BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION MATHEMATICS AND COMPUTER SCIENCE

DIPLOMA THESIS

An Ecosystem Simulation Model Inspired by Genetic Algorithms, Natural Selection Rules and Ant Colony Optimization Algorithms

Supervisor
Conf. dr. habil. Lisei Hannelore – Inge

Author
Sofian Răzvan

2022

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA MATEMATICĂ ȘI INFORMATICĂ

LUCRARE DE LICENŢĂ

Un model de simulare a ecosistemelor inspirat din algoritmi genetici, reguli de selecție naturală și algoritmi de optimizare a coloniilor de furnici

Conducător științific
Conf. dr. habil. Lisei Hannelore – Inge

Absolvent
Sofian Răzvan

2022

ABSTRACT

Ecosystem models are used to explain and analyze the system under study. Typically, this begins as a theoretical model that depicts plausible cause-and-effect linkages between major ecosystem components. This project presents an implementation of the olfactory sense, on top of the sense of sight, within an ecosystem, of independent agents with individual brain clocks and of the genetics system which is color coded. This approach is built on classic features of ecosystem simulations in order to outline the complex behaviours resulting from simulating both more senses and intelligence.

A compact introduction is provided within the first chapter, Introduction 1, presenting the motivation, the objectives and the original contributions noted above. The second chapter, Literature review 2, is illustrating the current state of similar ecosystem models, while in the third chapter, Theoretical Foundations 3, we are talking about the Theory of Evolution and what pathfinding algorithms have been used. After these three chapters, we will start to describe the solution 4 and how the system is implemented, going through every module, marking out how the genetics, smell and brain systems are approached in a unique way. This project is supported by the Experimental results 5, the fifth chapter, in which we are describing the results of a 7 hour long simulation run that also proves that biodiversity is part of most ecosystems.

Contents

1 Introduction **2**

1.1 Context and motivation 2

1.2 Objectives 4

1.3 Thesis structure 4

1.4 Original contributions 5

2 Literature review/state of the art **8**

2.1 EcoSim: An ecosystem simulation 8

2.2 Primer 9

2.3 Sebastian Lague 11

2.4 Fun Master Ed 12

3 Theoretical foundations **14**

3.1 Evolutionary strategies 14

3.1.1 Theory of Evolution 14

3.2 Path finding 15

3.2.1 A* search 16

3.2.2 Lee algorithm 19

4 Proposed solution **20**

4.1 Solution overview 20

4.1.1 Coordinator 22

4.1.2 Grid 22

4.1.3 Pathfinding 23

4.1.4 Entity 23

4.1.5 Energy System 26

4.1.6 Genetics 26
4.1.7 Smell 27
4.1.8 Animations 27
4.1.9 Configurations 27
4.2 Solution design 28
4.2.1 Design Patterns 28
4.3 Implementation details 33
4.3.1 Software engine used 33
4.3.2 General approach 33
4.3.3 Coordinator 34
4.3.4 GridMap 34
4.3.5 Entity 38

5 Experimental results **43**
5.1 Population 43
5.2 Genes 44

6 Conclusion **47**

Bibliography **48**

Chapter 1

Introduction

1.1 Context and motivation

Charles Darwin [HD05], the great British naturalist, geologist, biologist, and book author who founded the idea of species evolution, also known as Evolutionary Theory or Darwinism Theory [Wal07], noticed that all species of living forms developed through time from common ancestors through a mechanism he named *natural selection* [Fis58].

This process through which populations of living creatures adapt and change is highly dependent on the environment. This means that if the same creature was to be placed in two different ecosystems, it would develop unique traits specific to those two ecosystems, resulting in a solutions called local optima. Members within a species are naturally diverse, which means that they are all distinct in some way. This variation suggests that certain species have undergone mutations that have resulted in them being more suited to their environment than others. Adaptive features (traits that would provide them advantages) that are formed are likely to boost a creature's chances of survival and reproduction. These folks then pass on their adaptability to their children. These advantageous characteristics become more prevalent in the population over time. As a result of this natural selection process, advantageous genes are transmitted down across generations.

Mutations are changes in the structure of the molecules that make up genes, known as DNA. A key source of genetic variety within a species is gene mutation. They can develop at chance or as a consequence of exposure to hazardous com-

pounds or radiation in the ecosystem. While evolutionary changes take place over time, new forms of life are created as lineages separate through a process called *speciation*.[CO+04]

Natural selection and the ability to adapt of various species within an environment, depending on the vast number of variables that determine mutations, is difficult to predict. For this predictions to be easier to obtain, simulations become effective and help us predict how some environments will impact natural selection and how the ecosystem will be defining itself.

Ecosystem modelling [HD] is a field that is growing because of the technologies that are getting better with each year. Long before, complex simulations were not possible due to the fact that the computational power needed to run these simulation was far from reachable and even if there could exist such a computer, it was not easily accessible for most of the population. This hindered the development of ecosystem modelling for many years to come. These models usually simplify the systems to a limited number of components. This opens up opportunities for the development of ecosystem simulations that run on computers. The main goal of these simulation is to observe and predict how different behaviours and environments affect the evolution of an ecosystem over certain periods of time.

However, after the year 2000, the computational technology growth was getting close to exponential and millions of people started to have access to personal computers and internet. This is also the time when scientists have obtained super computers capable of running simulations that were considered impossible to run not too long ago. These two facts alone opened a tremendous number of opportunities for new research to be done in every field, including Ecosystem modelling.

The project takes advantage of these opportunities and illustrates an implementation that is focused on the senses of the observed species. Using biology and Charles Darwin theories, the project depicts the strengths of one of the senses over the others. This is seen through the behaviours of the individuals, which become more and more aware, and, thus, more natural in their actions within their environment. This is also seen in various applications of ecosystem modelling such as Ecosim [SMG19].

1.2 Objectives

The goal is to simulate and illustrate Charles Darwin's theory of evolution, as well as other models that explain the complex phenomenon of how populations evolve and how evolutionary changes occur, in order to obtain results that would better predict real world outcomes. The agent-based model [Abb12] describes the main behaviors and features of an ecosystem, whereas the simulation illustrates how the model changes over time, in order to predict the operation of real-world evolution processes or systems.

This implementation aims to simulate an ecosystem behaviour that is as close as possible to the real life behaviour, meaning that it will take advantage of multiple senses and will allow every observed individual to have its own brain and act at their thinking speed limit. However, this implementation is more costly due to the fact that every observed individual will need to obtain information from all his senses at the same time, which can duplicate some sensed information. On the other hand, another objective is to overcome this disadvantage by using efficient solutions for implementing the components of the application. This is advantageous for some of the observed individuals as they might have better sight or smell sense, which in return will be more likely to survive in the environment, but will consume more energy.

1.3 Thesis structure

The second chapter, Literature review 2, is illustrating the current state of similar ecosystem models represented using simple implementations mathematical models In the third chapter, Theoretical Foundations 3, we are discussing about the Theory of Evolution and what pathfinding algorithms have been used and why are these algorithms appropriate. After these three chapters, we will start to describe the solution 4 by talking about the solution overview going through every module. After this, the solution design will be presented showing the structural design of the project, while, lastly, in the subsection *Implementation details* we will mark out how the genetics, smell and brain systems are approached in a unique way. This project is supported by the Experimental results 5 in the fifth chapter in which we

are describing the results of a 7 hour long simulation run. These results show how natural selection is choosing the best solution for the current environment, changing the genes over time.

1.4 Original contributions

Simulating as many behaviours as possible being the main objective, the project benefits from implementing a couple of behaviours that complement the already known patterns of simulating an ecosystem.

1. **Smell sense**

   Inspired by the ant colony simulations [DBS06] and how the ants follow paths using pheromone, the project implements the smell sense, as a way of interaction for the observed animals, similarly using nodes that are placed where the creatures are walking.

   The basic concepts of an ant colony simulation are illustrated by the Softology's Blog on their post about this topic [sof20]. The simulations are based on simple concepts. The ants start to move randomly from a starting point, resembling the colony nest, and if an ant finds food, it will take that food and bring it back to the nest. While the ant does this, it leaves a trail of pheromone that will help other ants to find the food and take it to the nest.

   This simulations are about surviving as a colony, where teamwork is important, whereas in the simulation of an ecosystem, the accent is on the survival of the fittest, where competition is the defining factor, in order to obtain the optimal evolution of the ecosystem, namely, the optimal solution.

   In this project, the smell sense is not acting as a communication mechanism. Instead, if the creatures have the smell sense developed enough to sense the smell, they will recognize if the smell comes from a Plant or from another Creature and will choose what to do based on their needs and gene based behaviour.

2. **Individual brain timings and Competition**

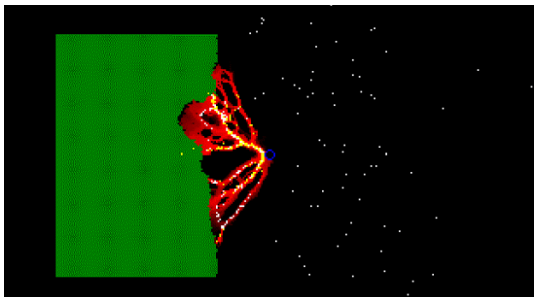

Figure 1.1: Ant colony simulation - Source: Softology's Blog [sof20]

Another concept that differentiates the project from the others is the use of individual brain for creatures. This allows the creatures, through their chromosome genes, to be faster or slower in their actions. This comes as a cost as the faster the creature brain is, the more chance of stealing food from others it has. This also benefits the creatures in scenarios in which they fight, the faster creature being favored. The concept allows the simulation to have a more natural behaviour and not use theories like Game Theory [FT91], which is the study of mathematical models of strategic interactions between rational agents, that have an exact outcome that can be predicted. Between two opponents with their own strategy, game theory gives a mathematical technique for picking an optimal strategy, that is, an optimum option or series of options. For example, in our simulation the outcome in a fight between two creatures is not wanted to be expressed by exact mathematics, instead the outcome will be expressed by chance influenced by genes, feature which makes the simulation closer to reality, because it manages the case in which two creatures could be fighting and a third creature that may come could steal their food, neither of the two winning.

3. **Energy System**

   The energy mechanism of this project is also trying to get as close to reality as possible by integrating a system based on kilocalories. Each entity that is considered food will also have a property called *nutrition*, which will tell how many kilocalories is the food worth. After being eaten, for simplification, the kilocalories number is transformed into a number from 0 to 1 as each creature

has a hunger meter that takes values only between 0 and 1 inclusive. Creatures energy will be depleted over time and when they perform actions. Each action will cost a certain amount of energy depending on how genes affect that action. If a creature moves faster, it will deplete its energy faster, but will have an edge when fighting other creatures for food. The actions are defined with default neutral cost values and genes apply modifiers to that values accordingly. Another point is that the system is replicating a behaviour in which creature gain the energy the moment they eat and can starve anytime in their lifetime, not depending if a day had past.

4. **Complex and upgradable genetics system**

   It is worth mentioning that the Genetics System used in this application is making it easy to add new genes to the creature and lets the creatures take advantage of how many genes there are to be added. At the current state, the creatures have six genes, meaning that there are twelve different types of behaviour and it is varying with float values between -1 and 1, 0 meaning it is neutral or it is using the default values for that gene.

Chapter 2

Literature review/state of the art

2.1 EcoSim: An ecosystem simulation

In this work [SMG19], the author managed to implement a simulation that is very complex and detailed in most of the aspects, while still running rather well.
The simulation's world is resembled by a grid with dimensions 1000 by 1000 cells which can hold multiple entities at the same time. Not using a system in which a cell can hold only one entity can have a big impact performance wise, because of pathfinding algorithm. His approach is a very large scale simulation which takes into account a multitude of aspects in regards to the behaviour of the agents. This came with need of intense optimizations because of the computational challenge for both memory management and computational power for such a complex representation. The fact that the simulation works in steps favours the ability to slowly but surely run the simulation for periods that can take weeks and obtain important data about the evolution of the ecosystem.

Agents behaviour is described with the help of the Fuzzy Cognitive Map model [SGG97], which describes how much a certain impulse affects the agent's decision making or behaviour.

This model gets very close to simulating an accurate ecosystem because of the fact that it is running tens of thousands of agents at the same time with a complex Fuzzy Cognitive Map for each individual.

Dr. Robin Gras was interested in understanding how complex and predictable their system is and to understand this, they examine if there exists a chaotic be-

haviour in the signals represented by the number of preys and predators and their respective number of species. The simulation process was indeed chaotic, which indeed is important for the research of ecosystem as showing the existence of complex behaviour being developed is very important in any attempt of creating a realistic system.

Another feature that is worth mentioning is the ability of agents to have a memory of their past and make decisions according to their past experience.

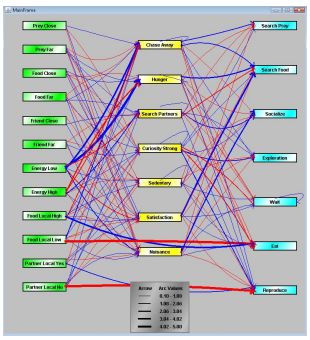

Figure 2.1: Fuzzy Cognitive Map diagram of a predator agent

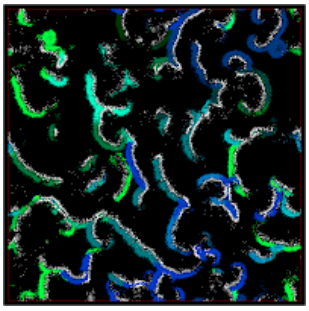

Figure 2.2: Image of the simulation running after an impactful number of steps

Links:

EcoSim: An ecosystem simulation

2.2 Primer

Justin Helps has developed several applications that simulate behaviours that are typical in an ecosystem. Three applications are relevant to this project: *Simulating Natural Selection*[Hel18], *Simulating the Evolution of Aggression*[Hel19] and *Simulating the Evolution of Sacrificing for Family*[Hel21]. All of them depict visualisations of isolated ecosystems that simulate a specific aspect about them. Another important aspect is that it takes into account the interaction between traits and environment.

His approach is keeping the implementation simple because he is using Game Theory. Most of the implemented modules are simplified. For example, the 3D graphical interface is easy to understand and color coded for better readability. Also, the logic of the application is also simplified. Creatures have energy and consume it by moving while searching for food and only gets replenished the following day if

the creature survives. If a creature runs out of energy, it starves. If the creature find 1 food, it survives and returns home. If a creature finds 2 food, returns home and also replicates. The replication simplicity is similar to how cells divide and does not use a male, female concept.

It is worth noting that its application takes advantage of the survival of the fittest strategy, in which creatures pass their gene to their offspring, similar to real life DNA. This integration however, does not benefit the creatures with multiple genes. The creatures only resemble one behaviour at a time, which again, holds its isolated case scenarios.

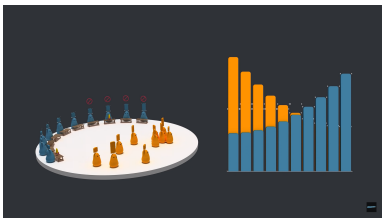

Figure 2.3: Primer graph [Hel19]

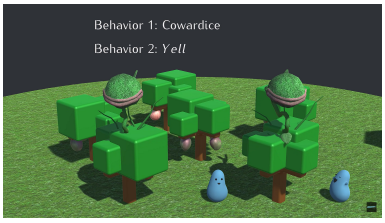

Figure 2.4: Primer 3D graphics [Hel21]

Most of its applications do not have any interface to represent creatures hunger or other needs that they might have. It also could have taken into account the differences between females and males when running the tests because often times gender behave differently. Moreover, more behaviours or interactions might benefit the accuracy of the simulation.

Primer also mentions Probability theory [Loe17], Hamilton's rule [WMM+01] and Nash Equilibrium [Mas99] in his work, which, together with graphs that graphically show the results, make his applications mathematics oriented.

Links:

1 Simulating Natural Selection

2 Simulating the Evolution of Aggression

3 Simulating the Evolution of Sacrificing for Family

2.3 Sebastian Lague

Sebastian Lague's Coding Adventures are about implementing solutions to known problems. In this case, Lague's application [Lag19] for simulating an ecosystem is consistent and the closest related to how reality functions, compared to other mentioned projects.

His approach consists in making a 3-dimensional simulation of an environment that is as natural as possible and which include more than a couple of interesting integrations, including needs meter, several species and the desirability variable of animals.

The most intriguing features implemented in his project are the ones related to reproduction. His observed individuals consists of more than one species, defined by the programmer and each of these species have males and females. For reproduction to be possible, creatures have to be of different genders. The individuals have been defined with a random desirability variable that acts similar to a gene that make the individual more desirable for the other mates. Before the reproduction itself, there are the needs, which are implemented as meters that increase over time and if these needs are satisfied, the meters values decrease.

Another interesting aspect about his project is the graphical interface. The color coding of the animals, for example. If a male is more desirable, it will have color that tends more to red, which helps to visualize how the simulation works. Also, he is using 3D low polygon models or each object included in his scene, which makes the application easier to follow.

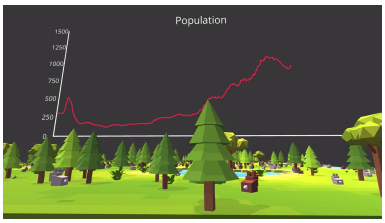

Figure 2.5: Sebastian Lague graph [Lag19]

Figure 2.6: Sebastian Lague graphics [Lag19]

When we are talking about the evolution of the ecosystem, the implementation of Sebastian Lague in his ecosystem simulation is taking into consideration the mutation of genes, which makes the application more complex and also closer to how

real world would work.

On the other hand, these genes and mutations could be more diverse, by implementing several more traits that could influence the evolution of the observed individuals in the selected environment.

His senses implementation is a simplified approach, all senses being treated as a one big circle around the animal. This could have been improved by using different types of senses, varying the ecosystem even more.

Even though he has created different species, there are only two, that represent the predator and the pray and there has not been so much emphasis on the predator species as they do not have a thirst meter and hunt the whole time, while the pray has other needs that has to take care of. This could make the simulation results to be one sided in the favor of the predator.

Links:

1 Coding Adventure: Simulating an Ecosystem

2.4 Fun Master Ed

This project [Ed20] takes on an approach on this subject that combines features from the previous two described applications, using a 3D graphical interface, a predator and pray predefined and a simplified approach to the problem, but not without using mathematical models.

The graphical interface is friendly, but used to only roughly represent the objects. In return, the simulation is easier to follow, because the type of species is abstracted. Similar to the previous project, the programmer has implemented more than one species, which consist of predators and pray.

The most interesting fact about this project is the use of theories from the economics field, related to changes in population, along with graphs that consolidate those uses of theories. One of them is the use of happiness score, using the utility or diminishing returns in Economics[SF74], for how food delivers less happiness each time a certain type of food has been eaten by one individual. Given the fact that the predator species alive can only eat so much - one or two types of animals, the

utility should not matter to the point that the predators would stop eating even with animals to hunt nearby.

There could have been more emphasis on details such as a more diverse environment, having multiple species and food. A needs meter would have helped with the visualisation of the progression. One aspect that could have made the simulation a little bit more realistic would have been the use of different genders.

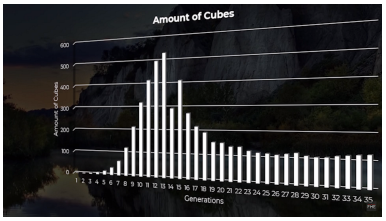

Figure 2.7: Fun Master Ed graph [Ed20]

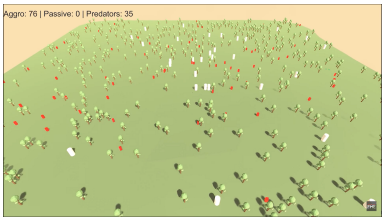

Figure 2.8: Fun Master Ed graphics [Ed20]

Links:

1 Simulating an Ecosystem

Chapter 3

Theoretical foundations

3.1 Evolutionary strategies

3.1.1 Theory of Evolution

Anaximander of Miletus [Bur49], a Greek philosopher who lived in the 500's, proposed theories to explain how creatures change or evolve over time. Anaximander hypothesized that because human newborns are born helpless, humans must have descended from another species whose offspring could live without assistance. He deduced that those predecessors must be fish, because the fish hatch from eggs and start life without the help of their parents. Based on this logic, he suggested that all life began in water. [Gou85]

"Anaximander was correct; humans can indeed trace our ancestry back to fish. His idea, however, was not a theory in the scientific meaning of the word, because it could not be subjected to testing that might support it or prove it wrong. In science, the word "theory" indicates a very high level of certainty. Scientists talk about evolution as a theory, for instance, just as they talk about Einstein's explanation of gravity as a theory."[Soc22]

A theory is a concept of how something behaves in nature, which has been rigorously tested through observations and experiments to show the correct or incorrect notion. Regarding the development of life, many philosophers and scientists, especially an eighteenth-century English doctor named Erasmus Darwin, postulated various components of what would eventually become evolutionary theory [But82]. But it was not until Darwin's nephew, the famous Charles Darwin, launched his fa-

mous book, The Origin of Species, that evolution became a scientific theory. Darwin and a fellow scientist, Alfred Russel Wallace, argued that evolution occurs as a result of a phenomenon known as natural selection.

According to the theory of natural selection [PB90], organisms create more offspring than they can survive in their environment. Those who are more physically able to live, mature and multiply. Those who lack such fitness, on the other hand, either do not reach reproductive age or have fewer children than their peers. Natural selection is usually summed up as "survival of the fittest", as the "most suitable" organisms - the most adapted to their environment - are the ones that reproduce most efficiently and are most likely to pass on their qualities to the next generation. The above implies that if an environment changes, so will the characteristics that help people survive in that ecosystem. Natural selection has been such a convincing concept in describing the evolution of life that it has become a scientific theory. Since then, biologists have found several examples of natural selection that influence evolution. It is now understood that it is one of the ways in which life develops. A mechanism known as genetic drift [LAG+16], for example, can cause species to develop. In genetic drift, certain species generate more offspring than would have been predicted by chance. These creatures may not be the most suitable of their species, however their genes are passed on to the next generation.

3.2 Path finding

By definition, pathfinding [GMS03] means the action of finding or marking a path or way, avoiding obstacles, in an optimal manner and its earliest use was found in Saturday Review, mid 19th century. In computer science, the term pathfinding refers to the plotting, by a computer application, the shortest path or route between two given points, considering obstacles. Pathfinding is a field of research that relies considerably on the Dijkstra's algorithm, used for finding the shortest path on a weighted graph. The root idea of a pathfinding algorithm is starting from a fixed point in a coordinate system and then exploring adjacent nodes until it reaches the goal node. The goal of this field of research is optimising the time it is required to solve such a problem for large amounts of data and for different situation that might

need a separate optimal solution, because the more complicated problem is to find the optimal path in the optimal time. The exhaustive approach would be the case of the Bellman-Ford algorithm which results in a quadratic time. Since it is not needed to search all the possible cases, algorithms like A Star algorithm and Dijkstra's algorithm which eliminate, through dynamic programming, impossible paths. A star algorithm is one of the optimal algorithms often used in games, while Lee algorithm is one possible solution for maze escaping problems, but not as optimal.

The main problem that needs to be treated in the case of this application is that when the number of agents rise in the simulation, each of the creatures needs to arrive to its destination without colliding unintentionally. The problem was overcome by letting the creatures follow their calculated path until the path is blocked and only then search again for another optimal path. This works as expected and does also benefit the performance as there are not many calculations that have to be updated constantly at every grid tile that has been traveled by the creatures. This approach was chosen taking into account that the simulation does not work in steps, as every creature has a separate internal clock which updates according to their brain speed gene.

3.2.1 A* search

As mentioned earlier, the A* algorithm [FGK+21] is usually used in games for strategy games units, bots in most games and other similar examples. This algorithm is actually a variant o Dijkstra's algorithm. The way the algorithm works will be described in the following lines. Each open node is assigned a weight equal to the weight of the edge to that node plus an approximation of distance between that node and the finish node.

The approximation is calculated by a heuristic function, which ranks alternatives in search at each branching step. Based on the information that is available, it decides which branch to follow. This heuristic represents the minimum distance possible between that node and the goal node. This will remove the possibility of choosing longer paths once an initial optimal path is found. This heuristic function is actually what makes the algorithm better than the Dijkstra's algorithm. One aspect to note here is that when the heuristic method evaluates to zero, the A star

algorithm becomes equivalent to the Dijkstra's algorithm. An important aspect to notice about this algorithm is the fact that while it continues to find optimal solutions, it runs faster, because of the nature of examining fewer nodes. When the value of the heuristic function is equal to the exact distance between the start and the end node, then it means that the path is a straight line between the two and thus, it is the fastest case that the algorithm runs. Most commonly it is implemented using the Manhattan distance over the Euclidean distance in the 2D space, because of the same result being reached by easier calculations.

The reason this application uses the A* algorithm for pathfinding, is to demonstrate how fast and convenient the algorithm is in most cases. This simulation does not have a lot of cases in which a creature would need to find a path to a distant location that may be covered in obstacles. Most of the time there are direct paths leading to where the creatures want to go. This means that the best solution was to choose an algorithm that does not waste processing power on direct paths, but does find an optimal solution if obstacles appear. For example, there are times when there are a lot of plants and creatures roaming around. The special case that will be costly most of the time is the case in which a creature needs to go towards its mate, sensed by smell, as smell trails remain active for several grid tiles.

```
void FindPath(int2 startPos, int2 targetPos, Grid grid)
{
    Node startNode = grid.grid[startPos.x, startPos.y];
    Node targetNode = grid.grid[targetPos.x, targetPos.y];

    List<Node> openSet = new List<Node>();
    HashSet<Node> closedSet = new HashSet<Node>();
    openSet.Add(startNode);

    while (openSet.Count > 0)
    {
        Node node = openSet[0];
        for (int i = 1; i < openSet.Count; i ++) {
            if (openSet[i].fCost < node.fCost || openSet[i].fCost ==
```

```
                    node.fCost) {
                    if (openSet[i].hCost < node.hCost)
                        node = openSet[i];
                }
            }
            openSet.Remove(node);
            closedSet.Add(node);
            if (node == targetNode) {
                RetracePath(startNode,targetNode, grid);
                return;
            }

            foreach (Node neighbour in grid.GetNeighbours(node)) {
                if (!neighbour.walkable || closedSet.Contains(neighbour)) {
                    continue;
                }

                int newCostToNeighbour = node.gCost + GetDistance(node,
                    neighbour);
                if (newCostToNeighbour < neighbour.gCost ||
                    !openSet.Contains(neighbour))
                {
                    neighbour.gCost = newCostToNeighbour;
                    neighbour.hCost = GetDistance(neighbour, targetNode);
                    neighbour.parent = node;

                    if (!openSet.Contains(neighbour))
                        openSet.Add(neighbour);
                }
            }
        }
    }
```

3.2.2 Lee algorithm

This algorithm [Lee61], even if it a possible solution for maze routing problems, it is quite expensive and time inefficient. It was chosen to show how different path searching algorithms compare in similar situations in the simulation. The steps through which the algorithm works is by marking the start point and then going through all neighbours of the current cell repeatedly until the current cell is the target node. The visualization of this algorithm is similar to a circular wave propagating from a starting point.

Chapter 4

Proposed solution

4.1 Solution overview

The user interface is color coded. The ecosystem is represented from a 2D perspective and uses a grid matrix as it simplifies the visual interpretation of the simulation.

In terms of implementation, the project idea is broken down into as many modular and simple pieces as possible, so that evolution choices are made by natural selection rather than by the programmer. In this respect, the emphasis will be on the impact of evolution rather than its physical form. Information on whether an animal develops bipedal or quadrupedal movement will be abstracted since the speed with which the animal travels is what is important. As a result, mutations will have an impact on an attribute that depicts the creature's speed of movement and not on how many limbs it has developed through mutations. This approach uses abstraction because the resulting data will already be classified in simple terms.

Even though the simulation is reduced to simple modules, the goal is to be as close as possible to reality. In this regard, the simulation is characterised by several features that make it unique.

1. **States and actions**

   The creatures behaviour is defined by a set of states in which the creature can be in, depending on its current needs and surroundings. The states are: *none*, *thinking*, *exploring*, *goingToEat*, *goingToMate*. These states will then determine what action is going to be done by the creature, while the actions store the information about how much energy does it cost for that action to be finished.

2. **Alimentation system and energy system**

   All edible *LivingEntities* are worth an amount of kilocalories if eaten, based on different genes and randomness. These foods are the digested and transformed into energy, which helps creatures to survive for longer.

3. **Cross-over**

   All species have both females and males. This means that for reproduction to be possible, a male must find a female, and then that female must accept the male as a mate. Through reproduction, genes are passed over to the offspring.

4. **Mutations**

   Including mutation is necessary in such a project, but the biggest challenge is to make them balanced in a way that the environment will shape the ecosystem and not the settings made by the programmer. I have chosen a large range of possible values between which genes values can move through. This, in return, will leave plenty of room for natural selection to choose the best species.

5. **Selection**

   Selection is done naturally through the survival of the creatures that are suited better for the current environment. Creatures that do not manage to eat before their energy meter is depleted will starve.

6. **Competition**

   Instead of using Game Theory, competition is simulated using the advantages and disadvantaged that some agents have over the other agents through the combinations of the genes that they have.

7. **Senses**

   For the agents to be able to interact with the world they live in, they need to sense it in some way. In this project, the simulated senses are the sight sense and the smell sense, both of which can be influenced by the creature's genes and can provide an advantage or a disadvantage to their survival ability.

4.1.1 Coordinator

It is the main module of the project. Its purpose is to act as a global clock for the implemented systems and entities in the simulation and to do a set of actions, repeatedly or not, that affect the whole world. It can change the timescale of the application, running faster or slower depending on what is needed.

At start this module is responsible for calling the constructor of the Grid in which the world is simulated and over the entire simulations it will call certain systems every set few seconds and will also send update signals to the individual brain for every of the living creatures.

4.1.2 Grid

One of the more important components of the simulation is the Grid namespace in which various classes are used for creating and running the matrix in which the simulation takes place. It is divided into the GridMap class, Cell class, Tile class and the PathFinder class.

1. **GridMap**

   The *GridMap* class is the actual class in which the 2-dimensional array is stored and where all the methods for modifying this array are. Each position in the 2-dimensional array stores a Cell object.

   Its importance is notable as almost all the changes that happen on the grid matrix are processed through this class instance. Some responsibilities of the GridMap include displaying graphically the simulation, spawn entities and verify for possible moves, return paths through calling the Pathfinder class methods, updating the brain of LivingEntities, store Entities that are in the Alive state, remove entities from the matrix, return what creatures can sense through their sight or smell sense, considering the agent's genes.

2. **Cell**

   Cells are the objects that can hold what entity is on that specific cell and can only contain one entity. This also can contain multiple Smell class instances and one Tile object associated with it.

3. **Tile**

   The *Tile* class is responsible for rendering a graphical texture for a grass square for each Cell there is on the Grid.

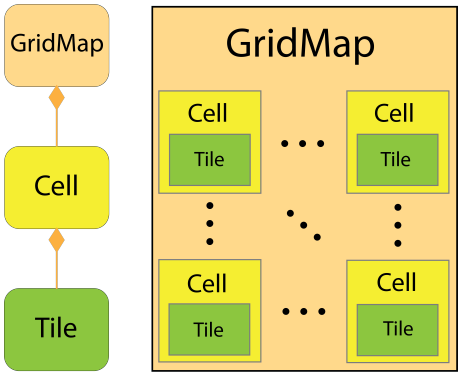

Figure 4.1: Structure of the Grid

4.1.3 Pathfinding

This is the module in which the class family Pathfinder is implemented, where multiple implementation for pathfinding algorithms are defined and can be called and interchanged during runtime. Here is where the adapted A star and Lee algorithms are implemented using the Strategy Design Pattern.

4.1.4 Entity

The *Entity* namespace contains all the classes that inherit or are related to the Entity class.

1. **Entity**

   The *Entity* class itself is a generalized class that characterizes entities as instances which have an object and is located at coordinates x and y on the GridMap. It is inherited directly by the LivingEntity class and SmellNode class.

2. **LivingEntity**

   LivingEntity is the class inherited directly from the Entity class and is responsible for all energy related tasks and also contains the Chromosome of the living

entities. It contains the properties related to how much energy does a living entity currently have, whether it is alive or not, how much kilocalories it values if it is being eaten and also the methods that manipulate those attributes. This class is inherited directly by the Creature class and Plant class.

3. **Creature and Plant**

   The *Creature* and *Plant* classes are the first classes that can be instantiated, besides the SmellNode class. These classes are also the classes that define all the behaviour of the agents present in the simulation and are the final classes in the tree of inheritance in the Entity namespace. The reason for this is that in order to have a very large variety of species and behaviour, the agents need to be as abstract as possible and develop new behaviour through the process of speciation and mutation.

4. **Creature**

   This class holds most of the behaviours that influence most of the simulation's outcome. It resembles all the species of animals or insects that exist. *Creature* class, inherited from the LivingEntity class, is directly defining the behaviour of the agents and contains all the attributes related to the current physical and mental state of the creature, as well as methods that can manipulate those attributes. The behaviour of the creatures is characterized by states. One can be in only one state at a time and will perform specific actions while in that state. Since the simulation does not run in sequential steps, creatures have been implemented using a brain model which means that the creatures think by themselves and take action, with their own artificial consciousness, at time intervals determined by how intelligent is that creature. The actions that the creatures decide to make, are defined in this same module and cost a certain amount of energy to perform.

   A creature is defined as a living being, either female or male, that can move using a 4 directional movement and which has needs that needs to be fulfilled. It has an artificial consciousness that is similar to the finite state machine as it takes action according to the current state and sensed environment. The current state is determined first in the chain of its action taking into account

its current needs and its surroundings. Next, it changes its state and follows to execute the action associated with that state. If a creature fails to fulfill its needs, it is most likely going to starve and thus, not be able to reproduce. This, however benefits natural selection based on survival of the fittest.

Figure 4.2: Female and Male symbols each on a Tile of grass

Creatures can also age and thus, only reproduce after getting mature. This means that it can also die of old age.

Reproduction can be achieved only if both the female and male creature are mature and their need to reproduce is high enough. If a male creature wants to reproduce, it starts to search for females using his sense of sight and his sense of smell. After finding a female, he first needs her approval to reproduce, otherwise he will continue searching. If a male is rejected, he is going to have a memory, remembering which females rejected him, not trying to mate with them for a period of time.

Because of the use of chromosomes and genes, creatures have, at this current state, 6 genes that resemble motor speed, brain speed, strength of the sight sense, strength of the smell sense, food preference and behaviour, which influences how likely is the creature going to hunt and eat another creature or not. These genes have a big impact on the ability of the creature to survive in its current environment.

5. **Plant**

   Plant class is also inherited from the LivingEntity and resembles all the edible plants that will be eaten by the creatures that are more herbivore. The plants will be spawned regularly at the call of the Coordinator and using GridMap methods.

Figure 4.3: Plant symbol

6. **State**

   The *State* module is responsible for defining the states in which a *LivingEntity* can be in, be it a physical or mental state.

7. **Assets service**

   Assets service is used to store the sprites and make them to be easily accessible through the singleton instance of its class.

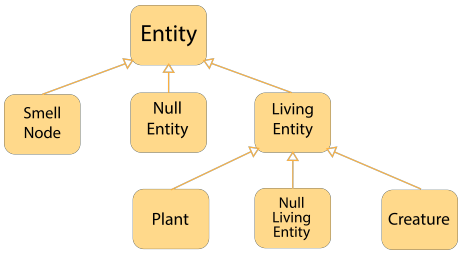

Figure 4.4: OOP structure within Entity domain

4.1.5 Energy System

Is responsible for updating, mainly decreasing, creatures energy every few seconds. It also stores default values for energy consumption tasks.

4.1.6 Genetics

All the information and data related to the chromosomes and genes of creatures are processed in this module, where Chromosome class is defined. A chromosome can have any number of genes and its responsibility is to handle the insertion of new genes, when a Chromosome is defined, to handle the gene inheritance from parents chromosomes and also to control how mutations occur. A gene resembles a feature of a chromosome that influences certain behaviours or functionalities of Living Entities.

This module also implements a way to compare if two chromosomes are similar by considering an average difference between the genes, after being multiplied by some weights.

4.1.7 Smell

Responsibility to how the smell works in the simulation is carried by the *SmellNode* class which functions as a self constructing graph that always has a maximum amount of nodes and looks like a trail left behind the agents. A *SmellNode* has an intensity, and contains information about what type of LivingEntity left the trail and the direction from where the smell is coming from. It will be used by creatures to identity other creatures as food or mates from distances beyond sight. To create a new chain of nodes, one needs to use the constructor that takes as an argument only a LivingEntity and to continue an already existing chain, the constructor with a LivingEntity and the current head SmellNode needs to be used. When a creature dies, its smell chain will dissipate along with it.

4.1.8 Animations

This module is used to implement methods that run animations using Unity's GameObject transforms.

4.1.9 Configurations

Configurations is responsible for storing all the settings that are not changing throughout the runtime of the application, but have a big influence on how the simulation will work and this way, making them easier to be changed globally.

4.2 Solution design

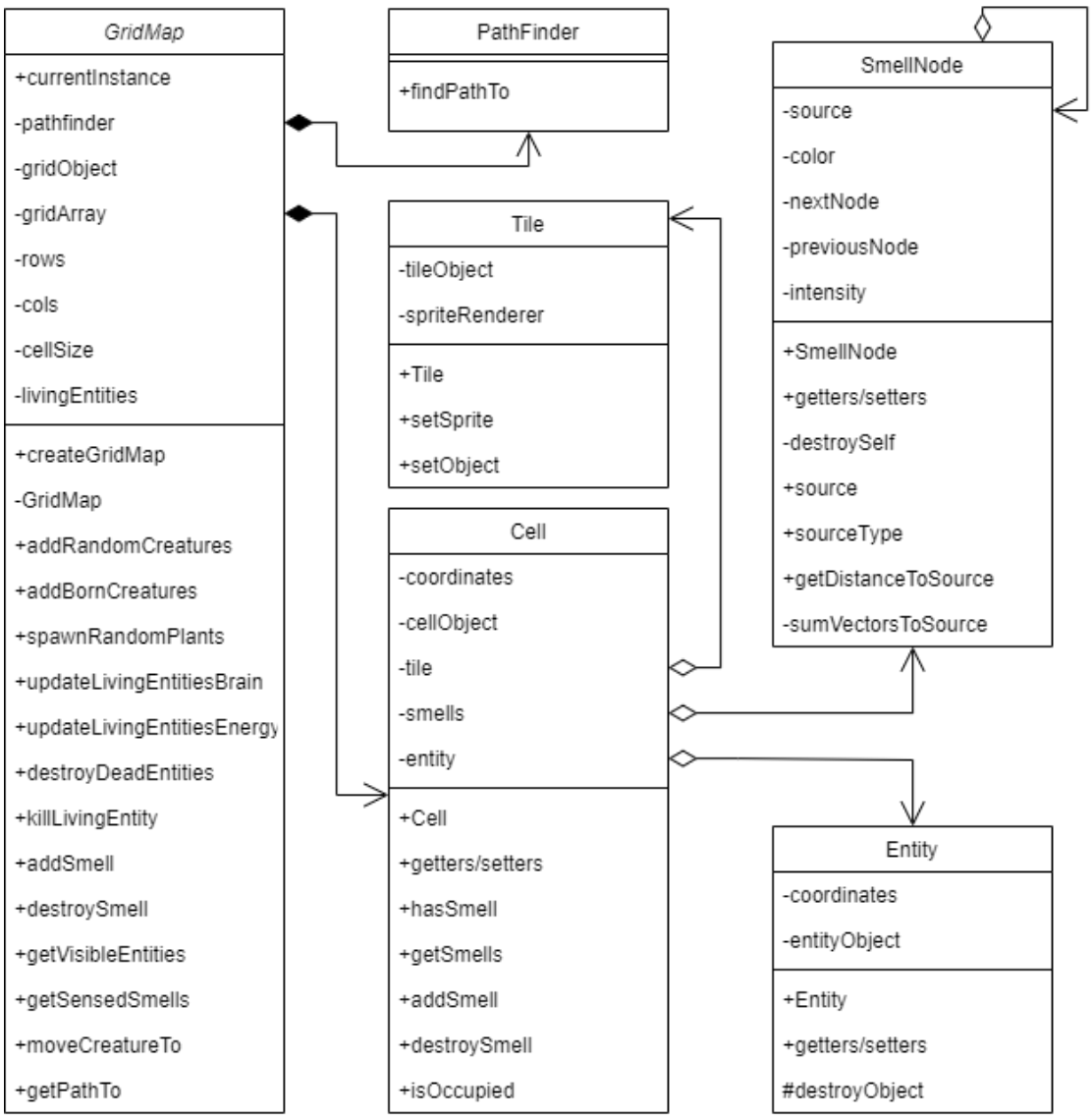

Figure 4.5: Class diagram of the GridMap and its related classes

4.2.1 Design Patterns

Christopher Alexander initially introduced the notion of patterns in A Pattern Language: Towns, Buildings, and Construction [Ale77]. The book provides a ”language” for creating the urban environment. This language’s units are patterns. They may specify the height of windows, the number of levels in a structure or the size of green spaces in a community.

Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm were among the

authors who took up the topic. They released Design Patterns: Elements of Reusable Object-Oriented Software [GHJ+95] in 1994, which used design patterns to apply to programming. The book offered 23 patterns that solved different object-oriented design difficulties and rapidly became a best-seller.

Design patterns are conventional approaches to common challenges in software design. They function similarly to pre-made blueprints that you may modify to overcome a persistent design problem in your code. Patterns and algorithms are sometimes misconstrued since both notions represent typical solutions to well-known situations. A pattern is a more high-level description of a solution than an algorithm, which always describes a defined set of activities that can achieve some goal. The code for the same pattern in two separate programs may differ.

Furthermore, all patterns may be classified according to their goal or purpose. Object creation techniques provided by creational patterns enable the flexibility and reuse of existing code. Structural patterns describe ways to group items and classes into bigger structures while keeping them flexible and efficient. Finally, behavioral patterns are responsible for successful communication and the assignment of duties amongst objects.

This project takes advantage of a variety of design patterns from all three pattern categories, which improve code readability and long-term stability.

Creational pattern – Singleton pattern

As previously stated, the creational patterns give an object creation method. The Singleton pattern [Fre15c], which was used numerous times in the project across different classes, ensured that some of the classes had only one instance, while also establishing a global access point to this instance for all other classes inside the project that may utilize these classes.

A few steps have to be taken into account in order for the Singleton pattern to be effectively implemented. To begin, the default constructor was made private in order to prevent other objects from using the new operator with the Singleton class. Second, a creation method had to be written; however, because the constructor was specified to be private, we implemented a static creation method that functions as a constructor. This function uses the private constructor to build an object, which is

then saved in a static field attribute. If the object does not already exist, the static creational function will invoke the private constructor and return it. If the instance already exists, it does not create a new object and instead returns the already built instance by using private constructor.

Using this pattern comes with advantages that benefit the project as a whole. If needed, you can be sure that a class implemented with the Singleton pattern has only a single instance, making it impossible to overwrite existing data or to interrupt running tasks. The instance of the class implemented with this pattern will also gain a global access point to that instance, meaning that the class is also easily accessible

It may also have drawbacks since it contradicts the Single Responsibility Principle by addressing two problems at the same time. Another issue is the difficulty in unit testing the Singleton's client code because many test tools depend on inheritance when creating fake objects. The constructor of the singleton class is private, and most languages make it hard to override static methods. In addition, the pattern requires particular handling in a multithreaded context so that several threads do not build the same singleton object repeatedly.

Structural pattern – Flyweight

Due to the sheer large number of objects modified during the runtime, the Flyweight design pattern [Fre15b] is among the most helpful design patterns utilized in the project. Flyweight is a structural design approach that allows you to load more instances into the available RAM by sharing common components of information across several objects rather than maintaining all of the data in each object. Because there is frequently a large collection of items active at any given moment, this pattern is more popular in game development and simulation projects.

The intrinsic state of an entity refers to its static data. It exists within the object and cannot be changed by other aspects. The rest of the object's state, which is frequently affected by other objects, is referred to as the extrinsic state. The Flyweight design proposes that you avoid storing the object's extrinsic state. Instead, transfer this state to specified functions that rely on it. Only the intrinsic state is preserved within the object, allowing it to be reused in other situations. As a result, fewer of these objects are required because they only differ in the intrinsic state, which has

far fewer changes than the extrinsic.

In this project, the pattern was used in the implementation of the classes that inherit the Entity class, namely *LivingEntity* class, Creature class, Plant class and *SmellNode* class. All the attributes that are common between all the instances of a class are stored in a separate class named *EntityConfig*, which also makes this data available not only to the Entities, but also to any classes that need to manipulate Entities data, such as the *GridMap* class, which is responsible for the whole matrix in which the simulation takes place. These attributes, defined in the *EntityConfig* class, are defined as static and can be accessed through the *ConfigsInterface* class which is implementing the Facade pattern. Some of these attributes are: *SightDistanceInCells*, which is the default sight distance for all Creatures, *CreatureNutritionValue* and *PlantNutritionValue*, which represent how many kilocalories is the Creature or Plant worth after being eaten.

Along with this pattern come advantages and disadvantages. The biggest advantage of using this pattern is the fact that it saves a considerable amount of RAM, because the project has many similar objects. On the other hand, disadvantages are still present as this pattern trades RAM usage for more CPU usage and also the code is more complicated than when using the traditional implementations.

Structural pattern – Facade

As noted previously, the Facade design [Fre15a] is used in conjunction with the Flyweight pattern to organize static data in one location and make it available exclusively through an interface. Facade is a structural design pattern that gives a simple interface to a library, framework, or any other complicated collection of classes. A facade comes in useful when you need to connect your app with a complex library that has hundreds of functions but only needs a fraction of them.

Using this pattern, the application had the code isolated from the complexity of the configurations and how different values are computed. Even though it is not the case, one disadvantage of using the facade is that it can become a god object coupled to all classes of an application.

Behavioural pattern – Strategy pattern

The project utilizes the Strategy Pattern [Fre15d] to define pathfinding behavior, which includes a collection of pathfinding techniques to select from. Strategy is a behavioral design pattern that allows you to construct a family of strategies, classify them, and make their objects interchangeable. We wish to disassociate users of this behavior from any specific implementation. This enables the design of each unique implementation of the behavior to be hidden from the client in the multiple versions of the behavior. We may additionally include the fact that the behavior varies based on implementation.

The abstract base class *PathFinder*, which is the client's method signature, represents the Interface object. The inheritance structure therefore illustrates dynamic *polymorphism*, and the client may use the function *findPathTo* to abstract which algorithm is utilized. All classes that derive *PathFinder* override the *findPathTo* function, implementing their own pathfinding technique. This pattern's implementation is used to demonstrate the value of using one pathfinding technique above others. The base class was implemented as abstract since it cannot be instantiated because it lacks implementation for the abstract function *findPathTo*, which must be provided by its descendants classes A Star or Lee.

The most significant benefit of this design pattern is the ability to switch algorithms used inside an object at runtime via a property of type of the base class *PathFinder*, which is populated with one of its child classes that implements the desired pathfinding method. It is also important to isolate the implemented algorithms within the child classes. It is built on the Open/Closed concept, which means that new techniques may be introduced without changing the environment of implementation.

On the other hand, one drawback is that clients must understand the distinctions across methods in order to choose the best one. Another drawback is that functional type support in current programming languages allows you to construct several variations of an algorithm within a collection of anonymous functions. You may then utilize these methods in the same way you would have used strategy objects, but without cluttering your code with unnecessary classes and interfaces.

Behavioural pattern – Null object pattern

The purpose of the Null object pattern [McD17] is to hide the absence of an object of a class by implementing an alternative whose behaviour is changed to do nothing.

Its sometimes used when an object requiers a collaborator, but in this project it is implemented as an abstraction for null object handling throughout the runtime of the application.

It is used in multiple classes such as *Entity* (NullEntity) or LivingEntity (NullLivingEntity), but implemented the same. I will describe the implementation in the *Entity* class case. The *NullEntity* is a class derived directly from the Entity class. *NullEntity* implements a constructor that initializes the attributes with some default values that should resemble nothing. The setters are overriden such that they do not change any values and do not do any operations and this is applicable for all of the other methods. This means that this null object is treated as non existent, but it is not necessary for the client to check the instance for null equality.

The implementation of this pattern was necessary for when there are no entities in the cells of the grid. matrix.

4.3 Implementation details

4.3.1 Software engine used

The simulation is developed with the help of the game development engine Unity as it has tools to develop a 2D scene and most importantly, it is a real-time development platform and it is fast and agile.

4.3.2 General approach

Most of the classes defined in the project do not inherit from the Unity's *Monobehaviour* scripts class because it offered more control. *Monobehaviour* scripts need to be attached to an object in order to be able to run. Because of this, most of the data has been separated from the objects within the scene by implementing only one main coordinator class that derives from *Monobehaviour* that will be attached to an empty, but not null, object, while the rest of the classes are independent from the

Unity's classes. All the objects in the scene are created dynamically and not with the Unity's editor.

4.3.3 Coordinator

The *Coordinator* is the director of the application, controlling what should happen and when.

It manages to do this by being inherited from the *Monobehaviour* class which is the scripts class available from Unity. This class has access to Unity's update functions.

The project updates were implemented into the *Coordinator* using the *FixedUpdate* function as it does not depend on the framerate that the application runs at. This is because it will not affect how fast the simulation's time passes relative to real world time and thus, it will make it possible to obtain results much sooner.

At the initialization of the *Coordinator*, the time scale property of the application is set using the multiplier value from the configurations classes, which values can be changed before the program is ran. While still at this stage of initialization, the coordinator tries to start the initialization of the instance of the *GridMap* class, using its custom constructor. If there is an error caught while this operation is running, an exception will be thrown and caught by the Coordinator, displaying what problem did arise during the execution of the initialization of the *GridMap*.

The most important role of the *Coordinator* is to send update signals to a series of systems during the execution repeatedly over a specified interval of time. This is done through monitoring seconds passed in the *FixedUpdate* function, using *Time.deltatime* variable available from Unity in a *Monobehaviour* script. Every task that needs to be ran every few seconds, will have its own function that checks if that number of seconds have passed and if this is true, it sends the update signal to the corresponding system.

4.3.4 GridMap

The *GridMap* class can have only one instance, which was made possible through use of the singleton pattern, that does not allow the creation of more than one in-

stance through the use of a custom public constructor-like method which calls the actual private constructor only if there is not an already an instance created and returns the instance. If there is already an instance created, that public method returns the existing instance. The instance of this class is stored in a public static variable, accessible through the class name of *GridMap*.

The matrix of the simulation is stored privately in this instance as a 2-dimensional array of type Cell, which can be modified from the outside of the class only through use of public methods. This grid is programmed separately from Unity's usual *Monobehaviour* approach in order to increase performance because accessing data is faster when not trying to access it from a large number of Unity *GameObjects* multiple times a second. The number of rows and columns of this matrix is controlled from the configurations classes of the project which will be discussed below.

Within the same *GridMap* instance there are stored the references to the instances of the current *LivingEntities* (class objects). Even though the same instances are referenced in the Cells they belong to, this List collection of *LivingEntities* is used to iterate through every *LivingEntity* in a much faster way. The first method which was implemented to achieve this was to iterate through the whole matrix and check if there exists a *LivingEntity* within the Cell instance. This iteration, because it is called numerous times a second, needed to be optimized. Thus, after implementing a List with *LivingEntities*, the performance of the application increased, comparing by framerate, by at least 30

To organize all the visual objects in the scene, an attribute of type *GameObject* stores a reference to the container in which all the grid related objects will be inserted. This was also done for the cells, creatures and plants. Each of the containers for these is a child of the *GridMap*'s *GameObject* container attribute.

The *GridMap* is created dynamically, by positioning the *GridMap*, in the center of the scene, where the camera is also centered. This centering was done using mathematical calculation based on the cells size and number of rows and columns. *GridMap*'s constructor also calls the constructors for every Cell in the matrix and every Tile for these cells. This means that *GridMap*, Cell and Tile classes are strongly related by a composition relationship hierarchically.

As mentioned, *GridMap* is responsible for storing the matrix that represents the

world of the simulation and also for manipulating this matrix.

1. **Spawn creatures**

   At the initialization, the *GridMap* will try to spawn a random number of creatures based on a probability set before the launch of the program. The *GridMap* will spawn a creature roughly every N cells, where N is the input number set before launch. This means that, on average, there will be *TotalNumberOfCells/N* creatures in the scene. This random spawning is done by iterating through the whole matrix and for every cell, a random number between 0 and N will be generated. If the number is equal to 0, a creature will be spawned.

2. **Spawn plants**

   While still at the initialization process, the function responsible for spawning plants will have a much higher probability of spawning. This method, however, does not iterate through the whole matrix. Instead, it obtains a random number of plants that need to be spawned, taking into consideration probability, and then tries to spawn these plants at random locations on the matrix. For each plant, two x and y coordinates are generated. If the cell at the position [x;y] is free, spawn the plant there, otherwise, jump over this plant spawning process. This simulates how in real world, paths that are being walked the most, have a less dense vegetation.

   After the initialization process is finished, during the rest of the run of the simulation, the plants are spawned randomly at a much lower rate. This takes into account the fact that more plants are eaten in the first few seconds of the simulation and then the vegetation consumption will level over time.

3. **Spawn and remove Smells**

   When a creature creates a smell, it sends a signal to the *GridMap* to let it know that the smell created needs to be added to the matrix in order for it to be visible to other creatures. The same applies for when a smell dissipates or a creature starves or is eaten, in which case all its smell trail is removed.

4. **Execute Update signals received**

   Another aspect that the *GridMap* executes are the update signals sent from

the *Coordinator* and *Energy System* to update the creatures brain and to consume creatures energy over time. The update of the creature brain means that for the creatures in the *LivingEntities* List the main "brain update" function is called multiple times a second, which makes the creatures able to think by themselves, without the need of the *Monobehaviour* script.

5. **Return senses information**

   An important task that the *GridMap* is executing is returning a list of the entities that can be seen or smelled by a creature when that creatures requests it. This is done by obtaining the information about how strong the asking creature's senses are and check what can be sensed by that creature. After constructing the list of those sensed creature, the list is returned to the creature, leaving the reasoning to happen in the brain of the creature, *GridMap* being only a way of getting information from the environment.

6. **Move creatures**

   When a creature wants to explore, it requests from the *GridMap* a valid next neighbouring position towards it can move. The *GridMap* takes into consideration the direction towards which the creature was moving, and tries to search towards the same direction. If a cell is occupied, it tries to check other cells, but only for a few tries, because otherwise the program can get stuck in an infinite loop if all 4 neighbouring positions are occupied. The tries will make the creature wait a few seconds before trying to request for a next cell again.

   Besides requesting a next cell, creatures can request to move to a specific cell. This will trigger a series of operations. First, the creature will occupy the target next cell and only then the current cell will be freed and the moving animation will start towards the target next cell. This algorithm ensures that two creatures can not end up on the same cell at the same time and trigger tricky to handle situations.

7. **Return paths from PathFinder**

   When a creature senses something that is further than one cell from its current position, it will request a path towards that cell. The request is done only once and the creature will proceed to follow the path until arrives or when

something is blocking the path, case in which the creature will ask for a path towards its target again. This is less resource intensive than the solution in which the creature would have to ask for a path after every cell it moves.

GridMap responsibility is to pass the required arguments to the *PathFinder* stored within itself. The *PathFinder* class will return a Queue with the path and pass it in a chain towards the asking creature.

4.3.5 Entity

1. **Entity**

   Entity domain was designed with *polymorphism* in mind. Entity is the base parent class and the most abstract in the hierarchy. It can not be instantiated as it is abstract, but provides implementation for methods that manage coordinates and the main *GameObject*. The coordinates are represented as two integer, using the built in *int2* type. Both *Entity* and *LivingEntity* classes are provided with a Null pattern such that null handling in *polymorphism* is easily handled.

2. **LivingEntity**

   LivingEntity is a generalized type of anything that is alive. It can not be instantiated and it manages the energy of creatures and also manages what happens when a creature or plant dies. *LivingEntity* is also responsible for holding information about the chromosome.

   The energy is managed through two main methods which handle the change in energy of the *LivingEntity*. One of them is *modifyEnergyBy*, which takes a float as an argument, that resembles the amount of energy that should be added to the current energy. This float number can be positive or negative, modifying the energy in both ways. The other method is by using the function *consumeEnergy* that instead of taking a float as an argument, it obtains an object of type *ActionType* which should encapsulate the amount of energy that should be consumed based on the action performed by the *LivingEntity*.

   The death mechanism is a series of tasks executed by the *die* function in order for the instance to be fully destroyed and emptied. First, it checks if the creature has already been killed, then proceeds to set its *LivingState* to Dead.

Second, remove its smell trail before removing the *LivingEntity*, because the smell nodes are linked to the *LivingEntity*. Third, destroy its visual *GameObject*. Lastly, remove any references to this *LivingEntity* instance in order to be cleaned from memory by the garbage collector.

The *LivingEntity* class also declares a number of abstract methods that need to be implemented by the derived classes in order for the *polymorphism* to function as expected.

3. **Creature**

   This class holds most of the behaviours that influence most of the simulation's outcome. It resembles all the species of animals or insects that exist in the simulation. It is the class that encapsulates the most attributes and which has the most complex behaviour.

   As noted previously, the creatures are able to think for themselves and are not manipulated by an external system. This is possible by the way the project is structured and implemented. The *Coordinator* is sending update signals on a *FixedUpdate* towards the *GridMap*, which will forward this signal to the Living Entities stored in its list. This is done through the call of the *updateBrain* method of the *Living Entities*, which are the creatures in this case. Now the *updateBrain* method has the ability to work as an individual brain for each creature. This brain will think or act after a time interval randomly chosen from an interval determined by the creature's genes. This means that some brains think faster than others.

   The brain thinks with the use of the *act* method, which uses a switch statement to check which state is the creature in and acts accordingly. It works like a *Finite State Machine* [Per90]. There are two types of states used in this context. First, there is the physical state, which refers to what the creature is doing or is going to do. Secondly, there is the brain state, which refers to whether the creature is actively thinking or doing something already.

   (a) **Thinking state**

   This is the state in which the *Creature* is evaluating its needs and checks what need has the biggest priority and changes to the corresponding

state, then consumes energy equivalent to how much energy does thinking consume, plus the modifiers that the genes add. In this state, in order to evaluate how to do achieve fulfilling its need, it updates its sight and smell senses and checks its surrounding sensed entities. This state is also responsible for searching a path to the chosen target by requesting one from the *GridMap*.

(b) **Exploring state**

Is the state that is responsible for exploring. It moves the creature one cell and returns to Thinking state. This movements speed is influenced by the genes of the creatures, making it faster or slower and consuming more or less energy per cell traveled. The movement is random, but not completely. The *Creature* stores its current direction and can only explore forward, to the left or to the right to its current direction. This ensures that the creatures movement is more natural, roughly preserving its direction. The direction is determined using an array of four directions that stores at each position the relative coordinates of the next cell. For example, if a creature want to move up one cell, the direction will be described by the unit vector *(1,0)*.

The same behaviour, but even more natural, can be approached by implementing the direction as a vector stored into the creature and rotate it through the trigonometric circle by an amount of degrees randomly, then transformed into the closest 4-directional unit vector.

(c) **GoingToEat state** In this state, the creature has acquired a path during its Thinking state towards its target food, using the *PathFinder* instance stored in the *GridMap*, and then starts to follow it. If its target food is in a neighbouring cell, it starts to eat it, otherwise, continues to follow its path towards the food. If something blocks its path, it requests a new path from the *GridMap*. Until the target is reached, the *GoingtoEat* state will repeat recursively. Each time it moves a cell, it consumes energy.

(d) **GoingToMate state** The *Creature* turns into this state only after it has been accepted by a female, in the males case, or triggered by a male, in females case. Anytime in this simulation if there is one creature in this state, there

must be another one that is its mate. In this state, these two creatures go towards each other until reached, time when they are going to reproduce and give birth to offspring. Each time it moves a cell, it consumes energy. After reproduction, it also consumes energy.

Creatures actions are performed sequentially and separated into steps using multiple functions. This way, code refactoring is easier to be done.

(a) **Movement and Senses**

Creature's movement is possible only through communication with the environment i.e. the *GridMap*, using their sight and smell senses. When their sight sense is updated, it sends a request to the *GridMap* which returns all the Entities that can be seen by the creature. *GridMap* searches in a box around the creature that has its sides at a distance in cells equal to the creature's sight distance value.

The same approach was used for the smell sense, but instead of using a variable box size, a fixed box size was used. The different sense strength is valued by creatures being able to sense only smells that have an intensity higher than the minimum intensity that can be sensed by a creature, trait influenced by genes.

The smell nodes trail has a variable length depending on the genes of the creatures, being computed using weights, some genes having a bigger impact on how intense is the smell of a creature.

(b) **Priorities of choice**

The way the creatures treat allies and enemies is determined by a function that computes the average difference between all the genes between the asking creature and the target creature. If this difference is below a set value, the target is considered as a similar species or as a friend, otherwise it is considered an enemy or edible.

A creature will have most of the time multiple plants and creatures around it and will have to choose one that will benefit the creature the most. The way it was accomplished was through calculating and obtaining the closest creature and the closest plant for each of the senses and then multiply-

ing those distances with a modifier that implies the food preference gene. After these computations, the closest distance will be chosen and its corresponding creature or plant. In this regard, if a specie is fully carnivore, the distance to the closest creature will be 0, thus, the smallest. Distances to creatures obtained through sight were calculated directly with the formula of distance between two point with coordinates. For distances to creatures obtained through smell the distance is more challenging to be computed as the smell is represented as *SmellNodes* that can be placed in various shape. The distance to the target is computed by the found *SmellNode* by adding recursively the vectors from the current *SmellNode* to the next, until the source is reached, and then adding the distance between the creature and the *SmellNode*. This approach will make sure that smell helps creatures track targets only when sight can not help them.

(c) **Mating**

In this simulation, every creature has a need to reproduce and is represented by a variable which value is between 0 and 1 of type float. Reproduction can only happen if the creature is mature, meaning that the creature is old enough. After reaching the mature age, the urge to reproduce will start to increase. This need will trigger males to start searching for females. If a female is found it is tested and if the female rejects the male, the male remembers that female for a few rejections more in a List which consists of these Creatures. If a male finds a female which he has been rejected by, he ignores it and searches for others. Rejection has a chance of happening which can be fine tuned within the editor. If the male is accepted by the female, a trigger is sent to the female in order to change its current state to *GoingToMate*. This way, both creatures will meet at a middle point in the path and reproduce. Reproduction implies the female giving birth to a random number of children between two set numbers. The children are then passed to the *GridMap* to be correctly spawned in the world. These child creatures have a smaller size which increases every few years that pass, until a mature age is reached.

Chapter 5

Experimental results

The following results are from a larger simulation of 200 by 200 cells over a period of almost 7 hours. The first hour was the most impactful as most of the changes to the genes happened in that time frame.

5.1 Population

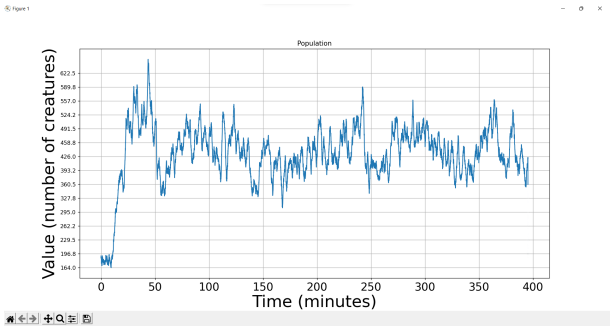

Figure 5.1: Graph result of population number

The population of the species started around 180 individuals and had grown rapidly until over 600 individuals, at which point the population started to decrease. This is due to the fact that in the beginning there was food that could accommodate more than 180 agents. This led to a population exponential growth rate which stopped after the food could not be enough to sustain over 600 individuals. After about 100 minutes, the population started to fluctuate less and be more stable. The fluctuation is determined by the multitude of species that compete constantly and by the randomness of the food generated over time.

5.2 Genes

The results in the graphs below are the average value for each of the genes for the entire population over the course of the simulation.

1. **Brain Speed Gene**

   As shown in the graph, *thinking speed* did not change over time, meaning that there was little benefit from acting faster.

2. **Motor Speed Gene**

   On the other hand, the gene that is influencing the *movement speed* of the creature has a bigger impact on the survival of the species as the majority, but not all, of the creatures ended up being slower, thus, being more conservative on the energy. This technique may be due to the fact that food appears randomly and waiting for it to appear consumes less energy than constantly searching for new food locations, even though there might be fast predators that could catch them.

3. **Food Preference Gene**

   Even though the similarity or the average difference test between genes is set to a very small value, the creatures do not tend to be too aggressive. They have been led to be more *herbivore* as the vegetation may be the more reliable food source, because aggression led to extinction in other tests.

4. **Sight Distance Gene**

   Natural selection has shown that sight is the better to have sense in this environment, because the outcome is more reliable and consistent. This is due to the fact that most of the time the neighbourhood of the creature was more important in this test.

5. **Smell Strength Gene**

   It is noted that the smell was not needed to be better or worse. The optimal value was the already predefined one. In other test in which the smell sense was disabled, it was very hard for the creatures to find each other and thus, leading to extinction because of the inability to reproduce. This leads to the conclusion that smell is in fact very important in the act of survival.

6. **Behaviour Gene**

   Behaviour gene did not seem to be impacted by the environment as the average choice was to be neutral.

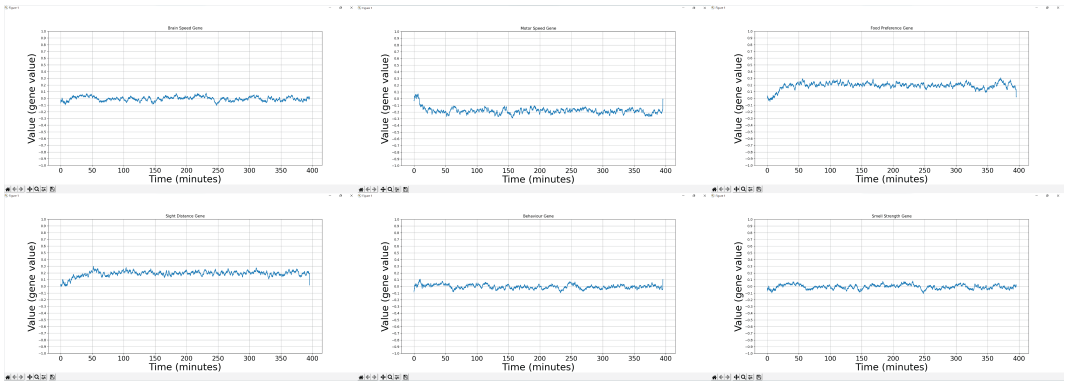

Figure 5.2: Subplots for brain speed, food preference, sight distance, behaviour and smell strength

The fluctuation in the graph results is determined by the fact that natural selection did not choose only a single best local minimum solution for this environment, which means that there are multiple species that continuously live in the same habitat. Interesting to note during the multiple simulations tests is the fact that almost every time there were multiple species living in the same ecosystem at the same time, sustaining biodiversity.

As expected, the genes of the creatures are adapted to the environment over generations through natural selection, resulting in many local optimum solutions.

Another interesting behaviour observed through the numerous tests ran was the tendency of the creatures to slowly group in a certain part of the ecosystem, while the other part is being overpopulated by vegetation. This separation happened a couple of times over the course of hours. The reason for this was the way the creature sense. They request the *GridMap* to find what they can see and the *GridMap*

iterates through a matrix in the traditional way, starting from the left upper corner and ending in the right bottom corner. This resulted in a tendency of creatures to sometimes favor targets that are in the left upper corner, thus, slowly moving towards the left of the environment. This could be partially solved by iterating from both ends one step at a time.

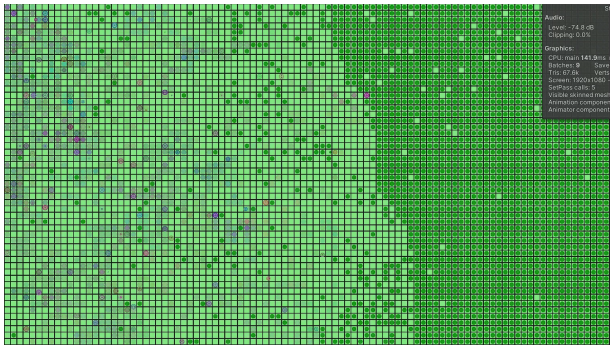

Figure 5.3: Creatures tendency to move to the left

Chapter 6

Conclusion

The most important outcome of this project is the successful approach of implementing a simulation that includes new features that construct complex behaviour. Implementing the smell sense along with the sight sense was successful in terms of implementation as the priority system manages priorities correctly and helps the creatures assess what will the next step be. The different speeds at which the brain of the creatures functions has also shown that acting faster can benefit survival, but being only one out of six genes, it does not grant high chances of survival that could turn the species into dominant ones. However, this approach simulates a more realistic way of treating competition between species. Moreover, the implementation of chromosomes shows a flexible and fast way of adding multiple traits to a category of living being. These chromosomes have also shown that the more genes available, the more likely diversity will arise within the ecosystem. Also, the genes influence behaviour and competitiveness of the agents, while also being able to represent the dominant genes through the use of colors within the graphical interface. Another aspect worth mentioning is the fact that even though there exists competition within the simulation, biodiversity is thriving in the simulated ecosystems, similar to real life.

One remaining question is how would other senses influence the resulting behaviour of a simulation.

Bibliography

[Abb12] Montasir Abbas. Agent-based modeling and simulation. *Artificial Intelligence Applications to Critical Transportation Issues*, page 58, 2012.

[Ale77] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[Bur49] George Bosworth Burch. Anaximander, the first metaphysician. *The Review of Metaphysics*, 3(2):137–160, 1949.

[But82] Samuel Butler. *Evolution, old and new; or, The theories of Buffon, dr. Erasmus Darwin, and Lamarck, as compared with that of mr. Charles Darwin. Op. 4.* London, 1882.

[CO+04] Jerry A Coyne, H Allen Orr, et al. *Speciation*, volume 37. Sinauer Associates Sunderland, MA, 2004.

[DBS06] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[Ed20] Fun Master Ed. Simulating an ecosystem, 2020. `https://www.youtube.com/watch?v=I5ICps2a9vo&t=481s&ab_channel=FunMasterEd`, online; accessed 20 March 2022.

[FGK+21] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, and Eric Gunawan. A systematic literature review of a* pathfinding. *Procedia Computer Science*, 179:507–514, 2021. 5th International Conference on Computer Science and Computational Intelligence 2020.

[Fis58] Ronald Aylmer Fisher. *The genetical theory of natural selection*. , 1958.

[Fre15a] Adam Freeman. The façade pattern. In *Pro Design Patterns in Swift*, pages 325–338. Springer, 2015.

[Fre15b] Adam Freeman. The flyweight pattern. In *Pro Design Patterns in Swift*, pages 339–356. Springer, 2015.

[Fre15c] Adam Freeman. The singleton pattern. In *Pro Design Patterns in Swift*, pages 113–136. Springer, 2015.

[Fre15d] Adam Freeman. The strategy pattern. In *Pro Design Patterns in Swift*, pages 491–502. Springer, 2015.

[FT91] Drew Fudenberg and Jean Tirole. *Game theory*. MIT press, 1991.

[GHJ+95] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[GMS03] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in computer games. *ITB Journal*, 8:57–81, 2003.

[Gou85] Stephen Jay Gould. *Ontogeny and phylogeny*. Harvard University Press, 1985.

[HD] C A.S. Hall and J W Day, Jr. Ecosystem modeling in theory and practice: an introduction with case histories. , 1977. `https://www.osti.gov/biblio/5835861`, online; accessed 15 March 2022.

[HD05] Sandra Herbert and Charles Darwin. *Charles Darwin, Geologist*. Cornell University Press, 2005.

[Hel18] Justin Helps. Simulating natural selection, 2018. `https://www.youtube.com/watch?v=0ZGbIKd0XrM&t=2s&ab_channel=Primer`, online; accessed 12 March 2022.

[Hel19] Justin Helps. Simulating the evolution of aggression, 2019. `https://www.youtube.com/watch?v=YNMkADpvO4w&t=3s&ab_channel=Primer`, online; accessed 12 March 2022.

[Hel21] Justin Helps. Simulating the evolution of sacrificing for family, 2021. `https://www.youtube.com/watch?v=iLX_r_WPrIw&t=230s&ab_channel=Primer`, online; accessed 12 March 2022.

[LAG+16] Michael Lynch, Matthew S Ackerman, Jean-Francois Gout, Hongan Long, Way Sung, W Kelley Thomas, and Patricia L Foster. Genetic drift, selection and the evolution of the mutation rate. *Nature Reviews Genetics*, 17(11):704–714, 2016.

[Lag19] Sebastian Lague. Coding adventure: Simulating an ecosystem, 2019. `https://www.youtube.com/watch?v=r_It_X7v-1E&t=0s&ab_channel=SebastianLague`, online; accessed 20 March 2022.

[Lee61] Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, (3):346–365, 1961.

[Loe17] Michel Loeve. *Probability theory*. Courier Dover Publications, 2017.

[Mas99] Eric Maskin. Nash equilibrium and welfare optimality. *The Review of Economic Studies*, 66(1):23–38, 1999.

[McD17] James E McDonough. Null object pattern. In *Object-Oriented Design with ABAP*, pages 265–269. Springer, 2017.

[PB90] Steven Pinker and Paul Bloom. Natural language and natural selection. *Behavioral and brain sciences*, 13(4):707–727, 1990.

[Per90] Dominique Perrin. Finite automata. In *Formal Models and Semantics*, pages 1–57. Elsevier, 1990.

[SF74] Ronald W Shephard and Rolf Färe. The law of diminishing returns. In *Production theory*, pages 287–318. Springer, 1974.

[SGG97] Chrysostomos D Stylios, Voula C Georgopoulos, and Peter P Groumpos. The use of fuzzy cognitive maps in modeling systems. In *Proceedings of 5th IEEE Mediterranean conference on control and systems*, pages 21–23. Paphos Cyprus, 1997.

[SMG19] Ryan Scott, Brian MacPherson, and Robin Gras. Ecosim, an enhanced artificial ecosystem: Addressing deeper behavioral, ecological, and evolutionary questions. In *Cognitive Architectures*, pages 223–278. Springer, 2019.

[Soc22] National Geographic Society. *Theory Of Evolution*. 2022. `https://education.nationalgeographic.org/resource/theory-evolution`, online; accessed 04 February 2022.

[sof20] softologyblog. Ant colony simulations, 2020. `https://softologyblog.wordpress.com/2020/03/21/ant-colony-simulations/`, online; accessed 04 February 2022.

[Wal07] Alfred Russel Wallace. *Darwinism: an exposition of the theory of natural selection with some of its applications*. Cosimo, Inc., 2007.

[WMM+01] Stuart A West, Martyn G Murray, Carlos A Machado, Ashleigh S Griffin, and Edward Allen Herre. Testing hamilton's rule with competition between relatives. *Nature*, 409(6819):510–513, 2001.