

Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Querétaro

**Actividad Integradora**

Docentes

Alejandro Fernández Vilchis

Denisse Lizbeth Maldonado Flores

Pedro Oscar Pérez Murrieta

**Presenta:**

Alan Patricio González Bernal - A01067546

Alan Rodrigo Castillo Sánchez – A01708668

Miguel Ángel Tena García – A01709653

Querétaro, Querétaro México 05 de septiembre del 2023

# Índice

[Índice 2](#_Toc144773868)

[Contexto 3](#_Toc144773869)

[Solución implementada 4](#_Toc144773870)

[Código .ipynb 4](#_Toc144773871)

[Servidor y cliente 11](#_Toc144773872)

[Unity 11](#_Toc144773873)

[Mecanismos de comunicación 11](#_Toc144773874)

[Nivel conceptual 11](#_Toc144773875)

[Nivel código 11](#_Toc144773876)

# Contexto

El sistema es una simulación de una oficina, siendo el tamaño y datos de la oficina siendo brindados por los datos de un .txt. El txt brindará varios tipos de datos diferentes:

1. X
   1. Un obstáculo, lugar inaccesible
2. Valor numérico (0-9)
   1. La cantidad de basura que existe en esa celda
3. S
   1. El punto de partida de los robots que a partir de ahora serán llamados “aspiradoras”
4. P
   1. La papelera, punto en el que las aspiradoras deben dejar la basura antes de seguir recogiendo más
5. R
   1. La aspiradora en cuestión. Esta se identificará de las demás por un número a lado de R.

Las limitaciones del sistema son las siguientes:

1. Cada aspiradora solo puede recoger 5 unidades de basura antes de tener que ir a vaciar todo en la papelera.
2. Las aspiradoras no pueden estar en la misma celda varias a la vez, solo en el punto S esto es permitido.
3. Las aspiradoras no conocen su alrededor, por lo que tendrán que mapear todo o moverse de forma aleatoria.
4. Las aspiradoras no pueden pasar por los puntos X
5. Las aspiradoras no pueden saber la cantidad de basura que tengan las celdas vecinas, Solo pueden saber la cantidad exacta una vez estén en esa celda.

Tomando en cuenta todas estas limitaciones y lo aprendido en clase, se desarrolló una solución la cual toma en cuenta librerías de Python como Mesa, Matplotlib, numpy, pandas, etc. Para poder generar una solución que satisfaga lo que se indica y cumpla con las limitaciones brindadas. Esta solución se analizará más adelante a lo largo de este documento.

# Solución implementada

La solución generada incluye varios pasos.

## Código .ipynb

Utilizando las libretas de jupyter, se comenzó el desarrollo para la solución de este problema. Principalmente, se utilizó para testeo el siguiente archivo .txt

mapa.txt

6 5

S 4 0 6 0

6 X 0 X 4

0 X 7 X 0

5 0 0 X 8

0 0 X 0 0

0 0 0 0 P

Tomando en cuenta este archivo de entrada, se generó la solución. Primero se realizaron los imports y las lecturas de mapa.txt. Aquí se generaron dos veces la lectura del mapa:

def leer\_mapa(filename):

    with open(filename, 'r') as f:

        lines = f.readlines()

        mapaREAL = [list(line.replace(" ", "").strip()) for line in lines[1:]]

    filas = len(mapaREAL)

    columnas = len(mapaREAL[0])

    mapaUNKNOWN = [fila[:] for fila in mapaREAL]  *# Crear una copia de mapaREAL*

    for i in range(filas):

        for j in range(columnas):

            if mapaREAL[i][j] not in ['S', 'P']:

                mapaUNKNOWN[i][j] = '?'

    return mapaREAL, mapaUNKNOWN

La variable mapaREAL contiene el mapa de igual forma que mapa.txt, mientras que mapaUNKNOWN reemplaza todos los valores que no sean S o P con el signo ?, siendo este mapa el que utilizarán las aspiradoras para realizar el mapeo inicial.

A partir de este punto, se realiza un grafo del mapa, siendo los puntos marcados como X nodos inexistentes. Después de esto, se implementa el algoritmo A\*, el cual le permite a las aspiradoras encontrar el camino más próximo a sus objetivos.

def heuristic(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])

*# Implementación del algoritmo A\**

def a\_star\_search(graph, start, goal):

    frontier = []

    heapq.heappush(frontier, (0, start))

    came\_from = {}

    cost\_so\_far = {}

    came\_from[start] = None

    cost\_so\_far[start] = 0

    while frontier:

        \_, current = heapq.heappop(frontier)

        if current == goal:

            break

        for next in graph[current]:

            new\_cost = cost\_so\_far[current] + 1

            if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:

                cost\_so\_far[next] = new\_cost

                priority = new\_cost + heuristic(goal, next)

                heapq.heappush(frontier, (priority, next))

                came\_from[next] = current

*# Reconstruct path*

    current = goal

    path = []

    while current != start:

        path.append(current)

        current = came\_from[current]

    path.append(start)

    path.reverse()

    return path

Cabe destacar que estos algoritmos solo se definen, más no se utilizan hasta que el mapeo está completado.

Después de esto, se realiza la definición de agentes.

class Robot(Agent):

    def \_\_init\_\_(self, unique\_id, model, inicio):

        super().\_\_init\_\_(unique\_id, model)

        self.posicion = inicio

        self.capacidad = 5

        self.mapeoTerminado = False

        self.mapeoIniciado = False

        self.pasos = 0

El agente cuenta con 5 definiciones: mapeoInicial(), actualizarMapa(), mover(), recogerBasura(), step().

mapeoInicial():

Las aspiradoras salen del punto de inicio, del cual se distribuyen entre las columnas para así poder realizar el mapeo de la forma más eficiente posible. Una vez que estas aspiradoras iniciales terminan el mapeo se considera que esta definición ha terminado.

    def mapeoInicial(self):

        mapaU = self.model.mapaUNKNOWN

        columnas = len(mapaU[0])

        filas = len(mapaU)

        if columnas % 3 == 0:

            agentesIniciales = int(np.ceil(columnas / 3))

            desplazamiento = 3

        else:

            agentesIniciales = int(np.floor(columnas / 2))

            desplazamiento = 2

        if self.unique\_id in range(agentesIniciales):

            posicionInicial = (0, 1)

            if self.unique\_id == 0:

                destino = (filas - 1, 1)

            else:

                posicionInicial = (posicionInicial[0], posicionInicial[1] + (desplazamiento \* self.unique\_id))

                destino = (filas - 1, posicionInicial[1])

            return posicionInicial, destino

        else:

            inicialesTerminados = True  *# Variable para verificar si todos los agentes iniciales terminaron*

            for agente in self.model.schedule.agents[:agentesIniciales]:

                if not agente.mapeoTerminado:  *# Si el agente no ha terminado*

                    inicialesTerminados = False  *# Cambiamos la variable a False*

                    break

            if inicialesTerminados:

*# Si todos los agentes iniciales terminaron, se verifica que no queden ? en el mapa*

                terminaMapeo = True

                for i in range(filas):

                    for j in range(columnas):

                        if mapaU[i][j] == '?':

                            terminaMapeo = False

*# Si hay ? se activa un agente auxiliar para que termine el mapeo*

                            if self.unique\_id == agentesIniciales:

                                self.mapeoIniciado = True

                                return self.posicion, (i, j)

*# Si ya no hay ? se establece que el mapeo ha terminado*

                if terminaMapeo:

                    self.mapeoTerminado = True

            return None, None

actualizarMapa():

Como el nombre lo indica, se actualiza el mapa tomando en cuenta los nodos vecinos, de esta forma, los agentes tienen un mapa actualizado a cada paso que se da.

    def actualizarMapa(self):

        self.model.mapaUNKNOWN[self.posicion[0]][self.posicion[1]] = self.model.mapaREAL[self.posicion[0]][self.posicion[1]]

*# Vecinos del nodo actual (Pero incluyendo nodos con obstaculos)*

        pos\_vecinos = []

        i, j = self.posicion

        if i - 1 >= 0: pos\_vecinos.append((i-1, j))

        if i + 1 < len(self.model.mapaUNKNOWN): pos\_vecinos.append((i+1, j))

        if j - 1 >= 0: pos\_vecinos.append((i, j-1))

        if j + 1 < len(self.model.mapaUNKNOWN[0]): pos\_vecinos.append((i, j+1))

        for pos\_vecino in pos\_vecinos:

            if self.model.mapaREAL[pos\_vecino[0]][pos\_vecino[1]] == 'X':

                self.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] = 'X'

            elif self.model.mapaREAL[pos\_vecino[0]][pos\_vecino[1]] == '0':

                self.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] = '0'

            elif self.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] == '?':

                self.model.mapaUNKNOWN[pos\_vecino[0]][pos\_vecino[1]] = 'B'

mover():

Función que se repetirá a cada step, permite al agente cambiar de posición y aumentar 1 a su contador de pasos. Además de llamar a la función de actualizar mapa.

    def mover(self, nueva\_posicion):

        self.posicion = nueva\_posicion

        self.pasos += 1

        self.actualizarMapa()

recogerBasura():

Si está en una posición donde exista basura, recoge la cantidad que le sea posible y actualiza el mapa para representar la cantidad que no se pudo recoger o en su defecto un 0.

    def recogerBasura(self):

        if self.model.mapaREAL[self.posicion[0]][self.posicion[1]].isdigit() and int(self.model.mapaREAL[self.posicion[0]][self.posicion[1]]) > 0:

            basura = min(int(self.model.mapaREAL[self.posicion[0]][self.posicion[1]]), self.capacidad)

            self.capacidad -= basura

            self.model.mapaREAL[self.posicion[0]][self.posicion[1]] = str(int(self.model.mapaREAL[self.posicion[0]][self.posicion[1]]) - basura)

step():

En esta definición se llaman las demás definiciones, de esta forma con llamar solo una función, por cada paso se llamarán las que se requieran.

    def step(self):

*# Si la capacidad es 0, se dirige a la papelera para vaciarse*

        if self.capacidad == 0:

            papelera = a\_star\_search(self.model.grafo, self.posicion, self.model.Papelera)

            self.mover(papelera[1])

            if self.posicion == self.model.Papelera:

                self.capacidad = 5

        else:

*# Mapeo inicial*

            if self.mapeoTerminado == False:

                posicionInicial, destino = self.mapeoInicial()

                if posicionInicial is not None and destino is not None:

                    if self.mapeoIniciado == False:

                        recorrido = a\_star\_search(self.model.grafo, self.posicion, posicionInicial)

                        if len(recorrido) > 1:

                            self.mover(recorrido[1])

                        if self.posicion == posicionInicial:

                            self.mapeoIniciado = True

                    elif self.posicion != destino and self.mapeoIniciado == True:

                        recorrido = a\_star\_search(self.model.grafo, self.posicion, destino)

                        if len(recorrido) > 1:

                            self.mover(recorrido[1])

                        if self.posicion == destino:

                            self.recogerBasura()

                            self.mapeoTerminado = True

*# Si la capacidad es mayor a 0, se dirige a la basura más cercana*

            else:

*# Encuentra la basura más cercana usando A\**

                basura\_cercana = None

                camino\_corto = [self.model.Inicio]

                for i in range(len(self.model.mapaUNKNOWN)):

                    for j in range(len(self.model.mapaUNKNOWN[0])):

                        if self.model.mapaUNKNOWN[i][j] == '?' or self.model.mapaUNKNOWN[i][j] == 'B' or (self.model.mapaUNKNOWN[i][j].isdigit() and int(self.model.mapaUNKNOWN[i][j]) > 0):

                            camino = a\_star\_search(self.model.grafo, self.posicion, (i, j))

                            if camino\_corto or len(camino) < len(camino\_corto):

                                camino\_corto = camino

                                basura\_cercana = (i, j)

*# Moverse hacia la basura más cercana*

                if len(camino\_corto) > 1:

                    self.mover(camino\_corto[1])

                    self.recogerBasura()

*# Si no hay basura, se dirige al INICIO (PARA PRUEBAS)*

                elif self.posicion != self.model.Inicio:

                    camino\_corto = a\_star\_search(self.model.grafo, self.posicion, self.model.Inicio)

                    self.mover(camino\_corto[1])

A partir de aquí, se comienza la sección referente al modelo. Es en este punto donde definimos cosas como la cantidad de aspiradoras, su activación, los mapas, etc.

def update(num, model):

    model.step()

class LimpiezaModel(Model):

    def \_\_init\_\_(self, mapa\_txt):

        self.num\_robots = 5

        self.schedule = RandomActivation(self)

        self.mapaREAL, self.mapaUNKNOWN = leer\_mapa(mapa\_txt)

        self.grafo = construir\_grafo(self.mapaREAL)

        for i in range(len(self.mapaREAL)):

            for j in range(len(self.mapaREAL[0])):

                if self.mapaREAL[i][j] == 'S':

                    self.Inicio = (i, j)

                elif self.mapaREAL[i][j] == 'P':

                    self.Papelera = (i, j)

*# Establecer el tamaño de la figura*

        self.fig, self.ax = plt.subplots(figsize=(10, 10))

*# Desactivar los ejes*

        self.ax.axis('off')

*# Creación de robots*

        for i in range(self.num\_robots):

            robot = Robot(i, self, self.Inicio)

            self.schedule.add(robot)

El modelo cuenta con solo dos definiciones, la de draw\_map() y step().

draw\_map():

Se realiza la animación de los agentes y de los demás detalles del mapa. Se le asigna un color, tamaño, etc.

    def draw\_map(self):

        self.ax.clear()

        self.ax.set\_xlim(0, len(self.mapaREAL[0]))  *# Ajustar límites del eje X*

        self.ax.set\_ylim(len(self.mapaREAL), 0)  *# Ajustar límites del eje Y*

        for i, row in enumerate(self.mapaUNKNOWN):

            for j, cell in enumerate(row):

                if cell == 'X':

                    self.ax.text(j, i, cell, ha='center', va='center', fontsize=60, color='black')

                else:

                    self.ax.text(j, i, cell, ha='center', va='center', fontsize=60, color='blue')

        for agent in self.schedule.agents:

            self.ax.text(agent.posicion[1], agent.posicion[0], 'R'+ str(agent.unique\_id), ha='center', va='center', fontsize=40, color='red')

step():

Se realizan los pasos para ejecutar el programa.

    def step(self):

        self.schedule.step()

        self.draw\_map()

El último bloque es el responsable de la animación.

model = LimpiezaModel('mapa.txt')

ani = animation.FuncAnimation(model.fig, update, fargs=(model,), frames=50)

ani.save('animation.gif', writer='imagemagick', fps=1)

html = HTML(ani.to\_jshtml())

display(html)

## Servidor y cliente

Para uso de esta clase, los profesores nos brindaron un código de Python el cual actúa como servidor, desplegándose en el puerto 3535. Para Unity se nos brindó un cliente en C# el cual nos permite desplegar lo que se genera en el archivo de Python en Unity.

## Unity

En el motor gráfico de Unity se realizaron pasos muy simples. Se generaron sprites para cada elemento de la simulación y a un objeto vacío se le añadió el script de WebClient, permitiendo que el código de Python funcione en Unity.

# Mecanismos de comunicación

Para comunicarse entre agentes, se genera un mapa “vacío”, el cual es el que se marca con ? los puntos que no se conocen, a excepción de los puntos S y P que siempre se sabrán. A partir de aquí, las aspiradoras comenzarán su mapeo, el cual actualiza ese mapa con datos reales para generar un mapa completo.

## Nivel conceptual

En el área de Servidor y cliente, nuestro código, al utilizar un grafo en lugar de una grid, el sistema tiene más facilidad de pasar las posiciones de los elementos a través de un JSON y un método POST, el cliente con un GET recibe esa información y la interpreta en Unity, brindando así la solución final de este reto.

## Nivel código

PENDIENTE SE TERMINA MAÑANA