COMP 4106 PROJECT

Connor Raymond Stewart (101041125)
Gabrielle Latreille (101073284)
Abdaljabbar Hersi (100919600)

APRIL 16, 2021
CARLETON UNIVERSITY
Dr. Matthew Holden

0 – Abstract

The report herein is information about a group project for COMP 4106 Artificial Intelligence, an AI course at Carleton University taught by Matthew Holden over the Winter of 2021. This report aims to introduce, describe, and explain the results of a final group project based on the course. Contained in the report herein are introductions to our group's project topic, our motivations for exploring the topic, and an explanation of the theory and methods used to solve it. A discussion of the project results and future implications or directions for the project are also touched on. The report also consists of a user manual to describe how to use the project code and its functionality. The project outcome was positive, with successful results yielded as a function of the project code.

Table of Contents

1 – Intro

The chosen project was creating an evolution simulator inspired by evolution games. The simulator the project was based around models an environment by using a discrete event loop. The environment itself consisted of a matrix which individual numbers on the matrix representing tiles in a two-dimension map. Organisms on the tiles can move around and collect food based on a set of organism attributes. The organisms attempt to collect food and safely make it across tiles. If the organism fails to collect any food or the tile's dangers are too great, they die. If the organisms make it to the end of the day, they have a chance of reproducing. Every unit of food it can collect increases its desirability to the general population. Children of an organism inherit some of the same attributes as their parents, saying the simulation only model's sexual reproduction (no asexual reproduction exists). There is a 10% chance a mutation will occur in the offspring, which will modify their distribution of character attributes.

Figure 1 below shows an example of the simulated environment, the world is a 6X6 grid with the population being initialized to 6 organisms. Also, each tile has an attribute associated with it (e.g., Food, danger) even though it is not being shown in the figure above. Exactly how everything comes together will further be elaborated on in the sections below.

Organism				Organism	
		Organism			Organism
	Organism			Organism	

Figure 1 Environment Visual

2 – Motivations

The problem's motivation is to create a simulated evolution environment to refine the best set of traits to increase our organisms' survival using the genetic algorithm. The genetic algorithm simulates how real-world microevolution transpires and how organisms can evolve to have vastly different traits over short periods than their ancestors. This simulation can further explain how humans and other species evolved to have today's traits, which is very exciting. For example, most dog breeds originate from the last one hundred and fifty years and are the by-product of human breeding; natural selection occurs similarly but at a different pace. Thus, the simulation above represents what may happen if animals get introduced onto an island they never inhabited before. Therefore, traits that evolve in the island's organisms will be tweaked with each passing generation until we reach an optimal equilibrium.

3 – Description of Methods

3.1 – A.I Technique

We used a genetic algorithm to determine which organisms survived and reproduced; the steps we followed are as follow:

a) Randomly populate our world with organisms
b) Once a simulated day goes by, evaluating the fitness function of every organism in our population, which is f(x) = age * food gathered in a day. The idea is that the organism that can survive longer and gather more food is a better organism than those that do not.
c) We then select pairs of organisms to reproduce randomly. Our genetic operator id g(x, y) = randomly select attributes from x and y to create a new organism.
d) We call our mutation function, which is m(x) = randomly increase one attribute by 1 to 5; mutations only occur in 10% of the new organisms.
e) We then eliminate organisms from the population based on their fitness function. Additionally, an organism with fitness of zero will get eliminated from the population. The idea is that the fitness function will only be 0 if the organism was unsuccessful in collecting food and the unsuccessful organism starves and dies (natural selection).
f) Repeat the steps above.

3.2 – Organisms

The organisms have a finite number of points to allocate to their attributes. Organism attributes include endurance, strength, agility, and intelligence. Endurance enables an organism to move across more tiles each time the discrete event loop iterates. Strength enables an organism to unlock tiles to collect food. Agility determines an organism's ability to evade and defend against dangers on a tile. Finally, intelligence increases the food collected per tile by an organism.

Each turn of the discrete event loop, each organism will undergo a check to determine if they are still alive. Organisms' statistics also determine how much food they collect if they are on a tile with food. Organisms possess an inertia value, meaning they maintain momentum during movement across tiles rather than blitz across tiles and then

pause instantly. The inertia simulates a dynamic movement of organisms flowing across the grid environment.

Food collected by the organism is scaled based on a predefined logarithmic formula. The formula is conditional based on a conditional formula:

f = [food *math.log(intelligence/difficulty, multiplier)]

Where if the result f is positive, the organism collects the value of food the formula returns. If the formula's result f returns negative, the organism collects no food. The formula produces a minimum intelligence requirement to collect food - based on the tile's 'difficulty' value - since logs of numbers less than one are harmful; however extra intelligence multiplies the food collected. An intelligence that is too high results in diminishing returns, though, since a logarithmic scale tends to be asymptotic where a significant increase in intelligence results in a minimal increase in food. The diminishing returns suggest overinvesting in intelligence is a waste of organism statistics.

Organisms have a random chance of death based on the 'danger' value of the tile the organism is passing over. The age of an organism also increases the chance of death proportionally, meaning older organisms have a higher chance of dying. The formula above is a product of Euler's-number, meaning an exponential growth in the probability of dying increases with proportion to [age * (danger - agility)]. Therefore, the formula is e^[age * (danger - agility)]. 'age' represents the organism's current age in the simulation, danger represents the danger value of the tile it has just passed. Agility represents the organism's chance of avoiding danger. Slight differences between agility and danger only slightly increase the risk of dying, whereas significant differences make death an almost inevitable outcome. Therefore, there becomes a trade-off between agility and danger since organisms will find a balance between the chance of dying and putting their points elsewhere. More so, agility will increase the organism's longevity since the organism's age increases the chance of dying. All the above reasons result in agile organisms living far longer, but the trade-off is that extra points in agility go to waste if the danger level in the environment is not that high.

Strength and endurance are straightforward mechanics compared to intelligence and agility. Strength unlocks an organism's ability to collect food from a tile, and endurance changes how many tiles an organism can cross per turn. Since endurance also has an inertia value, if an organism attempts to cross a tile with a higher movement value, the organism's endurance and organisms can 'build' up inertia over time to cross a hard-to-pass tile. If an organism cannot cross a tile at any given turn, they store it as inertia. There is a trade-off for strength and endurance like the other traits. Strength can unlock extra tiles to get food from; but having too high strength may not be worth losing points in other areas. For example, if an organism has no endurance and only strength,

all the food will be taken by the time it reaches any tiles. Likewise, if endurance is high and strength is minimal, an organism will not harvest any food to live, even though it makes it to the food tiles before any other organism.

3.3 – Environment/World Explained

The simulation's environment is a two-dimensional matrix-based environment, which represents the world the organisms inhabit. Each tile within the matrix is randomly assigned a set of values representing if they have food, how strong an organism must be to collect food, and difficulty it takes to cross the tile, and the tile's danger value to cross the tile. Since all the matrix variables are randomly assigned, the discrete event loop begins to function as a simulated Markov-chain-based environment. Initial probabilities determine the world the organisms live in, and random mutations and genetic inheritance patterns determine an organism's traits. More so, random values assign an organism's chance of death per turn. Hypothetically, we see that the entire simulation is representable as a Markov chain, where each iteration of the discrete event loop causes all organisms within the Markov chain to move around based on a random probability value.

For the simulation, we made many simplifications to highlight how the organism attributes change on a micro-evolutionary scale. Many simplified things include organism path-making, organisms fighting each other, organisms cooperating, sexual selection, and damaging/helpful mutations. Organisms cannot choose the path they make; they wander around at random and hope to find food. Intelligence does not modify path-making, and organisms can get unlucky and never find food during a simulated day. Furthermore, organisms do not fight each other, meaning a high-strength organism will not attempt to rob food from other organisms or kill off competition. Also, organisms cannot cooperate either, meaning no teams or clans get made in the simulation. Organisms wander past each other and ignore each other's existence. Organisms do not engage in Darwinian sexual selection, which is a principle that states organisms 'choose' their mates based on attractiveness, their ability to protect offspring, and other characteristics. Sexual selection is not modelled in the simulation, meaning a perfectly viable organism will 'randomly' choose to mate with other organisms. Thus, perfectly designed organisms can choose to produce offspring with organisms that can hardly survive (or have only survived up to this point through luck). Lastly, we do not model harmful or beneficial mutations; all mutations are neutral in a sense. Organisms cannot *gain* new points through a mutation, nor can they *lose* points; they can only

shuffle the points they have. The simulation represents a more micro-evolution scale - like how humans bred dogs and plants - rather than the macro-scale of evolution that would turn bacteria-sized organisms into complex animals. The timescale for evolution in the simulation is over-simplified as well, real-world organisms cannot evolve traits as quickly as is shown in the simulator, and mutations do not work in an environment this constrained. The purpose is to show how microevolution can occur in an organism over a small number of generations (like what occurred with dog breeds).

Each day within the simulation lasts for twenty-four hours, with organisms collecting food four twelve hours per day. The organisms collect food for twelve hours. Each iteration of the discrete event loop represents an hour, meaning twelve iterations of the loop occur before the end of the day. Upon the day's end, a thirteenth step occurs where organisms reproduce and get ready for the next day. The day repeats with the same setup as the previous day, and our simulation runs for 100 days. After 100 days, the simulation ends, and the program outputs the results onto plots.

4 – Results

The below figures show the two graphs; each first shows the average endurance, strength, agility, intelligence, and age on the y-axis and days on the x-axis. The second graph shows the population (# of organisms in our environment) on the y-axis and days on the x-axis. An expected pattern shows up in most of our runs in which, in the first few days, the population of our environment dwindles drastically before picking back up and growing exponentially. The patterns seen in the graphs align with what the group expected since the organisms only reproduce if they survive. As anticipated, in a few test cases, the population goes to zero; this was since the organisms are given attributes randomly at initialization and might be unlucky.

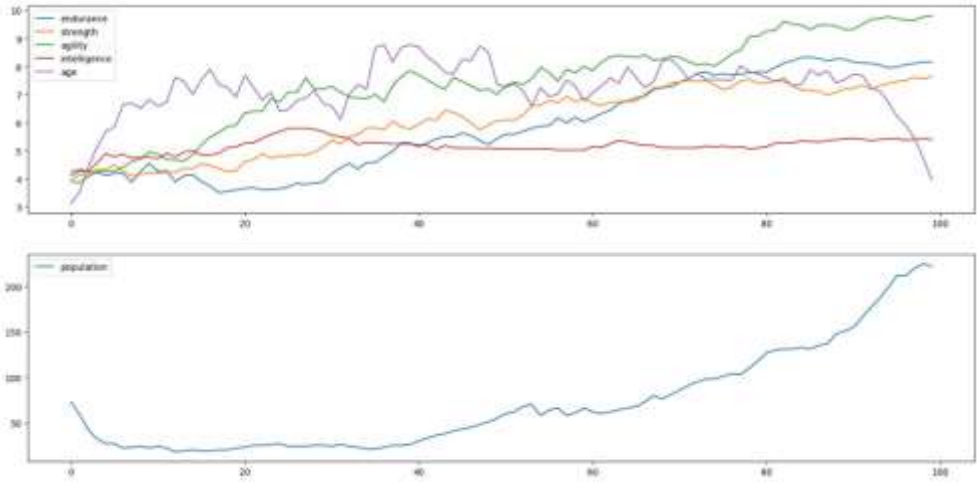

Figure 2 Evolution Plot 1

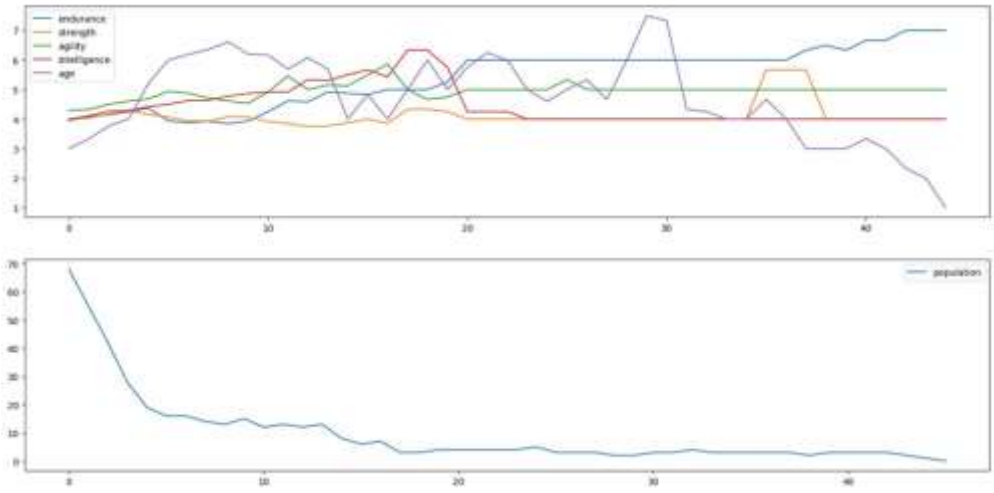

Figure 3 Evolution Plot 3

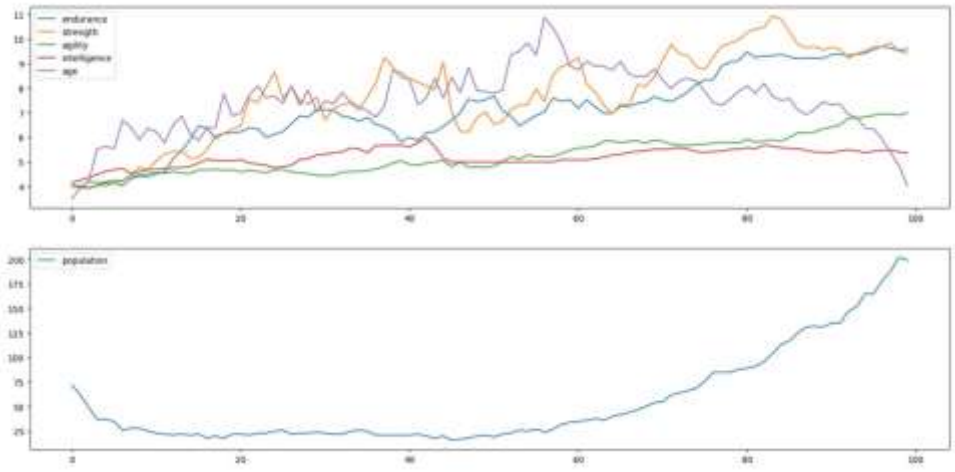

Figure 4 Evolution Plot 4

5 – Discussion of the implications of the work

The implications of the work show that it is possible to simulate micro-evolution within a computation environment if the variables and mechanisms behind the evolution are known. The tool shows that it is possible to simulate the results or probabilistic evolution using a discrete event Markov simulation environment within a computer. The mathematical nature of the tool can significantly lower the costs of running experiments to determine the result of biology experiments within self-enclosed ecosystems. For example, rather than running microbiology experiments on a petri dish or ecology experiments on islands and biodomes, initial stages of testing could be used with the application. More complexity would be added to the model for researchers to find the helpful tool, but such features could get introduced in future iterations. Also, the tool has robust implementations in an educational environment since the tool provides a more concrete way of showing the relationship between microevolution and mathematics than

a lab-based environment would show. Many students misunderstand the relationship between evolution and algorithms, probability, and discrete Markov-chains; however, the project could provide a more transparent way of showing how evolutionary phenomena occur in a mathematical setting. The tool provided by the project can help train future students to understand biology in a mathematical/informational context, which is much more helpful to our digital world than lab-based approaches.

6 – Directions for future work

Future implications of the work can include adding to the work created, studying the results of the experiments further, and creating easy-to-understand models for the results. One of the primary goals for future implications of the work would be to keep adding to the project to build increasingly sophisticated models for the evolution simulation. More complex models can allow controlling for more variables that would affect microevolution in the real world. These extra variables include examples like sexual selection, harmful and beneficial mutations, or predator-prey relationships. The project would allow for models depicting the dynamic relationship between organisms in a real-world environment and how the equilibrium between genetic traits and the organism's population gets established.

Studying the results of these types of experiments further can allow for the discovery and gentle explanation of various ecological and micro-genetic phenomena. The project can be helpful either as an educational tool (which shows a simulated ecosystem evolving for students) or as a scientific model to predict environmental equilibria in isolated environments. As an educational tool, this model can help train future elementary and high school students to see how natural selection panes out in a real-world environment. It can also help train university students by showing the mathematical and probabilistic relationship between micro-evolutionary phenomena and an organism's environment. If the project's development results in a sufficiently complex tool, ecologists or microbiologists could use the tool to predict the outcomes of placing a set of organisms in an isolated environment (like placing deer and wolves onto an island or placing bacteria on a petri dish). A complex simulation could help to model future events to predict how a successful test or experiment will occur for the researchers.

In either event, the tool is useful mainly as a computational aid for researchers or students. The general population would not be interested in this tool outside of a limited number of people who have casual interests in ecology or micro-evolution. Therefore, future implementations would revolve around building more complicated models for the simulation to cater to students and researchers. The model could have a complexity slider built-in so that simpler models - for elementary and high school students - are possible to show. The program can also execute complex models if need be, like simulating a researcher's ecosystem.

7 – User Manual

The code written for this project can be found here:

https://github.com/GabrielleSL/COMP4106Project.

To run it, use the command `python game.py` in the same folder as the game.py file. The code was written and tested with python 3.8. NumPy and matplotlib need to be imported for the code to run.

The program will ask the user how many days to run the simulation for (expects an integer) and outputs the result as a plot as seen in the 'Results' section above. If the population falls to 0 before it reaches the number of days specified, the simulation will stop there and output the resulting graph.

8 – Statement of contributions

Overall, the project was completed in group meetings through discord. All group members made significant contributions to the overall project, with approximately equal contribution among each member. Since the group work was evenly divided amongst members and completed in a shared environment, all members of the group completed all aspects of the assignment together.

9 – REFERENCES

[1]Ha, D. (2017, October 29). *A Visual Guide to Evolution Strategies*. Retrieved from https://blog.otoro.net/2017/10/29/visual-evolution-strategies/

[2]Holden, M. (2021). *Genetic Algorithm.* Ottawa: Carleton University .

APPENDIX – CODE USED FOR PROJECT

```
from evolution import Evolution
import matplotlib.pyplot as plt

# initialize world
country = Evolution(100, 100)
country.intialize_population()

# run the simulation through a single day
def day():
temp_pop = []
# 12 hours of day
for _ in range(12):
temp_pop = []
# for every organism in the country
for org in country.population:

organism, loc_x, loc_y = org
# organism has a turn: eats food, moves, dies, etc.
new_x, new_y = org[0].turn(country.world.get_tile(loc_x, loc_y)[0], loc_x, loc_y)

if new_x is None and new_y is None:
    a. continue
elif not organism.alive:
    a. country.world.set_organism(loc_x, loc_y, None)
    b. continue
temp_pop.append([org[0], new_x, new_y])
country.world.set_organism(loc_x, loc_y, None)
# END for
# END for
genetic_population = country.genetic_algorithm(temp_pop)
new_population = country.populate(genetic_population)
country.population = new_population
return new_population
# END day
# create plots for the simulation
def simulation_plot(populations):
endurance_avg = []
strength_avg = []
agility_avg = []
intelligence_avg = []
age_avg = []
pop_size = []
attribute_size = len(populations)
for pop in populations:
endurance = []
strength = []
agility = []
intelligence = []
age = []
pop_size.append(len(pop))
if len(pop) == 0:
attribute_size = len(populations) - 1
break
# END if
for org in pop:
endurance.append(org[0].endurance)
strength.append(org[0].strength)
agility.append(org[0].agility)
intelligence.append(org[0].intelligence)
age.append(org[0].age)
# END for
endurance_avg.append(sum(endurance) / len(endurance))
strength_avg.append(sum(strength) / len(strength))
agility_avg.append(sum(agility) / len(agility))
intelligence_avg.append(sum(intelligence) / len(intelligence))
age_avg.append(sum(age) / len(age))
# END for

# create plots
# attributes plot
t = range(0, attribute_size)
plt.subplot(2, 1, 1)
plt.plot(t, endurance_avg, label='endurance')
plt.plot(t, strength_avg, label='strength')
plt.plot(t, agility_avg, label='agility')
plt.plot(t, intelligence_avg, label='intelligence')
plt.plot(t, age_avg, label='age')
plt.legend()

# population plot
plt.subplot(2, 1, 2)
t = range(0, len(populations))
plt.plot(t, pop_size, label='population')

plt.legend()
plt.show()
# END day_plot
# user input
print("How may days to run the simulation:")
x = input()
# create array with day 0 population
days_population = [country.population]
for i in range(int(x)):
pop = day()
days_population.append(pop)
if len(country.population) == 0:
print(f"Population reached 0 at day {i}. Organism was wiped out.")
break
# END if
# END for

simulation_plot(days_population)
```

```
from worldMap import World
from organ import Organism
import numpy as np
import random

class Evolution:
def __init__(self, width, height):
self.width = width
self.height = height
self.population = []
self.world = World(width, height)

# fitness function for the genetic algorithm
# f(x) = x.age * x.food
def fitness(self, organism):
return organism.age * organism.food

# initialize population with a random value between 3 and 5 for each attribute
def intialize_population(self):
for i in range(self.width):
organ = Organism(random.randint(3, 5), random.randint(3, 5), random.randint(3, 5),
    i. random.randint(3, 5))
# randomly assign the location of the organism on the map
locx = random.randint(0, self.width - 1)
locy = random.randint(0, self.height - 1)
self.world.set_organism(locx, locy, organ)
# add organism to the population
self.population.append([organ, locx, locy])

def populate(self, population):
newp = []
for org in population:
locx = random.randint(0, self.width - 1)
locy = random.randint(0, self.height - 1)
newp.append([org, locx, locy])
self.world.set_organism(locx, locy, org)

return newp

def genetic_algorithm(self, population):
# fitness function
sorted_pop_fitness = self.population_fitness(population)
# genetic_operator
pop_size = len(sorted_pop_fitness)
r_sample = random.sample(range(0, pop_size), int(pop_size / 2))
new_orgs = []
for i in range(0, len(r_sample), 2):
new_org = self.genetic_operator(sorted_pop_fitness[i][0], sorted_pop_fitness[i + 1][0])
# mutate
self.mutate(new_org)
new_orgs.append(new_org)
# remove members with a fitness of 0
remaining_pop = list(filter(lambda x: x[1] != 0, sorted_pop_fitness))

new_pop = []
for org in remaining_pop:
org[0].age += 1
org[0].food = 0
new_pop.append(org[0])

for org in new_orgs:
new_pop.append(org)

return new_pop

# egnetic operator of the genetic algorithm
# randomly uses the attributes from each parent
def genetic_operator(self, x, y):
temp = [x, y]
new_org = Organism(temp[random.randint(0, 1)].endurance, temp[random.randint(0, 1)].strength,
    i. temp[random.randint(0, 1)].agility,
    ii. temp[random.randint(0, 1)].intelligence)
return new_org
# mutates one of the organism's attribute by a random factor of 1 to 5
def mutate(self, org):
chance_of_mutation = random.randint(0, 100)
# 10% chance of mutation
if chance_of_mutation <= 10:
choice = random.randint(0, 3)
if choice == 0:
    a. org.endurance += random.randint(1, 5)
elif choice == 1:
    a. org.strength += random.randint(1, 5)
elif choice == 2:
    a. org.agility += random.randint(1, 5)
elif choice == 3:
    a. org.intelligence += random.randint(1, 5)
# sort the population by it's fitness
def population_fitness(self, population):
if len(population) > 0:
temp = []
for org in population:
    a. temp.append([org[0], self.fitness(org[0])])
temp2 = np.array(temp)
return temp2[temp2[:, 1].argsort()]
else:
return []
```

```
import math
import random
"""
Name:
Organism
Input:
endurance - this is the amount of space an organism can cross per turn
strength - this determines if an organism can unlock a food tile
agility - this determines an organisms defences
intelligence - this multiplies the food collected per tile
Output:
Instantiates a data structure which describes an organism
Description:
Used to create an organism and define its properties for an evolutionary simulation
Note that the organisms inertia represents built up momentum from a previous or current turn
Essentially, if an organism attempts to cross a difficult tile, it can build movement across turns to do so
Also, if an organism has leftover movement after crossing a tile, it can use excess inertia to cross more tiles
"""
class Organism:
def __init__(self, endurance, strength, agility, intelligence):
self.endurance = endurance
self.strength = strength
self.agility = agility
self.intelligence = intelligence
self.food = 0  # the amount of food gathered in a day
self.age = 1  # its age in days
self.inertia = 0  # this is the amount of movement built up over time
self.alive = True  # flags the organism as alive
# END __init__
"""
Name:
turn
Input:
tileStats - refers to the current tile space the organism is on
locx and locy - location of the organism on the map
Output:
Determines if the organism crosses the tile and how much food it collects from the tile
Returns the new location of the organism if it was able to survive
Description:
Used to determine if an organism successfully crosses a tile alive and how much food it gets by doing so
"""
def turn(self, tileStats, locx, locy):
# checks if we can enter the tile this turn
if self.endurance + self.inertia >= tileStats.cross:
# add any extra movement for the organism
self.inertia += self.endurance - self.inertia
# report that the organism has entered the tile
else:  # otherwise we build inertia
self.inertia += tileStats.cross
# report nothing has changed this turn
return [None, None]
# END IF
# checks our chance of death this turn
death = self.chanceOfDeath(self.agility, tileStats.danger)
if death > random.uniform(0, 100):
self.alive = False  # flags the organism is now dead
# report no food was collected this turn
else:
# this means that the tile has food on it
if tileStats.food != 0:
    # this checks if the organism is strong enough to collect from this tile
    if self.strength >= tileStats.food_diff:
    # finds the amount of food collected as a product of intelligence and tiles complexity
    yield_ammount = self.foodYield(tileStats.food, self.intelligence, tileStats.difficulty, 2)
    self.food += yield_ammount
#     END IF
# END IF
choice = random.randint(0, 1)
if choice == 1:
    if random.randint(0, 1) == 0:
    return [locx + 1, locy + 1]
    else:
    return [locx - 1, locy + 1]
else:
    if random.randint(0, 1) == 0:
    return [locx - 1, locy - 1]
    else:
    return [locx + 1, locy - 1]
return [None, None]
# END turn
# calculates - on a logarithmic scale - the amount of food yielded on the current tile
def foodYield(self, food, intelligence, difficulty, multiplier=2):
if math.log(intelligence / difficulty, multiplier) < 0:
return 0
else:
return food * math.log(intelligence / difficulty, multiplier)
# END foodYield
# calculates - on an exponential scale - the chance that the organism has died
def chanceOfDeath(self, agility, danger):
return math.exp(self.age * (danger - agility))
# END chanceOfDeath
# END Organism
```

```
1   # list of imported libraries
2   import random
3   class Tiles:
4       def __init__(self):
5           # difficulty of crossing tile
6           self.cross = random.randint(0, 4)
7           # amount of food on the tile
8           self.food = random.randint(0, 4)
9           # danger level of the tile
10          self.danger = random.randint(1, 4)
11          # difficulty level of getting the food
12          self.food_diff = random.randint(1, 4)
13          # difficulty level of the tile
14          self.difficulty = random.randint(1, 4)
15      # END __init__
16  # END Tiles

17  class World:
18      def __init__(self, width, height):
19          self.width = width
20          self.height = height
21          # 2D map of the world: [Tile, Organism]
22          self.data = [[[Tiles(), None] for y in
    range(height)] for x in range(width)]
23      # END __init_

24      def set_organism(self, x, y, value):
25          self.data[y][x][1] = value
26      # END set_organism

27      def get_tile(self, x, y):
28          return self.data[y][x]
29      # END get_tile
30  # END World
```