Proyecto 1: Informe

Agente Inteligente para Navegación en Laberinto Dinámico

Sofia Castillo Giraldo - 2266149

Merly Velásquez Cortez - 2266016

Faculta de Ingeniera, Universidad del Valle

Proyecto Integrador I

Juan Pablo Pinillos Reina

24 de abril del 2025

1. Introducción

El propósito de este proyecto es desarrollar un agente inteligente capaz de navegar en un laberinto dinámico utilizando técnicas de búsqueda como Búsqueda en Amplitud (BFS), Búsqueda en Profundidad (DFS) y A*. El laberinto puede cambiar durante la ejecución debido a obstáculos móviles o cambios en la posición del objetivo, lo que requiere que el agente adapte su comportamiento dinámicamente.

Requisitos del Proyecto:

- Implementar algoritmos de búsqueda clásicos: BFS, DFS y A*.
- Adaptación dinámica del agente ante cambios en el entorno (obstáculos móviles, meta móvil).
- Interfaz gráfica obligatoria con animaciones paso a paso.
- Laberinto configurable mediante dimensiones variables y archivos JSON.

Alcance del Proyecto:

Este proyecto se enfoca en la implementación de técnicas de búsqueda clásicas: Búsquedas Informadas y no informadas y su adaptación a un entorno dinámico. No se incluyen métodos avanzados como aprendizaje automático o redes neuronales.

2. Representación del Problema

Modelo del Laberinto

El laberinto se representa como una matriz bidimensional (filas x columnas), donde cada celda puede tener uno de los siguientes estados:

0: Espacio vacío.
1: Obstáculo.
2: Posición inicial.
3: Posición objetivo.

Ejemplo de inicialización del laberinto:

```python
filas, columnas = 10, 15
laberinto = [[0 for _ in range(columnas)] for _ in range(filas)]
inicio = (0, 0)  # Posición inicial
objetivo = (filas - 1, columnas - 1)  # Posición objetivo
laberinto[inicio[0]][inicio[1]] = 2  # INICIO
laberinto[objetivo[0]][objetivo[1]] = 3  # OBJETIVO
```

Operadores*:*

El agente puede moverse hacia arriba, abajo, izquierda o derecha, siempre que la celda adyacente no sea un obstáculo.

```
def obtener_vecinos(posicion):
    x, y = posicion
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        nx, ny = x + dx, y + dy
        if 0 <= nx < filas and 0 <= ny < columnas and laberinto[nx][ny] != 1:
            yield (nx, ny)
```

Prueba de Meta

La prueba de meta verifica si la posición actual del agente coincide con la posición

```
if actual == objetivo:
    break
```

Costo de Ruta

El costo de moverse entre celdas adyacentes es uniforme (costo = 1).

3. Implementación

1. Estructura del Código

Estructura General:

El código fue organizado modularmente para facilitar su mantenimiento y extensión. Las principales funciones y estructuras utilizadas son:

- **Funciones de búsqueda:** Implementación de Búsqueda por amplitud, Búsqueda por profundidad, Costo Uniforme, A*, Avara.
- **Función híbrida:** Combinación de técnicas para adaptarse dinámicamente al entorno.
- **Funciones auxiliares:** Obtención de vecinos, cálculo de heurísticas y reconstrucción de rutas.
- **Interfaz gráfica:** Visualización del laberinto, animación del agente y panel lateral con instrucciones.

Búsqueda por amplitud (BFS)

```python
def busqueda_amplitud(inicio, objetivo, pantalla):
    cola = deque([inicio])  # Cola para BFS
    ruta_anterior = {inicio: None}
    while cola:
        actual = cola.popleft()
        if actual == objetivo:
            break
        for vecino in obtener_vecinos(actual):
            if vecino not in ruta_anterior:
                cola.append(vecino)
                ruta_anterior[vecino] = actual
```

- Explora todos los nodos en el mismo nivel antes de avanzar al siguiente nivel.
- Usa una cola (`deque`) para garantizar que los nodos se exploren en orden FIFO.
- Es óptima para encontrar la ruta más corta en un laberinto sin pesos.

Búsqueda por profundidad (DFS)

```python
def busqueda_profundidad(inicio, objetivo, pantalla):
    pila = [inicio]  # Pila para DFS
    ruta_anterior = {inicio: None}
    while pila:
        actual = pila.pop()
        if actual == objetivo:
            break
        for vecino in obtener_vecinos(actual):
            if vecino not in ruta_anterior:
                pila.append(vecino)
                ruta_anterior[vecino] = actual
```

- Explora tan profundamente como sea posible antes de retroceder.
- Usa una pila (lista) para explorar nodos en orden LIFO.
- No garantiza la ruta más corta, pero puede ser útil para explorar nuevas áreas rápidamente.

Búsqueda A*

```
def busqueda_a_estrella(inicio, objetivo, pantalla):
    frontera = []  # Cola de prioridad para A*
    heapq.heappush(frontera, (0, inicio))
    ruta_anterior = {inicio: None}
    costo_acumulado = {inicio: 0}
    while frontera:
        _, actual = heapq.heappop(frontera)
        if actual == objetivo:
            break
        for vecino in obtener_vecinos(actual):
            nuevo_costo = costo_acumulado[actual] + 1
            if vecino not in costo_acumulado or nuevo_costo < costo_acumulado[vecino]:
                costo_acumulado[vecino] = nuevo_costo
                prioridad = nuevo_costo + distancia_manhattan(vecino, objetivo)
                heapq.heappush(frontera, (prioridad, vecino))
                ruta_anterior[vecino] = actual
```

- Combina el costo acumulado (`g(n)`) con una heurística admisible (`h(n)`), como la distancia Manhattan.
- Usa una cola de prioridad (`heapq`) para seleccionar el siguiente nodo basado en f(n)=g(n)+h(n).
- Es ideal para laberintos grandes o cuando el objetivo está lejos.

Búsqueda por costo uniforme

```
def busqueda_costo_uniforme(inicio, objetivo, pantalla):
    frontera = []
    heapq.heappush(frontera, (0, inicio))  # (costo, posición)
    vino_desde = {inicio: None}
    costo_hasta_ahora = {inicio: 0}

    while frontera:
        pygame.time.delay(50)  # pausa para animar
        costo_actual, actual = heapq.heappop(frontera)

        if actual == objetivo:
            break

        for vecino in obtener_vecinos(actual):
            nuevo_costo = costo_hasta_ahora[actual] + 1  # costo uniforme: cada paso tiene costo 1
            if vecino not in costo_hasta_ahora or nuevo_costo < costo_hasta_ahora[vecino]:
                costo_hasta_ahora[vecino] = nuevo_costo
                prioridad = nuevo_costo  # La prioridad es solo el costo acumulado
                heapq.heappush(frontera, (prioridad, vecino))
                vino_desde[vecino] = actual
                # marcar como explorado
                if laberinto[vecino[0]][vecino[1]] not in [2, 3]:
                    dibujar_casilla(pantalla, vecino, 'visitado')
        pygame.display.flip()

    return reconstruir_ruta(vino_desde, inicio, objetivo)
```

- La búsqueda por costo uniforme garantiza encontrar el camino más corto en términos de costo acumulado, siempre que el costo de cada paso sea consistente.
- En un laberinto dinámico, esta técnica es útil cuando los costos de los pasos pueden variar (por ejemplo, si algunos caminos tienen obstáculos que aumentan el costo). Sin embargo, en este caso, todos los pasos tienen el mismo costo (1),

por lo que la búsqueda por costo uniforme actúa de manera similar a BFS (Búsqueda en Anchura).

Búsqueda avara

```python
def busqueda_avara(inicio, objetivo, pantalla):
    frontera = []
    heapq.heappush(frontera, (manhattan(inicio, objetivo), inicio))  # (heurística, posición)
    vino_desde = {inicio: None}

    while frontera:
        pygame.time.delay(50)  # pausa para animar
        _, actual = heapq.heappop(frontera)

        if actual == objetivo:
            break

        for vecino in obtener_vecinos(actual):
            if vecino not in vino_desde:
                prioridad = manhattan(vecino, objetivo)  # Solo usa la heurística
                heapq.heappush(frontera, (prioridad, vecino))
                vino_desde[vecino] = actual
                # marcar como explorado
                if laberinto[vecino[0]][vecino[1]] not in [2, 3]:
                    dibujar_casilla(pantalla, vecino, 'visitado')
        pygame.display.flip()

    return reconstruir_ruta(vino_desde, inicio, objetivo)
```

- La búsqueda avara prioriza los nodos que están "más cerca" del objetivo según la heurística (distancia Manhattan en este caso).
- Es una técnica rápida porque ignora el costo acumulado, pero no garantiza encontrar el camino óptimo. Puede quedar atrapada en mínimos locales o seguir rutas subóptimas si el laberinto tiene estructuras complejas.

Búsqueda Hibrida

```python
def busqueda_hibrida(inicio, objetivo, pantalla):
    print("Intentando BFS...")
    ruta = bfs(inicio, objetivo, pantalla)
    if ruta:
        print("Ruta encontrada con BFS.")
        return ruta

    print("BFS falló. Intentando A*...")
    ruta = a_estrella(inicio, objetivo, pantalla)
    if ruta:
        print("Ruta encontrada con A*.")
        return ruta

    print("A* falló. Intentando Búsqueda por Costo Uniforme...")
    ruta = busqueda_costo_uniforme(inicio, objetivo, pantalla)
    if ruta:
        print("Ruta encontrada con Búsqueda por Costo Uniforme.")
        return ruta

    print("Costo Uniforme falló. Intentando Búsqueda Avara...")
    ruta = busqueda_avara(inicio, objetivo, pantalla)
    if ruta:
        print("Ruta encontrada con Búsqueda Avara.")
        return ruta

    print("Búsqueda Avara falló. Intentando DFS...")
    ruta = dfs(inicio, objetivo, pantalla)
    if ruta:
        print("Ruta encontrada con DFS.")
        return ruta

    print("No se encontró una ruta válida con ninguna técnica.")
    return None
```

- La búsqueda híbrida es especialmente útil en laberintos dinámicos donde las condiciones pueden cambiar (por ejemplo, obstáculos móviles o cambios en la estructura del laberinto).
- Al probar diferentes algoritmos en secuencia, maximiza las posibilidades de encontrar una solución, incluso si algunos algoritmos fallan debido a las características del laberinto.
- Además, proporciona información sobre qué técnica encontró la ruta, lo que puede ser útil para analizar el rendimiento de cada algoritmo.

Interfaz Gráfica

La interfaz gráfica fue desarrollada usando Pygame, incluye:

- Una cuadrícula para representar el laberinto.
- Un robot representado como un círculo azul.
- Animaciones paso a paso de la exploración y el movimiento del robot.
- Un panel lateral con instrucciones claras.

Fragmento de código para dibujar el laberinto:

```
def dibujar_celda(pantalla, posicion, tipo_celda):
color = COLORES[tipo_celda] if isinstance(tipo_celda, int) else COLORES[tipo_celda]
pygame.draw.rect(pantalla, color, (posicion[1] * TAMANO_CELDA, posicion[0] * TAMANO_CELDA, TAMANO_CELDA, TAMANO_CELDA))
pygame.draw.rect(pantalla, (180, 180, 180), (posicion[1] * TAMANO_CELDA, posicion[0] *
TAMANO_CELDA, TAMANO_CELDA, TAMANO_CELDA), 1)
```

```
def dibujar_laberinto(pantalla, camino=None, pos_robot=None):
    for i in range(FILAS):
        for j in range(COLUMNAS):
            dibujar_casilla(pantalla, (i, j), laberinto[i][j])
    if camino:
        for pos in camino:
            if laberinto[pos[0]][pos[1]] not in [INICIO, META]:
                dibujar_casilla(pantalla, pos, 'camino')
    if pos_robot:
        # Dibujar robot como círculo
        x, y = pos_robot
        centro = (y * TAMAÑO_CASILLA + TAMAÑO_CASILLA // 2, x * TAMAÑO_CASILLA + TAMAÑO_CASILLA // 2)
        radio = TAMAÑO_CASILLA // 3
        pygame.draw.circle(pantalla, COLOR_ROBOT, centro, radio)
    pygame.display.flip()
```

2. Estructuras de Datos Utilizadas

Para representar el laberinto y gestionar las búsquedas, se utilizaron las siguientes estructuras de datos:

Matriz Bidimensional (laberinto):

- Representa el laberinto como una matriz de filas x columnas.
- Cada celda tiene un estado: 0 (vacío), 1 (obstáculo), 2 (inicio) o 3 (objetivo).

Fragmento de código:

```
filas, columnas = 10, 15
laberinto = [[0 for _ in range(columnas)] for _ in range(filas)]
inicio = (0, 0)
objetivo = (filas - 1, columnas - 1)
laberinto[inicio[0]][inicio[1]] = 2  # INICIO
laberinto[objetivo[0]][objetivo[1]] = 3  # OBJETIVO
```

Colas y Pilas:

BFS usa una cola (`deque`) para explorar nodos en orden FIFO.

DFS usa una pila (lista) para explorar nodos en orden LIFO.

A* usa una cola de prioridad (`heapq`) para seleccionar el siguiente nodo basado en la función f(n)=g(n)+h(n).

Fragmento de código:

```
# BFS usa una cola
cola = deque([inicio])

# DFS usa una pila
pila = [inicio]

# A* usa una cola de prioridad
frontera = []
heapq.heappush(frontera, (0, inicio))
```

Diccionarios:

- Diccionario ruta_anterior para almacenar el camino recorrido.
- Diccionario costo_acumulado para almacenar el costo acumulado en cada nodo.

Fragmento de código:

```
# Diccionario para almacenar el camino
ruta_anterior = {inicio: None}


# Diccionario para almacenar el costo acumulado
costo_acumulado = {inicio: 0}
```

3. Adaptación Dinámica

El agente adapta su comportamiento cuando detecta cambios en el laberinto, como obstáculos o una meta que cambia de posición. Si una técnica falla, intenta otra.

```
elif evento.key == pygame.K_m:  # Mover meta
    nombres_algoritmos.clear()
    tiempos.clear()
    mensajes.clear()
    laberinto[meta[0]][meta[1]] = VACIO  # Limpiar la antigua posición
    nueva_meta = (random.randint(0, FILAS - 1), random.randint(0, COLUMNAS - 1))
    meta = nueva_meta
    laberinto[meta[0]][meta[1]] = META  # Establecer nueva posición
```

Cambios en el Laberinto:

- El agente detecta cambios en el laberinto (obstáculos que aparecen/desaparecen, meta dinámica).
- Si el agente detecta que su ruta actual es inválida, recalcula la ruta usando la misma técnica o cambia a otra técnica.

Cambio de Técnica:

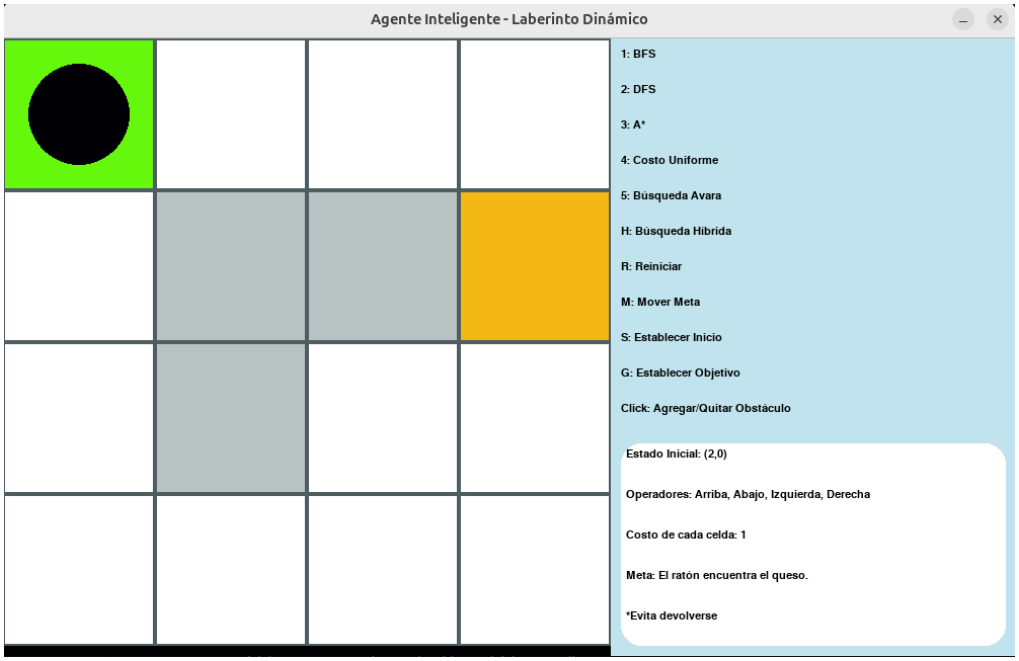
```
def hybrid_search(start, goal, screen):
    almacenamiento_mensajes("Intentando BFS...")
    path = bfs(start, goal, screen)
    if path:
        almacenamiento_mensajes("Ruta encontrada con BFS.")
        return path

    almacenamiento_mensajes("- BFS falló. Intentando A*...")
    path = a_star(start, goal, screen)
    if path:
        almacenamiento_mensajes("- Ruta encontrada con A*.")
        return path

    almacenamiento_mensajes("- A* falló. Intentando Búsqueda por Costo Uniforme...")
    path = uniform_cost_search(start, goal, screen)
    if path:
        almacenamiento_mensajes(":) Ruta encontrada con Búsqueda por Costo Uniforme.")
        return path

    almacenamiento_mensajes("- Costo Uniforme falló. Intentando Búsqueda Avara...")
    path = greedy_best_first_search(start, goal, screen)
    if path:
        almacenamiento_mensajes(":) Ruta encontrada con Búsqueda Avara.")
        return path

    almacenamiento_mensajes("- Búsqueda Avara falló. Intentando DFS...")
    path = dfs(start, goal, screen)
    if path:
        almacenamiento_mensajes(":) Ruta encontrada con DFS.")
        return path

    almacenamiento_mensajes(":( No se encontró una ruta válida con ninguna técnica.")
    return None
```

- Primero intenta BFS (Búsqueda en amplitud), que es eficiente para encontrar el camino más corto en laberintos sin costos variables.
- Si BFS falla, intenta A *, que combina costo uniforme y heurística para encontrar el camino óptimo.
- Si A* falla, intenta Búsqueda por Costo Uniforme, que es robusta, pero puede ser más lenta.
- Si todas las anteriores fallan, intenta Búsqueda Avara, que es rápida pero no garantiza optimalidad.
- Finalmente, intenta DFS (Búsqueda en Profundidad), que puede encontrar un camino, aunque no sea óptimo.

4. Pruebas y Resultados

1. Escenarios de Prueba

Describe varios escenarios de prueba y cómo el agente se comportó en cada uno.



- **Escenario 1: Laberinto Pequeño (4x4):**

Laberinto con pocas celdas y obstáculos mínimos. El objetivo está cerca del inicio.

Resultados:

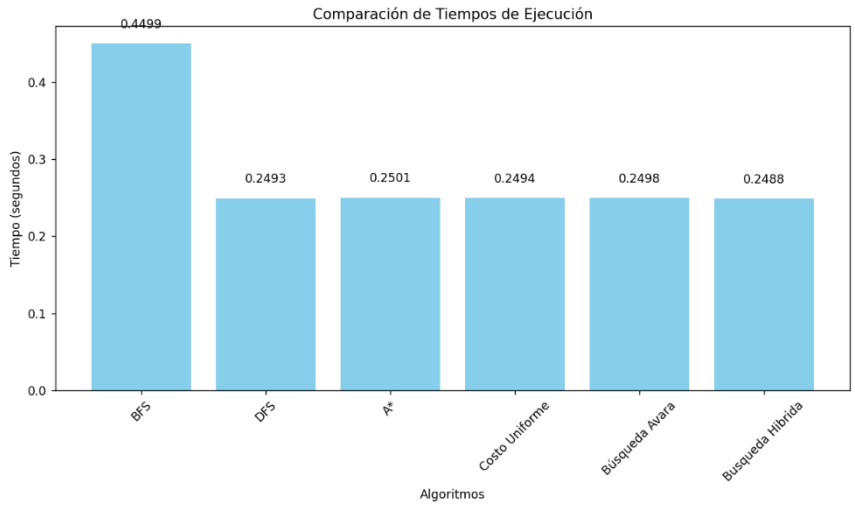
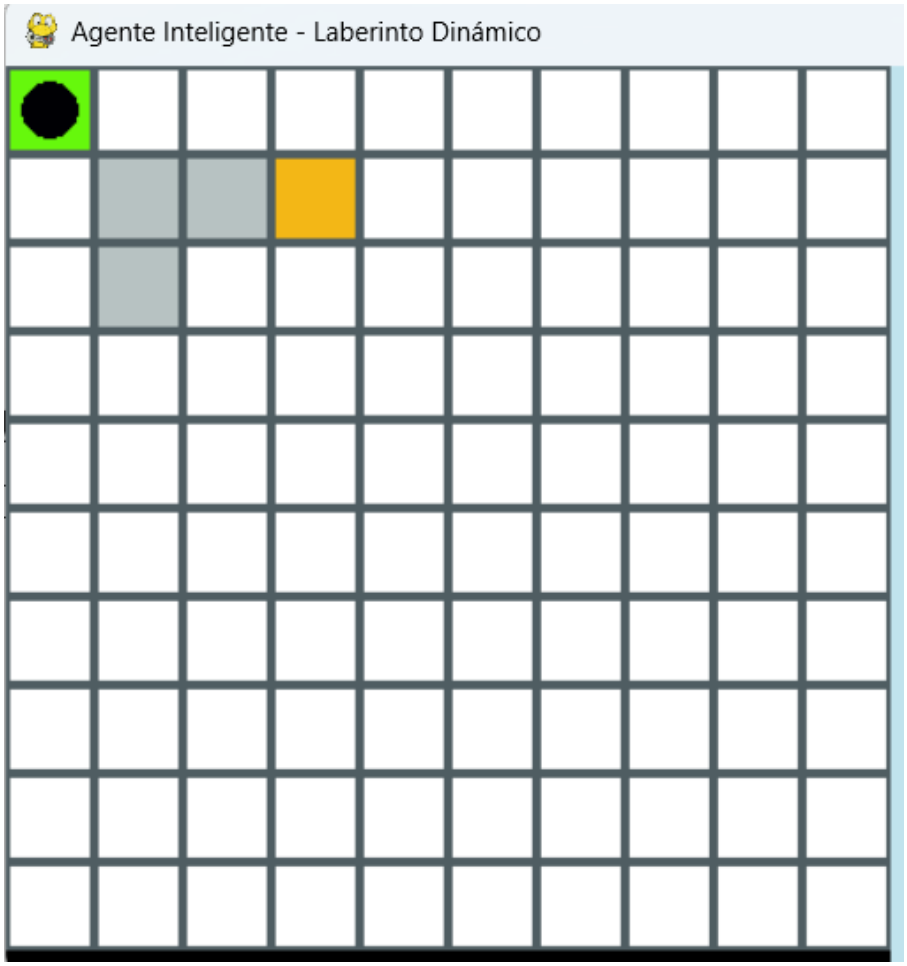


Fig. Comparación de los tiempos de ejecución de cada algoritmo, laberinto 4x4.

- **Escenario 2: Laberinto Grande (10x10):**



Laberinto con muchas celdas y obstáculos distribuidos aleatoriamente. El objetivo está lejos del inicio.

Resultados:

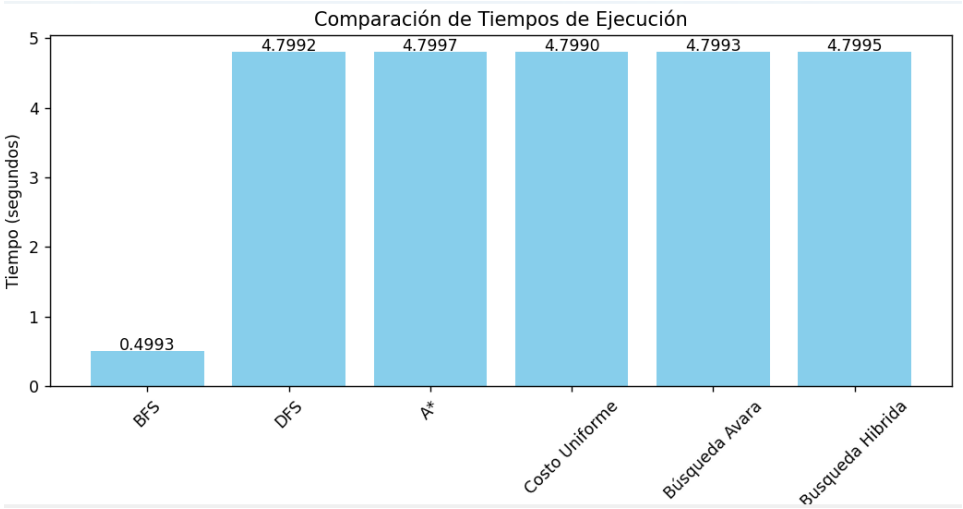


Fig. Comparación de los tiempos de ejecución de cada algoritmo, laberinto 10x10.

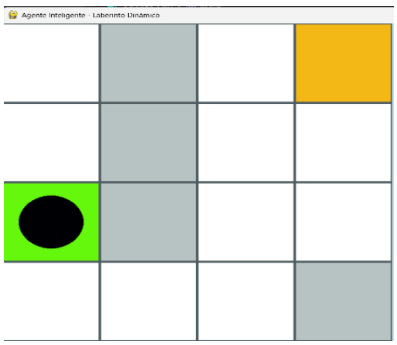
- **Escenario 3: Laberinto Dinámico (Cambios Frecuentes):**

Obstáculos aparecieron/desaparecieron durante la ejecución, lo que requirió que el agente recalculase la ruta, con búsqueda Hibrida:

Fragmento de Código para Adaptación Dinámica:

```
for evento in pygame.event.get():
    if evento.type == pygame.QUIT:
        ejecutando = False
    elif pygame.mouse.get_pressed()[0]:  # Click izquierdo
        x, y = pygame.mouse.get_pos()
        fila, columna = y // TAMAÑO_CASILLA, x // TAMAÑO_CASILLA
        if (fila, columna) not in [inicio, meta]:
            laberinto[fila][columna] = OBSTACULO if laberinto[fila][columna] == VACIO else VACIO
```

Resultados:



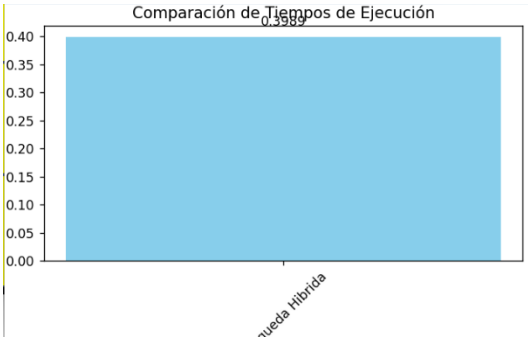
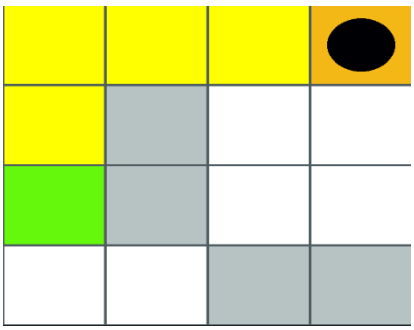


Fig. Tiempo de ejecución cuando los obstáculos están en esas posiciones.



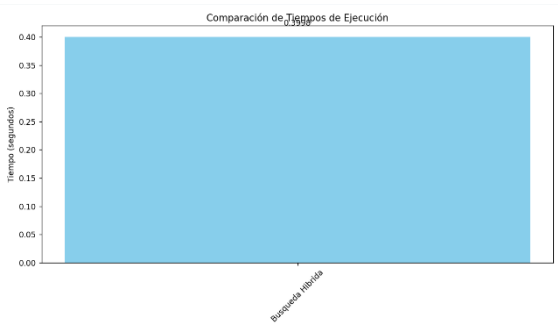


Fig. Tiempo de ejecución cuando los obstáculos están en esas posiciones.

- **Escenario 4 (Meta Dinámica):**

El objetivo cambió de posición varias veces durante la ejecución, con BFS:

Resultados:

Fig. Tiempo de ejecución cuando la meta está en la posición (1,3)

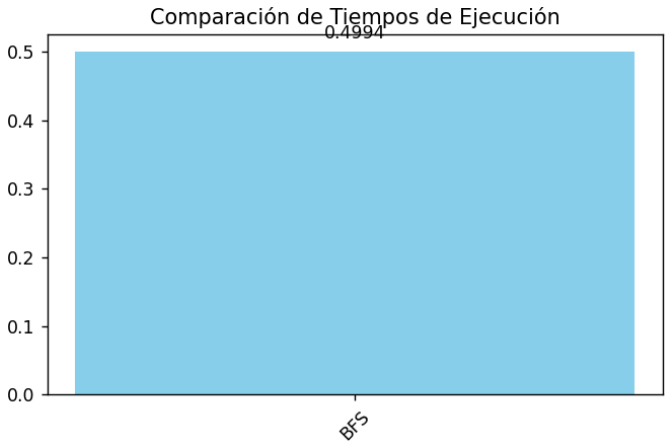


Fig. Tiempo de ejecución cuando la meta está en la posición (0,3)

2. Resultados

Eficiencia en Laberintos Pequeños:

- BFS, A* y Costo Uniforme mostraron un rendimiento similar en laberintos pequeños.
- Búsqueda Avara fue más rápida, pero no siempre encontró la ruta más corta.
- DFS exploró áreas innecesarias y no garantizó la ruta más corta.

Escalabilidad en Laberintos Grandes:

- A* fue significativamente más eficiente que BFS y DFS en laberintos grandes.
- Costo Uniforme mostró un rendimiento similar a A*, pero fue ligeramente más lento.
- Búsqueda Avara encontró rutas rápidamente, pero no siempre fueron válidas.
- DFS fue ineficiente y encontró rutas largas

Adaptabilidad en Laberintos Dinámicos:

- BFS y A* fueron más efectivos que DFS para recalcular rutas.
- Costo Uniforme mostró un rendimiento similar a A*.
- Búsqueda Avara encontró rutas rápidamente, pero no siempre fueron válidas.

- La Búsqueda Híbrida combinó las fortalezas de todas las técnicas y permitió al agente adaptarse dinámicamente.

Comportamiento con Meta Dinámica:

- A* demostró ser la técnica más adecuada para manejar metas móviles.
- Costo Uniforme fue más lento que A* debido a la falta de heurística.
- Búsqueda Avara encontró rutas rápidamente, pero no siempre fueron válidas.
- La Búsqueda Híbrida combinó las fortalezas de todas las técnicas y encontró la ruta más corta.

Gráficos:

- Incluye gráficos que muestren el tiempo de ejecución de cada técnica en diferentes escenarios.
- Ejemplo: Un gráfico de barras comparando el tiempo de BFS, DFS y A* en un laberinto grande.

3. Comportamiento del Agente

Adaptación Dinámica:

- El agente demostró ser capaz de adaptarse a cambios en el laberinto, como obstáculos y metas móviles.
- En algunos casos, el agente cambió de técnica para mejorar su rendimiento (por ejemplo, BFS → A*).

Robustez:

- El agente encontró rutas válidas en la mayoría de los escenarios, incluso cuando el laberinto cambiaba constantemente.
- En escenarios extremadamente complejos (muchos obstáculos móviles), el agente pudo explorar nuevas áreas usando DFS.

5. Conclusión

El proyecto implementó un agente inteligente capaz de navegar en un laberinto dinámico utilizando algoritmos como BFS, DFS, A*, Búsqueda por Costo Uniforme, Búsqueda Avara y una estrategia híbrida que combina todas las técnicas; el agente se evaluó en escenarios variados, demostrando que BFS es eficiente en laberintos pequeños, A* es más adecuado para laberintos grandes o dinámicos, y la estrategia híbrida ofrece versatilidad al adaptarse a diferentes situaciones, mientras que la interfaz gráfica interactiva desarrollada con Pygame permitió visualizar el comportamiento del agente paso a paso, resaltando la importancia de la heurística y la flexibilidad en entornos cambiantes.

Anexos

Código Fuente:

- https://github.com/sfcg2/Proyecto1-IA/tree/main

Archivo de Configuración:

- Archivo JSON utilizado para configurar el laberinto.