Planisuss

Report of the final exam project

Computer programming, algorithms and Data structures -Mod 1

Artificial Intelligence 2022-2023

Aprile Giorgia, Cappa Nicolò

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.

2. Modules

The project is divided in multiple modules, and it try to follow the **object-oriented programming** paradigm.

Now each module will be discussed:

2.1 Settings

It holds all the global constants used by the various algorithms. The user can alter each constant to study what changes in the simulation. The current values of each constant were decided trying to maximise the length of the simulation without a major loss of population.

The NUMCELL constant define the dimension of the Planisuss world. Its value is set to 50 to balance the simulation computational cost.

2.2 Cell

This module contains the class Cell. This is one of the classes on which the whole simulation is based.

It is initialised by giving the coordinates as input and has two main properties: *cell_contents* and *cell_types*. Both are lists, the first one contains all the objects present in a cell, while the second just contain all the type of the object currently in the cell (the type is just a string variable), this is useful for checking the content of the cell by other methods.

The methods of this class are quite self-explanatory. The contain method just append to *cell_contents* the input object, while updating also the *cell_types*. The *remove_from_cell* method instead does the exact opposite: remove from both list the object and its type. This is done to keep the cell updated after each change.

The most crucial method is *get_color*: it computes and return a tuple containing the RGB value that the cell should be displayed with. Red channel is associated with carviz, the Green channel with herbast and the Blue one with the vegetob. Each channel value is computed by making a proportion with the number of creatures of each type contained in the cell and a max value. Each channel value is then combined with the others and returned. Empty cells return white, while water cells return a specific RGB combination (the computation are designed in a way to avoid having values outside the normal RGB range i.e., each value is between (0,0,0) and (255,255,255)).

For the water cells, a specific set of values has been used: (51,153,255) since its very unlikely that cells end up having this exact same RGB combination.

To compute the number of each "inhabitant" a simple algorithm has been used: the algorithm iterates through the *cell_content* list and when a specific type of object is detected it update the associated creature counter accordingly. For groups, it adds up the length of the *group_memebers* list, while for individuals it adds only 1.

2.3 Animal

This module contains two classes: *animal* and *social_group*. They will be later used to create more specific classes. Those classes have been created to avoid redundancy while coding the more specific groups, since most of their basic properties are shared. This also helped in the debugging of the code.

2.3.1 Animal Class

It has all the useful properties characterising an individual like: x, y positions, energy level, age and so on. Important properties are intelligence and social attitude, those will be used later to make decision and will dictate the behaviour of larger groups.

This class contain only two methods: *surroundings* and *its_alive*. The first method was only used during the early stage of development when the world was populated exclusively by individuals. In the final code the surrounding method is implemented by social groups. This method was not removed in order to allow some flexibility in further revisitation of the project.

The *its_alive* method simply check if the individual satisfies the condition to be considered alive, like age or energy, and then return a Boolean value accordingly.

2.3.2 Social group Class

The main property of this class are x and y coordinates, the id with which each group is identified and the *group_member* property that is a list of individuals composing the group. Properties like *average_social_attitude* and *average_intelligence*, holds information about the combination of individual members properties.

Now the most relevant methods of this class will be discussed.

The class has various methods that are used to compute the mean or total value of their member properties. Each value is then assigned to the corresponding class property and returned. This is done to keep all the class properties updated.

The *its_alive* method, work similarly to the one of the *animal* class but with different conditions: this method iterate through the *group_member* property and check if each individual is alive using their respective *its_alive* methods. Dead members are then removed from *group_members*. After this process if the *group_member* property is empty the group is considered dead. The method returns the corresponding Boolean value.

The *compute_group_surroundings* method, simply creates a list in which are stored all the cells immediately close to the current cell. This is an important method that allows the groups to see and make decisions accordingly. An interesting project would be to devise distinct types of ways in which the groups can "know", such as using scout members or communicating with friendly groups. These possibilities have not been explored in this project because they would have required a significant amount of time, which would have been taken from the development of the most important feature of the project: the visualization of world information. Nevertheless, it remains a captivating development idea worth considering in future revisitations.

The *life_cycle* method, is a simple but truly important one of this class. After each call of this method all members of *group_members* will have their age increased by one. If certain conditions are met than they die and reproduce or just lose some energy.

One interesting observation is that when spawning new members, the function used is from the parent class of the *social_group* class. Therefore, each superclass of *social_group* must have a *spawn_new_member* method, or it will raise a 'method not found' error. While some may perceive this as a questionable design choice, it reduces redundant code and simplifies the implementation for classes based on the *social_group* class. And this simplification was the purpose of the whole animal module. This method is also one that could be tuned and changed to make the simulation even more "organic".

The *Morale_check* method, is the last interesting method of this class. It simply increases or decrease some properties of the social group members according to their morale, and since this is a simple world, their morale is strictly based on their energy level. This method has only the merit of making the simulation even harder to balance, meaning that it increases the noise in the simulation by a lot. There will be phases of great morale and phases of extremely low morale, impacting in significant ways the simulation state. Despite all this the feature was kept because it gives the simulation a more "organic" feeling.

2.4 Vegetob

This module contains only the *vegetob* class. This class has simple properties, the most important of them is *cell_density* that regulated the growth state of the vegetob. The cell density has a maximum value.

This class has 2 methods: *its_alive* and *grow.* The *its_alive* method work like the one of the *animal* class, returning False if the cell density is lower or equal to 0. The *grow* method when called, simply increase the cell density by 1, if possible.

This is a simple class, but it constitutes one of the three main entities of this simulation. That is why it was considered necessary to use a separate class dedicated to it.

2.5 Herbast

This module is made of two classes, *herbast* and *herd*, which are fundamental components of the main simulation. The *herbast* class represents individual animals with specific characteristics, while the *herd* class represents social groups of *herbasts* that exhibit collective behaviour.

Both classes inherit properties from other classes: *herbasts* inherits from the *animal* class, and *herd* inherits from the *social_group* class. This inheritance allows them to access essential methods from their respective parent classes, ensuring an easier implementation and proper functioning of the simulation.

The animal module, on which both *herbast* and *herd* rely, plays a vital role by providing shared methods and functionalities. By consolidating these common features within the animal module, the code remains simple, contributing to the smooth and efficient execution of the simulation.

2.5.1 Herbast class

The 'herbast' class only contain the *type* property that is set to "herbast". This property will be used in all the entity classes of the simulation in order to get the nature of the object without using the python condition: *isistance(el, obj)* that far too python specific.

The only method contained in this class is *eat_vegetob* this is a method used in the early phases of development that was not deleted for the same reason of the surrounding method of the animal class.

2.5.2 Herd class

An important remark both for herds and prides, is that when initialised a positive integer number of members need to be passed as input, and each class will be populated by a set of members according to that number.

For this class will only be discussed the important methods that are essential for the simulation to work.

Each social group before deciding if they want to move or not compute the appeal of the surrounding cells, this is done by the method *compute_surrounding_cells_appeal*. This method simply iterates through each cell in the surrounding and compute for each cell the appeal, it then appends to a list the tuple having this structure (appeal, x, y) in order to identify the position of each cell with their corresponding appeal. The appeal for herd is compute like this: +1 for each *vegetob_density*, +1 for each *herbast* (herbast are designed to be more social than carviz) and -2 for each *carviz* in the cell. After computing the appeal, this method makes sure that the group avoid counting itself in the appeal computations. The list containing the tuple is then sorted by the appeal and returned.

compute_surrouding_cells_appel is one of the few methods allowing herds to make "intelligent" movement decision. This is a truly simplistic solution, but it was deemed decent enough for reaching the goal of the project. Other solutions could include simple neural networks, or just a set of more precise rules to compute the best cells in the surroundings. In this case it was kept like this both for simplicity and computational cost, since is an easy value to compute.

Two other important methods of this class are *decide_to_move* and *decide_to_split*. Both return a Boolean value according to the decision that the group took. In both cases the decision is took using weighted probability. The weights are computed based on some parameters like energy or social attitude of the group. Outside of the decision there are other factors, like current cell appeal or average energy, which can influence on the final group decision by a lot. A herd may for example decide not to move, but they are in a negative appeal cell, then they are forced to move anyway. This is done to make the world less static, otherwise herd would be far too stationary, and since carviz do not have a fully developed "brain", they would have a hard time finding those static herds.

The *herd_graze* method enables the herd to graze on the vegetob present in a cell. Each food portion is considered equal to 1, and this portion is then reduced from the vegetob cell density. Each member of the herd attempts to eat once during this process. If there is enough cell density in the vegetob for all the members, each one can eat.

However, if the vegetob has not enough cell density to feed all members, the herd faces a decision-making scenario. This decision is based on a weighted probability calculation, where the weightings are influenced by the group's average intelligence. If the herd makes an intelligent decision, they will feed only the members with lower energy levels, leaving a portion of the vegetob cell density intact.

On the other hand, if the herd decides not to make an intelligent choice, they will feed the members with less energy until the cell density reaches zero. This is considered a non-intelligent decision, since in doing so they “kill” the vegetob, also reducing their change of survival as a specie.

As a result of this grazing activity, members that have successfully eaten will receive an increase in their social attitude, encouraging positive group dynamics. Instead, members that were unable to eat due to limited resources will experience a decrease in their social attitude. So well fed groups are more inclined in staying together while hungry groups prefer to make individualist choices.

The last method worth discussing is *spawn_new_members*. This method simply, spawn the offspring of a parent, if the group has enough space to accommodate them. Each parent generates 2 new members, each members properties are based on the one of the parent. This is an essential method since entities in this simulation can die because of aging.

This was also an attempt to study the evolution of properties across generations. To do this in a significative way, more properties were necessary, properties that would have made the simulation even more complex. For this reason, having some small fluctuations on the social attitude and intelligent was considered enough.

2.6 Carviz

This module is made of two classes, *herbast* and *herd*. Like in the Herbast module, both these classes are based on the *animal* and *social_groups* classes.

This is the module containing the information about the third and last type of entity of the world, the carnivores. Both classes are like the one previously described in section 2.5 so they will be discussed only were there are major differences.

2.6.1 Carviz

This class has the same type of properties of the herbast class. It was created for easing the implementation of the different entities in the first development phases. In the final code both carviz and herbast class could be merged into a single class, but to allow some flexibility this has not been done.

The only interesting method of this class is *eat_herbast*, like the herbast’s *eat_vegetob* this is not used in the final simulation but has been left to ease future implementation of the code.

2.6.2 Pride

This class has an important property called *leader* that will be used later during the fights among prides or in the hunts for herbast. The leader is just the member with the most energy. This has been done to make a more interesting fight system. The most important methods will now be discussed.

The *compute_surroundings_cells_appeal* work exactly like the herds ones, but the appeal is computed in a modified way: +1 for each herbast in the cell and -1 for each carviz in the cell. Since carviz have been designed to be a not very social species, they try to avoid each other, if they end up in the same cells they may fight.

Other properties like *decide_to_split* or *decide_to_move* are similar to the herd ones.

An interesting observation is the absence of a dedicated method for hunting, unlike the graze method available for herds.

This is a consequence of a deliberate decision to move the hunting functionality into a separate module called *battlefield*. The hunting process involves intricated interactions between two entities, making it more complex compared to grazing. To address these complexities and keep a modular design, the hunting functionality was organized in the *battlefield* module. This also helped by enabling a more organized and efficient implementation of this critical aspect of the simulation.

2.7 Battlefield

This method only holds the *battlefield* class. This class has two simple properties: the x and the y coordinates. The especially important thing of a *battlefield* are its two methods: *fight* and *hunting_ground*. Since they are important and rather complex, they will be discussed separately.

2.7.1 Fight

A fight only takes place among prides. When calling the method, a list holding the fighting prides need to be given as input. Each fight involves only 2 prides at a time. To do this a new list called *fight_now* is created, this list holds the two prides that will fight each other. The fight in the cell ends only when there is only one pride left in the *fighter_list*.

The leader of the two prides duels each other until one of the two dies. The duel is particularly simple: a number between one and the leader energy is picked randomly. Then this number is removed from the opponent leader energy. The winning leader is rewarded with a small boost of energy, while the losing leader is removed from its pride. If the two leader ends up in a **draw**, all the fights in the cells will be suspended, since it is a rare event and all carviz are astonished by what happened.

After each iteration, the *fight_now* list will be recreated, and new leader assigned. This result in a messy fight style, like a brawl, this ends only when one pride remains in the *fighter_list*.

This method was devised by mixing the leader fight idea and the chaos of a brawl, since carviz were imagined as an aggressive species. This fighting methodology should not be considered the “right” one by any means, it is just a design choice, took by interpreting the project specifications.

2.7.2 Hunting ground

This is one of the core methods of the simulation. A hunting ground needs a prey and a predator. Those are input given when calling the method. In the final code, the prey are herds while predator are prides.

Now the algorithm that this method uses will be broken down and explained.

Before even starting, the method checks if any of the two groups received as input has no members. If it is the case the method is at once returned and stopped. After that two useful variables are created, one called *energy_gain* that keep track of the energy collected while hunting, and the other is *pride_need* that keeps track of the total energy needed by the pride. Once the pride has all the energy that it needs, it will stop hunting.

Now the core of the algorithm will be explained. Each member of the predator group tries to hunt all the member of the prey group. This is done through simple iterations. If both prey and predator exist (are not *None)* then the hunting take place.

A modifier is computed for both the prey and the predator. This modifier is then added to a simulated throw of a D20 (iconic D&D dice). Obviously, the predator is more likely to win. But if the prey wins, the result of the prey throw is subtracted from the predator energy and the defeated predator cannot fight anymore. If the predator wins instead the energy gain is increased of a value equal to the prey energy, which is then set to zero. If the energy gain is greater or equal to the *pride_need* the hunting phase is over and the algorithms continue distributing the energy among the predators. Otherwise, the hunt keeps going until either all the predators have attacked, or all the pride members are dead.

After the hunting phase, the pride leader is reassigned (the leader is the member with more energy). The leader always tries to eat until they are full. The leftovers are then divided equally among the other pride members. Each member cannot eat more than their maximum energy. If a member cannot eat its whole portion the rest of the energy will be left for others to eat.

After the eating phase all the members that have ate have their social attitude increased.

2.8 World generator

This module contains the last class used in the simulation: *world_generator*. This class is used for building the starting world grid. To do so, it requires just two properties: the width and the height of the world. Planisuss is a square grid world, so both properties are equal, but as always, to have a more flexible simulation, they were left separated.

The starting world grid is then initialized as a NumPy array holding in each position of the array a cell object with the corresponding coordinates. The dimension of the array is NUMCELLS x NUMCELLS.

Each cell is then filled by the various methods of the class following those guidelines: border cells hold water, random cell in the world grid holds water. Then all the cells that are not water are set as ground cells. First in each ground cell a vegetob can spawn, after that in all cells were there is no vegetob all the other creatures can spawn. The spawn of each creature is controlled by some random variable, to give more unicity at each generated world.

An interesting method of this class is *generate_pool*. This is a method has a chance to fill with water cells that are next to a water cell. This was done in the hope of building a more interesting landscape. This method could be explored further to achieve a genuinely interesting ground generation. For example, if there are enough water cells close to each other they can be considered a lake, or an ocean, with interesting properties, and there could be entities capable of living only under water further enriching the world.

2.9 world plotter

This is a support module; it holds different methods useful for plotting the main grid. It is not strictly essential, since it is short and could be implemented in the main file, but it was left in a separate file in the hope of simplifying the main module.

It has only three methods: *initilialise_plot*, *build_plot_array*, *plot_the_world*.

The first method simply creates a matplotlib figure that is then used to plot the world.

The *build_plot_array* method takes as input the world grid and iterating through its cell, calling the *get_color* method and storing the RGB tuple in the corresponding array location. It then returns the whole array.

The animation array is created outside the methods to avoid having to recreate it at each call of the method. This can be considered a poor design choice since if the module is imported then the array is created by default, and this could raise some errors or conflicts. Anyhow this has been done to optimise, even slightly the computational cost of running the simulation.

The *plot_the_world* method is the one called by the main module after each day. It is responsible for calling the *build_plot_array* method for updating the array and then plots it.

2.10 Event logger

This module serves as the final supporting component of the project. Initially, it was designed to handle the saving of all world information and plots using the pickle library. However, implementing this approach introduced significant constraints in the object-oriented architecture. The fact that for saving all the necessary information was needed to pass an excessive number of variables as input complicated the overall design.

As a result, the decision was made to separate the saving functionality, leaving the actual saving of information to the main module. The primary focus of this module shifted towards managing the plotting of the saved information. This adjustment allowed for a more clear and flexible object-oriented structure.

Within this module, two key methods were kept: *write_event_to_file* and *save_simulation_state*. The *write_event_to_file* method takes a string input and writes the event into a dedicated text file, effectively serving as a log of major world events. This feature provides a valuable record of relevant occurrences throughout the simulation.

On the other hand, the *save_simulation_state* method uses the pickle library to save the world grid object. By capturing the world grid and the current day, this method effectively stores the entire simulation state. Its relative simplicity and minimal input requirements justified its placement within this module.

Overall, this supporting module plays a crucial role in ensuring that the simulation's essential data is preserved and that noteworthy events are documented for later analysis and review.

2.11 Main

This core module stands as the principal component of the simulation, responsible for managing all other modules. The entire architecture of the project was carefully designed to keep this module with only the essential functionalities, aiming to keep it as simple and efficient as possible. Despite investing considerable energy and resources in the object-oriented design, the module stays complex and extensive. Nevertheless, efforts have been made to break it down into its main methods to supply a clearer view of its organization and functionality.

Now its principal functionalities will be discussed and hopefully made clearer.

The simulation is represented thanks to the matplotlib *FuncAnimation* method. This mean that the simulation is visualized as an animation. The time on Planisuss is divided in days and each day is strictly connected to the animation frames. This means that there is no while loop, or other iterative solution, each time the main animation moves forward of one frame, it calls the *planisuss_day* function start starts a new day. The strict correlation between the simulation speed and the animation speed was used to avoid the simulation running behind the animation, in this way, at each frame of the animation correspond a Planisuss day.

Of course, this strategy has several drawbacks like the fact that the simulation speed is not constant. It depends on the number of computation that each day require (some days could run faster, other slower, according to what happens in each day). But after a careful evaluation, this feature was left, because the problem of the animation being faster than the simulation was considered more relevant than the identified drawbacks.

The method *planisuss_day* is the core of the simulation. Each day the day is increased by one, and each 100 days the simulation state is saved. More importantly it is responsible for calling the *world_population_iteration* method.

This method iterates through the all the world grid making each creature do a set of defined actions. But before that this method also check and merge all the “pridless” carviz present in the same cell in a pride and all the “herdless” herbast in the same cell in a herd. This is done to alleviate the computational cost of having multiple entities in the simulation. This merging function could have been moved into another method to make it clearer, but iterating through all the simulation is quite expensive and so it was kept all in the same iteration of this method.

Before making the creatures do their thing, this method also check if each creature is alive or not calling the *cleanse_the_death* method, dead creatures should not be able to do anything. The *world_population_iteration* methos also check for carviz and herbast even if they should not be part of the simulation as individuals anymore (they have been merged into groups). This was also kept to have a more flexible code for eventual future changes.

Each creature has a predetermined set of action that must do each day. Each creature act once at a time, until all the grid has been iterated through and the next day can begin.

An interesting feature added is the ability of vegetob to expand to near cells if certain conditions are met. This was done to try keeping the vegetob population at a constant level, since herd will eventually wipe them out. Their reproductive rates are slow though.

If two pride are in the same cell, they decide if they want to fight or simply merge. Accordingly to their intelligence level and social attitude. If two herd are in the same cell they always try to merge.

Each creature computes their surroundings and their surroundings cells appeal. Then they decided if they want to move or not in the best cell in the surroundings. After the group took a decision, each member decide if they want to follow the group in its decision or split from it. All the members that decide to split form a new group.

Sometimes creature may move randomly without carefully evaluating their surroundings. This is done to avoid having to compute in the middle of the simulated day all the properties necessaries for taking a decision for newly generated groups. Individuals also move randomly since their decisional capabilities are lacking.

After moving each individual lose one point of energy.

If a group did not move, then it can eat. If it is a herd it grazes, if it is a pride it hunts. Hunt and Graze only take place if the average energy of the group is below a certain threshold.

This ends the simulation bit of this module. All other methods are responsible for plotting the various graph with matplotlib. Those graphs show the real time evolution of various properties like average intelligence, or vegetob density.

Each graph is an animation that plots a list containing all the days over a list containing the value of different properties of each day (there is a one-to-one corresponding between a day and the associated property value). This solution for plotting was used, since after doing some research, this was the most practical and simple solution found.

The simulation can be controlled by the user mainly pressing space, pausing the simulation. Or clicking on the main grid and showing the population trend of a singular cell.

3. Testing and results

The simulation work quite fluently, even if sometimes the computational complexity rises due to many things happening at once. This could cause the simulation to crash, but there is an automatic saving feature that was devised just for those eventuality.

Overall, the hardest challenge among all, the one that can be considered failed, is the balancing of the word over extended periods of time. The simulation work with a large number of entities and then they mostly die in the first 500 days.

The fact that the simulation starts with many more herbast than carviz is because the MAX_PRIDE_SIZE properties is half the MAX_HERD_SIZE properties. This is because usually carnivores' prides are less numerous than herbivores herds.

The fact that the simulation is hard to balance over extended periods may be due to the number of complex features added. Those feature helps into making the simulation fell more "real", but they also make it a balancing nightmare.

4. Resources and libraries

Great inspiration for this project was took from many different entity based simulations mainly by: "Wa-Tor", "Conway of life" and "lux" .

The project was developed following those guidelines: "22_23_planisuss_v0.95"

This project has been coded using python 3.11.3

The NumPy version used is: 1.24.3

The Matplotlib version used is: 3.7.1

5.Conclusion

Overall, the project has been explored extensively. There may be a thousand more improvement to make in a simulated environment like Planisuss. This could be considered one of those never-ending projects: when you add something, at least other five unique features come to mind.

Developing this project presented many different challenges, and even if not in the best, more elegant or optimal way, they have been mostly overcomed. Much has been learned whilst building this interesting world.

Even if it cannot be considered "concluded" the project was considered ready to be presented. And so, in the hope that this report may help to clarify the code that should comes alongside we can consider this report concluded.

Welcome to our Planisuss!