**Instituto Tecnológico y de Estudios Superiores de Monterrey**

Campus Monterrey

**Modelación de sistemas multiagentes con gráficas computacionales (Gpo 302)**

**Profesor:**

Luis Alberto Muñoz Ubando

Raúl Valente Ramírez Velarde

**Avance 3**


**Integrantes:**

Alonso Abimael Morales Reyna A01284747

Marco Ottavio Podesta Vezzali A00833604

Ernesto Poisot Avila A01734765

Sergio Ortiz Malpica A01284951


**Equipo 4**

**Fecha de entrega:** 28/11/2023

## Introducción

Durante este bloque nosotros aprendimos sobre la modelación de sistemas multiagentes para simular situaciones y procesos reales, con el propósito de encontrar una forma de optimizarlos, debido a que llegar a optimizar el resultado de procesos es de interés para muchas empresas y organizaciones ya que les permite ahorrar tiempo y recursos.

## Descripción del Reto

La industria de la agricultura es esencial para la subsistencia de la sociedad, debido a que los cultivos que genera son una fuente importante de alimentos. Sin embargo, los procesos de recolección de cultivos pueden ser complicados, haciendo que los procesos cuesten más y sean menos productivos, por lo que nuestro socio formador John Deere nos pidió crear una simulación de la cosecha de trigo con el fin de encontrar la forma más eficiente de cosechar este cultivo. De esta forma no solo se reducen los costos de recursos cómo la gasolina, también se reduce la cantidad de dióxido de carbono generado al reducir el tiempo que se utilizan los tractores.

## Agentes Identificados

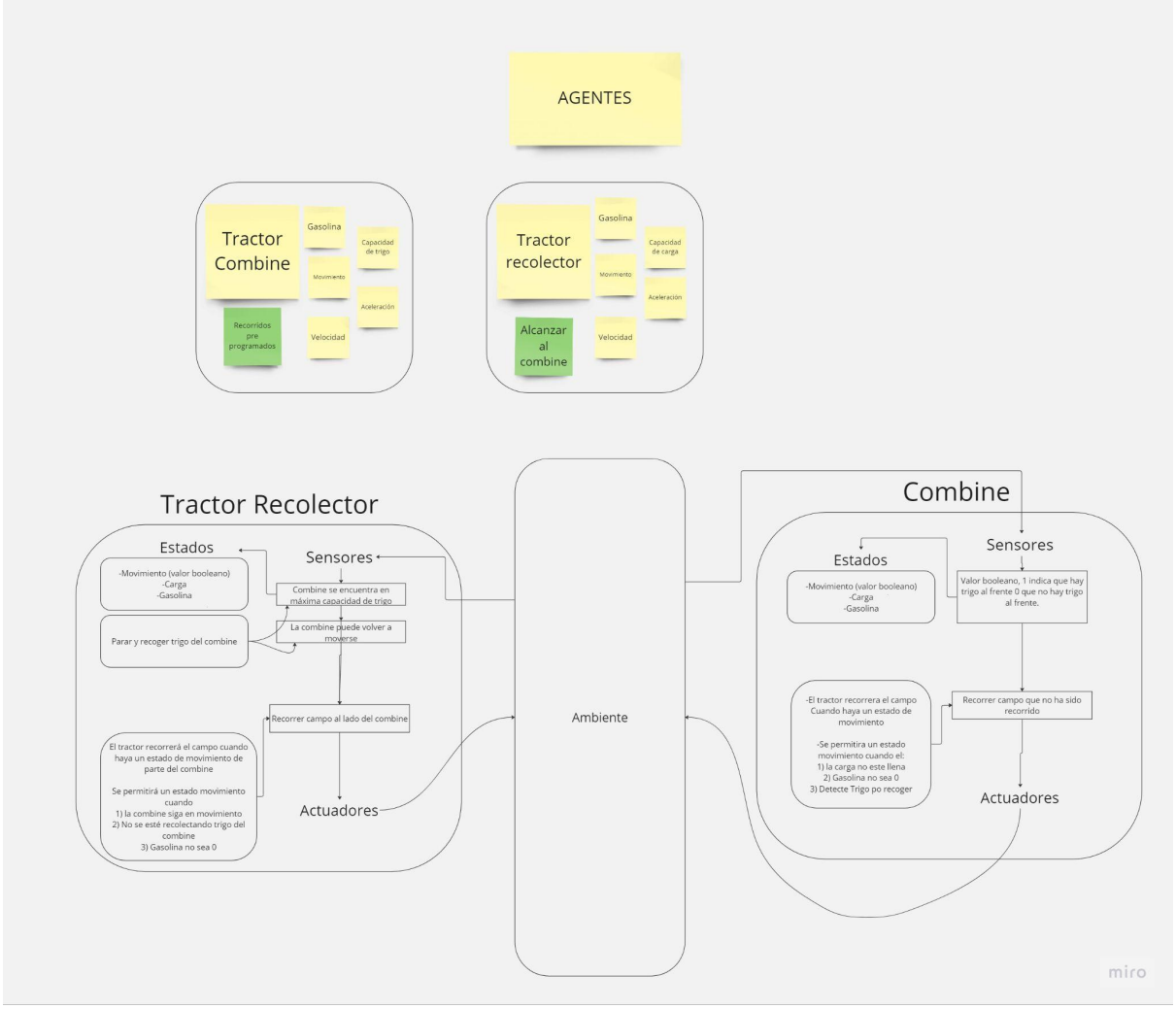
### Cosechadora

El agente recolecta trigo y es capaz de seguir un camino para recolectar el trigo que se encuentre en el camino.El agente tiene un límite de capacidad de trigo y de gasolina. Cada paso que hace, consume gasolina y cada trigo que recoge, es acumulado en su capacidad de carga. Si se llega a una capacidad de carga mayor al 75%, o su capacidad de gasolina se reduce del 25%, este se detiene para esperar al tractor recolector, ya sea para que recolecta su trigo o recargue su gasolina.

### Tractor Recolector

El agente representa un tractor que recolecta el trigo que ha procesado la cosechadora. Además, ofrece la posibilidad de descargar el combine y rellenar su gasolina. El agente tiene un límite de capacidad de almacenamiento de trigo y gasolina. Cada paso que hace, consume gasolina. Si se llega a una capacidad de carga mayor al 75%, o su capacidad de gasolina se reduce del 25%, este se detiene para esperar al tractor recolector. El agente está en un monitoreo constante de los estados de la cosechadora, si determina que la cosechadora necesita ayuda, este va a ir a hacer las acciones necesarias (descargar trigo, rellenar gasolina).

## Diagrama de clase y protocolo de interacción entre agentes




## Explicación de la solución

En nuestro proyecto, creamos un código en Python que tiene la lógica del movimiento de los agentes, los cuáles son la cosechadora y el tractor recolector. El código controla el comportamiento de los agentes y dicta qué movimientos deben de tomar. Para esto se toma en cuenta el ambiente que será el campo, su tamaño y las recompensas que ofrece dependiendo de la acción que el agente realice. Utilizamos una técnica de machine learning que se llama Q-Learning, la cuál consiste en utilizar una tabla Q que contiene los valores de cada acción. Se le asigna un valor negativo a los movimientos para entrenar al agente para que tome la menor cantidad de acciones en llegar al resultado, en este caso, cosechar todo el trigo en el menor tiempo posible. Después de cada iteración de entrenamiento se cambian los valores de la tabla Q para obtener un resultado más optimizado.

En Unity, simulamos un campo de una granja el cuál es un espacio cuadrado que se encuentra rodeado por árboles y en el centro se encuentra un campo de trigo que se

genera una vez que se inicia el programa. El tamaño del campo puede ser programado desde Unity.

Unity utiliza exclusivamente C# para los scripts que dictan el comportamiento de objetos, por lo que unimos la lógica de Python con Unity utilizando una conexión cliente-servidor mediante una api de javascript, que detecta el resultado que se obtenga del código de la simulación de python, enviándolo al cliente de unity.

### Código de implementación entre agentes

```
import numpy as np
import gymnasium as gym
from gymnasium import spaces

EPISODES = 10
MAX_STEPS = 200
COLUMNS = 10
ROWS = 5
ACTION_MAPPINGS = {
    0: (-1, 0),  # Move up
    1: (1, 0),   # Move down
    2: (0, -1),  # Move left
    3: (0, 1)    # Move right
}

class GridEnv1(gym.Env):
    def __init__(self):
        super(GridEnv1, self).__init__()
        self.action_space = spaces.Discrete(4) # Up, Down, Left, Right
        self.observation_space = spaces.Box(low=np.array([0, 0]), high=np.array([ROWS - 1,
COLUMNS - 1]), dtype=np.int32)
        self.reward_map = None
        self.obstacle_position = [np.random.randint(ROWS), np.random.randint(COLUMNS)]
        self.step_counter = 0
        self.total_rewards_collected = 0
        self.reset()

    def reset(self):
        self.state = np.array([0, 0]) # Starting at top-left corner
        self.initialize_reward_map()
        self.step_counter = 0  # Reset the step counter at the start of each episode
        self.total_rewards_collected = 0
        return self.state

    def initialize_reward_map(self):
        self.reward_map = np.full((ROWS, COLUMNS), 1)
        self.reward_map[self.obstacle_position[0], self.obstacle_position[1]] = -100
        # self.reward_map[self.state[0], self.state[1]] = 100  # Set the starting column to 0
        # self.reward_map[ROWS-1,0] = 100
        # self.reward_map[-1, :] = 100  # Set the bottom row to 100

    def reset_reward_map(self):
        self.initialize_reward_map()
        self.reward_map[0,0] = -1  # Mark the starting position
```

```
    def step(self, action):
        # Initialize done as False
        done = False
        reward = 0.


        # Update state based on action with x, y format
        delta = ACTION_MAPPINGS.get(action, (0, 0))
        new_state = np.array([self.state[0] + delta[0], self.state[1] + delta[1]])
        # if np.array_equal(new_state, prev_state):
        #     reward = -10

        # Check and handle boundary conditions
        if new_state[0] < 0 or new_state[0] >= ROWS or new_state[1] < 0 or new_state[1] >=
COLUMNS:
            reward = -100
            done = True
        else:
            # Update the current state and the previous state
            self.state = new_state                # Update the current state to new state

            self.step_counter += 1
            reward += self.calculate_reward()
            done = self.is_done()

            y = self.state[0]
            x = self.state[1]
            self.reward_map[y, x] = -1

        return self.state, reward, done, {}

    def calculate_reward(self):
        # Check if the agent has reached the bottom of the grid
        return self.reward_map[self.state[0], self.state[1]]

    def is_done(self):
        # Check if all cells in the grid have been visited or if a step limit is reached
        max_steps = MAX_STEPS  # Example step limit
        count_minus_one = np.sum(self.reward_map == -1)
        count_minus_hundred = np.sum(self.reward_map == -100)
        return count_minus_one == (self.reward_map.size - 1) and count_minus_hundred == 1 or
self.step_counter >= max_steps


class QLearningAgent1:
    def __init__(self, env, learning_rate=0.2, discount_factor=0.9, epsilon=0.1,
gasoline_capacity=1000, wheat_capacity=100):
        self.env = env
        self.gasoline = gasoline_capacity  # Initial gasoline level
        self.wheat = 0  # Initial wheat level
        self.gasoline_capacity = gasoline_capacity
        self.wheat_capacity = wheat_capacity
```

```
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.q_table = np.zeros((ROWS, COLUMNS, env.action_space.n))

    def choose_action(self, prev_state, state, neg_reward):

        def get_new_state(action):
            delta = ACTION_MAPPINGS.get(action, (0, 0))
            return np.array([state[0] + delta[0], state[1] + delta[1]])

        def is_valid_action(action):
            new_state = get_new_state(action)
            return not np.array_equal(new_state, prev_state)

        def find_second_best_action(q_values):
            # Copy to avoid modifying the original array
            temp_q_values = np.copy(q_values)

            # Find the index of the best action
            best_action = np.argmax(temp_q_values)

            # Mask the best action by setting its value to negative infinity
            temp_q_values[best_action] = -np.inf

            # Find the second best action
            second_best_action = np.argmax(temp_q_values)

            return second_best_action

        action = None
        state_index = (state[0], state[1])
        if neg_reward > 3:
            # Implement logic to find the nearest positive reward
            action = self.find_nearest_positive_reward_action(state)
        elif np.random.uniform(0, 1) < self.epsilon:
            action = self.env.action_space.sample()  # Explore: random action
        else:
            action = np.argmax(self.q_table[state_index])  # Exploit: best known action

        if not is_valid_action(action):
            action = find_second_best_action(self.q_table[state_index])



        # Additional check or fallback strategy if needed
        if not is_valid_action(action):
            # Implement fallback strategy, e.g., select a random action
            action = self.env.action_space.sample()

        return action

    def find_nearest_positive_reward_action(self, state):
        min_distance = float('inf')
```

```
        best_action = None

        # Search the grid for the nearest positive reward
        for y in range(ROWS):
            for x in range(COLUMNS):
                if self.env.reward_map[y, x] > 0:  # Check for positive reward
                    distance = abs(state[0] - y) + abs(state[1] - x)
                    if distance < min_distance:
                        min_distance = distance
                        best_action = self.determine_action_to_reward(state, (y, x))

        return best_action

    def determine_action_to_reward(self, current_state, reward_state):
        dy = reward_state[0] - current_state[0]
        dx = reward_state[1] - current_state[1]

        if abs(dy) > abs(dx):
            return 1 if dy > 0 else 0  # Move down (1) or up (0) based on the y-difference
        else:
            return 3 if dx > 0 else 2  # Move right (3) or left (2) based on the x-difference

    def learn(self, state, action, reward, next_state):
        state_index = (state[0], state[1])
        next_state_index = (next_state[0], next_state[1])
        # Update rule for Q-learning
        best_next_action = np.argmax(self.q_table[next_state_index])
        td_target = reward + self.discount_factor * self.q_table[next_state_index][best_next_action]
        td_error = td_target - self.q_table[state_index][action]
        self.q_table[state_index][action] += self.learning_rate * td_error

class GridEnv2(gym.Env):
    def __init__(self):
        super(GridEnv2, self).__init__()
        self.action_space = spaces.Discrete(4) # Up, Down, Left, Right
        self.observation_space = spaces.Box(low=np.array([0, 0]), high=np.array([ROWS - 1,
COLUMNS - 1]), dtype=np.int32)
        self.reward_map = None
        self.obstacle_position = [np.random.randint(ROWS), np.random.randint(COLUMNS)]
        self.step_counter = 0
        self.total_rewards_collected = 0
        self.reset()

    def reset(self):
        self.state = np.array([ROWS-1, COLUMNS-1]) # Starting at top-left corner
        self.initialize_reward_map()
        self.step_counter = 0  # Reset the step counter at the start of each episode
        self.total_rewards_collected = 0
        return self.state

    def initialize_reward_map(self):
        self.reward_map = np.full((ROWS, COLUMNS), 1)
        self.reward_map[self.obstacle_position[0], self.obstacle_position[1]] = -100
        # self.reward_map[self.state[0], self.state[1]] = 100  # Set the starting column to 0
```

```
        # self.reward_map[ROWS-1,0] = 100
        # self.reward_map[-1, :] = 100  # Set the bottom row to 100

    def reset_reward_map(self):
        self.initialize_reward_map()
        self.reward_map[ROWS-1,COLUMNS-1] = -1  # Mark the starting position

    def step(self, action):
        # Initialize done as False
        done = False
        reward = 0.


        # Update state based on action with x, y format
        delta = ACTION_MAPPINGS.get(action, (0, 0))
        new_state = np.array([self.state[0] + delta[0], self.state[1] + delta[1]])
        # if np.array_equal(new_state, prev_state):
        #     reward = -10

        # Check and handle boundary conditions
        if new_state[0] < 0 or new_state[0] >= ROWS or new_state[1] < 0 or new_state[1] >=
COLUMNS:
            reward = -100
            done = True
        else:
            # Update the current state and the previous state
            self.state = new_state                  # Update the current state to new state

            self.step_counter += 1
            reward += self.calculate_reward()
            done = self.is_done()

            y = self.state[0]
            x = self.state[1]
            self.reward_map[y, x] = -1

        return self.state, reward, done, {}

    def calculate_reward(self):
        # Check if the agent has reached the bottom of the grid
        return self.reward_map[self.state[0], self.state[1]]

    def is_done(self):
        # Check if all cells in the grid have been visited or if a step limit is reached
        max_steps = MAX_STEPS  # Example step limit
        count_minus_one = np.sum(self.reward_map == -1)
        count_minus_hundred = np.sum(self.reward_map == -100)
        return count_minus_one == (self.reward_map.size - 1) and count_minus_hundred == 1 or
self.step_counter >= max_steps

class QLearningAgent2:
    def __init__(self, env, learning_rate=0.2, discount_factor=0.9, epsilon=0.1,
gasoline_capacity=1000, wheat_capacity=100):
        self.env = env
```

```
        self.gasoline = gasoline_capacity  # Initial gasoline level
        self.wheat = 0  # Initial wheat level
        self.gasoline_capacity = gasoline_capacity
        self.wheat_capacity = wheat_capacity

        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.q_table = np.zeros((ROWS, COLUMNS, env.action_space.n))

    def choose_action(self, prev_state, state, neg_reward):

        def get_new_state(action):
            delta = ACTION_MAPPINGS.get(action, (0, 0))
            return np.array([state[0] + delta[0], state[1] + delta[1]])

        def is_valid_action(action):
            new_state = get_new_state(action)
            return not np.array_equal(new_state, prev_state)

        def find_second_best_action(q_values):
            # Copy to avoid modifying the original array
            temp_q_values = np.copy(q_values)

            # Find the index of the best action
            best_action = np.argmax(temp_q_values)

            # Mask the best action by setting its value to negative infinity
            temp_q_values[best_action] = -np.inf

            # Find the second best action
            second_best_action = np.argmax(temp_q_values)

            return second_best_action

        action = None
        state_index = (state[0], state[1])
        if neg_reward > 3:
            # Implement logic to find the nearest positive reward
            action = self.find_nearest_positive_reward_action(state)
        elif np.random.uniform(0, 1) < self.epsilon:
            action = self.env.action_space.sample()  # Explore: random action
        else:
            action = np.argmax(self.q_table[state_index])  # Exploit: best known action

        if not is_valid_action(action):
            action = find_second_best_action(self.q_table[state_index])



        # Additional check or fallback strategy if needed
        if not is_valid_action(action):
            # Implement fallback strategy, e.g., select a random action
            action = self.env.action_space.sample()
```

```
        return action

    def find_nearest_positive_reward_action(self, state):
        min_distance = float('inf')
        best_action = None

        # Search the grid for the nearest positive reward
        for y in range(ROWS):
            for x in range(COLUMNS):
                if self.env.reward_map[y, x] > 0:  # Check for positive reward
                    distance = abs(state[0] - y) + abs(state[1] - x)
                    if distance < min_distance:
                        min_distance = distance
                        best_action = self.determine_action_to_reward(state, (y, x))

        return best_action

    def determine_action_to_reward(self, current_state, reward_state):
        dy = reward_state[0] - current_state[0]
        dx = reward_state[1] - current_state[1]

        if abs(dy) > abs(dx):
            return 1 if dy > 0 else 0  # Move down (1) or up (0) based on the y-difference
        else:
            return 3 if dx > 0 else 2  # Move right (3) or left (2) based on the x-difference

    def learn(self, state, action, reward, next_state):
        state_index = (state[0], state[1])
        next_state_index = (next_state[0], next_state[1])
        # Update rule for Q-learning
        best_next_action = np.argmax(self.q_table[next_state_index])
        td_target = reward + self.discount_factor * self.q_table[next_state_index][best_next_action]
        td_error = td_target - self.q_table[state_index][action]
        self.q_table[state_index][action] += self.learning_rate * td_error


def train_agent(env, agent, episodes):
    best_total_reward = -float('inf')
    best_path = []
    best_wheat_collected = []  # List to keep track of wheat collection

    for episode in range(episodes):
        state = env.reset()
        prev_state = None
        current_path = [state]
        wheat_collected = []  # List for the current episode
        done = False
        total_reward = 0
        neg_reward = 0

        while not done:
            action = agent.choose_action(prev_state, state, neg_reward)
            next_state, reward, done, _ = env.step(action)
```

```
            agent.learn(state, action, reward, next_state)

            total_reward += reward
            prev_state = state
            state = next_state
            current_path.append(state)

            if reward == 1:  # Check if wheat is collected
                wheat_collected.append(True)
            else:
                wheat_collected.append(False)

            if reward < 0:
                neg_reward += 1
            else:
                neg_reward = 0

        if total_reward > best_total_reward:
            best_total_reward = total_reward
            best_path = current_path
            best_wheat_collected = wheat_collected  # Update the best wheat collection list

    return best_path, best_wheat_collected

# Train the agent
env1 = GridEnv1()
obstacle_position1 = env1.obstacle_position
agent1 = QLearningAgent1(env1)
best_path1, wheat_collected1 = train_agent(env1, agent1, episodes=EPISODES)

env2 = GridEnv2()
obstacle_position2 = env2.obstacle_position
agent2 = QLearningAgent2(env2)
best_path2, wheat_collected2 = train_agent(env2, agent2, episodes=EPISODES)

best_path1_reformat = [element for element in best_path1]
best_path2_reformat = [element for element in best_path2]

join_paths = [best_path1_reformat, best_path2_reformat]

path_with_obstacle = [join_paths, obstacle_position1, obstacle_position2]

def convert_arrays(element):
    if isinstance(element, np.ndarray):
        # Convert NumPy arrays to Python lists
        return element.tolist()
    elif isinstance(element, list):
        # Recursively apply this conversion to each element in the list
        return [convert_arrays(sub_element) for sub_element in element]
    else:
        # If it's neither a NumPy array nor a list, return the element as is
        return element

path_converted = convert_arrays(path_with_obstacle)
```

```
# print(join_paths)
print(path_converted)
```

**Plan de trabajo**

Actividades Pendientes

1. Mejora de gráficos en Unity
    - Tiempo estimado: 2 días
2. Aumentar complejidad/efectividad del algoritmo
    - Tiempo estimado: 2 días
3. Probar que no haya errores en la simulación
    - Tiempo estimado: 1 día
4. Asegurar que la interacción entre los agentes sea correcta
    - Tiempo estimado: 1 día

Actividades planeadas para la primera revisión

1. Utilizar Q-Learning para mejorar la solución inicial

Tiempo estimado: 3 días

2. Creación del ambiente en Unity

Tiempo estimado: 3 días

3. Importar la lógica de Python a Unity

Tiempo estimado: 4 días

4. Asegurar que la interacción entre los agentes sea correcta

Tiempo estimado: 2 días

5. Probar que no haya errores en la simulación

Tiempo estimado: 1 día

Actividades Completadas

1. Identificación de agentes
    - Tiempo estimado: 1 día
    - Tiempo completado: 1 día
2. Planeación de la solución
    - Tiempo estimado: 2 días
    - Tiempo completado: 2 días

3. Creación de primer algoritmo para la solución del reto
    - Tiempo estimado: 3 días
    - Tiempo completado: 2 días
4. Creación del ambiente en Unity
    - Tiempo estimado: 8 días
    - Tiempo completado: 10 días
5. Conexión entre Python y Unity
    - Tiempo estimado: 3 días
    - Tiempo completado: 4 días

**Aprendizaje adquirido**

Como equipo tuvimos la oportunidad de aprender sobre el modelado de sistemas multiagentes, al igual que de su funcionamiento. Al trabajar en este proyecto también mejoramos nuestro conocimiento de fundamentos matemáticos para gráficas, el cuál nos fue de utilidad para generar el modelado en Unity. También tuvimos la oportunidad de aprender más de machine learning y aplicarlo para intentar optimizar un proceso real, al igual que ver cómo es el desarrollo profesional de software en base a lo que nos pide un socio.

<u>Alonso</u>

Aprender de los sistemas multiagentes me ayudó a pensar más en las interacciones entre agentes en un sistema, ya que el comportamiento de un agente en un ambiente puede cambiar el comportamiento de otros agentes de formas inesperadas. Me pareció muy interesante poder utilizar machine learning para resolver problemas reales y ver cómo se comporta el modelo después de entrenarlo. Siento que lo que aprendí en esta materia será importante para mi carrera.

<u>Ernesto</u>

La perspectiva única que se obtiene a base de trabajar en un sistema de múltiples agentes ha cambiado mi forma de solucionar problemas al abrirse nuevas puertas de conocimiento. Comprender el paradigma de resolución de problemas teniendo múltiples agentes en un ambiente que interactúen con este y eso traiga cambio a lo que se trabaje es toda una fuente de conocimiento que al utilizar herramientas como

agent.py, unity, javascript y demás se puede dar soluciones que se implementen igual en la vida real

Marco

El trabajar en el desarrollo del algoritmo me ha ayudado a obtener un mayor conocimiento sobre el funcionamiento de Machine Learning y sus usos aplicados en la vida real; el poder ver los efectos directamente en la simulación ha sido una experiencia de aprendizaje sumamente valiosa para refinar mis habilidades de programación e implementación de algoritmos.

Sergio

La modelación de agentes para la realización de una simulación definitivamente es una forma nueva que no conocía antes de atacar un problema. En nuestro caso teníamos que ser capaces de simular un entorno en el que a su vez pudiéramos resolver una eficiente manera de recolectar el trigo. A lo largo de este proyecto se fue posible poner en práctica lo aprendido en la teoría en cuanto a machine learning y modelación de estructuras de agentes para nuestra simulación, y soy de los que piensan que la teoría se refuerza mucho mejor en la práctica, por lo que fui nutritivo aplicar la teoría a la práctica a través de este proyecto.

**Repositorio en Github**

https://github.com/marcopod/multiagentes-TC2008B

**Modelo en Unity**

https://drive.google.com/file/d/1Ztlefn5wrn4GBQIyJiskUvfILaKMnFsJ/view?usp=sharing

**Video del Proyecto**

https://youtu.be/EGTTat3tlP8