

# Actividad Integradora

Daniel Emilio Fuentes Portaluppi - A01708302

## Introducción

Felicidades! Eres el orgulloso propietario de 5 robots nuevos y un almacén lleno de cajas. El dueño anterior del almacén lo dejó en completo desorden, por lo que depende de tus robots organizar las cajas en algo parecido al orden y convertirlo en un negocio exitoso.

Cada robot está equipado con ruedas omnidireccionales y, por lo tanto, puede conducir en las cuatro direcciones. Pueden recoger cajas en celdas de cuadrícula adyacentes con sus manipuladores, luego llevarlas a otra ubicación e incluso construir pilas de hasta cinco cajas. Todos los robots están equipados con la tecnología de sensores más nueva que les permite recibir datos de sensores de las cuatro celdas adyacentes. Por tanto, es fácil distinguir si un campo está libre, es una pared, contiene una pila de cajas (y cuantas cajas hay en la pila) o está ocupado por otro robot. Los robots también tienen sensores de presión equipados que les indican si llevan una caja en ese momento.

Lamentablemente, tu presupuesto resultó insuficiente para adquirir un software de gestión de agentes múltiples de última generación. Pero eso no debería ser un gran problema... ¿verdad? Tu tarea es enseñar a sus robots cómo ordenar su almacén. La organización de los agentes depende de ti; siempre que todas las cajas terminen en pilas ordenadas de cinco.

### Puntos a considerar

- La semilla para generación de números aleatorios será 67890.
- El almacén es 20x20 celdas.
- Al inicio de la simulación, tu solución deberá colocar 200 cajas repartidas en grupos de 1 a 3 cajas en posiciones aleatorias.
- Todos los robots empiezan en posiciones aleatorias vacías. \* Y, sólo puede haber un robot por celda.
- La simulación termina cuando todas las cajas se encuentran apiladas en pilas de exactamente 5 cajas.

### ¿Qué debes entregar?

- Un cuaderno de Jupyter Notebook conteniendo un reporte de la actividad. El cuaderno deberá contener:
  - Código fuente documentado.
  - Descripción detallada de la estrategia y los mecanismos utilizados en tu solución.
  - Una visualización que permita ver los diferentes pasos de la simulación.
  - El número de pasos necesarios para terminar la simulación.
  - ¿Existe una forma de reducir el número de pasos utilizados? Si es así, ¿cuál es la estrategia que se tendría en implementar?

### Criterios de evaluación

Los criterios que se utilizarán para evaluar sus soluciones y seleccionar a los tres primeros ganadores son los siguientes:

- Aplicación original, innovadora y efectiva de algoritmos computacionales para resolver problemas específicos.
- El rendimiento de la implementación. El rendimiento de la implementación se medirá en función los pasos necesarios para terminar la simulación.
- La calidad de la descripción de análisis, diseño e implementación del sistema multiagente, la elegancia de su diseño e implementación.

In [2]:

```
# 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

# Debido a que necesitamos que todos los agentes inicien en una celda, elegimos 'MultiGrid'.
from mesa.space import SingleGrid, MultiGrid

# Con 'RandomActivation', hacemos que todos los agentes se activen "al mismo tiempo".
from mesa.time import SimultaneousActivation

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

# 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"
matplotlib.rcParams['animation.embed_limit'] = 2**128

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

# Definimos otros paquetes que vamos a usar para medir el tiempo de ejecución de nuestro algoritmo.
import time
import datetime

# Importamos el paquete 'seaborn' para hacer gráficas más atractivas.
import seaborn as sns

In [3]:
# Definimos la clase 'Organizer' que hereda de 'Model'.
# Esta clase define el comportamiento de los agentes que van a organizar las cajas.
class Organizer(Model):
    def __init__(self, id, model):
        super().__init__(id, model)
        self.random.seed(67890)
        self.isCarrying = False
        self.attempts = 0
        self.max_attempts = 10
        self.minimal = 0

    # Definimos el método 'move' que permite a los agentes moverse a una celda vacía.
    def random_move(self):
        possible_cells = self.model.grid.get_neighborhood(
            self.pos, moore=False, include_center=False
        )

        empty_cells = [
            cell for cell in possible_cells if self.model.grid.is_cell_empty(cell)
        ]

        if empty_cells:
            new_position = self.random.choice(empty_cells)
            self.model.grid.move_agent(self, new_position)

    # Definimos el método 'carry' que permite a los agentes tomar una caja de una celda.
    def carry(self):
        neighbors = self.model.grid.get_neighborhood(
            self.pos, moore=False, include_center=False
        )
        for neighbor_pos in neighbors:
            x, y = neighbor_pos

            if self.model.piles < 39:
                if self.model.boxes[x, y] >= 1 and self.model.boxes[x, y] < 5 and self.isCarrying == False:
                    self.isCarrying = True
                    self.minimal = self.model.boxes[x, y]
                break
            else:
                if (self.model.boxes[x, y] == 1 or self.model.boxes[x, y] == 2) and self.isCarrying == False:
                    self.isCarrying = True
                break

    # Definimos el método 'drop' que permite a los agentes dejar una caja en una celda.
    def drop(self):
        neighbors = self.model.grid.get_neighborhood(
            self.pos, moore=False, include_center=False
        )
        for neighbor_pos in neighbors:
            x, y = neighbor_pos
            if self.model.boxes[x, y] >= self.minimal and self.model.boxes[x, y] < 5 and self.isCarrying == 1:
                self.model.boxes[x, y] += 1
                self.isCarrying = False
                return

    # Definimos el método 'step' que permite a los agentes moverse, tomar y dejar cajas.
    def step(self):
        if not self.isCarrying:
            self.carry()
        else:
            self.drop()

    self.random_move()

In [4]:
# Definimos la función 'get_agents' que nos permite obtener la posición de cada agente.
def get_agents(model):
    agents = np.zeros((model.grid.width, model.grid.height))
    for agent in model.schedule.agents:
        x, y = agent.pos
        agents[x][y] = 1
    return agents

In [5]:
# Definimos la clase 'Warehouse' que hereda de 'Model'.
# Esta clase define el entorno en el que se van a mover los agentes.
# Además, este método va a detener la simulación cuando se hayan organizado 40 pilas de 5 cajas.
class Warehouse(Model):
    def __init__(self, num_agents, width, height):
        self.random.seed(67890)
        self.grid = SingleGrid(width, height, torus=False)
        self.boxes = np.zeros((width, height))
        self.schedule = SimultaneousActivation(self)
        self.piles = 0

        self.running = True

    reporters = {"Agents": get_agents,
                 "Boxes": lambda m: self.boxes.copy(),
                 "Piles": lambda m: self.piles}

    self.datacollector = DataCollector(model_reporters=reporters)

    # Creamos los agentes en ubicaciones aleatorias.
    for i in range(num_agents):
        a = Organizer(i, self)
        self.schedule.add(a)

    # Intentar encontrar una celda vacía para el nuevo agente
    placed = False
    while not placed:
        x = self.random.randrange(width)
        y = self.random.randrange(height)
        if self.grid.is_cell_empty((x, y)):
            self.grid.place_agent(a, (x, y))
            placed = True

    # Creamos las cajas en ubicaciones aleatorias.
    num_boxes = 200
    for i in range(num_boxes):
        x = self.random.randrange(width)
        y = self.random.randrange(height)
        while self.boxes[x][y] >= 3:
            x = self.random.randrange(width)
            y = self.random.randrange(height)
        # Agregar una caja a la celda
        self.boxes[x][y] += 1

    # Definimos el método 'count_piles' que permite contar el número de pilas de 5 cajas.
    def count_piles(self):
        temp_piles = 0
        for x in range(self.grid.width):
            for y in range(self.grid.height):
                if self.boxes[x][y] == 5:
                    temp_piles += 1
        return temp_piles

    # Definimos el método 'step' que va a guardar la información de cada paso de la simulación.
    # Además, este método va a detener la simulación cuando se hayan organizado 40 pilas de 5 cajas.
    def step(self):
        self.count_piles()
        self.piles = self.count_piles()
        self.datacollector.collect(self)
        self.schedule.step()

        if self.piles == 40:
            self.running = False

In [6]:
# Definimos los parámetros que se van a usar para la simulación.
WIDTH = 20
HEIGHT = 20
N = 5

In [7]:
# Creamos una instancia de la clase 'Warehouse', y ejecutamos la simulación.
model = Warehouse(N, WIDTH, HEIGHT)
while model.running:
    model.step()

# Obtenemos la información de cada paso de la simulación.
all_grid = model.datacollector.get_model_vars_dataframe()
```

In [8]:

```
# Print total steps
print("Tomó", model.schedule.steps, "pasos organizar", model.piles, "pilas de 5 cajas.")
```

Tomó 4509 pasos organizar 40 pilas de 5 cajas.

In [9]:

```
# Heatmap de la distribución de las cajas al inicio de la simulación.
all_grid = model.datacollector.get_model_vars_dataframe()
```

```
heatmap_data = all_grid["Boxes"].iloc[0]
```

```
sns.heatmap(heatmap_data,
            annot=True,
            fmt="g",
            cmap="Spectral",
            cbar_kws={"label": "Número de cajas"},
```

```
plt.title("Heatmap - Inicio de la simulación")
```



Once Loop Reflect

In [10]:

```
# Heatmap de la distribución de las cajas al final de la simulación.
all_grid = model.datacollector.get_model_vars_dataframe()
```

```
heatmap_data = all_grid["Boxes"].iloc[model.schedule.steps - 1]
```

```
sns.heatmap(heatmap_data,
            annot=True,
            fmt="g",
            cmap="Spectral",
            cbar_kws={"label": "Número de cajas"},
```

```
plt.title("Heatmap - Fin de la simulación")
```



In [11]:

```
# Código para crear una animación de cada paso de la simulación.
fig, axs = plt.subplots(5,5)
```

```
axs.set_xticks([])
axs.set_yticks([])
```

```
agents = all_grid.get('Agents')
```

```
boxes = all_grid.get('Boxes')
```

```
data = boxes + (4 * agents)
```

```
patch = axs.imshow(data[0], cmap='Greys')
```

```
plt.close()
```

```
def animate(i):
    patch.set_data(data[i])
```

```
anim = animation.FuncAnimation(fig, animate, frames=model.schedule.steps, interval=100)
```

```
anim
```

```
Out[11]:
```



In [12]:

```
# Grafica del número de pilas por paso de la simulación.
```

```
plt.figure(figsize=(5, 5))
plt.plot("piles", linewidth=1, color="green", label="Pilas")
plt.xlabel("Paso")
plt.ylabel("Número de pilas")
```

```
plt.title("Pilas x paso")
```

```
# Líneas verticales para los pasos específicos.
```

```
plt.axvline(x=100, linestyle='--', color='red', label='Paso 100')
plt.axvline(x=1000, linestyle='--', color='cyan', label='Paso 1000')
plt.axvline(x=model.schedule.steps, linestyle='--', color='black', label=f'Paso (Total {model.schedule.steps})')
```

```
plt.legend()
```

```
plt.show()
```

```
plt.title("Pilas x paso")
```



## Solución

### Descripción de la estrategia

En mi estrategia, he diseñado un enfoque en el cual los robots se desplazan de manera aleatoria por el almacén. Cada robot posee una variable que indica si está actualmente cargando una caja, dado que el reto impone la restricción de transportar únicamente una caja a la vez. Cuando un robot no está cargando ninguna caja, examina las 4 celdas adyacentes en busca de una pila de cajas. Si encuentra una pila, procede a cargar una caja solo si la cantidad de cajas en la pila es menor a 5.

Sin embargo, hay una condición especial para cuando solo falta completar una pila: en ese caso, el robot solo puede mover cajas de pilas que contengan 1 o 2 cajas. Para optimizar el proceso, el robot, al tomar una caja, registra la cantidad de cajas originalmente en la pila de la que la obtuvo. Posteriormente, mientras carga una caja, el robot se desplaza aleatoriamente por el almacén.

Si durante su movimiento aleatorio encuentra en sus 4 casillas adyacentes otra pila de cajas, la caja que transporta se deposita en dicha pila. Sin embargo, para dejar la caja en la pila, esta debe contener un número de cajas mayor o igual al registro de la cantidad original de cajas en la pila de la que se tomó la caja, y además, la pila debe tener menos de 5 cajas en total. De esta manera, el robot continúa desplazándose de manera aleatoria por el almacén, cargando y depositando cajas, hasta que todas las cajas se encuentren en pilas de 5 unidades cada una. Por lo que mi estrategia se basa en un mecanismo de búsqueda aleatoria de pilas de cajas, y en un mecanismo de depósito de cajas en pilas que cumplen con ciertas condiciones.

### Optimización de la estrategia

En mi estrategia, que se basa principalmente en movimientos aleatorios, podría lograrse una mayor eficiencia al implementar un enfoque más direccionalizado para los robots. Se podría hacer que los robots escaneen el almacén mientras se desplazan. Al depositar una caja, el robot puede adquirir información sobre la ubicación de otras cajas en el almacén y dirigirse directamente hacia ellas en lugar de depender de un movimiento aleatorio.

Además, se podría mejorar la coordinación entre los robots mediante la comunicación. Si un robot encuentra cajas durante su exploración, podría compartir esta información con otros robots, permitiéndoles dirigirse directamente a las ubicaciones identificadas. Esta colaboración podría aumentar la eficiencia global del sistema.

En términos de selección de pilas para depositar cajas, se puede introducir una lógica de elección más sofisticada. Por ejemplo, los robots podrían evaluar la cantidad de cajas en diferentes pilas y dirigirse hacia aquella que les resulte más conveniente. Si hay una pila con 4 cajas y otra con 2, un robot podría optar por ir a la pila con 4 cajas para depositar su carga, contribuyendo así a completar esa pila de manera más eficiente.

### Conclusión

Fue una actividad retadora, ya que tuve que ir buscando cada vez nuevas estrategias y soluciones para irlo optimizando. Al principio lograba la tarea hasta después de 165 mil pasos. Eso no es para nada eficiente, luego fueron 70 mil, después 22 mil y ya por último la solución actual. Me gustó el trabajo logrado, pero se que hay áreas de oportunidad que quizás después me gustaría resolver como reto personal.