Actividad Integradora

Alumna:

Eliuth Balderas Neri | A01703315

Modelación de sistemas multiagentes con gráficas computacionales (Gpo 301)

Profesores:

Denisse Lizbeth Maldonado Flores

Alejandro Fernández

Pedro Oscar Pérez Murueta

Tecnológico de Monterrey, campus Querétaro

Febrero- Junio 2024

06 de junio del 2024

Evidencia-Integradora-TC2008b
evidencia ubicada en: https://github.com/BalnerEli/Evidencia-Integradora-TC2008b.git

Gráficas
Carpeta "scripts" que contiene todos los archivos utilizados para el videojuego "My little bear game" Un osito quiere llegar a su casa. Sin embargo, para lograrlo, necesita matar a los enemigos, los cuáles son máquinas expendedoras que disparan tenedores. Si logras vencerlos, tienes que enfrentarte con el malvado Mellowy, un bombón enojado que lanza balas sin parar. Una vez que logres vencerlo, ganas y podrás ir tranquilamente a tu casa

Demo:
https://www.youtube.com/watch?si=Vp7SJvolEIkOFxSO&v=cXAmFq6T9kQ&feature=youtu.be
Nota: La primera parte busca demostrar todos los modos de disparo del jefe, mientras que en la segunda parte o intento, ya se gana

Scripts: https://github.com/BalnerEli/Evidencia-Integradora-TC2008b.git

Multiagentes
Link del cuaderno en donde se presentan todas las especificaciones de la entrega:
https://colab.research.google.com/drive/18npRaU9kAiXx1F-GDdMeGgzlDzieBHma?usp=sharing

Código completo:

```
# Importación de las librerías necesarias

# Importamos las clases que se requieren para manejar los agentes (Agent)
y su entorno (Model).
# Cada modelo puede contener múltiples agentes.
from mesa import Agent, Model

# Con ''SimultaneousActivation, hacemos que todos los agentes se activen
''al azar''.
from mesa.time import RandomActivation
import random

# Haremos uso de ''DataCollector'' para obtener información de cada paso
de la simulación.
from mesa.datacollection import DataCollector
```

```
# Haremos uso de ''MultriGrid'' para obtener una cuadrícula
from mesa.space import MultiGrid


# matplotlib lo usaremos crear una animación de cada uno de los pasos del
modelo.
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.animation as animation
plt.rcParams["animation.html"] = "jshtml"

# Importamos los siguientes paquetes para el mejor manejo de valores
numéricos.
import numpy as np
import pandas as pd

from IPython.display import HTML

# Establecer la semilla establecida en situación problema
random.seed(67890)

# Aumentar el límite de tamaño de la animación incrustada
plt.rcParams['animation.embed_limit'] = 50  # Aumentar el límite a 50 MB

# Clase CollectorAgent que representa un robot colector de cajas
class CollectorAgent(Agent):
    def __init__(self, id, model):
        super().__init__(id, model)
        self.carrying_box = False  # Indicador de si el agente lleva una
caja
        self.boxes_moved = 0  # Contador de cajas movidas por el agente

    # Método para el comportamiento del agente en cada paso
    def step(self):
        if self.carrying_box:
            # Si lleva una caja, busca una celda con una o más cajas para
apilarla
            target = self.find_pile()
```

```
            if target:
                self.model.grid.move_agent(self, target)
                self.boxes_moved += 1
                self.model.place_box(target)
                self.carrying_box = False
        else:
            # Si no lleva una caja, busca una celda con cajas para recoger
(menos de 4 cajas)
            target = self.find_box()
            if target:
                self.model.grid.move_agent(self, target)
                self.carrying_box = True
                self.model.remove_box(target)

    # Método para encontrar una caja
    def find_box(self):
        # Buscar cualquier celda con cajas (menos de 4 cajas) esto hace
que sea un poco más eficiente pues solo agarra cajas de pilas menores a 4
cajas
        for neighbor in self.model.grid.get_neighborhood(self.pos,
moore=True, include_center=False):
            if 0 < self.model.box_positions.get(neighbor, 0) < 4:
                return neighbor
        return None

    # Método para encontrar pilas de cajas con menos de 5 cajas
    def find_pile(self):
        # Priorizar pilas de cajas con menos de 5 cajas, buscando las más
llenas primero
        neighbors = self.model.grid.get_neighborhood(self.pos, moore=True,
include_center=False)
        neighbors_with_boxes = [(neighbor,
self.model.box_positions.get(neighbor, 0)) for neighbor in neighbors if
self.model.box_positions.get(neighbor, 0) < 5]
        if neighbors_with_boxes:
            neighbors_with_boxes.sort(key=lambda x: x[1], reverse=True)
            return neighbors_with_boxes[0][0]
        return None

# Clase CollectorBoxModel que representa el grid
```

```
class CollectorBoxModel(Model):
    def __init__(self, num_agents, width, height, num_boxes):
        super(CollectorBoxModel, self).__init__()
        self.num_agents = num_agents
        self.width = width
        self.height = height
        self.num_boxes = num_boxes
        self.grid = MultiGrid(width, height, False)
        self.schedule = RandomActivation(self)
        self.current_step = 0
        self.all_boxes_piled = None
        self.box_positions = self.initialize_boxes()  # Inicializar las
cajas
        self.initialize_agents()  # Inicializar los agentes
        self.datacollector = DataCollector(
            agent_reporters={"Boxes Moved": lambda a: a.boxes_moved}
        )
        assert self.total_boxes() == 200, f"Error en inicialización: el
total de cajas es {self.total_boxes()}, pero debería ser 200."

    # Método para inicializar las cajas de forma aleatoria
    def initialize_boxes(self):
        all_cells = [(x, y) for x in range(self.width) for y in
range(self.height)]
        random.shuffle(all_cells)
        box_positions = {}

        for i in range(min(self.num_boxes, len(all_cells))):
            cell = all_cells[i]
            box_positions[cell] = 1  # Colocar una caja en cada celda

        return box_positions

    # Método para inicializar los agentes
    def initialize_agents(self):
        for i in range(self.num_agents):
            agent = CollectorAgent(i, self)
            self.schedule.add(agent)
            x, y = self.empty_cell()
            self.grid.place_agent(agent, (x, y))
```

```
    # Método para encontrar una celda vacía
    def empty_cell(self):
        while True:
            x = random.randrange(self.width)
            y = random.randrange(self.height)
            if self.grid.is_cell_empty((x, y)):
                return (x, y)

    # Método para verificar si una celda tiene cajas
    def has_box(self, pos):
        return self.box_positions.get(pos, 0) > 0

    # Método para colocar cajas
    def place_box(self, pos):
        if pos in self.box_positions:
            self.box_positions[pos] += 1
        else:
            self.box_positions[pos] = 1

    # Método para cargar cajas
    def remove_box(self, pos):
        if pos in self.box_positions and self.box_positions[pos] > 0:
            self.box_positions[pos] -= 1

    # Método que define el comportamiento del modelo en cada paso, se
comprueba que siempre hay 200 cajas
    def step(self):
        self.schedule.step()
        self.datacollector.collect(self)
        self.current_step += 1
        if self.piled_percentage() == 100.0 and self.all_boxes_piled is
None:
            self.all_boxes_piled = self.current_step
            self.running = False
        if self.current_step % 100 == 0 or self.current_step == 1:
            if self.total_boxes() < 200:
                self.adjust_boxes(add=True)
            elif self.total_boxes() > 200:
                self.adjust_boxes(add=False)
```

```
            assert self.total_boxes() == 200, f"Error: el total de cajas
es {self.total_boxes()}, pero debería ser 200."

    # Método para ajustar el número total de cajas
    def adjust_boxes(self, add):
        if add:
            while self.total_boxes() < 200:
                pos = random.choice(list(self.box_positions.keys()))
                self.place_box(pos)
        else:
            while self.total_boxes() > 200:
                pos = random.choice(list(self.box_positions.keys()))
                if self.box_positions[pos] > 0:
                    self.remove_box(pos)

    # Método para calcular el porcentaje de cajas apiladas en pilas de 5
cajas
    def piled_percentage(self):
        total_boxes = sum(self.box_positions.values())
        stacks_of_five = sum(1 for v in self.box_positions.values() if v
== 5)
        return (stacks_of_five * 5 / total_boxes) * 100

    # Método para obtener la distribución final de las cajas
    def get_final_grid(self):
        final_grid = np.zeros((self.height, self.width))
        for (x, y), count in self.box_positions.items():
            final_grid[y, x] = count
        return final_grid

    # Método para calcular el número total de cajas
    def total_boxes(self):
        return sum(self.box_positions.values())

# Función para animar el modelo
def animate_model(model, steps):
    fig, ax = plt.subplots()
    cmap = plt.get_cmap('viridis', 8)
    def update(frame):
        ax.clear()
```

```
        grid_display = np.zeros((model.grid.width, model.grid.height))
        for (x, y), value in model.box_positions.items():
            grid_display[y][x] = value
        for agent in model.schedule.agents:
            grid_display[agent.pos[1]][agent.pos[0]] = 7 if
agent.carrying_box else 6
        ax.imshow(grid_display, cmap=cmap, vmin=0, vmax=7)
        ax.set_xticks([])
        ax.set_yticks([])
        model.step()

    anim = animation.FuncAnimation(fig, update, frames=steps,
repeat=False)
    plt.close()
    return anim

# Parámetros de la simulación
width, height = 20, 20
num_boxes = 200
num_agents = 5
max_steps = 5000  # Máximo número de pasos

# Ejecutar la simulación y mostrar la animación
model = CollectorBoxModel(num_agents, width, height, num_boxes)
anim = animate_model(model, max_steps)
html_anim = HTML(anim.to_jshtml())
display(html_anim)

# Recopilar y mostrar los resultados de la simulación
if model.all_boxes_piled is not None:
    print(f"Tiempo necesario hasta que todas las cajas están en pilas de
máximo 5 cajas: {model.all_boxes_piled} pasos")
else:
    print("No se logró apilar todas las cajas en pilas de máximo 5 cajas
dentro del tiempo máximo establecido.")

movimientos_totales = sum([agent.boxes_moved for agent in
model.schedule.agents])
print(f"Número de movimientos realizados por todos los robots:
{movimientos_totales}")
```

```
# Verificar el total de cajas
total_cajas = model.total_boxes()
print(f"Total de cajas en el almacén: {total_cajas}")

# Visualizar el estado final del almacén
final_grid = model.get_final_grid()
plt.figure(figsize=(10, 10))
plt.imshow(final_grid, cmap='viridis', vmin=0, vmax=7)  # Ajustar vmin y
vmax a 7
plt.colorbar(label='Número de cajas')
plt.title('Distribución final de las cajas en el almacén')
plt.show()

#Nota
print("en el lado derecho se muestra 7 colores, sin embargo, las cajas son
representadas con color de 0-5, mientras que 6 y 7 son los agentes ")
# ¿Existe una forma de reducir el número de pasos utilizados? Si es así,
¿cuál es la estrategia que se tendría en implementar?
print("Se podría dividir el grid den diferentes zonas y cada una darsela a
los agentes, así evita que los agentes se desplazen mucho")
print("Implementar algoritmos de búsqueda, en donde los agentes sepan en
dónde hay cajas y la condición de as pilas")

# •Descripción detallada de la estrategia y los mecanismos utilizados en
tu solución.
print("Para esta simulación se siguiero los parámetros establecidos en la
situación problema")
print("Sin embargo, se utilizó la priorización de pilas, en donde busca
pilas más largas para poner las cajas y únicamente puede tomar cajas de
pilas de menos de 4 cajas, esto logra que las pilas se estén haciendo y
deshaciendo constantemente")
```