



Campus Querétaro

Materia:

Modelación de sistemas multiagentes con gráficas computacionales

(Gpo 301)

Reto final

Profesores:

Pedro Perez

Alejandro Fernández

Denisse Lizbeth Maldonado Flores

Alumnos:

Luis Gabriel Delfín Paulín A01701482

Eliuth Balderas Neri A01703315

Carlos Isaac Dávalos Lomelí A01706041

Introducción

La gestión de residuos es un gran problema a nivel mundial y necesita una atención seria. No existe una gestión adecuada de los desechos y la basura en las zonas rurales y urbanas puede suponer una amenaza para la seguridad sanitaria, la higiene, la seguridad humana y la seguridad de la vida silvestre.

Actualmente, el sistema manual de recolección de basura existe en la mayoría de los lugares, donde está involucrada la intervención humana. La recolección manual de basura y la gestión de desechos son una buena fuente para generar empleo, pero existen algunos problemas asociados, como que en algún momento no hay disponibilidad de trabajo manual durante días. Existe una gran preocupación por la seguridad humana cuando existen gases nocivos. La unidad de recolección de basura autónoma o semiautónoma puede tener un alto costo de fabricación pero un menor costo de mantenimiento. Los robots autónomos pueden ser una opción mucho mejor cuando se trata de abolir la monotonía de las tareas, superar los problemas de seguridad durante el trabajo manual y llegar a áreas remotas.

Descripción del reto

El reto consiste en desarrollar la solución para que un equipo de 5 robots limpien una oficina desordenada llena de basura en el menor tiempo posible.

Cada robot cuenta con un depósito de almacenamiento de basura, 5 unidades, sensores que les permiten determinar la cantidad de basura en cada zona, sensores de colisión y brazos de recolección. La oficina cuenta con una papelera de capacidad infinita y todos los robots saben en qué lugar se encuentra. Cuando un robot ha llenado su depósito de basura, deberá dirigirse a la papelera para vaciarlo. Una vez que hayan hecho esto, siguen limpiando. En el momento en que hayan terminado de limpiar toda la oficina, todos los robots se detienen.

Narrativa

El Estudiante Innovador

Iba Leo un día caminando por el campus cuando se dio cuenta de un problema recurrente en la escuela: la gestión ineficiente para recoger la basura. Los botes de basura estaban desbordados y los papeles se acumulaban en las esquinas. Leo es un estudiante de ingeniería apasionado por la tecnología y la robótica. Siempre había querido aplicar sus conocimientos para resolver problemas reales y mejorar la vida de las personas.

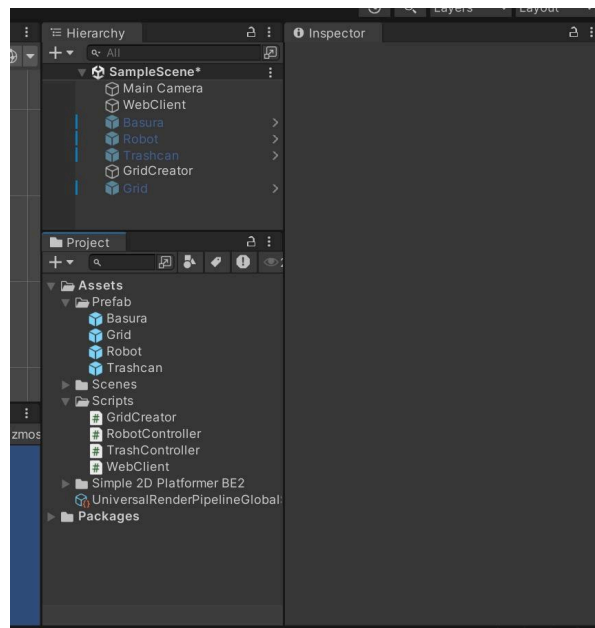
Intrigado y motivado, Leo decidió abordar este desafío como su proyecto final. Sabía que la recolección a mano de la basura era costosa e ineficiente. Así que junto con su equipo se propuso desarrollar una solución innovadora: un sistema de robots autónomos que pudieran limpiar una oficina o cualquier otro espacio de manera eficiente y segura.

Arquitectura del Sistema

- Google Colab: Desarrollo y simulación de la lógica de los robots.
<https://colab.research.google.com/drive/1saOSV6fuzRVLeM3c4Bj90X81DusZTx0r?usp=sharing>
- Unity: Visualización 2D de la simulación.
- Servidor Python: Intermediario para la comunicación entre Colab y Unity.

Usar código

- Entrar a GitHub con el siguiente link:
<https://github.com/Charlie-Davalos/Reto-Final-Multiagentes>
- Descargar el archivo “Final” que es el que incluye Unity
- Descargar los archivos *office_layout.txt*, *office_simulation.py* y *tc2008B_server.py*
- En la terminal corre “python tc2008B_server.py”, para ello, es necesario hacer un cd a la ruta del archivo.
- Una vez hecho esto, abrir el proyecto en unity. Este debe tener la siguiente estructura.

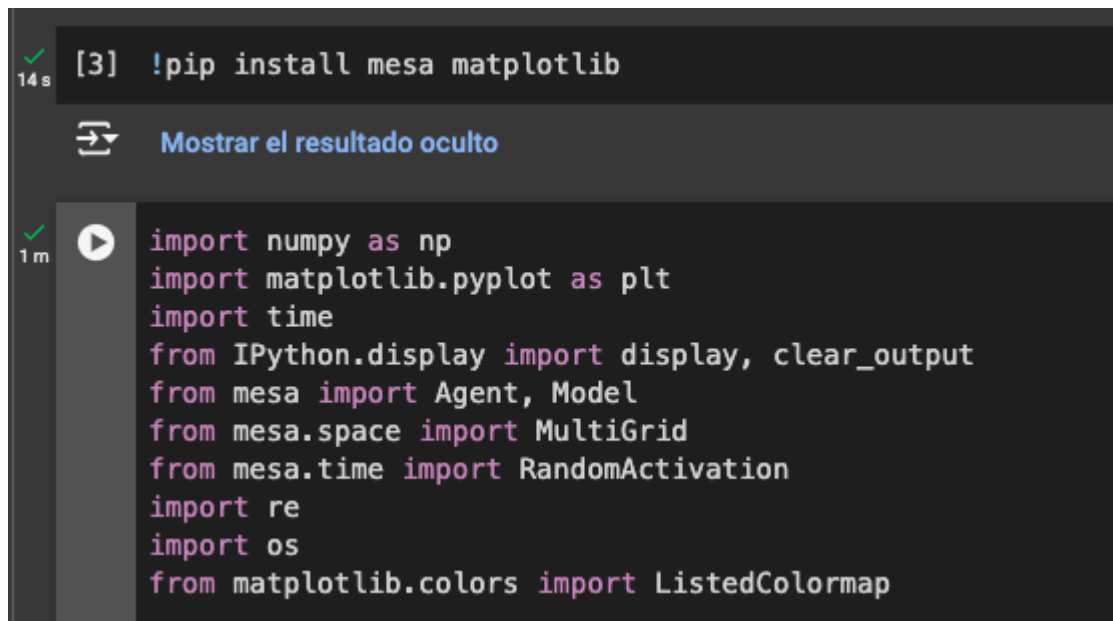


- Darle “play”
- Se visualizará lo siguiente: <https://youtu.be/cxLzbUHxwcA?si=H1S2EhaUvloyqhTP>

Código

Los agentes tienen movimientos aleatorios con capacidad de recoger unidades de basura dentro de un rango de visión de vecindario moore. Al momento de tener 5 unidades de basura, el agente robot está lleno y cambia su estado para ir a vaciar su contenedor en la papelera, donde se utiliza un algoritmo de Manhattan para desplazarse de manera eficiente a la papelera.

Importa las librerías necesarias para la simulación, visualización y manipulación de datos. mesa se utiliza para la creación de agentes y el modelo.

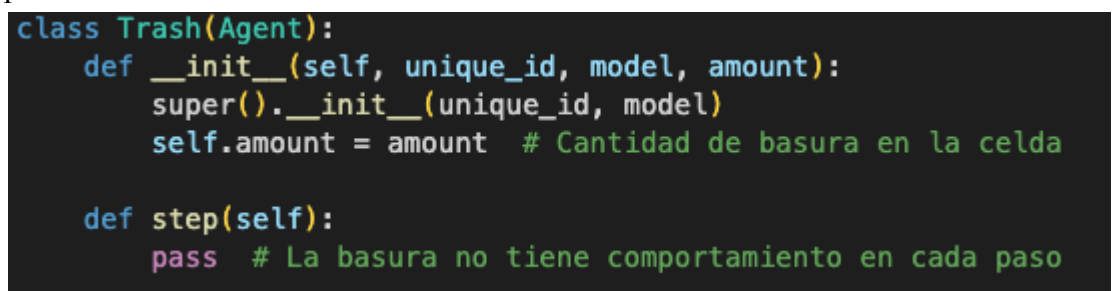


```
[3] !pip install mesa matplotlib

Mostrar el resultado oculto

import numpy as np
import matplotlib.pyplot as plt
import time
from IPython.display import display, clear_output
from mesa import Agent, Model
from mesa.space import MultiGrid
from mesa.time import RandomActivation
import re
import os
from matplotlib.colors import ListedColormap
```

La clase Trash representa los agentes de basura en el modelo. La basura tiene una cantidad pero no realiza acciones.



```
class Trash(Agent):
    def __init__(self, unique_id, model, amount):
        super().__init__(unique_id, model)
        self.amount = amount # Cantidad de basura en la celda

    def step(self):
        pass # La basura no tiene comportamiento en cada paso
```

Clase Robot que tiene 2 funciones

Función move_to - Mueve el robot hacia un destino considerando obstáculos y límites del grid.

Función step - Define el comportamiento del robot en cada paso: moverse, recolectar basura y regresar a la papelera si está lleno.

```

class Robot(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.collected_trash = 0 # Cantidad de basura recolectada por el robot
        self.returning = False # Indicador de si el robot está regresando a la papelerera
        self.origin_pos = None # Posición original del robot antes de ir a la papelerera

    def move_to(self, destination):
        """
        Mover el robot hacia la posición destino utilizando la distancia Manhattan.
        """
        current_x, current_y = self.pos
        dest_x, dest_y = destination
        possible_steps = []

        # Determinar posibles movimientos en dirección al destino
        if current_x < dest_x:
            possible_steps.append((current_x + 1, current_y))
        elif current_x > dest_x:
            possible_steps.append((current_x - 1, current_y))

        if current_y < dest_y:
            possible_steps.append((current_x, current_y + 1))
        elif current_y > dest_y:
            possible_steps.append((current_x, current_y - 1))

```

✓ Conectado a del backend de Google Compute Engine en Python 3

Clase TrashCan que representa la papelerera en el modelo. No realiza acciones.

```

class TrashCan(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.amount = 0 # Cantidad de basura en la papelerera

    def step(self):
        pass # La papelerera no tiene comportamiento en cada paso

```

Crea la oficina con un grid para activar los agentes, coloca la papelerera y la basura según el diseño que se le pase de la oficina.

```

class OfficeModel(Model):
    def __init__(self, width, height, office_layout):
        super().__init__()
        self.grid = MultiGrid(width, height, False) # MultiGrid no toroidal
        self.schedule = RandomActivation(self)
        self.office_layout = office_layout
        self.trashcan_pos = None
        self.trashcan = None

        # Inicializa la oficina y encuentra la posición de la papelera
        for y in range(height):
            for x in range(width):
                if office_layout[y][x] == 'P':
                    self.trashcan_pos = (x, y)
                    self.trashcan = TrashCan(1, self)
                    self.grid.place_agent(self.trashcan, self.trashcan_pos)
                    self.schedule.add(self.trashcan)
                elif office_layout[y][x] != 'X' and office_layout[y][x] != '0':
                    amount = int(office_layout[y][x])
                    trash = Trash((x, y), self, amount)
                    self.grid.place_agent(trash, (x, y))
                    self.schedule.add(trash)

```

visualize_simulation - Muestra el estado del grid en cada paso de la simulación, usando colores para representar diferentes tipos de celdas y agentes.

```

def visualize_simulation(model, steps=100):
    fig, ax = plt.subplots()
    cmap = ListedColormap(['white', 'red', 'green', 'blue', 'orange', 'purple']) # Mapa de colores personalizado

    for step in range(1, steps + 1):
        ax.clear()
        model.step()
        grid = np.full((model.grid.height, model.grid.width), 0)
        for cell in model.grid.coord_iter():
            cell_content, (x, y) = cell
            if len(cell_content) > 0:
                if any(isinstance(agent, Robot) for agent in cell_content):
                    grid[y][x] = 5 # Morado para los robots
                elif any(isinstance(agent, Trash) for agent in cell_content):
                    grid[y][x] = 4 # Naranja para la basura
            else:
                grid[y][x] = 0 # Blanco para celdas vacías

            if model.office_layout[y][x] == 'X':
                grid[y][x] = 1 # Rojo para las celdas con "X"
            elif model.office_layout[y][x] == 'P':
                grid[y][x] = 3 # Verde claro para la papelera

        # Contador de basura total en la oficina
        total_trash = sum(trash.amount for trash in model.schedule.agents if isinstance(trash, Trash))

        ax.imshow(grid, cmap=cmap, vmin=0, vmax=5)
        ax.text(0.5, 1.01, f"Steps: {step}", transform=ax.transAxes, ha="center")
        ax.text(0.5, 1.05, f"Total Trash: {total_trash}", transform=ax.transAxes, ha="center")

```

✓ Conectado a del backend de Google Compute Engine en Python 3

También muestra el número de pasos y la cantidad de basura restante.

Blanco: Celdas vacías

Rojo: Obstáculos (X)

Verde: Papelera (P)

Naranja: Basura

Morado: Robots

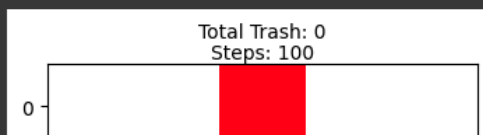
run_simulation - Ejecuta la simulación creando el modelo y visualizando la simulación durante un número especificado de pasos.

También se verifica si el archivo de entrada existe antes de ejecutar la simulación.

```
def run_simulation(file_path, steps=100):
    n, m, office_layout = read_office_layout(file_path)
    model = OfficeModel(m, n, office_layout)
    visualize_simulation(model, steps)

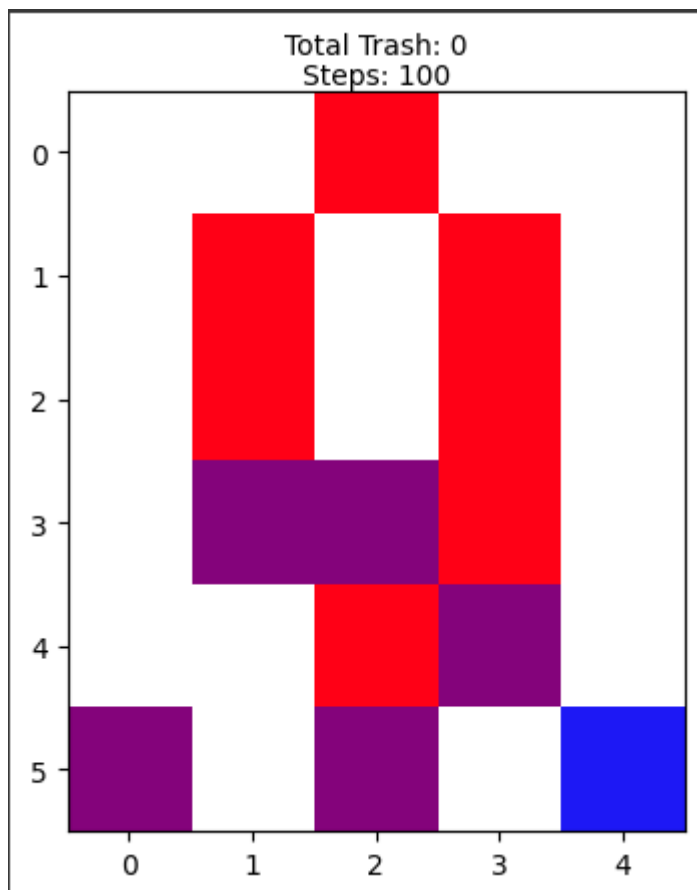
if __name__ == "__main__":
    # Ruta del archivo en Google Colab
    input_csv = "/content/input.txt"

    # Comprobar si el archivo existe
    if not os.path.isfile(input_csv):
        print("El archivo no existe. Por favor, comprueba la ruta del archivo e inténtalo de nuevo.")
    else:
        run_simulation(input_csv)
```



✓ Conectado a del backend de Google Compute Engine en Python 3

Resultado de la simulación



Mecanismos de Comunicación

Los robots no se comunican directamente entre sí ni con la basura o la papelera. La "comunicación" sucede de manera implícita a través de sus interacciones con el entorno (grid). Los robots detectan la presencia de basura o la papelera y actúan en consecuencia. La comunicación implícita se implementa mediante:

- Detección de contenido en celdas: Los robots usan `get_cell_list_contents` para identificar qué agentes están presentes en su posición actual.
- Decisiones basadas en el entorno: Los robots toman decisiones de movimiento y recolección basadas en el contenido de las celdas y las reglas del modelo.

Simulación gráfica en Unity

Ejecución del Servidor:

```
pip install mesa numpy matplotlib  
python tc2008B_server.py
```

- Inicia el servidor HTTP en el puerto especificado (por defecto 8585) y lo ejecuta de forma indefinida hasta que se interrumpe manualmente.



```
tc2008B_server.py 1 • C# TrashController.cs  
1 import json  
2 from http.server import BaseHTTPRequestHandler, HTTPServer  
3 import logging  
4 from office_simulation import OfficeModel, read_office_layout, Robot, Trash  
5  
6 class Server(BaseHTTPRequestHandler):  
7     model = None  
8
```

- Importa las bibliotecas necesarias para manejar el servidor HTTP y el modelo de simulación y define una clase que maneja las solicitudes HTTP.


```

9      @staticmethod
10     def initialize_model():
11         n, m, office_layout = read_office_layout('office_layout.txt')
12         Server.model = OfficeModel(m, n, office_layout)
13
14     @staticmethod
15     def get_robot_positions():
16         positions = []
17         for agent in Server.model.schedule.agents:
18             if isinstance(agent, Robot):
19                 positions.append({'id': agent.unique_id, 'x': agent.pos[0], 'y': agent.pos[1], 'collected': agent.collected})
20         return positions
21
22     @staticmethod
23     def get_trash_positions():
24         trash_positions = []
25         for agent in Server.model.schedule.agents:
26             if isinstance(agent, Trash):
27                 trash_positions.append({'x': agent.pos[0], 'y': agent.pos[1], 'amount': agent.amount})
28         return trash_positions

```

- Son los métodos con los que lee el diseño de la oficina desde un archivo y crea una instancia, obtiene las posiciones de todos los robots y basuras en la simulación.

```
tc2008B_server.py 1 • TrashController.cs

6 class Server(BaseHTTPRequestHandler):
30     @staticmethod
31     def get_office_layout():
32         layout = []
33         for row in Server.model.office_layout:
34             layout.append(" ".join(row))
35         return layout
36
37     def _set_response(self):
38         self.send_response(200)
39         self.send_header('Content-type', 'application/json')
40         self.end_headers()
41
42     def do_GET(self):
43         self._set_response()
44         response = {
45             'robots': Server.get_robot_positions(),
46             'trash': Server.get_trash_positions(),
47             'office_layout': Server.get_office_layout()
48         }
49         self.wfile.write(json.dumps(response).encode('utf-8'))
50
51     def do_POST(self):
52         if Server.model is None:
53             Server.initialize_model()
54
55         Server.model.step()
56         response = {
57             'robots': Server.get_robot_positions(),
58             'trash': Server.get_trash_positions(),
59             'office_layout': Server.get_office_layout()
60         }
```

Lín. 59, col. 56 Espacios: 4 UTF-8 LF Python

`_set_response`: Método para configurar las respuestas HTTP con un código de estado 200 y el tipo de contenido JSON.

`do_GET`: Maneja las solicitudes GET devolviendo las posiciones de los robots, basura y el diseño de la oficina en formato JSON.

`do_POST`: Maneja las solicitudes POST inicializando el modelo si no está ya inicializado, avanzando un paso en la simulación y devolviendo el nuevo estado en formato JSON.

Archivos de C# para Unity

WebClient.cs

Define un componente Unity llamado WebClient que se encarga de la comunicación con un servidor web para obtener y actualizar información sobre la simulación. Este componente se encarga de enviar datos al servidor, recibir configuraciones iniciales y actualizar la posición y el estado de los robots y la basura en la escena.

TrashController.cs

Define el comportamiento de los objetos de basura en la escena. Este script controla la cantidad de basura en cada objeto y maneja la lógica de recolección por parte de los robots.

RobotController.cs

Define el comportamiento de los robots en la escena. Maneja el movimiento de los robots hacia posiciones objetivo y la lógica de recolección de basura.

GridCreator.cs

Se encarga de crear una cuadrícula de celdas en la escena, basada en un diseño de oficina obtenido de un archivo. Este script se utiliza para configurar la disposición inicial del entorno en el que operan los robots y la basura.

StepCounterTMP.cs

Se encarga de actualizar los steps dados en la simulación, este dato se ve reflejado en un objeto TextMeshPro