

Introducción

Este proyecto implementa diversos algoritmos de búsqueda para resolver problemas de navegación en un laberinto. El objetivo es encontrar el camino desde una posición inicial (representada como "ratón") hasta un objetivo (representado como "queso") dentro de una matriz que representa el laberinto. Para ello, se utilizan estas estrategias de búsqueda:

búsqueda en amplitud, búsqueda en profundidad, búsqueda por costo uniforme, búsqueda Limitada por Profundidad, búsqueda por Profundización Iterativa y búsqueda avara.

El proyecto está compuesto por dos archivos principales:

1. **estrategias.py**: Contiene la implementación detallada de los algoritmos de búsqueda. Es el núcleo del proyecto, donde se define cómo cada estrategia explora el grafo generado a partir de la matriz.
2. **busquedas.py**: Organiza y encapsula las estrategias de búsqueda implementadas en estrategias.py, facilitando su uso. Este archivo actúa como una interfaz para ejecutar las diferentes estrategias.

En este informe, se explica en detalle cada parte del código, incluyendo cómo funcionan las estrategias de búsqueda y cómo los archivos interactúan entre sí.

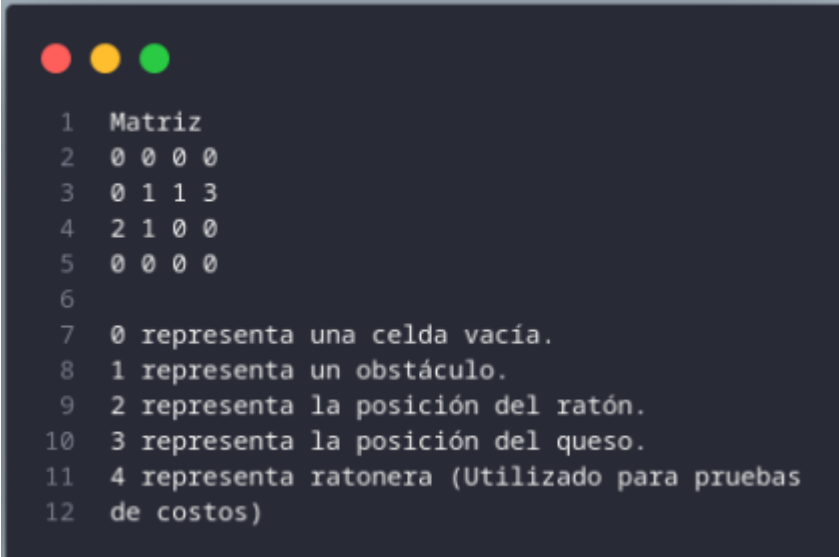
Librerías Utilizadas:

- **networkx**: Crear y analizar grafos.
- **collections.deque**: Manejo eficiente de colas y pilas.
- **matplotlib.pyplot**: Visualizar grafos y resultados.
- **pygraphviz**: Renderizar grafos con Graphviz.
- **io.BytesIO**: Manipulación de datos binarios en memoria.
- **Pillow (Image, ImageTk)**: Procesar y mostrar imágenes.
- **time**: Medir tiempo y pausas.
- **random**: Generar datos aleatorios.
- **heapq**: Implementar colas de prioridad.

Estructura General del Código

Ambos archivos trabajan en conjunto de la siguiente manera:

- **Preparación del Grafo:**
 - La matriz del laberinto se convierte en un grafo, donde cada celda transitable es un nodo.
 - Los obstáculos (representados por el valor 1) se eliminan del grafo, ya que no son accesibles.



```
1  Matriz
2  0 0 0 0
3  0 1 1 3
4  2 1 0 0
5  0 0 0 0
6
7  0 representa una celda vacía.
8  1 representa un obstáculo.
9  2 representa la posición del ratón.
10 3 representa la posición del queso.
11 4 representa ratonera (Utilizado para pruebas
12 de costos)
```

Imagen 1. Matriz y representación de valores de la matriz.

- **Dibujar el árbol:**

Propósito: Dibujar y renderizar un árbol utilizando Graphviz con nodos estilizados según su estado en el recorrido.

Parámetros:

- arbol: Diccionario que representa el árbol, donde las claves son nodos y los valores son listas de hijos.
- nodo_actual: Nodo destacado como el actual en el recorrido.
- path: Lista de nodos que forman el camino recorrido hasta el momento.

```

1 def dibujar_arbol(arbol, nodo_actual, path=[]):
2     G = pgv.AGraph(strict=False, directed=True)
3
4     # Crear las aristas del árbol
5     for nodo in arbol:
6         if arbol[nodo]:
7             for hijo in arbol[nodo]:
8                 G.add_edge(str(nodo), str(hijo))
9
10    # Dibujar los nodos
11    for nodo in G.nodes():
12        G.get_node(nodo).attr['label'] = nodo
13        G.get_node(nodo).attr['shape'] = 'ellipse'
14        G.get_node(nodo).attr['style'] = 'filled' # Hacer que el nodo esté relleno
15        G.get_node(nodo).attr['fontcolor'] = 'white' # Cambiar el color del texto del label a blanco
16
17        if str(nodo) in [str(p) for p in path]:
18            G.get_node(nodo).attr['color'] = 'green'
19            G.get_node(nodo).attr['fillcolor'] = 'green' # Rellenar con color verde
20        elif str(nodo) == str(nodo_actual):
21            G.get_node(nodo).attr['color'] = 'red'
22            G.get_node(nodo).attr['fillcolor'] = 'red' # Rellenar con color rojo
23        else:
24            G.get_node(nodo).attr['color'] = 'black' # Cambiar el borde a negro
25            G.get_node(nodo).attr['fillcolor'] = 'black' # Rellenar con color negro
26
27    # Generar el layout y guardar la imagen
28    G.layout(prog='dot')
29
30    buf = BytesIO()
31    G.draw(buf, format='png')
32    buf.seek(0)
33
34    return Image.open(buf)

```

Imagen 2. Matriz y representación de valores de la matriz.

- **Estrategias de Búsqueda:**
 - En estrategias.py, se definen varios algoritmos que exploran el grafo de manera diferente dependiendo de la estrategia seleccionada.
- **Coordinación de Estrategias:**
 - En busquedas.py, se encapsulan las estrategias en funciones individuales que configuran los parámetros necesarios y gestionan su ejecución.
- **Resultados:**
 - Cada estrategia devuelve:
 - El camino encontrado (si existe).
 - Los nodos visitados durante la búsqueda.
 - El árbol de búsqueda generado.

Primera Parte: estrategias.py

Estrategias de Búsqueda

En este archivo se implementan diferentes enfoques para explorar el grafo del laberinto. A continuación, se explican cada una de las estrategias con detalle y la función que las implementa(*buscar_ruta*):

1. Búsqueda por Amplitud (*busqueda_por_amplitud*)

Inicialización:

- a. Se crea un grafo 2D (*graph*) que representa la matriz, donde cada celda es un nodo conectado a sus vecinos adyacentes.
- b. Los obstáculos se eliminan del grafo para evitar que el algoritmo los considere en el recorrido.
- c. La función verifica si hay nodos padres con hijos sin explorar en el árbol (*arbol*) y, si encuentra alguno, toma ese nodo como punto de inicio.

2. Cola y Exploración por Niveles:

- a. La búsqueda comienza desde *nodo_inicial*, y se usa una cola (*queue*) para almacenar los nodos por explorar, en niveles crecientes.
- b. La búsqueda se expande a un número limitado de niveles (*nodos_expandir*), para controlar la profundidad de la exploración.

3. Exploración de Vecinos:

- a. Para cada nodo, si no ha sido visitado, se marca como visitado, se agrega a la ruta y se revisa si es el nodo objetivo (*posicion_queso*).
- b. Si encuentra el objetivo, reconstruye el camino hacia atrás usando el diccionario *parent* y devuelve el camino completo.
- c. Si el objetivo no se encuentra, se obtienen los vecinos no explorados y se agregan a la cola para futuras expansiones, además de actualizar el árbol visual (*arbol*), que se dibuja en cada expansión.

4. Retroceso a Niveles Superiores:

- a. Al finalizar un nivel, revisa si existen nodos no explorados en niveles superiores y, de ser así, retrocede para explorar esos nodos.

Resultados:

La función devuelve la ruta al objetivo (si se encuentra), el conjunto de nodos visitados y el diccionario *parent*, que guarda la relación entre nodos y sus padres para reconstruir caminos.

[Código de busqueda por amplitud](#)

2. Búsqueda Preferente por Profundidad (*busqueda_preferente_por_profundidad*)

1. Inicialización:

- a. Se construye un grafo 2D (*graph*) a partir de la matriz, donde cada celda representa un nodo conectado a sus vecinos adyacentes. Los obstáculos se eliminan del grafo para que no sean considerados en el recorrido.
- b. La búsqueda comienza con una pila (*stack*) que almacena tuplas de nodos y sus niveles de profundidad, iniciando en el nodo de inicio a nivel 1.
- c. Se crea un diccionario *visited_levels* para registrar el nivel de cada nodo visitado.

2. Recorrido en Profundidad:

- a. La pila se utiliza para realizar el recorrido en profundidad, donde cada nodo se extrae junto con su nivel de profundidad. Si el nivel supera el límite definido por *nodos_expandir*, el nodo se omite para controlar la profundidad de la búsqueda.
- b. Si el nodo actual es el objetivo (*posicion_queso*), se reconstruye y retorna el camino encontrado hacia el objetivo, empleando el diccionario *parent* para enlazar cada nodo con su predecesor.

3. Exploración de Vecinos en Orden:

- a. Los vecinos del nodo actual se obtienen en un orden predefinido: arriba, derecha, abajo, e izquierda.
- b. Luego, los vecinos válidos (no visitados y dentro del grafo) se agregan en orden inverso a la pila. Este enfoque permite que la búsqueda se expanda primero hacia el último vecino de la lista, manteniendo la preferencia en una dirección específica.

4. Construcción y Visualización del Árbol:

- a. Cada nuevo nodo agregado a la pila se incorpora al árbol de búsqueda (*arbol*) como hijo del nodo actual. El árbol se actualiza visualmente en cada expansión, mostrando el estado del árbol de búsqueda en tiempo real.

Resultados:

Si encuentra el objetivo, devuelve el camino hacia el mismo, los nodos visitados y el diccionario *parent*. Si no encuentra el objetivo, retorna el camino explorado hasta ese punto.

[Código de busqueda_preferente_por_profundidad](#)

3. Búsqueda por Costo Uniforme(*busqueda_por_costo_uniforme*)

1. Inicialización:

- a. Se construye un grafo bidimensional (*graph*) basado en la matriz, donde cada celda representa un nodo conectado a sus vecinos. Los obstáculos (celdas con valor 1) se eliminan del grafo.

- b. Se utiliza una cola de prioridad (*queue*) implementada con un min-heap para almacenar los nodos junto con sus costos acumulados, comenzando con el nodo inicial y un costo de 0.
 - c. Un diccionario *costs* guarda el costo mínimo registrado para cada nodo.
- 2. **Recorrido por Costo Mínimo:**
 - a. Mientras haya nodos en la cola y no se haya superado el límite de expansiones (*nodos_expandir*), se extrae el nodo con el menor costo.
 - b. Si el nodo actual ya fue visitado, se omite. De lo contrario, se marca como visitado, y si es el nodo objetivo (*posicion_queso*), se reconstruye el camino desde el objetivo hasta el nodo inicial usando el diccionario *parent*, que guarda la relación padre-hijo entre nodos.
- 3. **Exploración de Vecinos:**
 - a. Para el nodo actual, se obtienen sus vecinos en orden horario (arriba, derecha, abajo e izquierda) y se filtran aquellos que no son obstáculos y están en el grafo.
 - b. Se calcula el costo para cada vecino, considerando un costo adicional de +3 si el vecino es de tipo 4 (un tipo de celda especial), o +1 en otros casos.
 - c. Si el vecino no está en *costs* o el nuevo costo es menor que el registrado, se actualizan el costo, el nodo padre y se agrega el vecino a la cola de prioridad con su costo actualizado.
- 4. **Actualización y Visualización del Árbol:**
 - a. Cada nuevo vecino se añade al árbol de búsqueda (*arbol*) y se visualiza en tiempo real, mostrando el estado del árbol de búsqueda a medida que se exploran los nodos.

Resultados:

Si se encuentra el objetivo, la función devuelve el camino más corto hasta él, los nodos visitados y el diccionario *parent*. Si no se encuentra un camino al objetivo, retorna el último nodo inicial y el conjunto de nodos visitados.

[Código de busqueda_por_costo_uniforme](#)

4. Búsqueda Limitada por Profundidad(*busqueda_limitada_por_profundidad*)

- 1. **Inicialización:**
 - a. Se construye un grafo bidimensional (*graph*) basado en la matriz, donde cada celda representa un nodo y los obstáculos (celdas con valor 1) se eliminan para que no sean considerados en la búsqueda.
 - b. Se utiliza una pila (*stack*) que almacena tuplas de nodos y su profundidad actual, comenzando con el nodo inicial y una profundidad de 0.
 - c. Se guarda *ultimo_nodo_expandido* para registrar el último nodo visitado dentro del límite de profundidad.
- 2. **Recorrido en Profundidad con Límite:**

- a. En cada iteración, se extrae un nodo y su profundidad de la pila.
 - b. Si la profundidad supera el límite *nodos_expandir*, el bucle termina.
 - c. Si el nodo no ha sido visitado, se marca como visitado y se actualiza el último nodo expandido. Si el nodo coincide con la posición del objetivo (*posicion_queso*), se reconstruye el camino usando el diccionario *parent* para enlazar los nodos en orden inverso.
3. **Exploración de Vecinos:**
- a. Para cada nodo, se obtienen sus vecinos en un orden predefinido: arriba, derecha, abajo, e izquierda.
 - b. Los vecinos válidos se agregan en orden inverso a la pila para garantizar que se procesen en el orden de expansión deseado.
 - c. Cada vecino se registra en el árbol de búsqueda (*arbol*) como hijo del nodo actual, y se visualiza la expansión del árbol en tiempo real.
4. **Visualización y Actualización del Árbol:**
- a. Cada vez que un nodo se expande, se dibuja el árbol actualizado, permitiendo la visualización de la estructura de exploración hasta ese momento.

Resultados:

Si encuentra el objetivo, devuelve el camino hacia él, los nodos visitados y el diccionario *parent*. Si no se encuentra el objetivo dentro del límite de profundidad, la función devuelve el último nodo expandido y el conjunto de nodos visitados.

[Código de busqueda limitada por profundidad](#)

5. Búsqueda por Profundización Iterativa(*busqueda_profundidad_interactiva*)

1. **Inicialización del Grafo:**
- a. Se crea un grafo bidimensional (*graph*) basado en la matriz de entrada, donde cada celda representa un nodo. Los obstáculos en la matriz (celdas con valor 1) se eliminan del grafo, de modo que no sean considerados en la búsqueda.
2. **Función de Búsqueda Limitada (*busqueda_limitada*):**
- a. Esta función realiza una búsqueda en profundidad hasta una profundidad específica (*profundidad_limite*), partiendo desde un nodo dado.
 - b. Utiliza una pila (*stack*) que almacena tuplas de nodos y su profundidad.
 - c. En cada iteración, se expande un nodo, marcándolo como visitado.
 - d. Si el nodo actual es el objetivo (*posicion_queso*), se reconstruye el camino hacia él utilizando el diccionario *parent*, y se devuelve el camino, los nodos visitados y el diccionario *parent*.
3. **Exploración de Vecinos:**
- a. Para cada nodo expandido, se obtienen sus vecinos en el orden: arriba, derecha, abajo e izquierda.

- b. Los vecinos válidos (no obstáculos) se insertan en la pila para seguir expandiendo, siempre y cuando no hayan sido visitados y la profundidad no exceda *profundidad_limite*.
 - c. Los vecinos se agregan al árbol de búsqueda (*arbol*), y se actualiza la visualización del árbol para reflejar la expansión.
4. **Iteración con Profundidad Creciente:**
- a. La función principal comienza con una profundidad límite de θ y la incrementa en cada iteración hasta alcanzar el límite *nodos_expandir*.
 - b. En cada iteración, se llama a *busqueda_limitada* con el nuevo límite de profundidad.
 - c. Si *busqueda_limitada* encuentra el objetivo, devuelve el resultado. Si no, se continúa con un límite de profundidad mayor.

Resultados:

Si encuentra el objetivo, devuelve el camino, los nodos visitados y el diccionario *parent*. Si no encuentra el objetivo dentro de la profundidad máxima permitida, devuelve el nodo inicial, el conjunto de nodos visitados y el diccionario *parent*.

[Código de búsqueda profundidad interactiva](#)

6. Búsqueda Avara(*busqueda_avara*)

1. **Inicialización del Grafo y Obstáculos:**
 - a. Se crea un grafo bidimensional a partir de la matriz, donde cada celda es un nodo y las celdas con valor 1 se eliminan del grafo, representando obstáculos.
2. **Configuración de la Cola de Prioridad:**
 - a. Se usa una cola de prioridad para almacenar nodos según el valor de la heurística (distancia Manhattan hasta el objetivo). Esto permite expandir siempre el nodo más cercano al objetivo.
3. **Bucle Principal de Búsqueda:**
 - a. Mientras haya nodos en la cola y el límite de expansión no se haya alcanzado:
 - i. Se extrae el nodo con la menor distancia heurística.
 - ii. Si el nodo ya fue visitado, se omite.
 - iii. Si el nodo es el objetivo, se reconstruye el camino mediante los nodos padres y se devuelve.
4. **Exploración de Vecinos:**
 - a. Para cada vecino del nodo actual, si no ha sido visitado:
 - i. Se asigna el nodo actual como su padre.
 - ii. Se calcula la heurística y se añade a la cola de prioridad.
 - iii. Se actualiza el árbol visual y se espera un breve tiempo antes de continuar.
5. **Límite de Expansión:**

- a. La expansión se detiene cuando se alcanza el número máximo de nodos especificado por *nodos_expandir*.

Resultados:

Si se encuentra el objetivo, se devuelve el camino hacia él; de lo contrario, devuelve el último nodo expandido, los nodos visitados y la estructura de nodos padres.

Código búsqueda avara

Buscar ruta: El código busca una ruta en una matriz desde la posición de un ratón hasta un queso utilizando diferentes estrategias de búsqueda (amplitud, profundidad, costo uniforme, etc.). Construye un árbol de búsqueda para rastrear las conexiones entre nodos explorados, evita repetir nodos visitados y cambia de estrategia aleatoriamente. Durante el proceso, actualiza visualizaciones en tiempo real para mostrar el estado del árbol y la estrategia en uso. Finaliza al encontrar el queso, devolviendo la ruta, o si se agotan las estrategias sin éxito.

```
1 def buscar_ruta(
2     matriz, posicion_raton, posicion_queso,
3     actualizar_arbol_callback, actualizar_estrategia_callback, nodos_expandir
4 ):
5     estrategias = [busqueda_por_amplitud,
6                   busqueda_por_costo_uniforme,
7                   busqueda_avara,
8                   busqueda_limitada_por_profundidad,
9                   busqueda_preferente_por_profundidad] # Lista de estrategias
10    arbol = {posicion_raton: []}
11    nodo_actual = posicion_raton
12    path = [] # Cambié para no inicializar con el nodo inicial
13    visited = set()
14    parent = {nodo_actual: None} # Mapa de padres global para todas las estrategias
15
16    while nodo_actual != posicion_queso:
17        if not estrategias:
18            print("No quedan estrategias por seleccionar.")
19            break
20
21        # Seleccionar la estrategia
22        estrategia = random.choice(estrategias)
23        estrategias.remove(estrategia)
24        nombre_estrategia = estrategia.__name__.capitalize()
25        actualizar_estrategia_callback(nombre_estrategia)
26        print(f"Estrategia actual: {nombre_estrategia}")
27        print(f"Nodo actual antes de la estrategia: {nodo_actual}")
28
29        # Ejecutar la estrategia
30        resultado, visited, parent = estrategia(
31            matriz, nodo_actual, posicion_queso,
32            arbol, actualizar_arbol_callback, actualizar_estrategia_callback,
33            nodos_expandir, visited, parent
34        )
35
36        if resultado:
37            print(f"Resultado de la estrategia {nombre_estrategia}: {resultado}")
38
39            # Si se encuentra una ruta válida, verificar si el queso está en el resultado
40            if resultado and resultado[-1] == posicion_queso:
41                return resultado
42
43        else:
44            # Obtener el nodo no explorado o usar el último nodo expandido
45            nuevo_nodo = obtener_nodo_no_explorado(arbol, visited)
46            nodo_actual = nuevo_nodo if nuevo_nodo else resultado[-1]
47            if nodo_actual not in visited: # Evitar agregar nodos ya visitados
48                path.append(nodo_actual)
49                print(f"Nuevo nodo actual: {nodo_actual}")
50
51        else:
52            print("No se pudo encontrar una ruta con la estrategia actual. Continuando con la siguiente.")
53            nuevo_nodo = obtener_nodo_no_explorado(arbol, visited)
54            nodo_actual = nuevo_nodo if nuevo_nodo else nodo_actual
55
56        print(f"Nodo actual después de la estrategia: {nodo_actual}")
57        print(f"Nodos visitados hasta ahora: {visited}")
58
59    return None
```

Imagen 3. Función buscar ruta selecciona aleatoriamente las estrategias y realiza la búsqueda.

Segunda Parte: busquedas.py

Este archivo encapsula las estrategias de búsqueda en funciones específicas, configurando los parámetros necesarios para su ejecución.

Funciones Principales

Cada estrategia tiene su propia función, que realiza los siguientes pasos:

- 1. **Inicialización:**
 - Define estructuras como el árbol, los nodos visitados y el diccionario de padres.
 - 2. **Ejecución de la Estrategia:**
 - Llama a la función correspondiente en estrategias.py.
 - 3. **Gestión de Resultados:**
 - Si encuentra el queso, devuelve el camino.
 - Si no hay más opciones, termina sin solución.
-

Comparación General de Estrategias

Estrategia	Camino más corto	Uso de memoria	Complejidad
Búsqueda por Amplitud (BFS)	Sí	Alta	Sencilla
Búsqueda por Profundidad (DFS)	No	Baja	Moderada
Costo Uniforme	Sí (costo mínimo)	Media	Compleja
Limitada por Profundidad	No	Baja	Moderada
Profundización Iterativa	Sí	Media	Moderada
Búsqueda Avara	No	Media	Compleja

Elaboración de Pruebas y Configuración de la Matriz

Pruebas para validar el funcionamiento de la búsqueda, y la matriz podrá generarse de forma aleatoria o configurarse manualmente según se necesite.

Matriz 4x4

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Salidas aleatorias:

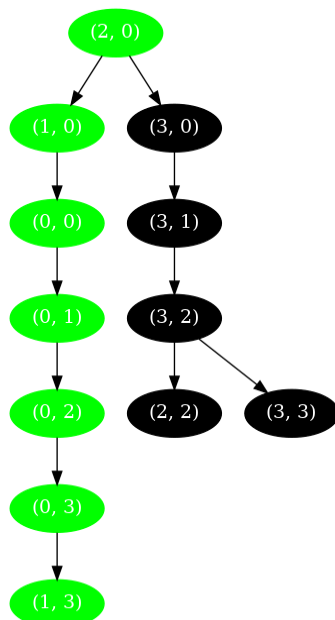
nodos de expansión = 4

Estrategias utilizadas:

1. Búsqueda por amplitud
2. Búsqueda por costo uniforme

Ruta encontrada:

[(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (0, 3), (1, 3)]



(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Matriz 4x4

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Salidas aleatorias:

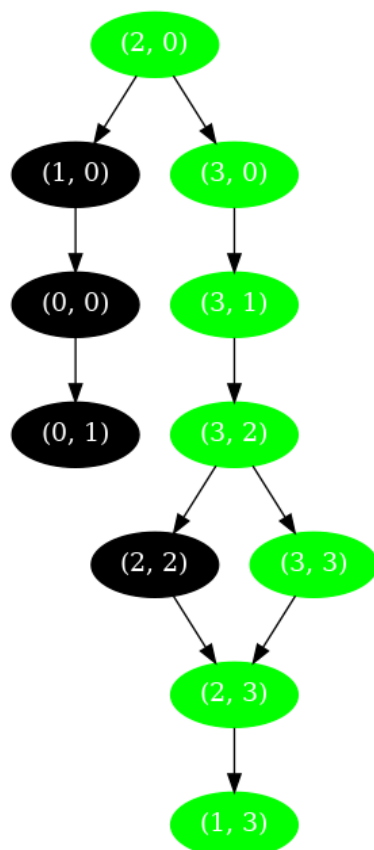
nodos de expansión = 2

Estrategias utilizadas:

1. Búsqueda preferente por profundidad
2. Búsqueda por costo uniforme
3. Búsqueda avara
4. Búsqueda limitada por profundidad

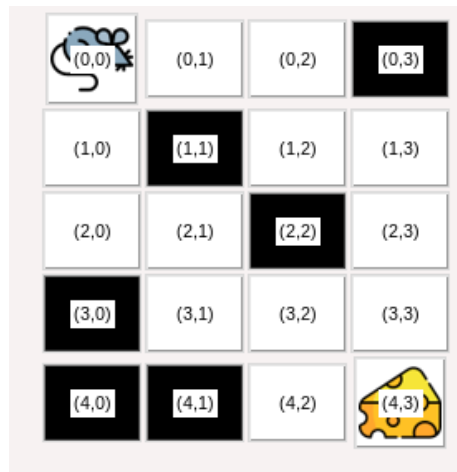
Ruta encontrada:

[(2, 0), (3, 0), (3, 1), (3, 2), (3, 3), (2, 3), (1, 3)]



(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Matriz 5x4



Salidas aleatorias:

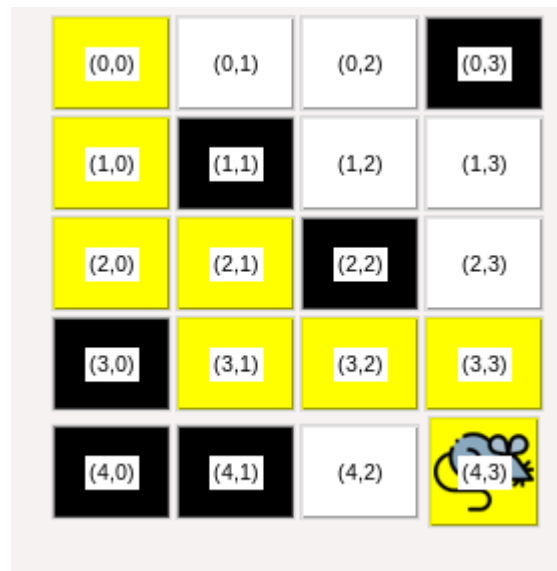
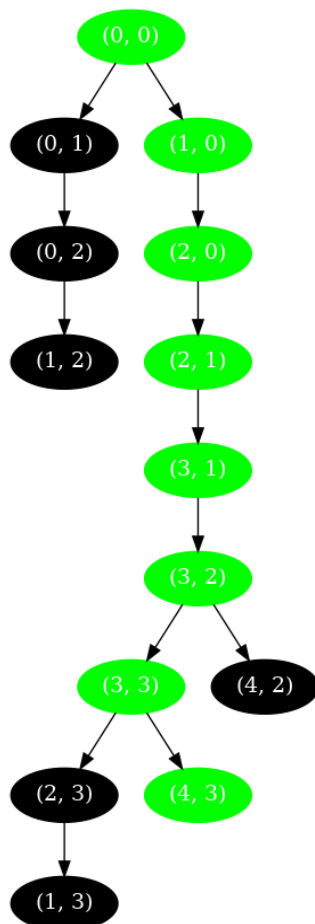
nodos de expansión = 4

Estrategias utilizadas:

1. Búsqueda limitada por profundidad
2. Búsqueda por costo uniforme
3. Búsqueda preferente por profundidad
4. Búsqueda por amplitud
5. Búsqueda avara

Ruta encontrada:

[(0, 0), (1, 0), (2, 0), (2, 1), (3, 1), (3, 2), (3, 3), (4, 3)]



Conclusión

Este proyecto está diseñado para explorar un laberinto utilizando múltiples estrategias de búsqueda, cada una adecuada para diferentes escenarios. La modularidad de los archivos permite agregar nuevas estrategias o modificar las existentes con facilidad. Además, proporciona una excelente base para aprender sobre algoritmos de búsqueda y su aplicación en problemas prácticos.