the Erbast in the *movement* phase, the cells in the *currInCells* list are copied in the *prevCells* list so that the next day the prides will not be able to move there.

4.5. **SPAWNING**

In the last phase of the day, *spawning*, the function just iterates through herds and prides calling the *spawns* method which I previously explained.

5. **PARAMETERS SETTING**

I set the parameters in order to find the right balance between the size of the grid, the duration of the simulation and the computational complexity.

The simulation takes place in a *20x20* grid, due to computational complexity if *Numcells* is set from 25 up it starts slowing down, even if it is still possible to run.

After a sequence of trials and errors I also found the right values for all the other parameters:

- I decided to fix *Carviz_Probability* as 0.8 while *Erbast_Probability* as 0.3, having way less carvizes than erbasts at the beginning in order to avoid the death of all erbasts in a very short period of time.
- *Growing* set to 0.8 is enough to make them survive during the days when the Erbast population is at its peak, but a bigger value would have made them too powerful by making them fill quickly all the cells overwhelming the other species.
- *Aging* = 1*, Max_Density* = 100, *Max_Energy* = 100*, Max_Life* = 30*, Max_Herd* = 10 and *Max_Pride* = 10 are values that I set at the beginning and never needed to modify, they made the simulation run in the right way.
- *Neighborhood*, as said before, is a parameter which if set to a value bigger than 1 would make the animals move by more than one cell at a time, so I preferred not to change it.
- *Min_Social_Attitude* and *Move_Probability* are two values which I initially set to 0.5, but this brought to the issue of herds and prides stuck in their cells mainly due to a lack of social attitude and a 50% chance of moving didn't help, so I decided to fix two lower values, respectively 0.4 and 0.3.
- The parameter *Numdays* set to 250 was just a consequence of all the other values: by making the simulation run multiple times I observed that the general behaviour is the one where the Erbast population reaches its peak between day 50 and 100, then there is very little Vegetob density left, hence they graze way less and the Carviz species takes over them. The Carviz population reaches its peak around day 100, when there are almost no Erbast left, and this brings to food shortage and their consequential slow death, accompanied by an exponential growth of the Vegetob species, which will then overwhelm the remaining animals around day 250.

A way of improving the project would be optimizing it in such a way that it is possible to run it and obtain a fluent simulation even with bigger grid dimensions.

6. **DATA VISUALIZATION**

The main part of the data is stored in the *map* file, where I built the grid with each pixel corresponds to a cell of the world grid.

I decided to make the map a little bit different with respect to what it was written in the instructions, without combining the species characteristics to make the colour of a pixel but keeping them separate in order to have a clearer idea of the situation at every moment.

I created four matrixes with the same size as the world grid:

a. Water/Ground cells, adding 1 if a cell is of type *Ground* and 0 if a cell is of type *Water*.
b. Density of the vegetobs, adding 0 if it is a Water cell.
c. Energy of the erbasts, adding 0 if it is a Water cell.
d. Energy of the carvizes, adding 0 if it is a Water cell.

I initialized the background map with the 1st matrix, defining Water and Ground cells with two separate colours, and then I decided to add the Vegetob, the Erbast and the Carviz species as patches:

- Vegetob: I added them as squares with green as colour, and their density which defined their size. Hence, for example, if a vegetob had size equal to *Max_Density*, the green square would occupy the whole pixel covering the Ground cell.
- Erbast: I added them as circles, fixing their size to a precise value which could not be changed, and with their energy determining as colour their shade of blue. Their position inside the cell is randomly generated.
- Carviz: I did the same thing as the Erbast, adding them as triangles with a fixed size and their energy determining as colour their shade of red. Their position inside the cell is randomly generated.

In the *app* file this map is put inside a for loop of the *Numdays*, so that at the beginning of each day with *plt.cfa()* it is cleared and then updated.

In the *app* file I also created 3 plots which appear after the end of the simulation:

a. The total density of the Vegetob day by day.
b. The total number of the Erbast population day by day.
c. The total number of the Carviz population day by day.

The data for the three plots are obtained from the *visualizing()* function (path: *phases/visualizing*), which returns the density and the two energies computed in the *visualize_V()*, *visualize_E()*, *visualize_C()* functions (path: *visualization/visualize*).

7. **INTERACTION**

Talking about the interactive part, I used two libraries: tkinter and tkhtmlview, which I discovered by watching the “Python GUI Buttons” tutorial by Bro Code on YouTube.

tkhtmlview was useful to create a homescreen which briefly introduces to the simulation, explaining its controls and the plots at the end. There is also a list of all the parameters with their present value since the first thing which can be done, apart from directly starting the simulation, is interactively changing them.

With tkinter I created several buttons:

- From the homescreen you have:
    - *startButton*: start the simulation. You can control it with:
        - *pauseButton*: pause the simulation. I used the flag *isRunning* which at the beginning of the simulation is set to True and by pausing it becomes False.

        - *resumeButton*: the *isRunning* flag is set back to True and the simulation resumes.
        - *speedupButton*: you can increase the speed until the pause time becomes 0.01 and the button is disabled.
        - *slowdownButton*: you can decrease the speed of the simulation.
    - *settingButton*: by clicking it you get the opportunity to change the parameters, in fact you'll get:
        - *listBox*: it is the list of the parameters, which once at a time can be selected.
        - *submitButton*: if you submit the select parameter you can change it with a *tkinter.simpledialog*. This is the reason why I put the parameters inside a class and created an instance, in this way inside the *submit()* function I could use *getattr()* and *setattr()* methods to obtain the value of the select parameter and set the new one.
        - *closeButton*: you can close the current window and go back to the homescreen.

A way to improve the project could be adding other controls, like zooming on a specific part of the grid while the simulation is running, or being able to stop it and go back. Other plots can be implemented, pointing out more specific information about single cells, or also a heat map which shows the areas of the grid which were mostly inhabited during the simulation.