Temas Tratados en el Trabajo Práctico 1

- Diferencia entre Inteligencia e Inteligencia Artificial.
- Concepto de omnisciencia, aprendizaje y autonomía.
- Definición de Agente y sus características. Clasificación de Agentes según su estructura.
- Identificación y categorización del Entorno de Trabajo en tabla REAS.
- Caracterización del Entorno de Trabajo.

Anotaciones

"Acordarse de la definición de agente"

Ejercicios Teóricos

1. Defina con sus propias palabras inteligencia natural, inteligencia artificial y agente.

Inteligencia natural: capacidad de los seres vivos (especialmente nosotros los humanos) para aprender, razonar, resolver problemas y adaptarse al entorno. Surge de procesos biológicos y tiene cierto factor emocional y subjetivo.

Inteligencia artificial (IA): Capacidad de un sistema computacional para realizar tareas que normalmente requieren inteligencia humana, como reconocer patrones, tomar decisiones y aprender de la experiencia. Surge de procesos algorítmicos y carece de factor emocional y subjetivo.

Agente: Es una entidad (física o virtual) que percibe su entorno mediante sensores y actúa sobre dicho entorno mediante actuadores, siguiendo un objetivo o meta.

1. ¿Qué es un agente racional?

Un agente racional es aquel que, para cada posible secuencia de percepciones, elige la acción que maximiza su rendimiento esperado, considerando la información disponible y sus capacidades.

1. ¿Un agente es siempre una computadora?

No siempre. Puede ser un software, un robot físico o incluso un ser biológico (como los seres humanos). Lo importante es que cumpla con el ciclo percibir (mediante sensores) → razonar/decidir → actuar (mediante actuadores). Ejemplo:

Software: Buscador web.

Hardware: Robot aspiradora.

Biológico: Abeja buscando néctar.

1. Defina Omnisciencia, Aprendizaje y Autonomía.

Omnisciencia: Capacidad (teórica) de conocer todo lo que ocurre en el entorno, tanto pasado como presente. No es realista en agentes prácticos dado que ni el propio ser humano tiene dicha capacidad

Aprendizaje: Proceso por el cual un agente mejora su desempeño mediante la experiencia viendo asi afectados sus parámetros internos

Autonomía: Grado en que un agente puede operar sin intervención humana directa, usando su propio conocimiento para decidir acciones.

1. Defina cada tipo de agente en función de su **estructura** y dé un ejemplo de cada categoría.

Agentes reactivos simples Estructura: No almacenan historial ni modelo del mundo. Responden únicamente a la percepción inmediata con una acción predefinida (regla condicional "si… entonces…").

Ventaja: Son rápidos y simples.

Desventaja: No pueden planificar ni adaptarse a cambios que no estén contemplados en las reglas.

Ejemplo: Termostato que enciende la calefacción si la temperatura baja de 18 °C.

Agentes reactivos basados en modelos Estructura: Mantienen un modelo interno (estado del mundo) para compensar la información que no está disponible en el momento.

Ventaja: Pueden tomar mejores decisiones con información incompleta.

Desventaja: El modelo puede volverse inexacto si no se actualiza correctamente.

Ejemplo: Robot aspiradora que recuerda qué zonas ya limpió.

Agentes basados en objetivos Estructura: Deciden acciones evaluando si los acercan a una meta concreta, no solo en base a condiciones instantáneas.

Ventaja: Mayor flexibilidad; pueden adaptarse a cambios para alcanzar la meta.

Desventaja: Mayor costo computacional en la toma de decisiones.

Ejemplo: GPS que calcula rutas para llegar a un destino.

Agentes basados en la utilidad Estructura: No solo buscan alcanzar un objetivo, sino elegir la acción que maximice una función de utilidad (beneficio o satisfacción).

Ventaja: Permite priorizar entre varias metas o caminos posibles.

Desventaja: Requiere definir y cuantificar la utilidad, lo que no siempre es fácil.

Ejemplo: Sistema de recomendación que sugiere la película con mayor probabilidad de que te guste.

Agentes que aprenden Estructura: Ajustan su comportamiento mediante la experiencia, mejorando su desempeño en el tiempo. Pueden combinarse con cualquiera de las arquitecturas anteriores.

Ventaja: Adaptativos a entornos cambiantes.

Desventaja: Necesitan datos, tiempo de entrenamiento y pueden cometer errores durante el aprendizaje.

Ejemplo: Asistente de voz que mejora el reconocimiento cuanto más lo usas.

1. Para los siguientes entornos de trabajo indique sus **propiedades**:

a. Una partida de ajedrez.

Tomamos una partida normal entre 2 personas

Observabilidad: Total (tablero visible en todo momento) Determinismo: Determinista (no hay azar) Episodicidad: Secuencial (cada jugada influye en las siguientes) Agentes: Múltiples (dos jugadores). Dinamismo: Estático (no cambia mientras se piensa) Discretización: Discreto (movimientos y posiciones definidos)

b. Un partido de baloncesto.

Un partido de baloncesto de 5 vs 5 visto desde el punto de vista de los jugadores.

Observabilidad: Parcial (no se puede percibir todo al mismo tiempo desde la cancha) Determinismo: Estocástico (rebotes, errores, decisiones humanas) Episodicidad: Secuencial (cada jugada influye en el resto) Agentes: Múltiples (dos equipos + árbitros) Dinamismo: Dinámico (el juego cambia constantemente, especialmente la ubicación de la pelota y los jugadores) Discretización: Continuo (movimientos, posiciones y tiempos son variables con infinitos valores)

c. El juego Pacman.

Observabilidad: Total (siempre se ve todo el mapa) Determinismo: Parcialmente estocástico (movimiento de fantasmas si bien está programado puede ser interpretado como aleatorio) Episodicidad: Secuencial Agentes: Múltiples (Pacman + fantasmas) Dinamismo: Dinámico (los fantasmas se mueven mientras decidimos por donde movernos) Discretización: Discreto (posiciones en cuadrícula)

d. El truco.

Entendido como una partida normal entre pares de personas

Observabilidad: Parcial (cartas ocultas) Determinismo: Estocástico (depende de cartas repartidas al inicio) Episodicidad: Secuencial (depende de que se cante o tire primero) Agentes: Múltiples (2-4-6 jugadores) Dinamismo: Estático (el estado no cambia si nadie canta o tira) Discretización: Discreto (jugadas definidas)

e. Las damas.

Una partida clásica entre 2 personas

Observabilidad: Total Determinismo: Determinista Episodicidad: Secuencial Agentes: Múltiples (dos jugadores) Dinamismo: Estático Discretización: Discreto

f. El juego tres en raya.

Partida clásica entre 2 jugadores

Observabilidad: Total Determinismo: Determinista Episodicidad: Secuencial Agentes: Múltiples Dinamismo: Estático Discretización: Discreto

g. Un jugador de Pokémon Go.

Tomamos como modelo un jugador promedio

Observabilidad: Parcial (no conoce todas las ubicaciones y eventos) Determinismo: Estocástico (aparición aleatoria de Pokémon) Episodicidad: Secuencial (basado en lo que puedes hacer en determinados eventoss o ya habiendo obtenido Pokemons) Agentes: Múltiples (otros jugadores y servidores del juego) Dinamismo: Dinámico (cambia según ubicación y tiempo real) Discretización: Continuo (mapa geográfico real, y por lo tanto con infinitas ubicaciones posibles)

h. Un robot explorador autónomo de Marte.

Robot explorador único.

Observabilidad: Parcial (sensores limitados) Determinismo: Parcialmente estocástico (condiciones del terreno, fallos mecánicos) Episodicidad: Secuencial Agentes: Único (aunque podría coordinarse con otros) Dinamismo: Dinámico (cambia por clima o desplazamientos) Discretización: Continuo (movimiento y entorno físico, infinitas posiciones)

1. Elabore una tabla REAS para los siguientes entornos de trabajo:

	Rendimiento	Entorno	Actuadores	Sensores
Crucigrama	Número de palabras completadas, exactitud, tiempo empleado	Tablero del crucigrama, pistas dadas	Lápiz/teclado (escribir letras)	Vista para leer pistas y tablero
Taxi circulando	Tiempo de llegada, seguridad, cumplimiento de normas, satisfacción del pasajero	Calles, semáforos, tráfico, clima, pasajeros	Volante, acelerador, freno, luces, intermitentes	GPS, velocímetro, cámaras, sensores de proximidad, micrófono (para indicaciones)
Robot clasificador de piezas	Porcentaje de piezas correctamente clasificadas, velocidad de trabajo	Cinta transportadora, piezas con distintas formas y colores	Brazo robótico, mecanismo de empuje/clasificación	Cámara de visión artificial, sensores de forma/color

Ejercicios Prácticos

1. La Hormiga de Langton es un agente capaz de modificar el estado de la casilla en la que se encuentra para colorearla o bien de blanco o de negro. Al comenzar, la ubicación de la hormiga es una casilla aleatoria y mira hacia una de las cuatro casillas adyacentes. Si...

- ... la casilla sobre la que está es blanca, cambia el color del cuadrado, gira noventa grados a la derecha y avanza un cuadrado.
- ... la casilla sobre la que está es negra, cambia el color del cuadrado, gira noventa grados a la izquierda y avanza un cuadrado.

Caracterice el agente con su tabla REAS y las propiedades del entorno para después programarlo en Python:

¿Observa que se repite algún patrón? De ser así, ¿a partir de qué iteración?

```
Propiedades del entorno:

Observabilidad: Total (con conocimiento de todo el tablero).

Determinismo: Determinista (las reglas no incluyen azar).

Episodicidad: Secuencial (cada acción depende del estado previo).

Agentes: Único.

Dinamismo: Estático (no cambia sin que actúe la hormiga).

Discretización: Discreto (celdas en cuadrícula).


Patrón repetitivo: después de unas 10,000 iteraciones aparece un
patrón llamado highway que se repite indefinidamente.



################################## INFORME DE LA HORMIGA DE LANGTON
##################################

iostream> – Entrada y salida estándar.

<vector> – Uso de matrices dinámicas bidimensionales.

<thread> y <chrono> – Control del retardo de ejecución para animación.

---

1. Objetivo

Implementar en C++ una simulación de la Hormiga de Langton, un
autómata celular bidimensional que sigue reglas simples para recorrer
```

un tablero, generando patrones emergentes complejos.

---

2. Descripción del funcionamiento

Tablero:
Representado por una matriz grid de tamaño 20 filas × 40 columnas (vector<vector<bool>>).

false → Celda blanca ('.')

true → Celda negra ('#')

Posición inicial:
La hormiga comienza en el centro del tablero (x = COLUMNAS / 2, y = FILAS / 2).

Dirección inicial:
ARRIBA (enum Direccion).

Visualización:

A → posición actual de la hormiga.

. → celda blanca.

→ celda negra.

Ejecución en bucle infinito:

1. Mostrar tablero en la consola (system("clear") en Linux/Mac o "cls" en Windows).

2. Aplicar reglas de la Hormiga de Langton:

Si la celda es blanca → gira a la derecha, pinta negro.

Si la celda es negra → gira a la izquierda, pinta blanco.

3. Mover la hormiga una celda en la dirección actual (con efecto “wrap-around” en los bordes).

4. Pausar 250 ms para que el usuario pueda ver el movimiento.

---

3. Reglas implementadas

Reglas estándar de la Hormiga de Langton:

1. Blanco → Derecha → Negro
(dir + 1) % 4

2. Negro → Izquierda → Blanco
(dir + 3) % 4

---

4. Características adicionales

El tablero no tiene bordes finitos, sino que la hormiga reaparece al otro lado si cruza el límite (movimiento toroidal).

Uso de enumeraciones (enum Direccion) para mayor legibilidad.

Uso de std::vector para representar la matriz, lo que permite escalabilidad.

Control del tiempo de simulación con this_thread::sleep_for.

---

5. Posibles mejoras

Permitir que el tamaño de la grilla se ingrese por teclado.

Ajustar la velocidad de animación dinámicamente.

Usar colores en la consola para diferenciar más claramente celdas blancas y negras.

Guardar el estado final del tablero en un archivo.

También de cierta forma limitamos el tamaño de la grilla por cuestiones de rendimiento, lo cual hace que el juego termine sin necesariamente llegar a la convergencia

```
############################## CODIGO DE LA HORMIGA DE LANGTON
####################################

#include <iostream>
#include <vector>
#include <thread>
#include <chrono>
```

```
using namespace std;
bool endgame=false;

enum Direccion { ARRIBA, DERECHA, ABAJO, IZQUIERDA };

int main() {
    const int FILAS = 40;
    const int COLUMNAS = 80;
    vector<vector<bool>> grid(FILAS, vector<bool>(COLUMNAS, false));
// false=blanco, true=negro

    int x = COLUMNAS / 2;
    int y = FILAS / 2;
    Direccion dir = ARRIBA;


    while (endgame == false) {
        // Mostrar grilla
        system("clear"); // en Linux/Mac
        for (int i = 0; i < FILAS; i++) {
            for (int j = 0; j < COLUMNAS; j++) {
                if (i == y && j == x)
                    cout << 'A'; // Hormiga
                else
                    cout << (grid[i][j] ? '#' : '.'); // negro o
blanco
            }
            cout << '\n';
        }

        // Reglas de la hormiga
        if (!grid[y][x]) { // blanco
            dir = (Direccion)((dir + 1) % 4); // gira derecha
            grid[y][x] = true; // pinta negro
        } else { // negro
            dir = (Direccion)((dir + 3) % 4); // gira izquierda
            grid[y][x] = false; // pinta blanco
        }

        // Avanza (con borde que envuelve)
        switch (dir) {
            case ARRIBA:    y = (y - 1 + FILAS) % FILAS; break;
            case DERECHA:   x = (x + 1) % COLUMNAS; break;
            case ABAJO:     y = (y + 1) % FILAS; break;
            case IZQUIERDA: x = (x - 1 + COLUMNAS) % COLUMNAS; break;
        }

        if(x == 0){
            endgame=true;
            return 0;
```

```
        }
        if(y == 0){
            endgame=true;
            return 0;
        }
        if(x == COLUMNAS){
            endgame=true;
            return 0;
        }
        if(y == FILAS){
            endgame=true;
            return 0;
        }
        this_thread::sleep_for(chrono::milliseconds(5)); // más lento
para ver el cambio

    }

    return 0;
}
```

1. El Juego de la Vida de Conway consiste en un tablero donde cada casilla representa una célula, de manera que a cada célula le rodean 8 vecinas. Las células tienen dos estados: están *vivas* o *muertas*. En cada iteración, el estado de todas las células se tiene en cuenta para calcular el estado siguiente en simultáneo de acuerdo a las siguientes acciones:

- Nacer: Si una célula muerta tiene exactamente 3 células vecinas vivas, dicha célula pasa a estar viva.
- Morir: Una célula viva puede morir sobrepoblación cuando tiene más de tres vecinos alrededor o por aislamiento si tiene solo un vecino o ninguno.
- Vivir: una célula se mantiene viva si tiene 2 o 3 vecinos a su alrededor.

Caracterice el agente con su tabla REAS y las propiedades del entorno para después programarlo en Python:

```
Propiedades del entorno:

Observabilidad: Total (el estado de todas las celdas es visible).

Determinismo: Determinista (las reglas no incluyen azar).

Episodicidad: Secuencial (cada generación depende de la anterior).

Agentes: Único (aunque puede verse como múltiples células actuando en
paralelo).

Dinamismo: Dinámico (el estado cambia en cada iteración).
```

Discretización: Discreto (tiempo y espacio discretos).

############################## *INFORME DEL JUEGO DE LA VIDA* ##############################

1. Objetivo del código
Este programa implementa una simulación interactiva del “Juego de la Vida” creado por John Conway. Es un autómata celular donde cada celda del tablero puede estar viva (1) o muerta (0) y evoluciona en pasos de tiempo siguiendo cuatro reglas básicas basadas en el número de celdas vecinas vivas.
En este caso: - Se usa Pygame para mostrar la simulación gráficamente. - Se usa NumPy para manejar eficientemente las operaciones con matrices.

2. Herramientas y librerías utilizadas
Pygame: librería para gráficos 2D, manejo de eventos y dibujo.
NumPy: librería para operaciones numéricas y manejo de arreglos/matrices.

3. Estructura del código
A) Configuración inicial
Dimensiones de la ventana: WIDTH, HEIGHT.
Tamaño de celda: CELL_SIZE.
Cálculo de filas y columnas: ROWS, COLS.
Definición de colores en RGB: BLACK, WHITE, GRAY.
B) Inicialización de Pygame
pygame.init() para iniciar módulos.
display.set_mode() para crear la ventana.
Clock() para controlar la velocidad de refresco.
C) Tablero
Matriz NumPy grid con ceros (0 = muerta, 1 = viva).

4. Funciones principales
4.1. Dibujo del tablero
Fondo negro.
Celdas vivas en blanco.
Líneas de cuadrícula en gris.
4.2. Actualización del tablero
Aplica las reglas de Conway: 1. Celda viva con menos de 2 vecinos vivos → muere (soledad). 2. Celda viva con 2 o 3 vecinos vivos → sobrevive. 3. Celda viva con más de 3 vecinos vivos → muere (sobrepoblación). 4. Celda muerta con exactamente 3 vecinos vivos → nace.

5. Bucle principal
Control de ejecución con running y paused.
Eventos:
QUIT: cerrar ventana.
SPACE: pausar/reanudar.

```
X: finalizar.
Click izquierdo: pintar celdas vivas.
Pintado con el mouse detectando posición y activando celdas.
Actualización del tablero solo si no está pausado.
Dibujo y refresco con pygame.display.flip().

6. Interactividad
Espacio: Pausar/Reanudar.
Click izquierdo: Añadir células vivas.
Tecla X: Salir.
Cerrar ventana: Finaliza el programa.

7. Observaciones y mejoras posibles
Optimizar cálculo de vecinos.
Cargar patrones predefinidos.
Click derecho para borrar células.
Ajuste dinámico de velocidad.
Guardar/cargar estado del tablero.

Resumen: El código implementa una simulación interactiva del Juego de
la Vida con Pygame y NumPy, permitiendo al usuario modificar el
tablero en tiempo real y observar la evolución de las células según
las reglas de Conway.

############################## CODIGO DEL JUEGO DE LA VIDA
####################################

import pygame
import numpy as np

# --------------------------
# CONFIGURACIÓN DEL JUEGO
# --------------------------
WIDTH, HEIGHT = 800, 600
CELL_SIZE = 10
COLS = WIDTH // CELL_SIZE
ROWS = HEIGHT // CELL_SIZE

BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GRAY = (50, 50, 50)

# --------------------------
# INICIALIZACIÓN DE PYGAME
# --------------------------
pygame.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Juego de la Vida - Conway")
clock = pygame.time.Clock()
```

```
# --------------------------
# CREACIÓN DEL TABLERO
# --------------------------
grid = np.zeros((ROWS, COLS), dtype=int)

# --------------------------
# FUNCIÓN PARA DIBUJAR
# --------------------------
def draw_grid(surface, grid):
    surface.fill(BLACK)
    for row in range(ROWS):
        for col in range(COLS):
            if grid[row, col] == 1:
                pygame.draw.rect(
                    surface,
                    WHITE,
                    (col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE,
CELL_SIZE)
                )
    for x in range(0, WIDTH, CELL_SIZE):
        pygame.draw.line(surface, GRAY, (x, 0), (x, HEIGHT))
    for y in range(0, HEIGHT, CELL_SIZE):
        pygame.draw.line(surface, GRAY, (0, y), (WIDTH, y))

# --------------------------
# FUNCIÓN PARA ACTUALIZAR
# --------------------------
def update_grid(grid):
    new_grid = np.copy(grid)
    for row in range(ROWS):
        for col in range(COLS):
            neighbors = np.sum(
                grid[max(0, row-1):min(ROWS, row+2),
                     max(0, col-1):min(COLS, col+2)]
            ) - grid[row, col]

            if grid[row, col] == 1 and neighbors < 2:
                new_grid[row, col] = 0
            elif grid[row, col] == 1 and (neighbors == 2 or neighbors
== 3):
                new_grid[row, col] = 1
            elif grid[row, col] == 1 and neighbors > 3:
                new_grid[row, col] = 0
            elif grid[row, col] == 0 and neighbors == 3:
                new_grid[row, col] = 1
    return new_grid

# --------------------------
# BUCLE PRINCIPAL
# --------------------------
```

```
running = True
paused = True
mouse_held = False  # Variable para saber si el botón está presionado

while running:
    clock.tick(30)  # Aumenté a 30 FPS para que el pintado con el
mouse sea más fluido

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE:
                paused = not paused
            if event.key == pygame.K_x:
                print("Juego finalizado por usuario")
                running = False

        elif event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:  # Botón izquierdo
                mouse_held = True  # Marcamos que está presionado

        elif event.type == pygame.MOUSEBUTTONUP:
            if event.button == 1:  # Botón izquierdo
                mouse_held = False  # Dejamos de pintar

    # Si el mouse está presionado, dibujamos en la celda
correspondiente
    if mouse_held:
        mouse_x, mouse_y = pygame.mouse.get_pos()
        col = mouse_x // CELL_SIZE
        row = mouse_y // CELL_SIZE
        grid[row, col] = 1  # La marcamos como viva

    # Actualizamos el tablero solo si no está pausado
    if not paused:
        grid = update_grid(grid)

    draw_grid(screen, grid)
    pygame.display.flip()

pygame.quit()
```

Bibliografía

Russell, S. & Norvig, P. (2004) *Inteligencia Artificial: Un Enfoque Moderno*. Pearson Educación S.A. (2a Ed.) Madrid, España

Poole, D. & Mackworth, A. (2023) *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press (3a Ed.) Vancouver, Canada