Informe proyecto 1

Autores
Johan Sebastián Acosta - 2380393
Kevin Andrés Bejarano - 2067678
Juan David Gutiérrez Florez - 2060104

Docente
Joshua David Triana Madrid

Curso
Inteligencia Artificial

Semestre
Sexto - VI

Tuluá, Valle del cauca
abril, 2025

1. Introducción y Objetivo

El presente proyecto tiene como objetivo desarrollar un agente inteligente capaz de navegar en un laberinto dinámico utilizando técnicas de búsqueda clásicas de la Inteligencia Artificial. Entre estas técnicas se incluyen búsqueda en amplitud (BFS), búsqueda en profundidad (DFS), búsqueda A* y una estrategia adaptativa que elige automáticamente la mejor técnica en función del entorno. El agente debe ser capaz de adaptarse a un entorno en constante cambio, donde pueden aparecer o desaparecer paredes, cambiar la posición de la meta e incluso moverse obstáculos con diferentes costos asociados.

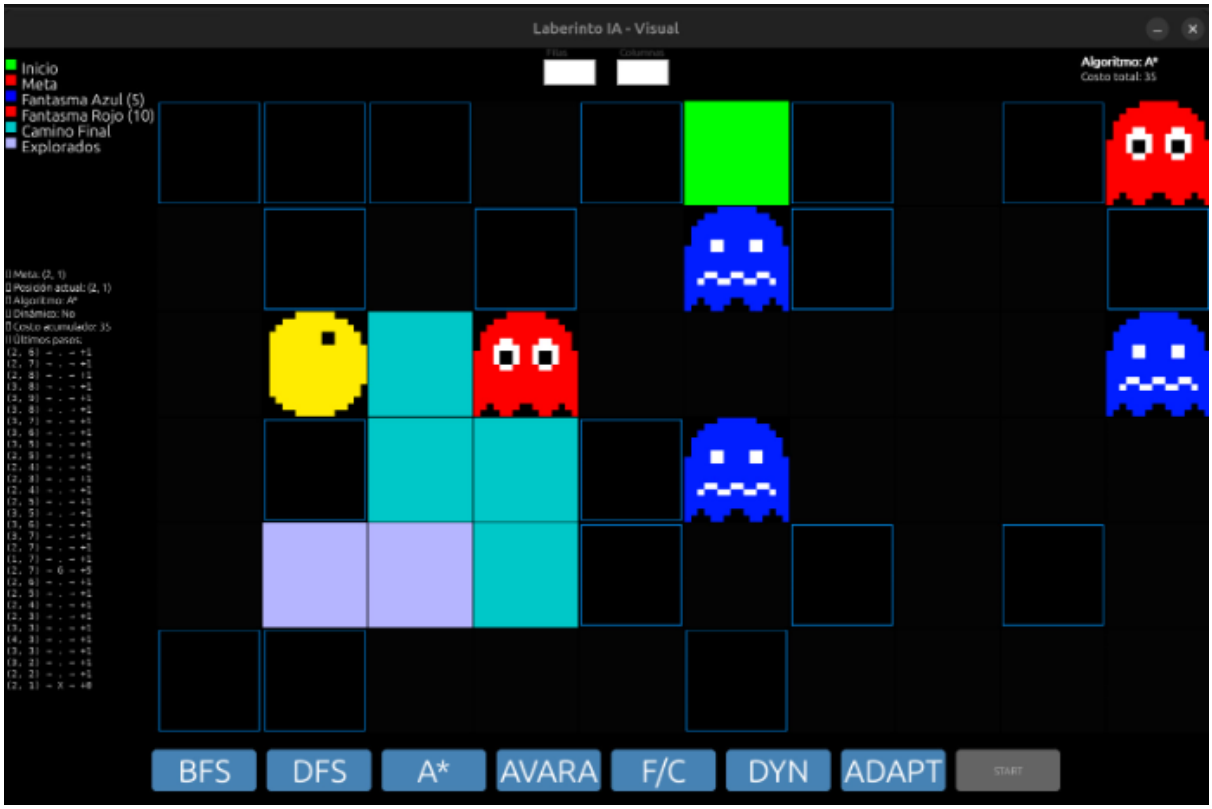
2. Justificación

La resolución de problemas en entornos dinámicos es un desafío clave en la Inteligencia Artificial. Este proyecto no solo permite entender el funcionamiento de los algoritmos de búsqueda, sino también su comportamiento frente a cambios del entorno. La inclusión de obstáculos con diferentes costos y movimiento de la meta permite simular condiciones más realistas, lo cual fortalece el análisis y comprensión de los algoritmos utilizados. Además, el uso de visualización gráfica facilita la interpretación de los resultados, sirviendo también como herramienta educativa.

3. Metodología y Tecnologías utilizadas

Se empleó el lenguaje de programación Python, junto con la biblioteca PyGame para el desarrollo de la interfaz gráfica. El proyecto fue organizado en módulos independientes para los agentes, entorno, estrategias, y visualización. Cada algoritmo fue implementado en archivos separados, facilitando su prueba individual. Se utilizó un enfoque incremental de desarrollo, comenzando con entornos estáticos para validar la lógica de búsqueda, y posteriormente implementando la lógica dinámica y adaptativa.

4. Diseño del Agente Inteligente



El agente se modela como una entidad que toma decisiones en función de su entorno. Internamente, utiliza un nodo Estado que contiene la posición, el padre (para reconstruir el camino), el costo real (g) y la heurística (h). Con base en estos atributos, se evalúan las mejores rutas según el algoritmo activo. El agente adaptativo analiza los costos de los caminos generados por BFS, DFS y A* y selecciona el de menor costo real.

5. Algoritmos de Búsqueda Implementados

-Algoritmo BFS (Búsqueda en Amplitud)

Garantiza la ruta más corta en términos de pasos, sin considerar los costos. Usa una cola. Explora primero los nodos más cercanos. Ideal para encontrar caminos cortos (en pasos), pero no considera costos.

```python
# Búsqueda por amplitud
from collections import deque
from core.costo import costo
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta

def bfs(laberinto, inicio_pos, meta_pos):

    estado_inicial = Estado(posicion=inicio_pos)
    frontera = deque([estado_inicial])
    visitados = set()
    visitados.add(inicio_pos)

    while frontera:
        actual = frontera.popleft()

        if es_meta(actual, meta_pos):
            camino = actual.reconstruir_camino()
            costo_total = sum(
                costo(camino[i], camino[i+1], laberinto) for i in range(len(camino) - 1)
            )
            return camino, costo_total, visitados

        sucesores = generar_sucesores(actual, laberinto)

        for sucesor in sucesores:
            if sucesor.pos not in visitados:
                frontera.append(sucesor)
                visitados.add(sucesor.pos)

    return None, 0, set() # No se encontró camino
```

IMPORTACIONES Y UTILIDADES

```
from collections import deque
from core.costo import costo
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
```

Importamos funciones y clases auxiliares:

deque: cola eficiente para BFS.

costo: para calcular el costo de moverse de un punto a otro.

Estado: clase que representa un nodo del laberinto, guarda posición y camino.

generar_sucesores: devuelve los movimientos posibles desde una posición.

es_meta: verifica si se alcanzó la meta.

DEFINICIÓN DE LA FUNCIÓN PRINCIPAL

```
def bfs(laberinto, inicio_pos, meta_pos):
```

Esta es la función principal

laberinto: es la matriz que representa el mapa.

inicio_pos: coordenadas del punto de inicio (por ejemplo, Pacman).

meta_pos: coordenadas del destino (por ejemplo, el queso).

INICIALIZACIÓN DEL ESTADO INICIAL Y LA FRONTERA

```
estado_inicial = Estado(posicion=inicio_pos)
frontera = deque([estado_inicial])
visitados = set()
visitados.add(inicio_pos)
```

Aquí se prepara todo para iniciar la búsqueda:

- Se crea el primer estado (el agente en su punto inicial).
- Se inicializa la **frontera** como una cola FIFO (deque) que guarda los nodos por visitar.
- Se lleva un registro de nodos **visitados** en un set para evitar repeticiones.

CICLO PRINCIPAL DE BÚSQUEDA

```
while frontera:
    actual = frontera.popleft()
```

Mientras haya elementos en la frontera:

- Se toma el **primer elemento** (por eso es una cola FIFO).

VERIFICACIÓN DE META

```
if es_meta(actual, meta_pos):
    camino = actual.reconstruir_camino()
    costo_total = sum(
        costo(camino[i], camino[i+1], laberinto) for i in range(len(camino) - 1)
    )
    return camino, costo_total, visitados
```

Si se llegó al objetivo:

- Se reconstruye el **camino completo** desde el inicio hasta el estado actual, usando la cadena de padre.
- Se calcula el **costo total** del recorrido.
- Se retorna: el **camino**, el **costo total** y los **nodos explorados**.

EXPANSIÓN DE SUCESORES

```
sucesores = generar_sucesores(actual, laberinto)
```

- Se generan los movimientos posibles desde la posición actual.
- Esta función se asegura de no moverse hacia paredes o fuera del laberinto.

REGISTRO DE NUEVOS ESTADOS

```
for sucesor in sucesores:
    if sucesor.pos not in visitados:
        frontera.append(sucesor)
        visitados.add(sucesor.pos)
```

Para cada nuevo estado generado:

Si no ha sido visitado:

- Se agrega a la frontera (para ser explorado después).
- Se marca como visitado.

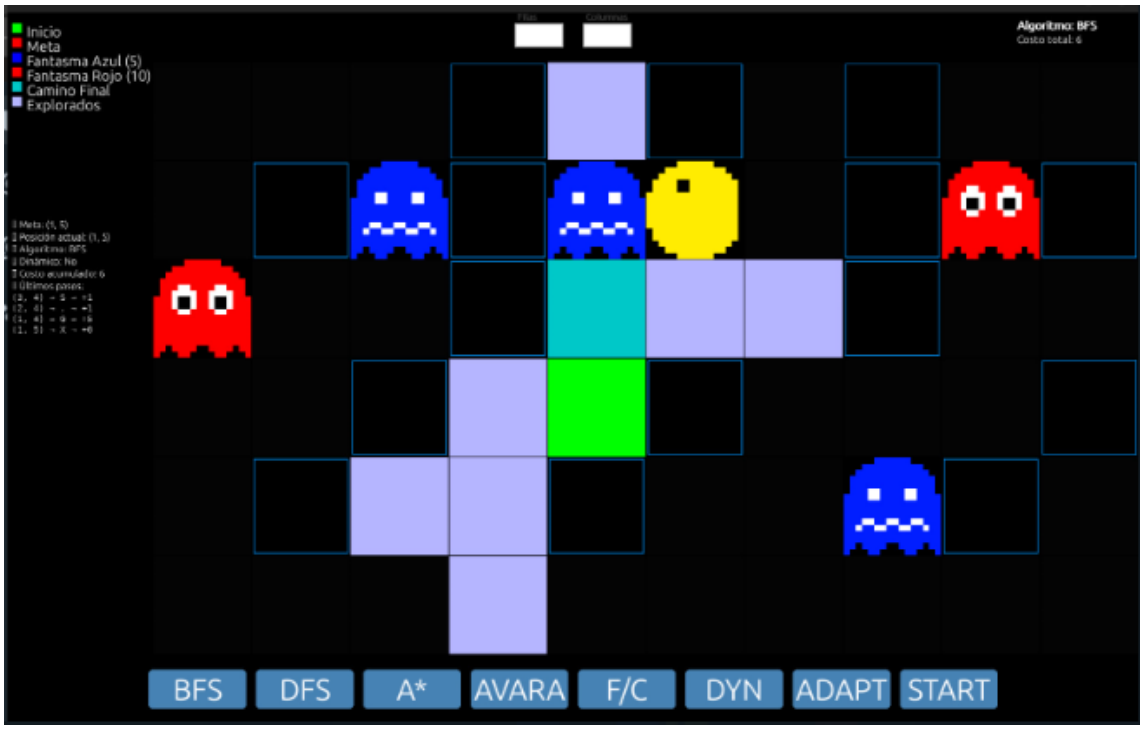
CASO SIN SOLUCIÓN

```
return None, 0, set() # No se encontró camino
```

Si se termina la frontera y nunca se encontró la meta:

- Se retorna: camino vacío, costo 0 y conjunto vacío de visitados.

Evidencia visual algoritmo BFS



-Algoritmo DFS (Búsqueda en Profundidad)

Puede ser más rápida pero no garantiza la mejor ruta. Usa una pila. Se va lo más profundo posible antes de retroceder. No garantiza el mejor camino, pero es rápido y simple.

```python
# Búsqueda por profundidad

from collections import deque
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
from core.costo import costo  # <- importante para calcular el costo final

def dfs(laberinto, inicio_pos, meta_pos):

    estado_inicial = Estado(posicion=inicio_pos)
    frontera = deque([estado_inicial])
    visitados = set()
    visitados.add(inicio_pos)

    while frontera:
        actual = frontera.pop()  # DFS usa LIFO

        if es_meta(actual, meta_pos):
            camino = actual.reconstruir_camino()
            costo_total = sum(
                costo(camino[i], camino[i+1], laberinto) for i in range(len(camino) - 1)
            )
            return camino, costo_total, visitados

        sucesores = generar_sucesores(actual, laberinto)

        for sucesor in sucesores:
            if sucesor.pos not in visitados:
                frontera.append(sucesor)
                visitados.add(sucesor.pos)

    return None, 0, set() # No se encontró camino
```

IMPORTACIONES Y UTILIDADES

```
# Búsqueda por profundidad

from collections import deque
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
from core.costo import costo  # <- importante para calcular el costo final
```

Estos módulos proveen herramientas necesarias:

deque: estructura doblemente enlazada, se usa como pila (LIFO) para DFS.

Estado: representa un nodo con su posición, padre, costo, etc.

generar_sucesores: genera los posibles movimientos desde el estado actual.

es_meta: evalúa si el estado actual es la meta.

costo: calcula el costo entre dos pasos en el laberinto.

DEFINICIÓN DE LA FUNCIÓN PRINCIPAL

```
def dfs(laberinto, inicio_pos, meta_pos):
```

Define la función de búsqueda en profundidad. Recibe:

- **laberinto:** la matriz que representa el mapa.
- **inicio_pos:** coordenadas iniciales.
- **meta_pos:** coordenadas del objetivo.

INICIALIZACIÓN DEL ESTADO INICIAL Y LA FRONTERA

```
estado_inicial = Estado(posicion=inicio_pos)
frontera = deque([estado_inicial])
visitados = set()
visitados.add(inicio_pos)
```

Se prepara el entorno inicial:

- Se crea un Estado con la posición inicial.
- La frontera se inicializa con ese estado. Esta vez se usará como pila LIFO, extrayendo el último elemento agregado.
- El conjunto visitado evita repeticiones durante la búsqueda.

CICLO PRINCIPAL DE EXPLORACIÓN

```
while frontera:
    actual = frontera.pop()  # DFS usa LIFO
```

Mientras haya nodos en la frontera:

- Se extrae el último (último en entrar, primero en salir). Esto hace que DFS se adentre en una rama antes de explorar otras.

VERIFICACIÓN DE META

```
if es_meta(actual, meta_pos):
    camino = actual.reconstruir_camino()
    costo_total = sum(
        costo(camino[i], camino[i+1], laberinto) for i in range(len(camino) - 1)
    )
    return camino, costo_total, visitados
```

Si el nodo actual es la meta:

- Se reconstruye el camino recorrido desde el inicio.
- Se calcula el costo total sumando los movimientos.
- Se retorna: **camino completo**, **costo total** y **nodos explorados**.

GENERACIÓN Y REGISTRO DE SUCESORES

```
sucesores = generar_sucesores(actual, laberinto)

for sucesor in sucesores:
    if sucesor.pos not in visitados:
        frontera.append(sucesor)
        visitados.add(sucesor.pos)
```

Se generan los estados vecinos (sucesores).

Si el sucesor no ha sido visitado:

- Se añade a la frontera (al final, como en una pila).
- Se marca como visitado.

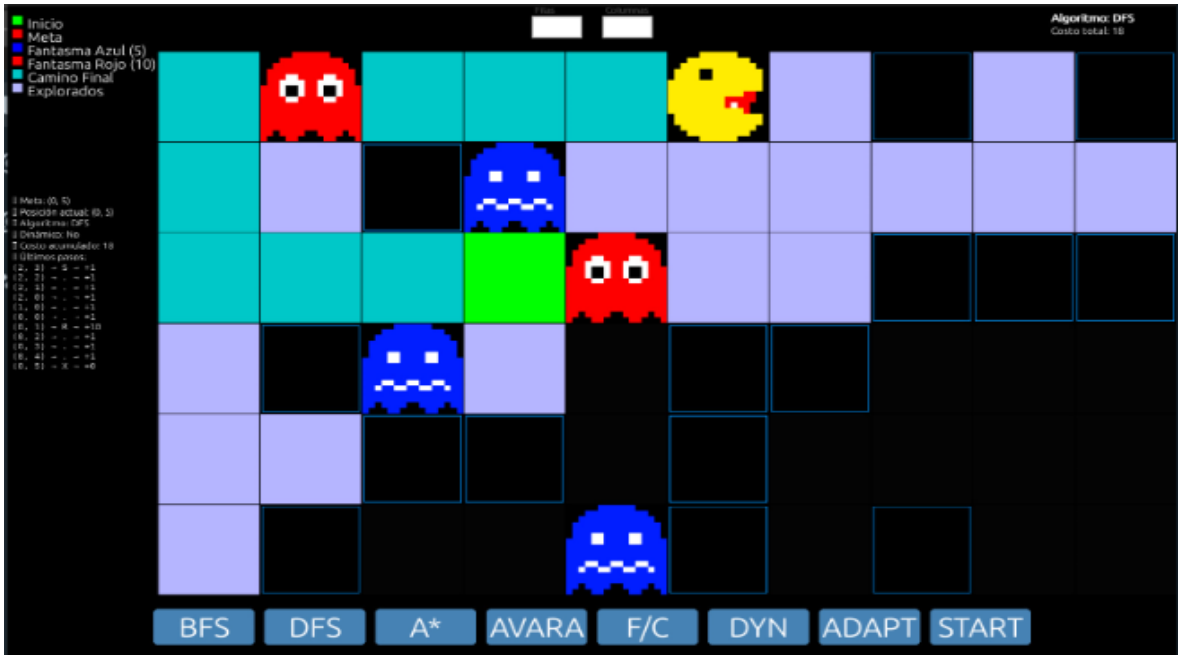
CASO SIN SOLUCIÓN

```
return None, 0, set() # No se encontró camino
```

Si no se encuentra camino después de explorar todo:

- Se retorna que no hay solución.

Evidencia visual algoritmo DFS



-Algoritmo A*

Combina el costo real (g) con una heurística admisible (distancia Manhattan) para encontrar caminos óptimos. Usa cola de prioridad. Considera costo real + heurística. Encuentra el camino más barato si la heurística es buena.

```python
import heapq
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
from core.costo import costo

def heuristica(pos, meta):
    """Calcula la distancia Manhattan entre dos posiciones"""
    return abs(pos[0] - meta[0]) + abs(pos[1] - meta[1])

def astar(laberinto, inicio_pos, meta_pos):
    estado_inicial = Estado(posicion=inicio_pos, g=0, h=heuristica(inicio_pos, meta_pos))
    frontera = [(estado_inicial.f(), estado_inicial)]
    visitados = {}
    explorados = set()

    while frontera:
        _, actual = heapq.heappop(frontera)
        explorados.add(actual.pos)

        if es_meta(actual, meta_pos):
            return actual.reconstruir_camino(), actual.g, explorados

        sucesores = generar_sucesores(actual, laberinto)

        for sucesor in sucesores:
            nuevo_costo = actual.g + costo(actual.pos, sucesor.pos, laberinto)

            if sucesor.pos not in visitados or nuevo_costo < visitados[sucesor.pos]:
                visitados[sucesor.pos] = nuevo_costo
                h = heuristica(sucesor.pos, meta_pos)
                nuevo_estado = Estado(posicion=sucesor.pos, padre=actual, g=nuevo_costo, h=h)
                heapq.heappush(frontera, (nuevo_estado.f(), nuevo_estado))

    return None, float('inf'), explorados
```

IMPORTACIONES Y UTILIDADES

```
import heapq
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
from core.costo import costo
```

Este bloque importa:

- **heapq:** estructura tipo cola de prioridad (min-heap), para elegir el mejor nodo (menor costo f).
- **Estado:** objeto que guarda información del nodo actual (posición, costo, heurística).
- **generar_sucesores:** genera los estados a los que se puede ir desde el actual.
- **es_meta:** revisa si llegamos a la meta.
- **costo:** retorna cuánto cuesta pasar de una celda a otra.

FUNCIÓN DE HEURÍSTICA (Manhattan)

```
def heuristica(pos, meta):
    """Calcula la distancia Manhattan entre dos posiciones"""
    return abs(pos[0] - meta[0]) + abs(pos[1] - meta[1])
```

Define la heurística usada por el algoritmo. En este caso, se usa la distancia de Manhattan, ideal para cuadrículas sin diagonales. Esta función se utiliza para estimar "cuánto falta" desde una posición hasta la meta.

FUNCIÓN PRINCIPAL

```
def astar(laberinto, inicio_pos, meta_pos):
```

Define el algoritmo A*:

- Recibe un laberinto, la posición inicial y la meta.
- Devuelve el **camino**, el **costo total** y los **nodos explorados**.

INICIALIZACIÓN DEL ESTADO INICIAL Y ESTRUCTURAS

```
def astar(laberinto, inicio_pos, meta_pos):
    estado_inicial = Estado(posicion=inicio_pos, g=0, h=heuristica(inicio_pos, meta_pos))
    frontera = [(estado_inicial.f(), estado_inicial)]
    visitados = {}
    explorados = set()
```

Aquí se preparan:

- **estado_inicial:** con g = 0 (costo acumulado) y h = distancia a la meta.
- **frontera:** es una cola de prioridad, ordenada por f = g + h.
- **visitados:** guarda el menor costo para llegar a cada celda.
- **explorados:** guarda las posiciones que se visitaron efectivamente.

CICLO PRINCIPAL DE EXPLORACIÓN

```
while frontera:
    _, actual = heapq.heappop(frontera)
    explorados.add(actual.pos)
```

Se toma el nodo con menor f() de la frontera. Esto permite explorar primero las rutas más prometedoras (baratas y cercanas a la meta).

VERIFICAR SI LLEGAMOS A LA META

```
if es_meta(actual, meta_pos):
    return actual.reconstruir_camino(), actual.g, explorados
```

Si llegamos a la meta:

- Se reconstruye el camino desde el nodo actual hacia el inicio.
- Se devuelve ese camino, el costo acumulado g y los nodos explorados.

GENERACIÓN Y PROCESAMIENTO DE SUCESORES

```
sucesores = generar_sucesores(actual, laberinto)

for sucesor in sucesores:
    nuevo_costo = actual.g + costo(actual.pos, sucesor.pos, laberinto)
```

Se calcula **nuevo_costo:** g acumulado desde el inicio hasta este nuevo nodo.

ACTUALIZACIÓN SI ES MEJOR CAMINO

```
if sucesor.pos not in visitados or nuevo_costo < visitados[sucesor.pos]:
    visitados[sucesor.pos] = nuevo_costo
    h = heuristica(sucesor.pos, meta_pos)
    nuevo_estado = Estado(posicion=sucesor.pos, padre=actual, g=nuevo_costo, h=h)
    heapq.heappush(frontera, (nuevo_estado.f(), nuevo_estado))
```

Se actualiza si:

- Es la primera vez que visitamos esa posición.
- O si el nuevo camino a esa posición es más corto que el anterior.

Entonces se:

1. Guarda el nuevo costo mínimo.
2. Crea un nuevo Estado con el padre, costo y heurística.
3. Se añade a la cola de prioridad con su f().

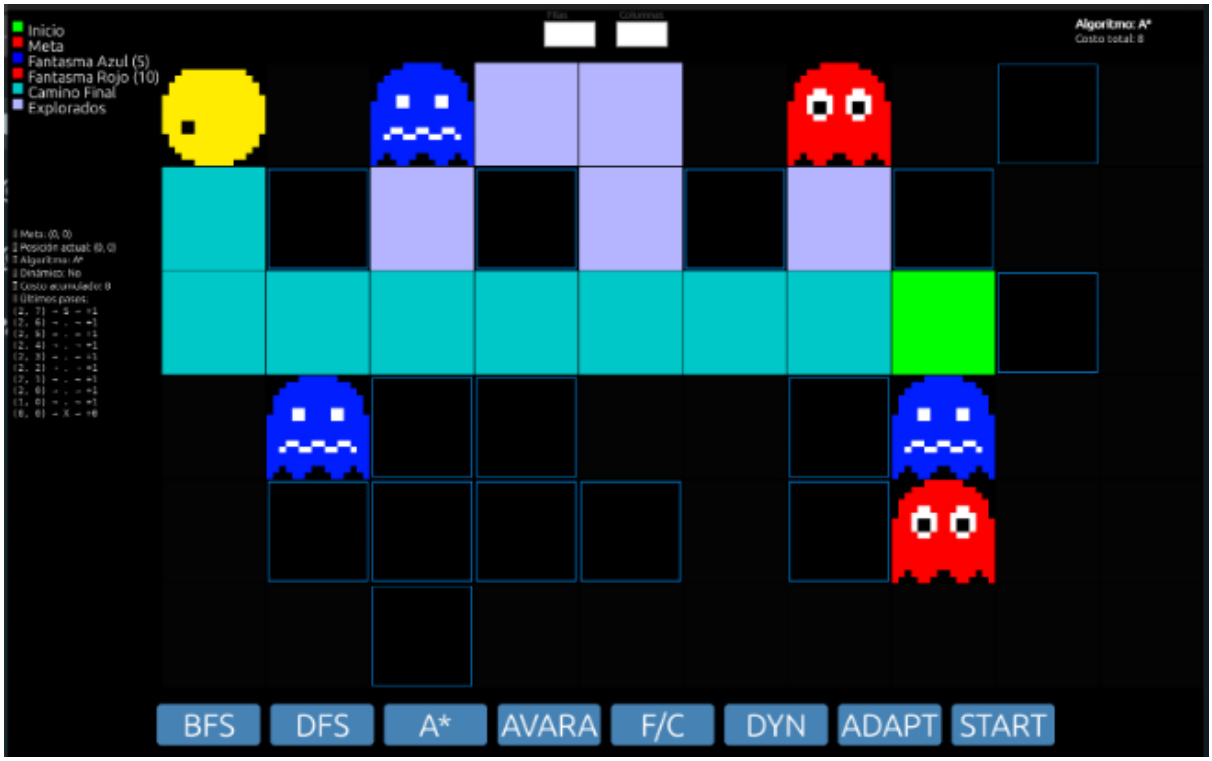
CASO SIN CAMINO

```
return None, float('inf'), explorados
```

Si se recorre todo y no se encuentra camino:

- Retorna None, costo infinito y lo que exploró.

Evidencia visual algoritmo A*



-Algoritmo Greedy (Avara)

Solo utiliza la heurística para elegir la ruta, sin considerar el costo real acumulado. Usa cola de prioridad, pero solo con heurística. Es rápido, pero puede ignorar caminos más baratos.

```python
import heapq
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
from core.costo import costo

def heuristica(pos, meta):
    """Distancia Manhattan como heurística"""
    return abs(pos[0] - meta[0]) + abs(pos[1] - meta[1])

def greedy(laberinto, inicio_pos, meta_pos):

    estado_inicial = Estado(posicion=inicio_pos, h=heuristica(inicio_pos, meta_pos))
    frontera = [(estado_inicial.h, estado_inicial)]
    visitados = set()
    explorados = set()

    while frontera:
        _, actual = heapq.heappop(frontera)
        explorados.add(actual.pos)

        if es_meta(actual, meta_pos):
            camino = actual.reconstruir_camino()
            try:
                costo_total = sum(
                    costo(camino[i], camino[i + 1], laberinto)
                    for i in range(len(camino) - 1)
                )
            except Exception as e:
                print("⚠️ Error al calcular costo del camino en GREEDY:", camino)
                print("❌ Detalle:", e)
                costo_total = float('inf')
            return camino, costo_total, explorados

        for sucesor in generar_sucesores(actual, laberinto):
            if sucesor.pos not in visitados:
                visitados.add(sucesor.pos)
                h = heuristica(sucesor.pos, meta_pos)
                nuevo_estado = Estado(posicion=sucesor.pos, padre=actual, h=h)
                heapq.heappush(frontera, (nuevo_estado.h, nuevo_estado))

    return None, float('inf'), explorados
```

IMPORTACIONES

```python
import heapq
from core.estado import Estado
from core.operador import generar_sucesores
from core.prueba_meta import es_meta
from core.costo import costo
```

Aquí se importan todos los componentes necesarios para el algoritmo:

- **heapq:** permite trabajar con una cola de prioridad.
- **Estado:** representa una celda del laberinto, con información sobre su posición y su heurística.
- **generar_sucesores:** devuelve las posibles posiciones a las que se puede mover el agente desde su estado actual.
- **es_meta:** verifica si un estado corresponde a la meta.
- **costo:** calcula el costo de moverse de una celda a otra (usado solo al final para sumar el camino recorrido).

FUNCIÓN DE HEURÍSTICA

```python
def heuristica(pos, meta):
    """Distancia Manhattan como heurística"""
    return abs(pos[0] - meta[0]) + abs(pos[1] - meta[1])
```

Define la heurística usada por el algoritmo. En este caso, se usa la distancia de Manhattan, ideal para cuadrículas sin diagonales. Esta función se utiliza para estimar "cuánto falta" desde una posición hasta la meta.

Función principal del algoritmo

```
def greedy(laberinto, inicio_pos, meta_pos):
```

Aquí comienza la definición del algoritmo greedy, que tomará:

- **laberinto:** una matriz que representa el entorno.
- **inicio_pos:** la posición inicial del agente.
- **meta_pos:** la posición del objetivo o queso.

Inicialización

```
estado_inicial = Estado(posicion=inicio_pos, h=heuristica(inicio_pos, meta_pos))
frontera = [(estado_inicial.h, estado_inicial)]
visitados = set()
explorados = set()
```

- Se crea el **estado_inicial**, con su heurística ya calculada.
- Se inicializa la frontera como una cola de prioridad, donde se almacenan los estados a explorar. Se ordenan por su heurística h.
- **visitados:** para no volver a visitar un nodo ya explorado.
- **explorados:** para llevar un registro de los nodos realmente procesados.

Bucle de exploración

```
while frontera:
    _, actual = heapq.heappop(frontera)
    explorados.add(actual.pos)
```

Mientras haya nodos por explorar en la cola (frontera), se toma el que tenga menor h (heurística). Este será el estado actual a evaluar. Luego, se registra su posición en explorados.

Verificación de meta

```
if es_meta(actual, meta_pos):
    camino = actual.reconstruir_camino()
    try:
        costo_total = sum(
            costo(camino[i], camino[i + 1], laberinto)
            for i in range(len(camino) - 1)
        )
    except Exception as e:
        print("⚠️ Error al calcular costo del camino en GREEDY:", camino)
        print("❌ Detalle:", e)
        costo_total = float('inf')
    return camino, costo_total, explorados
```

Verifica si el nodo actual es la meta.

Si lo es:

- Reconstruye el camino usando los padre.
- Calcula el costo total real del camino (aunque el algoritmo no lo usa para tomar decisiones).
- Devuelve el camino, el costo y los nodos explorados.

El try-except está por si ocurre un error calculando el costo.

Generación de sucesores

```
for sucesor in generar_sucesores(actual, laberinto):
    if sucesor.pos not in visitados:
        visitados.add(sucesor.pos)
        h = heuristica(sucesor.pos, meta_pos)
        nuevo_estado = Estado(posicion=sucesor.pos, padre=actual, h=h)
        heapq.heappush(frontera, (nuevo_estado.h, nuevo_estado))
```

Genera todos los movimientos válidos desde el nodo actual.

Si el sucesor aún no ha sido visitado:

- Se marca como visitado.
- Se calcula su heurística.
- Se crea un nuevo estado con esa información.
- Se agrega a la frontera para futuras exploraciones.

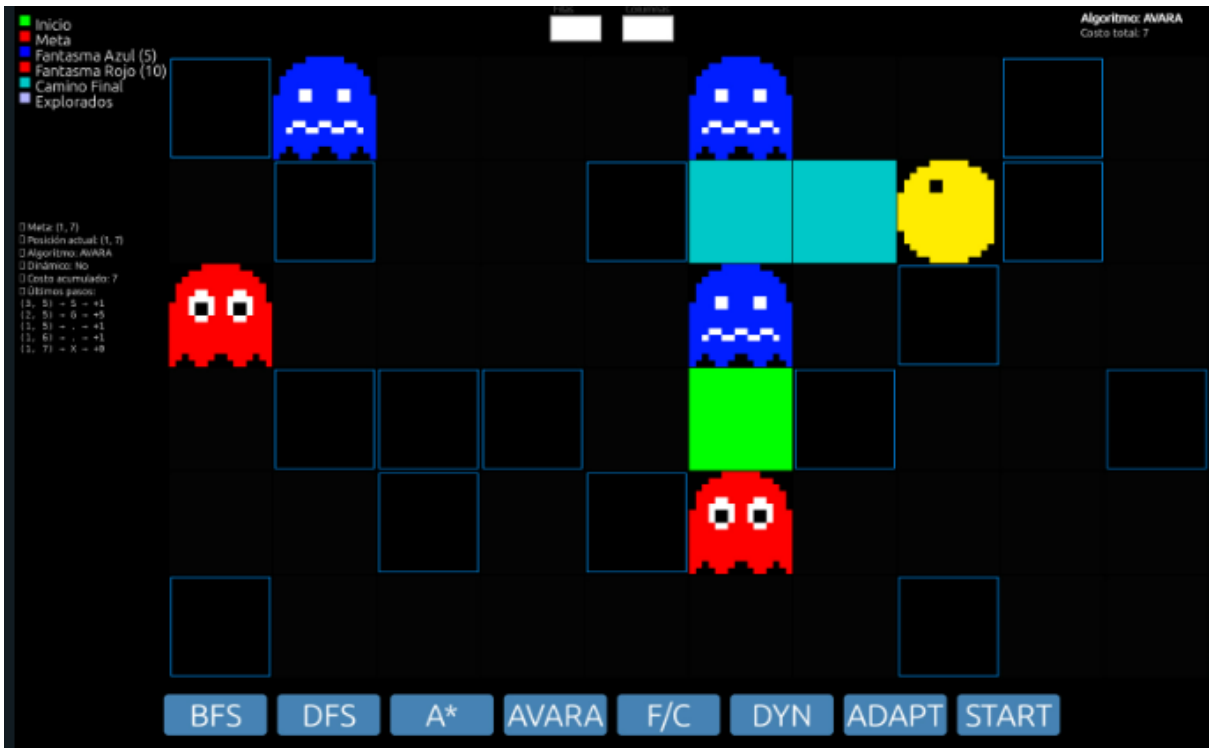
Si no hay solución

```
return None, float('inf'), explorados
```

Si se recorre toda la frontera sin encontrar la meta, se retorna:

- None como camino.
- float('inf') como costo (inaccesible).
- Los nodos que sí se exploraron.

Evidencia visual algoritmo Greedy (Avara)



Estrategia Adaptativa

Selecciona la mejor estrategia en función del entorno, basada en el menor costo total.

```python
# estrategia/adaptativo.py

from agentes.astar import astar
from agentes.bfs import bfs
from agentes.dfs import dfs

def agente_adaptativo(laberinto, inicio, meta):
    """
    Prueba todas las estrategias y elige la de menor costo real.
    """
    estrategias = [
        ("A*", astar),
        ("DFS", dfs),
        ("BFS", bfs),
    ]

    mejor_camino = None
    mejor_costo = float('inf')
    mejor_explorados = set()
    mejor_nombre = "Ninguna"

    for nombre, algoritmo in estrategias:
        print(f"🔍 Ejecutando {nombre}")
        camino, costo, explorados = algoritmo(laberinto, inicio, meta)
        if camino:
            print(f"✅ {nombre} encontró camino con costo {costo}")
            if costo < mejor_costo:
                mejor_camino = camino
                mejor_costo = costo
                mejor_explorados = explorados
                mejor_nombre = nombre
        else:
            print(f"❌ {nombre} no encontró camino")

    if mejor_camino:
        print(f"\n🏆 Estrategia elegida: {mejor_nombre} con costo {mejor_costo}")
    else:
        print("\n🚫 Ninguna estrategia funcionó.")

    return mejor_camino, mejor_costo, mejor_explorados, mejor_nombre
```

Importación de algoritmos

```python
# estrategia/adaptativo.py

from agentes.astar import astar
from agentes.bfs import bfs
from agentes.dfs import dfs
```

Importa los tres algoritmos de búsqueda que se van a evaluar:

- **astar:** búsqueda A* (considera costo + heurística)
- **bfs:** búsqueda por amplitud (explora en anchura)
- **dfs:** búsqueda por profundidad (explora en profundidad)

Cada uno debe devolver al menos (camino, costo_total, explorados)

Función principal

```python
def agente_adaptativo(laberinto, inicio, meta):
```

Define la función que evaluará todos los algoritmos con el mismo laberinto, posición inicial (inicio) y objetivo (meta), eligiendo el más eficiente en términos de costo real del camino.

Lista de estrategias

```python
    estrategias = [
        ("A*", astar),
        ("DFS", dfs),
        ("BFS", bfs),
    ]
```

Agrupa las estrategias a evaluar en una lista de tuplas, donde cada elemento contiene:

- Un nombre identificador del algoritmo.
- La referencia a la función correspondiente.

Esto facilita recorrerlas con un bucle para ejecutarlas de forma dinámica

Inicialización de resultados

```
mejor_camino = None
mejor_costo = float('inf')
mejor_explorados = set()
mejor_nombre = "Ninguna"
```

Inicializa variables que almacenarán los resultados de la mejor estrategia encontrada:

- **mejor_camino:** el camino más eficiente.
- **mejor_costo:** el costo total más bajo.
- **mejor_explorados:** nodos que exploró esa estrategia.
- **mejor_nombre:** el nombre del algoritmo que obtuvo ese camino.

Ciclo de prueba de estrategias

```
for nombre, algoritmo in estrategias:
    print(f"🔍 Ejecutando {nombre}")
    camino, costo, explorados = algoritmo(laberinto, inicio, meta)
```

Recorre todas las estrategias, ejecuta cada una sobre el laberinto, e imprime cuál está corriendo. Luego guarda los resultados de cada algoritmo.

Evaluación de resultados

```
if camino:
    print(f"✅ {nombre} encontró camino con costo {costo}")
    if costo < mejor_costo:
        mejor_camino = camino
        mejor_costo = costo
        mejor_explorados = explorados
        mejor_nombre = nombre
else:
    print(f"❌ {nombre} no encontró camino")
```

Si el algoritmo encontró un camino:

- Se imprime que fue exitoso.
- Se compara su costo con el del mejor camino actual.
- Si es más barato, se actualizan los "mejores" datos.

Si no encontró camino, también se informa.

Elección final

```
if mejor_camino:
    print(f"\n🏆 Estrategia elegida: {mejor_nombre} con costo {mejor_costo}")
else:
    print("\n🚫 Ninguna estrategia funcionó.")
```

Informa cuál fue la estrategia ganadora (la más eficiente en costo).

O avisa que ninguna fue capaz de encontrar un camino.

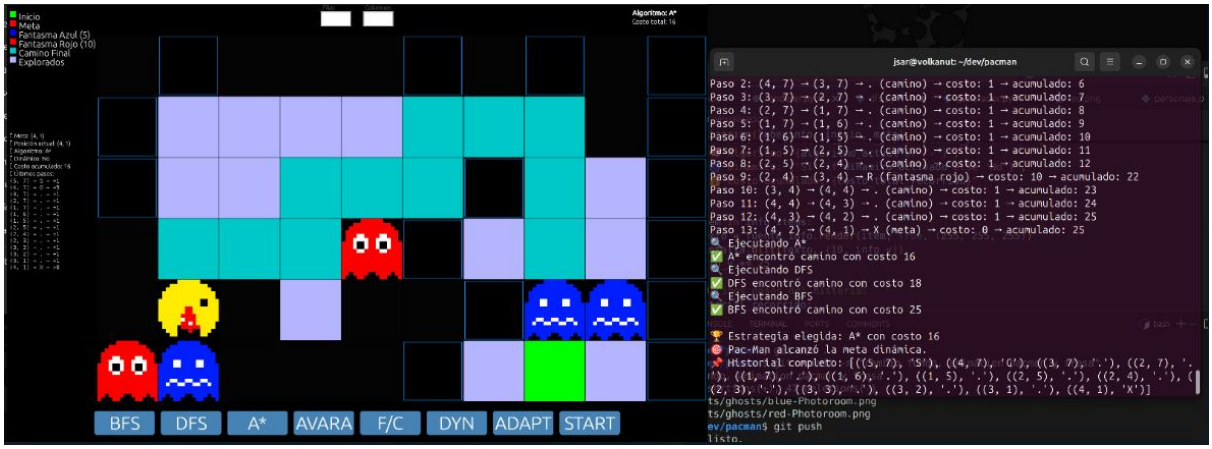
Retorno de resultados

```
return mejor_camino, mejor_costo, mejor_explorados, mejor_nombre
```

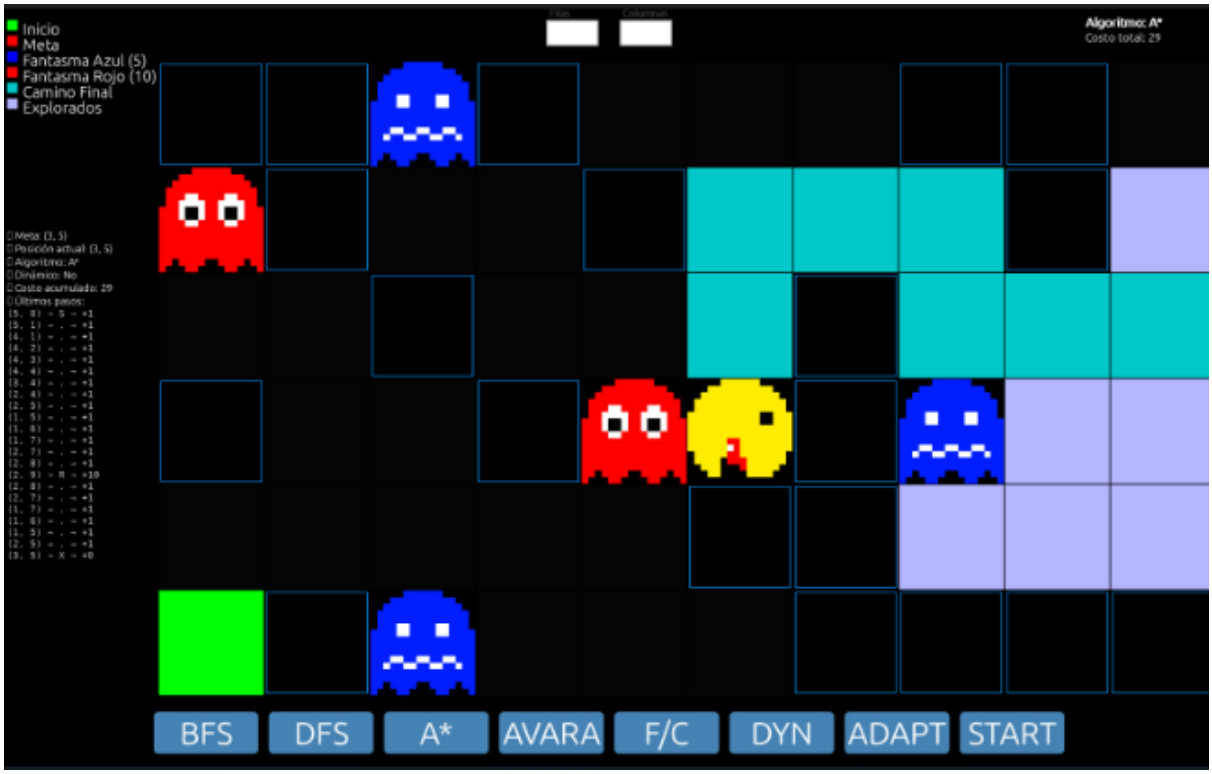
Devuelve los resultados obtenidos:

- El camino óptimo.
- Su costo real.
- Los nodos explorados por esa estrategia.
- El nombre del algoritmo que lo logró.

Evidencia visual algoritmo estrategia Adaptativa



6. Modo Dinámico del Entorno



En este modo, la meta puede moverse aleatoriamente cada ciertos intervalos de tiempo. Además, se permite la aparición o desaparición de paredes y el movimiento aleatorio de los fantasmas (obstáculos con costo). El agente debe recalcular su ruta constantemente y adaptarse a estos cambios. El costo total real acumulado es registrado, así como el historial completo de posiciones recorridas.

7. Interfaz gráfica y visualización

La interfaz gráfica permite visualizar el laberinto, el algoritmo en ejecución, el camino encontrado, y los nodos explorados. Además, se incluye una leyenda de colores para distinguir entre inicio, meta, fantasmas, camino óptimo y explorados. Se pueden ingresar filas y columnas dinámicamente y generar nuevos laberintos personalizados.

8. Evaluación de Resultados

Durante las pruebas, A* mostró consistentemente el menor costo en entornos con fantasmas y cambios dinámicos. DFS fue útil en mapas abiertos, mientras que BFS garantizó la ruta más corta en pasos, pero no necesariamente la de menor costo. El agente adaptativo permitió cambiar automáticamente entre estrategias, mostrando mensajes que indican cuál fue seleccionada y por qué.

9. Indicadores de Cumplimiento

1. Representación de estados, operadores y prueba de meta: Implementado mediante la clase Estado, operadores de movimiento y verificación.
2. Implementación de algoritmos no informados (BFS, DFS): Implementados y visualizados correctamente.
3. Heurística admisible en A*: Distancia Manhattan correctamente utilizada.
4. Algoritmo informado (A*) y avaro (Greedy): Ambos implementados y comparados.
5. Estrategia adaptativa con cambio de algoritmo dinámico: Completamente funcional en modo dinámico.
6. Visualización interactiva y entrada de dimensiones del laberinto: Implementadas mediante PyGame.
7. Modo dinámico: Meta, paredes y fantasmas se mueven aleatoriamente durante la ejecución.

10. Conclusiones

Se logró desarrollar un agente inteligente capaz de navegar en un laberinto cambiante utilizando diferentes técnicas de búsqueda. La implementación del modo dinámico permite que el entorno simule condiciones más cercanas al mundo real. A través de la interfaz gráfica fue posible visualizar el proceso de búsqueda y los resultados del agente.