# Assignment II: TUEvolution

Julian Gootzen 1676512
Tycho Brouwer 1753320
Group 17

February 2025

## 1 Question 1

*Extend the code with size and speed mutations, such that the **question1.toml** scenario can be simulated.*

It was chosen to make the new implementation of size and speed with mutations the default, this means detecting the old implementation and translating it to an equivalent in the new implementation. This was done such that if the data type of the variable was equal to an integer the **variations** were set to [**0**] and the **probabilities** to [**1**], if the provided speed or size is an object we set the creature variable to the provided variable, as shown in the code snippet below

```
class App:
  def __init__(..., creature_size, creature_speed, ...):

    ...

    if isinstance(creature_size, int):
      self.creature_size = {}
      self.creature_size["init"] = creature_size
      self.creature_size["variations"] = [0]
      self.creature_size["probabilities"] = [1]

    elif isinstance(creature_size, object):
      self.creature_size = creature_size

    if isinstance(creature_speed, int):
      self.creature_speed = {}
      self.creature_speed["init"] = creature_speed
      self.creature_speed["variations"] = [0]
      self.creature_speed["probabilities"] = [1]

    elif isinstance(creature_speed, object):
      self.creature_speed = creature_speed

    ...
```

The size and speed mutations are implemented in the **Creature** class using the **size_evo_data** and **speed_evo_data** (size and speed evolution data). The **reproduce** function was changed to create a new creature using the mutated size and speed values, the mutation is calculated in the **mutate** helper function using **numpy.random.choice** and using the **variations** and **probabilities** of the parent creature.

```python
class Creature:
  def __init__(self, size_evo_data, speed_evo_data, ...):
    self.radius = max(size_evo_data["init"] // 2, 2)
    self.size_evo_data = size_evo_data
    self.speed = max(speed_evo_data["init"], 1)
    self.speed_evo_data = speed_evo_data

    ...

  def reproduce(self):
    """
    Reproduce a new creature with slight variations.

    Returns:
    Creature: A new creature with slightly varied attributes
    """
    new_size_evo_data = self.mutate(self.size_evo_data)
    new_speed_evo_data = self.mutate(self.speed_evo_data)

    return Creature(new_size_evo_data, new_speed_evo_data, ...)

  def mutate(self, attribute_data):
    """Helper function to mutate a given attribute."""
    mutation = numpy.random.choice(attribute_data["variations"], p=
        attribute_data["probabilities"])

    mutated = max(attribute_data["init"] + int(mutation), 0)

    return {
      "init": mutated,
      "variations": attribute_data["variations"],
      "probabilities": attribute_data["probabilities"],
    }
```

## 2 Question 2

*Add two histogram plots showing the size and speed distributions.*

A new class for the histogram was created with arguments **xlabel**, **ylabel**, **barcolor**, and **fontsize**. The data array is initialised as an empty array. The class has member functions **add**, **clear**, **draw_grid**, and **draw**.

```
class Histogram:
    """
    A class to represent a histogram.
    """

    def __init__(self, *, left=None, top=None, width=None, height=None, border=None, xlabel, ylabel, barcolor, fontsize):
        """
        Initialize a Histogram object.

        Parameters:
        left (int, optional): The left position of the histogram. Defaults to None.
        top (int, optional): The top position of the histogram. Defaults to None.
        width (int, optional): The width of the histogram. Defaults to None.
        height (int, optional): The height of the histogram. Defaults to None.
        border (int, optional): The border size of the histogram. Defaults to None.
        xlabel (str): The label for the x-axis.
        ylabel (str): The label for the y-axis.
        barcolor (tuple): The color of the bars in the histogram.
        fontsize (int): The font size used in the histogram.
        """
        self.left = left
        self.top = top
        self.width = width
        self.height = height
        self.border = border

        self.xlabel = xlabel
        self.ylabel = ylabel
        self.barcolor = barcolor

        self.xmin = 0
        self.xmax = 1

        self.data = []
        self.font = pygame.font.SysFont('Arial', fontsize)
```

The **add** function adds a data point to the histogram by appending it to the **self.data** class variable.

```
class Histogram:
  def add(self, value):
    """
    Add a data value to the histogram.

    Parameters:
    value (int or float): The value to add to the histogram.
    """

    if isinstance(value, int):
      self.data.append(value)
```

The **clear** function clears the histogram data for resetting the plot between generations, the function resets the **self.data** class variable to an empty array.

```
class Histogram:
  def clear(self):
    """
    Clear the current data of the histogram
    """

    self.data = []
```

The **draw_grid** function draws the graph box, labels, and axis ticks. The function takes arguments for the **screen**, **bin_edges** (the values of the bins to display), **spacing** (spacing between bin edges), and **max_y_value** (maximum y value of the histogram to display on the y-axis).

The border and axis labels are drawn to the screen very similar to the given **XY** graph class.

The number of bins is inferred from the provided **bin_edges**. Using the **bin_edges** the x-axis labels are drawn where their position is determined using the total number of bins to display.

The y-axis labels are drawn using the **max_y_value** where the number of labels is determined using the **num_y_labels**, this was added to not overcrowd the y-axis with labels. For consistent displaying of the markers, **max_y_value** should nicely divide by **num_y_labels** - 1.

```
class Histogram:
  def draw_grid(self, screen, bin_edges, spacing, max_y_value):
    """
    Draw the grid for the histogram and display axis values.

    Parameters:
    screen (pygame.Surface): The screen to draw on.
    bin_edges (array) The edges of the bins.
    spacing (int): The spacing between the bins.
    max_y_value (int): max y value of the histogram.
    """
    pygame.draw.rect(screen, ...)

    # x and y axis labels
    label = self.font.render(self.xlabel, True, utils.color('black'))
    screen.blit(label, ...)

    label = self.font.render(self.ylabel, True, utils.color('black'))
    label = pygame.transform.rotate(label, 90)
    screen.blit(label, ...)

```

```
    bins = len(bin_edges)

    # Draw x-axis values using the provided xmin and xmax
    for i, edge in enumerate(bin_edges):
      label = self.font.render(f'{int(edge)}', True, utils.color('black'))
      x_pos = self.left + self.border + (i + 0.5) * (self.width - 2 * self.border) // bins - label.get_width() // 2
      y_pos = self.top + self.height - self.border
      screen.blit(label, (x_pos, y_pos))

    # Draw y-axis values
    num_y_labels = 6

    for i in range(num_y_labels):  # Draw 5 evenly spaced y-axis labels
      y_val = int((float(i) / (num_y_labels - 1)) * max_y_value)
      label = self.font.render(f'{y_val}', True, utils.color('black'))
      x_pos = self.left + self.border - label.get_width() - 5
      y_pos = self.top + self.height - self.border - (i * (self.height - 2 * self.border) // (num_y_labels - 1)) - label.get_height() // 2
      screen.blit(label, (x_pos, y_pos))
```

The **draw** class function draws the complete histogram to the screen. It first determines the **bin_edges** using a range from the minimum to maximum in the dataset where the **hist_values** are the number of occurrences of the **bin_edges** in the dataset.

For pretty displaying of the histogram, consistent zero values should be skipped to ensure no big voids are in the histogram if it has for example values at 50, 55, 60, and 65. This was done by determining if the spacing between non-zero values is the same or divisible by the minimum spacing for all bin edges. The **bin_edges** and **hist_values** are updated to skip the zero values if a spacing larger than 1 is found.

The **max_y_value** is set to the maximum value of the dataset rounded up to the nearest multiple of 5.

Finally, the histogram is drawn to the screen by first calling the **draw_grid** function and then for each value in **hist_values** drawing a rectangle with the right proportions using **pygame.draw.rect**.

```
class Histogram:
  def draw(self, screen):
    """
    Draw the histogram on the screen.

    Parameters:
    screen (pygame.Surface): The screen to draw on.
    """
    if not self.data:
      return

    bin_edges = list(range(min(self.data), max(self.data) + 1))
    hist_values = [self.data.count(x) if x in self.data else 0 for x in bin_edges]

    non_zero_bin_edges = [edge for i, edge in enumerate(bin_edges) if hist_values[i] > 0]
    non_zero_hist_values = [value for value in hist_values if value > 0]



```

```python
    # get diffence between successive bin edges
    spacing = numpy.diff(non_zero_bin_edges).tolist() if len(non_zero_bin_edges) > 1 else [1]
    min_spacing = min(spacing)

    # if the spacing between bins % min(spacing) is same for all bins, then we can use that as the spacing
    if len(set([x % min_spacing for x in spacing])) == 1 and min_spacing > 1:
      spacing = min_spacing
      bin_edges = non_zero_bin_edges
      hist_values = non_zero_hist_values
    else:
      spacing = 1

    # Get the maximum y value for the histogram
    max_y_value = max(hist_values) if len(self.data) > 0 else 1
    # Round up to the nearest multiple of 5
    max_y_value = (max_y_value - 1) // 5 * 5 + 5

    self.draw_grid(screen, bin_edges, spacing, max_y_value)

    bin_width = (self.width - 2 * self.border) / len(bin_edges)

    for i, hist_value in enumerate(hist_values):
      bar_height = (hist_value / max_y_value) * (self.height - 2 * self.border)
      bar_rect = pygame.Rect(
        self.left + self.border + i * bin_width,
        self.top + self.height - self.border - bar_height,
        bin_width - 2,
        bar_height
      )
      pygame.draw.rect(screen, self.barcolor, bar_rect)
```

To display the histograms in the **Cycler** the following was added to the **App** class init function to initialise the histogram plots with the starting population data by looping through the population. And finally, including them in the cycler graphs array to display them.

```python
class App:
  def initialize(self):
    ...

    self.size_hist = graphs.Histogram(xlabel='Size',
                                      ylabel='Number of creatures',
                                      barcolor=utils.color('royalblue'),
                                      fontsize=self.font_size)
    for _ in range(self.population):
      self.size_hist.add(self.creature_size['init'])

    self.speed_hist = graphs.Histogram(xlabel='Speed',
                                       ylabel='Number of creatures',
                                       barcolor=utils.color('royalblue'),
                                       fontsize=self.font_size)
    for _ in range(self.population):
      self.speed_hist.add(self.creature_speed['init'])

```

```
    self.graphs = graphs.Cycler(left=self.sim_width,
                                top=0,
                                width=self.graph_width,
                                height=self.graph_height,
                                border=self.border,
                                graphs=[self.population_graph, self.food_graph, self.size_hist, self.speed_hist],
                                font_size=self.font_size)
```

The histograms are updated in the **update** function of the **App** class. When the next generation is generated in the **update** function the histograms are cleared and then the data for every creature in the new generation is added to the histograms.

```
class App:
  def update(self):
    ...

    # End of day/generation check
    if (self.world.end_of_day() or all((creature.is_home() or creature.has_perished()) for creature in self.creatures)):
      ...

      # Next generation
      if self.generation < self.generations:
        ...

        self.creatures = next_generation

        self.size_hist.clear()
        self.speed_hist.clear()

        for creature in self.creatures:
          self.size_hist.add(creature.size_evo_data['init'])
          self.speed_hist.add(creature.speed_evo_data['init'])

        ...
```

# 3 Question 3

A sense trait was added in the same framework as the speed and size traits. The mutation of the sense trait also works exactly like the mutation of the already existing traits. The **update** function has a new action, when the creature is exploring the creature will sense its surroundings.

```python
class App:
  def update(self):
    ...

    # Actions
    for creature in self.creatures:
      if creature.is_home() or creature.has_perished():
        continue

      if creature.energy < 0:
        creature.perish()
        continue

      if creature.is_exploring() and creature.sense > 0:
        creature.sense_surroundings(self.creatures, self.food)

      ...
```

When a creature is sensing, the following function is executed. The function works in the following way

- Energy usage is calculated, the division by 5 has arbitrarily been chosen to make sensing less beneficial.
- The position of all possible predators (all creatures) are compared to the position of the creature itself, if a creature is within sensing radius and more than 20 percent larger. The creature runs away in the opposite direction of the predator. If a predator is found the function returns after setting the new destination.
- If no predators are within sensing range, the function checks whether the creature is already targetting something, food or a predator. If it is already tracking, the function returns.
- If the creature is not tracking anything, the position of all food pieces is compared to the creature's position. If any piece of food is within sensing range, the creature moves towards it. The creature does not have any preference for a piece of food that is closer to the creature, it moves towards the first piece of food it sees.

The targetting status has been added in order to prevent the creature from going in random directions and causing unwanted behaviour.

```python
class Creature:
  ...
  def sense_surroundings(self, predators, food):
    """
    Sense the surroundings for predators and food.

    Parameters:
    predators (list): The list of predators.
    food (list): The list of food.
    """
    self.energy -= self.sense/5



```

```
    # Always run from predators regardless of food or status
    for p in predators:
      distance = numpy.linalg.norm(p.position - self.position)

      # Preditor is self, so ignore
      if distance == 0:
        continue

      if distance <= self.sense and self.radius < 1.2 * p.radius and (
         not p.is_home()):
        # Run in opposite direction of predator
        run_direction = (self.position - p.position) / distance
        # Run distance is the remaining distance such that the predator
            is at the edge of the sense range
        run_distance = self.sense - distance

        self.destination = self.position + 1.2 * run_distance *
           run_direction
        self.targeting = True

        return True

    # If targeting food or predator, continue targeting
    if self.targeting is True and self.status == Status.EXPLORING:
      distance_target = numpy.linalg.norm(self.destination - self.
         position)

      # If target is out of range, stop targeting
      if distance_target > self.sense:
        self.targeting = False
        return False

      return True

    # Target food if hungry and in range
    for f in food:
      distance = numpy.linalg.norm(f.position - self.position)

      if distance <= self.sense and self.is_hungry():
        self.destination = f.position
        self.targeting = True
        return True

    return False
```