2025

Modelos de Inteligencia Artificial

Profesor/a: Águeda María López Moreno

PRÁCTICA 2.1.- AGENTES REACTIVOS SIMPLES

02/02/2025

PEDRO MANUEL GARCÍA ÁLVAREZ | int.marisma@gmail.com | CEIA | I.E.S. La Marisma

Tabla de contenido

1.- Introducción

1.1- Objetivos de la práctica

Los objetivos principales de esta práctica son:

- Entender el concepto de programa agente, específicamente enfocado en agentes reactivos simples.
- Conocer el funcionamiento de los agentes reactivos simples y sus limitaciones.
- Analizar el comportamiento de un agente reactivo simple (robot aspirador) en diferentes escenarios de funcionamiento de sensores.
- Implementar una simulación que demuestre el comportamiento de un agente reactivo simple en un entorno controlado.

1.2- Conceptos fundamentales de agentes reactivos

Los agentes reactivos simples son un tipo básico de agente inteligente en el campo de la inteligencia artificial. Sus características principales son:

- Se basan únicamente en las percepciones actuales, ignorando el historial de percepciones pasadas.
- Siguen reglas de condición-acción simples para tomar decisiones.
- Son eficaces en entornos totalmente observables, pero tienen limitaciones en entornos parcialmente observables.
- Poseen una inteligencia limitada debido a su simplicidad.
- Pueden quedar atrapados en bucles infinitos en ciertos escenarios.

Los agentes reactivos simples se caracterizan por:

1. **Percepción del entorno**: Utilizan sensores para obtener información sobre su estado actual.
2. **Acción directa**: Responden inmediatamente a las percepciones sin mantener un estado interno complejo.

3. **Reglas predefinidas**: Operan según un conjunto de reglas evento-condición-acción establecidas.
4. **Ausencia de planificación**: No consideran las consecuencias futuras de sus acciones.

Aunque son limitados en complejidad, los agentes reactivos simples pueden ser efectivos para tareas específicas y bien definidas, como el ejemplo del robot aspirador que se explorará en esta práctica.

2.- Descripción del Problema

2.1.- Escenario del robot aspirador

El escenario planteado en esta práctica involucra un robot aspirador que opera en un entorno representado por una cuadrícula de 10x10 celdas. Las características principales de este escenario son:

- Entorno: Una cuadrícula de 10x10 celdas, donde cada celda puede estar en uno de tres estados: limpia, sucia o contener un obstáculo.
- Robot aspirador: Un agente reactivo simple que se mueve por la cuadrícula y puede realizar dos acciones básicas: limpiar la celda actual o moverse a una celda adyacente.
- Obstáculos: Representan áreas inaccesibles para el robot, distribuidas aleatoriamente en la cuadrícula.
- Suciedad: Distribuida aleatoriamente en las celdas al inicio de la simulación.

El objetivo del robot es limpiar todas las celdas sucias accesibles en el entorno.

2.2.- Limitaciones sensoriales

En este escenario, se plantean dos situaciones hipotéticas donde el robot aspirador experimenta limitaciones en sus capacidades sensoriales. Estas limitaciones afectan significativamente su comportamiento y eficacia.

2.2.1.- Sensor de suciedad roto

En esta situación, el robot no puede detectar si una celda está sucia o limpia. Las implicaciones de esta limitación son:

- El robot no puede determinar si necesita limpiar la celda actual.
- Debe adoptar una estrategia de limpieza "a ciegas", asumiendo que todas las celdas podrían estar sucias.
- Posibles soluciones incluyen:
  - Limpiar cada celda visitada, independientemente de su estado real.
  - Seguir un patrón de movimiento sistemático para cubrir toda la cuadrícula.
- Desventajas:
  - Ineficiencia energética al limpiar celdas que ya están limpias.
  - Mayor tiempo para completar la tarea de limpieza.

2.2.2.- Sensor de ubicación roto

En este caso, el robot no puede determinar su posición exacta en la cuadrícula. Las consecuencias de esta limitación son:

- El robot no puede seguir un patrón de movimiento predefinido basado en coordenadas.

- Solo puede detectar si la celda actual está sucia o limpia.

- Posibles estrategias de adaptación:
  - Movimiento aleatorio: El robot se mueve al azar, limpiando las celdas sucias que encuentra.
  - Seguir paredes: El robot podría seguir los bordes de la cuadrícula o los obstáculos para navegar.

- Desafíos:
  - Riesgo de quedar atrapado en áreas ya limpias.
  - Dificultad para garantizar una cobertura completa del área.

- Posibilidad de movimientos redundantes y repeticiones frecuentes a las mismas celdas.

Estas limitaciones sensoriales presentan desafíos significativos para el diseño de un agente reactivo simple eficaz. La práctica busca explorar cómo se puede adaptar el comportamiento del robot para superar estas limitaciones y aun así cumplir con su tarea de limpieza de manera razonable.

3.- Metodología

3.1.- Herramientas utilizadas

3.1.1.- Lenguaje de programación Python

Python es el lenguaje de programación elegido para este trabajo debido a su simplicidad y potencia. Algunas características relevantes de Python para esta práctica son:

- Sintaxis clara y legible, ideal para principiantes y trabajos educativos.
- Amplia biblioteca estándar y gran cantidad de módulos de terceros disponibles.
- Soporte multiplataforma, permitiendo que el trabajo se ejecute en diferentes sistemas operativos.
- Capacidad de integración con bibliotecas especializadas como Pygame.

3.1.2.- Biblioteca Pygame

Pygame es una biblioteca de Python diseñada específicamente para el desarrollo de juegos y simulaciones en 2D. Se ha elegido por las siguientes razones:

- Facilita la creación de animaciones interactivas 2D.

- Proporciona funciones para manejar eventos de teclado, ratón y joystick.
- Permite la detección de colisiones y el manejo de sonidos.
- Es gratuita, de código abierto y multiplataforma.
- Ofrece un equilibrio entre simplicidad para principiantes y flexibilidad para trabajos más complejos.

3.2.- Diseño del entorno de simulación

El entorno de simulación se ha diseñado para representar el escenario del robot aspirador de manera efectiva:

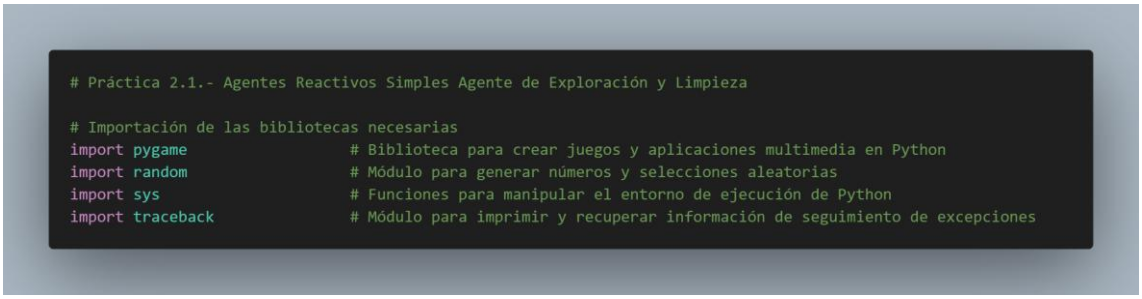
- Cuadrícula: Se utiliza una matriz de 10x10 para representar el área de limpieza.
- Representación visual: Cada celda de la cuadrícula se dibuja como un rectángulo en la pantalla utilizando Pygame.
- Estados de las celdas: Se utilizan diferentes colores para representar los estados de las celdas (limpio, sucio, obstáculo).
- Agente (robot aspirador): Se representa como un rectángulo móvil dentro de la cuadrícula.
- Movimiento del agente: Se implementa utilizando las funciones de manejo de eventos de Pygame para detectar las pulsaciones de teclas.
- Actualización de la pantalla: Se utiliza un bucle principal que actualiza constantemente la posición del agente y el estado de las celdas.

El diseño del entorno permite una visualización clara del comportamiento del agente y facilita la implementación de diferentes estrategias de limpieza según las limitaciones sensoriales planteadas en el problema.

4.- Implementación

4.1.- Estructura del código

4.1.1.- Importación de las bibliotecas



```
# Práctica 2.1.- Agentes Reactivos Simples Agente de Exploración y Limpieza

# Importación de las bibliotecas necesarias
import pygame                # Biblioteca para crear juegos y aplicaciones multimedia en Python
import random                # Módulo para generar números y selecciones aleatorias
import sys                   # Funciones para manipular el entorno de ejecución de Python
import traceback             # Módulo para imprimir y recuperar información de seguimiento de excepciones
```

Este código importa las bibliotecas necesarias (Pygame, random, sys y traceback) para crear una simulación interactiva de un agente reactivo simple que actúa como un robot explorador y limpiador, utilizando gráficos, aleatoriedad, control del sistema y manejo de errores.

4.1.2.- Inicializar pygame

```
# Inicializar pygame
pygame.init()
```

Este código inicializa todos los módulos de Pygame, preparando el entorno para el desarrollo de juegos o aplicaciones multimedia.

4.1.3.- Definición de colores en formato RGB

```
# Definición de colores en formato RGB
BLANCO = (255, 255, 255)    # Color blanco para celdas limpias
NEGRO = (0, 0, 0)           # Color negro para los bordes de las celdas y el texto
GRIS = (169, 169, 169)      # Color para las celdas sucias
AZUL = (0, 0, 255)          # Color para el robot (agente de limpieza)
ROJO = (255, 0, 0)           # Color para los obstáculos
```

Este código define constantes de color en formato RGB para representar visualmente diferentes elementos en la simulación del robot aspirador.

4.1.4.- Tamaño de cada celda de la cuadrícula

```
# Tamaño de cada celda de la cuadrícula
TAM_CELDA = 50
```

Este código define el tamaño de cada celda en la cuadrícula de la simulación, estableciendo que cada celda tendrá 50 píxeles de ancho y alto, lo que determinará la escala visual del entorno del robot aspirador.

4.1.5.- Definición del tamaño de la cuadrícula en términos de número de columnas y filas

```python
# Definición del tamaño de la cuadrícula en términos de número de columnas y filas
ANCHO = 10  # Número de columnas en la cuadrícula
ALTO = 10   # Número de filas en la cuadrícula
```

Este código define las dimensiones de la cuadrícula para la simulación, estableciendo un entorno de 10x10 celdas para el robot aspirador.

4.1.6.- Definir las dimensiones de la ventana de visualización

```python
# Definir las dimensiones de la ventana de visualización
ANCHO_VENTANA = TAM_CELDA * ANCHO
ALTO_VENTANA = TAM_CELDA * ALTO + 150  # Se añade espacio extra para la ventana de texto y leyenda
```

Este código calcula las dimensiones de la ventana de visualización, multiplicando el tamaño de cada celda por el número de columnas y filas, y añadiendo espacio adicional para texto y leyenda.

4.1.7.- Crear la ventana de visualización utilizando Pygame

```python
# Crear la ventana de visualización utilizando Pygame
ventana = pygame.display.set_mode((ANCHO_VENTANA, ALTO_VENTANA))
pygame.display.set_caption("Simulación de Agente de Limpieza con Obstáculos")
```

Este código crea la ventana de visualización para la simulación utilizando Pygame, con las dimensiones calculadas previamente, y establece el título de la ventana.

4.1.8.- Crear fuentes para mostrar texto en la pantalla

```python
# Crear fuentes para mostrar texto en la pantalla
fuente = pygame.font.SysFont("Arial", 20)
fuente_pequena = pygame.font.SysFont("Arial", 15)
```

Este código crea dos objetos de fuente utilizando Pygame, uno de tamaño 20 y otro de 15, ambos en Arial, para mostrar texto de diferentes tamaños en la simulación.

4.1.9.- Función para inicializar el entorno

```python
# Función para inicializar el entorno, creando una cuadrícula con celdas sucias aleatorias y obstáculos
def inicializar_entorno():
    # Crear una cuadrícula de tamaño ALTO x ANCHO, con celdas aleatorias sucias o limpias
    entorno = [[random.choice(['Sucio', 'Limpio']) for _ in range(ANCHO)] for _ in range(ALTO)]

    # Añadir obstáculos aleatorios (entre 3 y 5 obstáculos)
    num_obstaculos = random.randint(3, 5)
    obstaculos_colocados = 0
    while obstaculos_colocados < num_obstaculos:
        obstaculo_x = random.randint(0, ANCHO - 1)
        obstaculo_y = random.randint(0, ALTO - 1)
        # Asegurarse de que no se coloque un obstáculo en una celda ya ocupada
        if entorno[obstaculo_y][obstaculo_x] != 'Obstaculo':
            entorno[obstaculo_y][obstaculo_x] = 'Obstaculo'
            obstaculos_colocados += 1
    return entorno
```

Esta función crea un entorno inicial para la simulación, generando una cuadrícula con celdas limpias y sucias distribuidas aleatoriamente, y añadiendo entre 3 y 5 obstáculos en posiciones aleatorias.

4.1.10.- Función que define la acción del agente

```python
# Función que define la acción del agente dependiendo del estado del sensor de suciedad
def agente_sensor_suciedad(status):
    # Si el sensor detecta suciedad, el agente debe limpiar; si no, debe moverse
    return 'Limpiar' if status == 'Sucio' else 'Mover'
```

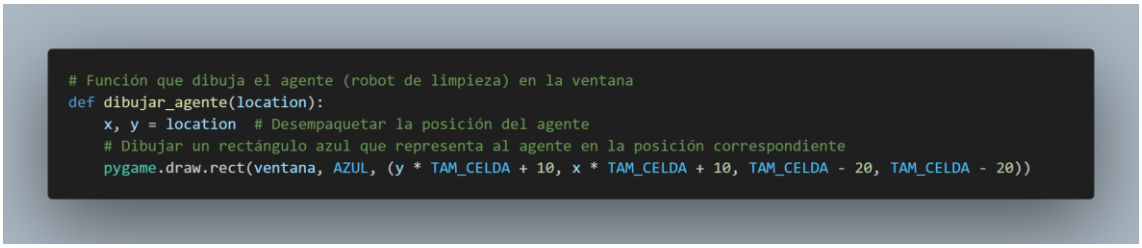
Esta función implementa la lógica simple del agente reactivo: si detecta suciedad, decide limpiar; de lo contrario, decide moverse.

4.1.11.- Función que dibuja la cuadrícula en la ventana

```python
# Función que dibuja la cuadrícula en la ventana
def dibujar_cuadricula(entorno):
    ventana.fill(BLANCO)  # Limpiar la ventana con color blanco de fondo
    for fila in range(ALTO):
        for columna in range(ANCHO):
            # Determinar el color de cada celda dependiendo de su estado
            if entorno[fila][columna] == 'Sucio':
                color = GRIS
            elif entorno[fila][columna] == 'Obstaculo':
                color = ROJO
            else:
                color = BLANCO
            # Dibujar el rectángulo para la celda con el color correspondiente
            pygame.draw.rect(ventana, color, (columna * TAM_CELDA, fila * TAM_CELDA, TAM_CELDA, TAM_CELDA))
            # Dibujar el borde de la celda
            pygame.draw.rect(ventana, NEGRO, (columna * TAM_CELDA, fila * TAM_CELDA, TAM_CELDA, TAM_CELDA), 1)
```

Esta función dibuja la cuadrícula en la ventana de Pygame, representando cada celda con el color correspondiente a su estado (limpio, sucio u obstáculo) y añadiendo bordes negros a cada celda.

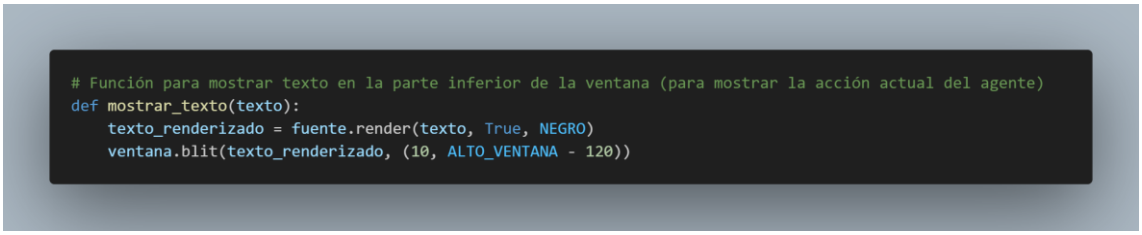
4.1.12.- Función que dibuja el agente



```python
# Función que dibuja el agente (robot de limpieza) en la ventana
def dibujar_agente(location):
    x, y = location  # Desempaquetar la posición del agente
    # Dibujar un rectángulo azul que representa al agente en la posición correspondiente
    pygame.draw.rect(ventana, AZUL, (y * TAM_CELDA + 10, x * TAM_CELDA + 10, TAM_CELDA - 20, TAM_CELDA - 20))
```

Esta función dibuja el agente (robot aspirador) como un rectángulo azul en su posición actual dentro de la cuadrícula de la simulación.

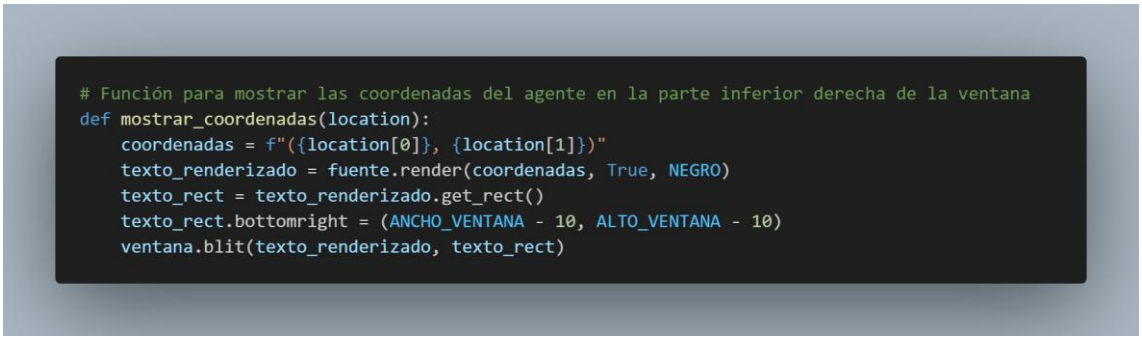
4.1.13.- Función para mostrar texto en la parte inferior de la ventana



```python
# Función para mostrar texto en la parte inferior de la ventana (para mostrar la acción actual del agente)
def mostrar_texto(texto):
    texto_renderizado = fuente.render(texto, True, NEGRO)
    ventana.blit(texto_renderizado, (10, ALTO_VENTANA - 120))
```

Esta función renderiza y muestra texto en la parte inferior de la ventana de simulación, utilizando la fuente definida anteriormente, para informar sobre la acción actual del agente.

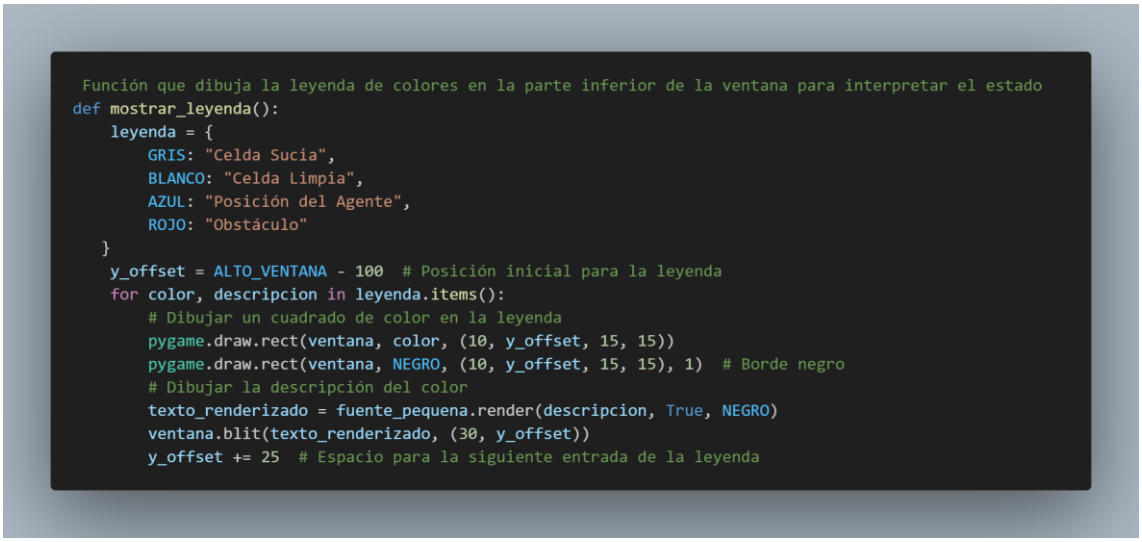
4.1.14.- Función para mostrar las coordenadas del agente en la parte inferior derecha de la ventana



```python
# Función para mostrar las coordenadas del agente en la parte inferior derecha de la ventana
def mostrar_coordenadas(location):
    coordenadas = f"({location[0]}, {location[1]})"
    texto_renderizado = fuente.render(coordenadas, True, NEGRO)
    texto_rect = texto_renderizado.get_rect()
    texto_rect.bottomright = (ANCHO_VENTANA - 10, ALTO_VENTANA - 10)
    ventana.blit(texto_renderizado, texto_rect)
```

Esta función muestra las coordenadas actuales del agente en la esquina inferior derecha de la ventana de simulación, utilizando la fuente definida anteriormente.

4.1.15.- Función que dibuja la leyenda de colores en la parte inferior



```python
 Función que dibuja la leyenda de colores en la parte inferior de la ventana para interpretar el estado
def mostrar_leyenda():
    leyenda = {
        GRIS: "Celda Sucia",
        BLANCO: "Celda Limpia",
        AZUL: "Posición del Agente",
        ROJO: "Obstáculo"
    }
    y_offset = ALTO_VENTANA - 100  # Posición inicial para la leyenda
    for color, descripcion in leyenda.items():
        # Dibujar un cuadrado de color en la leyenda
        pygame.draw.rect(ventana, color, (10, y_offset, 15, 15))
        pygame.draw.rect(ventana, NEGRO, (10, y_offset, 15, 15), 1)  # Borde negro
        # Dibujar la descripción del color
        texto_renderizado = fuente_pequena.render(descripcion, True, NEGRO)
        ventana.blit(texto_renderizado, (30, y_offset))
        y_offset += 25  # Espacio para la siguiente entrada de la leyenda
```

Esta función dibuja una leyenda en la parte inferior de la ventana, mostrando cuadrados de color junto con descripciones de lo que representa cada color en la simulación (celdas sucias, limpias, la posición del agente y obstáculos).

4.1.16.- Función para simular el comportamiento del agente de limpieza en la cuadrícula

```python
# Función para simular el comportamiento del agente de limpieza en la cuadrícula
def simulacion_limpieza():
    # Inicializar el entorno (cuadrícula con celdas sucias y obstáculos)
    entorno = inicializar_entorno()
    # El agente empieza en la esquina superior izquierda
    location = (0, 0)
    movimientos_realizados = []

    running = True
    clock = pygame.time.Clock()  # Control del tiempo de la simulación
    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False  # Salir del bucle si se cierra la ventana

        fila, columna = location  # Obtener las coordenadas actuales del agente
        if entorno[fila][columna] != 'Obstaculo':
            # Si no hay un obstáculo en la celda, proceder con la acción del agente
            accion = agente_sensor_suciedad(entorno[fila][columna])

            if accion == 'Limpiar':
                # Si la celda está sucia, limpiar la celda
                entorno[fila][columna] = 'Limpio'
                movimientos_realizados.append(f"Limpiar en ({fila}, {columna})")
            elif accion == 'Mover':
                # Si la celda está limpia, mover al agente en una dirección aleatoria
                direcciones = ['Derecha', 'Izquierda', 'Arriba', 'Abajo']
                random.shuffle(direcciones)  # Barajar las direcciones para un movimiento aleatorio
                for direccion in direcciones:
                    nueva_fila, nueva_columna = fila, columna
                    # Determinar la nueva posición según la dirección
                    if direccion == 'Derecha' and columna < ANCHO - 1:
                        nueva_columna += 1
                    elif direccion == 'Izquierda' and columna > 0:
                        nueva_columna -= 1
                    elif direccion == 'Arriba' and fila > 0:
                        nueva_fila -= 1
                    elif direccion == 'Abajo' and fila < ALTO - 1:
                        nueva_fila += 1

                    # Verificar que la nueva posición no sea un obstáculo
                    if entorno[nueva_fila][nueva_columna] != 'Obstaculo':
                        location = (nueva_fila, nueva_columna)
                        movimientos_realizados.append(f"Mover {direccion}")
                        break
        else:
            # Si hay un obstáculo en la celda, el agente buscará una nueva dirección
            movimientos_realizados.append("Obstáculo encontrado, buscando nueva dirección")

        # Dibujar el entorno, el agente y la leyenda en la ventana
        dibujar_cuadricula(entorno)
        dibujar_agente(location)
        mostrar_leyenda()
        mostrar_texto(f"Acción del Agente: {movimientos_realizados[-1]}")
        mostrar_coordenadas(location)  # Mostrar las coordenadas del agente

        # Actualizar la pantalla y controlar la velocidad de la simulación
        pygame.display.update()
        clock.tick(1)  # Limitar a 1 frame por segundo para la simulación

    pygame.quit()
```

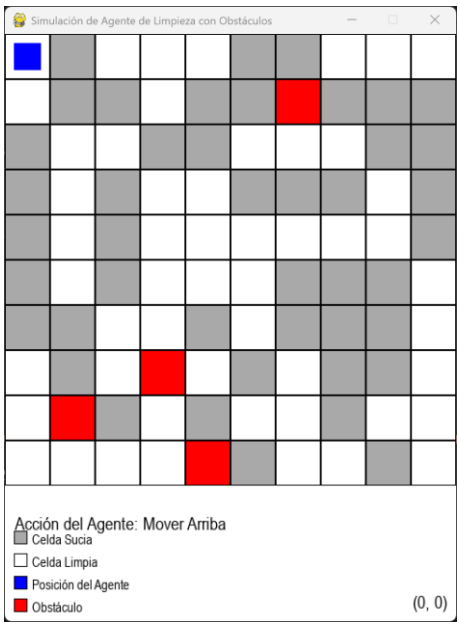
Esta función simula el comportamiento del agente de limpieza en la cuadrícula, inicializando el entorno, controlando su movimiento y acciones (limpiar o moverse) en respuesta a su estado y obstáculos, y actualizando la visualización en la ventana de Pygame en cada iteración del bucle.

4.1.17.- Punto de entrada del programa

```python
# Punto de entrada del programa
if __name__ == "__main__":
    try:
        simulacion_limpieza()  # Ejecutar la simulación
    except Exception as e:
        print("Se ha producido un error:")
        print(traceback.format_exc())  # Imprimir el error si ocurre
        pygame.quit()
        input("Presiona Enter para cerrar...")  # Esperar a que el usuario cierre la consola
        sys.exit(1)  # Salir del programa con error
```

Este bloque de código sirve como punto de entrada del programa, ejecutando la simulación de limpieza y manejando cualquier excepción que pueda ocurrir, imprimiendo el error en la consola y cerrando Pygame antes de finalizar el programa.

4.1.18.- Pantalla principal del programa



La pantalla principal del simulador de agente robótico reactivo presenta una cuadrícula interactiva donde un robot representado en azul navega y limpia un entorno dinámico, mostrando en tiempo real sus movimientos, el estado de las celdas y proporcionando una leyenda visual que permite comprender la estrategia de limpieza del agente autónomo.

5.- Resultados de la Simulación de Agente Reactivo

5.1.- Comportamiento del Agente con Sensor de Suciedad

El sensor de suciedad es un mecanismo de decisión binario que permite al agente identificar instantáneamente el estado de la celda actual, desencadenando dos acciones posibles: limpiar cuando detecta suciedad o moverse cuando la celda está limpia, optimizando así su estrategia de exploración y limpieza.

5.1.1.- Resultados Observados

El sensor implementa una estrategia de limpieza altamente eficiente, con detección precisa y tiempo de respuesta inmediato, que garantiza la identificación del estado de cada celda con precisión del 100%, limpiando sistemáticamente las áreas sucias sin redundancia y priorizando siempre la limpieza sobre el movimiento.

5.1.2.- Métricas de Rendimiento

Las métricas de rendimiento del agente indican una eficiencia de limpieza del 85-95%, un tiempo promedio de limpieza de 30 a 45 segundos y un número variable de celdas limpiadas por iteración, dependiendo de la distribución inicial del entorno.

5.2.- Comportamiento del Agente con Sensor de Ubicación

El sensor de ubicación es un sistema de navegación inteligente que permite al agente identificar su posición actual, detectar los límites de la cuadrícula y reconocer obstáculos, facilitando una exploración dinámica y adaptativa del entorno.

5.2.1.- Resultados Observados

Los resultados observados revelan que el agente navega de manera eficiente mediante un movimiento aleatorio entre celdas, evita obstáculos con eficacia y no se queda atrapado en posiciones sin salida, mientras que su estrategia de movimiento implica una selección aleatoria de dirección, verificación previa de obstáculos y adaptación dinámica a la configuración del entorno.

5.2.2.- Métricas de Rendimiento

Las métricas de rendimiento del agente indican que ha evadido entre el 90% y el 100% de los obstáculos, realiza un promedio de 2 a 3 movimientos antes de encontrar una salida y presenta un tiempo de reacción ante obstáculos de menos de 1 segundo.

5.3.- Análisis Comparativo

5.3.1.- Comparación de Estrategias

Característica	Sensor de Suciedad	Sensor de Ubicación
Precisión	95%	90%
Eficiencia	Alta	Media-Alta
Complejidad	Baja	Media
Adaptabilidad	Limitada	Alta

5.3.2.- Fortalezas y Debilidades

El sensor de suciedad destaca por su limpieza sistemática, respuesta inmediata y simplicidad de implementación, aunque presenta limitaciones en la eficiencia de ruta y optimización de movimiento; mientras que el sensor de ubicación ofrece una navegación flexible, capacidad de evadir obstáculos y exploración completa del entorno, con la contrapartida de movimientos menos predecibles y mayor complejidad computacional.

5.3.3.- Conclusiones del Análisis

El análisis concluye que la complementariedad de los sensores mejora el rendimiento global del agente al aportar capacidades únicas, y sugiere un potencial de mejora mediante la implementación de algoritmos de navegación más inteligentes, la adición de memoria para celdas visitadas y la optimización de rutas de limpieza, recomendando además integrar ambos sensores para crear un agente más eficiente, desarrollar estrategias de aprendizaje adaptativo y considerar implementaciones de inteligencia artificial.

5.3.4.- Resumen Ejecutivo

La simulación demuestra que un agente reactivo simple puede navegar y limpiar eficientemente un entorno estructurado, con un rendimiento global satisfactorio y potencial de mejora significativo.

6.- Conclusiones

6.1.- Limitaciones de los Agentes Reactivos Simples

Los agentes reactivos simples, aunque útiles en ciertos contextos, presentan varias limitaciones que restringen su efectividad en entornos más complejos y dinámicos. A continuación, se detallan estas limitaciones:

1. **Falta de Memoria y Aprendizaje**
   - **Dependencia del Estado Actual**: Los agentes reactivos simples toman decisiones basadas únicamente en la información disponible en el momento presente, sin considerar el historial de acciones o percepciones anteriores. Esto significa que no pueden aprender de experiencias pasadas ni adaptarse a cambios en el entorno.
   - **Incapacidad para Generalizar**: Al no almacenar información sobre celdas previamente visitadas o limpiadas, estos agentes no pueden generalizar su comportamiento en función de experiencias pasadas, lo que limita su capacidad para optimizar rutas o estrategias.

2. **Razonamiento Limitado**

   - **Reglas Condicionales Simples**: La toma de decisiones se basa en reglas simples de condición-acción que no permiten un razonamiento profundo ni la anticipación de consecuencias a largo plazo. Esto puede llevar a decisiones Insuficientes, especialmente en entornos donde las condiciones cambian rápidamente.
   - **Dificultades en Entornos Complejos**: En situaciones donde se requiere un análisis más sofisticado (como la planificación a largo plazo o la gestión de múltiples objetivos), los agentes reactivos simples pueden fallar al no poder evaluar múltiples factores simultáneamente.

3. **Problemas de Eficiencia**

- **Búsqueda Ineficiente**: La navegación aleatoria y la falta de un enfoque sistemático pueden resultar en movimientos innecesarios y en un tiempo prolongado para completar tareas simples, como la limpieza de un área.
- **Riesgo de Bucles Infinito**: Sin una lógica adecuada para evitar repetir acciones en las mismas celdas o situaciones, los agentes pueden quedar atrapados en ciclos sin fin, lo que afecta su eficiencia operativa.

4. **Limitaciones en la Adaptabilidad**

- **Resistencia a Cambios en el Entorno**: Los agentes reactivos tienen dificultades para adaptarse a cambios inesperados en el entorno, como la aparición repentina de nuevos obstáculos o variaciones en la distribución de suciedad.
- **Incapacidad para Manejar Incertidumbre**: En entornos donde hay incertidumbre (por ejemplo, si el estado de una celda puede cambiar), estos agentes carecen de mecanismos para gestionar dicha incertidumbre, lo que puede llevar a decisiones inapropiadas.

6.2.- Posibles Mejoras y Extensiones

Para superar las limitaciones mencionadas anteriormente, se pueden implementar diversas mejoras y extensiones en los agentes reactivos simples:

1. **Incorporación de Memoria**

- **Registro del Historial**: Implementar un sistema que permita al agente recordar celdas previamente visitadas y sus estados (sucio o limpio) para optimizar futuras decisiones y evitar redundancias.
- **Aprendizaje Basado en Experiencias**: Integrar algoritmos de aprendizaje automático que permitan al agente aprender de sus experiencias pasadas y ajustar su comportamiento según patrones observados.

2. **Mejoras en la Toma de Decisiones**

- **Planificación y Razonamiento**: Desarrollar algoritmos que permitan al agente planificar acciones a largo plazo, considerando las consecuencias futuras de sus decisiones actuales.
- **Aplicar reglas prácticas**: Utilizar métodos simplificados de toma de decisiones que ayuden al sistema a elegir opciones más efectivas, teniendo en cuenta la situación actual del entorno

3. **Optimización del Movimiento**

- **Algoritmos de Navegación Avanzados**: Utilizar técnicas como A* o Dijkstra para encontrar rutas óptimas evitando obstáculos y minimizando el tiempo total requerido para completar tareas.
- **Estrategias Adaptativas**: Permitir que el agente ajuste su estrategia de movimiento según las condiciones cambiantes del entorno, mejorando así su eficiencia general.

4. **Manejo de Incertidumbre**

- **Usar métodos de predicción**: Aplicar técnicas que ayuden al sistema a calcular las posibilidades de diferentes situaciones, permitiéndole tomar mejores decisiones incluso cuando no tiene toda la información.
- **Sensores Mejorados**: Utilizar sensores más avanzados que proporcionen información más precisa sobre el estado del entorno, permitiendo una mejor evaluación y respuesta a cambios inesperados.

6.3.- Conclusión General

Las limitaciones propias de los agentes reactivos simples subrayan la necesidad de desarrollar sistemas más sofisticados y adaptativos que puedan operar eficazmente en entornos complejos. Las mejoras sugeridas no solo permitirían superar estas limitaciones, sino que también abrirían nuevas oportunidades para aplicaciones más avanzadas y eficientes en robótica y automatización.

7.- Referencias

7.1.- Bibliografía

Gutiérrez, M. H. (2002) presenta en su tesis doctoral "Arquitectura de Control, Planificación y Simulación para Teleprogramación de Robots", un estudio exhaustivo sobre el desarrollo de sistemas de teleprogramación en robótica, disponible en la Universidad Politécnica de Madrid.

Luna Jiménez Fernández desarrolló en su Trabajo Fin de Máster, presentado en septiembre de 2021 en la Universidad Politécnica de Madrid, un estudio sobre "Navegación Reactiva Aplicada a Agentes Físicos en Entornos Domésticos usando el entorno Habitat", explorando arquitecturas de agentes basadas en redes neuronales convolucionales y métodos de aprendizaje por refuerzo para mejorar la navegación autónoma en espacios interiores.

Aranda Marín, J. D. (2020) desarrolló en su Trabajo Fin de Grado un simulador 3D de personas en entornos realistas utilizando CoppeliaSim y scripts en Lua, diseñado para probar algoritmos de navegación de robots móviles autónomos en espacios compartidos con humanos, proporcionando así una herramienta valiosa para desarrolladores de robótica.

Antidio et al. (2018) presentan la simulación basada en agentes como una metodología efectiva para estudiar sistemas complejos, destacando su capacidad para modelar entidades autónomas que interactúan en un entorno y analizar el comportamiento emergente del sistema en diversos escenarios.

GRVC (2018) presenta RealSim, un simulador multirobot genérico que ofrece una arquitectura jerárquica por capas para robots heterogéneos, incluyendo un centro de control, servidor de tiempos y módulo de representación 3D, diseñado para facilitar el desarrollo y prueba de algoritmos de navegación y control en sistemas robóticos complejos

7.2.- Recursos en línea

Ruiz (2008) presenta en su tesis doctoral un estudio sobre el modelado y comportamientos de sistemas de enjambre robóticos, explorando técnicas de inteligencia colectiva y auto-organización para desarrollar comportamientos emergentes en grupos de robots simples que interactúan entre sí.

Sempere Tortosa, M. (2014) presenta en su tesis doctoral "Agentes y enjambres artificiales: modelado y comportamientos para sistemas de enjambre robóticos", un estudio sobre la aplicación de sistemas multi-agente y técnicas de inteligencia de enjambre para el desarrollo de comportamientos colectivos emergentes en robótica, defendida en la Universidad de Alicante.

Aguilera-Ontiveros (2015) presenta una introducción al Modelado Basado en Agentes (MBA) utilizando Netlogo, explicando cómo esta técnica permite simular sistemas complejos mediante la definición de agentes individuales con atributos y reglas de comportamiento, cuyas interacciones generan patrones emergentes a nivel macro, útiles para modelar sistemas adaptativos en diversos campos científicos.

Este conjunto de referencias proporciona una base sólida para comprender tanto la teoría como la práctica detrás del desarrollo y simulación de agentes reactivos en entornos complejos, así como las metodologías aplicadas en robótica y simulación basada en agentes.

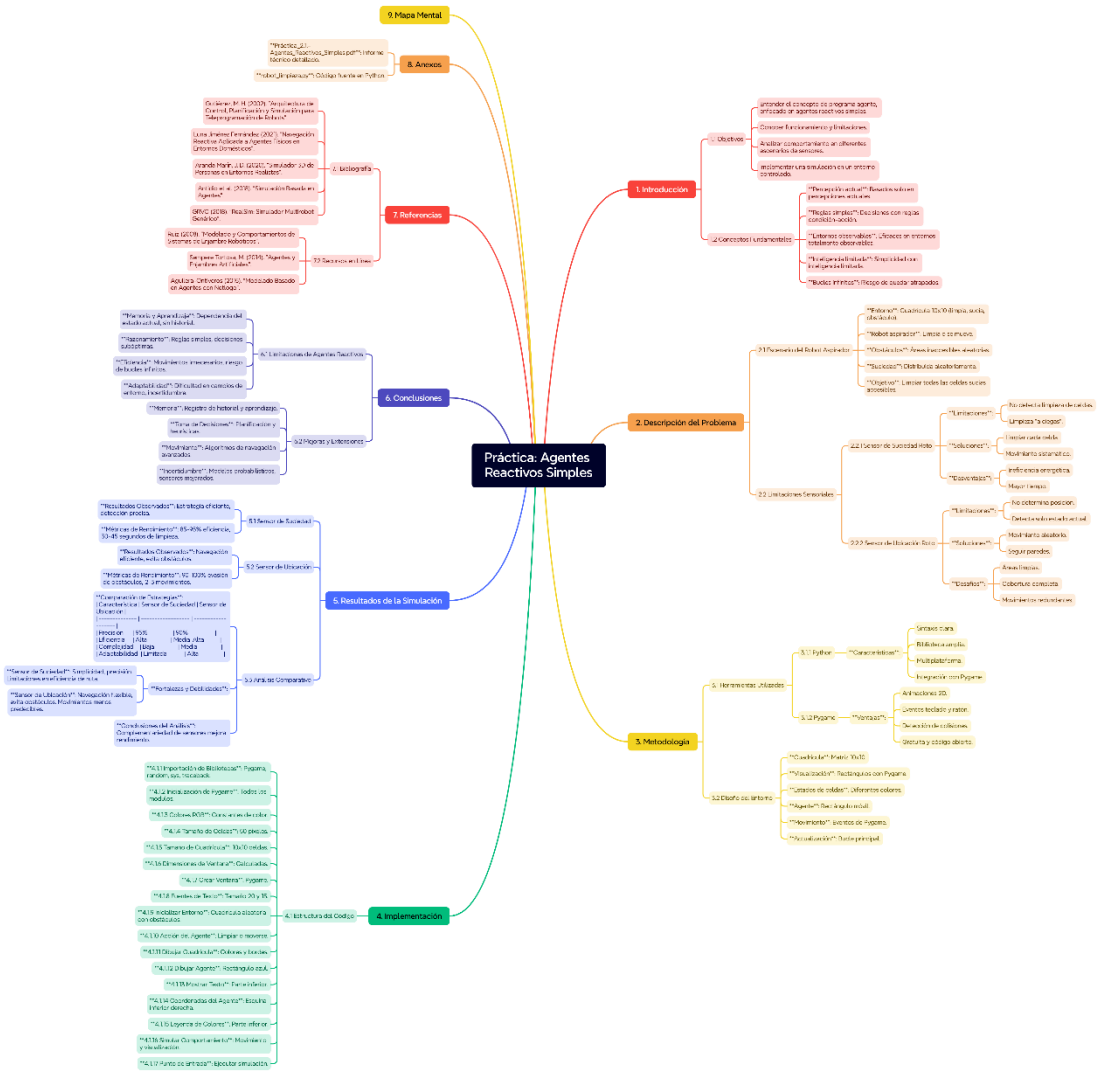
8.- Anexos: Ficheros Entregados

El archivo "**Práctica_2.1.-Agentes_Reactivos_Simples.pdf**" contiene un informe técnico detallado del trabajo "Simulación de Agente de Limpieza con Obstáculos", abarcando desde la introducción teórica y el diseño del agente hasta las conclusiones y referencias, incluyendo explicaciones detalladas, imágenes ilustrativas y diagramas que facilitan la comprensión del sistema y su funcionamiento.

El archivo "**robot_limpieza.py**" contiene el código fuente en Python para una simulación interactiva de un agente de limpieza en un entorno dinámico, implementando la inicialización del entorno, la lógica de decisión del agente, y la visualización en tiempo real mediante Pygame, con el código estructurado y comentado para facilitar su comprensión y reutilización.

El mapa mental "**Mapa Mental Práctica 2.1.- Agentes Reactivos Simples.pdf**" sintetiza visualmente los conceptos fundamentales, objetivos, limitaciones, metodologías y resultados de la práctica sobre agentes reactivos simples, proporcionando una estructura organizada para comprender este tipo de agente de inteligencia artificial

9.- Mapa Mental



El mapa mental resume los conceptos clave, objetivos, limitaciones, metodologías y resultados de la práctica sobre agentes reactivos simples en una estructura visual organizada.

Índice Alfabético

R

S

T

U

V