



FACULTAD DE CIENCIAS

INTELIGENCIA ARTIFICIAL

Algoritmo A^*

Equipo: Skynet Scribes

Número de practica: 04

Sarah Sophía Olivares García
318360638

Marco Silva Huerta
316205326

Carlos Daniel Cortés Jiménez
420004846

Laura Itzel Tinoco Miguel
316020189

Luis Enrique García Gómez
315063880

Fernando Mendoza Eslava
319097690

Profesora: Cecilia Reyes Peña

Ayudante teoría: Karem Ramos Calpulalpan

Ayudante laboratorio: Tania Michelle Rubí Rojas

Semestre 2024-2

Fecha de entrega:
13 de Marzo del 2024

1. Algoritmo A^*

Introducción

El algoritmo A^* , concebido en 1968 por Peter Hart, Nils Nilsson, y Bertram Raphael, se erige como un pilar en la búsqueda de caminos dentro del vasto dominio de la inteligencia artificial. Este algoritmo trasciende la simpleza de métodos tradicionales, como la Búsqueda en Anchura (BFS) y la Búsqueda en Profundidad (DFS), mediante la incorporación de una función heurística. Esta heurística orienta la exploración hacia el objetivo de manera eficiente, optimizando el trayecto en términos de coste y distancia. Su versatilidad le permite adaptarse a una amplia gama de aplicaciones, desde la conducción autónoma y la robótica hasta la generación de rutas en videojuegos y aplicaciones de mapeo geográfico.

¿Cómo funciona?

El propósito del algoritmo A^* es encontrar el camino más corto desde un punto de inicio hasta un punto objetivo en un mapa con diferentes caminos y obstáculos con la aplicación de una estrategia de búsqueda informada, optimizando el recorrido basado en el costo y una estimación heurística.

Aquí hay algunos pasos clave que sigue el algoritmo:

- **Evaluación de funciones:** El algoritmo evalúa tres funciones principales. La primera función, $g(n)$, calcula el costo real desde el punto de inicio hasta el punto actual en el que nos encontramos. La segunda función, $h(n)$, es una estimación del costo más bajo desde el punto actual hasta el objetivo. Y la tercera función, $f(n) = g(n) + h(n)$, representa el costo total estimado de la ruta de menor costo que pasa por el punto actual.
- **Listas abierta y cerrada:** El algoritmo mantiene dos listas. La lista abierta contiene los nodos que aún no hemos explorado, mientras que la lista cerrada contiene los nodos que ya hemos evaluado. La función heurística $h(n)$ que estima el costo más bajo desde n hasta el objetivo, y la función $f(n) = g(n) + h(n)$ que representa el costo total estimado de la ruta más barata pasando por n .
- **Selección del siguiente nodo:** En cada paso, el algoritmo selecciona el nodo de la lista abierta con el valor más bajo de $f(n)$. Es como si estuviéramos eligiendo el siguiente paso más prometedor en el laberinto.
- **Llegar al objetivo:** Si el nodo seleccionado es el objetivo, ¡lo encontramos! El algoritmo termina y podemos reconstruir el camino óptimo desde el objetivo hasta el punto de inicio.
- **Explorar vecinos:** Si el nodo seleccionado no es el objetivo, lo movemos de la lista abierta a la lista cerrada y examinamos todos sus vecinos. Para cada vecino, el algoritmo calcula $g(n)$ y $f(n)$. Si el vecino ya está en la lista abierta con un costo más alto, se actualiza con el nuevo valor más bajo. Si el vecino no está en ninguna lista o está en la lista cerrada con un costo más alto, se agrega o se mueve a la lista abierta con el nuevo costo y se guarda un puntero al nodo actual como su padre.

Este proceso se repite hasta que encontremos el objetivo o hasta que la lista abierta esté vacía, lo que significa que no hay camino disponible.

La clave para que el algoritmo A^* funcione de manera eficiente es la función heurística $h(n)$. Para que sea óptima, $h(n)$ no debe sobreestimar el costo real para llegar al objetivo. Hay diferentes formas de calcular $h(n)$, como la distancia Manhattan, que se utiliza en mapas cuadrículados donde solo se pueden mover en línea recta, o la distancia euclidiana, que se prefiere cuando se pueden hacer movimientos diagonales.

En resumen, el algoritmo A^* es una herramienta muy útil para encontrar el camino más corto en un mapa complicado. Nos ayuda a tomar decisiones inteligentes y a ahorrar tiempo y esfuerzo al encontrar la mejor ruta posible.

Algoritmo A^* vs BFS

A diferencia de la Búsqueda en Anchura (BFS), que adopta un enfoque no ponderado y equitativo en su exploración expandiendo todos los nodos vecinos con igual prioridad, el algoritmo A^* introduce una estrategia de búsqueda ponderada. Esta estrategia se centra en minimizar una función de coste $f(n) = g(n) + h(n)$, donde $g(n)$ representa el coste exacto desde el nodo inicial hasta el nodo n , y $h(n)$ es la heurística que estima el coste mínimo desde n hasta el objetivo. Esta dualidad permite a A^* explorar de forma selectiva aquellos caminos que parecen más prometedores, reduciendo significativamente el volumen de cálculo y garantizando la identificación del trayecto óptimo.

Ventajas

- **Optimalidad y Complejidad:** A^* garantiza encontrar la ruta más corta hacia el objetivo, siempre que la heurística empleada sea admisible y consistente. Esta característica lo distingue como un algoritmo de búsqueda óptima.
- **Eficiencia:** Al priorizar nodos basándose en el coste total estimado $f(n)$, A^* es capaz de descartar rutas menos prometedoras de manera precoz, agilizando la consecución de su meta.
- **Flexibilidad Heurística:** La posibilidad de adaptar la heurística $h(n)$ según las particularidades del problema permite una optimización específica del rendimiento de búsqueda.

Desventajas

- **Dependencia Heurística:** La eficacia del algoritmo está intrínsecamente ligada a la calidad de la heurística $h(n)$. Una heurística pobre puede resultar en una exploración ineficiente y un mayor consumo de recursos.
- **Requerimientos de Memoria:** El mantenimiento de las listas abierta y cerrada, especialmente en espacios de búsqueda vastos, puede conducir a un elevado consumo de memoria, superando en ocasiones a alternativas más simples como BFS.

2. Distancia Manhattan

La *distancia Manhattan* mide la distancia entre dos puntos en un espacio euclidiano multidimensional, y esta basada en líneas paralelas a los ejes de coordenadas en lugar de una línea recta, como puede ser un plano cartesiano, se emplea en inteligencia artificial y en diversos campos que incluyen la teoría de grafos, geometría computacional y optimización.

A esta distancia se le llama *Manhattan* porque en una cuadrícula de calles de ciudades como Manhattan, para ir de un punto a otro hay que moverse siguiendo las manzanas delimitadas por las calles horizontales y verticales en ángulos rectos.

En el ámbito de la inteligencia artificial, se usa con frecuencia en algoritmos heurísticos como A^* para resolver problemas de búsqueda de caminos, su uso principal es en situaciones donde el desplazamiento entre dos ubicaciones se limita a movimientos verticales u horizontales, como sucede en la planificación de rutas.

Se puede usar en una app de mapas para hallar la ruta más corta desde donde estás hasta un punto específico en una ciudad con calles que se cruzan formando cuadrados. La *distancia Manhattan* puede servir como una guía para calcular la distancia que queda hasta llegar a tu destino, por ejemplo imagina que estás manejando en tu carro y tienes que ir desde un cruce A hasta otro cruce B en una ciudad con calles dispuestas en forma de cuadrícula. ¿Puedes determinar la *distancia de Manhattan* entre A y B sumando el número de cuadrados que necesitas recorrer en dirección horizontal y vertical para llegar de A a B , sin considerar los obstáculos como edificios o restricciones de tráfico?

La distancia Manhattan proporciona una estimación precisa del recorrido necesario en automóvil, ya que las rutas siguen generalmente un patrón de calles y avenidas en ángulos rectos. A pesar de que en la práctica puedan existir variaciones debido a las limitaciones del tráfico o la disposición exacta de las calles, aún es una buena aproximación y puede ayudar a los sistemas de mapas para calcular rutas eficientes y ayuda a los conductores a tomar decisiones informadas sobre sus rutas.

La fórmula para calcular la distancia de Manhattan entre dos puntos $A(x_1, y_1)$ y $B(x_2, y_2)$, en un espacio bidimensional es:

$$D = |x_1 - x_2| + |y_1 - y_2|$$

Donde $|x_1 - x_2|$ representa la diferencia en la coordenada x entre los dos puntos y $|y_1 - y_2|$ representa la diferencia en la coordenada y entre los dos puntos.

En pocas palabras la elección de la distancia Manhattan como heurística en la implementación de A^* se fundamenta en su capacidad para estimar costes de manera coherente y admisible en entornos cuadrículados, donde los movimientos están restringidos a las direcciones cardinales. Esta simplicidad y eficacia la convierten en una heurística ideal para muchos escenarios de búsqueda en laberintos y grillas.

Pseudocódigo

```
función distancia_manhattan(nodo_actual, nodo_objetivo):  
    diferencia_x = abs(nodo_actual.x - nodo_objetivo.x)  
    diferencia_y = abs(nodo_actual.y - nodo_objetivo.y)  
    return diferencia_x + diferencia_y
```

3. Otras heurísticas

Otros dos ejemplos de heurísticas que se pueden usar en el algoritmo A^* son:

- **Distancia Euclidiana:** Se utiliza como métrica para calcular la separación entre puntos en un espacio euclidiano, ya sea en el plano o en el espacio tridimensional. Se utiliza el teorema de Pitágoras para calcular la longitud del segmento de línea recta que une los dos puntos. La distancia euclidiana es una heurística admisible, no obstante, puede no ser uniforme en todos los casos, especialmente en ambientes con obstáculos donde la ruta más directa puede no ser una línea recta.

La fórmula para la distancia euclidiana entre dos puntos $A(x_1, y_1)$ y $B(x_2, y_2)$, en un espacio bidimensional es:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

■ Distancia de Chebyshev:

- La distancia de Chebyshev se usa como métrica para calcular la distancia entre dos puntos en un espacio de rejilla, donde el movimiento está permitido en todas las direcciones (horizontal, vertical y diagonal).
- La distancia de Chebyshev en entornos de rejilla es una heurística que cumple con la propiedad de consistencia, lo que la hace admisible.
- Estas reglas proporcionan estimaciones del costo restante para alcanzar el objetivo desde cualquier nodo dado en un grafo de búsqueda, lo que asiste al algoritmo A^* a dirigir su búsqueda hacia el objetivo de forma más eficaz. Depende del problema específico y de las características del espacio de búsqueda la elección de la heurística.

La fórmula para la distancia euclidiana entre dos puntos $A(x_1, y_1)$ y $B(x_2, y_2)$, en un espacio bidimensional se calcula como la máxima de las diferencias absolutas en las coordenadas x y y

$$D = \max(|x_2 - x_1|, |y_2 - y_1|)$$

4. Documentación

Pseudocódigo A^*

Es importante mencionar que el algoritmo que la ayudante Tania presentó en su clase grabada fue el que utilizamos para implementar el algoritmo.

Entrada: Posición inicial, posición objetivo

Salida: Camino desde la posición inicial hasta la posición objetivo

Variables:

`lista_cerrada`: Lista de nodos ya examinados
`lista_abierta`: Lista de nodos por examinar
`nodo_actual`: Nodo actual que se está evaluando
`vecino`: Cada uno de los vecinos del nodo actual

Algoritmo:

1. Inicializar:

- `lista_cerrada` vacía
- Agregar nodo inicial a `lista_abierta`

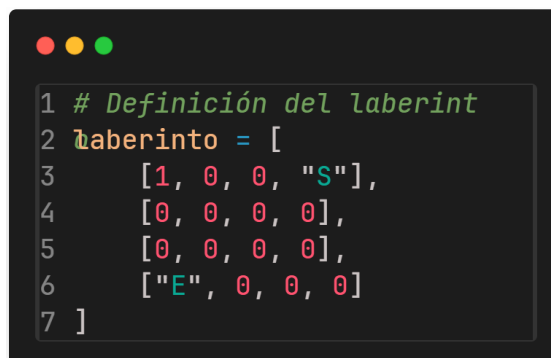
2. **Mientras** `lista_abierta` no esté vacía:

- a) **Obtener** `nodo_actual` con el menor valor de f de `lista_abierta`
- b) **Eliminar** `nodo_actual` de `lista_abierta`
- c) **Agregar** `nodo_actual` a `lista_cerrada`
- d) **Obtener vecinos** de `nodo_actual`
- e) **Para cada vecino:**
 - Si vecino es la posición objetivo:
 - **Retornar camino** desde `nodo_actual`
 - **Calcular** g del vecino
 - **Calcular** h del vecino
 - **Calcular** f del vecino
 - Si vecino ya está en `lista_abierta` y f del vecino en `lista_abierta` es mayor:
 - **Actualizar** f del vecino en `lista_abierta`
 - **Actualizar** padre del vecino en `lista_abierta`
 - Si vecino ya está en `lista_cerrada` y f del vecino en `lista_cerrada` es mayor:
 - **Eliminar** vecino de `lista_cerrada`
 - **Agregar** vecino a `lista_abierta`
 - **Actualizar** padre del vecino
 - Si vecino no está en ninguna lista:
 - **Agregar** vecino a `lista_abierta`
 - **Actualizar** padre del vecino

Fin del algoritmo

5. Resultados obtenidos

En este apartado, observaremos el comportamiento de diferentes técnicas empleadas por un agente para encontrar el camino en un laberinto representado mediante una matriz. En esta representación, cada celda de la matriz indica el estado de una posición en el laberinto, donde 0 representa un camino libre, 1 es un muro, E denota la entrada y S representa la salida. Además, el agente intenta encontrar una salida siguiendo un orden predefinido: arriba, abajo, izquierda, derecha.



```
1 # Definición del laberint
2 laberinto = [
3     [1, 0, 0, "S"],
4     [0, 0, 0, 0],
5     [0, 0, 0, 0],
6     ["E", 0, 0, 0]
7 ]
```

Figura 1: Ejemplar para el agente.

Ejemplos usados

Primer ejemplo

Se usará un laberinto de 7×7 con la característica de un solo camino en forma de espiral, en una esquina se encuentra la entrada y en el centro de la matriz se encuentra la salida. Esto con el fin de dar un ejemplar sencillo para el agente, donde se pueda observar el comportamiento a partir del algoritmo A^* .

```
# Representación del laberinto
laberinto_espiral = [
    ["E", 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 1, 1, 0],
    [0, 1, 1, 1, 0, 1, 0, 0],
    [0, 1, "S", 0, 0, 1, 1, 0],
    [0, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1]
]

# Posición inicial del agente
agente = Agente([0, 0])
```

Figura 2: Ejemplar para el agente con un camino en espiral.

En la siguiente imagen se muestra el resultado obtenido por el agente, donde se puede observar que el agente encuentra el camino más corto para llegar a la salida y las celdas que recorre para llegar a su destino. Con **X** se marcan las celdas que recorre el agente para llegar a la salida, con los **cuadrados** se marcan los muros y con los **guion** se marcan las celdas que no recorre el agente.

```
[5, 7]
[5, 6]
[6, 6]
[6, 5]
[6, 4]
[6, 3]
[6, 2]
[6, 1]
[6, 0]
[5, 0]
[4, 0]
[3, 0]
[2, 0]
[2, 1]
[2, 2]
[2, 3]
[2, 4]
[3, 4]
[4, 4]
[4, 3]
[4, 2]
```

Se encontró la salida :D

X	X	X	X	X	X	X	
						X	X
	X	X	X	X			X
X				X	-		X
X		X	X	X			X
X						X	X
X	X	X	X	X	X	X	

Figura 3: Ejemplar para el agente con un camino en espiral.

Segundo ejemplo

Se usara un laberinto de 7x7 con la característica que sea simétrico, es decir, que la entrada y la salida se encuentran en la misma columna y con muros simétricos.

```

1 # Representación del laberinto
2 laberinto = [
3     [0, 0, 0, "E", 0, 0, 0],
4     [0, 1, 1, 0, 1, 1, 0],
5     [0, 0, 0, 0, 0, 0, 0],
6     [0, 1, 1, 1, 1, 1, 0],
7     [0, 0, 0, 0, 0, 0, 0],
8     [0, 1, 1, 0, 1, 1, 0],
9     [0, 0, 0, "S", 0, 0, 0]
10 ]

```

Figura 4: Ejemplar para el agente, con simetría.

Notemos que es posible que el agente encuentre múltiples soluciones viables y evalúe diferentes caminos hacia la salida. Sin embargo, el algoritmo A^* no tiene una lógica para elegir entre caminos equivalentes cuando varios caminos tienen la misma longitud. En tales casos, la selección del camino puede depender de la lógica de prioridad implementada en el algoritmo.

```

Ruta seguida:
[0, 3]
[1, 3]
[2, 3]
[2, 2]
[2, 1]
[2, 0]
[3, 0]
[4, 0]
[5, 0]
[6, 0]
[6, 1]
[6, 2]
[6, 3]
Se encontró la salida :D
- - - X - - -
-  1  1  X  1  1 -
X X X X - - -
X  1  1  1  1  1 -
X - - - - -
X  1  1 -  1  1 -
X X X X - - -

```

Figura 5: Solución camino por la izquierda

```

Ruta seguida:
[0, 3]
[1, 3]
[2, 3]
[2, 4]
[2, 5]
[2, 6]
[3, 6]
[4, 6]
[5, 6]
[6, 6]
[6, 5]
[6, 4]
[6, 3]
Se encontró la salida :D
- - - X - - -
-  1  1  X  1  1 -
- - - X X X X
-  1  1  1  1  1 X
- - - - - X
-  1  1 -  1  1 X
- - - X X X X

```

Figura 6: Solución camino por la derecha

El agente encuentra el camino más corto en ambos casos, pero es el orden de la heurística la que establece la prioridad de los caminos. En esta parte del código, se establece que el agente siempre buscara primero arriba, abajo, izquierda y derecha, por lo que asiendo el cambio colocando derecha antes que izquierda, el agente encontrara el camino más corto por la derecha.


```

1 devuelve:
2     list: Una lista de objetos Agente representando los vecinos válidos.
3     """
4     vecinos = []
5     if casilla_actual is not None:
6         for direccion in ["arriba", "abajo", "izquierda", "derecha"]:
7             nueva_posicion = casilla_actual.mover(direccion, laberinto)

```

Figura 7: Código para cambiar la prioridad del agente en la búsqueda de caminos.

Tercer ejemplo

Consideremos un laberinto de 7x7 con la característica de que la salida del laberinto no se encuentra accesible debido a que se encuentra rodeada de muros, es decir, no existe un camino para llegar a la salida.

```

# Representación del laberinto
laberinto = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 0],
    ["E", 0, 1, "S", 0, 1, "S"],
    [0, 0, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]

# Posición inicial del agente
agente = Agente([3, 0])
buscar_camino_A_estrella(agente, laberinto)

```

Figura 8: Ejemplar para el agente, con salida inaccesible.

En este caso, el agente no encontrara un camino para llegar a la salida, por lo que el agente no encontrara una solución.

```

Camino seguido hasta el momento:
- - - - -
X X X X X X X
X -   |   |   X
X -   | S -   X
- -   |   |   -
- - - - -
- - - - -
No se encontró un camino válido.

```

Figura 9: Momento en el que el agente no encuentra una solución.

Con fines académicos (esta visualización no refleja la implementación, solo es ilustrativo de la pagina <https://qiao.github.io/PathFinding.js/visual/>), la siguiente imagen muestra el comportamiento del agente con el algoritmo A^* en el laberinto, desde el inicio de la ejecución, el agente descarta a los vecinos de arriba y abajo por ser casillas más lejanas a la salida y se dirige a la derecha ya que es casilla que se encuentra más cerca de la salida.

Posteriormente sigue buscando casillas vecinas más a la derecha, hasta que encuentra muros, por lo que el agente opta por ir rodeando el muro hasta encontrar la salida. Cuando finalmente encuentra una posible entrada al muro que rodea la salida, se da cuenta que no es posible llegar a la salida, por lo que el agente termina la ejecución.

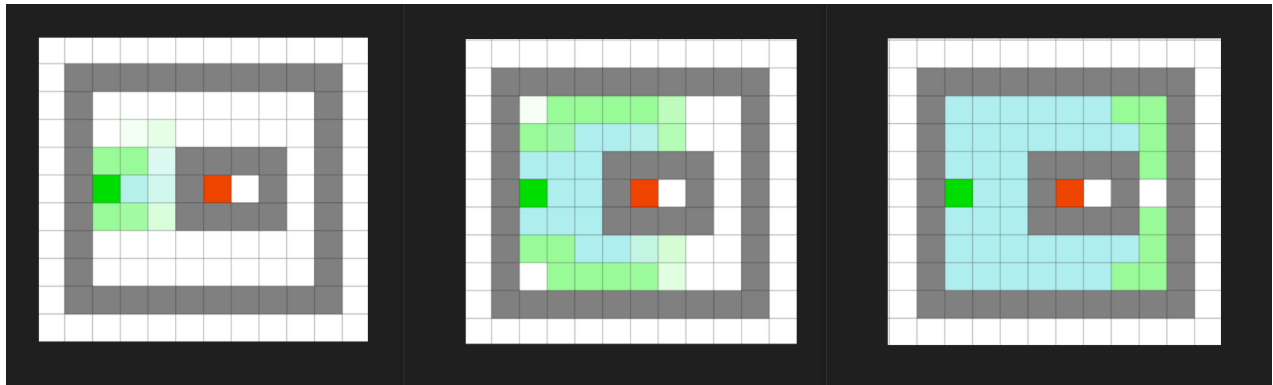


Figura 10: Ejemplo ilustrativo de Algoritmo A^* en el laberinto, con salida inaccesible.

Versiones del código

En seguida haremos notar los cambios que se hicieron entre dos versiones de implementación y el por que de estos cambios.

```

1  def reconstruir_camino(casilla_actual):
2      """
3      Esta funcion reconstruye el camino desde la casilla actual hasta la posicion inicial
4      del agente.
5
6      recibe:
7          casilla_actual (Agente): La casilla actual.
8
9      devuelve:
10         list: Una lista que representa el camino desde la casilla actual hasta la
11         posicion inicial del agente.
12     """
13     camino = []
14     while casilla_actual:
15         camino.append(casilla_actual.posicion)
16         casilla_actual = casilla_actual.padre
17
18     return list(reversed(camino))

```

Listing 1: Version 1: reconstruircamino

```

1  def reconstruir_camino(casilla_actual):
2      """

```

```

3      Esta funcion reconstruye el camino desde la casilla actual hasta la posicion inicial
      del agente.
4
5      recibe:
6          casilla_actual (Agente): La casilla actual.
7
8      devuelve:
9          list: Una lista que representa el camino desde la casilla actual hasta la
      posicion inicial del agente.
10     """
11     camino = []
12     while casilla_actual:
13         camino.append(casilla_actual.posicion)
14         casilla_actual = casilla_actual.padre
15
16     camino = list(reversed(camino))
17     print("Ruta seguida:")
18     for posicion in camino:
19         print(f"[{posicion[0]}, {posicion[1]}]")
20     print("Se encontro la salida :D")
21
22     print("\nCamino encontrado:")
23     for i in range(len(laberinto)):
24         fila = ""
25         for j in range(len(laberinto[0])):
26             if [i, j] in camino:
27                 fila += "X "
28             else:
29                 fila += "- "
30         print(fila)
31
32     return camino

```

Listing 2: Version 2: reconstruircamino

Notemos que en la version 2 el cambio es mas que nada la salida por pantalla, es decir la ruta que el agente tomo para encontrar la salida, y además mostramos el camino seguido por el agente, aquí iteramos sobre el laberinto y determinamos la posición del agente, si se encuentra dicha posición colocamos una (X), en caso contrario solo colocamos una guion (-). Este cambio lo consideramos importante ya que como bien se dijo anteriormente hace que se mas visible, clara y permite que se pueda entender mejor como se llego a la salida.

```

1      def distancia_manhattan(nodo_actual, nodo_objetivo):
2          """
3          Calcula la distancia de Manhattan entre dos nodos.
4
5          recibe:
6              nodo_actual (Agente): El nodo actual.
7              nodo_objetivo (Agente): El nodo objetivo.
8
9          devuelve:
10             int: La distancia de Manhattan entre los dos nodos.
11         """
12         diferencia_x = nodo_actual.posicion[0] - nodo_objetivo.posicion[0]
13         diferencia_y = nodo_actual.posicion[1] - nodo_objetivo.posicion[1]
14         distancia = abs(diferencia_x) + abs(diferencia_y)
15         return distancia

```

Listing 3: Version 1: distanciamanhattan

```
1 def distancia_manhattan(laberinto):
2     """
3     Calcula la distancia de Manhattan entre dos nodos.
4
5     recibe:
6         laberinto (list): El laberinto representado como una matriz
7
8     devuelve:
9         int: La distancia de Manhattan entre los dos nodos.
10    """
11    nodo_actual = punto_de_partida(laberinto) #Buscamos la posicion inicial del agente
12    nodo_objetivo = salida_laberinto(laberinto) #Buscamos la salida del laberinto
13
14    x, y = nodo_actual
15    w, z = nodo_objetivo
16
17    distancia = abs(x - w) + abs(y - z) #Calculamos la distancia Manhattan dados dos
18    puntos
19    return distancia
```

Listing 4: Version 2: distanciamanhattan

Este cambio también es importante ya que notemos primero en la version 1 dependemos mucho de la posición actual del agente(**nodo actual**) y la posición de la salida(**nodo objetivo**) como parámetros en cambio en la version 2, al eliminar esto hacemos uso de dos funciones **punto de partida** con esta obtenemos las coordenadas de la posición inicial del agente y **salida laberinto** con esta obtenemos las coordenadas de la salida del laberinto (si la hay), y notemos que al hacer este cambio hacemos que la funcion de distancia Manhattan sea más versátil, es decir, podemos usarla en diferentes laberintos, pasando como parámetro algún laberinto, y así no modificar la funcion, con esto también añadimos que mejora la modularidad del código, haciendo que el código pueda ser usado en un futuro, y sea mas facil de comprender.

Las siguiente funciones son métodos auxiliares que nos ayudan a encontrar la posición inicial del agente y la salida del laberinto respectivamente.

Se realizo este añadido con la finalidad de que sea más fácil de entender y comprender como estamos buscando las coordenadas de la entrada y salida del laberinto, además de hacer que nuestro código sea mas flexible

```
1 def punto_de_partida(laberinto):
2     """
3     Encuentra la posicion inicial del agente en el laberinto
4
5     Recibe:
6         laberinto (list): El laberinto representado como una matriz.
7
8     Devuelve:
9         tupla: La posicion(coordenadas) inicial del agente
10    """
11    try:
12        for i in range(len(laberinto)):
13            for j in range(len(laberinto[0])):
14                if laberinto[i][j] == "E": # Comprobamos si encontramos la entrada
15                    return (i, j) # Devolvemos las coordenadas
16    except ValueError:
17        print("No se encontro la salida del laberinto.")
```

Listing 5: Función auxiliar: puntodepartida

```
1 def salida_labirinto(laberinto):
2     """
3     Encuentra la posición de la salida del laberinto
4
5     Recibe:
6         laberinto (list): El laberinto representado como una matriz.
7
8     Devuelve:
9         tupla: La posición(coordenadas) de la salida del laberinto.
10    """
11    try:
12        for i in range(len(laberinto)):
13            for j in range(len(laberinto[0])):
14                if laberinto[i][j] == "S": # Comprobamos si encontramos la salida
15                    return (i, j) # Devolvemos las coordenadas
16    except ValueError:
17        print("No se encontro la salida del laberinto.")
```

Listing 6: Función auxiliar: salidalaberinto

Otros pequeños cambios que se hicieron fue que en lugar de usar una condición para ejecutar el algoritmo, eso se hizo simplemente por comodidad y que sea más claro.

```
1 if camino:
2     print("Camino encontrado:", camino)
3 else:
4     print("No se encontro un camino valido.")
```

Listing 7: Versión 1: ejecución algoritmo

```
1 agente = Agente([0, 0])
2 buscar_camino_A_estrella(agente, laberinto)
```

Listing 8: Versión 2: ejecución algoritmo

Referencias

- [HNR68] Peter E Hart, Nils J Nilsson y Bertram Raphael. «A formal basis for the heuristic determination of minimum cost paths». En: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), págs. 100-107. URL: <https://ieeexplore.ieee.org/document/4082128> (visitado 05-03-2024).
- [RN16] Stuart Russell y Peter Norvig. *Inteligencia Artificial Un Enfoque moderno*. 2nd. Pearson Prentice Hall, 2016.
- [Ecu] EcuRed. *Algoritmo de Búsqueda Heurística A**. URL: https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A* (visitado 05-03-2024).
- [Gee] GeeksforGeeks. *A* Search Algorithm*. URL: <https://www.geeksforgeeks.org/a-search-algorithm/> (visitado 05-03-2024).
- [jar] jariasf. *ALGORITMO DE BÚSQUEDA: BREADTH FIRST SEARCH*. URL: <https://jariasf.wordpress.com/2012/02/27/algoritmo-de-busqueda-breadth-first-search/>.
- [UNAA] TECNOLOGIA EDUCATIVA UNAN. *Algoritmo A estrellas*. URL: <https://aeia.home.blog/algoritmo-a-estrellas-a/> (visitado 11-03-2024).
- [UNAb] TECNOLOGIA EDUCATIVA UNAN. *Heuristic Search Strategies*. URL: https://www.cs.unm.edu/~luger/ai-final2/4-BeyondSearch/4-Beyond_search.html (visitado 05-03-2024).