Taller sobre clustering aglomerativo y DBSCAN

Agente Come Frutas

Integrantes: Ayala Ivonne, Cumbal Mateo, Garcés Boris, Morales David, Pereira Alicia

1. Introducción

El presente informe detalla el proceso de diseño, implementación y experimentación para el desarrollo de un agente autónomo de Inteligencia Artificial en el marco del proyecto "Agente Come-Frutas". El desafío se centra en un problema clásico de toma de decisiones secuenciales en un entorno dinámico y con riesgos, sirviendo como un caso de estudio práctico para la aplicación de técnicas avanzadas de Machine Learning.

El problema consiste en un entorno de rejilla de 5x5 en el que un agente debe aprender a navegar de manera eficiente. El objetivo principal es maximizar una puntuación recolectando "frutas" (que otorgan recompensas positivas) y evitando "venenos" (que imponen castigos negativos). La meta final es desarrollar una política de comportamiento óptima que permita al agente limpiar el tablero de todas las frutas, garantizando su supervivencia al esquivar todos los venenos presentes.

Para alcanzar este objetivo, el proyecto transitó por un riguroso proceso de experimentación, explorando múltiples paradigmas de la Inteligencia Artificial. Se inició con un enfoque en el Aprendizaje por Refuerzo (Reinforcement Learning), implementando y depurando algoritmos de vanguardia como Deep Q-Networks (DQN) y su variante mejorada, Double DQN (DDQN).

Frente a los desafíos clásicos de convergencia y estabilidad inherentes a RL, la investigación se expandió para incluir otras estrategias. Se exploraron los Algoritmos Genéticos, un enfoque basado en principios de evolución, y el Aprendizaje por Imitación, una potente técnica de aprendizaje supervisado que requirió el desarrollo de un "oráculo" experto basado en el algoritmo de búsqueda A*.

A continuación, se presenta el código documentado de la implementación final, reflejando la culminación de este profundo y multifacético proceso de desarrollo.

2. Desarrollo y Experimentación

El proyecto se desarrolló de forma iterativa, comenzando con un algoritmo fundamental y avanzando hacia técnicas más complejas para superar los desafíos encontrados.

2.1. Punto de Partida: Q-Learning

El primer enfoque implementado fue el Q-Learning tradicional. Este algoritmo fundamental de Aprendizaje por Refuerzo sirvió como base para entender la interacción entre el agente y el entorno. El "cerebro" del agente en este modelo es una Tabla Q, una simple matriz que almacena el valor esperado para cada acción en cada posible estado (casilla) del tablero. El agente aprende actualizando esta tabla después de cada movimiento mediante la ecuación de Bellman, y decide sus acciones balanceando la exploración (probar movimientos al azar) y la explotación (usar la mejor ruta conocida). Aunque efectivo para problemas simples, este enfoque demostró tener limitaciones en escenarios más complejos, lo que motivó la transición a redes neuronales.

```
# --- PARÁMETROS DEL APRENDIZAJE POR REFUERZO (Q-LEARNING) ---
# Estos son los "hiperparámetros" que controlan cómo aprende el agente.
RECOMPENSA_FRUTA = 100          # Puntuación alta por encontrar una fruta.
CASTIGO_VENENO = -100           # Castigo fuerte por tocar un veneno.
RECOMPENSA_MOVIMIENTO = -0.1    # Pequeño castigo por cada movimiento para incentivar la efi
ALPHA = 0.1                     # Tasa de aprendizaje.
GAMMA = 0.9                     # Factor de descuento.
EPSILON = 1.0                   # Tasa de exploración inicial.
EPSILON_DECAY = 0.9995          # Factor de decaimiento de epsilon.
MIN_EPSILON = 0.01              # Mínima tasa de exploración.
NUM_EPISODIOS_ENTRENAMIENTO = 20000 # Número de partidas que jugará el agente para aprender.
```

```
# --- CLASE DEL AGENTE ---
# Define el "cerebro" del agente. Contiene la Tabla Q y la lógica para aprender y decidir.
class AgenteQLearning:
    def __init__(self, num_estados, num_acciones):
        self.num_acciones = num_acciones
        # La Tabla Q es una matriz que almacena el "valor" de cada acción en cada estado posi
        # Aquí, el estado está definido por la posición (x, y) del agente en el tablero.
        self.q_table = np.zeros((num_estados[0], num_estados[1], num_acciones))
        self.epsilon = EPSILON  # Tasa de exploración (curiosidad).
```

```
    def elegir_accion(self, estado):
        """Decide qué acción tomar usando la estrategia epsilon-greedy."""
        # Con probabilidad epsilon, toma una acción aleatoria (exploración).
        if random.uniform(0, 1) < self.epsilon:
            return random.randint(0, self.num_acciones - 1)
        # De lo contrario, elige la mejor acción conocida según la Tabla Q (explotación).
        else:
            return np.argmax(self.q_table[estado])

    def actualizar_q_table(self, estado, accion, recompensa, nuevo_estado):
        """Actualiza el valor en la Tabla Q usando la fórmula de Bellman."""
        valor_antiguo = self.q_table[estado][accion]
        # El valor futuro es el máximo valor Q que se puede obtener desde el nuevo estado.
        valor_futuro_maximo = np.max(self.q_table[nuevo_estado])

        # Fórmula de Q-Learning: se actualiza el valor antiguo basado en la recompensa
        # obtenida y el valor futuro esperado.
        nuevo_q = valor_antiguo + ALPHA * (
            recompensa + GAMMA * valor_futuro_maximo - valor_antiguo
        )
        self.q_table[estado][accion] = nuevo_q

    def decaimiento_epsilon(self):
        """Reduce gradualmente el valor de epsilon para pasar de explorar a explotar."""
        if self.epsilon > MIN_EPSILON:
            self.epsilon *= EPSILON_DECAY
```

2.2 DQN con CNN (Deep Q-Network con Red Convolucional)

Para superar las limitaciones de la Tabla Q, el proyecto evolucionó hacia las Deep Q-Networks (DQN). El cerebro del agente fue reemplazado por una

Red Neuronal Convolucional (CNN), lo que le permitió interpretar el tablero como una imagen y reconocer patrones espaciales. Esto es fundamental para que el agente entienda las relaciones entre su posición, las frutas y los venenos.

```
# Arquitectura de la red neuronal que actúa como el "cerebro" visual del agente. [cite: 1260]
class CNN_DQN(nn.Module):
    def __init__(self, h, w, outputs): [cite: 1279]
        super(CNN_DQN, self).__init__()
        # Capas convolucionales para "ver" el tablero [cite: 1265]
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
```

```
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)

        # Cálculo del tamaño para la capa lineal [cite: 1292]
        def conv2d_size_out(size, kernel_size=3, stride=1, padding=1):
            return (size + 2 * padding - kernel_size) // stride + 1

        convw = conv2d_size_out(conv2d_size_out(w))
        convh = conv2d_size_out(conv2d_size_out(h))
        linear_input_size = convw * convh * 32

        # Capas lineales para "decidir" el valor de cada acción [cite: 1269]
        self.fc1 = nn.Linear(linear_input_size, 256)
        self.fc2 = nn.Linear(256, outputs)

    def forward(self, x):
        x = nn.functional.relu(self.conv1(x))
        x = nn.functional.relu(self.conv2(x))
        x = x.view(x.size(0), -1)
        x = nn.functional.relu(self.fc1(x))
        return self.fc2(x)
```

2.3 DDQN

Para mejorar la estabilidad, se implementó la variante Double DQN (DDQN). Este algoritmo reduce la tendencia del DQN a sobreestimar el valor de las acciones. Lo logra utilizando dos redes: la

red principal (que se actualiza constantemente) para seleccionar la mejor acción futura, y la red objetivo (una copia más antigua y estable) para evaluar el valor de dicha acción. Esta separación hace el aprendizaje más robusto.

```
# El método de aprendizaje del agente DDQN. [cite: 1446]
def replay(self, batch_size):
    if len(self.memory) < batch_size: [cite: 1469]
        return

    minibatch = random.sample(self.memory, batch_size) [cite: 1474]

    states = torch.FloatTensor(np.array([e[0] for e in minibatch]))
    actions = torch.LongTensor([e[1] for e in minibatch]).unsqueeze(1)
    rewards = torch.FloatTensor([e[2] for e in minibatch]).unsqueeze(1)
    next_states = torch.FloatTensor(np.array([e[3] for e in minibatch]))
```

```
    dones = torch.BoolTensor([e[4] for e in minibatch]).unsqueeze(1)

    current_q_values = self.model(states).gather(1, actions)

    # --- LÓGICA CLAVE DE DOUBLE DQN --- [cite: 1485]
    with torch.no_grad():
        # 1. Red principal SELECCIONA la mejor acción para el siguiente estado. [cite: 1488]
        best_next_actions = self.model(next_states).max(1)[1].unsqueeze(1)

        # 2. Red objetivo EVALÚA el valor de la acción seleccionada. [cite: 1489]
        next_q_values_target = self.target_model(next_states).gather(1, best_next_actions)

    target_q_values = rewards + (self.gamma * next_q_values_target * (~dones))

    loss = self.criterion(current_q_values, target_q_values)

    self.optimizer.zero_grad()
    loss.backward()

    torch.nn.utils.clip_grad_value_(self.model.parameters(), 1)
    self.optimizer.step()
```

2.4. Paradigma Alternativo: Algoritmos Genéticos

Para explorar una solución completamente diferente, se implementó un Algoritmo Genético. Este enfoque no se basa en la experiencia de un solo agente, sino en la evolución de una población completa a lo largo de generaciones. El "ADN" de cada agente está representado por los pesos de su red neuronal. El proceso simula la selección natural: los agentes con mejor rendimiento ("fitness") son seleccionados como "padres", sus genes se combinan y mutan para crear una nueva generación de "hijos", reemplazando a los menos aptos. Este ciclo de evaluación, selección, cruce y mutación permite que la población evolucione gradualmente hacia soluciones óptimas

```
# El ciclo de vida de una generación en el Algoritmo Genético.
def run_generation(population, env):
    # FASE 1: EVALUACIÓN DE FITNESS
    evaluate_fitness(population, env)

    # FASE 2: SELECCIÓN DE PADRES
    parents = selection(population)

    # ANÁLISIS Y GUARDADO DEL MEJOR AGENTE
```

```
    best_agent_of_gen = parents[0]
    torch.save(best_agent_of_gen.network.state_dict(), "best_agent_genes.pth")

    # FASE 3: CREACIÓN DE NUEVA GENERACIÓN
    new_population = []

    # ELITISMO: Los mejores agentes pasan directamente
    new_population.extend(parents[:ELITISM_COUNT])

    # REPRODUCCIÓN: Llenar el resto con descendencia
    while len(new_population) < POPULATION_SIZE:
        parent1, parent2 = random.sample(parents, 2)
        # Cruzamiento: combinar genes de padres
        child = crossover(parent1, parent2)
        # Mutación: introducir variación genética
        child = mutate(child)
        new_population.append(child)

    # Reemplazar población anterior
    return new_population
```

2.5 Solución Definitiva: Aprendizaje por Imitación y Currículo

Tras los desafíos encontrados, la estrategia final y más exitosa fue el Aprendizaje por Imitación. Este enfoque cambia el paradigma: en lugar de que el agente descubra una estrategia por sí mismo, se le enseña directamente imitando a un experto perfecto. Para ello, se implementó el algoritmo de búsqueda A* como un "oráculo" capaz de calcular siempre la ruta óptima en cualquier tablero.

El primer paso fue generar un "libro de texto" para el agente, creando miles de ejemplos de jugadas perfectas. Se crearon datasets separados para escenarios de 1, 2, 3 y 4 frutas.

```
# Función clave para generar los datos de entrenamiento para el currículo.
def generate_expert_data_for_n_fruits(num_fruits, num_samples, output_file):
    """Genera un dataset experto para un número específico de frutas."""
    env = GridEnvironment()
    expert_data = []
    print(f"Generando {num_samples} muestras para {num_fruits} fruta(s)...")

    while len(expert_data) < num_samples:
        # Generar un escenario aleatorio con `num_fruits` frutas.
        # ... (código de generación de escenario) ...
```

```
        env.reset(agent_pos=agent_p, fruit_pos=fruit_p, poison_pos=poison_p)

        # Simular el episodio usando el experto A* paso a paso.
        for _ in range(50):
            if not env.fruit_pos: break

            # El experto A* recalcula la mejor ruta desde la posición actual.
            path = a_star_search(GRID_SIZE, env.agent_pos, goal_fruit, env.poison_pos)

            if path and len(path) > 0:
                # Se toma solo el primer paso de la ruta óptima.
                action = get_action(env.agent_pos, path[0])
                state = env.get_state()
                # Se guarda el par (estado_actual, accion_correcta).
                expert_data.append((state, action))
                env.step(action)
            else:
                break

    with open(output_file, "wb") as f:
        pickle.dump(expert_data, f)
    print(f"Dataset '{output_file}' creado.")
```

La innovación crucial fue entrenar al agente con un Aprendizaje por Currículo. En lugar de mostrarle todos los datos a la vez, el agente fue entrenado por etapas, comenzando con la tarea más fácil (1 fruta) y aumentando gradualmente la dificultad. Esto le permitió dominar primero los conceptos básicos de "ganar" antes de enfrentarse a escenarios más complejos.

```
# El script de entrenamiento que implementa la estrategia de Curriculum Learning.
if __name__ == "__main__":
    model = AgentNetwork()
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
    criterion = nn.CrossEntropyLoss()

    # El Currículo: una lista de "lecciones" de dificultad creciente.
    CURRICULUM = [
        ("expert_data_1_fruit.pkl", 25),   # Lección 1: Nivel básico
        ("expert_data_2_fruits.pkl", 30),  # Lección 2: Nivel intermedio
        ("expert_data_3_fruits.pkl", 40),  # Lección 3: Nivel avanzado
        ("expert_data_4_fruits.pkl", 50)   # Lección 4: Nivel experto
    ]
```

```
    # Bucle principal que itera a través de las lecciones del currículo.
    for i, (dataset_file, num_epochs) in enumerate(CURRICULUM):
        print(f"\n--- Iniciando Lección {i+1}/{len(CURRICULUM)}: {dataset_file} ---")

        # Cargar el dataset de la lección actual.
        with open(dataset_file, "rb") as f:
            data = pickle.load(f)

        # Preparar los datos para el entrenamiento.
        # ... (código de creación del DataLoader) ...

        # Entrenar el modelo (sin reiniciarlo) por el número de épocas de la lección.
        for epoch in range(num_epochs):
            # ... (bucle de entrenamiento estándar de aprendizaje supervisado) ...
            print(f"  Época {epoch+1}/{num_epochs}, Pérdida: {avg_loss:.4f}")

    # Guardar el modelo final, que ha aprendido de todo el currículo.
    torch.save(model.state_dict(), "imitacion_model.pth")
```

Jugador humano

```
"""
Modo de juego humano con controles aleatorios - Come Frutas.

Este módulo implementa una versión jugable del entorno donde un humano puede
controlar el agente directamente. La característica única es que los controles
de movimiento se asignan aleatoriamente cada vez que se inicia una partida,
añadiendo un elemento de desafío y adaptabilidad.

Características principales:
- Modo Setup: Configuración manual del escenario
- Modo Humano: Control directo del agente por el jugador
- Controles aleatorios: Mapeo aleatorio de teclas a movimientos
- Interfaz intuitiva: Gráficos y feedback visual
- Desafío adaptativo: Cada partida requiere aprender nuevos controles

Propósito educativo:
- Comparar rendimiento humano vs. IA
- Experimentar la dificultad de adaptación a controles cambiantes
- Entender la importancia de la consistencia en interfaces
```

```
- Apreciar la flexibilidad del aprendizaje humano

Autor: [Tu nombre]
Fecha: Agosto 2025
"""

import pygame
import os
import random
import string

# CONFIGURACIÓN DEL ENTORNO VISUAL
"""
Parámetros visuales y dimensiones de la interfaz de juego.
Utiliza celdas más grandes (120px) para mejor visibilidad durante el juego manual.
"""
GRID_WIDTH = 5              # Ancho de la cuadrícula en celdas
GRID_HEIGHT = 5             # Alto de la cuadrícula en celdas
CELL_SIZE = 120             # Tamaño de cada celda en píxeles (mayor para juego manual)
SCREEN_WIDTH = GRID_WIDTH * CELL_SIZE    # Ancho total de la ventana (600px)
SCREEN_HEIGHT = GRID_HEIGHT * CELL_SIZE  # Alto del área de juego (600px)

# PALETA DE COLORES CONSISTENTE
"""
Esquema de colores oscuro profesional, consistente con otros módulos del proyecto.
"""
COLOR_FONDO = (25, 25, 25)      # Gris muy oscuro para el fondo
COLOR_LINEAS = (40, 40, 40)     # Gris oscuro para líneas de cuadrícula
COLOR_CURSOR = (255, 255, 0)    # Amarillo brillante para cursor de selección
COLOR_TEXTO = (230, 230, 230)   # Gris claro para texto legible

# SISTEMA DE CONTROLES ALEATORIOS
"""
Genera un conjunto de teclas válidas para asignación aleatoria de controles.
Se evitan teclas especiales para prevenir conflictos con funciones del sistema.
"""
TECLAS_VALIDAS = [getattr(pygame, f"K_{c}") for c in string.ascii_lowercase + string.digits]

class EntornoHumano:
    """
    Entorno de juego optimizado para control humano directo.
```

```
    Esta clase maneja la lógica del juego cuando un humano controla el agente,
    incluyendo movimiento, colisiones, recolección de objetos y condiciones
    de victoria/derrota. Se enfoca en proporcionar feedback inmediato y
    una experiencia de juego fluida.

    Diferencias con entornos de IA:
    - Feedback inmediato con mensajes en consola
    - Lógica de juego simplificada (sin recompensas numéricas)
    - Terminación inmediata en victoria/derrota
    - Controles responsivos para jugabilidad humana

    Attributes:
        agente_pos (tuple): Posición actual del agente (x, y)
        frutas (set): Conjunto de posiciones con frutas
        venenos (set): Conjunto de posiciones con venenos
        paredes (set): Conjunto de posiciones con paredes/obstáculos
    """
    def __init__(self):
        """
        Inicializa el entorno con configuración vacía.

        El agente comienza en la esquina superior izquierda (0,0) y todos
        los conjuntos de elementos están vacíos, permitiendo configuración manual.
        """
        self.agente_pos = (0, 0)    # Posición inicial del agente
        self.frutas = set()         # Conjunto de posiciones de frutas
        self.venenos = set()        # Conjunto de posiciones de venenos
        self.paredes = set()        # Conjunto de posiciones de paredes

    def reset(self):
        """
        Resetea la posición del agente al inicio del juego.

        Coloca al agente en la posición inicial (0,0) sin modificar
        la configuración del escenario. Utilizado al comenzar una nueva partida.
        """
        self.agente_pos = (0, 0)

    def limpiar(self):
        """
        Elimina todos los elementos del entorno.
```

```
        Limpia frutas, venenos y paredes del escenario, dejando una
        cuadrícula vacía para configuración desde cero.
        """
        self.frutas.clear()
        self.venenos.clear()
        self.paredes.clear()

    def step(self, accion):
        """
        Ejecuta una acción del jugador humano en el entorno.

        Procesa el movimiento del agente, verifica colisiones y maneja
        las interacciones con elementos del entorno. Proporciona feedback
        inmediato al jugador mediante mensajes en consola.

        Args:
            accion (int): Dirección de movimiento:
                          0 = Arriba (decrementar y)
                          1 = Abajo (incrementar y)
                          2 = Izquierda (decrementar x)
                          3 = Derecha (incrementar x)

        Returns:
            bool: True si el juego terminó (victoria o derrota), False si continúa
        """
        # Calcular nueva posición basada en la acción
        x, y = self.agente_pos
        if accion == 0:     # Arriba
            y -= 1
        elif accion == 1:   # Abajo
            y += 1
        elif accion == 2:   # Izquierda
            x -= 1
        elif accion == 3:   # Derecha
            x += 1

        # Verificar colisiones: límites del tablero o paredes
        if x < 0 or x >= GRID_WIDTH or y < 0 or y >= GRID_HEIGHT or (x, y) in self.paredes:
            # Movimiento inválido: no actualizar posición
            return False

        # Movimiento válido: actualizar posición del agente
```

```
        self.agente_pos = (x, y)

        # Verificar interacciones con elementos del entorno
        if self.agente_pos in self.frutas:
            # Fruta recogida: eliminar del conjunto
            self.frutas.remove(self.agente_pos)

            # Verificar condición de victoria
            if not self.frutas:
                print("\n ¡Ganaste! Recolectaste todas las frutas.\n")
                return True  # Juego terminado con éxito

        elif self.agente_pos in self.venenos:
            # Veneno tocado: derrota inmediata
            print("\n ¡Oh no! Tocaste un veneno.\n")
            return True  # Juego terminado con fallo

        # Continuar juego
        return False

    def dibujar(self, pantalla, modo, cursor_pos, img_fruta, img_veneno, img_pared, img_agent
        """
        Renderiza el estado completo del entorno con interfaz interactiva.

        Dibuja todos los elementos visuales del juego incluyendo grid, objetos
        del entorno y cursor de selección. Proporciona feedback visual para
        la interacción del jugador en diferentes modos (colocación/juego).

        Args:
            pantalla (pygame.Surface): Superficie donde renderizar
            modo (str): Modo actual de la interfaz ('frutas', 'venenos', 'paredes', 'jugar')
            cursor_pos (tuple): Posición (x,y) del cursor en coordenadas de grid
            img_fruta (pygame.Surface): Sprite de las frutas
            img_veneno (pygame.Surface): Sprite de los venenos
            img_pared (pygame.Surface): Sprite de las paredes
            img_agente (pygame.Surface): Sprite del agente
            _ : Parámetro no utilizado (compatibilidad de interfaz)

        Note:
            Renderiza en orden específico: fondo, grid, objetos, agente, cursor.
            El cursor cambia de color según el modo de colocación activo.
        """
```

```
        # Limpiar pantalla con color de fondo
        pantalla.fill(COLOR_FONDO)

        # Dibujar líneas del grid para guía visual
        for x in range(0, SCREEN_WIDTH, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (x, 0), (x, SCREEN_HEIGHT))
        for y in range(0, SCREEN_HEIGHT, CELL_SIZE):
            pygame.draw.line(pantalla, COLOR_LINEAS, (0, y), (SCREEN_WIDTH, y))

        # Dibujar elementos del entorno
        for fruta in self.frutas:
            pantalla.blit(img_fruta, (fruta[0]*CELL_SIZE, fruta[1]*CELL_SIZE))
        for veneno in self.venenos:
            pantalla.blit(img_veneno, (veneno[0]*CELL_SIZE, veneno[1]*CELL_SIZE))
        for pared in self.paredes:
            pantalla.blit(img_pared, (pared[0]*CELL_SIZE, pared[1]*CELL_SIZE))

        # Dibujar agente (jugador) - siempre visible en primer plano
        pantalla.blit(img_agente, (self.agente_pos[0]*CELL_SIZE, self.agente_pos[1]*CELL_SIZ

        # Dibujar cursor de selección en modo configuración
        if modo == "SETUP":
            cursor_rect = pygame.Rect(cursor_pos[0]*CELL_SIZE, cursor_pos[1]*CELL_SIZE, CELL_
            pygame.draw.rect(pantalla, COLOR_CURSOR, cursor_rect, 3)

        # Renderizar información de interfaz
        font = pygame.font.Font(None, 30)
        pantalla.blit(font.render(f"Modo: {modo}", True, COLOR_TEXTO), (10, SCREEN_HEIGHT + 5
        pantalla.blit(font.render("F: Fruta, V: Veneno, W: Pared, C: Limpiar, H: Jugar", True
        pantalla.blit(font.render("Descubre los controles ocultos usando letras/números", Tru

def cargar_imagen(nombre, fallback_color):
    """
    Carga una imagen desde archivo con sistema de respaldo.

    Intenta cargar una imagen sprite desde el directorio actual.
    Si la carga falla, crea una superficie de color sólido como respaldo.
    Escala automáticamente al tamaño de celda definido.

    Args:
        nombre (str): Nombre del archivo de imagen a cargar
        fallback_color (tuple): Color RGB (r,g,b) para superficie de respaldo
```

```
    Returns:
        pygame.Surface: Superficie cargada y escalada, o superficie de color
                        si la carga falló

    Note:
        Todas las imágenes se escalan a CELL_SIZE x CELL_SIZE píxeles.
        Utiliza convert_alpha() para optimizar el renderizado con transparencia.
    """
    try:
        # Construir ruta completa al archivo de imagen
        ruta = os.path.join(os.path.dirname(__file__), nombre)
        # Cargar imagen con soporte de transparencia
        img = pygame.image.load(ruta).convert_alpha()
        # Escalar a tamaño de celda estándar
        return pygame.transform.scale(img, (CELL_SIZE, CELL_SIZE))
    except:
        # Crear superficie de respaldo con color sólido si falla la carga
        s = pygame.Surface((CELL_SIZE, CELL_SIZE))
        s.fill(fallback_color)
        return s

def generar_controles_aleatorios():
    """
    Genera un mapeo aleatorio de teclas para controles de movimiento.

    Crea una asignación aleatoria entre teclas del teclado y direcciones
    de movimiento para añadir un elemento de desafío y descubrimiento
    al juego. Los jugadores deben encontrar qué teclas controlan cada dirección.

    Returns:
        dict: Mapeo de códigos de tecla pygame a acciones de movimiento:
              {tecla_pygame: accion_int}
              donde accion_int es 0=Arriba, 1=Abajo, 2=Izquierda, 3=Derecha

    Note:
        Utiliza teclas alfanuméricas (A-Z, 0-9) para máxima compatibilidad.
        Garantiza que cada dirección tenga exactamente una tecla asignada.
    """
    # Seleccionar 4 teclas aleatorias del conjunto disponible
    teclas = random.sample(TECLAS_VALIDAS, 4)
    # Crear lista de acciones de movimiento
    acciones = [0, 1, 2, 3]  # Arriba, abajo, izquierda, derecha
```

```
    # Mezclar aleatoriamente las acciones
    random.shuffle(acciones)
    # Crear diccionario de mapeo tecla->acción
    return dict(zip(teclas, acciones))

def main():
    """
    Función principal del juego en modo humano.

    Inicializa Pygame, configura la ventana de juego y ejecuta el bucle
    principal que maneja dos modos: configuración del entorno y juego
    con controles aleatorios. Proporciona una experiencia interactiva
    donde el jugador puede diseñar niveles y luego jugarlos.

    Flujo del juego:
        1. Modo SETUP: Colocar frutas, venenos y paredes con el mouse
        2. Modo JUGAR: Controlar agente con teclas aleatorias descubiertas
        3. Victoria: Recolectar todas las frutas
        4. Derrota: Tocar veneno

    Controles SETUP:
        - Mouse: Mover cursor
        - F: Colocar fruta
        - V: Colocar veneno
        - W: Colocar pared
        - C: Limpiar todo
        - H: Iniciar juego

    Controles JUGAR:
        - Teclas aleatorias para movimiento (descubrir experimentando)
        - ESC: Volver a configuración
    """
    # Inicializar Pygame y configurar ventana
    pygame.init()
    pantalla = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT + 100))
    pygame.display.set_caption("Modo Humano Aleatorio - Come Frutas")

    # Inicializar entorno y variables de estado
    entorno = EntornoHumano()
    cursor_pos = [0, 0]
    modo = "SETUP"  # Modo inicial: configuración del entorno
    mapeo_controles = {}  # Mapeo de teclas aleatorias (generado al jugar)
```

```
# Cargar sprites con colores de respaldo
img_fruta = cargar_imagen("fruta.png", (40, 200, 40))
img_veneno = cargar_imagen("veneno.png", (255, 50, 50))
img_pared = cargar_imagen("pared.jpg", (80, 80, 80))
img_agente = cargar_imagen("agente.png", (60, 100, 255))

# Variables de control del juego
reloj = pygame.time.Clock()
corriendo = True

# Bucle principal del juego
while corriendo:
    # Procesar eventos de entrada
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            corriendo = False
        elif evento.type == pygame.KEYDOWN:
            if evento.key == pygame.K_s:
                modo = "SETUP"
            elif evento.key == pygame.K_h:
                modo = "HUMANO"
                entorno.reset()
                mapeo_controles = generar_controles_aleatorios()

            if modo == "SETUP":
                if evento.key == pygame.K_UP: cursor_pos[1] = max(0, cursor_pos[1]-1)
                elif evento.key == pygame.K_DOWN: cursor_pos[1] = min(GRID_HEIGHT-1, curs
                elif evento.key == pygame.K_LEFT: cursor_pos[0] = max(0, cursor_pos[0]-1
                elif evento.key == pygame.K_RIGHT: cursor_pos[0] = min(GRID_WIDTH-1, curs
                # Colocación de elementos con teclas específicas
                pos = tuple(cursor_pos)
                if evento.key == pygame.K_f:
                    # F: Colocar/quitar fruta (toggle)
                    entorno.frutas.symmetric_difference_update({pos})
                    entorno.venenos.discard(pos)
                    entorno.paredes.discard(pos)
                elif evento.key == pygame.K_v:
                    # V: Colocar/quitar veneno (toggle)
                    entorno.venenos.symmetric_difference_update({pos})
                    entorno.frutas.discard(pos)
                    entorno.paredes.discard(pos)
                elif evento.key == pygame.K_w:
```

```
                        # W: Colocar/quitar pared (toggle)
                        entorno.paredes.symmetric_difference_update({pos})
                        entorno.frutas.discard(pos)
                        entorno.venenos.discard(pos)
                    elif evento.key == pygame.K_c:
                        # C: Limpiar todo el entorno
                        entorno.limpiar()

                # Controles específicos del modo HUMANO
                elif modo == "HUMANO":
                    if evento.key in mapeo_controles:
                        # Ejecutar acción de movimiento con tecla aleatoria
                        accion = mapeo_controles[evento.key]
                        terminado = entorno.step(accion)
                        if terminado:
                            # Volver a configuración al terminar el juego
                            modo = "SETUP"

        # Renderizar estado actual del juego
        entorno.dibujar(pantalla, modo, cursor_pos, img_fruta, img_veneno, img_pared, img_ag
        pygame.display.flip()
        reloj.tick(30)

    # Limpiar recursos al salir
    pygame.quit()

if __name__ == '__main__':
    main()
```

3. Resultados y Conclusión

El viaje a través de estos diferentes paradigmas de IA fue revelador. Mientras que los enfoques de Aprendizaje por Refuerzo y Algoritmos Genéticos mostraron potencial, también exhibieron dificultades para converger de manera consistente en un entorno tan "frágil", donde un solo error puede llevar al fracaso.

La estrategia de Aprendizaje por Imitación combinada con el Aprendizaje por Currículo demostró ser la más efectiva y robusta. Al aprender de un experto perfecto y hacerlo de manera gradual, el agente final fue capaz de generalizar su conocimiento y resolver de manera confiable tanto escenarios simples como complejos. Para la demostración visual, se desarrolló una interfaz en Pygame que permite a los usuarios configurar sus propios niveles y observar al agente entrenado resolverlos en tiempo real.

Este proyecto subraya una lección fundamental en el desarrollo de IA: a menudo, la estrategia de entrenamiento y la calidad de los datos son tan o más importantes que la elección del algoritmo en sí. El resultado es un agente competente y una profunda comprensión práctica de los desafíos y soluciones en el campo del Machine Learning.