Snake Game

General logic and functions for running the snake game

```
import random
from collections import namedtuple
from enum import Enum
import pygame
```

```
pygame 2.6.1 (SDL 2.28.4, Python 3.11.12)
Hello from the pygame community. https://www.pygame.org/contribute.html
```

```
class Direction(Enum):
    RIGHT = 1
    LEFT = 2
    UP = 3
    DOWN = 4
```

```
pygame.init()
font = pygame.font.SysFont("arial", 25)
Point = namedtuple("Point", "x, y")
```

```
# rgb colors
WHITE = (255, 255, 255)
RED = (200, 0, 0)
BLUE1 = (0, 0, 255)
BLUE2 = (0, 100, 255)
BLACK = (0, 0, 0)

BLOCK_SIZE = 60
```

Class for the snake game

The snake class defines the behaviour of the game. The states, moves and changes in the game along with the game conditions.

The snake class also calculates and stores important information like distance of snake to food, current state and accumulated reward for the genetic algorithm.

```
class SnakeGame:
    def __init__(self, w=8*BLOCK_SIZE, h=8*BLOCK_SIZE, speed=999):
        self.w = w
        self.h = h
        self.speed = speed
        self.last_positions = []
        self.BLOCK_SIZE = BLOCK_SIZE

        # init display
        self.display = pygame.display.set_mode((self.w, self.h))
        pygame.display.set_caption("Snake")
        self.clock = pygame.time.Clock()

        # init game state
        self.direction = Direction.RIGHT

        self.head = Point(
            (self.w // BLOCK_SIZE // 2) * BLOCK_SIZE,
            (self.h // BLOCK_SIZE // 2) * BLOCK_SIZE
        )
        self.snake = [
            self.head,
            Point(self.head.x - BLOCK_SIZE, self.head.y),
            Point(self.head.x - (2 * BLOCK_SIZE), self.head.y),
        ]

        self.score = 0
        self.food = None
        self._place_food()
```

```
# Places food in random positions but avoids placing food in the same position twice (Used for training)
# def _place_food(self):
#     filas = self.h // BLOCK_SIZE
#     columnas = self.w // BLOCK_SIZE
#     snake_length = len(self.snake)

#     # Inicializar estructuras si no existen o si cambió el tamaño de la serpiente
#     if not hasattr(self, 'food_history') or self.food_history.get('length') != snake_length:
#         self.food_history = {
#             'length': snake_length,
#             'used_positions': set()
#         }

#     # Generar todas las posiciones posibles en la cuadrícula
#     todas_las_posiciones = {
#         Point(x * BLOCK_SIZE, y * BLOCK_SIZE)
#         for x in range(columnas)
#         for y in range(filas)
#     }

#     # Eliminar posiciones ocupadas por la serpiente o ya usadas
#     candidatas = list(
#         todas_las_posiciones - self.food_history['used_positions'] - set(self.snake)
#     )

#     if not candidatas:
#         # Reiniciar ciclo si ya se usaron todas (excepto las ocupadas por la serpiente)
#         self.food_history['used_positions'] = set()
#         candidatas = list(todas_las_posiciones - set(self.snake))

#     if not candidatas:
#         # No hay espacio libre (juego colapsado)
#         self.food = self.snake[0]
#         return

#     # Elegir nueva posición y registrar
#     self.food = random.choice(candidatas)
#     self.food_history['used_positions'].add(self.food)

# Places food in random positions
def _place_food(self):
    x = random.randint(0, (self.w - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
    y = random.randint(0, (self.h - BLOCK_SIZE) // BLOCK_SIZE) * BLOCK_SIZE
    self.food = Point(x, y)
    if self.food in self.snake:
        self._place_food()

# Predefined positions for food
# def _place_food(self):
#     food_positions = [
#         (3 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 2 * BLOCK_SIZE)
```

```
#         (7 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 4 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 1 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (0 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (1 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 3 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 2 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (5 * BLOCK_SIZE, 5 * BLOCK_SIZE),
#         (3 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (4 * BLOCK_SIZE, 7 * BLOCK_SIZE),
#         (7 * BLOCK_SIZE, 0 * BLOCK_SIZE),
#         (2 * BLOCK_SIZE, 6 * BLOCK_SIZE),
#         (6 * BLOCK_SIZE, 2 * BLOCK_SIZE)
#     ]

#     if not hasattr(self, 'food_position_index'):
#         self.food_position_index = 0

#     for _ in range(len(food_positions)):
#         x, y = food_positions[self.food_position_index]
#         self.food_position_index = (self.food_position_index + 1) % len(food_positions)
#         new_food = Point(x, y)
#         if new_food not in self.snake:
#             self.food = new_food
#             return

#     # fallback: si todas están ocupadas, ponerla donde sea (por ahora, centro)
#     self.food = Point(self.w // 2, self.h // 2)


#It represents all the logic needed to make a single move in the game
def play_step(self, action=None):
    # 1. collect user input
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            quit()

    # 2. move
    if action and self.is_valid_direction_change(action):
        self.direction = action

    # Calculate distance from snake head to food
    prevDistance = abs(self.head.x - self.food.x) + abs(self.head.y - self.food.y)

    self._move(self.direction)
    self.snake.insert(0, self.head)

    # Calculates reward to be used as the fitness of a individual
    # 3. reward
```

```
        reward = 0

        # Penalize for losing or for taking too many steps without eating food, additional penalty based on score to discourage early collisi
        # Reward for eating food, eating food quickly and moving towards the food
        if self._is_collision():
            reward = -25
            reward -= 10 / (self.score + 1)
        elif self.head == self.food:
            reward = 25
            # Bonus for efficient path to food

            if hasattr(self, 'steps_since_last_food'):
                # More reward for finding food quickly
                reward += max(15 - 0.1 * self.steps_since_last_food, 5)
                self.steps_since_last_food = 0
            else:
                self.steps_since_last_food = 0

        else:
            # Small penalty for each step to encourage efficiency
            reward = -0.05
            # Track steps since last food
            if hasattr(self, 'steps_since_last_food'):
                self.steps_since_last_food += 1
            else:
                self.steps_since_last_food = 1

            # Penalty for taking too many steps without finding food
            if self.steps_since_last_food > 100:
                reward -= 5

        # Distance-based rewards - improved
        newDistance = abs(self.head.x - self.food.x) + abs(self.head.y - self.food.y)
        if newDistance < prevDistance:
            reward += 0.3
        else:
            reward -= 0.2

        # Add exploration bonus for visiting new positions
        current_pos = (self.head.x, self.head.y)
        if not hasattr(self, 'positions_visited'):
            self.positions_visited = set()

        if current_pos not in self.positions_visited:
            self.positions_visited.add(current_pos)
            reward += 0.2

        # Survival bonus - small reward just for staying alive
        reward += 0.01 * self.score  # Scales with current score

        newDistance = abs(self.head.x - self.food.x) + abs(self.head.y - self.food.y)
        if newDistance < prevDistance:
            reward += 0.1

        # 5. check if game over
        game_over = False
        if self._is_collision():
            game_over = True
            return game_over, reward, self.score

        # 6. place new food or just move
        if self.head == self.food:
            self.score += 1
            self._place_food()
        else:
            self.snake.pop()

        # 7. update ui and clock
        self._update_ui()
        self.clock.tick(self.speed)

        # 8. return game over and score
        return game_over, reward, self.score

    # Detect collision with wall or with itself
    def _is_collision(self):
        if (
```

```
            self.head.x > self.w - BLOCK_SIZE
            or self.head.x < 0
            or self.head.y > self.h - BLOCK_SIZE
            or self.head.y < 0
        ):
            return True
        if self.head in self.snake[1:]:
            return True
        return False

    def _update_ui(self):
        self.display.fill(BLACK)

        for pt in self.snake:
            pygame.draw.rect(self.display, BLUE1, pygame.Rect(pt.x, pt.y, BLOCK_SIZE, BLOCK_SIZE))
            pygame.draw.rect(self.display, BLUE2, pygame.Rect(pt.x + 4, pt.y + 4, 12, 12))

        pygame.draw.rect(
            self.display, RED, pygame.Rect(self.food.x, self.food.y, BLOCK_SIZE, BLOCK_SIZE)
        )

        text = font.render("Score: " + str(self.score), True, WHITE)
        self.display.blit(text, [0, 0])
        pygame.display.flip()

    # Snake movement in the game
    def _move(self, direction):
        x = self.head.x
        y = self.head.y
        if direction == Direction.RIGHT:
            x += BLOCK_SIZE
        elif direction == Direction.LEFT:
            x -= BLOCK_SIZE
        elif direction == Direction.DOWN:
            y += BLOCK_SIZE
        elif direction == Direction.UP:
            y -= BLOCK_SIZE
        self.head = Point(x, y)

    # Avoid player from take one direction and inmedietly taking the opposite direction
    def is_valid_direction_change(self, new_direction):
      opposite = {
          Direction.RIGHT: Direction.LEFT,
          Direction.LEFT: Direction.RIGHT,
          Direction.UP: Direction.DOWN,
          Direction.DOWN: Direction.UP,
      }
      return new_direction != opposite[self.direction]

    def print_board(self):
      rows = self.h // BLOCK_SIZE
      cols = self.w // BLOCK_SIZE
      board = [["." for _ in range(cols)] for _ in range(rows)]

      def within_bounds(x, y):
          return 0 <= x < cols and 0 <= y < rows

      # Snake Body Position
      for pt in self.snake:
          x, y = int(pt.x // BLOCK_SIZE), int(pt.y // BLOCK_SIZE)
          if within_bounds(x, y):
              board[y][x] = "O"

      # Head Position
      head_x, head_y = int(self.head.x // BLOCK_SIZE), int(self.head.y // BLOCK_SIZE)
      if within_bounds(head_x, head_y):
          board[head_y][head_x] = "H"

      # Food Position
      food_x, food_y = int(self.food.x // BLOCK_SIZE), int(self.food.y // BLOCK_SIZE)
      if within_bounds(food_x, food_y):
          board[food_y][food_x] = "F"

      # Print table
      print("\n".join(" ".join(row) for row in board))
      print("-" * 40)

    # Get the current state of the game
```

```
    def getState(self):
        return(
            self.direction == Direction.RIGHT,
            self.direction == Direction.LEFT,
            self.direction == Direction.UP,
            self.direction == Direction.DOWN,
            # Calculates distance from food
            round(self.head.x - self.food.x, 1),
            round(self.head.y - self.food.y, 1),

            # Snake size
            len(self.snake)
        )
```

Genetic Algorithm

Genetic algoritm Generates a table that represents an individual. The table holds a move for every given state as the algorithm finds new states.

The expected result is to find the best table (Individual) with the most optimal moves for each possible state.

```
import random
import copy
import pickle
import time
import numpy as np
import os
from collections import namedtuple

Point = namedtuple("Point", "x, y")
```

Decision Table Class

This class generates and manages the tables of each individual, in addition it defines the logic for the genetic algorithm as it enables tables to crossover, mutate and calculte its fitness.

It contains the logic that enable the genetic algorithm to function

```
class DecisionTable:
    def __init__(self, states=None):
        self.states = states if states is not None else {}
        self.fitness = 0
        self.steps = 0
        self.score = 0
        self.games_played = 0
        self.avg_fitness = 0

    def getState(self, game:SnakeGame):
        # Enhanced state representation
        head = game.head
        food = game.food

        # Direction one-hot encoding
        dir_right = game.direction == Direction.RIGHT
        dir_left = game.direction == Direction.LEFT
        dir_up = game.direction == Direction.UP
        dir_down = game.direction == Direction.DOWN

        # Danger detection (check if moving in each direction would cause collision)
        danger_straight = self._is_direction_dangerous(game, game.direction)
        danger_right = self._is_direction_dangerous(game, self._get_right_direction(game.direction))
        danger_left = self._is_direction_dangerous(game, self._get_left_direction(game.direction))

        # Food location relative to head
        food_left = food.x < head.x
        food_right = food.x > head.x
        food_up = food.y < head.y
        food_down = food.y > head.y

        # Create a tuple of all state components
```

```
        state = (
            danger_straight,
            danger_right,
            danger_left,

            dir_right,
            dir_left,
            dir_up,
            dir_down,

            food_left,
            food_right,
            food_up,
            food_down,

            # Snake length (normalized)
            # Cap at 20 for normalization
            min(1.0, len(game.snake) / 20)
        )

        return state

    def _is_direction_dangerous(self, game, direction):
        """Check if moving in a direction would cause collision"""
        # Create a new Point instead of trying to modify the existing one
        x, y = game.head.x, game.head.y

        if direction == Direction.RIGHT:
            x += game.BLOCK_SIZE
        elif direction == Direction.LEFT:
            x -= game.BLOCK_SIZE
        elif direction == Direction.DOWN:
            y += game.BLOCK_SIZE
        elif direction == Direction.UP:
            y -= game.BLOCK_SIZE

        test_head = Point(x, y)

        # Check boundary collision
        if (test_head.x >= game.w or test_head.x < 0 or
            test_head.y >= game.h or test_head.y < 0):
            return True

        # Check self collision
        if test_head in game.snake[1:]:
            return True

        return False

    def _get_right_direction(self, direction):
        """Get the direction to the right of current direction"""
        if direction == Direction.RIGHT:
            return Direction.DOWN
        elif direction == Direction.DOWN:
            return Direction.LEFT
        elif direction == Direction.LEFT:
            return Direction.UP
        else:  # UP
            return Direction.RIGHT

    def _get_left_direction(self, direction):
        """Get the direction to the left of current direction"""
        if direction == Direction.RIGHT:
            return Direction.UP
        elif direction == Direction.UP:
            return Direction.LEFT
        elif direction == Direction.LEFT:
            return Direction.DOWN
        else:  # DOWN
            return Direction.RIGHT

    def act(self, state):
        # If state is unknown, return a completely random action
        if state not in self.states:
            # Just choose a random direction
            self.states[state] = random.choice(list(Direction))
            return self.states[state]
```

```
        # If we have this state in our table, use the stored action
        return self.states[state]

    def addState(self, state, action):
        self.states[state] = action

    def crossover(self, other):
        child = DecisionTable()
        allStates = set(self.states.keys()) | set(other.states.keys())

        # Use weighted crossover based on fitness
        total_fitness = max(1, self.fitness + other.fitness)
        self_weight = 0.5
        if total_fitness > 0:
            self_weight = self.fitness / total_fitness

        for state in allStates:
            if state in self.states and state in other.states:
                # Weighted choice based on fitness
                if random.random() < self_weight:
                    child.addState(state, self.states[state])
                else:
                    child.addState(state, other.states[state])
            elif state in self.states:
                child.addState(state, self.states[state])
            else:
                child.addState(state, other.states[state])
        return child

    def mutate(self):
        # Much lower base mutation rate to preserve good behaviors
        base_rate = 0.04  # Reduced significantly

        # Very conservative adaptive rate
        adaptive_rate = max(0.01, base_rate / (1 + 0.2 * len(self.states)))

        # Apply mutation with more care - focus on preserving good states
        for state in self.states:
            # Extract state components to make smarter mutation decisions
            if len(state) >= 11:  # Make sure state has enough components
                food_left, food_right, food_up, food_down = state[7], state[8], state[9], state[10]
                danger_straight, danger_right, danger_left = state[0], state[1], state[2]
                dir_right, dir_left, dir_up, dir_down = state[3], state[4], state[5], state[6]

                # Current action
                current_action = self.states[state]

                # Lower mutation probability for states that are moving toward food
                if ((food_right and current_action == Direction.RIGHT) or
                    (food_left and current_action == Direction.LEFT) or
                    (food_up and current_action == Direction.UP) or
                    (food_down and current_action == Direction.DOWN)):
                    # Much lower mutation rate for good food-seeking behavior
                    if random.random() < adaptive_rate * 0.3:  # 70% less likely to mutate
                        possible_actions = [d for d in Direction if d != current_action]
                        if possible_actions:
                            self.states[state] = random.choice(possible_actions)

                # Lower mutation for states that avoid danger
                elif danger_straight and current_action != self._get_current_direction(state):
                    # Lower mutation rate for good danger-avoiding behavior
                    if random.random() < adaptive_rate * 0.5:  # 50% less likely to mutate
                        possible_actions = [d for d in Direction if d != current_action]
                        if possible_actions:
                            self.states[state] = random.choice(possible_actions)

                # Normal mutation for other states
                elif random.random() < adaptive_rate:
                    possible_actions = [d for d in Direction if d != current_action]
                    if possible_actions:  # Make sure we have alternatives
                        self.states[state] = random.choice(possible_actions)
            else:
                # Fallback for states with unexpected format
                if random.random() < adaptive_rate:
                    current_action = self.states[state]
                    possible_actions = [d for d in Direction if d != current_action]
```

```
                    if possible_actions:
                        self.states[state] = random.choice(possible_actions)

    def _get_current_direction(self, state):
        """Helper method to determine current direction from state"""
        if len(state) >= 7:
            dir_right, dir_left, dir_up, dir_down = state[3], state[4], state[5], state[6]
            if dir_right:
                return Direction.RIGHT
            elif dir_left:
                return Direction.LEFT
            elif dir_up:
                return Direction.UP
            elif dir_down:
                return Direction.DOWN
        return None  # Default if can't determine

    def calculateFitness(self, num_games=3):
        """Calculate fitness by playing multiple games and averaging the results"""
        total_fitness = 0
        total_score = 0
        total_steps = 0

        for _ in range(num_games):
            game = SnakeGame()
            gameOver = False
            stepsNoFood, steps, score, fitness = 0, 0, 0, 0
            last_score_time = 0

            while not gameOver and stepsNoFood < MAX_STEPS_NO_FOOD:
                currentState = self.getState(game)
                action = self.act(currentState)

                gameOver, reward, score = game.play_step(action)

                steps += 1
                stepsNoFood += 1  # Increment steps without food
                fitness += reward

                # Reset stepsNoFood counter when food is eaten
                if score > game.score:
                    stepsNoFood = 0
                    last_score_time = steps
                    game.score = score  # Update the score reference

                # Add stagnation penalty
                if score > 0 and steps - last_score_time > 100:
                    fitness -= 10
                    break

            # Apply starvation penalty if snake didn't find food in time
            if stepsNoFood >= MAX_STEPS_NO_FOOD:
                fitness -= 15  # Significant penalty for starving
                gameOver = True  # End the game due to starvation

            total_fitness += fitness
            total_score += score
            total_steps += steps

        # Average the results
        self.games_played = num_games
        self.fitness = total_fitness / num_games
        self.score = total_score / num_games
        self.steps = total_steps / num_games
        self.avg_fitness = self.fitness

        return self.fitness
```

Creates the next generation by mutating and crossing the best inidividuals (Tables) of the previous generation.

```
def nextGen(population, gen):
    # Sort by fitness
    population.sort(key=lambda x: x.fitness, reverse=True)

    # Sort by score as a secondary criterion
```

```
    population.sort(key=lambda x: x.score, reverse=True)

    # Calculate statistics
    avg_fitness = sum([ind.fitness for ind in population]) / len(population)
    best_fitness = population[0].fitness

    # Track best score for reporting
    best_score = max([ind.score for ind in population])
    best_score_ind = max(population, key=lambda x: x.score)

    # Much stronger elitism - keep more of the best individuals unchanged
    elitism_count = max(5, int(len(population) * 0.25))  # Increased to 25%
    elites = [copy.deepcopy(ind) for ind in population[:elitism_count]]

    # Always keep the best scoring individual
    if best_score_ind not in population[:elitism_count]:
        elites.append(copy.deepcopy(best_score_ind))

    # Tournament selection with higher pressure
    tournament_size = 4  # Increased from 3
    selected = []

    # Fill the rest of the population with tournament selection
    while len(selected) < len(population) - len(elites):
        tournament = random.sample(population, tournament_size)
        # Select the best from tournament
        winner = max(tournament, key=lambda x: x.fitness)
        selected.append(copy.deepcopy(winner))

    # Create new population with elites and random individuals
    newPopulation = elites

    # Add offspring from crossover and mutation
    while len(newPopulation) < len(population):
        # Select parents using tournament selection
        parent1 = random.choice(selected)
        parent2 = random.choice(selected)

        # Ensure parents are different
        while parent1 is parent2:
            parent2 = random.choice(selected)

        # Create child through crossover
        child = parent1.crossover(parent2)

        # Apply mutation with reduced probability for high-fitness parents
        if (parent1.fitness + parent2.fitness) / 2 > avg_fitness:
            # Reduce mutation for children of good parents by setting a temporary lower rate
            old_mutate = child.mutate
            child.mutate()
        else:
            child.mutate()

        newPopulation.append(child)

    avg_score = sum([ind.score for ind in population]) / len(population)
    # Print generation statistics
    print(f"Generation: {gen} | Avg fitness: {avg_fitness:.2f} | Best fitness: {best_fitness:.2f} | "
          f"Best Score: {best_score:.1f} | Avg Score: {avg_score:.1f} | "
          f"Steps: {population[0].steps:.1f} | States: {len(population[0].states)}")

    return newPopulation
```

Functions for saving the population and the best individual in a permanent file.

Functions to graph the progress of the of the genetic algorithm. The average fitness of each generation.

```
def savePopulation(population, filename, gen=0):
    # Create a backup of the previous file if it exists
    if os.path.exists(filename):
        backup_name = f"{filename}.bak"
        try:
            if os.path.exists(backup_name):
                os.remove(backup_name)
            os.rename(filename, backup_name)
        except:
```

```
            pass

    with open(filename, 'wb') as file:
        pickle.dump((population, gen), file)
    print(f"Saved population to {filename} (Generation {gen})")

def loadPopulation(filename):
    with open(filename, 'rb') as file:
        data = pickle.load(file)

    if isinstance(data, tuple) and len(data) == 2:
        population, gen = data
        print(f"Loaded population from {filename} (Generation {gen})")
        return population, gen
    else:
        print(f"Loaded population from {filename} (old format)")
        return data, 0

def makeGraph(fitnessList, avgFitnessList, genAmount):
    import matplotlib
    matplotlib.use('Agg')  # This line avoids GUI-related errors
    import matplotlib.pyplot as plt

    plt.figure(figsize=(10, 5))
    generations = range(1, genAmount + 1)  # Comenzar desde 1 hasta genAmount

    # Crear dos subplots con diferentes escalas
    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8), sharex=True)

    # Graficar fitness en el primer subplot
    ax1.plot(generations, fitnessList, label='Max Fitness', color='blue')
    ax1.set_ylabel('Max Fitness')
    ax1.legend()
    ax1.grid(True)

    # Graficar score en el segundo subplot
    ax2.plot(generations, avgFitnessList, label='Avg Fitness', color='orange')
    ax2.set_ylabel('Avg Fitness')
    ax2.set_xlabel('Generation')
    ax2.legend()
    ax2.grid(True)

    # Configurar ejes para mostrar solo números enteros
    ax1.xaxis.set_major_locator(matplotlib.ticker.MaxNLocator(integer=True))
    ax2.xaxis.set_major_locator(matplotlib.ticker.MaxNLocator(integer=True))
    ax2.yaxis.set_major_locator(matplotlib.ticker.MaxNLocator(integer=True))  # Números enteros en el eje Y del score

    plt.tight_layout()  # Ajustar el espaciado entre subplots
    plt.savefig('fitness_score_graph.png')
    plt.close()  # Close the figure to free up memory
```

-- Main --

Contains functions to use the previous classes and fucntions to train the snake to play the game

```
def trainSnake(popSize, genAmount, resume_training=True):
    start_gen = 0
    fitnessList = []
    avgFitnessList = []

    try:
        if resume_training:
            population, start_gen = loadPopulation(FILENAME)
            if len(population) != popSize:
                # Adjust population size if needed
                if len(population) < popSize:
                    # Add new individuals
                    population.extend([DecisionTable() for _ in range(popSize - len(population))])
                else:
                    # Keep only the best
                    population = population[:popSize]
            print(f"Resuming from generation {start_gen}")
        else:
            # Ask for confirmation if there's an existing file
```

```
        if os.path.exists(FILENAME):
            user_input = input(f"File {FILENAME} already exists. Do you want to start fresh? (y/n): ")
            if user_input.lower() != 'y':
                population, start_gen = loadPopulation(FILENAME)
                if len(population) < popSize:
                    # Add new individuals
                    population.extend([DecisionTable() for _ in range(popSize - len(population))])
            else:
                raise FileNotFoundError("User chose not to start fresh")

        else:
            raise FileNotFoundError("File does not exist")
    except (FileNotFoundError, EOFError):
        population = [DecisionTable() for _ in range(popSize)]
        print("Created new population")

    # Track best individual across all generations
    best_ever = None
    best_ever_fitness = float('-inf')
    best_ever_score = float('-inf')
    best_ever_gen = 0

    # Main training loop
    for gen in range(start_gen, start_gen + genAmount):
        # Evaluate each individual
        for i, table in enumerate(population):
            print(f"Evaluating individual {i+1}/{len(population)}", end="\r")

            # Play more games for more accurate evaluation
            num_eval_games = 1
            table.calculateFitness(num_games=num_eval_games)

            # Track best individual ever seen
            if table.score > best_ever_score or (table.score == best_ever_score and table.fitness > best_ever_fitness):
                best_ever = copy.deepcopy(table)
                best_ever_fitness = table.fitness
                best_ever_score = table.score
                best_ever_gen = gen

        # Create next generation
        population = nextGen(population, gen)

        # Inject the best individual ever seen back into the population occasionally
        if gen % 5 == 0 and best_ever is not None:
            # Replace a random individual with the best ever
            if len(population) > 0:
                replace_idx = random.randint(len(population) // 2, len(population) - 1)  # Replace from bottom half
                population[replace_idx] = copy.deepcopy(best_ever)
                print(f"Injected best individual from gen {best_ever_gen} (score: {best_ever_score})")

        # Save periodically
        if gen % 5 == 0 or gen == start_gen + genAmount - 1:
            savePopulation(population, FILENAME, gen)

            # Also save the best individual ever in a separate file
            if best_ever is not None:
                best_filename = "best_individual.pkl"
                with open(best_filename, 'wb') as file:
                    pickle.dump((best_ever, best_ever_gen), file)
                print(f"Saved best individual from gen {best_ever_gen} to {best_filename}")

        # Calculate statistics for this generation
        maxFitness = max([ind.fitness for ind in population])
        avgFitness = sum([ind.fitness for ind in population]) / len(population)

        # Append to lists
        fitnessList.append(maxFitness)
        avgFitnessList.append(avgFitness)

    # Final save
    savePopulation(population, FILENAME, start_gen + genAmount)
    makeGraph(fitnessList, avgFitnessList, genAmount)
    return population
```

After training, enables the user to watch the best individual play the game using the decision table.

```
def play_with_best_table():
    try:
        population, gen = loadPopulation(FILENAME)
    except:
        print("No trained population found. Please train first.")
        return

    best = sorted(population, key=lambda x: x.fitness, reverse=True)[0]
    print(f"🧠 Cargando mejor tabla con fitness {best.fitness:.2f}, score {best.score:.1f}, "
          f"steps {best.steps:.1f}, estados {len(best.states)}")

    game = SnakeGame(speed=10)  # Slower speed for better visualization

    gameOver = False
    steps = 0
    last_state = None
    last_action = None

    while not gameOver and steps < 500:  # Increased max steps
        state = best.getState(game)

        if state in best.states:
            action = best.states[state]
            print(f"Estado encontrado: {action}")
        else:
            print("Estado nuevo, eligiendo acción...")
            safe_directions = []
            for d in Direction:
                if not best._is_direction_dangerous(game, d) and game.is_valid_direction_change(d):
                    safe_directions.append(d)

            if safe_directions:
                action = random.choice(safe_directions)
            else:
                valid_directions = [d for d in Direction if game.is_valid_direction_change(d)]
                action = random.choice(valid_directions) if valid_directions else game.direction

        # Store for debugging
        last_state = state
        last_action = action

        gameOver, reward, score = game.play_step(action)
        steps += 1

        # Print current state
        if steps % 10 == 0:
            print(f"Step {steps}, Score: {score}, Action: {action}")

        time.sleep(0.1)

    print(f"🎮 Game Over. Score: {score}, Steps: {steps}")
    if gameOver and score < 5:
        print(f"Last state: {last_state}")
        print(f"Last action: {last_action}")
```

Global Parameters for the AI training

```
POP_SIZE = 300
GEN_AMOUNT = 500
MAX_STEPS_NO_FOOD = 200
FILENAME = "population.pkl"
```

Main Menu and access to training and visualizing the best individual

```
def showMenu():
    print("---- Algoritmo Genético para Snake ----")
    print("1. Entrenar desde cero")
    print("2. Continuar entrenamiento")
    print("3. Jugar con la mejor tabla")
    print("4. Salir")
    print("---------------------------------------")

    try:
        option = int(input("Seleccione una opción: "))
```

```
        if option == 1:
            trainSnake(POP_SIZE, GEN_AMOUNT, resume_training=False)
        elif option == 2:
            trainSnake(POP_SIZE, GEN_AMOUNT, resume_training=True)
        elif option == 3:
            play_with_best_table()
        elif option == 4:
            print("Saliendo...")
        else:
            print("Opción inválida")
            showMenu()
    except ValueError:
        print("Por favor, ingrese un número válido")
        showMenu()
```

```
'''Instruccions:'''
#To use, run all cells. Now after running this cell, a menu will appear, select the wanted option.
#First 2 train the agent from zero or continuing by using the generated .pkl file.
#Third option runs the best individual in the last generation.
showMenu()
```

```
---- Algoritmo Genético para Snake ----
1. Entrenar desde cero
2. Continuar entrenamiento
3. Jugar con la mejor tabla
4. Salir
---------------------------------------
Seleccione una opción: 4
Saliendo...
```