Taller N°2

Tema: Uso de algoritmos de búsqueda

Integrantes:

- Ñacato Aurora
- Parra Carla
- Paredes Josue

1. Uso de Algoritmos de Búsqueda

1.1. Laberinto 1:

Este laberinto es de tamaño reducido, con una estructura clara y un camino bastante directo entre la entrada (E) y la salida (S). Fue una excelente prueba inicial para verificar que los algoritmos de búsqueda estuvieran correctamente implementados y que la visualización funcionara como se esperaba.

1.1.1. Solución 1: Búsqueda en Anchura (BFS)

Observaciones:

- Construcción del grafo: Se representó cada celda transitable (, E, S) como un nodo, y se conectaron celdas adyacentes en las cuatro direcciones cardinales (no diagonales).
- Visualización: El camino hallado por BFS fue graficado en color azul claro sobre el laberinto, confirmando que conecta correctamente la entrada con la salida.
- Precisión: BFS encontró el camino más corto en cuanto a número de pasos. La estructura lineal del laberinto facilitó un recorrido eficiente sin retrocesos.
- Validación: Se comprobó la existencia de E y S, y el algoritmo notificó correctamente si no había solución (lo cual no fue el caso aquí).
- Resultado: BFS fue exitoso, y la solución generada fue guardada como imagen en laberinto1_resuelto.png.

1.1.2. Solución 2: Búsqueda A* (A-Star)

Observaciones:

- Heurística aplicada: Se utilizó la distancia Manhattan entre cada celda y la meta (S) como función heurística.
- Eficiencia: A* encontró el mismo camino que BFS, pero explorando menos nodos gracias a su heurística.
- Visualización: La ruta generada se mostró en color naranja, y coincide con la hallada por BFS, dado que ambos trabajan sobre un grafo no ponderado y sin obstáculos adicionales.

- Resultado: A* ofreció una solución óptima y eficiente, y el resultado fue visualizado junto al de BFS para comparación directa.

- Resultado: A* demostró ser una solución óptima en tiempo y calidad de recorrido.

1.1.3. **Visualizaciones**

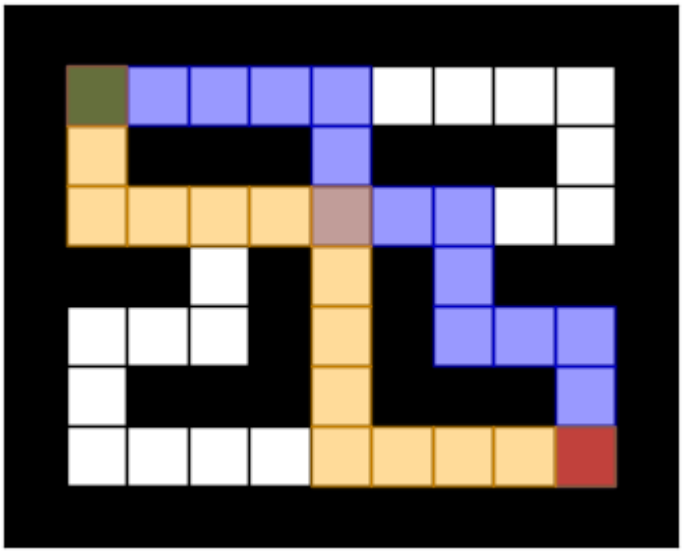


Gráfico Laberinto1

1.2. Laberinto 2:

Este laberinto es más complejo que el anterior, con múltiples rutas, bifurcaciones y una mayor cantidad de espacios transitables. Presenta un mayor reto en términos de exploración y validación de caminos.

1.2.1. Solución 1: Búsqueda en Anchura (BFS)

Observaciones:

- Exploración completa: BFS exploró sistemáticamente todos los niveles de profundidad desde la entrada, garantizando encontrar el camino más corto a la salida.

- Construcción del grafo: Similar al laberinto 1, se usaron como nodos todas las celdas vacías y conectables.

- Visualización: El camino resultante se graficó en color azul claro, confirmando la precisión del recorrido más corto posible.

- Rendimiento: Aunque BFS tuvo que recorrer más nodos que en el laberinto 1, completó la tarea sin complicaciones.

- Resultado: Se generó y guardó la imagen laberinto2_resuelto.png.

1.2.2. Solución 2: Búsqueda A* (A-Star)

Observaciones:

- Heurística efectiva: Gracias al mayor tamaño del laberinto, A* mostró una ventaja en eficiencia al reducir el número de nodos explorados.

- Ruta encontrada: El camino hallado por A* fue idéntico en longitud al de BFS, pero con menor costo computacional.

- Visualización: La ruta A* fue dibujada en color naranja y aparece sobrepuesta con la de BFS para su comparación.

1.2.3. Visualizaciones

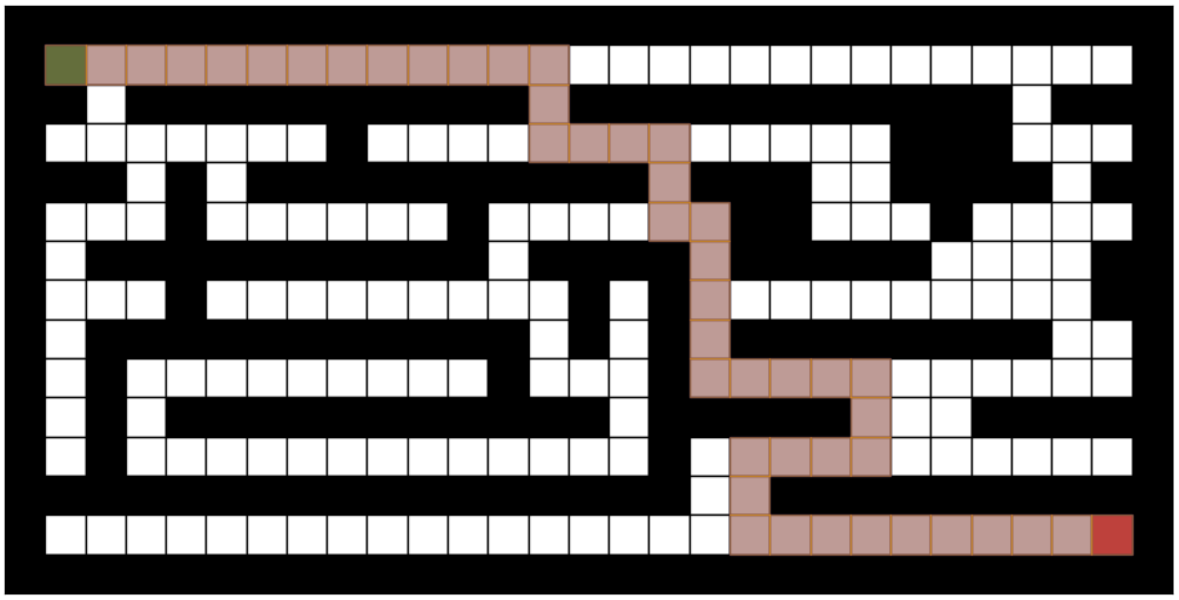


Gráfico Laberinto2

1.2.4. Conclusiones laberinto 1 & 2

- BFS garantiza siempre el camino más corto, siendo ideal para laberintos pequeños o cuando se desea asegurar la solución más eficiente.

- A*, con una buena heurística, ofrece un excelente balance entre tiempo de cómputo y calidad del camino, y fue especialmente útil en el laberinto 2.

- Ambos algoritmos funcionaron correctamente y las imágenes generadas permiten visualizar y validar los resultados.

- La representación del laberinto como grafo fue efectiva, permitiendo adaptar fácilmente otros algoritmos en el futuro.

1.3. Laberinto 3:

1.3.1 Solución 1: Búsqueda en Anchura (BFS)

1.3.1.1 Observaciones:

1. **Estructura del laberinto**

    a. El laberinto3.txt es un laberinto grande y denso, con paredes (#) y espacios ( ), así como una entrada E y una salida S.
    b. La estructura admite múltiples caminos potenciales, pero no todos conducen a la meta.

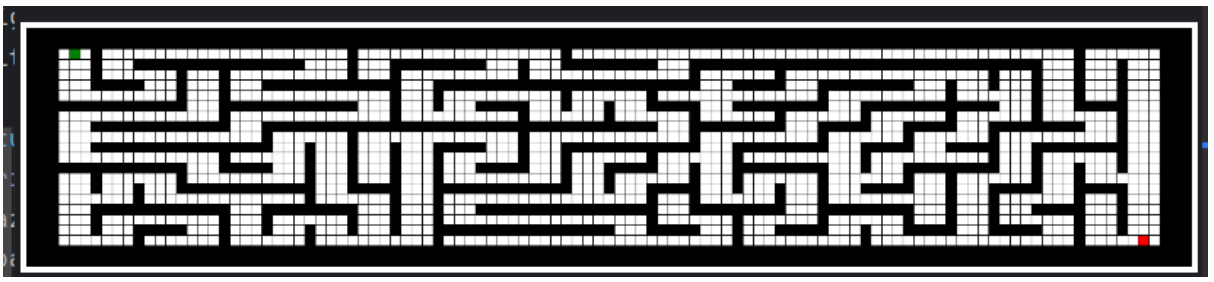
2. **Construcción del grafo**

    a. Cada celda vacía, E o S se transforma en un nodo.
    b. Las conexiones válidas son solo entre celdas adyacentes (arriba, abajo, izquierda, derecha).

3. **Visualización**

    a. El camino final encontrado por BFS fue dibujado sobre el laberinto usando matplotlib en color cyan.
    b. La visualización permite validar que el camino conecta E con S efectivamente.

Figura 1
Entrada y salida del laberinto 3



4. **Detección de errores**

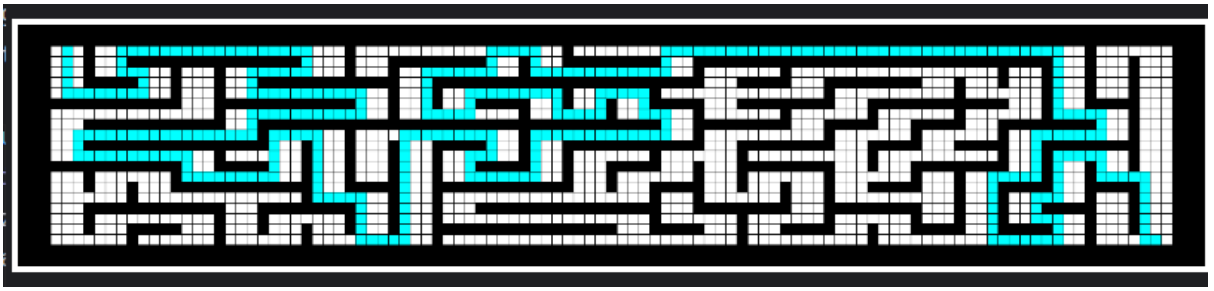
    a. Se incluyó validación para asegurar que E y S existan.
    b. El algoritmo devuelve un mensaje si no hay solución.

5. BFS resolvió el laberinto exitosamente, encontrando el camino más corto de la entrada a la salida.

a. El enfoque basado en grafos es muy efectivo para representar laberintos bidimensionales.
b. Visualizar el resultado es clave para depurar errores y validar la solución encontrada.
c. Este tipo de solución es escalable y puede adaptarse fácilmente a otros algoritmos (como DFS, A*).

Figura 2

Muestra la solución del laberinto 3 con BFS



1.3.2 Solución 2: Búsqueda en Amplitud(DFS)

1.3.2.1 Observaciones:

1. DFS no garantiza el camino más corto

a. DFS explora profundamente un camino hasta que encuentra la meta, sin considerar si es el más corto.
b. En laberintos con muchos pasadizos puede terminar siguiendo rutas más largas o menos eficientes.
c. En nuestro caso, el camino que encontró DFS fue funcional pero más extenso que el de BFS**.**

2. La solución de DFS depende del orden de vecinos

a. Si el orden de las conexiones cambia, el resultado de DFS también cambia, lo cual lo hace no determinista en términos de eficiencia.

3. Mayor profundidad y retroceso

a. DFS tiende a meterse en caminos sin salida y luego retrocede (backtracking).
b. Esto produce un recorrido más largo incluso cuando hay caminos cortos muy cerca del punto de inicio.

4. Menor consumo de memoria que BFS

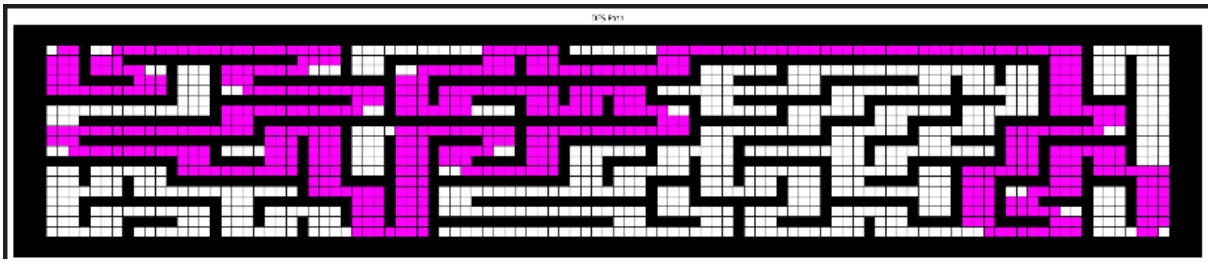
a. DFS almacena menos caminos al mismo tiempo (pila en vez de cola).
b. Esto lo hace más eficiente en uso de memoria, especialmente en laberintos muy grandes, aunque en este laberinto (mediano), la diferencia no es crítica.

5. Visualización del recorrido DFS

a. Al observar el camino pintado por DFS en la gráfica, se nota que va en zigzag o atraviesa zonas más extensas del laberinto, incluso regresando cerca del punto de partida antes de llegar a la meta.

Figura 3

Solución del laberinto 3 con DFS



1.3.3 Conclusiones:

1. BFS siempre encuentra el camino más corto en laberintos como este (grafos no ponderados).
2. DFS puede encontrar un camino, pero no garantiza que sea el más corto y puede tomar rutas más largas o erráticas.
3. En laberintos muy ramificados, DFS puede ser más rápido, pero menos eficiente en calidad del camino.

1.3.4 Métricas para Evaluar Algoritmos de Búsqueda en Laberintos

En la Tabla 1 se muestran las posibles métricas que pueden ser usadas para evaluar los algoritmos:

Tabla 1

Métricas de evaluación de los algoritmos de búsqueda usados para el Laberinto 3

Métrica	BFS	DFS
Longitud del camino	Mínimo garantizado	Puede ser más largo
Nodos visitados	Muchos	Menos
Uso de memoria	Alto	Bajo
Tiempo de ejecución	Medio	Ràpido pero errático
Determinismo	Alto	Depende del orden

2. Optimización de colonia de hormigas

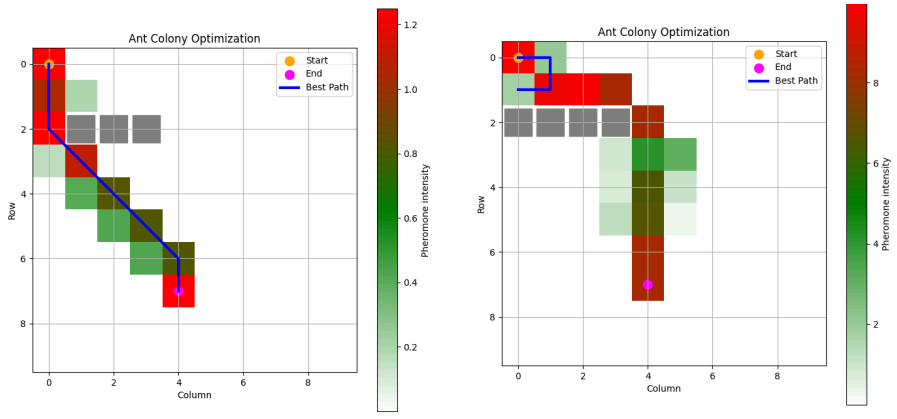
2.1. Análisis del código

El código tiene implementado un modelo de optimización de colonia de hormigas, en el que se simula el comportamiento de hormigas al buscar fuentes de alimento, depositando feromonas en el camino para guiarse mutuamente.

En la primera ejecución del código se obtienen estos resultados (véase figura 1) en donde se imprime el resultado del caso de estudio 1 a la izquierda y el caso de estudio 2 a la derecha. Su diferencia radica en el obstáculo extra en la posición `(0, 2)`.

Figura 1

Resultado caso estudio 1 y 2



2.2. ¿Qué ocurre con el segundo caso de estudio?

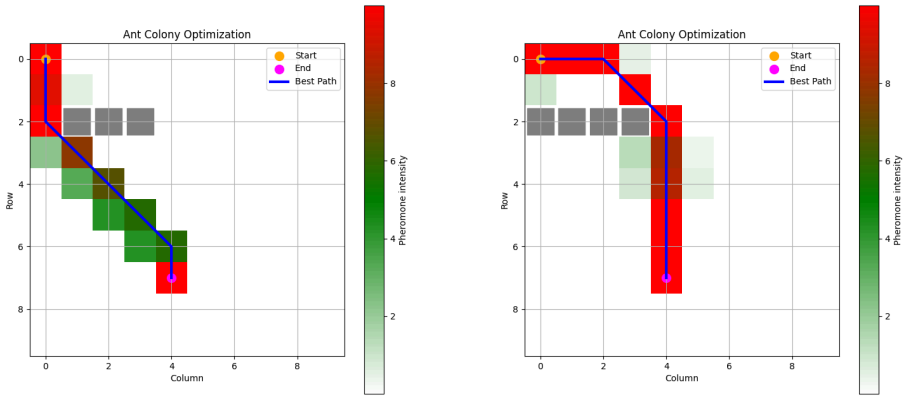
En el segundo caso de estudio, el algoritmo de Colonia de Hormigas (ACO) no logra encontrar el camino hasta el punto objetivo. Esto ocurre porque hay un obstáculo muy

cercano al punto de inicio, lo que reduce las opciones de movimiento iniciales de las hormigas. El problema se origina en la función _get_neighbors(position), que busca las celdas disponibles alrededor de la posición actual. Debido a la ubicación del obstáculo, las hormigas quedan atrapadas en un conjunto limitado de vecinos, sin poder explorar rutas óptimas hacia el destino. Además, La selección de la próxima posición (_select_next_position) tiende a favorecer caminos repetitivos debido a la falta de alternativas viables.

Para resolver este problema, se pueden aplicar mejoras como:
- Reinicio de hormigas si quedan atrapadas sin opciones válidas.
- Filtro en _get_neighbors para evitar zonas con demasiados obstáculos.
- Modificación de la selección de la mejor ruta con la distancia total del camino.

Estas mejoras permiten que el algoritmo mantenga su capacidad de exploración, en la figura 2 se muestran los resultados con la implementación:

Figura 2
Resultado caso estudio 1 y 2 con cambios de código



2.3. Describir los parámetros del modelo (¿Qué propósito tiene cada parámetro del modelo?)

- **start**
  Representa la posición inicial, de donde parten las hormigas.
- **end**
  Representa la posición final u objetivo en la cuadrícula.
- **obstacles**
  Representa una lista de posiciones en x e y, que son bloqueos en la cuadrícula por donde las hormigas no pueden pasar dificultando los caminos óptimos.
- **grid_size=(10, 10)**
  Representa el área de búsqueda(ancho y largo), por defecto 10,10
- **num_ants=10**
  Número de hormigas
  Representa cuántas hormigas exploran la cuadrícula en cada iteración.
- **evaporation_rate=0.1**
  Tasa de evaporación de feromonas
  Representa la rapidez con la que las feromonas desaparecen.
- **alpha=0.1**

Importancia de las feromonas
Ajusta el peso de las feromonas en la selección de caminos, con valores altos las hormigas recorren caminos ya explorados con más frecuencia

- **beta=15**
  Importancia de la heurística
  Representa el peso de la distancia al objetivo, con valores altos hacen que las hormigas prioricen moverse hacia la meta, incluso si las feromonas son bajas.
- **pheromones**
  Matriz de feromonas
  Representa la cantidad de feromonas en cada celda de la cuadrícula. Se actualiza con la evaporación y depósito de feromonas. Su estado inicial es una matriz de unos.
- **best_path**
  Almacena la ruta más corta descubierta hasta el momento y se actualiza en cada iteración.

2.4. ¿Será que se puede utilizar este algoritmo para resolver el Travelling Salesman Problem (TSP)?¿Cuáles serían los pasos de su implementación?

Si se puede utilizar el algoritmo de Optimización de colonia de hormigas, ya que permite encontrar las rutas más eficientes con el uso de feromonas. Prueba varias rutas y va optimizando en cada paso. además se pueden modificar los agentes (hormigas) para que cada una explore una solución.

Los pasos para implementar este algoritmo en el problema de TSP son:

1. Representar las ciudades como nodo (Cada ciudad en el TSP se convierte en un nodo dentro de un grafo.)
2. Inicializar feromonas en cada ruta con la matriz de unos, cada enlace entre ciudades tiene un nivel inicial de feromonas, que se actualizará con cada iteración.
3. Simular el recorrido de las hormigas
   3.1. Cada hormiga comienza en una ciudad aleatoria.
   3.2. Construye un camino visitando todas las ciudades sin repetir.
   3.3. Usa las feromonas para decidir la siguiente ciudad.
4. Evaluar y actualizar las feromonas
   4.1. Evaporar las feromonas
   4.2. Refuerza con el mejor camino encontrado con la distancia
5. Iterar hasta encontrar una buena solución

Conclusiones:

- Con el reinicio de hormigas atrapadas, se evita que las hormigas se queden estancadas en zonas sin salida, permitiéndoles explorar nuevas rutas en cada iteración. Esto garantiza que el algoritmo no se detenga prematuramente y tenga más oportunidades de encontrar el camino óptimo.

- La penalización de vecinos con demasiados obstáculos en el filtro en _get_neighbors ayudó a evitar áreas con demasiados obstáculos, lo que mejoró la movilidad de las hormigas. Con esta implementación el algoritmo ayuda a reducir la probabilidad de

quedar atrapado en caminos cerrados y promoviendo un mejor balance entre exploración y explotación.

- Al modificar la evaluación de la mejor ruta usando la distancia total del camino, en lugar de sólo el número de pasos, se logra que el algoritmo elija verdaderamente el trayecto más eficiente al tomar la mínima la distancia recorrida.