Informe Final del Proyecto de Inteligencia Artificial

1. Introducción

Este proyecto consiste en el desarrollo de una simulación en la que un ratón debe encontrar un queso en un laberinto, utilizando distintos algoritmos de búsqueda. Se ha implementado una interfaz para configurar el estado inicial y visualizar el árbol de búsqueda generado por cada algoritmo.

2. Objetivos

Objetivo General:
Desarrollar un sistema interactivo capaz de resolver un laberinto utilizando algoritmos de búsqueda (BFS, DFS y A*), visualizando el proceso y el árbol generado.

Objetivos Específicos:

- Diseñar una representación gráfica del laberinto.
- Implementar los algoritmos BFS, DFS y A*.
- Visualizar el árbol de búsqueda con NetworkX.
- Integrar una interfaz en Tkinter para cargar configuraciones iniciales.

3. Herramientas y Librerías Utilizadas

- **Python 3.13**
- **Tkinter:** Interfaz gráfica.
- **NumPy:** Manipulación de matrices.
- **NetworkX:** Visualización de grafos.
- **Matplotlib:** Integración gráfica.
- **PIL (Pillow):** Manejo de imágenes en Tkinter.

4. Diseño del Sistema

4.1 Representación del Laberinto

El laberinto se representa como una matriz, donde:

- `'1'` representa una pared
- `'0'` representa un espacio libre
- El ratón y el queso tienen posiciones específicas.

4.2 Formato del archivo de entrada

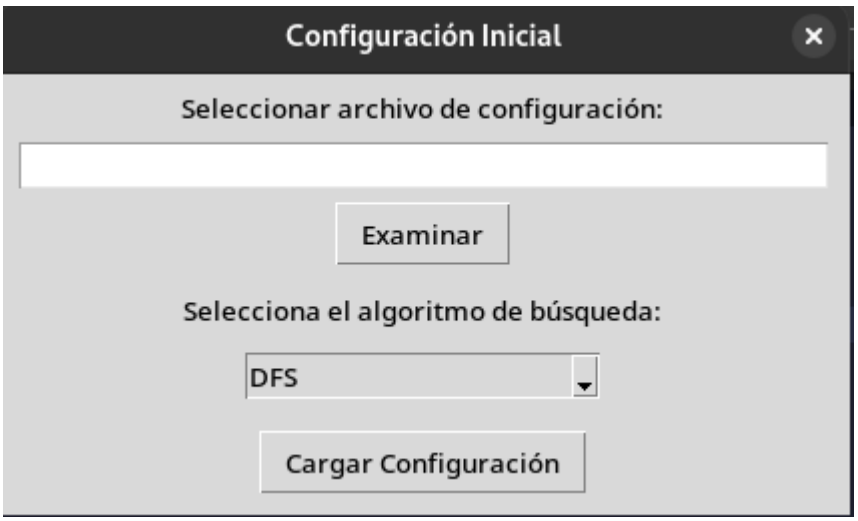
El archivo `.txt` cargado debe seguir el siguiente formato:

```
0 0
2 3
0 0 0 1
1 0 1 0
0 0 0 0
```

5. Interfaz Gráfica

Se utilizó **Tkinter** para permitir al usuario:

- Seleccionar el archivo de configuración.
- Escoger el algoritmo deseado (DFS, BFS o A*).
  Ejecutar la búsqueda y visualizar el árbol resultante.



6. Algoritmos Implementados y Dinámica del Entorno

Durante el desarrollo del proyecto se implementaron tres algoritmos clásicos de búsqueda: búsqueda en amplitud (BFS), búsqueda en profundidad (DFS) y búsqueda A*. Cada uno de ellos fue adaptado para interactuar con un entorno dinámico, donde tanto el laberinto como la posición del objetivo (el queso) pueden cambiar durante la ejecución.

Algoritmo	**Estructura**	**Orden de exploración**	**Garantiza el camino más corto**
BFS	Cola	Horizontal (nivel por nivel)	Solamente si todos los pasos cuestan igual
DFS	Pila	Vertical (ya que va hacía el fondo	No garantiza el camino más corto
A*	Cola de prioridad	Inteligente	Si la heurística es admisible

6.1 Consideraciones Generales

Cada nodo del árbol de búsqueda representa un estado del laberinto con la posición del ratón, la posición del queso, y la distribución actual de paredes. El sistema tiene la capacidad de simular un entorno **dinámico**, es decir, el laberinto puede mutar de forma controlada en ciertas profundidades del árbol de búsqueda. Estas mutaciones incluyen:

- **Movimientos aleatorios del queso:** En ciertos pasos, la posición del queso puede cambiar a una celda libre distinta.
- **Aparición de nuevas paredes:** Se pueden añadir obstáculos en celdas previamente libres.
- **Eliminación de paredes existentes:** Se abren caminos en lugares que antes eran muros.

Estas mutaciones ocurren, por ejemplo, cada 3 niveles de profundidad, lo que introduce un componente de imprevisibilidad y simula un entorno cambiante similar a problemas del mundo real.

6.2 Búsqueda en Amplitud (Breadth-First Search - BFS)

Este algoritmo utiliza una cola (`queue`) para explorar el laberinto **nivel por nivel**, garantizando que se encuentra el camino más corto en número de pasos, siempre que el entorno no cambie de manera significativa entre nodos vecinos.

Ventajas:

- Encuentra el camino más corto si los costos son iguales.
- Sencillo de implementar.

Desventajas:

- Requiere mucha memoria al generar múltiples nodos al mismo nivel.
- Su rendimiento puede verse afectado cuando el laberinto muta de forma inesperada, ya que puede invalidar caminos previamente viables.

Lógica específica en el proyecto:

- Se verifica si el nodo actual cumple la condición de meta (`is_goal()`).
- Si el nodo está en una profundidad múltiplo de 3, se aplica la función `mutate()` que puede cambiar paredes y la posición del queso.
- Los hijos generados se agregan a la cola de búsqueda.

6.3 Búsqueda en Profundidad (Depth-First Search - DFS)

DFS utiliza una **pila** (`stack`) para profundizar en una rama del árbol lo más posible antes de retroceder. Este enfoque puede ser útil si el queso está lejos en términos de profundidad del árbol.

Ventajas:

- Menor uso de memoria que BFS.
- Puede encontrar soluciones rápidamente si el queso está profundo.

Desventajas:

- No garantiza el camino más corto.
- En un laberinto dinámico puede quedar atrapado en caminos que dejan de ser válidos debido a mutaciones.

Lógica específica en el proyecto:

- Se prioriza la exploración de los últimos hijos generados (último en entrar, primero en salir).
- Se invierte la lista de hijos para simular orden de expansión en sentido horario.
- Se aplica la mutación si la profundidad es múltiplo de 3.

6.4 Búsqueda A*

El algoritmo A* combina la búsqueda informada con el costo acumulado y una **heurística** que estima la distancia al objetivo. En este caso, se utiliza la **distancia de Manhattan** como heurística.

Ventajas:

- Alta eficiencia al priorizar los caminos más prometedores.
- Puede adaptarse al entorno dinámico si la heurística se actualiza con los cambios.

Desventajas:

- Requiere un diseño cuidadoso de la función heurística.
- Puede volverse ineficiente si las mutaciones invalidan el estado evaluado.

Lógica específica en el proyecto:

- Se utiliza una cola de prioridad basada en `f(n) = g(n) + h(n)`, donde `g(n)` es el costo acumulado y `h(n)` la heurística.
- Cada nodo generado recalcula su heurística si el queso cambia de lugar.
- También se aplican mutaciones del laberinto cada 3 niveles.

6.5 Observaciones sobre el Entorno Dinámico

La inclusión de mutaciones hace que este proyecto no solo evalúe la eficiencia de los algoritmos de búsqueda, sino también su **capacidad de adaptación a entornos que cambian con el tiempo**. Esto simula situaciones reales como:

- Rutas que se bloquean por eventos imprevistos.

- Cambios en la posición del objetivo (como en aplicaciones de robótica o videojuegos).
- Nuevas rutas que se abren con el tiempo.

Este comportamiento dinámico añade una capa de complejidad al problema clásico del laberinto y permite estudiar cómo se comportan los algoritmos tradicionales en un contexto más realista.

7. Evaluación y Pruebas de los Algoritmos de Búsqueda

7.1 Objetivo

Evaluar el desempeño de los algoritmos BFS, DFS y A* en diferentes configuraciones del laberinto, considerando tanto escenarios estáticos como dinámicos. Se analizó el número de nodos generados, la longitud de la solución, la capacidad de adaptación ante mutaciones y el tiempo de respuesta.

7.2 Configuraciones de prueba

Se realizaron pruebas sobre laberintos de diferentes dimensiones y dificultades. En algunos casos, el laberinto permanecía estático, mientras que en otros, se aplicaban mutaciones dinámicas cada 3 pasos, como:

- Aparición o eliminación de paredes
- Cambio de posición del queso

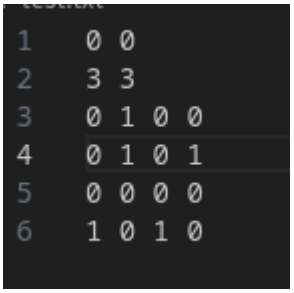
7.3 Métricas consideradas

Para cada algoritmo, se analizaron las siguientes métricas:

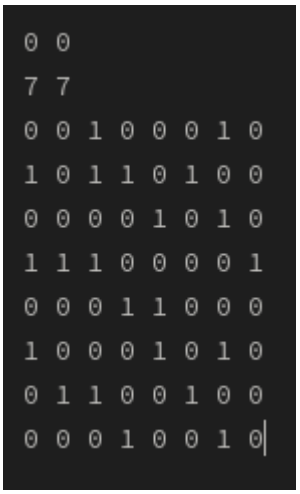
- **Nodos generados**: cantidad de nodos explorados.
- **Longitud del camino**: cantidad de pasos desde el inicio hasta el objetivo.
- **Tiempo de ejecución**: medido con `time.time()` o `time.perf_counter()`.
- **Capacidad de adaptación**: comportamiento del algoritmo ante cambios.

7.4 Matrices de prueba

Matriz 4x4, posición inicial del ratón 0,0 y posición del queso 3,3



Matriz 8x8, posición inicial del ratón 0,0 y posición inicial del queso 7,7



7.5 Resultados obtenidos

Algoritmo	Escenario	Longitud del camino	Tiempo (s)	Adaptación a mutaciones
DSF	Laberinto(8x8) con mutación	88	0.00306534 seg	Adaptable pero no óptimo
BSF	Laberinto(8x8) con mutación	16	0.01225 seg	Adaptable
A*	Laberinto(8x8) con mutación	13	0.000639438 seg	Excelente adaptación
DSF	Laberinto(8x8) sin mutación	15	0.000309467 seg	No aplica
BSF	Laberinto(8x8) sin mutación	15	0.0028927 seg	No aplica
A*	Laberinto(8x8) sin mutación	15	0.00047874 seg	No aplica
DFS	Laberinto(4x4) sin mutación	7	0.0017049312591552734 seg	No aplica

BSF	Laberinto(4x4) sin mutación	7	0.0008585453033447266 seg	No aplica
A*	Laberinto(4x4) sin mutación	7	0.0016136169433359375 seg	No aplica
DFS	Laberinto(4x4) con mutación	5	0.0011322498321533203 seg	Adaptable pero depende de la matriz
BSF	Laberinto(4x4) con mutación	5	0.0019841194152832030 seg	Adaptable
A*	Laberinto(4x4) con mutación	7	0.0015840530395507812 seg	Adaptable

7.6 Análisis

- BFS garantiza la solución más corta en laberintos estáticos, pero genera muchos nodos, especialmente en configuraciones amplias.
- DFS tiene un bajo consumo de memoria, pero puede desviarse en caminos poco prometedores, resultando en trayectos más largos.
- A* demostró un equilibrio entre eficiencia y óptima solución, especialmente en presencia de mutaciones, gracias al uso de heurística.

Además, todos los algoritmos fueron modificados para recalcular los nodos cuando el laberinto cambiaba, permitiendo una reevaluación dinámica de caminos viables.

7.6 Conclusión

Los resultados muestran que A* es el algoritmo más eficiente y robusto para matrices grandes y entornos dinámicos con mutaciones, mientras que BFS es ideal cuando se busca una solución garantizada en entornos estáticos. DFS es útil para búsquedas rápidas donde la óptima solución no es prioritaria.

8. Conclusiones

- Se logró implementar con éxito los tres algoritmos propuestos.
- La visualización gráfica permite analizar la eficiencia y comportamiento de cada algoritmo.
- La estructura modular del código facilita futuras ampliaciones (por ejemplo, nuevos heurísticos o entornos dinámicos).