Selim BEN JEDDI
3 IDL 1

Intelligent Computing Technique Project:

'Optimizing the Path of Ants Seeking Wheat Using Genetic Algorithms'

1- Introduction:

This work is carried out as part of the TP examination of the Intelligent Computing Technology subject for the academic year 2023-2024.

In this project, we addressed the problem of ants looking for wheat using an approach based on genetic algorithms. The objective was to develop an algorithm capable of optimizing the path each ant takes to reach wheat, by simulating natural evolution.

2- Objectives:

- Implementation of a simulation environment with ants, wheat, and obstacles.
- Creation of a genetic representation for ant behaviour in the form of genes.
- Use of genetic algorithms to evolve ant strategies to optimize the path to wheat.
- Analysis of the results to evaluate the effectiveness of the genetic algorithm in solving the problem.

3- Work environment:

In order to set up the project, we used the Python programming language for coding the solution, with the involvement of the Pygame library to create an interactive simulation environment including ants, wheat, and obstacles in the form of a small game of ants.

4- Solution Coding:

The code is organized into three main parts: the definition of the DNA class, the Ant class, and the Population class. These classes interact consistently to simulate the evolution of ants seeking wheat.

Classe DNA

This class represents the genes of ants. Genes are vectors that define the direction of the ant's movement at each stage of its life. The DNA class includes methods for the random creation of genes, crossing with another set of genes, and the possibility of mutation.

Genes are represented as two-dimensional vectors. Each component of the vector represents the force of the ant applied in a specific direction. These vectors are normalized and limited by maximum force.

The genes of the selected parents are crossed to create children, who inherit the characteristics of both parents.

A small mutation probability of 0.01 is introduced to ensure a wider exploration of the solution space

```
class DNA:
    def __init__(self, genes=None):
        if genes:
            self.genes = genes
        else:
            self.genes = [pygame.Vector2(random.uniform(-1, 1), random.uniform(-1, 1)) for _ in
range(life_span)]
            self.genes = [gene.normalize() * max_force for gene in self.genes]

    def crossover(self, partner):
        mid = random.randint(0, len(self.genes) - 1)
        new_genes = [self.genes[i] if i > mid else partner.genes[i] for i in range(len(self.genes))]
        return DNA(new_genes)

    def mutation(self):
        for i in range(len(self.genes)):
            if random.random() < 0.01:
                self.genes[i] = pygame.Vector2(random.uniform(-1, 1), random.uniform(-1, 1))
                self.genes[i] = self.genes[i].normalize() * max_force
```

Selim BEN JEDDI
3 IDL 1

Classe Ant

Each ant has an instance of the DNA class that determines its behavior. The Ant class has methods to apply force, update ant position, calculate its fitness, and display its image.

The fitness of each ant is evaluated according to its distance from wheat. Bonuses or penalties are applied depending on whether the ant hits wheat or collides with obstacles.

```python
class Ant:
    def __init__(self, dna=None):
        self.pos = pygame.Vector2(width/2, height)
        self.vel = pygame.Vector2()
        self.acc = pygame.Vector2()
        self.fitness = 0
        self.completed = False
        self.crashed = False
        self.reached_wheat = False  # Nouvelle variable pour suivre si la fourmi a atteint le blé

        if dna:
            self.dna = dna
        else:
            self.dna = DNA()

    def apply_force(self, force):
        self.acc += force

    def update(self):
        global count
        d = pygame.Vector2.distance_to(self.pos, sugar)
        if d < 30:
            self.completed = True
            self.reached_wheat = True  # Mettez à jour la variable si la fourmi atteint le blé


        if ox < self.pos.x < ox + ow and oy < self.pos.y < oy + oh:
            self.crashed = True

        if count < len(self.dna.genes):  # Ajout de la vérification de la plage valide
            self.apply_force(self.dna.genes[count])

        if not self.completed and not self.crashed:
            self.vel += self.acc
            self.pos += self.vel
            self.acc *= 0
    def show(self):
        # Dessiner l'image de la fourmi
        screen.blit(ant_image, (int(self.pos.x), int(self.pos.y)))

    def calc_fitness(self, ant_number):
        d = pygame.Vector2.distance_to(self.pos, sugar)
        print(f"Fitness de la fourmi numéro {ant_number} : {1 / (d + 1)}, Distance: {d}, Position:
{self.pos}")
        self.fitness = 1 / (d + 1)  # Ajoutez 1 pour éviter la division par zéro
        if self.completed:
            self.fitness *= 50
        if self.crashed:
            self.fitness /= 100
```

Population class

This class represents the total ant population. It contains a list of instances of the Ant class. The Population class manages the evolution of the population through the stages of selection, crossing, mutation, and fitness evaluation.

Ants are selected for breeding according to their fitness. The most successful ants have a greater chance of being selected.

```python
class Population:
    def __init__(self):
        self.ants = [Ant() for _ in range(20)]
        self.pop_size = 20
        self.pool = []
        self.fitness_generale_de_la_generation = 0  # Nouvelle variable

    def run(self):
        for ant in self.ants:
            ant.update()
            ant.show()

    def evaluate(self):
        max_fit = 0
        sum_fitness = 0  # Nouvelle variable pour stocker la somme des fitness individuelles
        for ant_number, ant in enumerate(self.ants):
            ant.calc_fitness(ant_number)
            sum_fitness += ant.fitness  # Ajoutez la fitness individuelle à la somme
            if ant.fitness > max_fit:
                max_fit = ant.fitness

        for ant in self.ants:
            ant.fitness /= max_fit

        self.pool = []
        for ant in self.ants:
            n = int(ant.fitness * 100)
            for _ in range(n):
                self.pool.append(ant)

        # Calculez la moyenne de fitness de la génération
        self.fitness_generale_de_la_generation = sum_fitness / len(self.ants)
        # Réinitialiser la valeur maximale de fitness après chaque génération
        max_fit = 0

    def selection(self):
        new_ants = []
        for ant in self.ants:
            parent_a = random.choice(self.pool).dna
            parent_b = random.choice(self.pool).dna
            child = parent_a.crossover(parent_b)
            child.mutation()
            new_ants.append(Ant(child))

        self.ants = new_ants

    def afficher_fitness_generale(self, screen, font):
        # Afficher la moyenne de fitness à l'écran
        print(f"Fitness générale de la génération: {self.fitness_generale_de_la_generation}")
```

And finally the main program which contains 5 main elements:

```python
# Initialisation de Pygame
screen = pygame.display.set_mode((width, height))
pygame.display.set_caption("Ant Game")

# Chargement de l'image d'arrière-plan
background_image = pygame.image.load("ant.jpg").convert()
# Chargement de l'image de blé
wheat_image = pygame.image.load("ble.png").convert_alpha()
# Redimensionnez l'image au besoin (par exemple, 10x10 pixels)
wheat_image = pygame.transform.scale(wheat_image, (80, 80))
# Chargement de l'image de la fourmi
ant_image = pygame.image.load("fourmi.png").convert_alpha()
# Redimensionnez l'image au besoin (par exemple, 20x20 pixels)
ant_image = pygame.transform.scale(ant_image, (30, 30))
# Chargement de l'image de la barrière
barriere_image = pygame.image.load("barriere.png").convert_alpha()
# Redimensionnez l'image au besoin
barriere_image = pygame.transform.scale(barriere_image, (200, 10))  # Ajustez la taille selon vos
besoins

# Initialisation de la population
population = Population()

# Boucle principale
running = True
font = pygame.font.Font(None, 36)  # Choisissez une police et une taille appropriées

# Ajoutez une liste pour stocker les fitness générales de chaque génération
all_fitness_scores = []
# Boucle principale
while running and generation_count < 50:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # Dessiner l'arrière-plan
    screen.blit(background_image, (0, 0))
    population.run()

    # Affichage du nombre de générations
    text = font.render(f"Generation count {generation_count}", True, (255, 255, 255))
    screen.blit(text, (10, 10))  # Choisissez la position appropriée

    # Affichage des rectangles
    pygame.draw.rect(screen, (255, 255, 255), (ox, oy, ow, oh))
    # Dessiner l'image de la barrière à la place du rectangle blanc
    screen.blit(barriere_image, (ox, oy))
    wheat_x = int(sugar.x) - 30  # Ajustez le décalage selon vos besoins
    wheat_y = int(sugar.y)  # Ajustez la position selon vos besoins
    screen.blit(wheat_image, (wheat_x, wheat_y))
    pygame.display.flip()

    # Vérifier si toutes les fourmis ont atteint le blé
    if all(ant.reached_wheat for ant in population.ants):
        # Afficher le message de félicitations
        text = font.render("Félicitations, toutes les fourmis ont atteint le blé!", True, (255, 255,
255))
        screen.blit(text, (width // 4, height // 2))
        pygame.display.flip()
        pygame.time.delay(3000)  # Pause pendant 3 secondes
        running = False  # Arrêter la boucle

    clock = pygame.time.Clock()
    fps = 200  # Réglez le nombre de mises à jour par seconde que vous souhaitez
    clock.tick(fps)

     # Gestion du compteur
    if count == life_span:
        population.evaluate()
        population.selection()
        count = 0
        generation_count += 1
        # Affichage de la fitness générale de la génération
        population.afficher_fitness_generale(screen, font)

        # Ajoutez la fitness générale actuelle à la liste
        all_fitness_scores.append(population.fitness_generale_de_la_generation)

    count += 1

# Afficher le meilleur résultat de la fitness générale de toutes les générations
best_fitness = max(all_fitness_scores)
print(f"Meilleur résultat de la fitness générale : {best_fitness}")
# Fermeture de Pygame
pygame.quit()
```

- **Visual Interface** : Using Pygame to create a graphical interface makes the evolution process of ants and their quest towards wheat intuitive and visually attractive. Images of ants, wheat and the fence add an interactive aspect to the simulation.

- **Main Loop** : The program's main loop manages Pygame events, refreshes the display at each iteration, and ensures a smooth communication between the user and the simulation by clearly indicating to each generation of ant its performance in relation to the reaching of the wheat as well as the final position of each ant of the generation.

  - **Information Display: The inclusion of information such as the number of generations, general fitness, and at the end of the treatment displays the best fitness function compared to all the generations that have gone through the program to be able to identify the best positions that ants were able to reach with this genetic algorithm.**

  Note: The program stops in 2 cases: Either the maximum number of generation reached (The most likely case) and in this case the program shows us at the end the result of the best fitness function of all generations, or all ants reached wheat (Very rare case since it takes a long time to iterate thousands of generations in this case).

  - **Evaluation and Selection**: The evaluate method in the Population class is called at the end of each generation, calculating the fitness of each ant and preparing the population for the next generation. Selection and crossing are key steps in the improvement of successive generations.
  - **Best Result Display: The best general fitness of all generations is displayed at the end of the simulation. This gives a clear indication of the overall performance of the genetic algorithm in solving the specific problem.**

**Display &amp
View**

Show method in the Ant class: The image of each ant is displayed at the current position of the ant in the environment.

Pygame Display: The simulation environment, including obstacles, wheat, and ants, is displayed using Pygame for interactive visualization.

Parameters and Adjustments

Parameter Adjustment: Parameters such as population size, ant lifespan, mutation probability, etc., can be adjusted to influence the overall performance of the genetic algorithm.

5- Results and tests:

This part will be covered in a short video through this link: https://drive.google.com/file/d/1jxOLYEtLfJOvhacQYxc5VGXF82oAb6f/view?usp=sharingqui clearly shows the different stages of testing a program and the succession of generations.

We observed a significant improvement in ant performance over generations. The general fitness of the population increased, indicating a successful adaptation to the constraints of the problem.

6- Conclusion:

In conclusion, this project demonstrates the effectiveness of genetic algorithms in the optimization of complex behaviors, such as the ant navigation path. The obtained results open the way to other potential applications of this approach in similar fields.