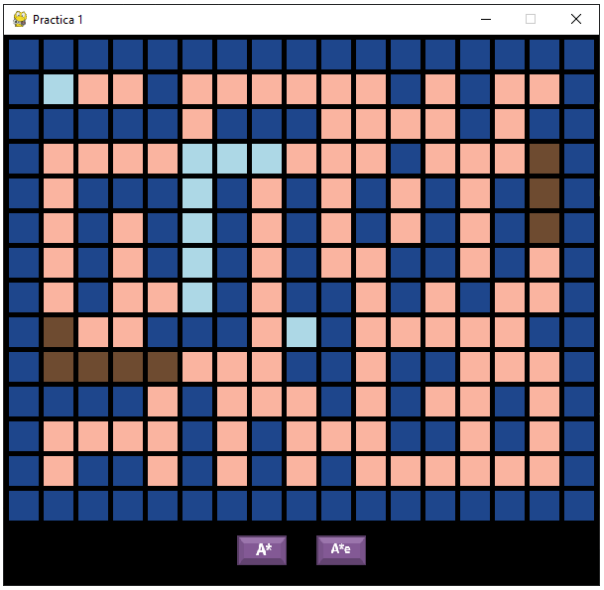
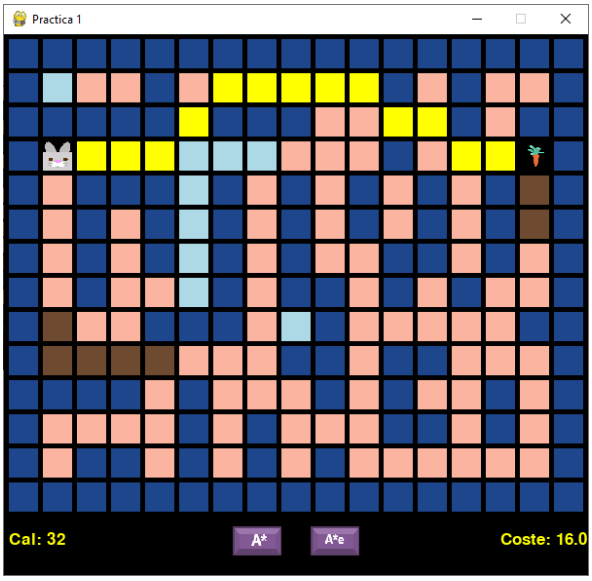
Universitat d'Alacant
Universidad de Alicante

Sistemas Inteligentes

Práctica 1: Búsqueda heurística





Daniel Jenaro Fernández Juan 53979820Q

Tabla de contenido

Introducción

En esta práctica se estudiará y se pondrá en práctica el funcionamiento de la búsqueda heurística, en concreto se utilizarán los algoritmos A* y A^*_ε. La búsqueda heurística es un enfoque de la inteligencia artificial que tiene como objetivo resolver problemas de búsqueda y optimización de manera eficiente. Esto se lleva a cabo mediante la implementación de ciertas reglas (heurísticas) que ayudan a seleccionar, de todas las opciones disponibles, las más prometedoras para alcanzar su objetivo en el menor tiempo posible y utilizando el menor número de recursos posibles.

Uno de los enfoques más comunes es la representación del espacio de estados, esto quiere decir que se crea un espacio en el que cada estado representa una configuración posible del problema, y los movimientos permiten la transición entre estos estados.

A este tipo de algoritmos se les llama algoritmos informados ya que disponen de ciertos datos sobre la proximidad de cada estado a un estado objetivo. Esta circunstancia les permite explorar en primer lugar los caminos más prometedores, y así, determinar qué estado se encuentra más cerca de la solución.

Conceptos básicos:

- Completitud: Capacidad de un algoritmo para garantizar que encontrará una solución si esta existe en el espacio de búsqueda.
- Admisibilidad: Propiedad de la heurística utilizada en el algoritmo. Una heurística es considerada admisible si nunca sobreestima el costo real de conseguir el objetivo desde un estado dado. $(h(n) \leq h^*(n))$
- Dominación: Un algoritmo A^1 es dominante sobre A^2 si cada nodo expandido por A^1es también expandido por A^2.
- Optimalidad: Un algoritmo es el óptimo de un conjunto de algoritmos si es el dominante sobre todos los algoritmos del conjunto (es el que menos nodos expande).

El algoritmo A* tiene como fórmula general $f^*(n) = g^*(n) + h^*(n)$, donde:

- $g^*(n) = c(s,n)$: Coste del camino de coste mínimo desde el nodo inicial s al nodo n. Estimada por $g(n)$.
- $h^*(n)$: Coste del camino de coste mínimo de todos los caminos desde el nodo n a cualquier estado solución. Estimada por $h(n)$.
- $f^*(n)$: Coste del camino de coste mínimo desde el nodo inicial hasta un nodo solución condicionado a pasar por n. Estimada por $f(n)$.
- C^*: Coste del camino de coste mínimo desde el nodo inicial a un nodo solución.

Algoritmo A*

La nomenclatura que se utilizará a partir de ahora para referirnos a ciertos componentes del algoritmo serán los siguientes:

- Estados frontera: Los estados frontera son los estados alcanzados que no han sido seleccionados para su expansión.
- Estado expandido: Un estado que ha sido seleccionado y ha generado todos los estados descendientes (alcanzados por el estado expandido).
- Lista frontera (LF): Estados alcanzados no seleccionados.
- Lista interior (LI): Estados seleccionados y expandidos.
- Estado abierto: Estados seleccionados (LF).
- Estado cerrado: Estados ya explorados (LI).

Explicación del pseudocódigo

Algoritmo Algoritmo A*

```
1:  listaInterior ← ∅
2:  listaFrontera ← inicio
3:  while listaFrontera ≠ ∅ do
4:      n ← obtener nodo de listaFrontera con menor f(n) = g(n) + h(n)
5:      listaFrontera.del(n)
6:      listaInterior.add(n)
7:      if n es meta then
8:          devolver reconstruir camino desde la meta al inicio (punteros)
9:      for all hijo m de n que no esté en listaInterior do
10:         g'(m) ← n.g + c(n, m)
11:         if m no está en listaFrontera then
12:             almacenar la f, g y h del nodo en (m.f, m.g, m.h)
13:             m.padre ← n
14:             listaFrontera.add(m)
15:         else if g'(m) es mejor que m.g then
16:             m.padre ← n
17:             recalcular f y g del nodo m
18: devolver no hay solución
```

Lo primero que hace el algoritmo A* es definir 2 listas, la lista frontera y la lista interior. En la lista frontera se añadirá el estado desde donde se inicia el algoritmo y se irán añadiendo los estados alcanzados que todavía no se han seleccionado para explorar, en la lista interior se irán almacenando los estados que ya hemos seleccionado y expandido.

Mientras haya estados en la lista frontera, es decir, mientras no se hayan explorado todos los estados alcanzables, realizaremos lo siguiente. Se extraerá de la lista frontera el estado con menor f(n), el cual añadiremos a la lista interior porque ya ha sido seleccionado y se borrará de la lista frontera. Se comprueba si el estado actual es igual al estado objetivo, si es así, se devuelve el camino de estados que se han seleccionado hasta llegar al estado objetivo.

Se calcula los estados alcanzables desde el estado actual, y para cada estado alcanzable, si no esta en la lista frontera, se añadirá y se calculará sus f, g y h. Si esta en la lista

frontera se calculará si es mejor llegar al estado desde el nodo estado o desde el estado padre, si es desde el estado actual se actualizarán sus f y g.

Si una vez explorados todos los estados posibles no se encuentra el estado objetivo, devolverá que no existe una solución.

Implementación A*

En esta práctica se pide que apliquemos el algoritmo A* para encontrar una solución óptima para el problema del camino mínimo entre un conejo y una zanahoria. Para realizar esta tarea se deberá implementar el pseudocódigo del apartado anterior en el lenguaje Python, se estudiarán varias heurísticas teniendo en cuenta el número de nodos explorados y si son admisibles o no. Para esta implementación, se han elegido 4 heurísticas diferentes: Euclídea, Manhattan, Octal y con heurística 0.

La distancia euclídea viene a ser, por lo general, la longitud de la hipotenusa del triángulo recto conformado por cada punto y los vectores proyectados sobre los ejes directores al nivel de la hipotenusa. En el plano cartesiano, siendo $A = (X_A; X_B)$; $B = (Y_A; Y_B)$ se puede definir la distancia euclídea como $d(A, B) = \sqrt{((X_A - Y_A)^2 + (X_B - Y_B)^2)}$. De manera que la distancia euclídea entre dos puntos puede definirse como la longitud del segmento de recta que une a dichos puntos. En Python se calcularía de la siguiente manera:

```
#Calculamos la h con euclídea
self.h = math.sqrt(pow((desColumna - posColumna),2) + pow((desFila - posFila),2))
self.f = self.g + self.h
```

La distancia calculada con el método Manhattan es una distancia entre dos puntos de una trayectoria en forma de cuadrícula. A diferencia de la distancia euclídea, que mide la línea más corta posible entre dos puntos, la distancia de Manhattan calcula la suma de las diferencias absolutas entre las coordenadas de dos puntos. La fórmula de esta distancia es la siguiente; $d(A, B) = |X_A - Y_A| + |X_B - Y_B|$. La distancia Manhattan incorpora la función del valor absoluto, que convierte cualquier diferencia negativa en un valor positivo. Esto es muy importante ya que garantiza que todas las mediciones de distancia sean no negativas, reflejando la verdadera distancia escalar independientemente de la dirección de desplazamiento. En Python se calcularía de la siguiente manera:

```
#Calculamos la h con Manhattan
self.h =abs(desColumna - posColumna) + abs(desFila - posFila)
self.f = self.g + self.h
```

La distancia octal estima el coste entre dos puntos considerando también el movimiento en diagonal, es decir, en 8 direcciones. En este caso, los costes en movimientos horizontales y verticales se evalúan de la misma forma que en la distancia Manhattan. Sin embargo, los movimientos diagonales tienen un valor de $\sqrt{2}$. La distancia octal entre dos puntos se calcula de la siguiente manera:

$dOctil(A, B) = c_{diagonal} * \min(\Delta x, \Delta y) + c_{Manhattan} * |\Delta x - \Delta y|$; donde:

$$\Delta x = |Ax - Ay|;\ \Delta y = |Bx - By|$$

En Python sería:

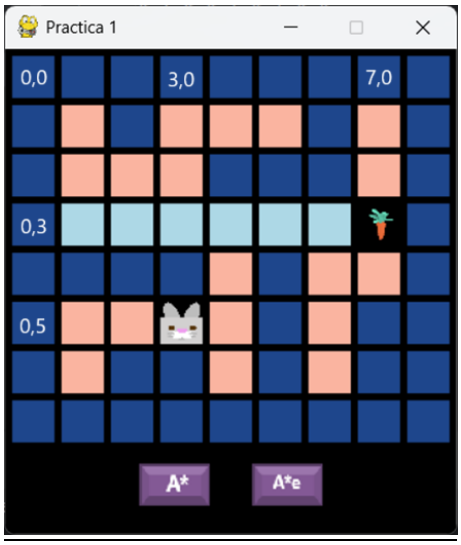
```python
# Fórmula de distancia octile
# Diferencias absolutas entre las coordenadas
dx = abs(desColumna - posColumna)
dy = abs(desFila - posFila)

self.h = min(dx, dy) * math.sqrt(2) + abs(dx - dy)
self.f = self.g + self.h
```

Teniendo en cuenta esto y, que todos los cálculos de las heurísticas se realizan dentro de la clase Nodo, el código del A* en Python y adaptado a nuestro problema del camino óptimo es el siguiente:

```python
def aestrella(mapi, origen, destino, camino, heuristica):  1 usage  djfj1 +1

    nodo_inicio = Nodo( padre: None, origen,  g: None, destino,  cal: 0, heuristica)
    nodo_inicio.g = 0
    nodo_inicio.h = 0
    nodo_inicio.f = 0

    listaFrontera = [nodo_inicio]  # Nodos no explorados
    listaInterior = []  # Nodos explorados

    while listaFrontera:  # Mientras queden nodos en la lista frontera
        listaFrontera = sorted(listaFrontera, key=lambda nodo: nodo.f)  # Ordeno la lista frontera
        n = listaFrontera[0]  # Nodo con menor f

        if not es_admisible(n, destino):
            # Opcional: detener el algoritmo si encuentra una heurística inadmisible
            print("La heurística no es admisible.")

        # Verificar si el nodo actual es el destino
        if n.posicion.getCol() == destino.getCol() and n.posicion.getFila() == destino.getFila():
            actual = n
            calorias = 0
            print(f'Número de nodos visitados -> {len(listaInterior)}')

            while actual.padre is not None:  # Recompone el camino y calcula las calorías
                coordX = actual.posicion.getFila()
                coordY = actual.posicion.getCol()
                calorias += calculoCalorias(mapi, coordX, coordY)
```

```
            actual = actual.padre
        return n.f, calorias  # Devuelve el coste total
    else:  # Si no es el destino
        listaFrontera.remove(n)
        listaInterior.append(n)

        vecinos = calculoVecino(mapi, n, destino, heuristica)

        for m in vecinos:
            if m not in listaInterior:  # Que no estén ya expandidos
                if m not in listaFrontera:  # Añadir a la frontera si no está
                    listaFrontera.append(m)
                else:
                    for l in listaFrontera:
                        if m == l and m.f < l.f:
                            m.padre = n
                            listaFrontera[listaFrontera.index(l)] = m
return -1, 0
```

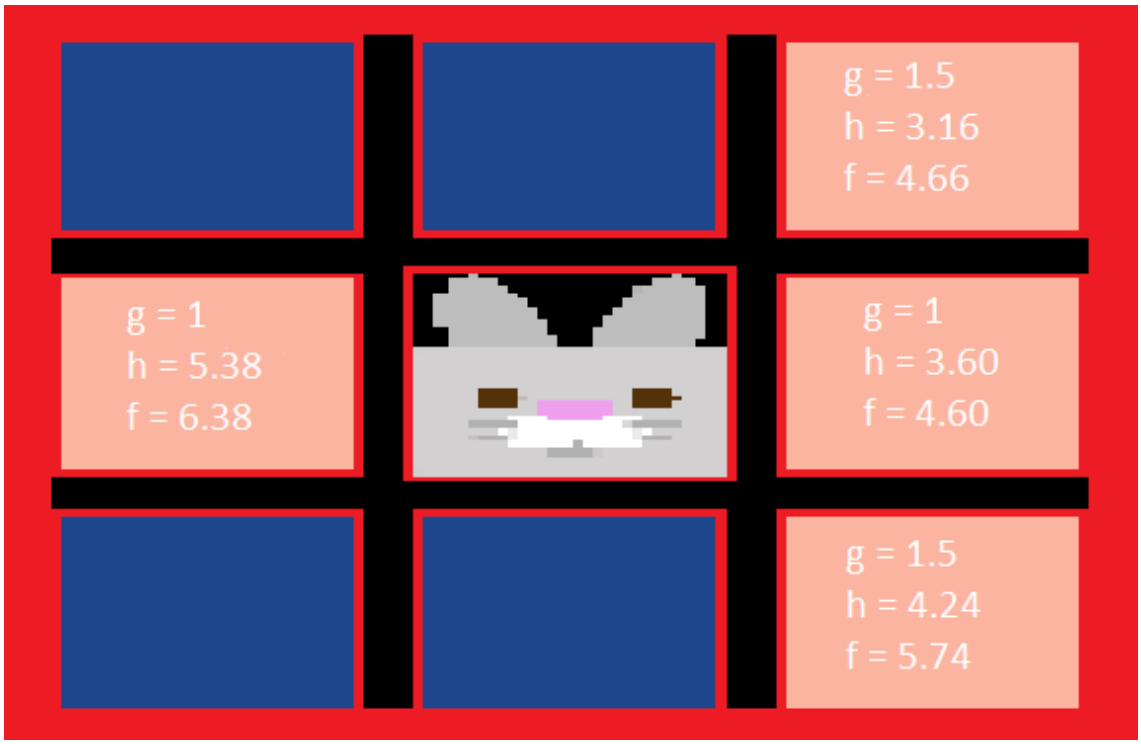
Utilizaremos el siguiente escenario para explicar con una traza el algoritmo:



Se puede observar, el conejo (origen) se encuentra en la posición 5,3 y la zanahoria (destino) en la posición 3,7. Nada más iniciar el algoritmo se crean las listas frontera e interior, donde en la lista frontera se inicializará con el nodo origen (3,7) y la lista interior, vacía.

Como la lista frontera no está vacía, el código entra dentro del while, ahora se escogerá el nodo perteneciente a la lista frontera que tenga menor f, en este caso, solo hay un nodo en la lista frontera. A partir de ahora, este nodo elegido lo llamaremos nodo actual, se debe verificar si el nodo actual es igual al nodo destino, como no lo es, se borra este nodo de la lista frontera ya que va a ser expandido y lo añadimos a la lista interior.

Ahora expandiremos el nodo actual creando sus nodos vecinos, por lo que ahora tendremos lo siguiente:



Tenemos los vecinos que no son un muro de la posición origen con el calculo de su heurística, de estos vecinos seleccionaremos el que tenga un menor valor de f, en este caso vemos que es la casilla 5,4 que tiene un valor de 4,60. Estos nodos que hemos calculado sus valores, se añadirán a la lista frontera, este sería el estado de la lista frontera y la lista interior:

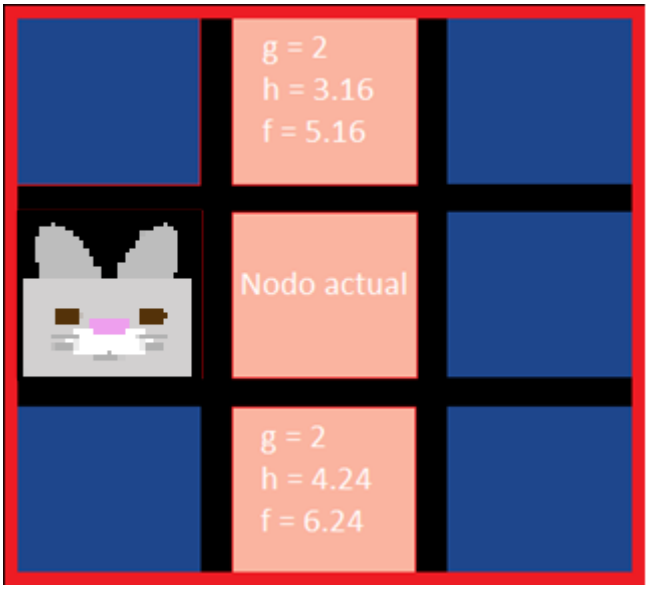
Lista frontera

4,4	5,2	5,4	6,4

Lista interior

5,3

Ahora, se hará lo mismo teniendo en cuenta que el nodo actual será el 5,4. Calcularemos sus vecinos:



En este caso pasa una peculiaridad, y es que hemos calculado el camino hasta el nodo 4,4 desde origen y desde 5,4. Actualizamos las listas:

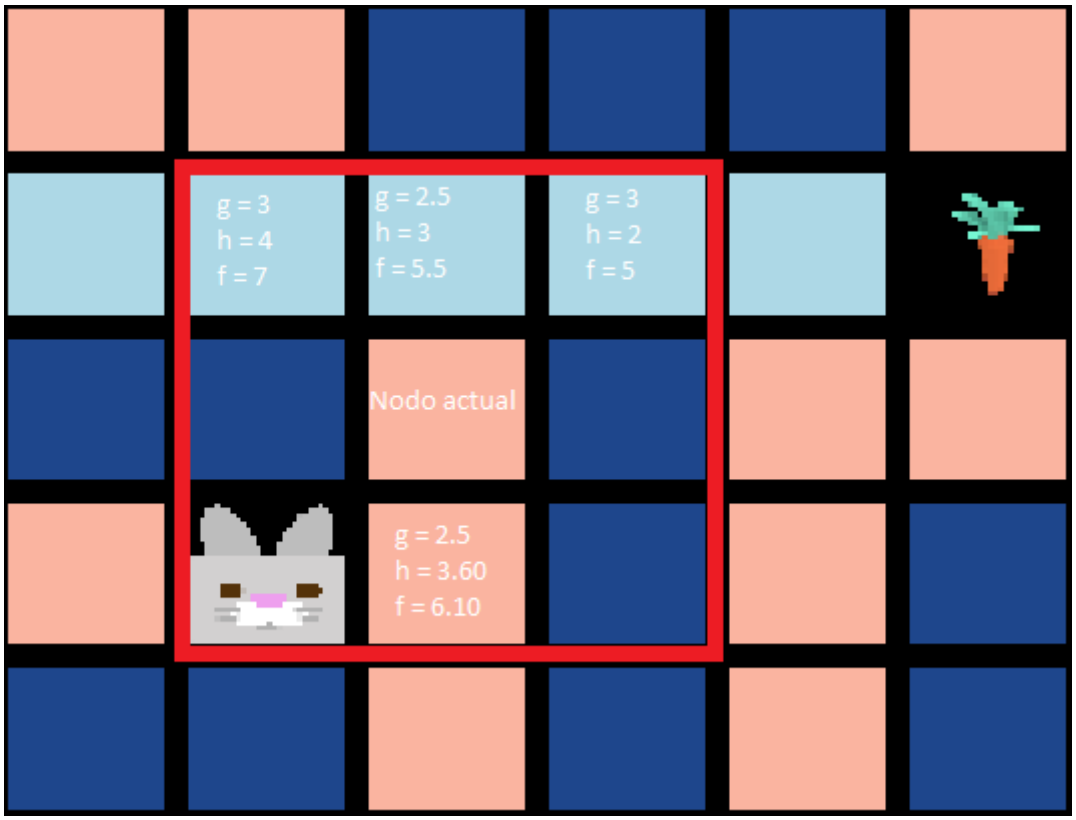
Lista frontera

4,4	5,2	6,4

Lista interior

5,3	5,4

Vemos que desde el origen llegamos al nodo 4,4 con un coste menor, f = 4,66. Por lo que el nodo actual será el 4,4:



Actualizamos las listas:

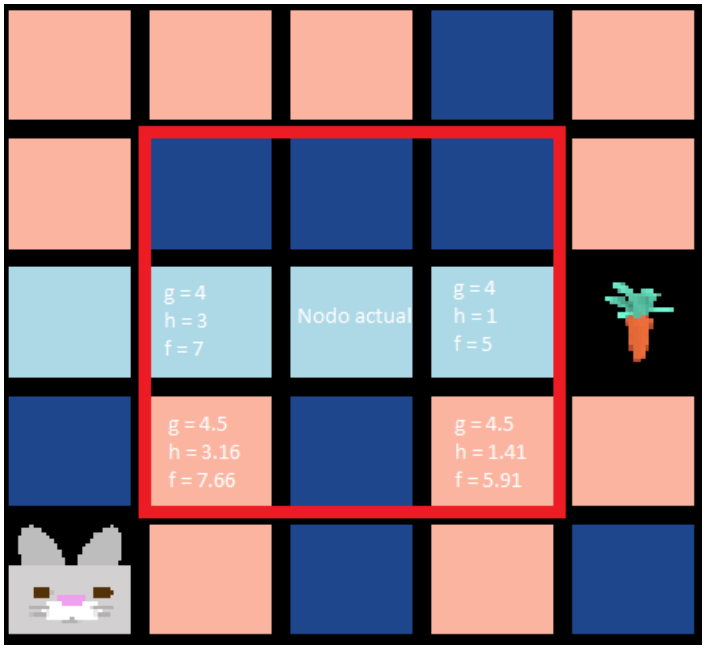
Lista frontera

3,3	5,2	6,4	3,4	3,5

Lista interior

5,3	5,4	4,4

Se puede observar que el nodo con una f más pequeña es el nodo 3,5 con una f = 5. Por lo que se escogerá como nodo actual:



Actualizamos las listas:

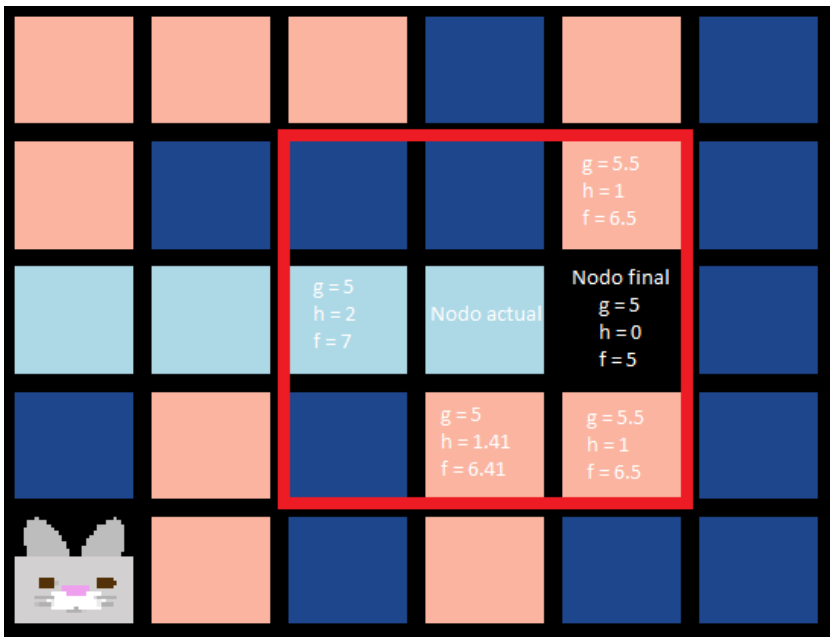
Lista frontera

3,3	5,2	6,4	3,4	3,6	4,6

Lista interior

5,3	5,4	4,4	3,5

Se seleccionará el nodo con menor f, en este caso se puede ver que es el nodo 3,6 con un f = 5:



Ahora se deben actualizar las listas:

Lista frontera

3,3	5,2	6,4	3,4	2,7	4,6	3,7	4,7

Lista interior

5,3	5,4	4,4	3,5	3,6

Por último, elegiremos el nodo con menor f, y nos daremos cuenta de que la posición 3,7 que es la que menor f tiene es la misma que la posición destino. Actualizamos una última vez las listas y ya se habrá terminado el algoritmo:

Lista frontera

3,3	5,2	6,4	3,4	2,7	4,6	4,7

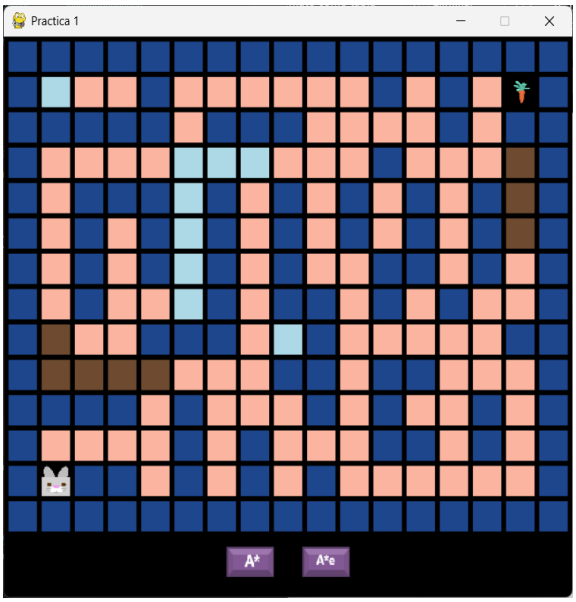
Lista interior

5,3	5,4	4,4	3,5	3,6	3,7

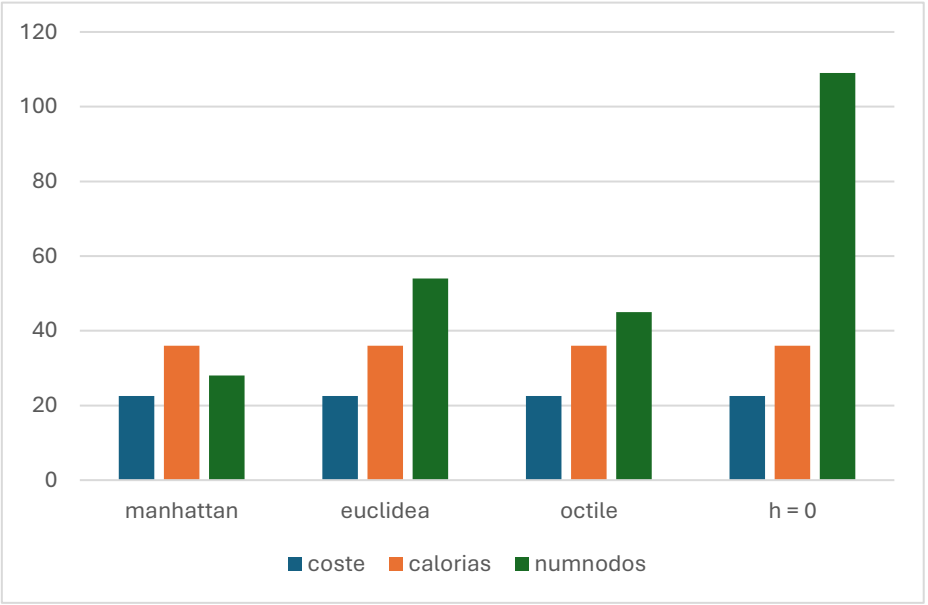
Análisis comparativos de las distintas heurísticas

Vamos a comparar las diferentes heurísticas con varios mapas que usaremos como ejemplos, tendremos en cuenta el coste del camino, las calorías que consume el conejo para llegar al destino y el número de nodos explorados que necesita el algoritmo para encontrar el camino óptimo.

Este es el primer mapa con el que compararemos las heurísticas:

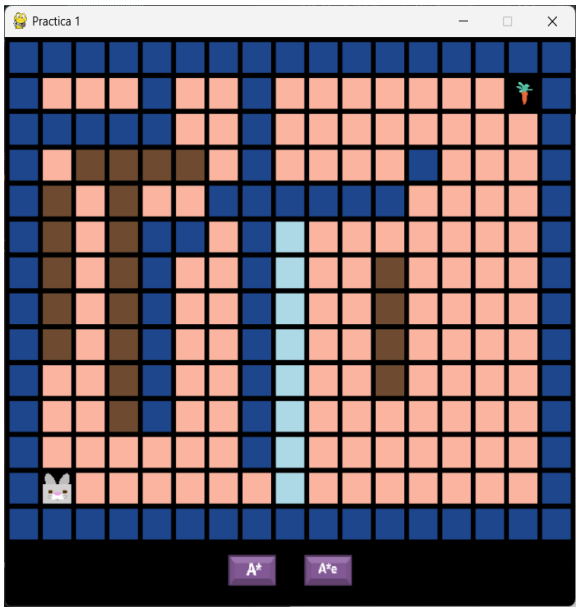


A continuación, podremos ver la gráfica donde se muestran los datos obtenidos:

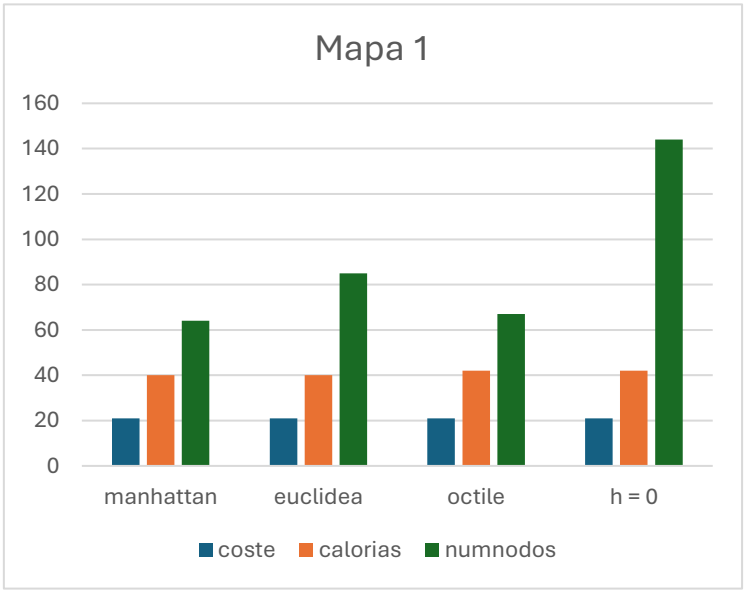


Es claro que la heurística h = 0, es la menos óptima ya que explora una cantidad de nodos exagerados. La que menos nodos explora es con la distancia Manhattan seguida de la octal y la euclídea.

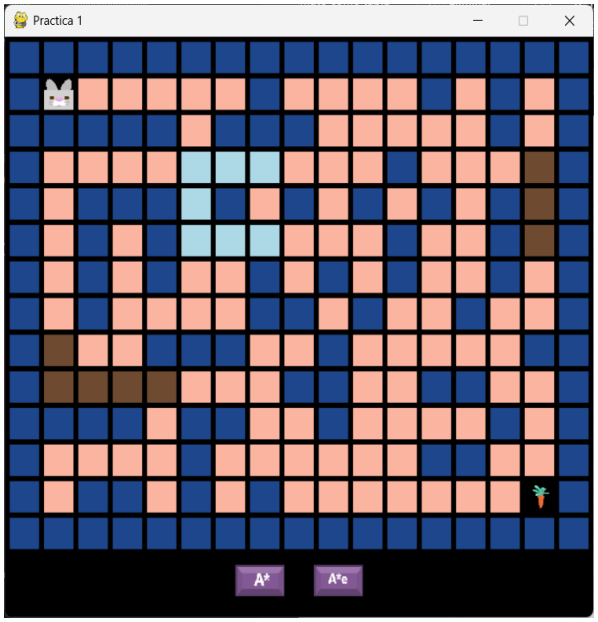
El segundo mapa que tendremos en cuenta es el siguiente:



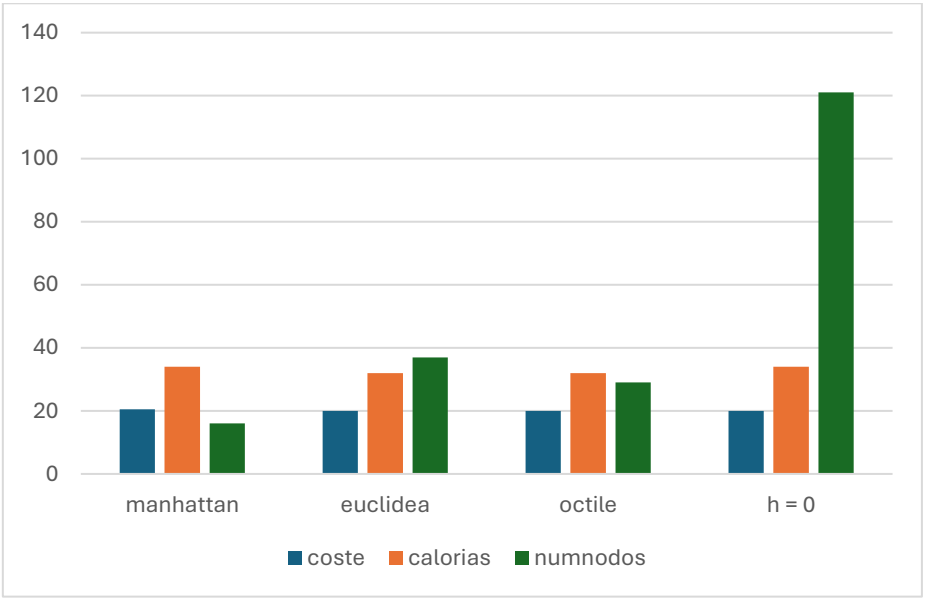
La gráfica asociada a este mapa es:



Como se puede observar, es muy parecida a la anterior. Se mostrará un mapa más ya que la mayoría de las soluciones dan el mismo resultado o muy parecido:



Y la gráfica es la siguiente:



También se va a analizar si estas heurísticas que se han explicado sean admisibles, esto quiere decir que nunca sobreestime el coste real del camino desde un nodo hasta el objetivo. Resumidamente, una heurística es admisible cuando siempre proporciona un valor igual o menor al coste real para llegar al destino desde cualquier punto en el mapa.

Para comprobar cual de estas heurísticas son admisibles, se ha implementado una función adicional que cuando reconoce que la distancia heurística de un nodo hasta el destino es mayor que la distancia real, se muestra en la pantalla una alerta. La función es la siguiente:

```
es_admisible(nodo_actual, destino):  2 usages  djfj1
distancia_real = math.sqrt((destino.getCol() - nodo_actual.posicion.getCol())**2 + (destino.getFila() - nodo_actual.posicion.getFila())**2)
if nodo_actual.h > distancia_real:
    print(f'Nodo actual -> {nodo_actual.h}\nDistancia real -> {distancia_real}')
    return False
return True
```

Ahora se volverá a ejecutar el algoritmo y se podrá ver cual heurística es admisible y cual no. Primero se hará uso de la heurística Manhattan:

```
Nodo actual -> 23
Distancia real -> 16.401219466856727
La heurística no es admisible.
Nodo actual -> 22
Distancia real -> 15.620499351813308
La heurística no es admisible.
Nodo actual -> 20
Distancia real -> 14.212670403551895
La heurística no es admisible.
```

Como se observa en la imagen, esta heurística se puede afirmar que no es admisible dado que la distancia del nodo actual es mayor a la distancia real.

Se cambiará la heurística por la Euclídea:

```
Número de nodos visitados -> 54
```

La única salida que obtenemos, tras la ejecución del programa repetidas veces para comprobar diferentes casos, es el número de nodos visitados, por lo que nunca entra en ejecución la función para comprobar si es admisible. Se puede asegurar que esta heurística es admisible.

Ahora, se cambiará la heurística por el octal:

```
Nodo actual -> 17.14213562373095
Distancia real -> 16.401219466856727
La heurística no es admisible.
Nodo actual -> 16.14213562373095
Distancia real -> 15.620499351813308
La heurística no es admisible.
Nodo actual -> 15.142135623730951
Distancia real -> 14.866068747318506
La heurística no es admisible.
Nodo actual -> 14.727922061357857
Distancia real -> 14.212670403551895
La heurística no es admisible.
Nodo actual -> 13.313708498984761
Distancia real -> 12.806248474865697
La heurística no es admisible.
Nodo actual -> 12.313708498984761
Distancia real -> 12.041594578792296
La heurística no es admisible.
```

Se puede observar que hay bastantes casos en los que ocurre que se sobreestima el costo real del camino desde un nodo hasta el objetivo. En este caso, se aprecia que la cantidad de casos en los que ocurre esto es mucho más elevado que con Manhattan.

Se usará ahora, la heurística h = 0:

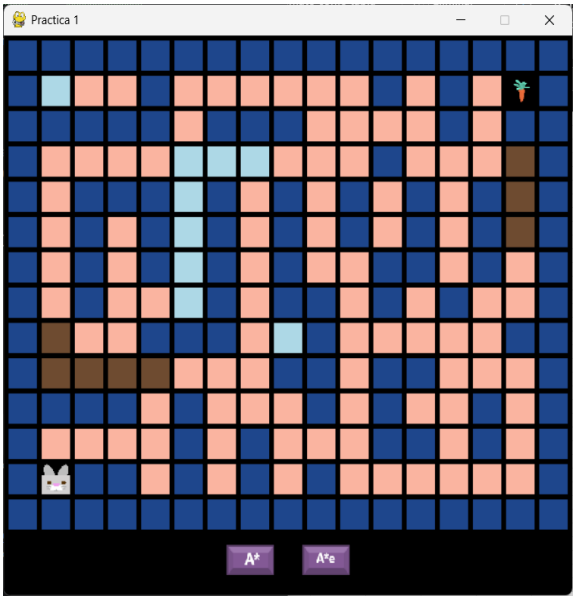
```
Número de nodos visitados -> 109
```

Al igual que con la Euclídea no aparece ningún caso en el que se sobreestime el costo del camino, por lo que se puede asegurar que esta heurística también es admisible.

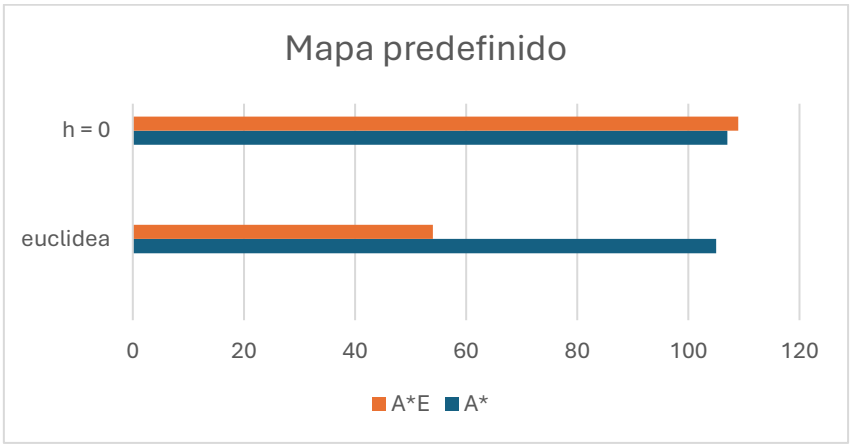
Análisis comparativos entre A* y A*Épsilon

A continuación, se realizará una comparación con tres mundos diferentes entre los dos algoritmos. Se tendrá en cuenta tanto el número de nodos explorados como las calorías que se obtienen al resolver el problema. Para esto, realizaremos el análisis utilizando únicamente las heurísticas admisibles.

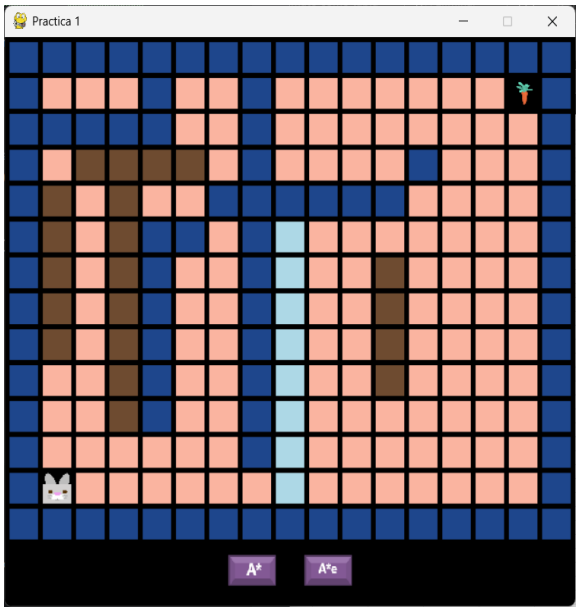
Para el primer ejemplo, usaremos el siguiente mapa:



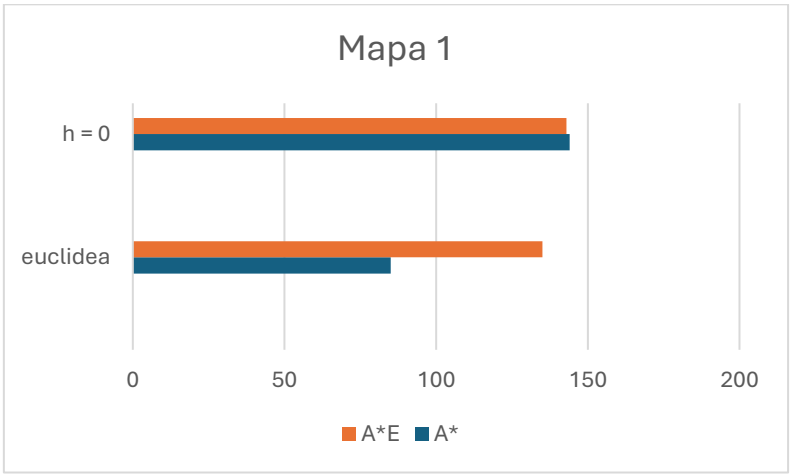
El resultado de la ejecución de los algoritmos es el siguiente:



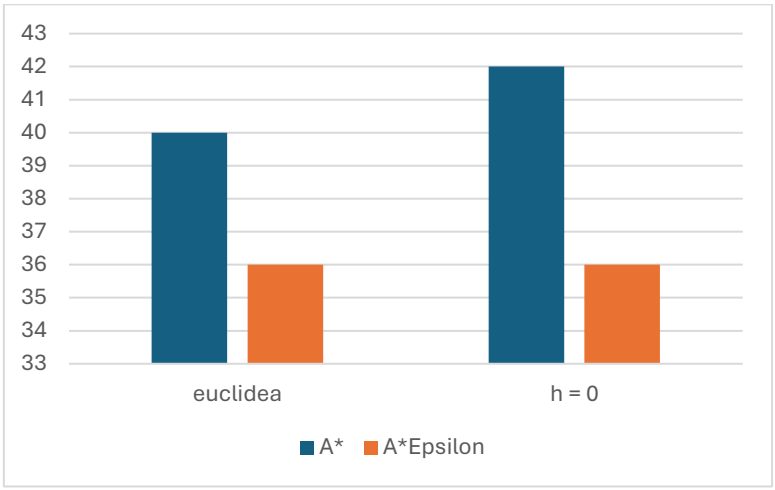
Se puede observar que en la heurística h = 0 es prácticamente igual el número de nodos explorados, pero en la Euclídea, hay una diferencia significativa. En este caso, las calorías y el costo son iguales en los dos escenarios. Esto va a ser constante en la mayoría los mapas que se han probado, se expondrá un ejemplo más:



Para este mapa, la gráfica es la siguiente:



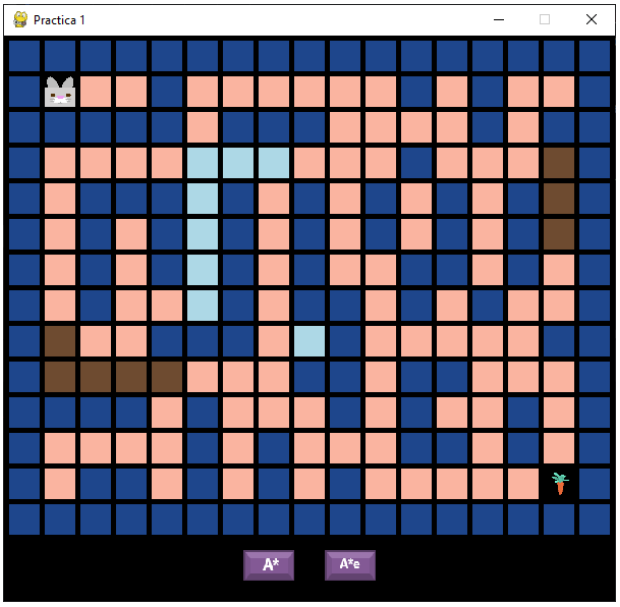
En este caso, también se puede apreciar que en la heurística Euclídea se nota la diferencia de nodos explorados. La diferencia es que, en el cálculo de las calorías, se puede observar en la siguiente gráfica:



Esta diferencia en este valor indica que, utilizando A*, el camino que ha encontrado no optimiza el cálculo de las calorías. Sin embargo, utilizando el algoritmo A*Épsilon encuentra un camino que requiere el mismo coste que con A* pero que, además, optimiza las calorías.

Casuísticas

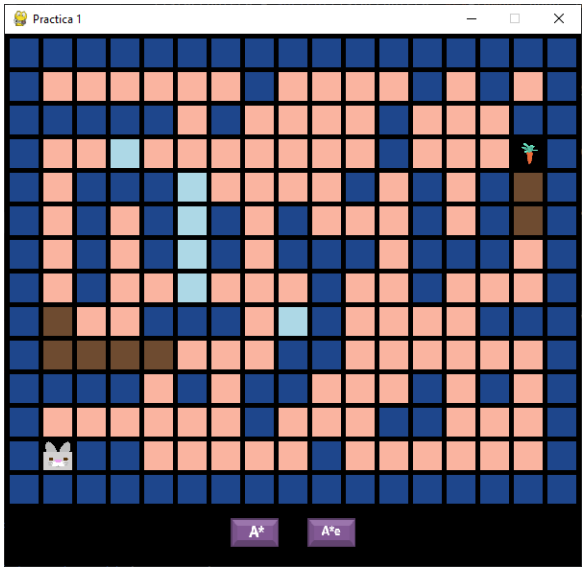
Se van a presentar varios casos que se pueden dar en la ejecución del algoritmo A*. El primero es el caso de que no haya solución:



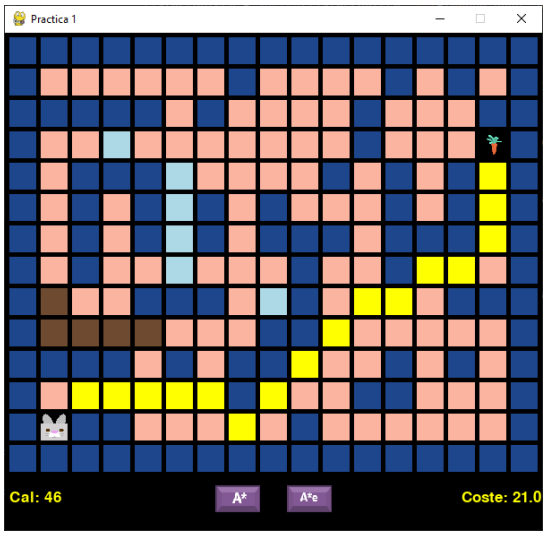
Aparece un mensaje indicando que no existe un camino válido entre origen y destino:

```
-------------------------------------Algoritmo A estrella simple-------------------------------------
Error: No existe un camino válido entre origen y destino
```

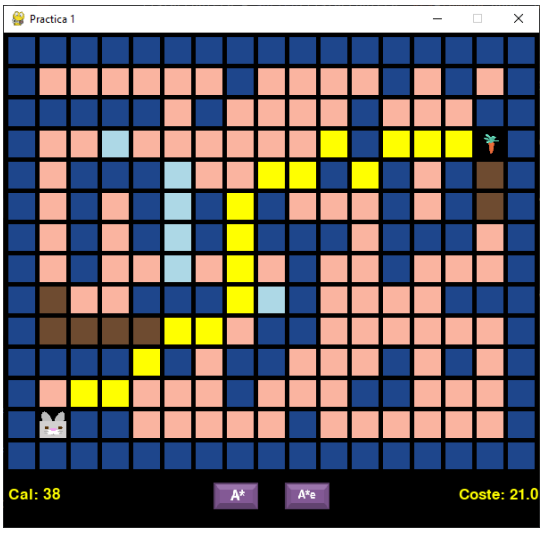
Otro caso que se puede presentar es que haya un camino diferente si se ejecuta el algoritmo A* o el A* Épsilon. Se utilizará el siguiente mapa:



Algoritmo A*:



Algoritmo A* Épsilon:



Esto es debido a que el primer algoritmo no tiene en cuenta el paso que tiene cada casilla, por eso el camino que calcula pasa por las casillas marrones y el segundo algoritmo no. También es notable en el cálculo de las calorías, que pasa de 46 a 38.

Bibliografía

- https://keepcoding.io/blog/que-es-la-busqueda-heuristica-y-para-que-sirve/#%C2%BFQue_es_la_busqueda_heuristica
- https://journalismcourses.org/wp-content/uploads/2020/07/nota-m2-1.pdf
- Chatgpt
- Materiales docentes