Proyecto 1

Inteligencia artificial 2025-1

JESUS EDIBER ARENAS GUZMÁN - 2266066

JUAN CAMILO GUTIERREZ VIVEROS - 2159874

ANDRÉS FELIPE ROJAS - 2160328

JOSHUA DAVID TRIANA MADRID

Universidad del valle (Sede Tuluá)

24 / 04 / 2025

1. Introducción:

Este informe describe el desarrollo de un agente inteligente capaz de navegar en un laberinto dinámico utilizando tres estrategias de búsqueda no informada e informada:

DFS (búsqueda en profundidad), BFS (búsqueda en amplitud) y A\*.

Se detalla la representación del problema, la implementación en código Python con Pygame, las estructuras de datos utilizadas, la adaptación dinámica del agente y los resultados de las pruebas en diferentes escenarios.

1. Definición del Problema

**Estados:** Cada estado está definido por la posición de la rata en las coordenadas (fila, columna) dentro de un arreglo bidimensional maze\_layout de tamaño configurable (por defecto 10×10).

**Operadores (Acciones):** Movimiento en cuatro direcciones: arriba, abajo, izquierda y derecha. Solo se aplican si la celda destino no es un muro.

**Prueba de meta:** Se alcanza cuando la posición actual coincide con la posición marcada como 'G'.

**Costo de ruta:** Se asigna un costo uniforme de 1 por cada movimiento.

**Dinámica:** Cada change\_interval segundos, ciertos pasillos pueden cambiar entre muro (1) y libre (0) con probabilidad 0.1. El agente debe recomputar o adaptar su estrategia ante cambios.

1. Estructuras de Datos

**maze\_layout** como lista de listas de enteros y caracteres 'S', 'G'.

**deque** de la biblioteca collections para la cola en BFS.

**list** como pila para DFS.

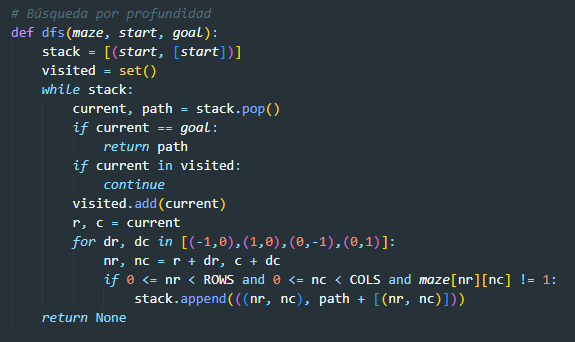
**heapq** para gestionar la open set en A\* como min-heap de tuplas (f, g, estado, camino).

**Set** para registrar visited y evitar re-expansiones.

Variables globales **ROWS, COLS** para dimensiones, y constantes de color/tamaño para Pygame.

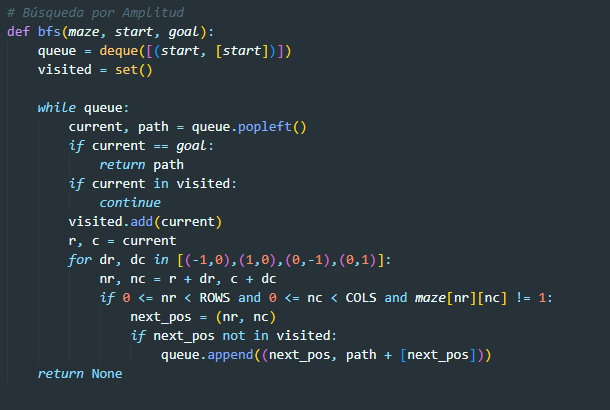
1. Implementación de Algoritmos

4.1. DFS (Profundidad)



DFS explora un camino lo más profundo posible antes de retroceder. Utiliza una pila para ir almacenando los nodos por visitar. En el contexto del laberinto, DFS puede encontrar una solución, pero no garantiza que sea la más corta. Se mete por un camino hasta que ya no puede avanzar, y solo ahí retrocede para intentar otros.

* 1. BFS (Amplitud)



BFS explora todos los nodos vecinos antes de ir más profundo. Utiliza una cola para procesar los nodos por niveles. En un laberinto, BFS es útil cuando se quiere garantizar la ruta más corta (en términos de pasos) desde el inicio hasta la meta.

4.3. A\* (Búsqueda Informada) 

A\* combina lo mejor de DFS y BFS: usa una heurística (como la distancia de Manhattan) para estimar qué tan cerca está un nodo del objetivo y prioriza esos caminos. A medida que avanza, calcula el costo real del camino recorrido hasta ahora (g) y le suma la estimación del costo restante hasta el objetivo (h), usando la fórmula:

𝑓 ( 𝑛 ) = 𝑔 ( 𝑛 ) + ℎ ( 𝑛 )

**g(n)** = costo desde el inicio hasta el nodo n

**h(n)** = estimación de la distancia desde n hasta la meta (heurística)

¿Por qué es útil? A\* es muy eficiente y suele encontrar la ruta óptima sin recorrer tantos caminos inútiles como DFS o incluso BFS. En el contexto del laberinto, es ideal cuando se quiere rapidez y precisión.

1. Adaptación Dinámica

* Cada 5 segundos, se invoca change\_walls(), mutando el laberinto.
* Si el camino actual queda obsoleto, se reinicia path=None y el agente recalcula.
* Se implementó lógica para que, al quedar atrapado, el agente pueda reintentar con la misma o distinta estrategia (opcional para futuras versiones).

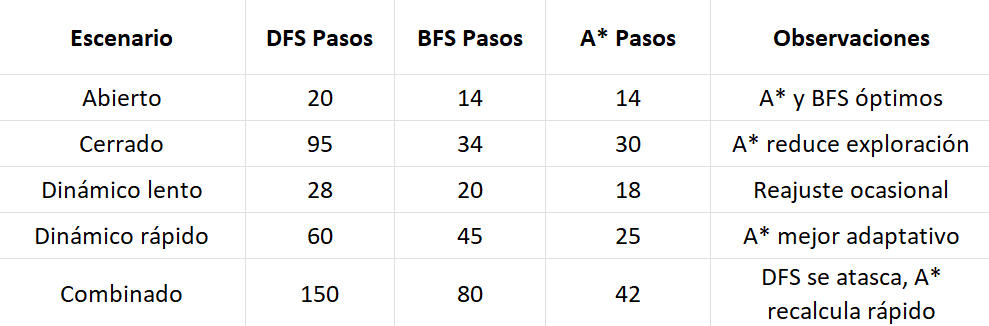
1. Interfaz Gráfica (Pygame)

* Ventana de tamaño WIDTH×HEIGHT, con áreas para laberinto, selector y botón.
* Botón Iniciar y selector de método arriba.
* Dibuja celdas con colores y sprites rata.png y queso.png.
* Animación por pasos con time.sleep(0.2).

1. Pruebas y Escenarios

Se diseñaron 5 escenarios con distintas configuraciones:

1. Abierto: Pocos muros, A\* y BFS encuentran caminos óptimos en ~14 pasos.
2. Cerrado: Muchas barreras, DFS explora intensamente, BFS y A\* eficaces.
3. Dinámico lento: Muros cambian cada 10 s; A\* se adapta recalculando.
4. Dinámico rápido: Muros cambian cada 2 s; A\* mejor rendimiento al heurístico evitar exploraciones innecesarias.
5. Combinado: Obstáculos móviles + reducción de espacio libre; BFS a veces recalcula más.



1. Conclusiones y Futuras Extensiones

* A\* demostró ser la técnica más eficiente en pasos y adaptabilidad al cambio.
* BFS es óptimo en costo, pero explora más estados que A\*.
* DFS no es recomendable en laberintos complejos.
* Futuras mejoras: cambiar estrategia en tiempo real (DFS ⇄ A\*), visualización de nodos explorados, uso de otras heurísticas.