Evolutionary Game Theory

By:
Name: Aditya Mukherjee
Roll No.:24ME10038

● ***About Evolutionary Game Theory(EGT):***

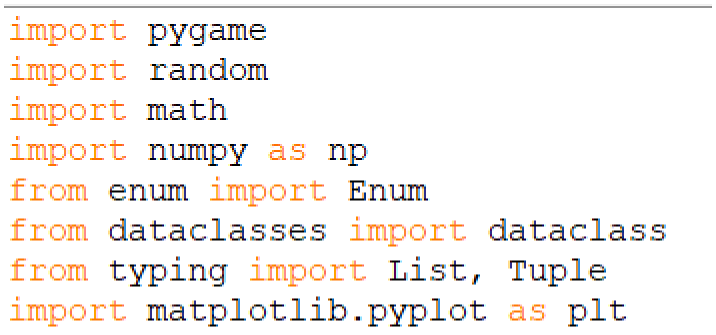
Evolutionary Game Theory (EGT) is a branch of game theory that studies how strategies evolve in a population over time.The key feature of evolutionary game theory is that many behaviors involve the interaction of multiple organisms in a population, and the success of any one of these organisms depends on how its behavior interacts with that of others.So the fitness of an individual organism can't be measured in isolation; rather it has to be evaluated in the context of the full population in which it lives.This opens the door to a natural game-theoretic analogy:
an organism's genetically-determined characteristics and behaviors are like its strategy in a game, its fitness is like its payoff , and this payoff depends on the strategies (characteristics) of the organisms with which it interacts.

● ***Problem Statement (Evolution of Swarm of Bees):***

Create an environment to simulate the swarm behavior of bees interacting with various pollen sources.Bees should move randomly and occasionally head toward a pollen source. Initially, all bees should follow random policies (i.e., make random decisions about which source to visit).

● ***Model (Pygame Code Description):***

1.Libraries Used:



```python
import pygame
import random
import math
import numpy as np
from enum import Enum
from dataclasses import dataclass
from typing import List, Tuple
import matplotlib.pyplot as plt
```

- Pygame: Graphics rendering,game loop

- Random: Stochastic processes
- Math: trigonometric calculations
- Numpy:Efficient array operations for preferences and matrix calc.
- Enum:Type-safe enumeration for different elements
- Dataclasses: Clean data structure definition
- Matplotlib.pyplot: Statistical graph plotting and data visualization
- Typing **:** Type hints for better code documentation

2.Screen Dimensions and Colour Coding (RGB Model):

```
# Constants
SCREEN_WIDTH = 1200
SCREEN_HEIGHT = 800
FPS = 60

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
YELLOW = (255, 255, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
PURPLE = (128, 0, 128)
ORANGE = (255, 165, 0)
```

Computers represent colors using the RGB (Red, Green, Blue) model. In this model, any color can be created by mixing different amounts of red, green, and blue light. The numbers in the brackets represent the intensity of the corresponding RGB colour while mixing(255 is maximum intensity).

3.Classes:

- Pollen Type (Quality of pollen source):

```
class PollenType(Enum):
    HIGH_QUALITY = 0
    MEDIUM_QUALITY = 1
    LOW_QUALITY = 2
```

High,low,medium are the members of enum , and 0,1,2 are their corresponding values.

- Pollen Source:

```python
@dataclass
class PollenSource:
    x: float
    y: float
    pollen_type: PollenType
    reward: float
    color: Tuple[int, int, int]
    radius: int = 15

    def get_reward(self):
        return self.reward
```

(x,y) is the is the position of the pollen source . pollen_type is the quality of the pollen source . reward is the payoff for each bee visiting the pollen source . radius is the radius of the circle that represents the pollen source.

- Bee:

```python
class Bee:
    def __init__(self, x, y, bee_id):
        self.x = x
        self.y = y
        self.bee_id = bee_id
        self.target = None
        self.speed = 2 + random.uniform(-0.5, 0.5)
        self.fitness = 0.0
        self.lifetime_reward = 0.0

        # Preference weights for different pollen sources (sum to 1)
        self.preferences = np.random.dirichlet([1, 1, 1])  # Random initial preferences

        # Movement
        self.angle = random.uniform(0, 2 * math.pi)
        self.last_decision_time = 0
        self.decision_interval = random.randint(60, 180)  # Frames between decisions

    def update_preferences_replicator_dynamics(self, payoff_matrix, population_preferences, dt=0.01):
        """Update preferences using replicator dynamics"""
        avg_payoffs = np.dot(payoff_matrix, population_preferences)
        avg_fitness = np.dot(self.preferences, avg_payoffs)

        for i in range(len(self.preferences)):
            self.preferences[i] += dt * self.preferences[i] * (avg_payoffs[i] - avg_fitness)

        # Normalize to ensure preferences sum to 1
        self.preferences = np.maximum(self.preferences, 0.001)  # Prevent negative values
        self.preferences /= np.sum(self.preferences)

    def mutate(self, mutation_rate=0.1):
        """Apply mutation to preferences"""
        if random.random() < mutation_rate:
            # Add small random noise to preferences
            noise = np.random.normal(0, 0.1, len(self.preferences))
            self.preferences += noise
            self.preferences = np.maximum(self.preferences, 0.001)
            self.preferences /= np.sum(self.preferences)

    def choose_pollen_source(self, pollen_sources):
        """Choose pollen source based on preferences and distance"""
        if not pollen_sources:
            return None

        best_source = None
        best_score = -float('inf')
```

```python
        for source in pollen_sources:
            # Distance factor (closer is better)
            distance = math.sqrt((self.x - source.x)**2 + (self.y - source.y)**2)
            distance_score = 1.0 / (1.0 + distance / 100.0)

            # Preference factor
            preference_score = self.preferences[source.pollen_type.value]

            # Combined score
            total_score = preference_score * distance_score

            if total_score > best_score:
                best_score = total_score
                best_source = source
```

```python
    def update(self, pollen_sources, frame_count):
        """Update bee behavior"""
        reached_target = False

        if self.target:
            reached_target = self.move_towards_target(self.target)
            if reached_target:
                self.target = None

        # Make decisions at intervals or after reaching target
        if (frame_count - self.last_decision_time > self.decision_interval) or reached_target:
            self.target = self.choose_pollen_source(pollen_sources)
            self.last_decision_time = frame_count
```

__init__ is the initialisation of the bees i.e the birth of bees . (x,y) give the starting position of the bee. bee_id is the unique identifier for tracking . target is the selected pollen source by the bees. Fitness is the reward collection in the current generation . fitness is responsible for the survival of the bees and its chances of reproduction in the next generation.preference is the probability vector of choosing the pollen source , and it sums to 1. decision_interval control how often a bee re-evaluates its strategy and chooses a new target. This prevents the bees from constantly switching targets every frame, creating more realistic behavior

<u>Replicator Dynamics</u>: This method implements the core of replicator dynamics. It adjusts a bee's preferences based on the success of different strategies in the entire population. In simple terms, if the population as a whole is getting high rewards from a certain type of pollen, this method will nudge the individual bee's preference slightly in that direction. This is how successful strategies spread through the population.

Update rule: `dx_i/dt = x_i * (f_i - f_avg)`

Normalization is done to maintain sum of probabilities to 1.

<u>Mutation</u>:introduces random variation. There's a small chance (mutation rate) that a bee's preferences will be slightly, randomly altered. This is vital for evolution, as it allows for new, potentially better strategies to emerge that might not have been present in the population otherwise.

Components:

- Stochastic application: Only occurs with probability mutation_rate
- Gaussian noise: Small random perturbations (mean=0, std=0.1)
- Bounds checking: Prevents negative preferences
- Renormalization: Maintains probability distribution

<u>Decision making for choosing pollen grains :</u>

Decision Algorithm:

1. Distance calculation: Euclidean distance to each source. Closer sources are preferred .
2. Distance score: Inverse relationship (closer = better) with scaling
3. Preference score: Individual preference for that pollen type
4. Combined utility: Multiplicative combination
5. Optimization: Greedy selection of highest utility

<u>Movement Mechanics:</u>
If the bee has a target, this method calculates the direction to the target and moves the bee a small step in that direction, scaled by its speed. When it reaches the target, it collects the reward, updates its fitness, and sets its target back to none. When it arrives within 5 pixels of the source it is considered to have reached the target . If the bee has no target (either because it just collected pollen or hasn't decided on one yet), it performs a random walk, slightly changing its angle of movement to simulate wandering or searching behavior . Boundary constraints are used to keep the bees within screen bounds.

<u>Main Update function:</u>
This is the main logic controller for the bee.It first checks if the bee should be moving towards a target. Then, it decides if it's time to make a new decision (by calling choose_pollen_source). Finally, if the bee has no target, it calls random_walk. This structure ensures the bee is always doing something logical.

- Bee Swarm Simulation:

1. __init__: Establishing the Stage

The __init__ function is the constructor, and it's used to initialize all aspects of the simulation when it initially starts.
Pygame Initialization: It initializes the screen, window title, and a clock to manage the frame rate.
Environment Setup: It sets up a static environment by filling self.pollen_sources list. It places strategically a fixed set of PollenSource objects of high, medium, and low quality with a precise location, reward value, and color.
Agent Population: It sets up the population of bees. It initializes a predefined number (num_bees) of Bee objects with each being positioned at a random start position.
Evolutionary Parameters: It sets up the basic rules of the evolution of the simulation:
generation: Begins the simulation at generation 0.
mutation_rate: Determines the chance that a new bee will be randomly mutated.
population_growth_rate: Determines the default rate at which the population will increase each generation.
max_population: A hard limit to keep the population from expanding without bounds, set at 5,000.
Game Theory Component: It establishes the payoff_matrix. This matrix is one of the central components of replicator dynamics, specifying the anticipated "payoff" for adopting a particular strategy (e.g., favor High quality) when interacting with the population mean strategy.
Data and UI: It creates empty lists (generation_stats, population_preferences_history) to collect data later for analysis purposes and loads fonts to use for displaying text on the screen.

2.Evolutionary Methods:

These techniques regulate how the population evolves and evolves from generation to generation.
moran_process_selection(): This method creates the Moran process, an old standard in population genetics. At each step of this process:
Selection for Reproduction: One "parent" bee is selected from the population. The likelihood of being selected is proportional to its fitness (the reward it earned

in the current generation). That is, successful bees have a higher chance of reproducing.
Random Removal: A random, "victim" bee is picked for removal from the population.
Replacement: A new offspring replaces the eliminated bee that has inherited the parent's preference (with an option of mutation).
This birth-death process maintains the population constant throughout this step while facilitating successful characteristics to spread.
update_population_replicator_dynamics(): This function simulates how strategies spread. It begins by computing the population's average preference. Next, it cycles through all the bees and gently pushes their personal preferences in the direction of how well their strategy is doing relative to the population average. This is the "social learning" or "imitation" part of the evolution.
add_new_bees(): This method manages population increase. It introduces a number of new bees according to the population_growth_rate. The parents of the new bees are chosen according to their lifetime_reward, i.e., bees that have always been successful are more probable to be involved in increasing the population. It also has a protection check in order to prevent the population from ever increasing beyond the max_population.
evolve_population(): This is the top function for changing generations. It coordinates the other methods in the right sequence:

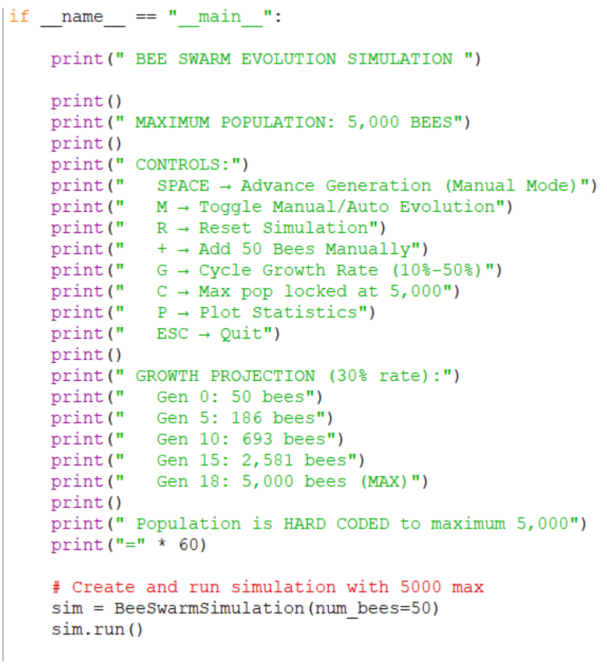
1. It executes the moran_process_selection several times to simulate selection and competition among the current population.
2. It executes update_population_replicator_dynamics to enable the spread of strategies.
3. It executes add_new_bees to manage population increase.
4. Lastly, it clears the fitness of all bees ready for the beginning of the next generation and increments the generation counter.

3.Simulation Loop and Graph Statistics:
These are the mechanisms that bring about the run-time execution and interactivity of the simulation.
run(): This is the top-level loop of the entire program. It runs continuously:
Processes User Input: It checks for keyboard press (such as 'P' to plot, 'M' to switch mode, 'SPACE' to step a generation) and if the user closes the window.

Updates Agents: It loops over each bee in the self.bees list and invokes their respective update() function, making them move, select targets, and garner rewards.
Renders the Scene: It clears the screen and then paints each pollen source and each bee at its new position. It has an optimization for performance that only draws a subset of bees if the population becomes very large, which keeps the simulation from slowing down.
Draws the UI: It invokes draw_ui() to show current information on the screen.
Manages the Frame Rate: self.clock.tick(FPS) keeps the simulation constant in speed.
draw_ui(): It manages all the text and information graphics shown on the screen, including the current generation, population count, a population growth progress bar, and a legend for pollen color.
collect_statistics() & plot_evolution_stats():
collect_statistics is invoked once every generation to capture a snapshot of the population's condition (average preferences, population size, etc.).
plot_evolution_stats is invoked only when the user hits 'P'. It plots all the data gathered so far and employs the matplotlib library to produce and display elaborate plots of the evolutionary trends.

4.Running the Simulation

:



```
if __name__ == "__main__":

    print(" BEE SWARM EVOLUTION SIMULATION ")

    print()
    print(" MAXIMUM POPULATION: 5,000 BEES")
    print()
    print(" CONTROLS:")
    print("   SPACE → Advance Generation (Manual Mode)")
    print("   M → Toggle Manual/Auto Evolution")
    print("   R → Reset Simulation")
    print("   + → Add 50 Bees Manually")
    print("   G → Cycle Growth Rate (10%-50%)")
    print("   C → Max pop locked at 5,000")
    print("   P → Plot Statistics")
    print("   ESC → Quit")
    print()
    print(" GROWTH PROJECTION (30% rate):")
    print("   Gen 0: 50 bees")
    print("   Gen 5: 186 bees")
    print("   Gen 10: 693 bees")
    print("   Gen 15: 2,581 bees")
    print("   Gen 18: 5,000 bees (MAX)")
    print()
    print(" Population is HARD CODED to maximum 5,000")
    print("=" * 60)

    # Create and run simulation with 5000 max
    sim = BeeSwarmSimulation(num_bees=50)
    sim.run()
```

1. if __name__ == "__main__": - The "Start Button"
This line is an essential and important Python idiom. Here's what it does in simple words: "Only execute the code within this block when this file is directly run by the user."
__name__: Special, built-in Python variable.
When you execute python your_script_name.py: Python automatically sets __name__ to the special value "__main__".
When you import your_script_name in another script: Python assigns __name__ as "your_script_name".
The above if statement is a safety check. It prevents the code to begin the simulation from executing automatically if you import this file into another project to leverage its classes (such as Bee or PollenSource). It ensures that this is the intended entry point.
2. The Body of the Block: User Interface and Simulation Launch
The code within the if block can be divided into three parts:
A. The Welcome Screen & Instructions (print() statements):
This whole part is focused on generating a user-friendly "welcome message" in the terminal.
Title: It prints out a sharp, stylized title for the simulation.
Key Parameters: It directly notifies the user of the most significant rule: the maximum population is hard-coded to 5,000.
Controls: This is an interactive manual. It describes all the keyboard controls (MACROS to use, like SPACE, M, R, etc.) and does exactly what each control does. This is crucial for making the interactive simulation accessible.
Growth Projection: This is a very useful feature that provides the user with a tangible prediction of what to expect. By displaying the population at various generations with a 30% growth factor, it informs the user about the simulation's dynamics prior to actual execution.
B. Creating the Simulation Instance:
python
sim = BeeSwarmSimulation(num_bees=50)
This is actually the line that initializes the simulation world. It:
Calls the __init__ method of the BeeSwarmSimulation class.
Passes num_bees=50 as an argument, instructing the simulation to begin with 50 bees.

The __init__ method is then executed, initializing the screen, inserting the pollen sources, instantiating the 50 initial bees, and setting up all the other simulation variables.
The newly initialized simulation object is assigned to the variable sim.

C. Running the Simulation:

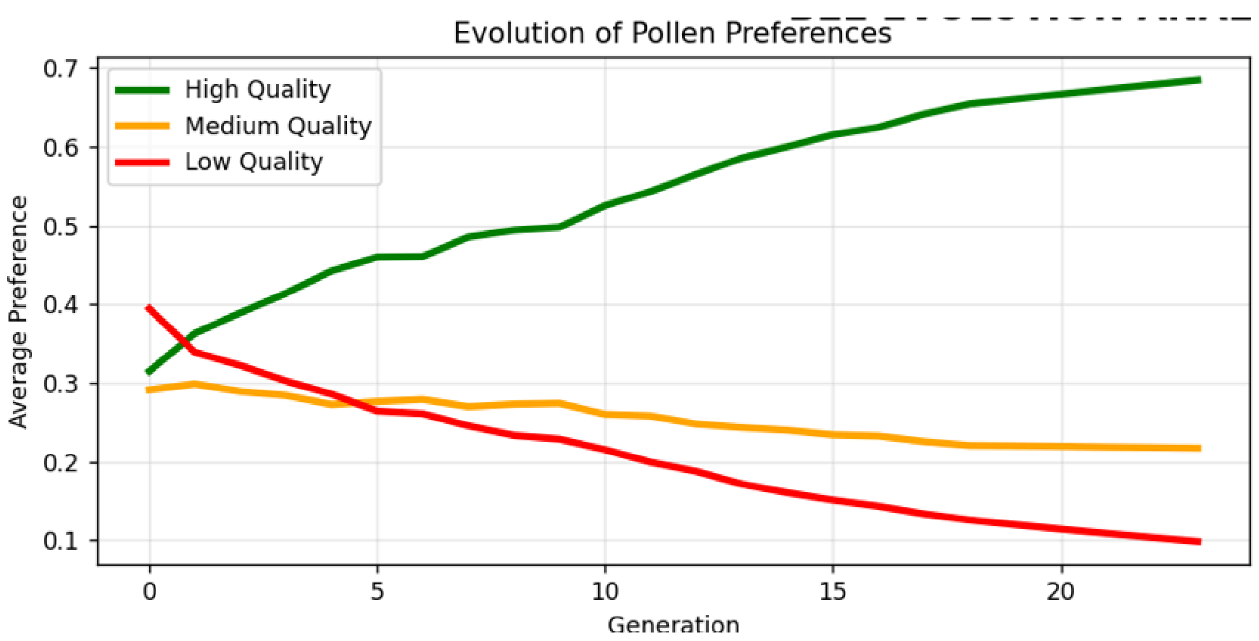
python
```
sim.run()
```
This last line is what begins the game. It invokes the run() method on the sim object you've created. This initiates the main while loop within the run() method, which will keep running:

- Listening for input from the user.
- Updating each of the bees' states.
- Drawing all the graphics on the screen.
- Managing the frame rate.

This loop will keep running, effectively turning over control of the program to the simulation, until the user exits by closing the window or hitting the 'ESC' key.
Essentially, this if __name__ == "__main__": block is the "start-up procedure" for your program. It prints out the mission briefing, builds the vehicle (BeeSwarmSimulation), and then runs it (run()).

● ***<u>Graphical and Statistical Analysis:</u>***

1. Evolution of Pollen Preferences :



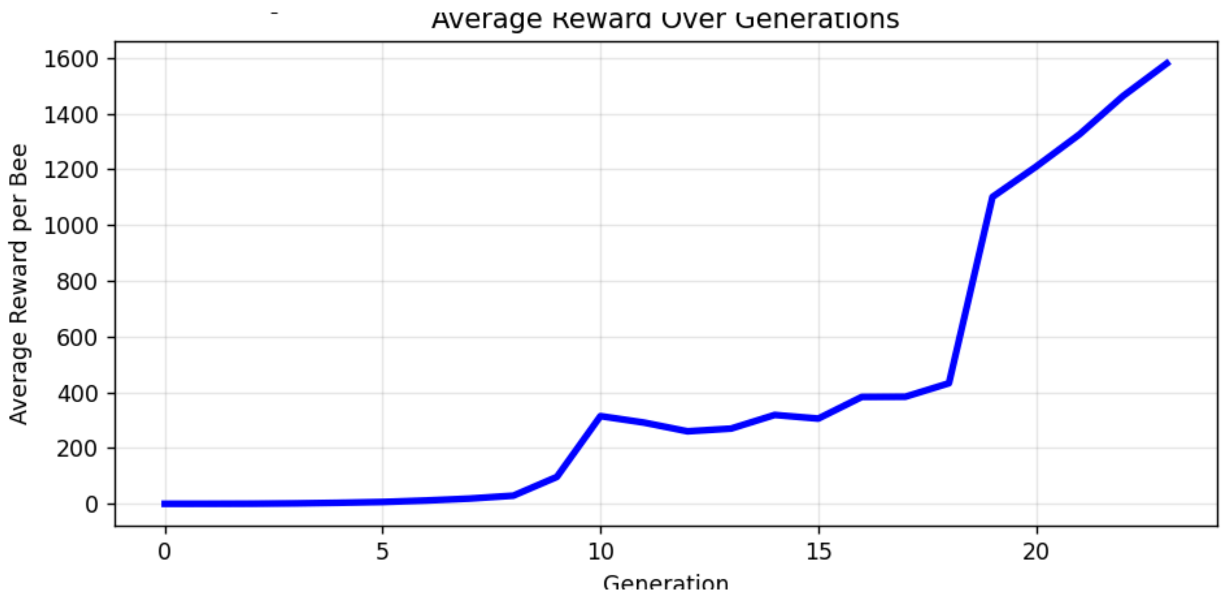
This is the main evidence that evolution is acting as designed.

Successful Strategy Emerges: The green line, which indicates the average preference for High Quality pollen, has a definite and consistent upward trend. It begins at approximately 0.33 (as would be expected from random initialization) and climbs to more than 0.7. This suggests that, generations after generation, the population has concluded that searching for high-reward pollen is the most successful strategy.

Unsuccessful Strategies are Eliminated: On the other hand, the red line (Low Quality) and the orange line (Medium Quality) both decline. The bees are actually evolving to steer clear of these poorer-quality pollen sources. The preference for low-quality pollen, in fact, declines by quite a bit, from a level higher than 0.3 to approximately 0.1.

<u>Interpretation:</u> The population is successfully coping with its environment. The selection pressures (reward-based fitness) are successfully eliminating unsuccessful strategies and favoring the successful ones. The simulation is accurately simulating natural selection.

2. Average Reward Over Generations



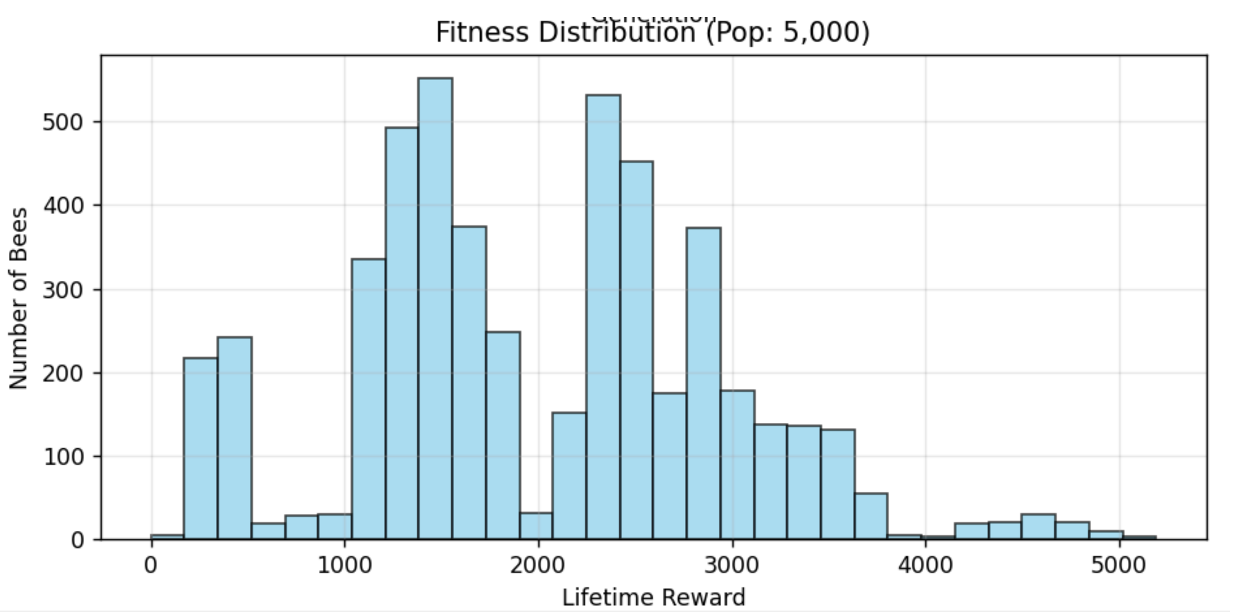
This graph shows how the bee society's economy works and tells the story of how things are getting more efficient.

At first, the average reward per bee is very low and stays the same for the first ten generations. This shows that the population wasn't very efficient at first while it was exploring.

- **A big jump in efficiency**: Starting with generation 10, the average reward per bee starts to go up a lot, which is when population boom starts. This means that the bees are not only getting better at what they like, but they are also getting much better at turning those likes into real rewards.
- **Sustained Improvement**: This path shows a more steady and lasting rise in performance, reaching a very high average reward by the end.

Interpretation: The bees evolve to prefer better food, they get more rewards, which makes their strategy even more successful. Over time, the whole population becomes more efficient.

3. Fitness Distribution



This histogram shows how fit (lifetime reward) all 5,000 bees were when the simulation was over.

- **Wide Distribution of Success**: This distribution is much wider than the last one, which had most of the bees in one peak. It looks like a bell curve (a normal distribution) that has been pushed to the right. This means that the people were more successful in a wider range of ways.
- **A "Middle Class" of Foragers**: The most successful foragers (the highest bars) make up a "middle class" that gets rewards for life of 1,000 to 2,500. These are the bees that always do a good job and are the foundation of the community.

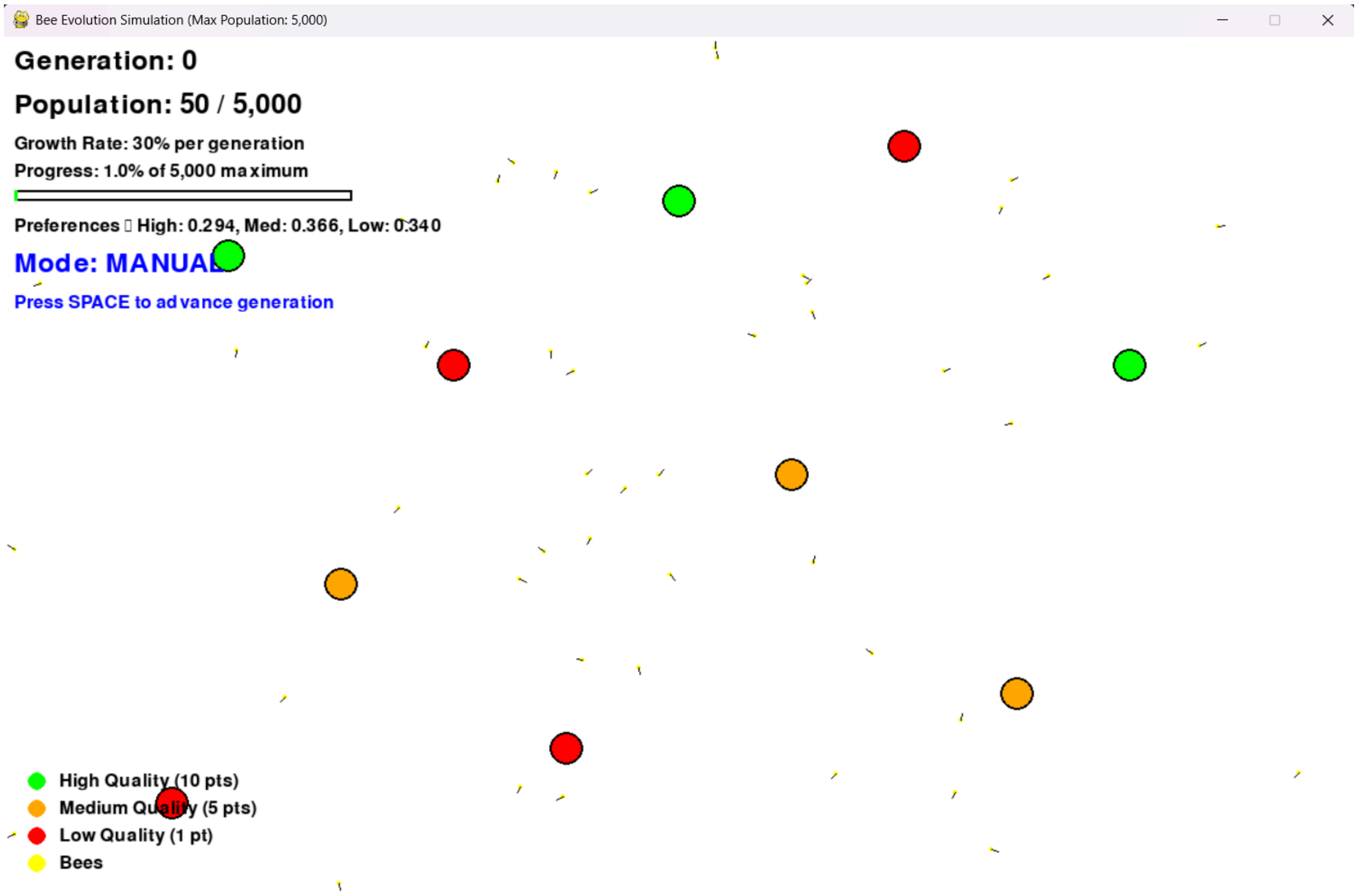
There are "tails" on both sides of the distribution for high achievers and those who are having trouble. Fewer "elite" bees have gotten very high rewards (3,000–5,000), and fewer less successful bees have gotten lower rewards.

This means that the population is healthy and diverse. Even though the most popular strategy is the one that works best (as shown in the preference graph), the results for each person are still very different. This could be due to chance, the location of the bees' birth, or minor variations in their strategic adaptations. It paints a more complete picture of a society than a single, large group of people who all do well.

Overall Conclusion

Together, these graphs reveal a clear narrative: The population of bees began with random tactics, but by means of evolutionary selection (reward-biased reproduction) and variation (mutation), it quickly learned and employed an optimal tactic of favoring high-quality pollen. This evolutionary triumph propagated explosive population growth until the carrying capacity of the environment was reached. At this point, the population leveled off into a highly specialized and efficient culture of proficient foragers.

Model Simulation Images:



Bee Evolution Simulation (Max Population: 5,000)

Generation: 0

Population: 50 / 5,000

Growth Rate: 30% per generation

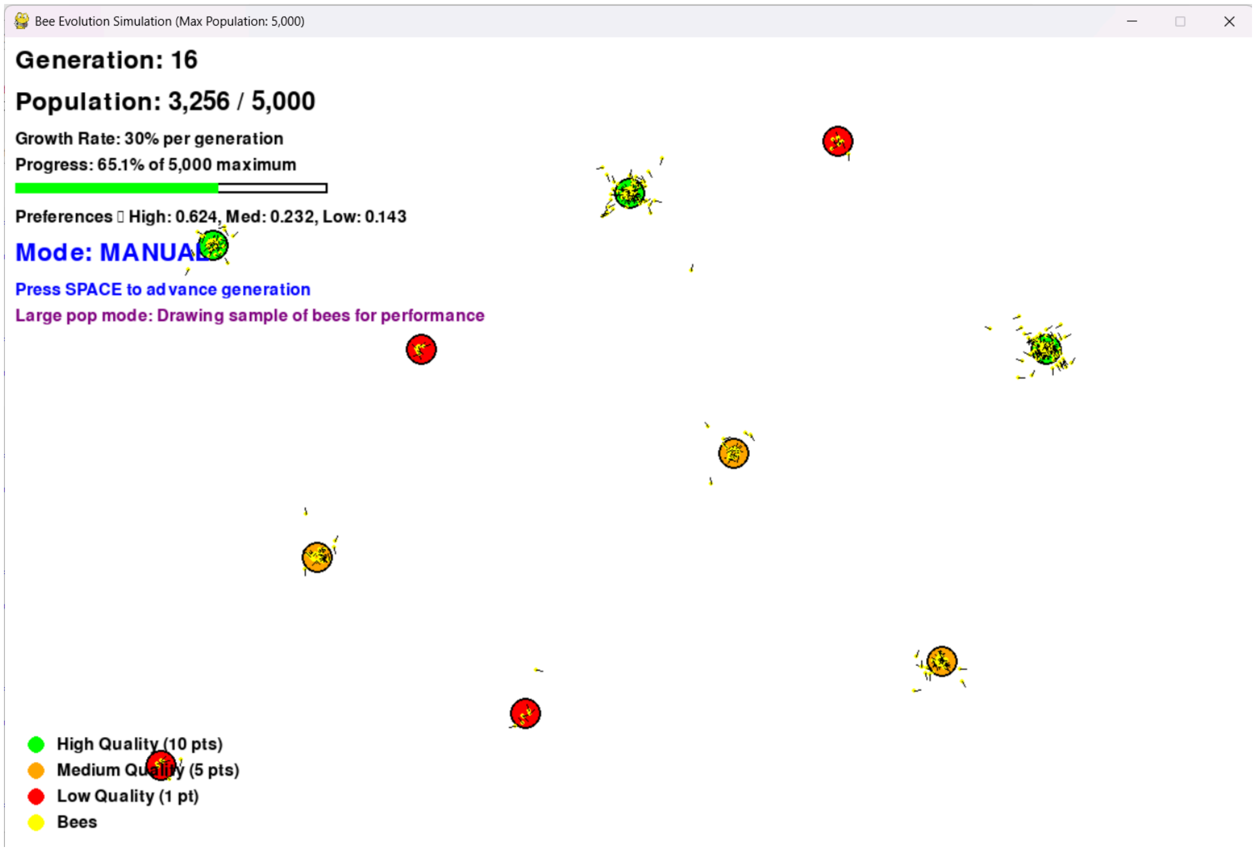
Progress: 1.0% of 5,000 maximum

Preferences High: 0.294, Med: 0.366, Low: 0.340

Mode: MANUAL

Press SPACE to advance generation

Initial stage(generation:0)



Bee Evolution Simulation (Max Population: 5,000)

Generation: 16

Population: 3,256 / 5,000

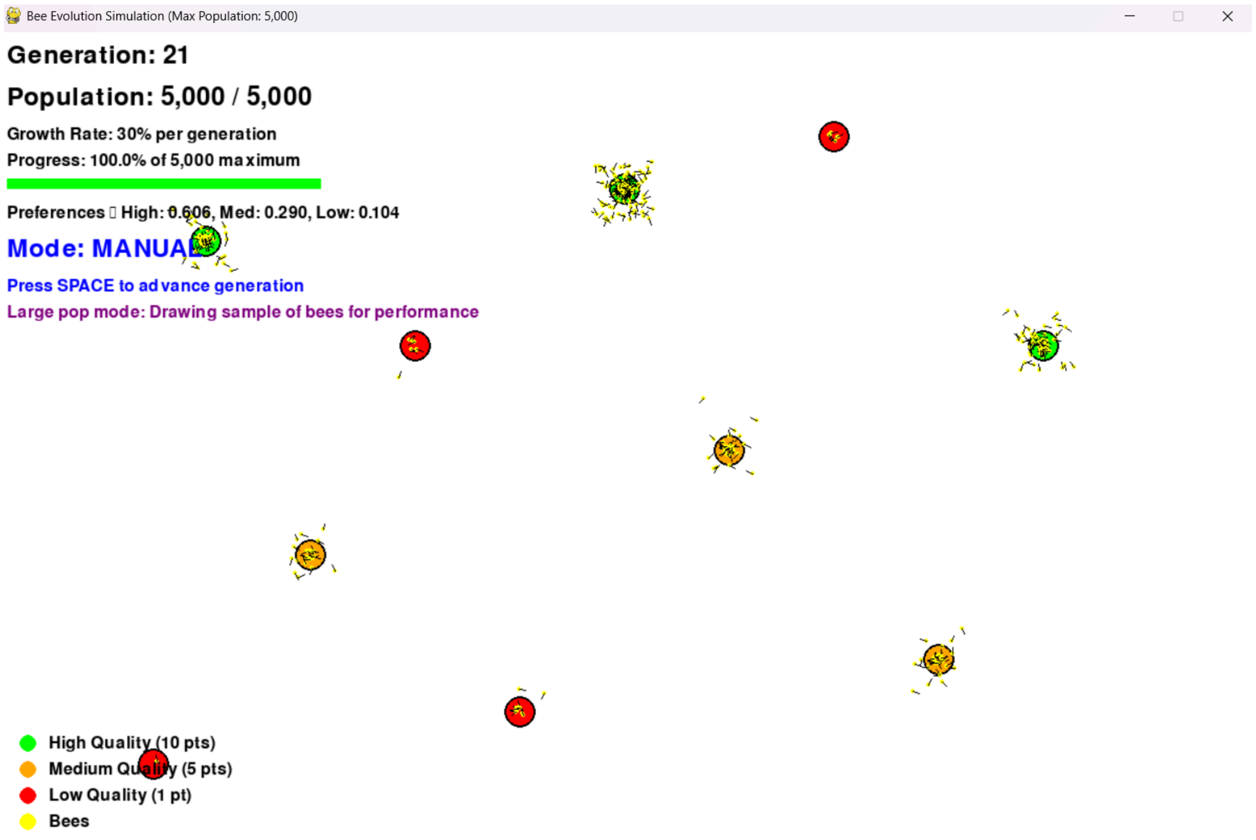
Growth Rate: 30% per generation
Progress: 65.1% of 5,000 maximum

Preferences High: 0.624, Med: 0.232, Low: 0.143

Mode: MANUAL

Press SPACE to advance generation
Large pop mode: Drawing sample of bees for performance

Simulation at generation:16



Bee Evolution Simulation (Max Population: 5,000)

Generation: 21

Population: 5,000 / 5,000

Growth Rate: 30% per generation
Progress: 100.0% of 5,000 maximum

Preferences ▯ High: 0.606, Med: 0.290, Low: 0.104

Mode: MANUAL

Press SPACE to advance generation
Large pop mode: Drawing sample of bees for performance

🟢	High Quality (10 pts)
🟠	Medium Quality (5 pts)
🔴	Low Quality (1 pt)
🟡	Bees

Simulation when max. Population is reached

Simulation Video Link: simulation video