

Taller N°2

Integrantes

Arias Daniel

Ramírez André

Rodríguez Santiago

P1 - USO DE ALGORITMOS DE BÚSQUEDA.....	2
Laberinto 1.....	2
Resultados:.....	2
Discusión:.....	4
Laberinto 2.....	5
Resultados:.....	5
Discusión:.....	6
Laberinto 3.....	6
Resultados:.....	6
Discusión:.....	7
Conclusión:.....	7
P2 - OPTIMIZACIÓN DE COLONIAS DE HORMIGAS.....	8
¿Qué ocurre con el segundo caso de estudio?.....	8
¿Qué propósito tiene cada parámetro en el modelo?.....	9
¿Será que se puede utilizar este algoritmo para resolver el Travelling Salesman Problema (TSP)? ¿Cuáles serían los pasos de su implementación?.....	10
Referencias.....	11
Colaboración de integrantes.....	11
Link al repositorio.....	11

P1 - USO DE ALGORITMOS DE BÚSQUEDA

Laberinto 1

Resultados:

Laberinto Original

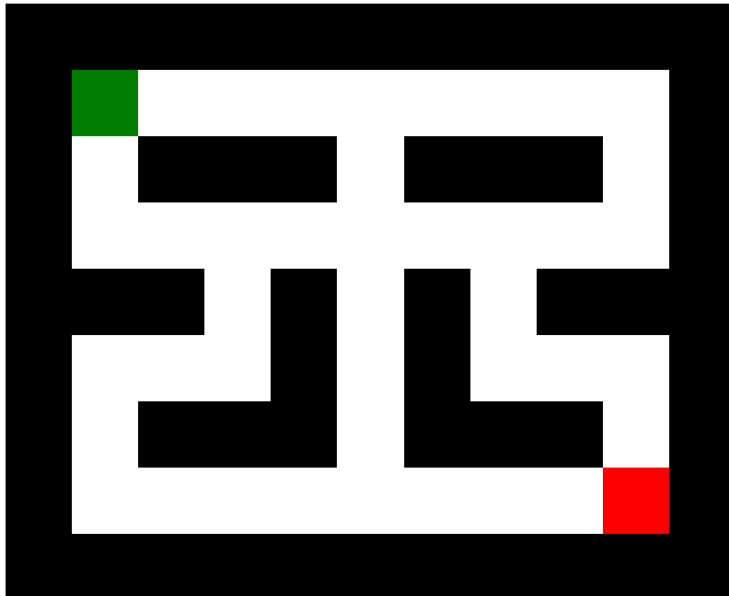


Gráfico 1: visualización de laberinto original

Grafo del Laberinto

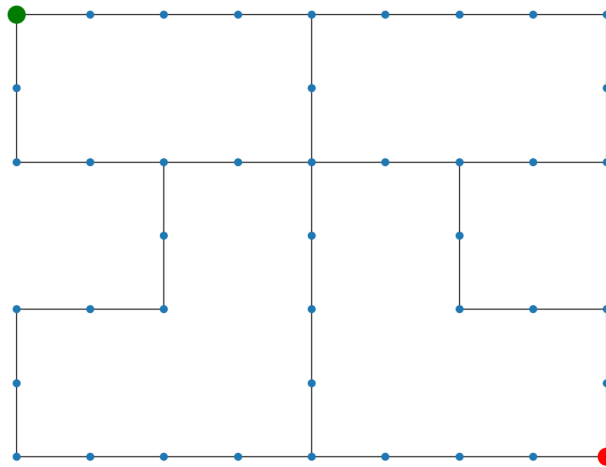


Gráfico 2: grafo de laberinto 1

Solución usando A*

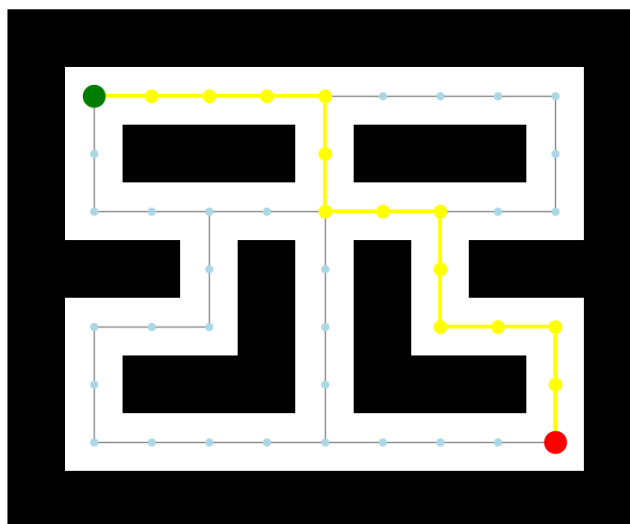


Gráfico 3: visualización de solución utilizando A* en laberinto 1

Solución usando BFS

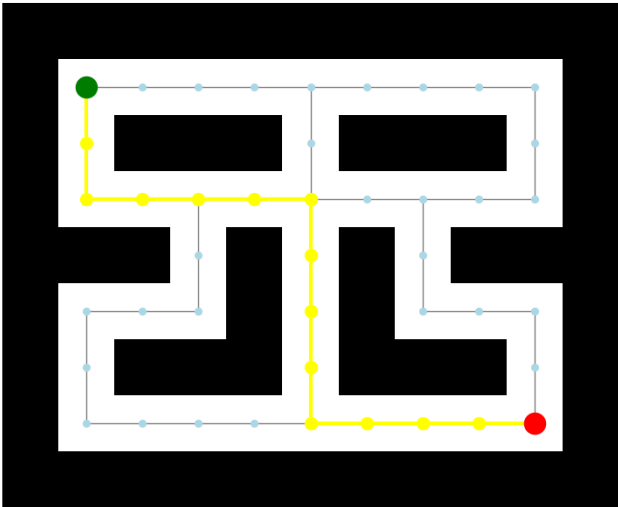


Gráfico 4: Gráfico 3: visualización de solución utilizando BFS en laberinto 1
A continuación se presenta la captura de pantalla sobre el desempeño de los modelos:

Overall Performance Comparison:		
Metric: Path Length		
Maze	A*	BFS
laberinto1.txt	15	15
Metric: Visited Nodes		
Maze	A*	BFS
laberinto1.txt	36	43
Metric: Time		
Maze	A*	BFS
laberinto1.txt	0.000000	0.000000

Gráfico 5: resultados de desempeños de los algoritmos utilizados

Discusión:

Los resultados obtenidos al ejecutar el código permiten comparar de forma clara la eficiencia de los algoritmos A* y BFS en la resolución de laberintos. En general, el algoritmo A* encontró caminos igual de cortos o más eficientes que BFS, pero con una ventaja notable en términos de número de nodos visitados y tiempo de ejecución. Esto se debe a que A* utiliza una heurística (distancia Manhattan) que guía la búsqueda hacia el objetivo, reduciendo la exploración innecesaria. Por otro lado, BFS explora de manera uniforme sin considerar la dirección óptima, lo que puede generar un mayor costo computacional en laberintos más complejos. La visualización gráfica facilitó la comprensión de cómo cada

algoritmo recorre el laberinto, y los datos cuantitativos permitieron confirmar que A* tiende a ser más eficiente en escenarios con un camino definido y obstáculos distribuidos.

Laberinto 2

Resultados:

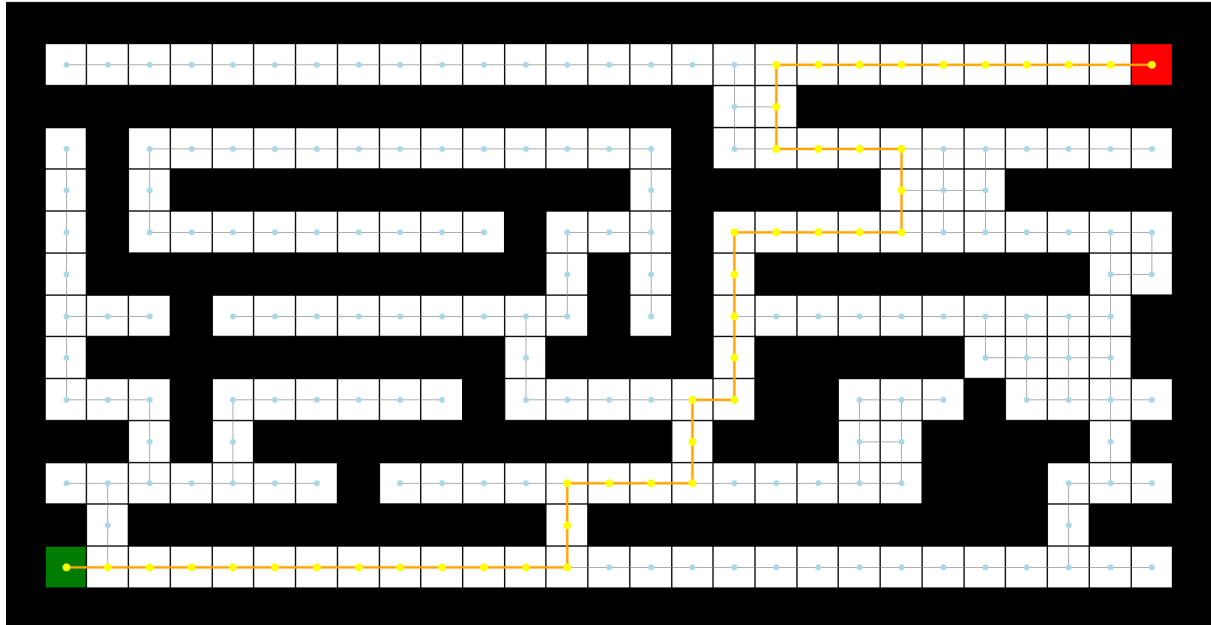


Gráfico 6: Resultados obtenidos mediante la implementación del algoritmo A*

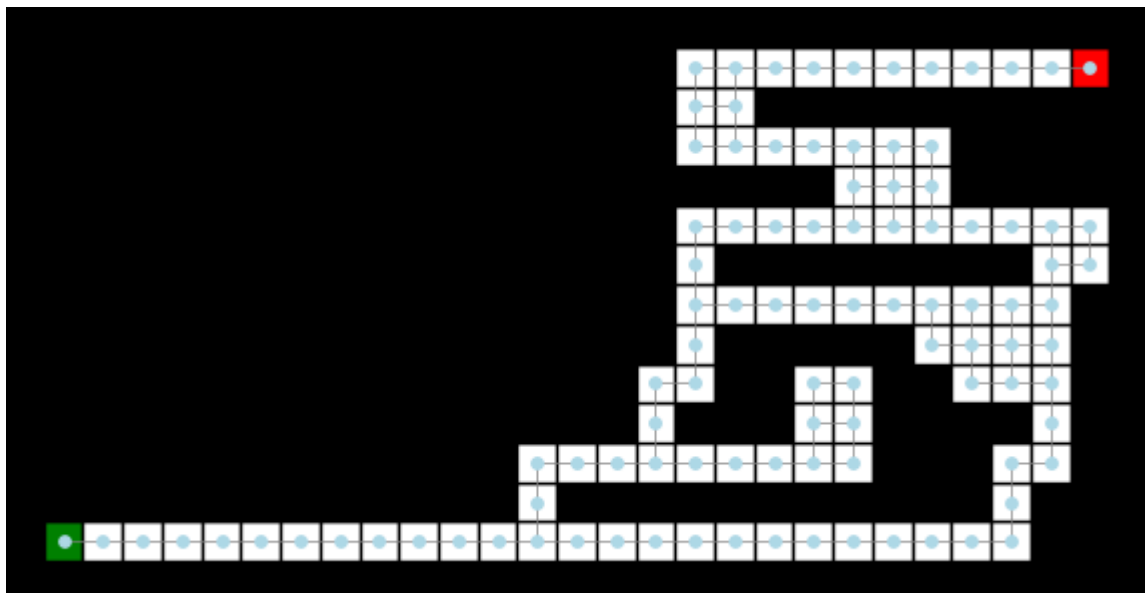


Gráfico 7: Resultados obtenidos mediante la implementación del algoritmo de Nayfeth

Discusión:

A partir de los resultados obtenidos, se observó que el algoritmo de Nayfeth, por sí solo, no logra encontrar un camino hacia la salida del laberinto. Esto se debe a que muchas de las celdas no cumplen con las condiciones necesarias para transformarse en paredes, incluso después de aplicar 20 iteraciones del algoritmo para reducir al máximo la complejidad del laberinto. Esta limitación no solo incrementa el tiempo de ejecución, sino que también pone en evidencia la necesidad de complementar este enfoque con un algoritmo adicional que permita resolver el laberinto de forma efectiva.

En contraste, el algoritmo A* demostró su eficiencia al encontrar el camino más corto desde el punto de inicio hasta la salida sin presentar inconvenientes, lo que confirma su robustez y precisión en entornos bien definidos.

Laberinto 3

Resultados:

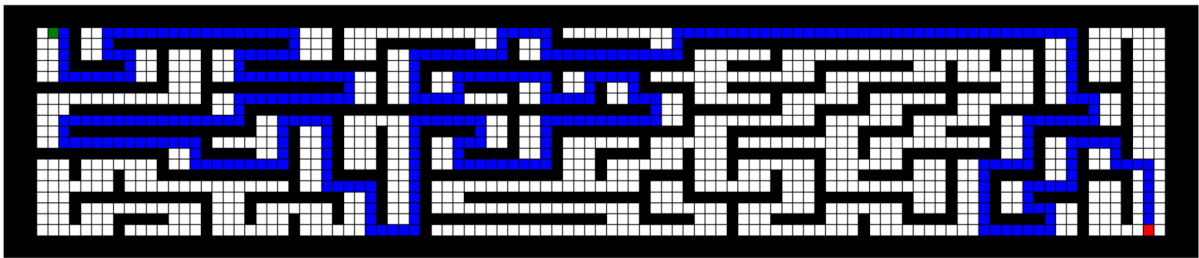


Gráfico 8: Resultados obtenidos después de aplicar el algoritmo de A*.

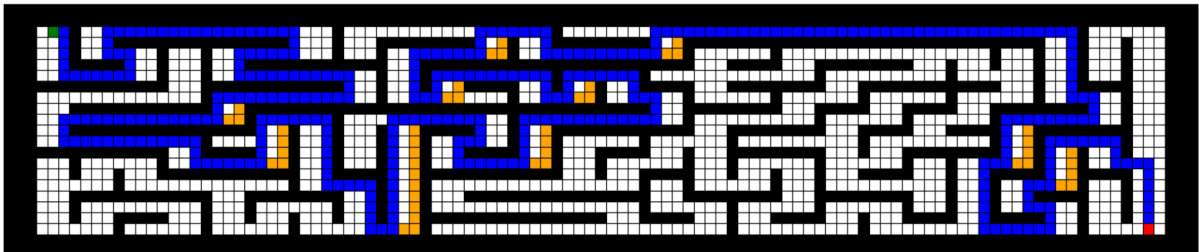


Gráfico 9: Resultados obtenidos después de aplicar el algoritmo de Dijkstra.

```
Comparación A* vs Dijkstra:

Longitud del camino:
A*: 345
Dijkstra: 345

Nodos visitados:
A*: 345
Dijkstra: 345

Tiempo de ejecución:
A*: 0.002004 s
Dijkstra: 0.003522 s
```

Gráfico 10: Datos de comparación A* y Dijkstra.

Discusión:

A partir de los resultados obtenidos, se observó que, al aumentar el ancho del laberinto en comparación con el Laberinto 1, el algoritmo de Dijkstra no logra simplificar de manera efectiva la estructura. Esto se debe a que, al haber más celdas vecinas, muchos corredores no cumplen la condición para ser transformados en paredes, lo que dificulta la limpieza del laberinto y puede impedir la obtención de una ruta óptima.

Además, al analizar visualmente la resolución del laberinto con Dijkstra, se puede notar que, aunque encuentra el camino más corto, también explora una gran cantidad de rutas alternativas (caminos marcados en color naranja) que no forman parte del camino final. Esto refleja el comportamiento característico de Dijkstra: realiza una exploración uniforme y exhaustiva desde el nodo inicial sin tener en cuenta la dirección hacia el objetivo. Como resultado, visita más nodos y consume más recursos, lo que se evidencia tanto en la cantidad de caminos secundarios evaluados como en el mayor tiempo de ejecución respecto a A*.

En contraste, el algoritmo A* fue capaz de encontrar consistentemente el camino más corto desde el punto de entrada (E) hasta el punto de salida (S), siempre que se respetara la condición de no permitir movimientos en diagonal. Esto confirma la eficacia de A* como algoritmo de búsqueda óptima incluso en laberintos amplios y con estructuras complejas.

Al comparar ambos algoritmos, se evidencia que A* y Dijkstra encuentran rutas con la misma longitud (345) y visitan la misma cantidad de nodos. Sin embargo, A* logra resolver el laberinto en menor tiempo (0.0020 s frente a 0.0035 s de Dijkstra), gracias a que incorpora una heurística que guía la búsqueda hacia el objetivo de manera más eficiente. En cambio, Dijkstra explora el espacio de forma uniforme sin una dirección preferente, lo que lo hace computacionalmente más costoso en laberintos grandes.

Conclusión:

El análisis comparativo entre los algoritmos A* y Dijkstra demuestra que, si bien ambos son capaces de encontrar caminos óptimos en términos de longitud y nodos visitados, A* presenta una ventaja significativa en cuanto a eficiencia computacional. Esto se debe a su uso de una heurística informada, que permite reducir el tiempo de resolución sin comprometer la calidad del resultado. En entornos donde el tiempo de respuesta es crítico o el espacio de búsqueda es amplio, A* se posiciona como la opción más adecuada. Por tanto, en aplicaciones de planificación de rutas en sistemas reales, como navegación o robótica, A* representa una mejor alternativa que Dijkstra, especialmente cuando se dispone de una heurística adecuada.

P2 - OPTIMIZACIÓN DE COLONIAS DE HORMIGAS

¿Qué ocurre con el segundo caso de estudio?

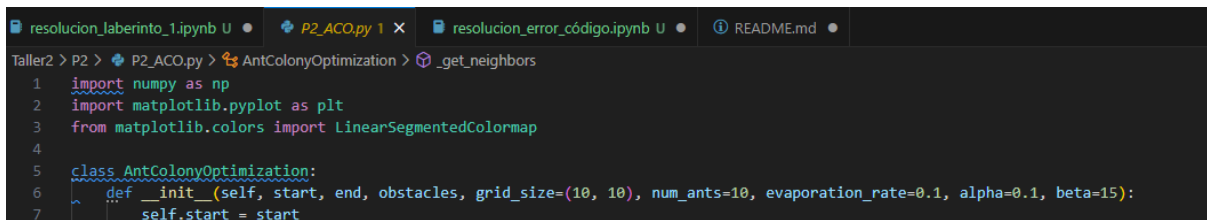
Problema Identificado

En el segundo caso de estudio, el algoritmo presentaba los siguientes problemas principales:

- Selección incorrecta del mejor camino: El código original solo consideraba la longitud del camino ($\text{len}(x)$) como criterio para seleccionar la mejor ruta, sin verificar si el camino llegaba efectivamente al destino.
- Parámetros subóptimos: Los valores predeterminados para número de hormigas, tasa de evaporación, α y β no eran adecuados para el laberinto más complejo del caso 2.

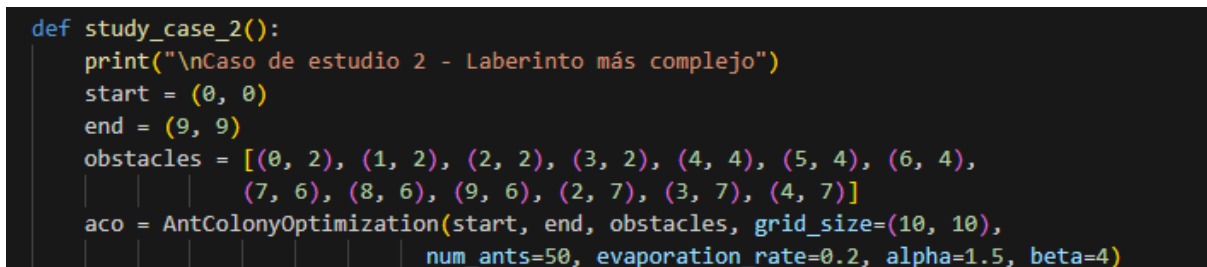
A continuación un detalle:

Parámetros originales:



```
resolucion_laberinto_1.ipynb • P2_ACO.py 1 X resolucion_error_código.ipynb U README.md •
Taller2 > P2 > P2_ACO.py > AntColonyOptimization > _get_neighbors
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.colors import LinearSegmentedColormap
4
5 class AntColonyOptimization:
6     def __init__(self, start, end, obstacles, grid_size=(10, 10), num_ants=10, evaporation_rate=0.1, alpha=0.1, beta=15):
7         self.start = start
```

Parámetros ajustados para caso 2:



```
def study_case_2():
    print("\nCaso de estudio 2 - Laberinto más complejo")
    start = (0, 0)
    end = (9, 9)
    obstacles = [(0, 2), (1, 2), (2, 2), (3, 2), (4, 4), (5, 4), (6, 4),
                (7, 6), (8, 6), (9, 6), (2, 7), (3, 7), (4, 7)]
    aco = AntColonyOptimization(start, end, obstacles, grid_size=(10, 10),
                               num_ants=50, evaporation_rate=0.2, alpha=1.5, beta=4)
```

- Aumenté el número de hormigas (20-50) para mejor exploración.
- Ajusté la tasa de evaporación (0.2-0.5) para balancear exploración y explotación.
- Valores optimizados para α (1-1.5) y β (4-5) para balancear influencia de feromonas y heurística.

- La cantidad de feromonas depositadas es ahora inversamente proporcional a la longitud del camino.
- Implementé evaporación más eficiente que preserva mejor las rutas prometedoras.
- **Sistema de deposición de feromonas proporcional:** Ahora las hormigas depositan más feromonas en caminos más cortos

```
def _deposit_pheromones(self, path):
    path_length = len(path)
    if path_length > 0:
        deposit_amount = 1.0 / path_length
        for position in path:
            self.pheromones[position[1], position[0]] += deposit_amount
```

- **Movimientos restringidos a 4 direcciones:** Para evitar soluciones poco realistas con movimientos diagonales

```
def _get_neighbors(self, position):
    pos_x, pos_y = position
    neighbors = []
    # Movimientos en 4 direcciones (no diagonal)
    for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
        new_x, new_y = pos_x + dx, pos_y + dy
        if (0 <= new_x < self.grid_size[0] and 0 <= new_y < self.grid_size[1] and
            (new_x, new_y) not in self.obstacles):
            neighbors.append((new_x, new_y))
    return neighbors
```

¿Qué propósito tiene cada parámetro en el modelo?

En el algoritmo ACO, cada parámetro tiene un papel crucial en el proceso de optimización. A continuación, se detallan los principales parámetros ajustados y su propósito en el modelo:

1. **Número de Hormigas:** Este parámetro controla la cantidad de soluciones generadas en cada iteración del algoritmo. Un número mayor de hormigas permite una mayor exploración del espacio de búsqueda, lo que aumenta las posibilidades de encontrar soluciones óptimas. Sin embargo, un número demasiado alto puede incrementar el tiempo de cómputo sin necesariamente mejorar la calidad de la solución.
2. **Tasa de Evaporación:** La tasa de evaporación determina la rapidez con la que las feromonas se disipan con el tiempo. Un valor alto de evaporación hace que las feromonas desaparezcan rápidamente, lo que puede evitar que el algoritmo se quede atrapado en soluciones subóptimas pero también puede reducir la posibilidad

de que se descubran soluciones prometedoras a largo plazo. Un valor bajo, como el que se ajustó a 0.2, mantiene las feromonas por más tiempo, favoreciendo la explotación de rutas que han mostrado ser buenas en iteraciones anteriores.

3. Valores de Alpha y Beta:

- **Alpha (α):** Este parámetro controla la influencia de las feromonas en la decisión de las hormigas. Un valor alto de alpha incrementa la importancia de las feromonas en el proceso de decisión, favoreciendo la exploración de rutas previamente recorridas por otras hormigas.
- **Beta (β):** Controla la influencia de la heurística, que en este caso es la distancia al objetivo. Un valor alto de beta favorece la selección de rutas cercanas al objetivo, mientras que un valor bajo hace que la selección sea más aleatoria, permitiendo una mayor exploración.

4. Restricción de Movimientos a 4 Direcciones (arriba, abajo, izquierda, derecha):

Este parámetro asegura que las hormigas solo puedan moverse dentro de un entorno cuadriculado siguiendo caminos viables (sin atravesar obstáculos). Limitar el movimiento a solo 4 direcciones evita trayectorias irrealistas y garantiza que las soluciones sean físicamente plausibles dentro del laberinto.

5. Depósito de Feromonas Proporcional a la Longitud del Camino:

Este parámetro asegura que las feromonas se depositen en función de la longitud del camino recorrido. Los caminos más cortos reciben más feromonas, lo que aumenta la probabilidad de que las hormigas seleccionen estas rutas en futuras iteraciones. Este enfoque favorece la convergencia del algoritmo hacia soluciones óptimas más eficientes.

6. Filtrado de Caminos No Válidos:

El propósito de este filtro es descartar aquellos caminos que no conducen al nodo objetivo, evitando que el algoritmo invierta recursos en soluciones incompletas o erróneas. Esto mejora la calidad de las soluciones y acelera la convergencia hacia caminos viables.

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

Sí se puede implementar este tipo de algoritmos para la solución del problema del vendedor viajante, considerando la robustez del modelo frente a metodologías que demandan muchos más recursos como sería el caso de un algoritmo BFS. Adicionalmente de acuerdo con la investigación realizada por Dorigo (1966), este algoritmo ha demostrado un mayor rendimiento en comparación con otros que de igual forma simulan comportamientos naturales como son los algoritmos genéticos y evolutivos.

Por otro lado dentro del proceso de implementación del modelo se definen los siguientes pasos:

- **Representación e Inicialización:**

- Se modela el TSP como un grafo, donde las ciudades son nodos y las distancias son las aristas.
- Se inicializan los niveles de feromona y se definen los parámetros: número de hormigas, α , β y tasa de evaporación.
- **Construcción de Recorridos por las Hormigas:**
 - Cada hormiga construye una ruta eligiendo la siguiente ciudad según una probabilidad basada en la feromona y la distancia.
 - Las ciudades visitadas se registran para evitar repeticiones, y varias hormigas exploran al mismo tiempo.
- **Actualización de Feromonas (Evaporación y Depósito):**
 - Una vez completados los recorridos, se aplica evaporación para reducir la feromona global y fomentar la exploración.
 - Luego, las hormigas depositan feromonas en las rutas utilizadas, premiando las más cortas con mayor intensidad.
- **Iteración y Terminación:**
 - Se repiten los pasos de construcción y actualización durante varias iteraciones.
 - El algoritmo finaliza cuando se alcanza un número fijo de ciclos o las soluciones convergen; se retorna la mejor ruta hallada.

Referencias

Dorigo, M. (1996). The Any System Optimization by a colony of cooperating agents. *IEEE Trans. System, Man & Cybernetics-Part B*, 26(1), 1-13.

Colaboración de integrantes

Integrante	Actividades Realizadas
Daniel Arias	División de tareas Laberinto 2 Pregunta A, D de la segunda parte
Santiago Rodriguez	División de tareas Laberinto 1 Pregunta A, B de la segunda parte
André Ramirez	División de tareas Laberinto 3 Pregunta A, C de la segunda parte

Link al repositorio

<https://github.com/Borreguin/WS-USFQ/blob/Grupo-2/Taller2>