TALLER 2

INTELIGENCIA ARTIFICIAL

Fecha: 28 de septiembre de 2024
Integrantes:

- Edgar Córdova (00342111)
- Jair Criollo (00341239)
- Milene Muñoz (00341391)

Este documento presenta las soluciones y análisis desarrollados en el marco del Taller 2. En este taller, se abordaron dos tipos de problemas: la resolución de laberintos mediante la implementación de algoritmos de búsqueda, y la optimización de rutas utilizando el enfoque inspirado en las colonias de hormigas (Ant Colony Optimization). A través de estas actividades, se buscó no solo resolver los problemas propuestos, sino también comparar el desempeño de diferentes algoritmos, evaluar su eficiencia y analizar su aplicabilidad en situaciones reales.

Las preguntas planteadas en el taller fueron respondidas aplicando herramientas avanzadas de inteligencia artificial y técnicas computacionales.

A continuación, se detallan las soluciones implementadas, la evaluación de los objetivos del taller y las conclusiones derivadas del proceso.

1. USO DE ALGORITMOS DE BÚSQUEDA

OBJETIVO: utilizar cualquier algoritmo de búsqueda para resolver los 3 laberintos propuestos, el reto es poder visualizar/representar los resultados, adicionalmente poder comparar al menos 2 algoritmos de búsqueda y mirar cómo se comportan para cada laberinto.

A. Leer el laberinto y representarlo como un grafo

Implementar una función que permita transformar la información del laberinto en un grafo, buscar la mejor manera de representar la información del laberinto.

Se implementa el método `get_graph` en la clase `MazeLoader` para convertir el laberinto (que se representa como una cuadrícula de caracteres) en una estructura de datos de grafo. El propósito de este método es crear un grafo en el que:

- Los nodos representan celdas transitables en el laberinto (celdas que no son paredes #)
- Las aristas representan caminos válidos entre celdas adyacentes (arriba, abajo, izquierda y derecha)

Estructura del grafo resultante

El resultado obtenido con `get_graph` es de una clase similar a un diccionario (defaultdict), en donde:

- Cada celda transitable (nodo) es una clave (key)
- El valor (value) de cada clave es una lista de celdas transitables adyacentes (conexiones o aristas)

¿ Por qué representar el laberinto como un grafo?

En este problema, el grafo se centra únicamente en las celdas transitables, ignorando las paredes (#), lo que reduce cálculos innecesarios y nos porporciona una representación eficiente del laberinto.

Muchos algoritmos de búsqueda de caminos como por ejemplo BFS, DFS, Dijkstra o A* están diseñados para trabajar en grafos. Por este motivo, convertir el laberinto en un grafo es crucial, ya que estos algoritmos pueden aplicarse directamente, ayudando a explorar el laberinto de manera eficiente y simplificando así la búsqueda de caminos.

Otra de las razones para representar el laberinto como un grafo es que una vez alcanzado el objetivo con el algoritmo, este permite rastrear fácilmente el camino de vuelta desde el objetivo hasta el inicio del laberinto utilizando las conexiones (reconstrucción del camino).

B. Aplicar algoritmos de búsqueda

Una vez obtenido el grafo, aplicar al menos dos algoritmos de búsqueda para comparar su comportamiento, efectividad y rapidez. ¿Se puede establecer alguna métrica para evaluar los algoritmos en este problema?

Para resolver los laberintos propuestos, se utilizó 3 enfoques diferentes: BFS, DFS y A*. A continuación se describe lo realizado para la implementación de cada uno de los algoritmos mencionados:

a. Algoritmo BSF

Además de implementar una función para ejecutar el algoritmo BFS, se crearon otras funciones para facilitar ciertas tareas:

- `find_start_and_goal(maze)` → se creo esta función para poder encontrar de manera dinámica las coordenadas del inicio y final (objetivo) del laberinto, haciendo la solución más flexible. Esta función se ejecuta antes de realizar la búsqueda BFS, ya que tanto las coordenadas de la entrada 'E' como de la salida 'S' forman parte de los parámetros que se requiere ingresar en el algoritmo.
- `visualize_maze_graph(maze_loader)` → Esta función crea un grafo basado en el laberinto cargado y lo visualiza utilizando la librería networkx. Se agregan nodos y aristas representando las posiciones y conexiones en el laberinto. Finalmente, se dibuja el grafo donde los nodos están ubicados en una cuadrícula, mostrándose así una representación gráfica del laberinto.

Implementación de BFS

- `bfs(graph, start, goal)` → Mediante la creación de esta función se implementó el algoritmo de búsqueda en amplitud (Breadth-First Search), con el fin de encontrar un camino desde el nodo de inicio (entrada) hasta el nodo final (salida) en el grafo que representa el laberinto. El proceso sigue estos pasos:

1. **Inicialización:** Se inicializan las estructuras `queue` (cola), `visited` (diccionario), `total_visited` y `backtracking_count` (contadores).
2. **Recorrido en anchura (BFS):** La función entra en un bucle que se ejecuta mientras haya nodos en la cola (mientras haya nodos por explorar). En cada iteración, se extrae el nodo actual de la cola. Este nodo es el siguiente en ser procesado. El contador `total_visited` se incrementa con cada nodo extraído de la cola.
3. **Verificación de meta:** Si el nodo actual es igual al nodo objetivo (salida del laberinto), se ha encontrado un camino de solución. En este punto, se procede a reconstruir el camino.
4. **Reconstrucción del camino:** Una vez que se encuentra la meta, se reconstruye el camino hacia atrás usando los nodos predecesores registrados, hasta llegar al punto de inicio. Finalmente, el camino se invierte con el método `reverse` y la función retorna el camino completo `path`, junto con el contador de retroceso `backtracking_count` y el contador de nodos visitados `total_visited`.

5. **Exploración de vecinos:** En caso de que no se ha alcanzado el objetivo, la función procede a explorar los nodos vecinos del nodo actual. Para cada vecino:
   (a) Si el vecino no ha sido visitado antes, se agrega al diccionario `visited` con el nodo actual como su predecesor y se añade a la cola para ser explorado posteriormente.
   (b) Si el vecino ya ha sido visitado, significa que se ha encontrado un camino hacia ese nodo por segunda vez. En este caso, se incrementa el contador `backtracking_count`.
6. **Fin de la búsqueda:** Si se exploran todos los nodos posibles sin encontrar la meta, la función terminará el bucle y devolverá el camino (vacío o incompleto), el contador de retroceso y el número total de nodos visitados.

Este proceso sigue la metodología típica de la búsqueda en amplitud, adaptada para incluir contadores adicionales que proporcionan información útil sobre el rendimiento de la búsqueda y pueden ser usados como métricas de comparación con otros algoritmos implementados para la resolución del laberinto.

Laberinto 1

RESULTADOS

A continuación se muestra la solución obtenida para el primer caso de estudio:

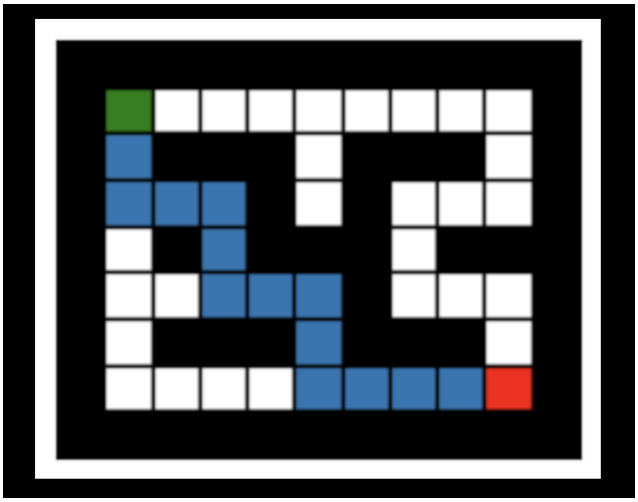


Figure 1: GRAFICO DE LA SOLUCION ENCONTRADA CON BFS PARA EL LABERINTO 1

Se presenta las métricas resultantes al utilizar BFS en este laberinto:

- **Número de retrocesos (backtracking):** 41
- **Número total de nodos visitados:** 39
- **Número de nodos en la solución:** 15

Laberinto 2

RESULTADOS

A continuación se muestra un gráfico del camino encontrado como solución para el caso de estudio 2:

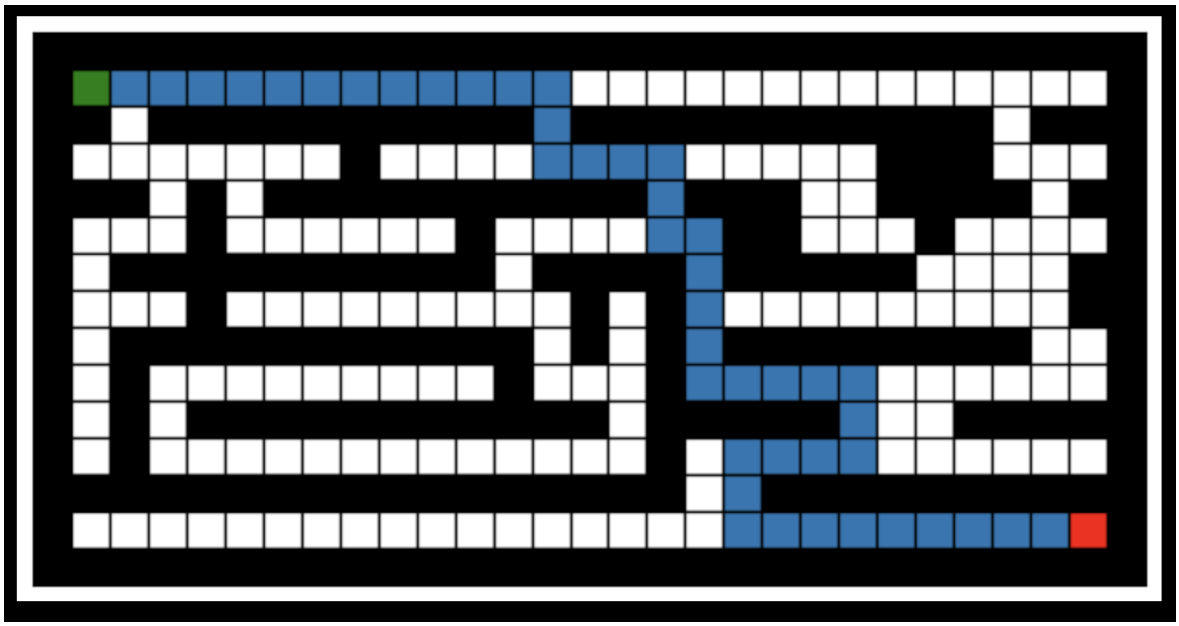


Figure 2: GRAFICO DE LA SOLUCION ENCONTRADA CON BFS PARA EL LABERINTO 2

Las métricas obtenidas para este caso son las siguientes:

- **Número de retrocesos (backtracking):** 213
- **Número total de nodos visitados:** 183
- **Número de nodos en la solución:** 45

Laberinto 3

RESULTADOS

El camino resultante para el tercer laberinto es el siguiente:

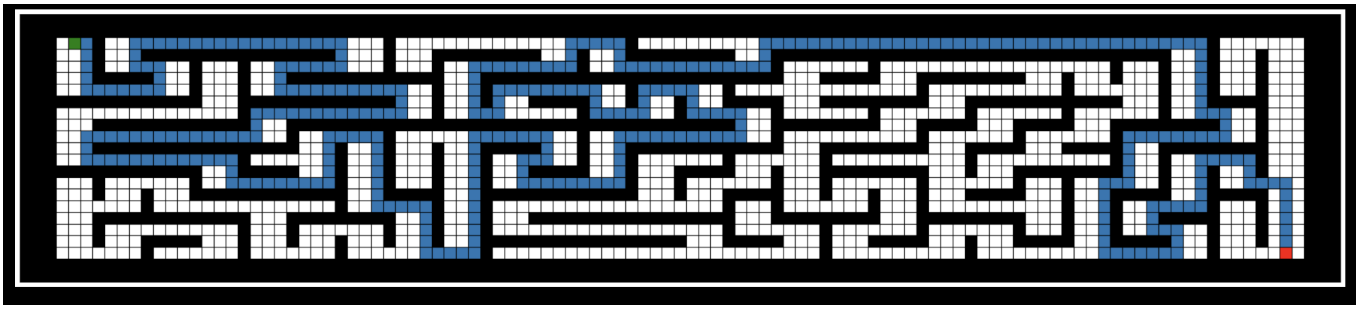


Figure 3: GRAFICO DE LA SOLUCION ENCONTRADA CON BFS PARA EL LABERINTO 3

Para este caso las métricas obtenidas son:

- **Número de retrocesos (backtracking):** 1481
- **Número total de nodos visitados:** 858
- **Número de nodos en la solución:** 345

Para mayor detalle de la resolución de los laberintos con el método BFS consultar el link:
`https://github.com/Borreguin/WorkShopUSFQ/tree/Grupo3/Taller2/P1/BFS`

b. Algoritmo DFS

Como segunda opción, se utilizó un enfoque de búsqueda en profundidad (Depth-First Search, DFS) para resolver los tres laberintos. El algoritmo DFS es un método de exploración de grafos que se adentra en las ramas del laberinto (grafo) tan profundamente como sea posible antes de retroceder cuando ya no haya más nodos por visitar. Este enfoque garantiza que se encuentre una solución, aunque no siempre sea la más corta. Sin embargo, el método DFS es eficiente en términos de memoria, ya que solo necesita rastrear el camino actual.

Se implementó una versión iterativa del DFS para asegurar que la exploración del laberinto pudiera manejarse sin recurrir al sistema de pila del lenguaje. Se resolvieron tres laberintos de distintas complejidades y se calcularon varias métricas para cada uno, incluyendo el número de nodos visitados, la profundidad de la solución y el backtracking.

Laberinto 1

RESULTADOS:

Gráfico obtenido: El DFS encuentra un camino válido desde el punto de inicio hasta el final.

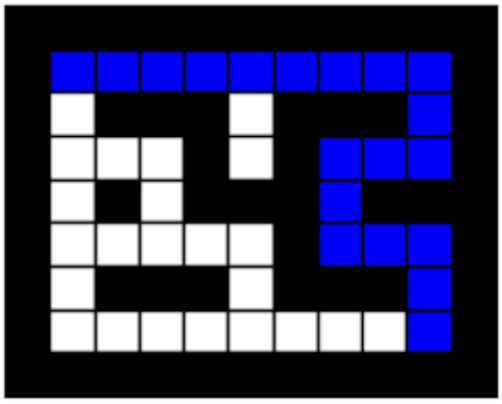


Figure 4: GRAFICO DE LA SOLUCION ENCONTRADA CON DFS PARA EL LABERINTO 1

Métricas:

- **Nodos visitados:** 19
- **Profundidad de la solución:** 18
- **Back Tracking realizado:** 0

Laberinto 2

RESULTADOS:

Gráfico obtenido: El segundo laberinto es más complejo que el primero, con más caminos posibles y zonas de mayor ramificación. El DFS aún logra encontrar una solución, aunque explora varios caminos antes de encontrar el correcto.

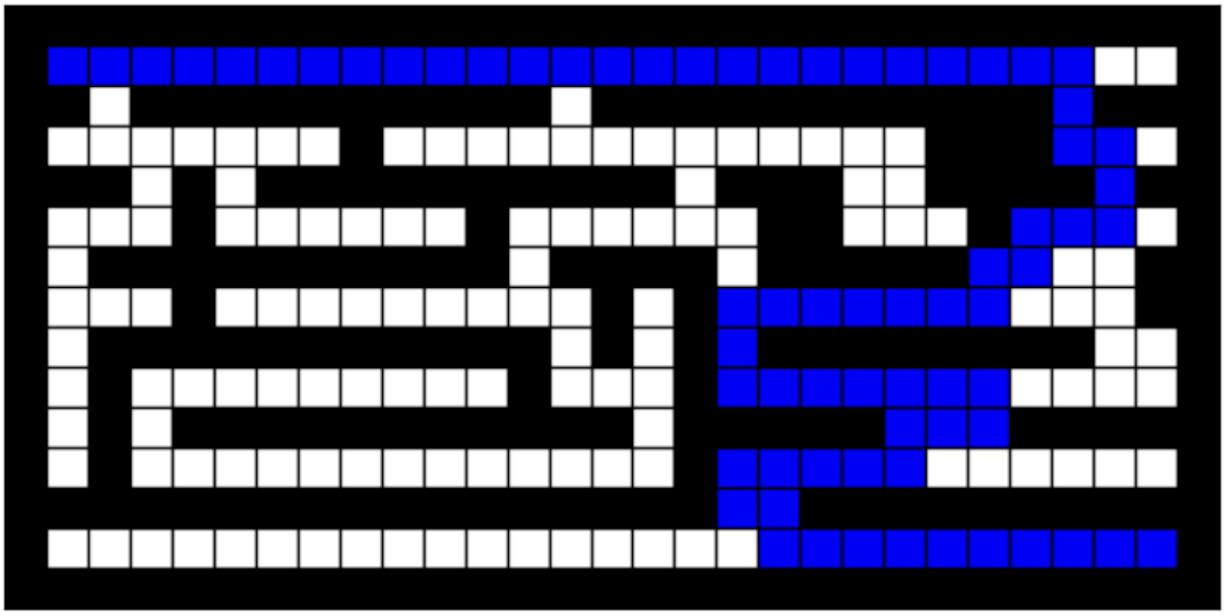


Figure 5: GRAFICO DE LA SOLUCION ENCONTRADA CON DFS PARA EL LABERINTO 2

Métricas:

- **Nodos visitados:** 167
- **Profundidad de la solución:** 78
- **Back Tracking realizado:** 4

Laberinto 3

RESULTADOS:

Gráfico obtenido: El tercer laberinto es el más complejo de todos, presentando una mayor cantidad de intersecciones y posibles bifurcaciones. El DFS recorre un número considerable de nodos, pero eventualmente encuentra un camino válido hacia la solución.

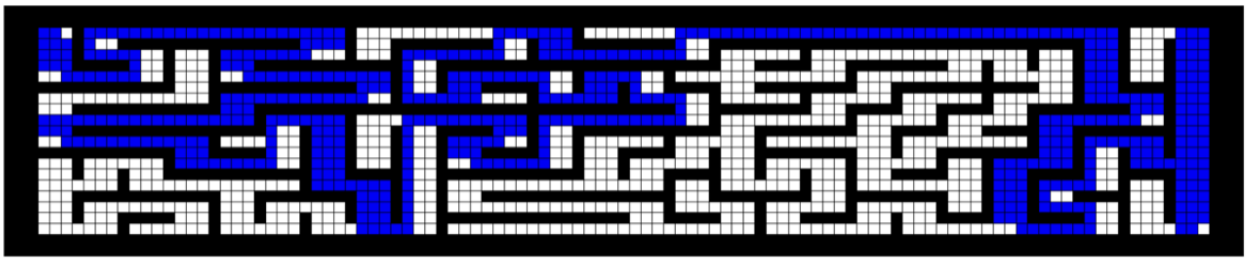


Figure 6: GRAFICO DE LA SOLUCION ENCONTRADA CON DFS PARA EL LABERINTO 3

Métricas:

- **Nodos visitados:** 1168

- **Profundidad de la solución:** 595
- **Back Tracking realizado:** 222

Para mayor detalle de la resolución de los laberintos con el método DFS consultar el link:
`https://github.com/Borreguin/WorkShopUSFQ/tree/Grupo3/Taller2/P1/DFS`

C. Algoritmo A*

El método A* (A-star) es un algoritmo de búsqueda heurística utilizado para encontrar la ruta más corta en un grafo o laberinto. Se basa en la combinación de dos factores: el costo real desde el punto de partida hasta un nodo determinado (generalmente llamado "g") y una estimación heurística del costo desde ese nodo hasta el destino (llamada "h"). El valor total para un nodo se expresa como f(x) = g(x)+h(x), donde "f" es la función que se minimiza. Este enfoque garantiza que A* explore rutas más prometedoras primero, lo que lo convierte en un algoritmo eficiente para la búsqueda de caminos óptimos en muchos tipos de laberintos y gráficos.

La elección de la función heurística "h" es clave para el rendimiento del algoritmo A*. Cuando la heurística es admisible, es decir, nunca sobreestima el costo restante para alcanzar el objetivo, A* es óptimo, encontrando siempre la solución más corta si existe. Un ejemplo común de heurística es la distancia euclidiana o de Manhattan, dependiendo de la estructura del entorno. Además, A* es un algoritmo de búsqueda en grafos que expande nodos de manera inteligente, evitando caminos que son claramente subóptimos, lo que lo diferencia de otras técnicas como la búsqueda de Dijkstra, que no utiliza heurísticas para guiarse.

- Paso 1: Identificación de punto de partida, punto de llegada y la heurística a ser utilizada.
- Paso 2: Se selecciona el nodo de la lista abierta con el menor valor de f(x). Este nodo es el más prometedor en términos de costo total estimado para llegar al objetivo. Se mueve el nodo a la lista cerrada. Luego se evalúan todos los vecinos accesibles de este nodo.
- Paso 3: Repetición del paso 2 hasta llegar al objetivo o la identificación de no solución.

Laberinto 1

RESULTADOS:

Gráfico obtenido: En este laberinto, el algoritmo A* encuentra una solución eficiente, identificando el camino más corto hasta la salida, explorando primero los nodos más prometedores basados en la heurística.

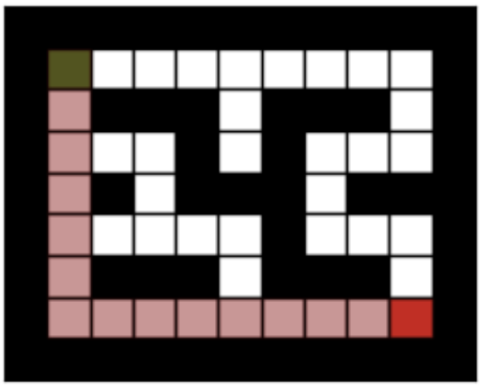


Figure 7: GRAFICO DE LA SOLUCION ENCONTRADA CON A* PARA EL LABERINTO 1

- **Número de retrocesos (backtracking):** 16
- **Número total de nodos visitados:** 34
- **Número de nodos en la solución:** 15

Laberinto 2

RESULTADOS:

Gráfico obtenido: El segundo laberinto presenta más bifurcaciones y rutas alternativas. A pesar de su complejidad, el algoritmo A* sigue encontrando el camino óptimo, visitando menos nodos innecesarios que el DFS.

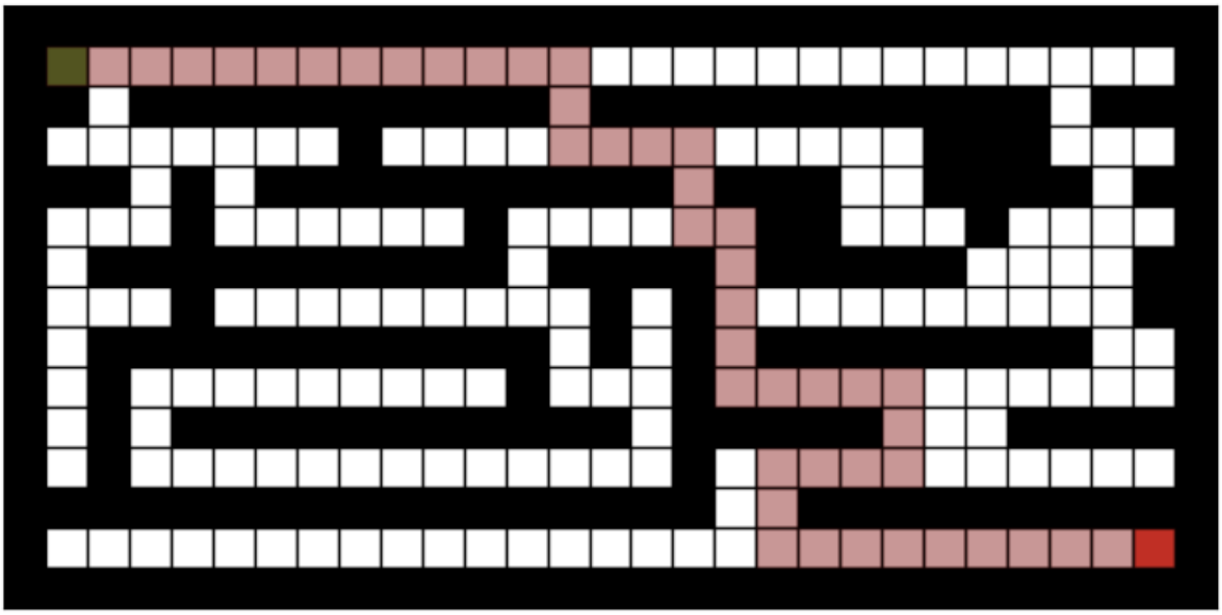


Figure 8: GRAFICO DE LA SOLUCION ENCONTRADA CON A* PARA EL LABERINTO 2

- **Número de retrocesos (backtracking):** 85
- **Número total de nodos visitados:** 139
- **Número de nodos en la solución:** 45

Laberinto 3

RESULTADOS:

Gráfico obtenido: El tercer laberinto, siendo el más complejo de todos, implica la visita a muchos nodos antes de llegar a la solución. Sin embargo, A* aún logra optimizar el proceso comparado con otros enfoques.

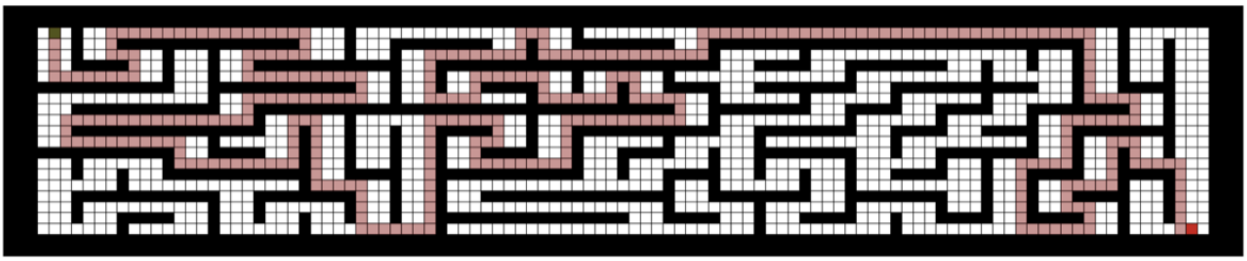


Figure 9: GRAFICO DE LA SOLUCION ENCONTRADA CON A* PARA EL LABERINTO 3

- **Número de retrocesos (backtracking):** 478
- **Número total de nodos visitados:** 787
- **Número de nodos en la solución:** 345

Para mayor detalle de la resolución de los laberintos con el método A* consultar el link:
`https://github.com/Borreguin/WorkShopUSFQ/tree/Grupo3/Taller2/P1/A_STAR`

Comparación de resultados

Se implementaron los algoritmos de búsqueda en amplitud (BFS), búsqueda en profundidad (DFS) y A* para resolver los laberintos. Se compararon los 3 algoritmos en términos de: cantidad de nodos explorados, backtracking y longitud del camino para cada laberinto. A continuación, se discuten los resultados:

En términos generales, se puede identificar que tanto los métodos BFS como A*, obtuvieron el recorrido más corto, en comparación con el método DFS que en todos los casos fue mayor, esto concuerda con la teoría en la que se conoce que el método DFS encuentra una solución que no puede ser la mejor.

- Para el primer laberinto, el método DFS, visitó la menor cantidad de nodos y realizó menor número de backtracking en comparación con A* y BFS quienes visitaron mayor número de nodos (A*: 34 y BFS: 39) y realizaron más backtracking (A*: 16 y BFS: 41).
- Para el laberinto 2, el algoritmo que menos nodos visitó fue A* con 139, seguido por DFS con 167 y por último BFS con 183 nodos. En lo que respecta a backtracking, el algoritmo que menos lo realizó es DFS con 4 veces, seguido por A* con 85 y por último BFS con 213.
- Para el laberinto 3, el algoritmo que menos nodos visitó fue A* donde se visitaron 787 nodos, seguido por BFS con 858 nodos visitados y por último DFS con 1168. Con relación a backtracking, el algoritmo que más tuvo que realizarlo es BFS con 1481, seguido por A* con 478, y finalmente el algoritmo DFS con el menor número de retrocesos realizados, presentando un backtracking de 222.

En las siguientes figuras se resume el rendimiento de cada algoritmo para cada una de las métricas evaluadas:

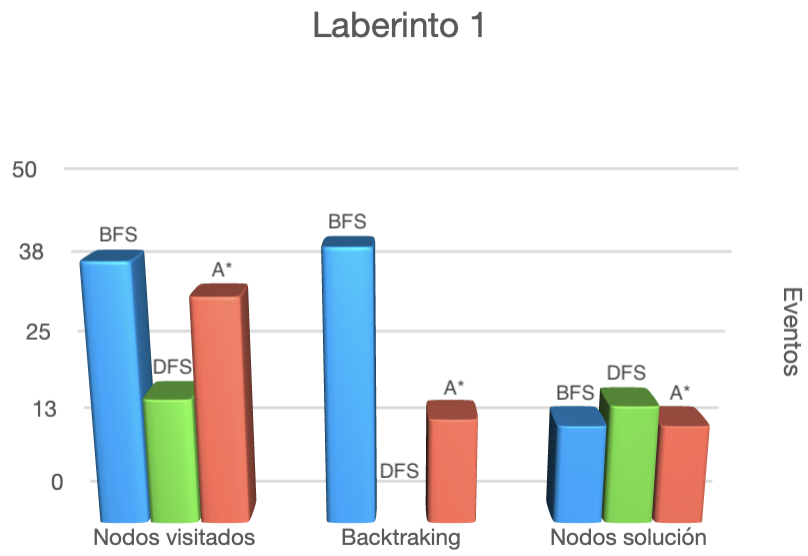


Figure 10: GRAFICO DE LAS METRICAS PARA EL LABERINTO 1

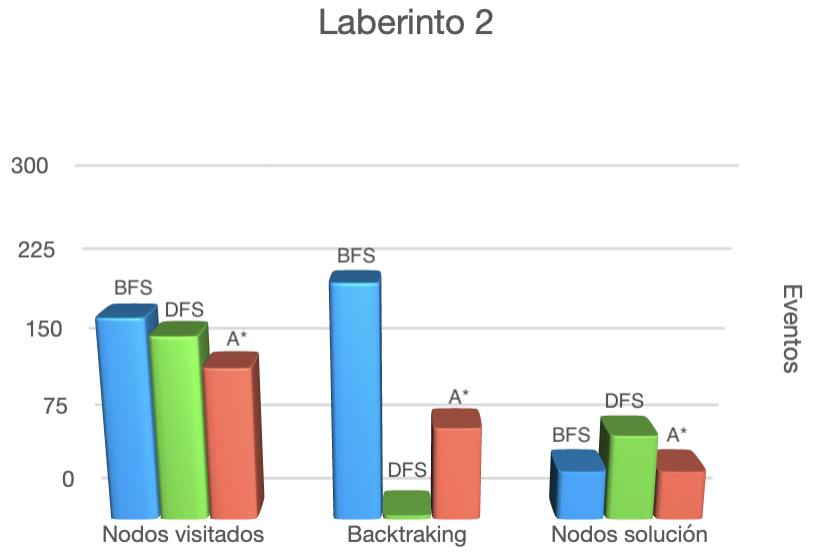


Figure 11: GRAFICO DE LAS METRICAS PARA EL LABERINTO 2

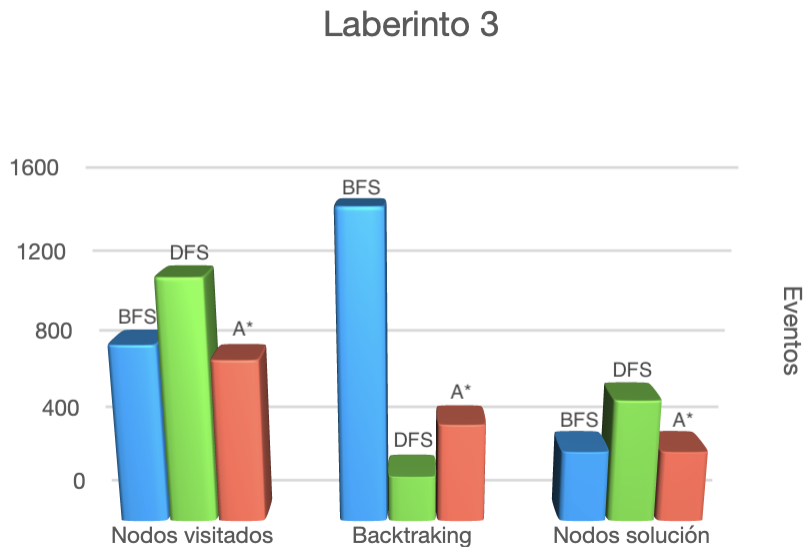


Figure 12: GRAFICO DE LAS METRICAS PARA EL LABERINTO 3

2. OPTIMIZACIÓN DE COLONIAS DE HORMIGAS

Ant Colony Optimization (ACO) es una técnica de optimización inspirada en el comportamiento de las hormigas reales cuando buscan recursos para su colonia. El propósito de este algoritmo en el campo de la IA es el de simular el comportamiento de las hormigas para encontrar el mejor camino desde el nido de la colonia a la fuente de recursos.

A. Correr la implementación planteada

En el repositorio, en la carpeta Taller2/P2/P2_ACO.py se plantea un ejemplo de este algoritmo, ejecutar el caso de estudio 1. Analizar el código.

El script implementa un algoritmo de optimización por colonia de hormigas (ACO) para encontrar el mejor camino en una cuadrícula desde un punto de inicio hasta un punto final, evitando obstáculos. El script incluye las siguientes partes principales:

- Inicialización del algoritmo.
- Obtención de vecinos válidos.
- Selección de siguiente posición.
- Evaporación de feromonas.
- Depósito de feromonas.
- Búsqueda del mejor camino.
- Visualización del laberinto y el mejor camino.
- Uso de clase implementada.

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

Se plantea el caso de estudio 2, sin embargo, algo está mal en la selección del camino, ¿puedes arreglarlo? Pistas:

1. **Al escoger el mejor camino una condición está faltando, ¿es suficiente elegir el camino con el menor tamaño?**
2. **Cambiar el número de hormigas, cambiar los parámetros: taza de evaporación, Alpha, Beta.**

En el caso de estudio 2, el problema radica en la selección incorrecta del mejor camino. Actualmente, el código selecciona el camino más corto, pero esto no siempre garantiza que sea el mejor camino en términos de calidad o cantidad de feromonas depositadas. Para mejorar esto, modificaremos el criterio de selección, considerando tanto la longitud del camino como la cantidad de feromonas acumuladas para elegir el mejor camino.

Al correr el código con esta modificación, no se encontró un camino óptimo que llegue a la meta. Dado este escenario, se pudo concluir que otro de los problemas detectados para el caso de estudio 2 fue la elección parámetros del modelo como los son el número de hormigas, el valor de evaporación de feromonas (tasa de evaporación), y los coeficientes alpha y beta. Un ajuste de dichos parámetros nos ayudará a conseguir una solución más acertada para el problema.

Solución implementada:

1. **Inclusión de criterios adicionales al elegir el mejor camino**

   No es suficiente elegir la ruta con el menor tamaño. Además de tomar en cuenta la longitud del camino, también se consideró importante la intensidad de las feromonas depositadas en las celdas. Por este motivo, se realizó una mejora del método `find_best_path`, incluyendo una condición adicional para la selección del mejor camino.

2. **Ajuste de parámetros**

   Cambiar el número de hormigas y ajustar los parámetros de la tasa de evaporación, alpha (influencia de las feromonas) y beta (influencia del valor heurístico) puede mejorar la exploración, explotación y convergencia del algoritmo. Es así, que se creo una función `probar_parametros_y_visualizar()`, la cual nos permite realizar una exploración de diferentes combinaciones de parámetros dentro del algoritmo de optimización basado en colonias de hormigas (ACO) y visualizar los resultados obtenidos. De manera resumida, mediante su

ejecución se lleva a cabo un experimento exhaustivo para evaluar cómo diferentes combinaciones de parámetros afectan el rendimiento del algoritmo, prueba variaciones en los pesos de la feromona, la heurística y la tasa de evaporación, y luego grafica los resultados en mapas de calor que muestran la longitud de los caminos encontrados. Además, identifica y reporta la combinación de parámetros óptima que produce el camino más corto hacia el destino.

Su proceso de implementación consta de los siguinetes pasos:

- **Definición de parámetros** → Se definen tres conjuntos de parámetros que controlan el comportamiento del algoritmo ACO: `alphas` (peso de la feromona), `betas` (peso de la heurística) y `evaporation_rates` (tasa de evaporación de la feromona). Además, se fija el número de hormigas en `num_ants` = 25.
- **Caso de estudio** → Se definen las condiciones dadas para el caso de estudio 2 con un punto de inicio `start` = (0, 0), un punto de destino `end` = (4, 7) y una serie de obstáculos en el mapa.
- **Inicialización de matrices** → Se crean cuatro matrices 3D: una llamada `results` para almacenar los resultados de todos los caminos encontrados, otra llamada `best_results` para almacenar únicamente las resultados de los caminos que llegaron efectivamente al destino, otra llamada `distances` para almacenar las longitudes de todos los caminos encontrados y una última llamada `best_distances` que almacena solo las longitudes de los caminos que cumplieron el propósito de hallar la meta.
- **Iteración sobre combinaciones de parámetros** → La función recorre todas las combinaciones posibles de los valores de `alpha`, `beta` y `evaporation_rate`. Para cada combinación:
  (a) Se crea una instancia del algoritmo ACO con esos parámetros.
  (b) Se ejecuta el método `find_best_path()` del ACO, el cual intenta encontrar el mejor camino en 100 iteraciones.
  (c) La longitud del camino encontrado se almacena en la matriz `distances`. Si el camino llega al destino, también se almacena en `best_distances`, de lo contrario, se asigna un valor infinito (`np.inf`). Para el cálculo de distancia de los caminos se utiliza la función `euclidean_distance(path)`.
- **Visualización de resultados** → Se crean dos mapas de calor (heatmaps) utilizando la librería `seaborn`:
  (a) El primer heatmap muestra las longitudes de todos los caminos encontrados para cada combinación de parámetros.
  (b) El segundo heatmap solo muestra las longitudes de los caminos que llegaron al destino, es decir, aquellos caminos que fueron soluciones exitosas.
- **Selección de los mejores parámetros** → Después de probar todas las combinaciones, se identifica la combinación de parámetros (`alpha`, `beta`, `evaporation_rate`) que produjo el camino más corto hacia el destino. Finalmente, se imprimen los valores de los mejores parámetros y la longitud del mejor camino.

A continuación se muestra el mapa de calor obtenido, en el cual se observa todas las longitudes de los diferentes caminos hallados para todas las combinaciones posibles de parámetros. Para cada uno de los 3 parámetros considerados (alpha, beta y tasa de evaporación) se tomaron 5 valores posibles, por lo cual se obtienen 125 combinaciones totales.

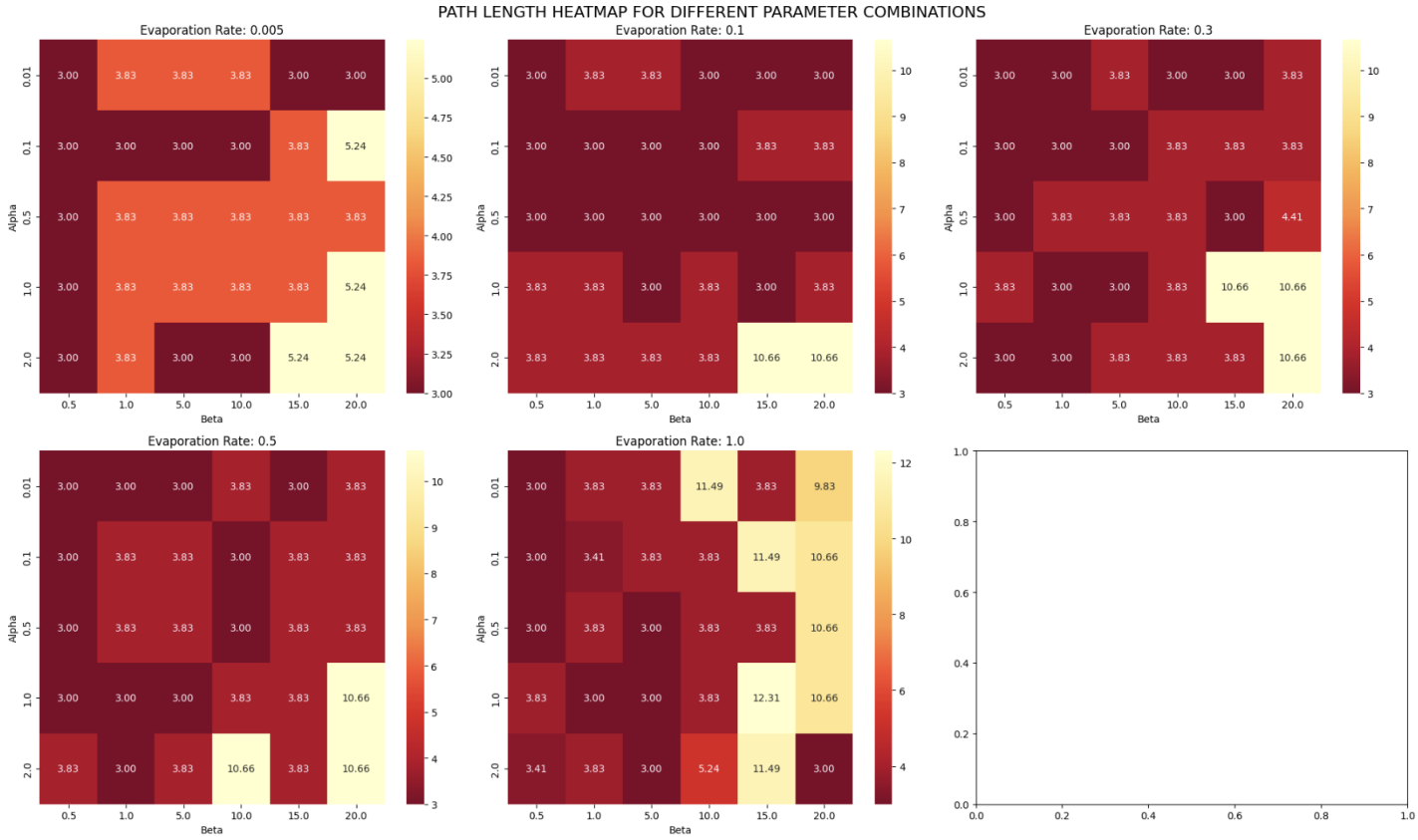
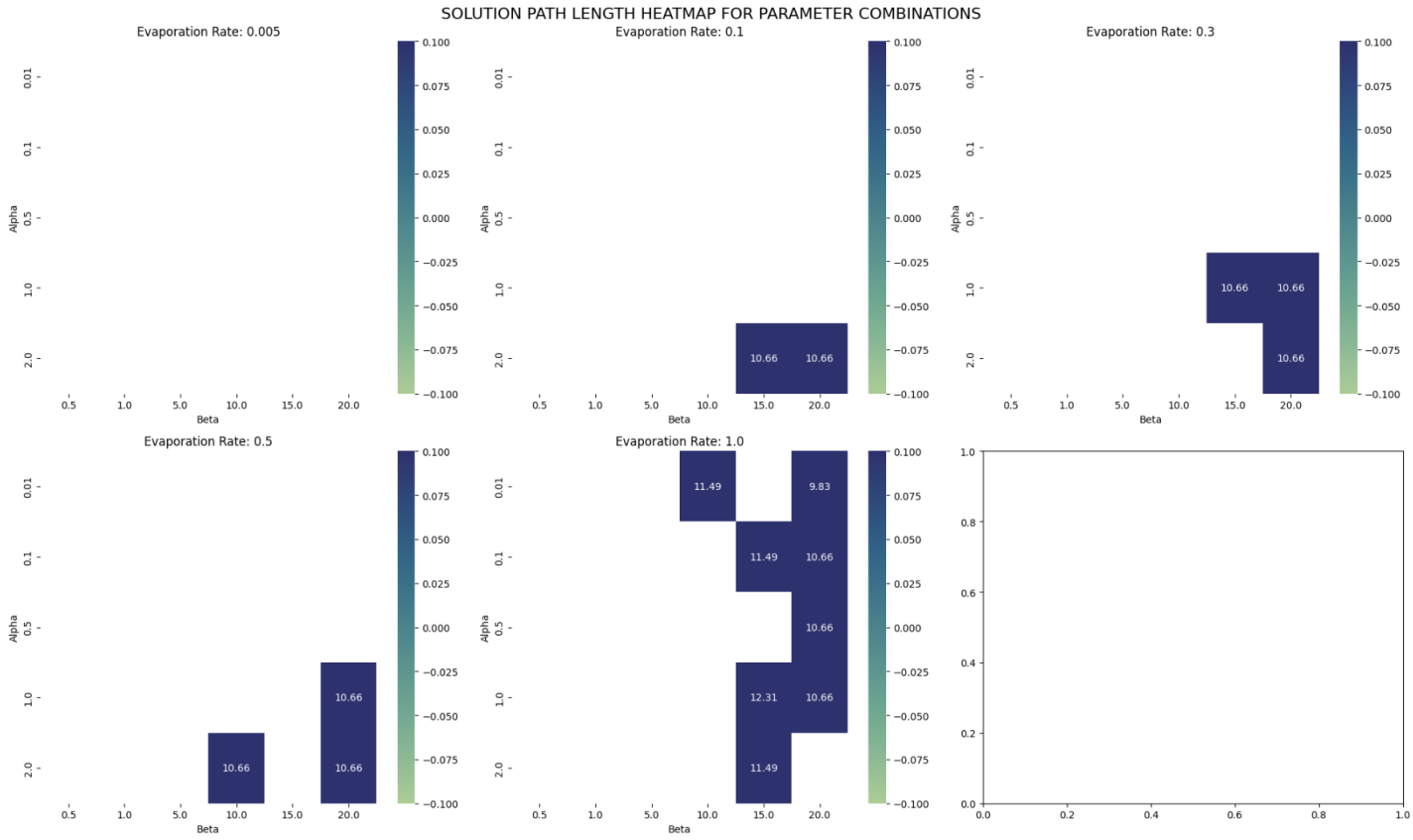


Figure 13: MAPA DE CALOR DE LONGITUD DE CAMINO PARA COMBINACIONES DE PARAMETROS

Como se observa en el gráfico anterior, tenemos caminos con longitudes pequeñas como por ejemplo 3 unidades, pero eso no nos indica que son los mejores caminos, de hecho ni siquiera son realmente soluciones al problema, ya que los caminos trazados no llegan al punto objetivo. Justamente por este motivo, se crea un filtro para detectar únicamente las rutas en las que se alcanzó el destino propuesto y se visualizan en un heatmap el cual se muestra a continuación:




```
Best parameters: ALPHA = 0.01, BETA = 20.0, EVAPORATION RATE = 1.0
BEST PATH LENGTH: 9.82842712474619
```

Figure 14: MAPA DE CALOR DE LONGITUD DE CAMINO DE SOLUCION EXITOSA

Una vez realizado el gráfico anterior, podemos identificar visualmente cual de todas las 16 combinaciones existosas de parámetros llega al destino recorriendo la menor distancia. La combinación óptima que se aprecia en el mapa de calor es: ALPHA = 0.01, BETA = 20.0, EVAPORATION RATE = 1.0 .

La mejor combinación de parámetros dada por el código fue:

ALPHA → 0.01

BETA → 20.0

EVAPORATION RATE → 1.0

La solución del camino más corto se muestra en la siguiente figura:

```
Probando con ALPHA = 0.01, BETA = 20.0, EVAPORATION RATE = 1.0
Camino encontrado: [(0, 0), (1, 1), (2, 1), (3, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7)]
Distancia euclidiana: 9.82842712474619
```

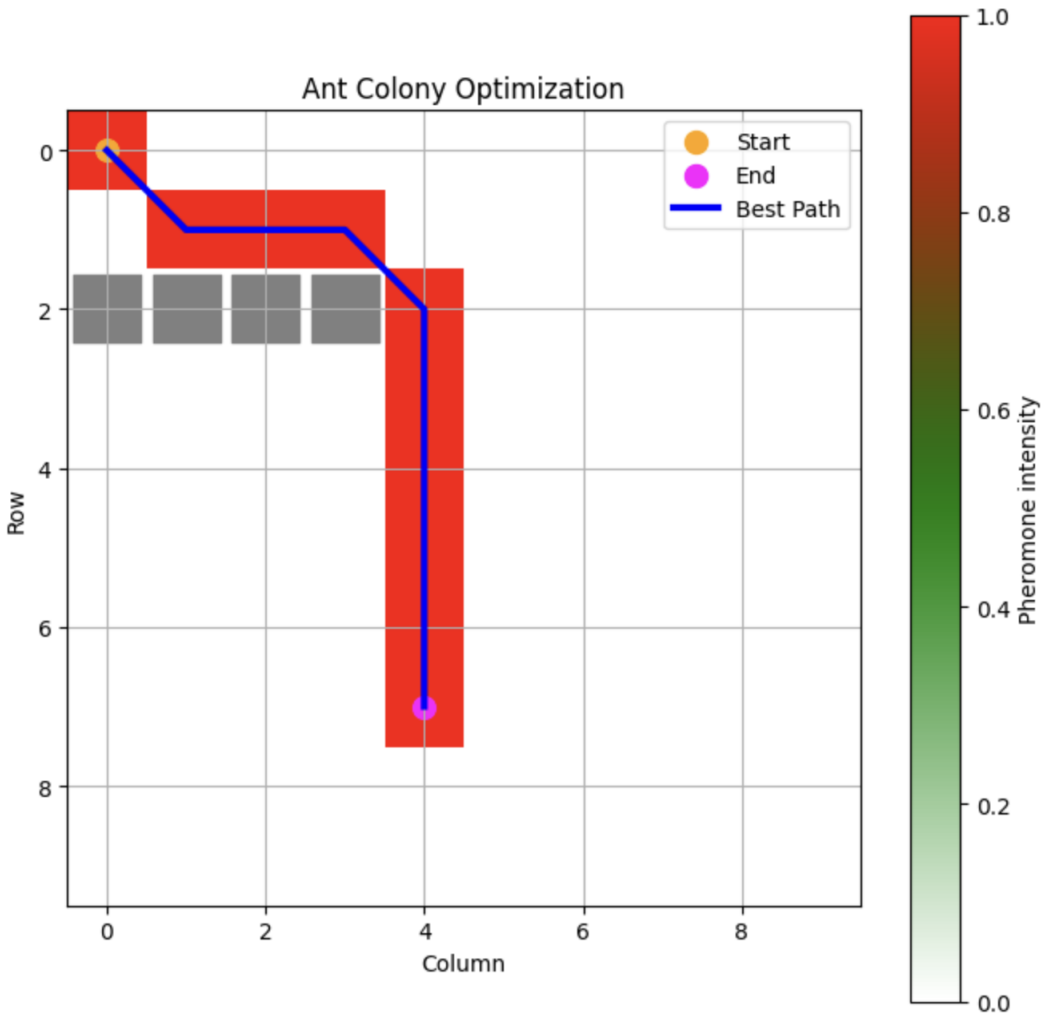


Figure 15: SOLUCION ACO PARA ALPHA = 0.01, BETA = 20.0, EVAPORATION RATE = 1.0

Para mayor detalle del código utilizado para las combinaciones de parámetros en ACO y sus visualizaciones consultar el link:

`https://github.com/Borreguin/WorkShopUSFQ/blob/Grupo3/Taller2/P2/P2_ACO.ipynb`

C. Describir los parámetros del modelo

¿Qué propósito tiene cada parámetro en el modelo?

A continuación se detalla el propósito que cumple cada parámetro del modelo:

- **start y end** → Estos parámetros definen las posiciones inicial y final de las hormigas en la cuadrícula. Cumplen el propósito de definir el espacio del problema.
- **obstacles** → Es una lista de coordenadas de la cuadrícula donde las hormigas no pueden moverse. Su objetivo es indicar las posiciones en la cuadrícula que están bloqueadas o son intransitables para las hormigas. Añaden complejidad a la búsqueda de un camino óptimo.
- **grid_size** → Tamaño de la cuadrícula en la que se mueven las hormigas.

- **num_ants** → Número de hormigas que se utilizan en cada iteración del algoritmo. Más hormigas pueden proporcionar una mejor exploración del espacio de búsqueda, pero a costa de un mayor esfuerzo computacional. Este parámetro es clave para equilibrar la compensación entre exploración y eficiencia.
- **evaporation_rate** → Tasa a la que las feromonas se evaporan en cada iteración. Es crucial para evitar que el algoritmo converja demasiado rápido en caminos subóptimos. Al controlar la velocidad a la que las feromonas se evaporan, ayuda a evitar que las soluciones anteriores dominen el proceso de búsqueda y permite fomentar la exploración de nuevas rutas (tasa de evaporación alta).
- **alpha** → Parámetro que controla la influencia de las feromonas en la selección del siguiente movimiento de las hormigas. Un valor más alto de alfa da más importancia a los rastros de feromonas, haciendo que las hormigas sean más propensas a seguir caminos que han sido reforzados por otras hormigas.
- **beta** → Este parámetro controla la influencia de la heurística (es decir, qué tan atractivo es un camino basado en su proximidad al destino). Un valor más alto de beta aumenta la importancia de la heurística, haciendo que las hormigas prioricen rutas más cortas o directas hacia el final.

¿Cómo afectan al modelo los parámetros tasa de evaporación, alfa y beta?

En cuanto a la tasa de evaporación, es importante que ésta disminuya gradualmente los niveles de feromonas presentes, ya que así asegura que los caminos más antiguos y menos óptimos se vuelvan menos atractivos, ayudando al algoritmo a evitar estancarse en soluciones subóptimas.

En el caso de alfa y beta, el equilibrio entre ellas determina la compensación entre exploración y explotación. Por ejemplo un alfa alto ocasionará que las hormigas dependan excesivamente de las feromonas, (lo cual puede derivar en una convergencia prematura), mientras que un beta alto fomenta que las hormigas sigan caminos impulsados por heurísticas, (lo cual puede acelerar la búsqueda de caminos más cortos pero descuidando rutas más óptimas con el tiempo).

D. Pregunta de investigación:

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

El algoritmo ACO es apto para resolver el problema del vendedor viajero (TSP). Los pasos de su implementación serían:

- **Representación del Problema:** Cada ciudad se representa como un nodo en un grafo. Las distancias entre las ciudades se representan como los pesos de las aristas entre los nodos.
- **Inicialización:** Inicializar las feromonas en todas las aristas con un valor pequeño pero positivo y definir los parámetros del modelo (número de hormigas, alfa, beta, tasa de evaporación, etc).
- **Construcción de Soluciones:** Cada hormiga construye una solución (ruta) comenzando desde una ciudad inicial y visitando todas las ciudades exactamente una vez. La selección de la siguiente ciudad se basa en la cantidad de feromonas y la distancia (heurística).
- **Actualización de Feromonas:** Después de que todas las hormigas han construido sus rutas, actualizar las feromonas en las aristas. Evaporar una fracción de las feromonas en todas las aristas. Depositar nuevas feromonas en las aristas utilizadas en las rutas construidas por las hormigas de manera proporcional a la calidad de las rutas (inversamente proporcional a la longitud de la ruta).
- **Iteración:** Repetir el proceso de construcción de soluciones y actualización de feromonas durante un número determinado de iteraciones o hasta que se cumpla un criterio de convergencia.
- **Mejor Solución:** Mantener un registro de la mejor solución encontrada durante todas las iteraciones.

Implementar estos pasos en el contexto del TSP nos permite utilizar el algoritmo ACO para encontrar rutas eficientes, minimizando la distancia total recorrida.

Evaluación de Cumplimiento de Objetivos:

Problema 1 (uso de algoritmos de búsqueda en laberintos)

La implementación cumplió con las expectativas, logrando una solución precisa y eficiente que encuentra el camino óptimo para salir de los laberintos propuestos. Se logró una visualización clara del recorrido en cada uno de los laberintos, lo que facilitó el análisis de los resultados obtenidos. El problema fue resuelto con éxito en los tres laberintos, y se obtuvo una aproximación bastante buena del camino más corto, mostrando la capacidad de estos algoritmos para adaptarse a distintas configuraciones de laberinto.

Problema 2 (ACO)

La implementación del algoritmo ACO también cumplió con los objetivos establecidos. En el primer caso de estudio, las hormigas fueron capaces de encontrar una solución cercana al óptimo con el código inicial proporcionado. Para el segundo caso de estudio, la modificación en los criterios de selección de caminos y el ajuste de parámetros clave como la tasa de evaporación de feromonas, alpha y beta, permitió corregir los errores iniciales y mejorar la eficiencia de la solución. Adicionalmente, se incluyó un mapa de calor como visualización de los mejores caminos encontrados al combinar los parámetros clave propuestos. Además, se logró crear una conexión teórica entre este algoritmo (ACO) y el problema del viajante (TSP), describiendo los pasos de su implementación para resolverlo.

De manera general se concluye con una evaluación positiva del cumplimiento de objetivos, habiendo dado solución a todos los problemas planteados en el taller.

Aplicabilidad

A lo largo del taller, los algoritmos de búsqueda y ACO mostraron su utilidad en escenarios que requieren tanto una búsqueda exhaustiva como soluciones óptimas aproximadas en problemas de optimización. Estos enfoques tienen la capacidad de adaptarse para distintos tipos de problemas como por ejemplo al de TSP (planificación de rutas). Otro tipo de aplicaciones puede ir desde la navegación autónoma hasta problemas de logística y sistemas inteligentes.

Conclusiones

Problema 1:

- Existen diferentes algoritmos que permiten encontrar una solución óptima a los laberintos planteados, sin embargo no todos garantizan llegar siempre a la solución óptima (camino más corto). Se debe considerar que para el presente taller se trabajó únicamente con 3 laberintos y se obtuvieron resultados favorables en cuanto a encontrar una ruta de solución para cada escenario y que esta ruta sea la de menor tamaño posible. Pero si se trabajara con más laberintos, posiblemente no siempre se encontraría el camino óptimo con todos los algoritmos o incluso en algún caso podría no hallarse una solución que alcance el nodo objetivo.
- Contar con una buena representación del problema es esencial para poder aplicar algoritmos. En la primera parte del taller, al convertir el laberinto en un grafo, se pudo ver la importancia de la forma en la que se describe un problema, y cómo ésta afecta directamente la eficiencia de los algoritmos aplicados.
- BFS es un algoritmo de búsqueda no informada que explora todas las celdas a un nivel antes de profundizar. Este algoritmo asegura que siempre se encuentre el camino más corto si existe, pero tiende a explorar más nodos para encontrar la solución. Esto se vio reflejado en las métricas como el número de retrocesos (indicador

del número de decisiones incorrectas que se tomó al ejecutar el algoritmo antes de llegar al camino de solución) y el número de nodos explorados.

- DFS: Es más eficiente en ciertos escenarios como por ejemplo el laberinto 1, pero no garantiza encontrar el camino más corto tal como ha sido comprobado en los 3 experimentos. Es más rápido que BFS en algunos casos debido a la menor sobrecarga de memoria.
- A* mostró un mejor balance en relación a la dificultad del laberinto, considerando que en el laberinto más grande y complejo obtuvo las mejores métricas, mientras que para el laberinto 1 fue el segundo mejor. Cabe indicar que al momento de implementar el algoritmo, este debe conocer tanto el origen como el objetivo que debe alcanzar.
- La comparación entre algoritmos permitió entender la importancia de considerar otras métricas más allá de la longitud del camino para poder evaluar su rendimiento, evidenciando sus ventajas y desventajas. Al comparar los resultados de los 3 algoritmos utilizados, se encontró que el algoritmo DFS muestra un mejor desempeño para resolver un laberinto sencillo mientras que para laberintos más complejos la mejor resolución se obtuvo a través del método A*.

Problema 2 :

- El estudio del algoritmo de colonias de hormigas permitió observar cómo el tomar inspiración de la naturaleza puede ser aplicado eficazmente para resolver problemas de optimización complejos.
- Se pudo observar cómo el **criterio de elección de condiciones** para determinar la mejor solución y el **ajuste de los parámetros** de un modelo de optimización, pueden influir directamente en los resultados. El tomar en cuenta esto en el segundo caso del problema 2, nos permitió corregir los errores iniciales y mejorar la eficiencia de la solución. Esto no solo demuestra la flexibilidad del algoritmo, sino también la importancia de un ajuste fino en modelos bioinspirados.
- Además, se pudo establecer un puente teórico entre ACO y el problema del vendedor viajero (TSP), lo que sugiere la posibilidad de aplicar este algoritmo en problemas más complejos de optimización combinatoria.
- Se obtuvieron 16 combinaciones distintas de parámetros que nos daban caminos correctos (completaban el recorrido del caso de estudio 2). Sin embargo, el código implementado para el ajuste de parámetros dio como mejor resultado una única combinación con ALPHA = 0.01, BETA = 20.0, EVAPORATION RATE = 1.0 y NUM_ANTS = 25. Este conjunto de valores entregó el camino más corto (9.83 unidades) para desplazarse desde el punto de inicio hasta el punto establecido como meta. Se puede concluir que en este caso se obtuvo una configuración de los mejores parámetros para resolver el problema, basándonos en la **distancia euclidiana**, pero podría considerarse la inclusión de otro tipo de métricas adicionales como la **distancia de Manhattan**.

Conclusiones generales:

- El taller 2 nos permitió profundizar en algoritmos de búsqueda y optimización, adquiriendo conocimiento de cómo utilizarlos para el desarrollo de soluciones eficientes a problemas complejos.
- Los algoritmos implementados son robustos y pueden ser adaptados para su uso en problemas reales. Este taller ha enriquecido el entendimiento sobre cómo se aplican los conceptos teóricos en situaciones prácticas, fomentando así habilidades analíticas esenciales para la ciencia de datos e inteligencia artificial.

Participación y Colaboración:

Los participantes del grupo participaron activamente en la ejecución de las siguientes actividades:

Integrante	Actividades realizadas
Milene Muñoz	Desarrollo de script (algoritmo BFS) para la resolución del problema 1. Implementación de función para realizar ajuste de parámetros en ACO del problema 2. Análisis de los scripts desarrollados por el resto de integrantes del grupo. Propuestas de mejora. Publicación en Github de scripts realizados. Elaboración de documento.
Edgar Córdova	Desarrollo de script (algoritmo A*) para la resolución del problema 1. Revisión y entendimiento de script proporcionado para la parte 2. Análisis de los scripts desarrollados por el resto de integrantes del grupo Propuestas de mejora. Publicación en Github de scripts realizados. Apoyo en la elaboración del documento.
Jair Criollo	Desarrollo de script (algoritmo DFS) para la resolución del problema 1. Revisión y entendimiento de script proporcionado para la parte 2. Análisis de los scripts desarrollados por el resto de integrantes del grupo Propuestas de mejora. Publicación en Github de scripts realizados. Apoyo en la elaboración del documento.

Link Github:

`https://github.com/Borreguin/WorkShopUSFQ/tree/Grupo3/Taller2`