4CCS1PPA Assignment III Report

Tetruashvili, David Simon
`david.tetruashvili@kcl.ac.uk`
1733299

Prabakar, Yeshvanth
`yeshvanth.prabakar@kcl.ac.uk`
1739681

Tsanova, Emiliyana
`emiliyana.tsanova@kcl.ac.uk`
1724310

February 23, 2018

Overview

This simulation simulates the populations of given actors according to a multitude of factors including the environment, the specific traits of each of the actors and the interactions of actors amongst each other. The current configuration simulates the populations of 5 animal actors mostly from the Mesozoic era. Note that due to our extension tasks, this may be easily reconfigured.

`Actors` by default age, reproduce and search for a food source. However, these behaviours may be altered and affected by the current state of the environment such as weather and time of day. Also, `Actors` may be created with randomised ages and other type specific characteristics.

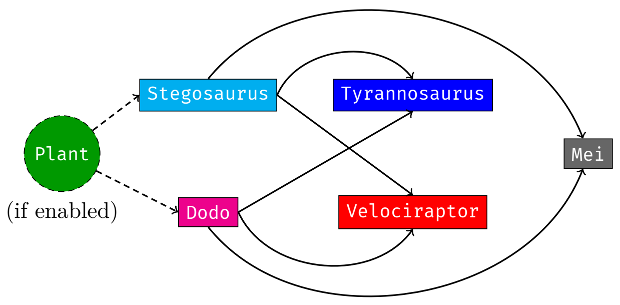
It is important to note that even though a **`Plant`** class does exist and is functional within the simulation, in the current configuration, it is not spawned in the field for population balancing reasons.

The five animal `Actors` are as follows. The **`Dodo`** lies at the bottom of the food chain and is, therefore, the prey. In the current configuration, the `Dodo` assumes a constant and sufficient flow of food, i.e., it does not die of hunger. The `Dodo` is abundant and reproduces quickly. The `Dodo` is represented by magenta.

The **`Stegosaurus`** is also a prey animal, as with the `Dodo` it does not hunger, however, is can only breed while the weather is clear or rainy. The `Stegosaurus` competes with the `Dodo` for space, as it reproduces nearly just as quickly The `Stegosaurus` is represented by cyan.

Both the `Dodo` and the `Stegosaurus` reproduce sexually, meaning that two `Actors` of the same species but the opposite gender have to be adjacent to each other for reproduction to take place.

Figure 1: Food chain in current configuration.



The **`Velociraptor`** preys on both the `Dodo` and the `Stegosaurus`. It is nocturnal, and therefore only breeds and hunts or moves during the night. The `Velociraptor` is represented by red.

The **`Tyrannosaurus`** is also nocturnal and preys on the same species as the `Velociraptor`, making it the `Velociraptor`'s main competitor. The `Tyrannosaurus` is represented by blue.

Finally, the **`Mei`** just as the two predators above, preys on both the `Dodo` and the `Stegosaurus`. It is the only `Actor` that is diurnal. The `Mei` is represented by grey.

Breeding of each `Actor` in the simulation has been set by their position within the food chain. Eg. The Dodo will reproduce the most while the Mei will reproduce the least. The same has been done with all other `Actor` traits.

Extension Tasks

Plants are implemented via the `Plant` class which extends the abstract `Actor` class. The `Plant`'s lifespan depends on its maximum age and water level.

`Plant` instances do not ever move to a new location after being created at one. `Plants` are made to react to their environment: While it is raining, `Plants` cease to spread, and start to absorb the rainwater increasing their water level up to a maximum. A `Plant`'s water level decrements each step that it spends alive while it isn't.

A `Plant` may also reproduce asexually during those steps. A `Plant` may die of old age, dehydration or disease which they may be born with but never spread.

Weather, along with the time of day was implemented via the newly created `Environment` Class. This class contains only static methods and lacks a constructor as it is not necessary.

The `Environment` class tracks the number of steps elapsed since the beginning of the simulation and calculates whether or not the current step count falls within the first or second half of a set length of one day/night cycle using merely a modulo operation, and sets the current weather field appropriately.

The `Environment` class generates the number of steps for which each weather condition persists at a time. That number is decremented each step. When it reaches 0, a new weather condition is chosen from the set of possible weather conditions (which are 'cloudy', 'rainy', or 'clear') excluding the one that has just ended; this eliminates the same whether condition from persisting for too long.

Each step a call is made to the `updateEnvironment()` method to update the weather condition and time of day if need be.

Adjustments were made to the GUI so that the time of day is displayed as a text on top of the simulation and the weather beneath. Additionally, the background colours of those GUI elements were modified to represent the specific time of day or weather.

The disease is implemented as a flag within the `Actor` abstract superclass. Hence all `Actors` can either be healthy or sick. Any given `Actor` may be created diseased, either at the first step of the simulation or as a newborn during it, both with their respective probabilities.

Diseased `Actors` are incapable of reproducing or seeking food. They die either of hunger or after taking a set number of steps after becoming diseased, whichever comes first.

Diseased `Actors` can spread the disease to other healthy `Actors`, independently from their species.

Within the GUI, diseased `Actors` are represented by a darkened colour.

Each direct subclass of `Actor` (e.g. `Animal` or `Plant`) contains an implementation of a default behaviour mentioned above via the `act()` method. Such default behaviour is in its most general form. For instance, both hunger and age increment each step regardless of the time of day, or that they die from overcrowding by their kind. Depending on the requirements of the simulation, subclasses to such superclasses may override this method, essentially introducing new behavioural patterns. For instance, behaviour affected by the current time of day for an `Animal` has been implemented in the `AnimalTimeSensitive` subclass, where a setting is set deciding whether the `Animal` is nocturnal or diurnal. Weather sensitive behaviour has been implemented similarly.

For ease of testing and configurability, the `Config` class contains the methods to first load and then read property-value pairs specified in the `config.cfg` file (which is located in the source directory) via the Java Properties API. As this class is used throughout the program, and only a single instance is necessary at run-time, it uses the singleton design pattern. The `config.cfg` file itself contains most significant parameters for the configuration of the simulation and the `Actors` within it. For example, the property `simulator.DELAY` sets the delay between consecutive steps of the simulation to a specified integer millisecond value.

The `getProperty()` method retrieves a value associated with a passed property name.

As the return type of this method is a string, in most cases it has to be parsed into a usable numerical format.

Via this functionality, it is possible to easily reconfigure most aspects of the simulation. For instance, the length of one day/night cycle or the seed to the `Randomizer` class. However, the most powerful usage of this functionality is to manipulate the initial populations of all `Actors` within the simulation.

We encourage the modification of lines 18 through 23 in `config.cfg` to the following to disable all `Actors` except for `Plants`.

config.cfg
```
mei.CREATION_PROBABILITY = 0
dodo.CREATION_PROBABILITY = 0
stegosaurus.CREATION_PROBABILITY = 0
tyrannosaurus.CREATION_PROBABILITY = 0
velociraptor.CREATION_PROBABILITY = 0
plant.CREATION_PROBABILITY = 0.05
```

If you wish to revert this change, the original value for those lines were.

config.cfg
```
mei.CREATION_PROBABILITY = 0.04
dodo.CREATION_PROBABILITY = 0.12
stegosaurus.CREATION_PROBABILITY = 0.12
tyrannosaurus.CREATION_PROBABILITY =
↪  0.04
velociraptor.CREATION_PROBABILITY = 0.04
plant.CREATION_PROBABILITY = 0.0
```

The above functionality is made possible by the abstraction of the initial population generator that has been implemented within the `Simulator` class to a newly created `PopulationGenerator` class whose sole responsibility is to randomly generate the initial populations of all `Actor`s implemented in the simulation based on probabilities specified in the `config.cfg` file.

We have achieved to make this class functional regardless of the number of `Actors` at play, via the Java Reflections API. This allows `PopulationGenerator` to call any implemented constructor within any concrete subclass of `Actor`.

However, this hinges on the presence of a `CREATION_PROBABILITY` configuration property specified for each `Actor` at play, (i.e., if the `config.cfg` line containing `dodo.CREATION_PROBABILITY` is missing, no instances of the `Dodo` class will ever be created).

The program was also additionally modified to halt when the window is closed.

Bugs and Limitations

One limitation of this simulation is that both the male and female `Actor`s of a species can give birth to offspring in the same step, where in reality for gendered reproduction only the females should be able to give birth.

The implementation of the disease is less extendable than other constructs in the simulation. Here we have decided to limit the disease to be singular and affect all `Actor`s, and by doing so have hard coded this functionality into both the `Actor` superclass and its subclasses. Perhaps a more extendable approach would've been to create a `Disease` class.

As Java does not support multiple inheritances, our approach to adding custom actor behaviour has a distinct flaw. Even though the solution is extendable in the sense that more behaviours may be added easily through the addition of behavioural superclasses, if per se, an `Actor` is required to exhibit two or more of already implemented behaviours a new superclass must be implemented for this new combination of behaviours.

The final major limitation of this simulation is that no two actors may inhabit the same location simultaneously, as many methods hinge on there may only be a single actor per location.

We have looked at possible solutions, the best of which would be to reimplement location handling in the current class structure and have the `Field` class, instead of holding an array of `Actor` instances, hold an array of so called `Cell` instances. These array of `Cells` would then hold references to individual `Actors`.

This is a more expandable approach than, say, adding a new field to the simulation, and that would restrict only at most two kinds of `Actors` to be coincident. Not to mention, the added concurrency limitations.

The `Cells` could be extended via inheritance. For example, differentiating between habitable and inhabitable cells.

Though this approach is attractive, we have not been able to implement it within the given deadline.