Práctica 1: Algoritmo A*

Ian Álvarez de Francisco y Alejandro Relanzón Coello

En esta práctica hemos desarrollado tres versiones del algoritmo A*, cada uno siendo una versión más avanzada del anterior. En cada uno se dará una explicación del algoritmo y detalles de su implementación junto con ejemplos. La práctica está desarrollada en python, usando la librería de matplotlib para la visualización de la matriz, por lo que esta debe estar instalada en el entorno donde se vaya a ejecutar el programa. Cada Parte tiene su archivo python ejecutable(aestrella1.py, aestrella2.py, aestrella3.py) y su matriz de ejemplo(matriz.txt, matriz2.txt, matriz3.txt).

Parte 1: A* Simple

La función de costo total que usa A* para priorizar los nodos es:

$f(n) = g(n) + h(n)$

Donde:

- g(n) es el costo real desde el punto de inicio hasta el nodo actual.
- h(n) es la estimación heurística del costo restante hasta el destino.

En este caso, se ha utilizado la **distancia euclidiana** como función heurística:

$h(n) = \sqrt{(xfinal - xnodo)2 + (yfinal - ynodo)2}$

El algoritmo A* en el código sigue los siguientes pasos:

1. **Cargar la matriz desde un archivo (`leer_matriz_fichero()`)**
    - Se lee la matriz desde un archivo `matriz.txt`, identificando obstáculos (`X`), el punto de inicio (`2`) y el destino (`3`).
    - Se almacena en una lista de listas en Python, donde:
        - `0` representa caminos transitables.
        - `-1` representa obstáculos.
        - `2` es el nodo de inicio.
        - `3` es el nodo objetivo.
2. **Inicializar estructuras de datos**
    - Se define una **lista abierta** (`lista_abierta`), que es una cola de prioridad (implementada con `heapq`) para almacenar los nodos a explorar, ordenados por su costo total f(n).

- Se define una **lista cerrada** (`lista_cerrada`), que guarda los nodos ya explorados para evitar procesarlos nuevamente.
- Se inicializan los diccionarios `g_coste` y `f_coste` para almacenar los costos de cada nodo.
- Se establece un diccionario `padres` para reconstruir la ruta al final.

3. **Iniciar el proceso de búsqueda**

   - Se agrega el nodo de inicio a la lista abierta con un costo f(n) calculado.
   - Se define un conjunto de **ocho direcciones de movimiento** (vertical, horizontal y diagonal) para explorar posibles movimientos.

4. **Bucle principal del algoritmo**

   - Mientras haya nodos en la lista abierta:
     - Se extrae el nodo con menor costo f(n).
     - Si el nodo extraído es el destino, se reconstruye el camino y se devuelve.
     - Se marcan los nodos ya explorados en la lista cerrada.
     - Se examinan los nodos vecinos (movimientos posibles en la matriz).
     - Si el nodo vecino no es un obstáculo ni ha sido explorado:
       - Se calcula el nuevo costo g(n).
       - Se actualizan los valores de g(n) y f(n) si se encuentra un mejor camino.
       - Se actualiza el nodo padre en `padres` para reconstruir la ruta.

5. **Reconstrucción del camino (`reconstruir_camino()`)**

   - Si el destino se encuentra, se reconstruye la ruta a partir del diccionario `padres`, recorriendo los nodos desde el destino hasta el inicio y almacenándolos en una lista.

6. **Visualización del resultado (`mostrar_matriz()`)**

   - Se genera una representación gráfica de la matriz con:
     - Color **negro** para obstáculos.
     - Color **blanco** para caminos transitables.
     - Color **verde** para el punto de inicio.
     - Color **rojo** para el destino.
     - Color **azul** para la ruta encontrada.

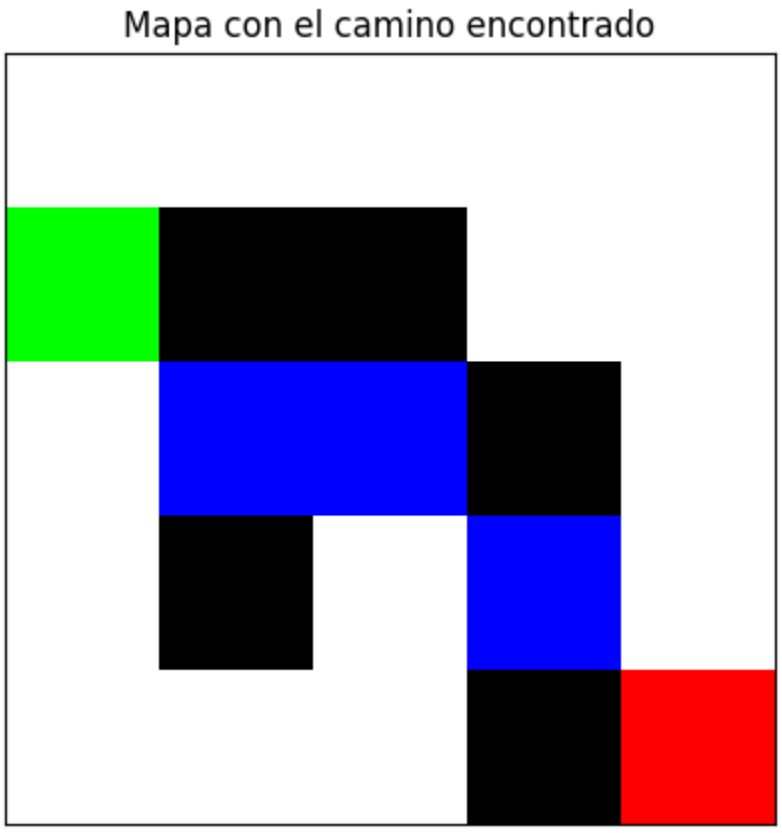
Ejemplo de uso:

```
0 0 0 0 0
2 X X 0 0
0 0 0 X 0
0 X 0 0 0
0 0 0 X 3
```

matriz.txt

Proceso

1. Se carga la matriz y se identifica el punto de inicio `(1,0)` y el destino `(4,4)`.
2. A* explora nodos, priorizando aquellos con menor costo total f(n).
3. Encuentra un camino óptimo evitando obstáculos
4. Devuelve la ruta como una lista de coordenadas:
   `(1, 0) (2, 1) (2, 2) (3, 3) (4, 4)`
5. Se muestra una imagen con la ruta en azul.



Código:

```
import heapq
import math
import matplotlib.pyplot as plt
import numpy as np

def leer_matriz_fichero(nombre_fichero):
    matriz = []
    inicio = None
    final = None

    conversion = {
        'O': 0,  # Camino libre
        'X': -1,  # Obstáculo
        '2': 2,  # Inicio
        '3': 3   # Final
    }

    with open(nombre_fichero, 'r') as fichero:
        for i, linea in enumerate(fichero):
            fila = []
            for j, caracter in enumerate(linea.strip().split()):
                if caracter in conversion:
                    valor = conversion[caracter]
                    fila.append(valor)
                    if valor == 2:
                        inicio = (i, j)
                    elif valor == 3:
                        final = (i, j)
                else:
                    raise ValueError(f"Caracter no reconocido:
{caracter}")
            matriz.append(fila)

    return matriz, inicio, final

def distancia_euclidiana(nodo1, nodo2):
    return math.sqrt((nodo1[0] - nodo2[0])**2 + (nodo1[1] - nodo2[1])**2)

def astar(matriz, inicio, final):
```

```
    filas, columnas = len(matriz), len(matriz[0])
    lista_abierta = []
    lista_cerrada = set()

    # Colocar los obstáculos directamente en la lista cerrada
    for i in range(filas):
        for j in range(columnas):
            if matriz[i][j] == -1:
                lista_cerrada.add((i, j))

    # Inicializar el nodo de inicio
    g_coste = {inicio: 0}
    f_coste = {inicio: distancia_euclidiana(inicio, final)}
    padres = {inicio: None}

    # Agregar el nodo inicial a la lista abierta
    heapq.heappush(lista_abierta, (f_coste[inicio], inicio))

    # Direcciones de movimiento: vertical, horizontal y diagonal
    direcciones = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1),
(1, -1), (1, 1)]

    while lista_abierta:
        _, nodo_actual = heapq.heappop(lista_abierta)

        # Si hemos llegado al objetivo, reconstruir el camino
        if nodo_actual == final:
            return reconstruir_camino(padres, nodo_actual)

        lista_cerrada.add(nodo_actual)

        # Explorar vecinos
        for direccion in direcciones:
            vecino = (nodo_actual[0] + direccion[0], nodo_actual[1] +
direccion[1])

            # Verificar que el vecino esté dentro de los límites y no sea
un obstáculo o ya esté cerrado
            if 0 <= vecino[0] < filas and 0 <= vecino[1] < columnas and
vecino not in lista_cerrada:
```

```
                g_nuevo = g_coste[nodo_actual] +
distancia_euclidiana(nodo_actual, vecino)

                # Si no está en lista_abierta o encontramos un camino más
corto, actualizamos
                if vecino not in g_coste or g_nuevo < g_coste[vecino]:
                    g_coste[vecino] = g_nuevo
                    f_coste[vecino] = g_nuevo +
distancia_euclidiana(vecino, final)
                    padres[vecino] = nodo_actual
                    heapq.heappush(lista_abierta, (f_coste[vecino],
vecino))

    # No se encontró un camino
    return None

def reconstruir_camino(padres, nodo):
    camino = []
    while nodo is not None:
        camino.append(nodo)
        nodo = padres[nodo]
    camino.reverse()
    return camino

def mostrar_matriz(matriz, camino=None):
    filas, columnas = len(matriz), len(matriz[0])
    imagen = np.zeros((filas, columnas, 3))  # Imagen en RGB

    colores = {
        0: (1, 1, 1),  # Blanco para caminos libres
        -1: (0, 0, 0), # Negro para obstáculos
        2: (0, 1, 0),  # Verde para inicio
        3: (1, 0, 0)   # Rojo para final
    }

    # Pintar la matriz
    for i in range(filas):
        for j in range(columnas):
            imagen[i, j] = colores.get(matriz[i][j], (1, 1, 1))
```

```
    # Si hay un camino, dibujarlo en azul
    if camino:
        for nodo in camino:
            if matriz[nodo[0]][nodo[1]] not in [2, 3]:  # No sobrescribir
inicio y final
                imagen[nodo[0], nodo[1]] = (0, 0, 1)  # Azul para el
camino

    # Mostrar la imagen con Matplotlib
    plt.figure(figsize=(5, 5))
    plt.imshow(imagen)
    plt.grid(visible=True, color="gray", linewidth=0.5)
    plt.xticks([])
    plt.yticks([])
    plt.title("Mapa con el camino encontrado")
    plt.show()

# Prueba del algoritmo
nombre_fichero = "matriz.txt"
matriz, inicio, final = leer_matriz_fichero(nombre_fichero)

print("Matriz cargada:")
for fila in matriz:
    print(fila)

print(f"Inicio: {inicio}, Final: {final}")

camino = astar(matriz, inicio, final)
""" if camino:
    print("Camino encontrado:")
    for paso in camino:
        print(paso)
else:
    print("No se encontró un camino.") """

# Mostrar la matriz con el camino encontrado
mostrar_matriz(matriz, camino)
```

Parte 2: A* Simple con Way Points

A diferencia de la versión básica de A*, aquí el algoritmo encuentra una ruta que pasa por varios puntos intermedios, llamados way points. Para ello:

1. Se ordenan los puntos según su identificador numérico, siendo el 2 el inicio y 6 el final, con 2-5 siendo los way points, siendo obligatorio usar todos ellos, sino fallará.
2. Se resuelven rutas sucesivas entre los puntos, almacenando los caminos parciales.
3. Se concatenan las rutas, evitando duplicar el último nodo de cada tramo.

Si en algún punto no se encuentra una ruta, el proceso se detiene y se reporta el fallo.

Para el recorrido entre los way points se usa el mismo algoritmo que el empleado en la parte 1.

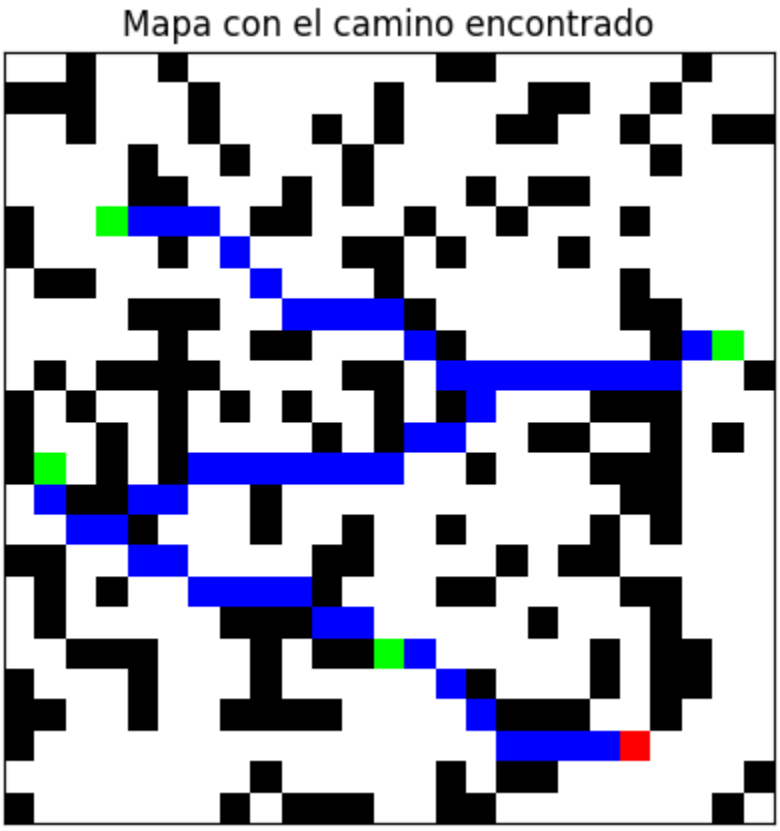
Ejemplo de uso:

```
0 0 X 0 0 X 0 0 0 0 0 0 0 X X 0 0 0 0 0 0 X 0 0
X X X 0 0 0 X 0 0 0 0 0 X 0 0 0 0 X X 0 0 X 0 0 0
0 0 X 0 0 0 X 0 0 0 X 0 X 0 0 0 X X 0 0 X 0 0 X X
0 0 0 0 X 0 0 X 0 0 0 X 0 0 0 0 0 0 0 0 0 X 0 0 0
0 0 0 0 X X 0 0 0 X 0 X 0 0 0 X 0 X X 0 0 0 0 0 0
X 0 0 2 0 0 0 0 X X 0 0 0 X 0 0 X 0 0 0 X 0 0 0 0
X 0 0 0 0 X 0 0 0 0 0 X X 0 X 0 0 0 X 0 0 0 0 0 0
0 X X 0 0 0 0 0 0 0 0 0 X 0 0 0 0 0 0 0 X 0 0 0 0
0 0 0 0 X X X 0 0 0 0 0 0 X 0 0 0 0 0 0 0 X X 0 0 0
0 0 0 0 0 X 0 0 X X 0 0 0 0 X 0 0 0 0 0 0 X 0 3 0
0 X 0 X X X X 0 0 0 0 X X 0 0 0 0 0 0 0 0 0 0 0 X
X 0 X 0 0 X 0 X 0 X 0 0 X 0 X 0 0 0 0 X X X 0 0 0
X 0 0 X 0 X 0 0 0 0 X 0 X 0 0 0 0 X X 0 0 X 0 X 0
X 4 0 X 0 X 0 0 0 0 0 0 0 0 0 X 0 0 0 X X X 0 0 0
0 0 X X 0 0 0 0 X 0 0 0 0 0 0 0 0 0 0 0 X X 0 0 0
0 0 0 0 X 0 0 0 X 0 0 X 0 0 X 0 0 0 0 X 0 X 0 0 0
X X 0 0 0 0 0 0 0 0 X X 0 0 0 0 X 0 X X 0 0 0 0 0
0 X 0 X 0 0 0 0 0 0 X 0 0 0 X X 0 0 0 0 X X 0 0 0
0 X 0 0 0 0 0 X X X 0 0 0 0 0 0 0 X 0 0 0 X 0 0 0
0 0 X X X 0 0 0 X 0 X X 5 0 0 0 0 0 0 0 X 0 X X 0 0
X 0 0 0 X 0 0 0 X 0 0 0 0 0 0 0 X 0 0 0 X 0 X X 0 0
X X 0 0 X 0 0 X X X X 0 0 0 0 0 0 X X X 0 0 X 0 0 0
X 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0
0 0 0 0 0 0 0 0 X 0 0 0 0 0 0 X 0 X X 0 0 0 0 0 0 X
X 0 0 0 0 0 0 X 0 X X X 0 0 X X 0 0 0 0 0 0 0 X 0
```

matriz2.txt

Proceso

1. Se carga la matriz y se identifica el punto de inicio `(6,4)` y el destino `(23,21)`, con los waypoints en este orden: `(10,24), (14,1), (20,13)`
2. A* explora nodos, priorizando aquellos con menor costo total f(n).
3. Encuentra un camino óptimo evitando obstáculos
4. Devuelve la ruta como una lista de coordenadas:
   `(5, 3),(5, 4),(5, 5),(5, 6),(6, 7),(7, 8),(8, 9),(8, 10),(8, 11),(8, 12),(9, 13),(10, 14),(10, 15),(10, 16),(10, 17),(10, 18),(10, 19),(10, 20),(10, 21),(9, 22),(9, 23),(9, 22),(10, 21),(10, 20),(10, 19),(10, 18),(10, 17),(10, 16),(11, 15),(12, 14),(12, 13),(13, 12),(13, 11),(13, 10),(13, 9),(13, 8),(13, 7),(13, 6),(14, 5),(14, 4),(15, 3),(15, 2),(14, 1),(13, 1),(14, 1),(15, 2),(15, 3),(16, 4),(16, 5),(17, 6),(17, 7),(17, 8),(17, 9),(18, 10),(18, 11),(19, 12),(19, 13),(20, 14),(21, 15),(22, 16),(22, 17),(22, 18),(22, 19),(22, 20)`
5. Se muestra una imagen con la ruta en azul.



Código: La diferencia con el punto uno, es que este código soporta distintos puntos. Estos los ordena y va añadiéndolos al nodo final recorriendo todos.

```
import heapq
import math
import matplotlib.pyplot as plt
import numpy as np

def leer_matriz_fichero(nombre_fichero):
    matriz = []
    puntos = {}

    conversion = {
        'O': 0,  # Camino libre
        'X': -1,  # Obstáculo
        '1': 1,  # Punto 1
        '2': 2,  # Punto 2
        '3': 3,  # Punto 3
        '4': 4,  # Punto 4
        '5': 5,  # Punto 5
        '6': 6,  # Punto 6
    }

    with open(nombre_fichero, 'r') as fichero:
        for i, linea in enumerate(fichero):
            fila = []
            for j, caracter in enumerate(linea.strip().split()):
                if caracter in conversion:
                    valor = conversion[caracter]
                    fila.append(valor)
                    if valor > 0:
                        puntos[valor] = (i, j)
                else:
                    raise ValueError(f"Caracter no reconocido:
{caracter}")
            matriz.append(fila)

    return matriz, puntos

def distancia_euclidiana(nodo1, nodo2):
```

```
    return math.sqrt((nodo1[0] - nodo2[0])**2 + (nodo1[1] - nodo2[1])**2)

def astar(matriz, inicio, final):
    filas, columnas = len(matriz), len(matriz[0])
    lista_abierta = []
    lista_cerrada = set()

    # Colocar los obstáculos directamente en la lista cerrada
    for i in range(filas):
        for j in range(columnas):
            if matriz[i][j] == -1:
                lista_cerrada.add((i, j))

    # Inicializar el nodo de inicio
    g_coste = {inicio: 0}
    f_coste = {inicio: distancia_euclidiana(inicio, final)}
    padres = {inicio: None}

    # Agregar el nodo inicial a la lista abierta
    heapq.heappush(lista_abierta, (f_coste[inicio], inicio))

    # Direcciones de movimiento: vertical, horizontal y diagonal
    direcciones = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1),
(1, -1), (1, 1)]

    while lista_abierta:
        _, nodo_actual = heapq.heappop(lista_abierta)

        # Si hemos llegado al objetivo, reconstruir el camino
        if nodo_actual == final:
            return reconstruir_camino(padres, nodo_actual)

        lista_cerrada.add(nodo_actual)

        # Explorar vecinos
        for direccion in direcciones:
            vecino = (nodo_actual[0] + direccion[0], nodo_actual[1] +
direccion[1])
```

```
            # Verificar que el vecino esté dentro de los límites y no sea
un obstáculo o ya esté cerrado
            if 0 <= vecino[0] < filas and 0 <= vecino[1] < columnas and
vecino not in lista_cerrada:
                g_nuevo = g_coste[nodo_actual] +
distancia_euclidiana(nodo_actual, vecino)

                # Si no está en lista_abierta o encontramos un camino más
corto, actualizamos
                if vecino not in g_coste or g_nuevo < g_coste[vecino]:
                    g_coste[vecino] = g_nuevo
                    f_coste[vecino] = g_nuevo +
distancia_euclidiana(vecino, final)
                    padres[vecino] = nodo_actual
                    heapq.heappush(lista_abierta, (f_coste[vecino],
vecino))

    # No se encontró un camino
    return None

def reconstruir_camino(padres, nodo):
    camino = []
    while nodo is not None:
        camino.append(nodo)
        nodo = padres[nodo]
    camino.reverse()
    return camino

def mostrar_matriz(matriz, camino=None):
    filas, columnas = len(matriz), len(matriz[0])
    imagen = np.zeros((filas, columnas, 3))  # Imagen en RGB

    colores = {
        0: (1, 1, 1),  # Blanco para caminos libres
        -1: (0, 0, 0), # Negro para obstáculos
        1: (0, 1, 0),  # Verde para puntos
        2: (0, 1, 0),  # Verde para puntos
        3: (0, 1, 0),  # Verde para puntos
        4: (0, 1, 0),  # Verde para puntos
        5: (0, 1, 0),  # Verde para puntos
```

```
        6: (1, 0, 0),  # Rojo para el final
    }

    # Pintar la matriz
    for i in range(filas):
        for j in range(columnas):
            imagen[i, j] = colores.get(matriz[i][j], (1, 1, 1))

    # Si hay un camino, dibujarlo en azul
    if camino:
        for nodo in camino:
            if nodo not in puntos.values():  # No sobrescribir puntos
                imagen[nodo[0], nodo[1]] = (0, 0, 1)  # Azul para el
camino

    # Mostrar la imagen con Matplotlib
    plt.figure(figsize=(5, 5))
    plt.imshow(imagen)
    plt.grid(visible=True, color="gray", linewidth=0.5)
    plt.xticks([])
    plt.yticks([])
    plt.title("Mapa con el camino encontrado")
    plt.show()

# Prueba del algoritmo
nombre_fichero = "matriz2.txt"
matriz, puntos = leer_matriz_fichero(nombre_fichero)

print("Matriz cargada:")
for fila in matriz:
    print(fila)

print(f"Puntos: {puntos}")

# Ordenar los puntos por su valor
puntos_ordenados = sorted(puntos.items())

camino_total = []
for i in range(len(puntos_ordenados) - 1):
    inicio = puntos_ordenados[i][1]
```

```
    final = puntos_ordenados[i + 1][1]
    camino = astar(matriz, inicio, final)
    if camino:
        camino_total.extend(camino[:-1])  # Evitar duplicar el nodo final
    else:
        print(f"No se encontró un camino de {inicio} a {final}")
        break

# Agregar el último nodo final
camino_total.append(puntos_ordenados[-1][1])

if camino_total:
    print("Camino encontrado:")
    for paso in camino_total:
        print(paso)
else:
    print("No se encontró un camino.")

# Mostrar la matriz con el camino encontrado
mostrar_matriz(matriz, camino_total)
```

Parte 3: A* Avanzado con Way Points y celdas relativamente peligrosas

Este apartado es muy parecido al 2, con diferencia de que hay 2 tipos de obstáculos. Algunos son obstáculos que no se pueden cruzar mientras otros obstáculos son parcialmente peligrosos, significando que se pueden cruzar pero en caso de cruzarse se debe tener en cuenta el factor de peligrosidad que se ha incluido en la toma de decisiones.

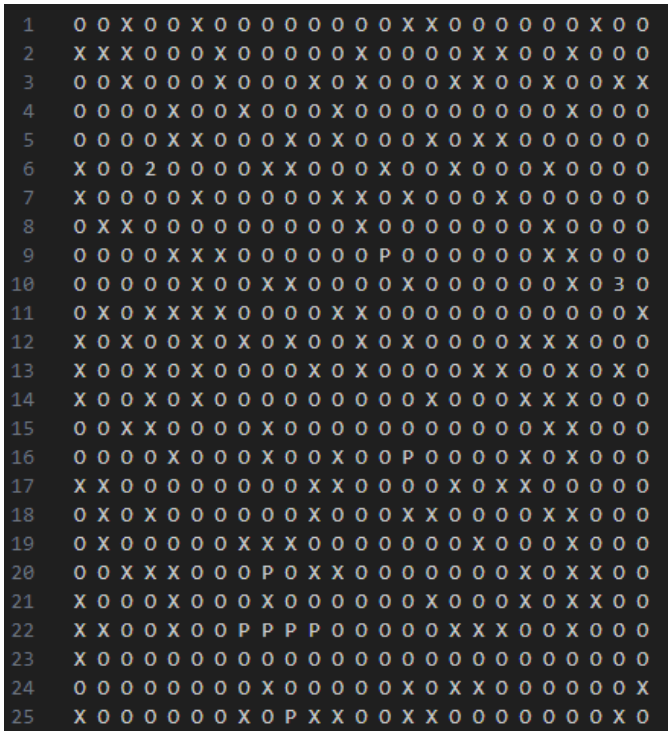
- **'O'** (Camino libre) se representa como `0`.
- **'X'** (Obstáculo) se representa como `-1`.
- **'P'** (Terreno peligroso) se representa como `0.5`, indicando que atravesarlo tiene un costo menor que un obstáculo, pero mayor que un camino libre.
- **'1', '2', ..., '6'** representan puntos clave del mapa, asignando valores únicos a cada uno.

Esta función también identifica y almacena las coordenadas de los puntos clave en un diccionario `puntos`.

Expansión de Nodos

- Se exploran las 8 direcciones posibles (horizontal, vertical y diagonal).
- Para cada vecino válido, se calcula su nuevo costo considerando el tipo de celda:
    - **Camino libre (0)** → Costo normal.
    - **Terreno peligroso (0.5)** → Incrementa el costo del trayecto.
- Si el nuevo costo es menor que el almacenado previamente, se actualiza y el nodo es añadido a la `lista_abierta`.

Ejemplo de uso:



matriz3.txt

Proceso

1. Se carga la matriz y se identifica el punto de inicio `(5, 3)` y el destino `(23,21)`, con los waypoints en este orden: `(10,24), (14,1), (20,13)`

   Se identifican obstáculos (X), caminos transitables (O) y **zonas peligrosas (P)**, que tienen un peso **de 0.5**, lo que **penaliza su tránsito**.

2. A* explora nodos, priorizando aquellos con menor costo total f(n).

   Novedad: Si una celda tiene P, su peso se multiplica por 0.5, lo que **desincentiva su uso**.

3. Genera el camino obteniendo la ruta conectando los waypoints en orden.
4. Devuelve la ruta como una lista de coordenadas:
   `(24, 9), (23, 9), (22, 9), (21, 8), (20, 7), (19, 7), (18, 6), (17, 6), (16, 6), (15, 6), (14, 5), (13, 4), (12, 4), (11, 3), (10, 2), (9, 2), (8, 2), (7, 3), (6, 3), (5, 3), (5, 4), (5, 5), (5, 6), (6, 7), (7, 8), (8, 9), (8, 10), (8, 11), (8, 12), (8, 13), (8, 14), (8, 15), (8, 16), (8, 17), (8, 18), (8, 19), (9, 20), (10, 21), (9, 22), (9, 23)`
5. Se muestra una imagen con la ruta en azul, y los caminos penalizados en amarillo

Código:

```
import heapq
import math
import matplotlib.pyplot as plt
import numpy as np

def leer_matriz_fichero(nombre_fichero):
    matriz = []
    puntos = {}

    conversion = {
        'O': 0,  # Camino libre
```

```
        'X': -1,  # Obstáculo
        'P': 0.5,  # Celda peligrosa
        '1': 1,  # Punto 1
        '2': 2,  # Punto 2
        '3': 3,  # Punto 3
        '4': 4,  # Punto 4
        '5': 5,  # Punto 5
        '6': 6,  # Punto 6
    }

    with open(nombre_fichero, 'r') as fichero:
        for i, linea in enumerate(fichero):
            fila = []
            for j, caracter in enumerate(linea.strip().split()):
                if caracter in conversion:
                    valor = conversion[caracter]
                    fila.append(valor)
                    if valor > 0:
                        puntos[valor] = (i, j)
                else:
                    raise ValueError(f"Caracter no reconocido:
{caracter}")
            matriz.append(fila)

    return matriz, puntos

def distancia_euclidiana(nodo1, nodo2):
    return math.sqrt((nodo1[0] - nodo2[0])**2 + (nodo1[1] - nodo2[1])**2)

def astar(matriz, inicio, final):
    filas, columnas = len(matriz), len(matriz[0])
    lista_abierta = []
    lista_cerrada = set()

    # Colocar los obstáculos directamente en la lista cerrada
    for i in range(filas):
        for j in range(columnas):
            if matriz[i][j] == -1:
                lista_cerrada.add((i, j))
```

```
    # Inicializar el nodo de inicio
    g_coste = {inicio: 0}
    f_coste = {inicio: distancia_euclidiana(inicio, final)}
    padres = {inicio: None}

    # Agregar el nodo inicial a la lista abierta
    heapq.heappush(lista_abierta, (f_coste[inicio], inicio))

    # Direcciones de movimiento: vertical, horizontal y diagonal
    direcciones = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1),
(1, -1), (1, 1)]

    while lista_abierta:
        _, nodo_actual = heapq.heappop(lista_abierta)

        # Si hemos llegado al objetivo, reconstruir el camino
        if nodo_actual == final:
            return reconstruir_camino(padres, nodo_actual)

        lista_cerrada.add(nodo_actual)

        # Explorar vecinos
        for direccion in direcciones:
            vecino = (nodo_actual[0] + direccion[0], nodo_actual[1] +
direccion[1])

            # Verificar que el vecino esté dentro de los límites y no sea
un obstáculo o ya esté cerrado
            if 0 <= vecino[0] < filas and 0 <= vecino[1] < columnas and
vecino not in lista_cerrada:
                riesgo = 1 if matriz[vecino[0]][vecino[1]] == 0 else
matriz[vecino[0]][vecino[1]]
                g_nuevo = g_coste[nodo_actual] +
distancia_euclidiana(nodo_actual, vecino) * riesgo

                # Si no está en lista_abierta o encontramos un camino más
corto, actualizamos
                if vecino not in g_coste or g_nuevo < g_coste[vecino]:
                    g_coste[vecino] = g_nuevo
```

```
                    f_coste[vecino] = g_nuevo +
distancia_euclidiana(vecino, final)
                    padres[vecino] = nodo_actual
                    heapq.heappush(lista_abierta, (f_coste[vecino],
vecino))

    # No se encontró un camino
    return None

def reconstruir_camino(padres, nodo):
    camino = []
    while nodo is not None:
        camino.append(nodo)
        nodo = padres[nodo]
    camino.reverse()
    return camino

def mostrar_matriz(matriz, camino=None):
    filas, columnas = len(matriz), len(matriz[0])
    imagen = np.zeros((filas, columnas, 3))  # Imagen en RGB

    colores = {
        0: (1, 1, 1),  # Blanco para caminos libres
        -1: (0, 0, 0), # Negro para obstáculos
        0.5: (1, 1, 0), # Amarillo para celdas peligrosas
        1: (0, 1, 0),  # Verde para puntos
        2: (0, 1, 0),  # Verde para puntos
        3: (0, 1, 0),  # Verde para puntos
        4: (0, 1, 0),  # Verde para puntos
        5: (0, 1, 0),  # Verde para puntos
        6: (1, 0, 0),  # Rojo para el final
    }

    # Pintar la matriz
    for i in range(filas):
        for j in range(columnas):
            imagen[i, j] = colores.get(matriz[i][j], (1, 1, 1))

    # Si hay un camino, dibujarlo en azul
    if camino:
```

```
        for nodo in camino:
            if nodo not in puntos.values():  # No sobrescribir puntos
                imagen[nodo[0], nodo[1]] = (0, 0, 1)  # Azul para el
camino

    # Mostrar la imagen con Matplotlib
    plt.figure(figsize=(5, 5))
    plt.imshow(imagen)
    plt.grid(visible=True, color="gray", linewidth=0.5)
    plt.xticks([])
    plt.yticks([])
    plt.title("Mapa con el camino encontrado")
    plt.show()

# Prueba del algoritmo
nombre_fichero = "matriz3.txt"
matriz, puntos = leer_matriz_fichero(nombre_fichero)

print("Matriz cargada:")
for fila in matriz:
    print(fila)

print(f"Puntos: {puntos}")

# Ordenar los puntos por su valor
puntos_ordenados = sorted(puntos.items())

camino_total = []
for i in range(len(puntos_ordenados) - 1):
    inicio = puntos_ordenados[i][1]
    final = puntos_ordenados[i + 1][1]
    camino = astar(matriz, inicio, final)
    if camino:
        camino_total.extend(camino[:-1])  # Evitar duplicar el nodo final
    else:
        print(f"No se encontró un camino de {inicio} a {final}")
        break

# Agregar el último nodo final
camino_total.append(puntos_ordenados[-1][1])
```

```
if camino_total:
    print("Camino encontrado:")
    for paso in camino_total:
        print(paso)
else:
    print("No se encontró un camino.")

# Mostrar la matriz con el camino encontrado
mostrar_matriz(matriz, camino_total)
```