Escuela Politécnica Nacional

ROBOT EN UN LABERINTO

Integrantes:

Bravo Aleman Francis Alian

Freire Muesmuerán Brandon Ismael

Menendez Farias Joshua Daniel

Tinitana Carrion Joel Stalin

Materia:

Inteligencia Artificial

FECHA DE ENTREGA: 23 de enero de 2026

ESCUELA POLITÉCNICA NACIONAL
FACULTAD DE INGENIERÍA DE SISTEMAS
INGENIERÍA DE SISTEMAS INFORMÁTICOS Y DE COMPUTACIÓN

1 OBJETIVOS

1.1 OBJETIVO GENERAL:

Desarrollar e integrar un sistema de búsqueda que permita a un robot encontrar la ruta más eficiente hacia una meta dentro de un laberinto, evitando obstáculos y considerando variaciones en el terreno

1.2 OBJETIVOS ESPECÍFICOS:

- Implementar y comparar los algoritmos de búsqueda: Primero en Profundidad (DFS), Primero en Anchura (BFS), Costo Uniforme (UCS) y A*.
- Definir un agente de búsqueda que utilice los algoritmos implementados para interactuar con el entorno del laberinto.
- Modelar el entorno del laberinto incluyendo diferentes tipos de terreno (plano y empinado) y obstáculos.
- Visualizar el comportamiento del agente y el camino resultante mediante una interfaz gráfica desarrollada en Pygame.
- Probar y depurar el sistema en escenarios de laberinto con niveles de complejidad variables

2 INTRODUCCIÓN

El presente proyecto consiste en el desarrollo de un sistema de búsqueda inteligente para un robot autónomo que debe navegar dentro de un laberinto. El sistema requiere la implementación de un agente de búsqueda capaz de procesar la estructura del entorno, identificar obstáculos y encontrar una ruta desde un punto de inicio hasta una meta específica. El enfoque principal es la aplicación de diversas estrategias de búsqueda para determinar cuál ofrece la mayor eficiencia en términos de costo y recorrido.

3 MARCO TEÓRICO

El proyecto se fundamenta en la teoría de búsqueda en espacios de estados, donde el laberinto se representa como un conjunto de celdas transitables, obstáculos y metas

3.1 ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda Primero en Profundidad (DFS): Este algoritmo explora cada rama del árbol de búsqueda tan lejos como sea posible antes de retroceder (backtracking). No garantiza encontrar la ruta más corta.

Búsqueda Primero en Anchura (BFS): Explora todos los nodos de un nivel determinado antes de pasar al siguiente nivel de profundidad. Es óptimo si todos los costos de paso son iguales.

Búsqueda de Costo Uniforme (UCS): Expande el nodo con el menor costo acumulado. En este proyecto, se diferencia entre terreno plano (costo =1) y terreno empinado (costo =2), lo que obliga al algoritmo a calcular la ruta de menor costo total.

3.2 ALGORITMOS DE BÚSQUEDA INFORMADA

Búsqueda A*: Combina las ventajas de UCS (costo acumulado) con una función heurística para guiar la búsqueda de manera más eficiente hacia la meta.

Heurística de Distancia Manhattan: Es la métrica seleccionada para el algoritmo A* en este proyecto. Se define como la suma de las diferencias absolutas de sus coordenadas:

$$D_{Manhattan} = |x_1 - x_2| + |y_1 - y_2|$$

4 METODOLOGÍA

Para el desarrollo de este proyecto, se adoptó un enfoque de Modelado de Espacio de Estados, transformando el laberinto físico en un problema de búsqueda formal. El diseño se estructuró bajo los siguientes componentes:

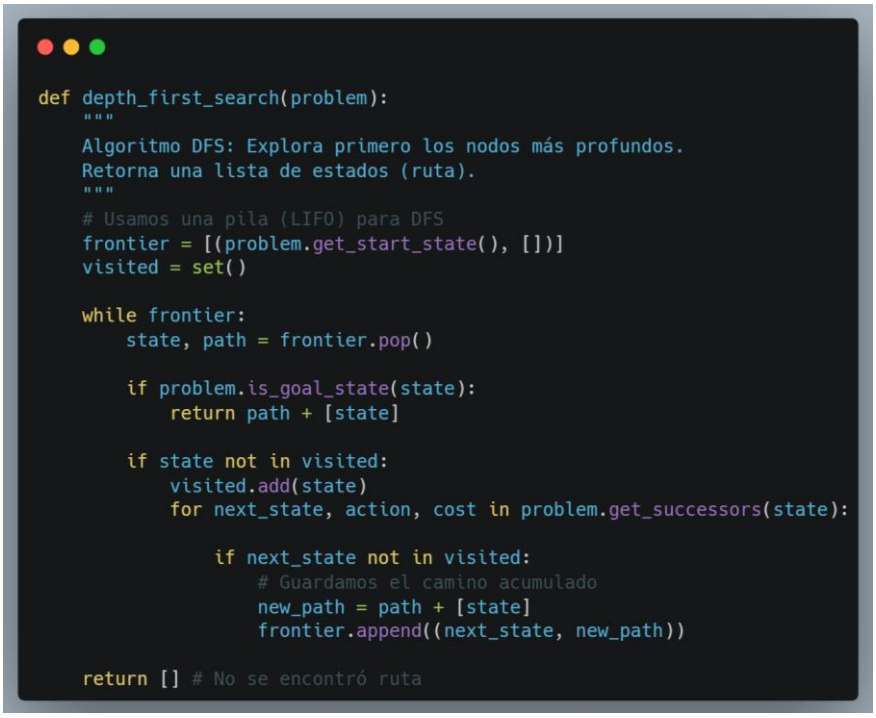
- **Definición del Estado:** Se representó cada posición del robot como una tupla de coordenadas (r,c), lo que permite una identificación única en la grilla del laberinto.

- **Modelo del Entorno:** A través de la clase `Labyrinth`, se implementó un sistema de carga de mapas desde archivos de texto, donde se distinguen obstáculos (`#`), terreno plano (costo 1) y terreno empinado (costo 2).
- **Abstracción del Problema:** Se utilizó la clase `LabyrinthSearchProblem` para encapsular las reglas del entorno. Esta clase define el estado inicial (`S`), la prueba de meta (`is_goal_state`) y la función de transición (`get_successors`), la cual devuelve los movimientos válidos y sus costos asociados basándose en la vecindad de la celda actual.

5 DESARROLLO TÉCNICO

La implementación se dividió en cuatro algoritmos fundamentales de búsqueda, cada uno con una lógica de exploración distinta:

- **Búsqueda no Informada (DFS y BFS):** Se implementó **DFS** utilizando una pila (LIFO) para priorizar la profundidad, y **BFS** mediante una cola (`collections.deque`) para garantizar el hallazgo de la ruta con menor número de pasos en entornos de costo uniforme.



```python
def depth_first_search(problem):
    """
    Algoritmo DFS: Explora primero los nodos más profundos.
    Retorna una lista de estados (ruta).
    """
    # Usamos una pila (LIFO) para DFS
    frontier = [(problem.get_start_state(), [])]
    visited = set()

    while frontier:
        state, path = frontier.pop()

        if problem.is_goal_state(state):
            return path + [state]

        if state not in visited:
            visited.add(state)
            for next_state, action, cost in problem.get_successors(state):

                if next_state not in visited:
                    # Guardamos el camino acumulado
                    new_path = path + [state]
                    frontier.append((next_state, new_path))

    return [] # No se encontró ruta
```

```python
def breadth_first_search(problem):
    """
    Algoritmo BFS (Breadth-First Search):
    Explora primero todos los nodos del mismo nivel antes de profundizar.
    Garantiza encontrar el camino más corto en número de pasos.
    """
    # Usamos una cola (FIFO) para BFS
    frontier = deque([(problem.get_start_state(), [])])
    visited = set()

    while frontier:
        # Extraemos el primer estado agregado (FIFO)
        state, path = frontier.popleft()

        # Verificamos si se alcanzó la meta
        if problem.is_goal_state(state):
            return path + [state]

        # Expandimos el estado si no ha sido visitado
        if state not in visited:
            visited.add(state)

            for next_state, action, cost in problem.get_successors(state):
                if next_state not in visited:
                    # Guardamos el camino acumulado
                    new_path = path + [state]
                    frontier.append((next_state, new_path))

    # No se encontró una ruta
    return []
```

- **Búsqueda por Costo Uniforme (UCS):** Para manejar la variabilidad del terreno (terreno empinado con costo c=2), se empleó una cola de prioridad gestionada por la librería `heapq`. Este algoritmo asegura la optimización del costo total acumulado, expandiendo siempre el nodo con el g(n) más bajo.

```python
def uniform_cost_search(problem):
    """
    Algoritmo UCS (Uniform Cost Search):
    Expande siempre el nodo con el menor costo acumulado.
    Considera el costo de los terrenos (plano = 1, empinado = 2).
    """
    # Cola de prioridad: (costo acumulado, estado, camino)
    frontier = []
    heapq.heappush(frontier, (0, problem.get_start_state(), []))

    # Diccionario para guardar el menor costo encontrado por estado
    visited = {}

    while frontier:
        cost_so_far, state, path = heapq.heappop(frontier)

        # Verificamos si se alcanzó la meta
        if problem.is_goal_state(state):
            return path + [state]

        # Expandimos si no se ha visitado o se encontró un costo menor
        if state not in visited or cost_so_far < visited[state]:
            visited[state] = cost_so_far

            for next_state, action, cost in problem.get_successors(state):
                new_cost = cost_so_far + cost
                new_path = path + [state]
                heapq.heappush(frontier, (new_cost, next_state, new_path))

    # No se encontró una ruta
    return []
```

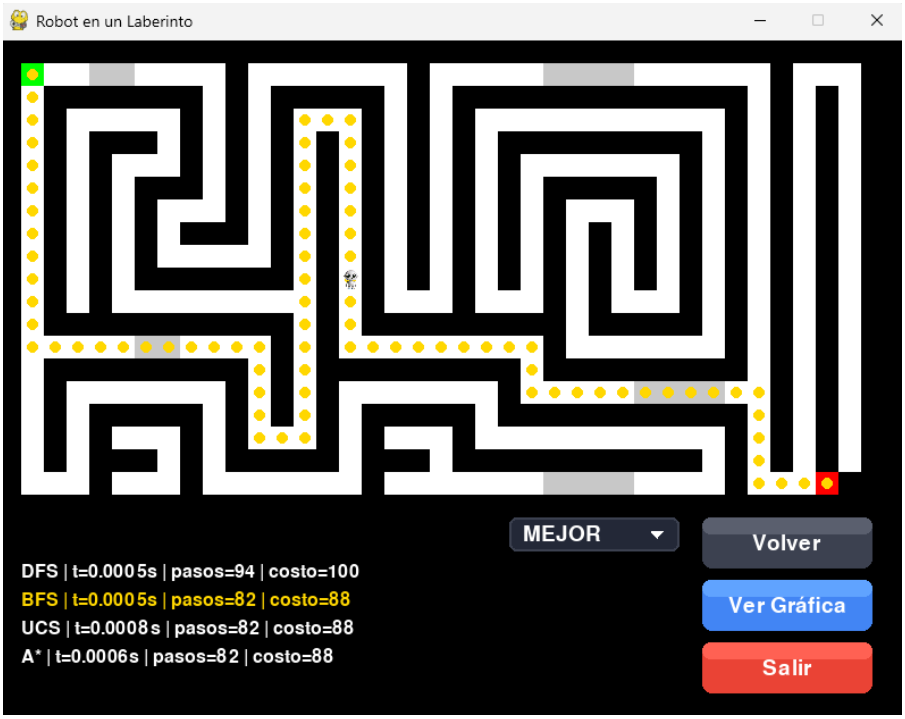
- **Búsqueda Informada (A*):** Se integró el algoritmo A* combinando el costo acumulado con una función heurística. Se seleccionó la Distancia Manhattan por ser una métrica admisible y consistente para movimientos en grillas de cuatro direcciones (arriba, abajo, izquierda, derecha).

```python
def manhattan(a, b):
    """
    Heurística Manhattan:
    Estima la distancia entre dos puntos en una
grilla"
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

```python
def a_star_search(problem):
    """
    Algoritmo A*:
    Combina UCS con una heurística (distancia Manhattan)
    para guiar la búsqueda de forma más eficiente.
    """
    start = problem.get_start_state()

    # Cola de prioridad: (prioridad, estado, camino, costo acumulado)
    frontier = []
    heapq.heappush(frontier, (0, start, [], 0))

    visited = {}

    while frontier:
        priority, state, path, cost_so_far = heapq.heappop(frontier)

        # Verificamos si se alcanzó la meta
        if problem.is_goal_state(state):
            return path + [state]

        # Expandimos el estado si es más barato que antes
        if state not in visited or cost_so_far < visited[state]:
            visited[state] = cost_so_far

            for next_state, action, cost in problem.get_successors(state):
                new_cost = cost_so_far + cost
                heuristic = manhattan(next_state, problem.labyrinth.goal_pos)
                new_priority = new_cost + heuristic
                new_path = path + [state]

                heapq.heappush(
                    frontier,
                    (new_priority, next_state, new_path, new_cost)
                )

    # No se encontró una ruta
    return []
```

6 RESULTADOS

Para validar el sistema, se realizaron pruebas de ejecución en el escenario **`Dificil_1.txt`**, el cual presenta múltiples rutas posibles y zonas de terreno empinado. El objetivo primordial fue evaluar la capacidad de cada algoritmo para encontrar la ruta más eficiente evitando obstáculos



Los datos obtenidos en la simulación se resumen en la siguiente tabla comparativa:

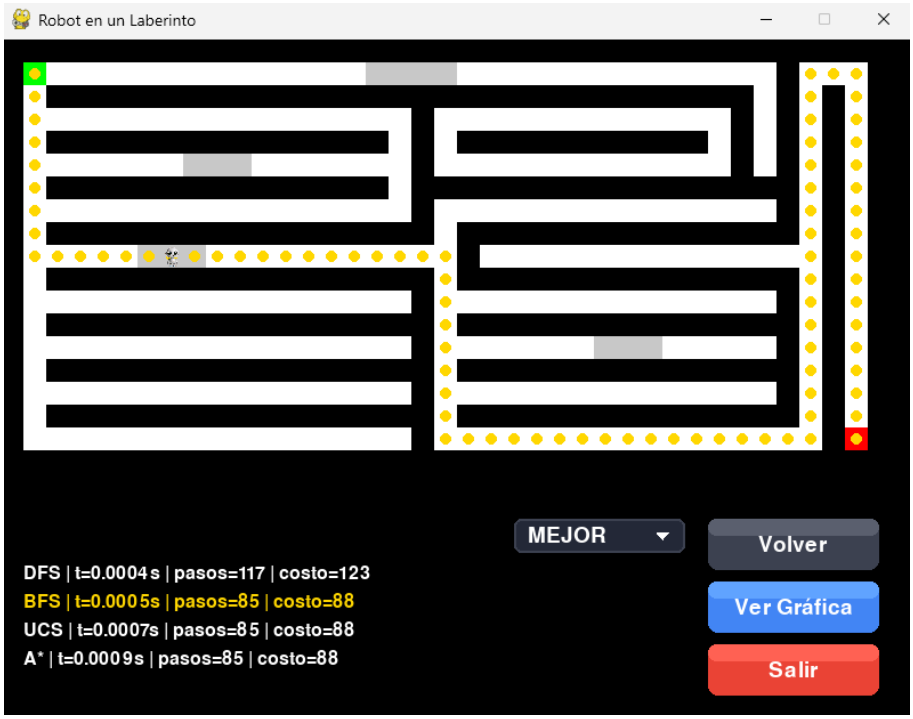
Algoritmo	Tiempo (s)	Pasos	Costo Total	Observación
DFS	0.0005	94	100	Ruta subóptima y más larga.
BFS	0.0005	82	88	Ruta óptima en pasos y costo.
UCS	0.0008	82	88	Ruta óptima, mayor tiempo de cómputo.
A*	0.0006	82	88	Ruta óptima con alta eficiencia.

Seguidamente, como se observa en los gráficos de la simulación, existe una disparidad notable en la calidad de las soluciones encontradas. Mientras que DFS encontró una ruta de 94 pasos con un costo de 100, los demás algoritmos convergieron en una ruta óptima de 82 pasos y 88 de costo.

Es importante notar que el costo (88) es mayor al número de pasos (82), lo que confirma que la ruta atraviesa celdas de terreno empinado con valor 2. En términos de tiempo, aunque los

valores son extremadamente bajos, BFS y DFS fueron los más rápidos, mientras que UCS fue el más lento debido a la gestión de la cola de prioridad sin una guía heurística.

La segunda fase de pruebas se llevó a cabo en el mapa **`Dificil_2.txt`.** Este entorno requiere una planificación de ruta más larga en comparación con el escenario anterior, lo que permite observar con mayor claridad la degradación de los algoritmos no óptimos frente a los óptimos.



A continuación, se detallan las métricas obtenidas en la simulación:

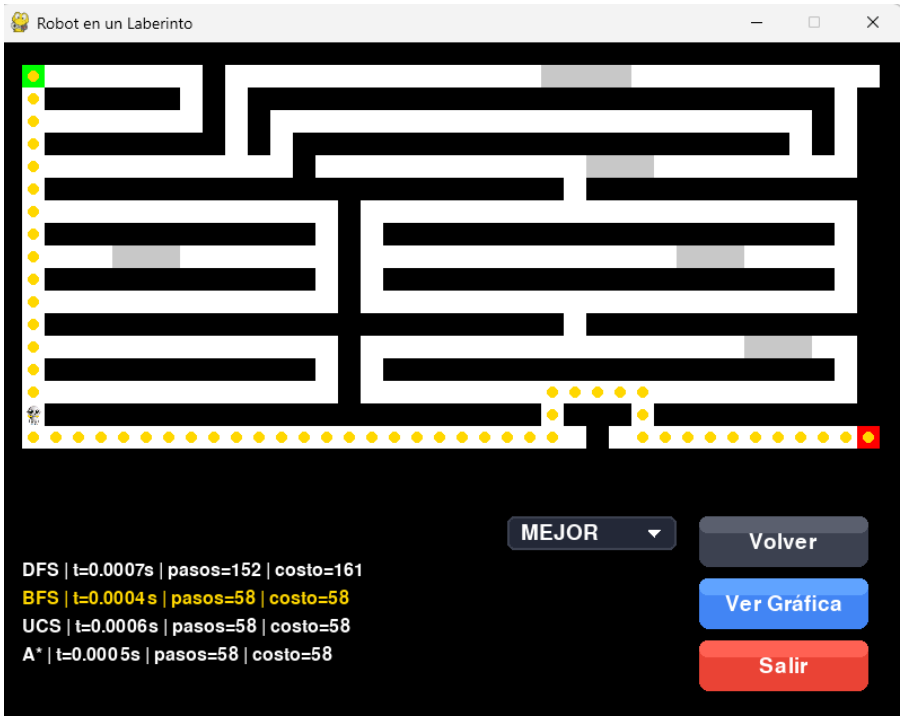
Algoritmo	Tiempo (s)	Pasos	Costo Total
DFS	0.0004	117	123
BFS	0.0005	85	88
UCS	0.0007	85	88
A*	0.0009	85	88

Seguidamente, como se observa en los gráficos de la simulación, se obtienen los siguientes detalles:

- Se observa una brecha significativa en la calidad de la solución proporcionada por DFS, el cual generó una ruta de 117 pasos. En contraste, BFS, UCS y A* convergieron nuevamente en la solución óptima de 85 pasos y un costo de 88.
- El costo total de 88 frente a los 85 pasos confirma la presencia de terrenos empinados (costo 2) en la ruta óptima, los cuales fueron correctamente procesados por el agente de búsqueda.

- Curiosamente, en este mapa específico, A* presentó un tiempo de ejecución ligeramente superior (0.0009s) al de UCS (0.0007s). Esto puede atribuirse a la sobrecarga computacional que implica el cálculo constante de la distancia Manhattan en una ruta más larga y con una estructura de laberinto que quizás genera más estados candidatos con valores f(n) similares.
- A pesar de la variación en tiempos, la consistencia en el costo final (88) entre los tres algoritmos óptimos valida la robustez de la implementación de la clase `LabyrinthSearchProblem`

Las pruebas realizadas en el mapa **`Medio_1.txt`** permiten observar el comportamiento de los algoritmos en un entorno con pasillos largos y áreas abiertas. Este escenario es crucial para identificar cómo el algoritmo de profundidad puede "perderse" en ramas extensas que no conducen directamente al objetivo.



Los resultados cuantitativos se presentan en la siguiente tabla:

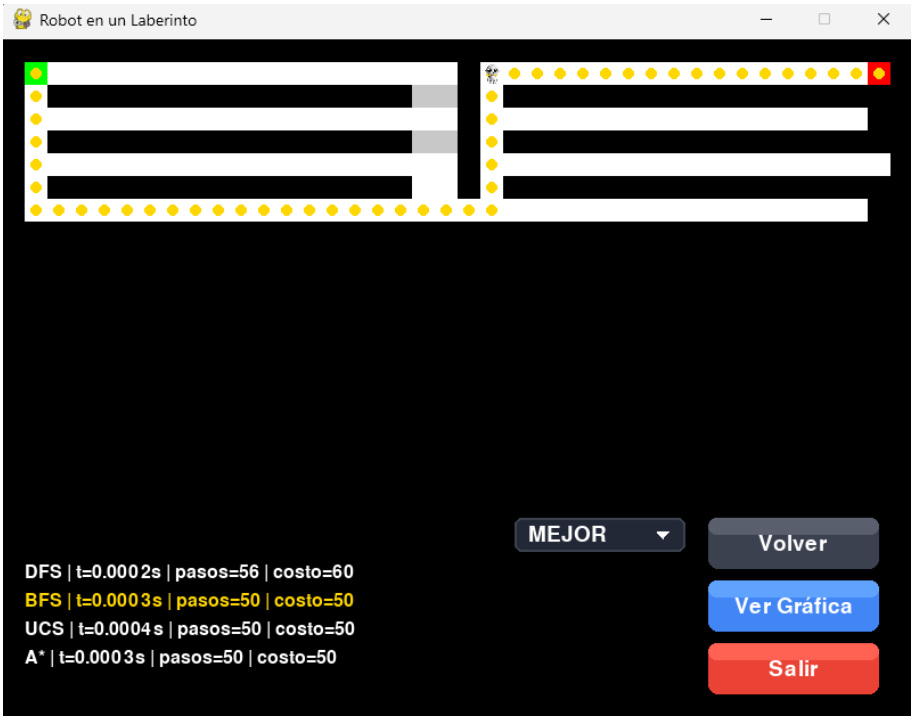
Algoritmo	Tiempo (s)	Pasos	Costo Total
DFS	0.0007	152	161
BFS	0.0004	58	58
UCS	0.0006	58	58
A*	0.0005	58	58

Seguidamente, como se observa en los gráficos de la simulación, se obtienen los siguientes detalles:

- En este escenario, la diferencia entre una búsqueda óptima y una no óptima es drástica. DFS generó una ruta de 152 pasos, lo que representa casi el triple de la distancia necesaria. Esto se debe a que el algoritmo exploró ramas profundas del laberinto antes de retroceder, acumulando un costo de 161.

- A diferencia de los escenarios difíciles, en la ruta óptima encontrada para este mapa (58 pasos), el costo total también fue de 58. Esto indica que los algoritmos BFS, UCS y A* lograron evitar completamente las celdas de terreno empinado (costo 2), seleccionando el camino más eficiente tanto en distancia como en esfuerzo.
- BFS obtuvo el mejor tiempo de ejecución (0.0004s), seguido muy de cerca por A* (0.0005s). En este caso, la simplicidad de la cola FIFO de BFS permitió encontrar la meta rápidamente en un entorno donde la solución óptima no requería ponderar costos complejos de terreno.

El escenario **`Facil_1.txt`** se caracteriza por una estructura lineal y directa con pocos obstáculos. Este mapa se utilizó para validar la consistencia de los algoritmos en condiciones ideales y verificar el comportamiento del agente ante terrenos de bajo costo.



Los resultados de la ejecución se detallan a continuación:

Algoritmo	Tiempo (s)	Pasos	Costo Total
DFS	0.0002	56	60
BFS	0.0003	50	50
UCS	0.0004	50	50
A*	0.0003	50	50

Seguidamente, como se observa en los gráficos de la simulación, se obtienen los siguientes detalles:

- En este entorno simplificado, los algoritmos BFS, UCS y A* encontraron la ruta perfecta de 50 pasos con un costo de 50. Esto indica que la ruta óptima consiste exclusivamente en terreno plano (costo 1), evitando cualquier celda de terreno empinado disponible en el mapa.

- A pesar de la simplicidad del laberinto, DFS volvió a entregar una solución subóptima de 56 pasos. El costo total de 60 revela que este algoritmo no solo tomó un camino más largo, sino que además transitó por celdas de terreno empinado (específicamente 4 celdas de costo 2), lo que aumenta innecesariamente el esfuerzo del robot.
- Debido a la baja profundidad del árbol de búsqueda, los tiempos son casi instantáneos. DFS registró el tiempo más bajo (0.0002s) debido a que encontró una solución válida rápidamente sin necesidad de explorar niveles adicionales o calcular heurísticas, aunque la calidad de dicha solución sea inferior.

7 CONCLUSIONES

El desarrollo e implementación del sistema de búsqueda para navegación de un robot en laberinto permitió validar de manera empírica el comportamiento teórico de los algoritmos estudiados. A partir de las pruebas realizadas en los diferentes escenarios, se obtienen las siguientes conclusiones:

El algoritmo de Búsqueda Primero en Profundidad (DFS) demostró ser consistentemente el menos adecuado para la navegación óptima en laberintos. En todos los escenarios evaluados,

DFS generó rutas subóptimas con mayor número de pasos y costos más elevados. Por ejemplo, en el mapa Medio_1.txt, DFS produjo una ruta de 152 pasos frente a los 58 pasos de la solución óptima, lo que representa casi el triple de la distancia necesaria. Esta característica se debe a su naturaleza de explorar ramas profundas sin considerar la proximidad a la meta.

Los algoritmos BFS, UCS y A* convergieron consistentemente en soluciones óptimas a través de todos los escenarios de prueba. La igualdad en el número de pasos y costos totales entre estos tres algoritmos confirma la correcta implementación de la función de transición y la gestión de costos en la clase LabyrinthSearchProblem.

La diferencia entre el número de pasos y el costo total en las rutas óptimas permitió verificar el correcto procesamiento de los terrenos con costos diferenciados. En los mapas difíciles, donde el costo (88) superó al número de pasos (82 u 85), se confirmó que los algoritmos transitaron por celdas de terreno empinado cuando era necesario para alcanzar la meta.

En términos de eficiencia temporal, BFS presentó los mejores tiempos de ejecución en la mayoría de escenarios debido a la simplicidad de su estructura de cola FIFO. Sin embargo, A* ofreció un equilibrio óptimo entre tiempo de cómputo y calidad de solución, siendo especialmente valioso en entornos donde la heurística de distancia Manhattan guía eficientemente la búsqueda hacia la meta.

La heurística de Distancia Manhattan demostró ser una métrica adecuada para el algoritmo A* en grillas de cuatro direcciones, cumpliendo con las propiedades de admisibilidad y consistencia requeridas para garantizar soluciones óptimas.

Finalmente, la interfaz gráfica desarrollada en Pygame permitió visualizar de manera efectiva el comportamiento de cada algoritmo, facilitando la comparación cualitativa de las rutas generadas y validando el cumplimiento de los objetivos planteados para el proyecto.