



# Tecnológico de Monterrey

**“Revisión de avance 3”**

**Instituto tecnológico y de estudios superiores de Monterrey**  
**Campus Monterrey**

**TC2008B**

**Grupo 302**

**Modelación de sistemas multiagentes con gráficas computacionales**

**Profesores:**

Raul V. Ramirez Velarde

Luis Alberto Muñoz Ubando

**Equipo 3:**

Ainhize Legarreta García	A01762291
Enrique Uzquiano Puras	A01762083
Christopher Pedraza Pohlenz	A01177767
David Cavazos Wolberg	A01721909
Fausto Pacheco Martínez	A01412004

## ÍNDICE

<b>1. DESCRIPCIÓN DEL RETO</b>	<b>3</b>
<b>2. AGENTES INVOLUCRADOS</b>	<b>3</b>
Diagramas de clase	4
Diagramas de protocolos de interacción	5
Cosechadora	5
Tractor	5
Campo	6
<b>3. IMPLEMENTACIÓN DE Q-LEARNING</b>	<b>6</b>
Código (Github)	7
<b>4. UNITY</b>	<b>20</b>
Scripts (Github)	21
AgentController.cs	21
ColorChange.cs	25
MainThreadDispatcher.cs.	26
PathDisplay.cs	27
PlaneGridGenerator.cs	28
<b>5. PLAN DE TRABAJO Y APRENDIZAJES ADQUIRIDOS</b>	<b>29</b>
<b>6. REPOSITORIO DE GITHUB</b>	<b>30</b>

## **1. DESCRIPCIÓN DEL RETO**

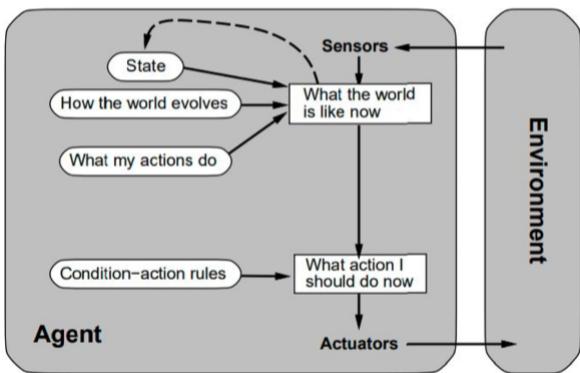
El reto que se nos presenta está relacionado con el entorno del campo y la agricultura, donde se nos dice que “*Optimizar las operaciones agrícolas a través de un sistema inteligente no sólo mejora la eficiencia en la producción de alimentos, sino que también puede reducir costos, lo que podría traducirse en precios más bajos para los consumidores. Además, al tener una logística más eficiente, disminuimos la congestión en las carreteras, lo que beneficia a todos los que viven y trabajan en áreas urbanas.*”. Por lo anteriormente mencionado, el equipo ha desarrollado la propuesta de crear una simulación de un sistema multiagente donde se tendrá un campo de dimensiones variables, donde una serie de tractores y cosechadoras interactúan entre sí, buscando minimizar el tiempo de cosecha (lo que se traduce a la reducción de costos) buscando la ruta más óptima.

## **2. AGENTES INVOLUCRADOS**

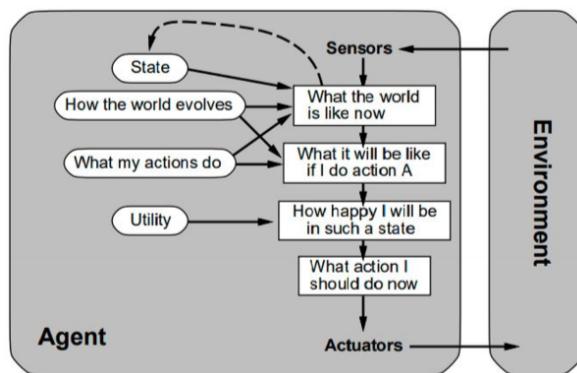
El sistema multiagente consta de 3 agentes: cosechadora, tractor y celda de campo. El agente “cosechadora” se encargará de ir cosechando el campo hasta llenar su capacidad máxima. Una vez que su capacidad se llena, un agente “tractor” va en su dirección para descargar el contenido de la cosechadora. Luego de descargar el contenido, el tractor vuelve a su posición de reposo esperando a que otra cosechadora se llene para volver a descargarla. Asimismo, una vez que la cosechadora ha sido descargada, continúa cosechando las celdas de campo. Los agentes “celda campo” se encargan de llevar registro de las partes del campo que ya han sido cosechadas y la densidad de trigo que hay en cada una, para que cuando una cosechadora pase sobre ellas, estas le transmitan la densidad que está cosechando.

Por el momento, se tiene 2 tipos de agentes: Agentes reflexivos con estados, y Agentes basados en utilidad. Siendo la celda de campo y el tractor el reflexivo con estados, y la cosechadora basada en utilidad. Esto se debe a que tanto el tractor como la celda cuentan con estados que cambian su comportamiento, y reaccionan a cambios del ambiente, sin embargo, no hacen sus acciones con un objetivo o para conseguir una utilidad. En cambio, la cosechadora tiene el objetivo de cosechar todo el campo (objetivo) en el menor tiempo (utilidad). A continuación dos diagrama describiendo a los agentes:

### Reflex agents with state



### Utility-based agents



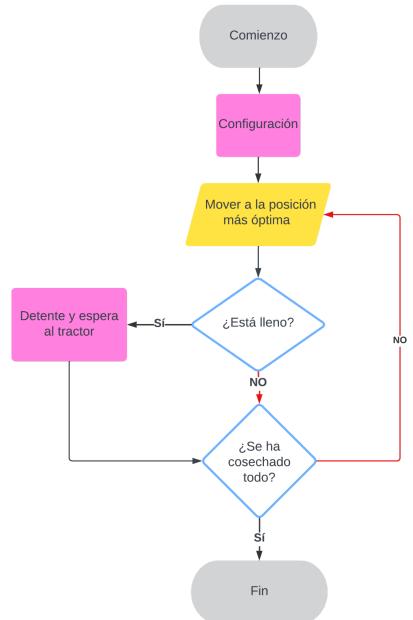
Además, los agentes interactúan entre sí por medio del cambio de estados. En el caso de la celda campo, inicialmente comienza con el estado de isCosechado como falso, significando que cuando una cosechadora pase por encima, podrá cosechar la densidad de esa celda. Tras esto, su estado cambiaría a verdadero para que no pueda ser cosechada otra vez. En el caso de la cosechadora, cuenta con 3 estados: cosechando, llena, y esperando. Cuando tiene todavía capacidad, su estado es cosechando, lo que le permite seguir moviéndose y cosechando las celdas por las que pase. Sin embargo, una vez que llega a su capacidad máxima, esta cambia al estado de llena. Aquí es donde se da la interacción con los tractores. Estos en todo momento buscan alguna cosechadora con estado lleno, mientras tanto, permanecen en su estado de reposo. Una vez que detectan una cosechadora llena, cambian el estado de la cosechadora a esperando, su estado propio a movimiento, y se empiezan a dirigir a la cosechadora para descargarla. El cambio de estado a esperando es una manera de decirle a otros tractores que esa cosechadora ya tiene un tractor asignado para ser descargada, por lo que no van a ir múltiples tractores a descargarla.

### Diagramas de clase

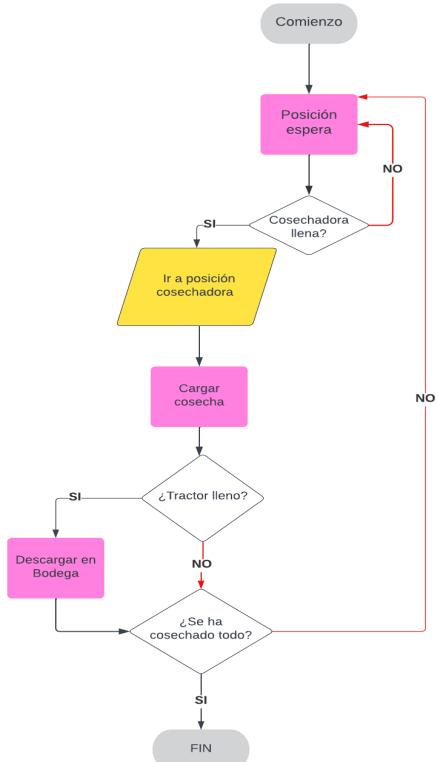
CeldaCampo	Cosechadora	Tractor	FieldModel
isCosechado: Bool densidad: int identificador: String  setup cosechar	velocity: int identificador: String capacidad_max: int capacidad: int estado: String space neighbors pos  setup setup_pos update_velocity update_position cosechar	velocity: int identificador: String target_position: npArray moving: Bool space neighbors pos  setup move setup_pos update_velocity update_position	space cosechadoras tractors celdas_campo  setup send_positions ws_handler_inner ws_handler run_simulation_with_ws

## Diagramas de protocolos de interacción

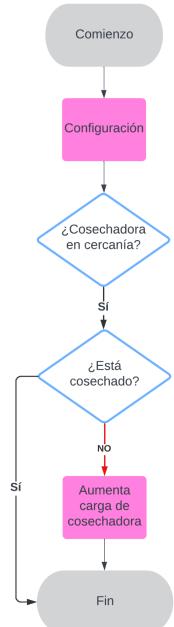
### Cosechadora



### Tractor



## Campo



### 3. IMPLEMENTACIÓN DE Q-LEARNING

El algoritmo Q-learning es una técnica fundamental en el campo del aprendizaje por refuerzo dentro del ámbito de la inteligencia artificial. Se utiliza para que una inteligencia artificial (IA) aprenda a tomar decisiones óptimas en un entorno basado en la teoría de recompensas y acciones. Nos hemos decidido por este algoritmo para optimizar la toma de decisiones de la cosechadora y así poder reducir el tiempo de cosecha, se le ha asignado una recompensa a cada una de las posibles acciones que el agente puede tomar siendo estas recompensas peores cuando menos deseable sea esa acción, obteniendo recompensas desde -100 en el caso menos deseable a -1 en el caso más deseable. El algoritmo se ejecutará una serie de episodios para poder mejorar su ruta y su toma de decisiones obteniendo así un Q value cada vez menor y una mejor solución.

Celda fuera del grid – reward = -100

Chocar con otros objetos – reward = -100

Campo opuesto – reward = -50

Celda cosechada – reward = -30

Celda sin cosechar – reward = -1

La implementación ha funcionado tal y como esperábamos, realizando la cantidad de episodios que se indique en los que va mejorando obteniendo un resultado final óptimo para

la recogida de la cosecha. Los valores de los rewards pueden ser cambiados en un futuro en caso de detectar fallas en el funcionamiento o posibles mejoras.

## Código ([Github](#))

```
Python
import tracemalloc
import asyncio
import websockets
import agentpy as ap
import numpy as np
import json
import random
from copy import deepcopy

# Inicia la herramienta de tracemalloc para el seguimiento de la asignación
# de memoria
tracemalloc.start()

# Función para normalizar un vector
def normalize(v):
    norm = np.linalg.norm(v)
    if norm == 0:
        return v
    return v / norm

# Clase que representa una celda en el campo
class CeldaCampo(ap.Agent):
    def setup(self):
        self.isCosechado = False
        self.densidad = self.p.densidad
        self.identificador = "celda"
        self.pertenencia = None

    def setup_pertenencia(self, id_cosechadora):
        self.pertenencia = id_cosechadora

    def cosechar(self):
        if not self.isCosechado:
            self.isCosechado = True
            return self.densidad
        else:
            return 0

# Clase que representa una cosechadora en el campo
class Cosechadora(ap.Agent):
    def setup(self):
```

```

        self.velocity = 1.0
        self.identificador = "cosechadora"
        self.capacidad_max = self.p.capacidad_max
        self.capacidad = 0
        self.estado = "cosechando" # cosechando / lleno / esperando
        self.moving = False

    def move(self, direction):
        reward = 0
        done = False

        if self.moving:
            # Calcular el vector de dirección hacia la posición objetivo
            direccion = self.target_position - self.pos
            distancia = np.linalg.norm(direccion)
            direccion = normalize(direccion)

            # Moverse hacia la posición objetivo
            if distancia >= self.velocity:
                # self.space.move_to(self, self.target_position)
                self.space.move_by(self, direccion * self.velocity)

            else:
                if (
                    self.target_position[0] > self.p.dimensiones_campo - 1
                    or self.target_position[0] < 0
                    or self.target_position[1] > self.p.dimensiones_campo - 1
                    or self.target_position[1] < 0
                ):
                    print("\t\tOUT OF BOUNDS")
                    reward = self.p.rewards_values["out_of_bounds"]
                    done = True
                    self.moving = False
                    return reward, done

        for nb in self.neighbors(self, distance=self.p.harvest_radius):
            if nb.identificador == "celda":
                if nb.pertenencia != self.id:
                    print("\t\tCELDA DE OTRA COSECHADORA")
                    reward = self.p.rewards_values["celda_otra"]
                    done = False
                    break
                elif nb.isCosechado:
                    print("\t\tCELDA COSECHADA")
                    reward = self.p.rewards_values["celda_cosechada"]
                    done = False
                    break
            # Si celda no ha sido cosechada

```

```

        elif not nb.isCosechado:
            print("\t\tNORMAL")
            reward = self.p.rewards_values["normal"]
            done = False
            break
        elif nb.identificador == "cosechadora":
            print("\t\tCOLISION")
            reward = self.p.rewards_values["colision"]
            done = True
            break

    # Si está en el objetivo dejar de moverse
    self.moving = False
else:
    if self.estado == "cosechando":
        x, y = self.pos[0], self.pos[1]
        if direction == "up":
            # print("Up")
            # self.target_position = np.array([x, y + 1])
            self.target_position = np.ndarray(
                (2,), buffer=np.array([x, y + 1]), dtype=int
            )
        elif direction == "down":
            # print("Down")
            # self.target_position = np.array([x, y - 1])
            self.target_position = np.ndarray(
                (2,), buffer=np.array([x, y - 1]), dtype=int
            )
        elif direction == "left":
            # print("Left")
            # self.target_position = np.array([x - 1, y])
            self.target_position = np.ndarray(
                (2,), buffer=np.array([x - 1, y]), dtype=int
            )
        elif direction == "right":
            # print("Right")
            # self.target_position = np.array([x + 1, y])
            self.target_position = np.ndarray(
                (2,), buffer=np.array([x + 1, y]), dtype=int
            )
        self.moving = True

    if (
        self.target_position[0] > self.p.dimensiones_campo - 1
        or self.target_position[0] < 0
        or self.target_position[1] > self.p.dimensiones_campo - 1
        or self.target_position[1] < 0
    ):

```

```

        print("\t\tOUT OF BOUNDS REWARD")
        reward = self.p.rewards_values[ "out_of_bounds" ]
        done = True
        self.moving = False
        return reward, done

    return reward, done

def setup_pos(self, space):
    self.space = space
    self.neighbors = space.neighbors
    self.pos = space.positions[self]
    self.pos_inicial = deepcopy(self.pos)

def cosechar(self):
    for nb in self.neighbors(self, distance=self.p.harvest_radius):
        if nb.identificador == "celda" and not nb.isCosechado:
            self.capacidad += nb.cosechar()

        if (
            self.capacidad + self.p.densidad > self.capacidad_max
            and self.estado == "cosechando"
        ):
            self.velocity = 0.0
            self.estado = "lleno"

# Clase que representa un tractor en el campo
class Tractor(ap.Agent):
    def setup(self):
        self.velocity = 1.5
        self.identificador = "tractor"
        self.target_position = np.ndarray((2,), buffer=np.array([0, 0]), dtype=int)
        self.moving = False

    def move(self):
        if self.moving:
            print(
                "YA ME ESTOY MOVIENDOOO. POS:",
                self.pos,
                "TARGET:",
                self.target_position,
            )
            # Calcular el vector de dirección hacia la posición objetivo
            direccion = self.target_position - self.pos
            distancia = np.linalg.norm(direccion)
            direccion = normalize(direccion)

        # Moverse hacia la posición objetivo

```

```

if distancia >= self.velocity:
    print(
        "MOVE BY",
        "DIRECCION:",
        direccion,
        "VELOCIDAD:",
        self.velocity,
        "RESULTADO:",
        direccion * self.velocity,
    )
    self.space.move_by(self, direccion * self.velocity)
else:
    # Si está cerca del objetivo, ajustar a la posición objetivo y dejar
    de moverse
    self.moving = False
    for nb in self.neighbors(self, distance=self.p.tractor_radius):
        if nb.identificador == "cosechadora" and nb.estado == "esperando":
            nb.capacidad = 0
            nb.estado = "cosechando"
            nb.velocity = 1.0
            self.target_position = self.pos_inicial
            self.moving = True

    else:
        for nb in self.neighbors(self, distance=self.p.dimensiones_campo**2):
            if nb.identificador == "cosechadora" and nb.estado == "lleno":
                self.target_position = nb.pos
                self.moving = True
                nb.estado = "esperando"
                break

def setup_pos(self, space):
    self.space = space
    self.neighbors = space.neighbors
    self.pos = space.positions[self]
    self.pos_inicial = deepcopy(self.pos)

# Clase principal que representa el modelo del campo
class FieldModel(ap.Model):
    def setup(self):
        self.space = ap.Space(self, shape=[self.p.size] * self.p.ndim)
        self.cosechadoras = ap.AgentList(
            self, self.p.cosechadora_population, Cosechadora
        )
        self.tractors = ap.AgentList(self, self.p.tractor_population, Tractor)

    # Dividir el campo en la cantidad de cosechadoras

```

```

size_segment = self.p.dimensiones_campo // self.p.cosechadora_population
positions = []
for i in range(self.p.cosechadora_population):
    positions.append(
        np.ndarray((2,), buffer=np.array([i * size_segment, 0]), dtype=int)
    )

self.space.add_agents(
    self.cosechadoras,
    positions=positions,
)
self.cosechadoras.setup_pos(self.space)

# Crear celdas_campo sin agregarlas al espacio aún
self.celdas_campo = ap.AgentList(
    self, self.p.dimensiones_campo**2, CeldaCampo
)

for i in range(self.p.tractor_population):
    self.space.positions[self.tractors[i]] = np.ndarray(
        (2,), buffer=np.array([0, 0]), dtype=int
    )
self.tractors.setup_pos(self.space)

# Asignar manualmente posiciones a cada celda_campo
grid_size = self.p.dimensiones_campo
contador = size_segment
cosechadora_index = 0
for x in range(grid_size):
    if contador == 0 and cosechadora_index < self.p.cosechadora_population - 1:
        contador = size_segment
        cosechadora_index += 1
    for y in range(grid_size):
        index = x * grid_size + y
        celda = self.celdas_campo[index]
        self.space.positions[celda] = (x, y)

    # Asignar pertenencia a cada celda de campo con respecto al segmento
    # en el que este
    celda.setup_pertenencia(self.cosechadoras[cosechadora_index].id)
    print(f"({x},{y}) -> {cosechadora_index}")

    contador -= 1
    self.cosechadoras.cosechar()

# Envía las posiciones de los agentes a través de WebSocket
async def send_positions(self, websocket):
    positions_cosechadoras = {

```

```

        f"Cosechadora_{str(agent)}": agent.pos.tolist()
        for agent in self.cosechadoras
    }
    capacidad_cosechadoras = {
        f"Capacidad_{str(agent)}": [float(agent.capacidad)]
        for agent in self.cosechadoras
    }
    positions_tractors = {
        f"Tractor_{str(agent)}": agent.pos.tolist() for agent in
self.tractors
    }
    positions = {
        **positions_cosechadoras,
        **capacidad_cosechadoras,
        **positions_tractors,
    }
    await websocket.send(json.dumps(positions))

def egreedy_policy(self, q_values, state, epsilon=0.1):
    # Se verifica si el valor epsilon es mayor a un número aleatorio
entre 0 y 1
    # Si es verdadero, se elige una acción aleatoria
    if np.random.random() < epsilon:
        # print("USO RANDOM")
        return np.random.choice(4)
    # Sino, se elige la mejor acción
    else:
        # print("USO QVALUES")
        return np.argmax(q_values[state])

def reset(self):
    print("\n*****\nRESET\n*****\n")
    self.cosechadoras.setup()
    self.tractors.setup()

    for cosechadora in self.cosechadoras:
        moving = True
        while moving:
            direccion = cosechadora.pos_inicial - cosechadora.pos
            distancia = np.linalg.norm(direccion)
            direccion = normalize(direccion)

            # Moverse hacia la posición objetivo
            if distancia >= cosechadora.velocity:
                # self.space.move_to(self, self.target_position)
                cosechadora.space.move_by(
                    cosechadora, direccion * cosechadora.velocity
                )

```

```

        else:
            moving = False
            # self.space.move_to(cosechadora, cosechadora.pos_inicial)
        for tractor in self.tractors:
            moving = True
            while moving:
                print(f"Pos de tractor {tractor.id}: {tractor.pos}")
                direccion = tractor.pos_inicial - tractor.pos
                distancia = np.linalg.norm(direccion)
                direccion = normalize(direccion)

                # Moverse hacia la posición objetivo
                if distancia >= tractor.velocity:
                    # self.space.move_to(self, self.target_position)
                    tractor.space.move_by(tractor, direccion *
tractor.velocity)
                else:
                    moving = False
                    # self.space.move_to(tractor, tractor.pos_inicial)
        for celda in self.celdas_campo:
            celda.isCosechado = False

        self.cosechadoras.cosechar()

    def q_learning(self):
        print("\n*****\nQLEARNING\n*****\n")
        num_states = self.p.dimensiones_campo**2
        num_actions = 4
        q_values = np.zeros((num_states, num_actions))
        ep_rewards = []
        direcciones = ["up", "down", "left", "right"]
        done = False
        reward_sum = 0

        for i in range(self.p.num_episodes):
            self.reset()
            done = False
            reward_sum = 0
            print(f"Episodio {i}")
            while not done:
                for cosechadora in self.cosechadoras:
                    if cosechadora.estado == "lleno":
                        continue

                state = (
                    cosechadora.pos[1] * self.p.dimensiones_campo
                    + cosechadora.pos[0]
                )

```

```

        action = self.egreedy_policy(
            q_values, state, self.p.exploration_rate_upper
        )

        reward = 0
        while reward == 0 and cosechadora.velocity != 0.0:
            print("CICLO000")
            reward, done = cosechadora.move(direcciones[action])

        if cosechadora.estado == "lleno":
            continue

        next_state = (
            cosechadora.pos[1] * self.p.dimensiones_campo
            + cosechadora.pos[0]
        )

        if cosechadora.pos[0] % 2 == 0 and action == 0:
            reward += self.p.rewards_values["up"]
        elif cosechadora.pos[0] % 2 == 1 and action == 1:
            reward += self.p.rewards_values["down"]

        if (
            cosechadora.pos[1] == self.p.dimensiones_campo - 1
            or cosechadora.pos[1] == 0
        ) and action == 3:
            reward += self.p.rewards_values["sides"]

        reward_sum += reward

        # La ecuación de Bellman se define como:
        # Q(s,a) = r + gamma * max(Q(s',a')) - Q(s,a)
        if done:
            td_target = reward
        else:
            td_target = reward + self.p.gamma * np.max(q_values[next_state])
            td_error = td_target - q_values[state][action]
        # Actualiza el valor Q para el estado y acción actuales con el
        valor
        # de la ecuación de Bellman.
        q_values[state][action] += self.p.learning_rate * td_error

        self.cosechadoras.cosechar()
        self.tractors.move()

        if self.p.exploration_rate_upper > self.p.exploration_rate_lower:
            self.p.exploration_rate_upper -= self.p.exploration_rate_decrease
    
```

```

        ep_rewards.append(reward_sum)

    print(f"EPISODE REWARDS: {ep_rewards}, \nQ_VALUES: \n{q_values}")

    return ep_rewards, q_values

# Manejador de WebSocket para la simulación
async def ws_handler_inner(self, websocket, q_values):
    print("\n*****\nSIMULACION\n*****\n")
try:
    self.reset()
    done = False
    direcciones = [ "up", "down", "left", "right"]

    while not done:
        for cosechadora in self.cosechadoras:
            state = (
                cosechadora.pos[1] * self.p.dimensiones_campo
                + cosechadora.pos[0]
            )
            action = self.egreedy_policy(q_values, state, 0.0)

            reward = 0
            while reward == 0:
                reward, done = self.cosechadoras.move(direcciones[action])[0]
            print(direcciones[action])

            self.cosechadoras.cosechar()
            self.tractors.move()
            await self.send_positions(websocket)
            await asyncio.sleep(0.1) # Ajustar la frecuencia según sea necesario
except websockets.exceptions.ConnectionClosed:
    pass

# Manejador principal de WebSocket
async def ws_handler(self, websocket, path, q_values):
    await self.ws_handler_inner(websocket, q_values)

# Ejecuta la simulación y WebSocket
async def run_simulation_with_websocket(self, q_values):
    loop = asyncio.get_running_loop()

    # Habilitar tracemalloc dentro del bucle de eventos
    tracemalloc.start()

    # Inicia el servidor WebSocket
    server = await websockets.serve(
        lambda ws, path: self.ws_handler(ws, path, q_values),
        "localhost",

```

```

    8765,
)

try:
    for _ in range(self.p.steps):
        self.step()
        await asyncio.sleep(0.1)
finally:
    # Cierre del servidor WebSocket
    server.close()
    await server.wait_closed()

async def ws_handler_inner_qlearning(self, websocket):
    try:
        print("\n*****\nQLEARNING\n*****\n")
        num_states = self.p.dimensiones_campo**2
        num_actions = 4
        q_values = np.zeros((num_states, num_actions))
        ep_rewards = []
        direcciones = [ "up", "down", "left", "right" ]
        done = False
        reward_sum = 0

        for i in range(self.p.num_episodes):
            self.reset()
            done = False
            reward_sum = 0
            print(f"Episodio {i}")
            while not done:
                for cosechadora in self.cosechadoras:
                    if cosechadora.estado == "lleno":
                        continue

                state = (
                    cosechadora.pos[1] * self.p.dimensiones_campo
                    + cosechadora.pos[0]
                )

                action = self.egreedy_policy(
                    q_values, state, self.p.exploration_rate_upper
                )

                reward = 0
                while reward == 0 and cosechadora.velocity != 0.0:
                    print("CICLOOOO")
                    reward, done = cosechadora.move(direcciones[action])

                if cosechadora.estado == "lleno":
                    continue

```

```

        next_state = (
            cosechadora.pos[1] * self.p.dimensiones_campo
            + cosechadora.pos[0]
        )

        if cosechadora.pos[0] % 2 == 0 and action == 0:
            reward += self.p.rewards_values["up"]
        elif cosechadora.pos[0] % 2 == 1 and action == 1:
            reward += self.p.rewards_values["down"]

        if (
            cosechadora.pos[1] == self.p.dimensiones_campo - 1
            or cosechadora.pos[1] == 0
        ) and action == 3:
            reward += self.p.rewards_values["sides"]

        reward_sum += reward

        # La ecuación de Bellman se define como:
        # Q(s,a) = r + gamma * max(Q(s',a')) - Q(s,a)
        if done:
            td_target = reward
        else:
            td_target = reward + self.p.gamma * np.max(
                q_values[next_state]
            )
        td_error = td_target - q_values[state][action]
        # Actualiza el valor Q para el estado y acción actuales con el
valor
        # de la ecuación de Bellman.
        q_values[state][action] += self.p.learning_rate * td_error

        self.cosechadoras.cosechar()
        self.tractors.move()

        await self.send_positions(websocket)
        await asyncio.sleep(1)

        if self.p.exploration_rate_upper > self.p.exploration_rate_lower:
            self.p.exploration_rate_upper -= self.p.exploration_rate_decrease

    ep_rewards.append(reward_sum)

    print(f"EPISODE REWARDS: {ep_rewards}, \nQ_VALUES: \n{q_values}")
    except websockets.exceptions.ConnectionClosed:
        pass

async def ws_handler_qlearning(self, websocket, path):

```

```
        await self.ws_handler_inner_qlearning(websocket)

    async def run_qlearning_with_websocket(self):
        loop = asyncio.get_running_loop()

        # Habilitar tracemalloc dentro del bucle de eventos
        tracemalloc.start()

        # Inicia el servidor WebSocket
        server = await websockets.serve(
            lambda ws, path: self.ws_handler_qlearning(ws, path),
            "localhost",
            8765,
        )

        try:
            for _ in range(self.p.steps):
                self.step()
                await asyncio.sleep(0.1)
        finally:
            # Cierre del servidor WebSocket
            server.close()
            await server.wait_closed()

# Parámetros del modelo en 2D
parameters2D = {
    "size": 50,
    "dimensiones_campo": 50,
    "seed": 123,
    "steps": 1000,
    "ndim": 2,
    "densidad": 10,
    "capacidad_max": 500,
    "cosechadora_population": 3,
    "tractor_population": 5,
    "inner_radius": 1, # 3
    "outer_radius": 3, # 10
    "harvest_radius": 0.2, # 1
    "border_distance": 1, # 10
    "tractor_radius": 2,
    "cohesion_strength": 0.005,
    "separation_strength": 0.1,
    "alignment_strength": 0.3,
    "border_strength": 0.5,
    # QLEARNING
    "exploration_rate_upper": 0.1,
    "exploration_rate_lower": 0.1,
```

```

"exploration_rate_decrease": 0.05,
"num_episodes": 200,
"learning_rate": 0.1, # 0.5
"gamma": 0.9,
"rewards_values": {
    "normal": -1,
    "celda_cosechada": -5,
    "celda_otra": -10,
    "colision": -100,
    "out_of_bounds": -100,
    "up": 4,
    "down": 4,
    "sides": 2,
},
}

# Crea una instancia del modelo y ejecuta la simulación con WebSocket
model = FieldModel(parameters2D)
model.setup()
asyncio.run(model.run_qlearning_with_websocket())

```

## 4. UNITY

Los assets de la cosechadora, el tractor y las celdas de trigo han sido modelados. La conexión entre Unity y AgentPy usando websocket ha sido lograda, es necesario seguir trabajando en el comportamiento de los assets para que realicen un movimiento más realista en la realización de tareas especialmente al girar.



Imagen de la cosechadora sobre el campo sin cosechar

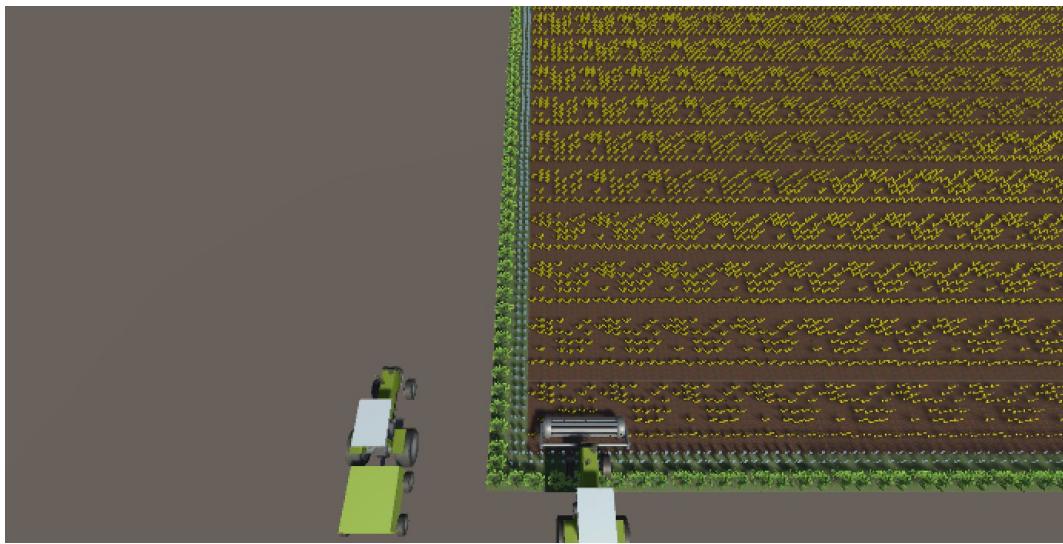


Imagen de la cosechadora junto al tractor al inicio de su recorrido

## Scripts ([Github](#))

### AgentController.cs

```
C/C++  
  
using System;  
using System.Collections;  
using UnityEngine;  
using WebSocketSharp;  
using Newtonsoft.Json;  
using System.Collections.Generic;  
  
public class BoidsController : MonoBehaviour  
{  
    // Prefabs para agentes en Unity  
    public GameObject cosechadoraPrefab;  
    public GameObject tractorPrefab;  
  
    // Conexión WebSocket y diccionario para almacenar agentes  
    private WebSocket ws;  
    private Dictionary<string, GameObject> agents = new Dictionary<string, GameObject>();  
  
    // Método Start que se ejecuta al inicio  
    void Start()  
    {  
        Debug.Log("Conectando a WebSocket...");  
        ws = new WebSocket("ws://localhost:8765");  
        ws.OnMessage += OnMessage;  
        ws.OnError += OnError;  
        ws.OnClose += OnClose;
```

```
agents = new Dictionary<string, GameObject>(); // Inicializa el diccionario

// Intenta conectar
ws.Connect();

// Verifica si la conexión fue exitosa
if (ws.ReadyState == WebSocketState.Open)
{
    // Conexión exitosa, inicia la rutina de actualización
    StartCoroutine(UpdateAgents());
}
else
{
    // Fallo de conexión, registra un error
    Debug.LogError("La conexión WebSocket falló.");
}

// Maneja los mensajes recibidos a través de WebSocket
void OnMessage(object sender, MessageEventArgs e)
{
    try
    {
        // Registra los datos JSON recibidos
        Debug.Log("Datos JSON recibidos: " + e.Data);

        // Parsea los datos JSON usando JSON.NET
        Dictionary<string, List<float>> agentsData;
        try
        {
            agentsData = JsonConvert.DeserializeObject<Dictionary<string,
List<float>>>(e.Data);
        }
        catch (Exception ex)
        {
            Debug.LogError("Error al analizar los datos JSON: " + ex);
            return;
        }

        // Registra el número de agentes recibidos
        Debug.Log($"Número de agentes recibidos: {agentsData.Count}");

        // Recorre los agentes y actualiza las posiciones
        foreach (var entry in agentsData)
        {
            string agentName = entry.Key;
            List<float> positionData = entry.Value;
```

```

    // Extrae la información del tipo (asumiendo que 'Cosechadora' o
    'Tractor' está incluido en agentName)
    string agentType = "";
    if (agentName.StartsWith("Cosechadora"))
    {
        agentType = "Cosechadora";
    }
    else if (agentName.StartsWith("Tractor"))
    {
        agentType = "Tractor";
    }
    else
    {
        agentType = "Campo";
    }

    if (!agentType.Equals("Campo"))
    {
        // Registra información para depuración
        Debug.Log($"Agente: {agentName}, Tipo: {agentType}, Posición:
{positionData[0]}, {positionData[1]}");

        MainThreadDispatcher.Instance.DispatchToMainThread(() =>
        {
            // Verifica si el agente ya está en el diccionario
            if (agents.ContainsKey(agentName))
            {
                // Actualiza la posición del agente existente
                agents[agentName].transform.position = new
                Vector3(positionData[0], 0.0f, positionData[1]);
                Debug.Log($"Actualizando posición del agente existente:
{agentName}");
            }
            else
            {
                if (cosechadoraPrefab != null && tractorPrefab != null)
                {
                    // Instancia un nuevo agente según el tipo
                    GameObject newAgent;
                    if (agentType == "Cosechadora")
                    {
                        newAgent = Instantiate(cosechadoraPrefab, new
                        Vector3(positionData[0], 0.0f, positionData[1]), Quaternion.identity);
                    }
                    else if (agentType == "Tractor")
                    {
                        newAgent = Instantiate(tractorPrefab, new
                        Vector3(positionData[0], 0.0f, positionData[1]), Quaternion.identity);
                    }
                }
            }
        });
    }
}

```



```

        Debug.LogWarning("La conexión WebSocket no está abierta.");
    }

    yield return new WaitForSeconds(0.5f); // Ajusta la frecuencia de
actualizaciones según sea necesario
}
}

// Método que se llama cuando se destruye el objeto
private void OnDestroy()
{
    if (ws != null)
    {
        ws.Close();
    }
}

// Clase para representar datos de un agente
[System.Serializable]
public class AgentData
{
    public List<float> position;

    // Agrega una propiedad para obtener Vector2 a partir de la lista
    public Vector2 GetVector2Position()
    {
        if (position != null && position.Count >= 2)
        {
            return new Vector2(position[0], position[1]);
        }
        return Vector2.zero; // Devuelve un valor predeterminado si la lista no es
válida
    }
}

// Clase para representar datos de varios agentes
[Serializable]
public class AgentsData
{
    public Dictionary<string, List<float>> agents;
}

```

## ColorChange.cs

C/C++  
using System.Collections;

```

using System.Collections.Generic;
using UnityEngine;

public class ColorChange : MonoBehaviour
{
    private Material originalMaterial;
    public Material highlightMaterial; // Assign this in the inspector

    private void Start() {
        originalMaterial = GetComponent<Renderer>().material;
    }

    private void OnTriggerEnter(Collider other) {
        Debug.Log(other.tag);
        if (other.CompareTag("cosechadora")) {
            // Change color to highlightMaterial
            GetComponent<Renderer>().material = highlightMaterial;
        }
    }

    // private void OnTriggerExit(Collider other) {
    //     if (other.CompareTag(tag)) {
    //         // Change color back to the originalMaterial
    //         GetComponent<Renderer>().material = originalMaterial;
    //     }
    // }
}

```

### *MainThreadDispatcher.cs.*

C/C++

```

using System;
using UnityEngine;
using System.Collections.Generic;

public class MainThreadDispatcher : MonoBehaviour
{
    private static MainThreadDispatcher _instance;

    public static MainThreadDispatcher Instance
    {
        get
        {
            if (_instance == null)
            {
                GameObject go = new GameObject("MainThreadDispatcher");
                _instance = go.AddComponent<MainThreadDispatcher>();
            }
            return _instance;
        }
    }
}

```

```

        }
        return _instance;
    }

private readonly Queue<Action> _actions = new Queue<Action>();
private object _lock = new object();

private void Awake()
{
    if (_instance != null && _instance != this)
    {
        Destroy(gameObject);
        return;
    }

    _instance = this;
    DontDestroyOnLoad(gameObject);
}

private void Update()
{
    lock (_lock)
    {
        while (_actions.Count > 0)
        {
            _actions.Dequeue()?.Invoke();
        }
    }
    Debug.Log("MainThreadDispatcher Update");
}

public void DispatchToMainThread(Action action)
{
    lock (_lock)
    {
        _actions.Enqueue(action);
    }
}
}

```

## PathDisplay.cs

C/C++

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```

public class PathDisplay : MonoBehaviour {
    public float targetTime;
    float time;
    public GameObject prefab;

    // Start is called before the first frame update
    void Start() {

    }

    // Update is called once per frame
    void FixedUpdate() {
        time -= Time.deltaTime;

        if (time <= 0.0f) {
            Vector3 position = transform.position;
            Instantiate(prefab, position, Quaternion.identity);
            time = targetTime;
        }
    }
}

```

## PlaneGridGenerator.cs

```

C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlaneGridGenerator : MonoBehaviour {
    public GameObject planePrefab; // Prefab for the plane
    public int gridWidth = 5;    // Number of planes in the X direction
    public int gridHeight = 5;   // Number of planes in the Z direction
    public float spacing = 1.0f; // Spacing between planes

    void Start() {
        GeneratePlaneGrid();
    }

    void GeneratePlaneGrid() {
        for (int i = 0; i < gridWidth; i++) {
            for (int j = 0; j < gridHeight; j++) {
                // Calculate the position for each plane based on the grid and
                // spacing
                Vector3 position = new Vector3(i * spacing, 0, j * spacing) +
transform.position;
            }
        }
    }
}

```

```

    // Instantiate a plane prefab at the calculated position
    GameObject plane = Instantiate(planePrefab, position,
    Quaternion.identity);

    // Set the parent of the instantiated plane to the empty GameObject
    plane.transform.parent = transform;
}
}
}
}

```

## 5. PLAN DE TRABAJO Y APRENDIZAJES ADQUIRIDOS

### Planeación de Actividades

Actividad	Esfuerzo	Fecha estimada	Fecha real	Diferencia	Encargado
Crear/conseguir el asset para la cosechadora	M	11/20/2023	11/20/2023	Se logró a tiempo	David
Crear/conseguir el asset para el tractor	M	11/20/2023	11/20/2023	Se logró a tiempo	David
Crear/conseguir el asset para el trigo	M	11/20/2023	11/20/2023	Se logró a tiempo	David
Integrar Q-Learning en el movimiento de las cosechadoras	M	11/20/2023	11/29/2023	9 días más	TODOS
Modelar una vista en primera persona para los tractores/cosechadoras	L	11/27/2023			Christopher
Llevar un contador del trigo cosechado	XS	11/20/2023			Christopher
Integrar obstáculos en el campo y el comportamiento para evitarlos	L	11/23/2023			Fausto
Conectar Unity con AgentPy usando un websocket	XL	11/14/2023	11/14/2023	Se logró a tiempo	Christopher
Identificar las cosechadoras Héreas y mandar un tractor a descargarlas	M	11/14/2023	11/14/2023	Se logró a tiempo	TODOS
Pasar todos los datos necesarios de la simulación a Unity	M	11/20/2023	11/20/2023	Se logró a tiempo	Christopher

Durante las últimas semanas de trabajo, hemos profundizado en nuestro conocimiento de Unity y en el diseño del sistema multiagente. En el caso de Unity, enfrentamos un desafío al diseñar modelos y luego integrarlos en nuestra simulación inicial, ya que las proporciones eran diferentes. Para superar esto, fue necesario escalar los datos recibidos de la simulación de AgentPy para representarlos con las dimensiones adecuadas en Unity.

En cuanto al sistema multiagente, la implementación de Q-Learning ha sido un desafío, pero a medida que avanzamos prueba a prueba, sentimos que comprendemos mejor su funcionamiento como equipo. De esta manera, estamos logrando realizar los ajustes necesarios y optimizaciones para que nuestro sistema se comporte según lo deseado.

Finalmente, uno de los aprendizajes clave ha sido la importancia de acotar el alcance del proyecto. Con cada iteración, hemos ajustado nuestras metas para poder presentar un producto mínimo viable de alta calidad. Este enfoque nos ha permitido adaptarnos de manera efectiva y mantenernos en camino hacia el éxito del proyecto.

## **6. REPOSITORIO DE GITHUB**

El siguiente enlace abre el Github del equipo:

[https://github.com/christopher-pedraza/multiagentes\\_john\\_deere](https://github.com/christopher-pedraza/multiagentes_john_deere)