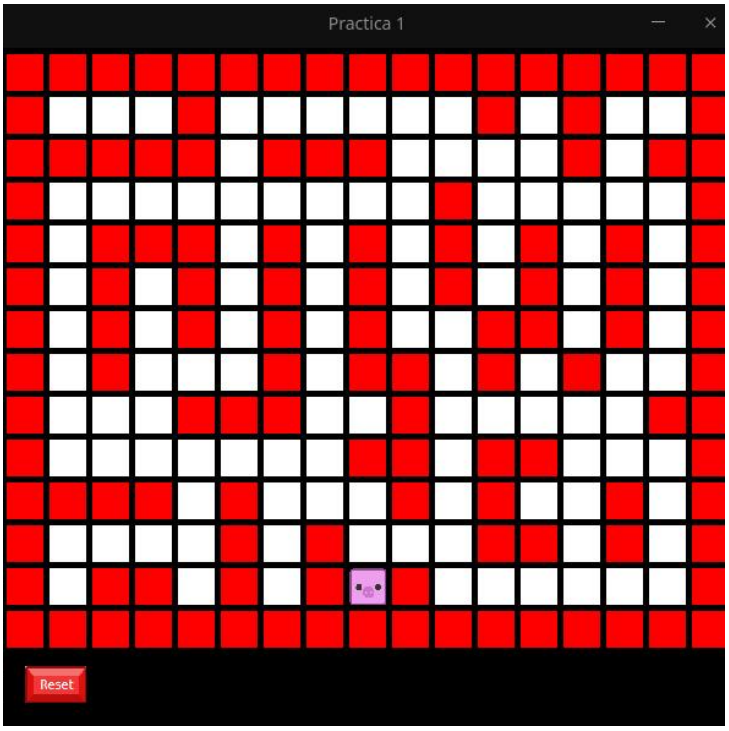
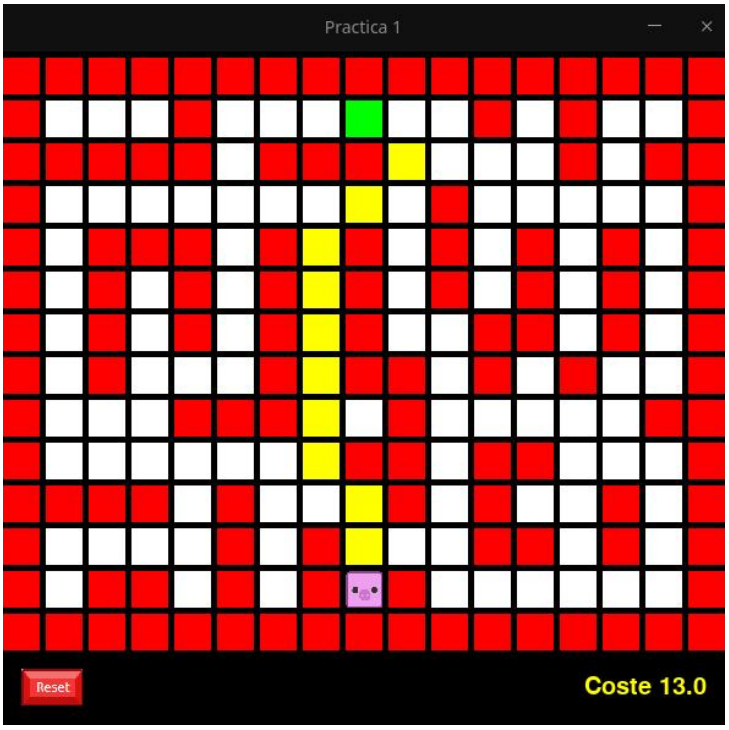
Práctica 1. Búsqueda heurística

En esta práctica se ha desarrollado el algoritmo A* con y sin ajuste de pesos para el problema planteado. En este problema, se debe calcular el camino óptimo y su coste para alcanzar el destino seleccionado desde un origen determinado. El mapa esta formado por casillas cuadradas blancas y rojas, no siendo accesibles estas últimas. El cerdito únicamente puede moverse de una casilla en una pero en 8 direcciones, estas son, las casillas adyacentes y las diagonales. El coste por movimiento es 1 para las adyacentes y 1,5 para las diagonales.





Algoritmo A*

El algoritmo que se ha implementado en esta práctica es un algoritmo de búsqueda que se emplea en el cálculo de caminos óptimos en una red, por lo que se clasifica dentro de los algoritmos de búsqueda en

grafos. Hace uso de una función de evaluación heurística mediante la cual etiqueta a los nodos dentro de la red y que determinará la probabilidad de que dichos nodos de pertenezcan al camino óptimo.

Esta función de evaluación, es la siguiente:

> `f(n) = g(n) + h(n)`

Como se aprecia, está a su vez compuesta por otra dos funciones. La primera de ellas indica la distancia del camino desde el nodo origen al nodo en evaluación `(n)`. La segunda, calcula la distancia estimada desde el nodo `(n)` hasta el nodo destino (esta última es la que utiliza las diferentes funciones heurísticas).

A continuación se detallan estas funciones g y h:

> - `g(n)`: `c(s, n)`: Calcula el coste desde el nodo origen `(s)` hasta `(n)`. En la implementación se ha hecho de manera que se calcula el coste del nodo `(n)` con su padre y se le suma el coste su padre, de manera que cada nodo acumula el total. Estima el valor de `g*(n)`.
> - `h(n)`: Hace una estimación del coste que supone ir desde el nodo actual hasta el nodo destino. Para ello realiza un cálculo siguiendo una heurística. Las heurísticas (heu.) implementadas son las heuristicas Cero, Diagonal, Manhattan, Euclidea y Minkowski; que se detallarán más adelante. Estima el valor de `h*(n)`.
> - `f(n)`: Por tanto, f es una composición entre la estimación del coste hasta el destino y el coste acumulado en el nodo `(n)`. Estima el valor de `f*(n)`.

Al tratarse de un algoritmo de busqueda en amplitud, es un algoritmo completo, es decir, en caso de existir una solución, dará siempre con ella.

Para que el algoritmo sea capaz de encontrar el camino óptimo, es necesario que la función `h(n)` sea admisible, es decir, que no sobreestime el coste real de alcanzar el nodo objetivo (`h*(n)`). Cuanto mejor se estime `h*(n)`, mejor optimizado estará el algoritmo, el caso contrario dará lugar a un mayor coste computacional debido a una más exahustiva busqueda de nodos.

La función heurística h(n) será admisible cuando se cumpla que:

> **$h(n) \leq h^*(n) \ \forall \ n$**

Algoritmo A* con ajuste de pesos

El objetivo y funcionamiento de este algoritmo es el mismo que el A*. El motivo para su uso radica en que el mantenimiento de la admisibilidad fuerza al algoritmo a consumir mucho tiempo en discriminar caminos cuyos costes no varían muy significativamente. Con tal de solucionar este problema, se decide aumentar la velocidad perdiendo acotadamente calidad a través de una relajación de la restricción de optimalidad. Esta relajación se logra mediante el ajuste de pesos.

Para al estimar `f*(n)` en el ajuste de pesos se utiliza una función ponderada:

> **$f_w(n) = (1-w) \times g(n) + w \times h(n)$**

Dentro del rango $0 \leq w \leq 1$ obtenemos estrategias mixtas intermedias. Si h(n) es admisible tenemos que:

- En el rango $0 \leq w \leq 1/2$, A* con fw(n) también es admisible.
- Dependiendo de la diferencia existente entre h(n) y h*(n), A con fw(n) puede perder la admisibilidad en el rango $1/2 < w \leq 1$

Es por esto que se ha decidido implementar un w = 0,5

Código implementado

Se ha implementado una clase Nodo que almacena la información necesaria de cada nodo. Esta información incluye, la casilla del propio nodo, el nodo padre y los valores de `g(n)`, `h(n)` y `f(n)` del nodo. Además de manera estática la clase guarda el método heurístico que se va a usar. Para cambiar entre un método y otro solo se debe cambiar el valor de la variable `heurisiticaSelec` a las diferentes opciones. A continuación se muestra:

In [ ]:

```
from math import sqrt
from casilla import Casilla

def heuristica0(c1, c2):
    return 0

def heuristicaDiagonal(c1, c2):
    return max(abs(c1.getFila() - c2.getFila()), abs(c1.getCol() - c2.getCol()))

def heuristicaManhattan(c1, c2):
    return abs(c2.getCol() - c1.getCol()) + abs(c2.getFila() - c1.getFila())

def heuristicaEuclidea(c1, c2):
    return sqrt((c2.getCol() - c1.getCol())**2 + (c2.getFila() - c1.getFila())**2)

def heuristicaMinkowski(c1, c2):    #generalización de la euclidea y manhattan, donde e
    p = 3                           #respectivamente para las otras, se ha escogido 3
    return (abs(c2.getCol() - c1.getCol())**p + abs(c2.getFila() - c1.getFila())**p)**(

class Nodo:
    heuristicaSelec = 'ma' #puede variar entre 'mi', 'eu', 'ma', 'di', 'ze'
    def __init__(self, casilla, padre, destino):
        self.casilla = casilla      #Casilla del nodo actual
        self.padre = padre          #Nodo padre
        self.calcular(destino)      #Calcula g, h y f

    def heuristica(self, c1, c2): #Segun la heurística que esté seleccionada, devuelve
        if Nodo.heuristicaSelec == 'mi':
            return heuristicaMinkowski(c1, c2)
        elif Nodo.heuristicaSelec == 'eu':
            return heuristicaEuclidea(c1, c2)
        elif Nodo.heuristicaSelec == 'ma':
            return heuristicaManhattan(c1, c2)
        elif Nodo.heuristicaSelec == 'di':
            return heuristicaDiagonal(c1, c2)
        elif Nodo.heuristicaSelec == 'ze':
            return heuristica0(c1, c2)

    def getHeuristica(): #Devuelve la heurística seleccionada en formato str
        if Nodo.heuristicaSelec == 'mi':
            return 'Minkowski'
        elif Nodo.heuristicaSelec == 'eu':
            return 'Euclidea'
        elif Nodo.heuristicaSelec == 'ma':
            return 'Manhattan'
        elif Nodo.heuristicaSelec == 'di':
            return 'Diagonal'
        elif Nodo.heuristicaSelec == 'ze':
            return 'Cero'

    def getF(self):
        return self.f

    def getG(self):
        return self.g

    def getH(self):
        return self.h

    def getCasilla(self):
        return self.casilla

    def getPadre(self):
```

```
        return self.padre

    def calcG(self): #Calcula g
        g = 0
        if self.casilla != self.padre: #Casilla y padre serán iguales en el caso en que
            if abs(self.casilla.getFila() - self.padre.getCasilla().getFila()) == 1:
                g = 1
            if abs(self.casilla.getCol() - self.padre.getCasilla().getCol()) == 1:
                if g == 1:
                    g = 1.5
                else:
                    g = 1
            g += self.padre.getG()
        self.g = g

    def calcH(self, destino): #Calcula h mediante la heurística seleccionada
        self.h = self.heuristica(self.casilla, destino)

    def calcular(self, destino): #Calcula g, h y f para el nodo
        self.calcG()
        self.calcH(destino)
        self.f = self.g + self.h
```

Además para el algortimo A* con ajuste de pesos se ha utilizado una nueva clase NodoAjustado, la cual hereda de Nodo y guarda de manera estática el coeficiente de ajuste. Se muestra a continuación:

In [ ]:
```
class NodoAjustado(Nodo):
    w = 0.5 #Ajuste de pesos
    def calcular(self, destino):
        self.calcG()
        self.calcH(destino)
        self.f = (1 - NodoAjustado.w)*self.g + NodoAjustado.w*self.h
```

El algoritmo A* utiliza dos listas, una llamada listaInterior en la que almacena los nodos ya explorados, y otra llamada listaFrontera en la que se irán almacenando los nodos a explorar que son frontera del nodo que se explore en cada iteración. Esta última se inicializa con el nodo origen.

Hasta que la listaFrontera se vacie, esto solo ocurre en caso de que no haya camino posible hasta el destino, se itera lo siguiente:

- Se obtiene el nodo más prometedor de la listaFrontera (`n`), este es el que menor `f(n)` tenga.
- En caso de ser el nodo destino el algoritmo habrá finalizado, construirá el camino en el mapa y devolverá el coste total de este.
- En caso contrario, añadirá `n` a la listaInterior después de eliminarlo de la listaFrontera e incluirá los nodos hijos de `n` , estos son aquellos nodos accesibles desde `n` , que no se encuentren en la listaInterior a la listaFrontera.
- Si la listaFrontera se ha vaciado y no se ha encontrado el destino la función devuelve `-1` .

A continuación se muestra la implementación del algoritmo y las funciones externas que utiliza:

In [ ]:
```
def equalsCasillas(c1, c2): #Dos casillas son iguales si sus filas y columnas son igual
    if c1.getFila() == c2.getFila() and c1.getCol() == c2.getCol():
        return True
    return False

def obtenerMejor(listaNodos): #Devuelce el nodo con mayor h
    mejor = 0
    for n in listaNodos:
        if mejor == 0:
            mejor = n
        elif mejor.getF() > n.getF():
            mejor = n
    return mejor
```

```
def construirCamino(camino, nodo): #devuelve el coste del camino trazado y actualiza la
    g = nodo.getG()
    while type(nodo.getPadre()) == Nodo or type(nodo.getPadre()) == NodoAjustado:
        nodo = nodo.getPadre()
        camino[nodo.getCasilla().getFila()][nodo.getCasilla().getCol()] = "*" #Pertenec

    return g

def bueno(mapi, casilla):
    pass

def listaHijos(nodo, mapa, destino, ajustado): #devuelve una lista de los nodos hijos d
    lista = []
    for i in [0, -1, 1]:
        for j in [0, -1, 1]:
            casillaPrueba = Casilla(nodo.getCasilla().getFila() - i, nodo.getCasilla().
            if bueno(mapa, casillaPrueba) and not (i == j == 0):
                if ajustado:
                    lista.append(NodoAjustado(casillaPrueba, nodo, destino))
                else:
                    lista.append(Nodo(casillaPrueba, nodo, destino))
    return lista

def seEncuentra(nodo, lista): #Comprueba si existe el nodo en la lista
    if len(lista) == 0:
        return False
    for n in lista:
        if equalsCasillas(nodo.getCasilla(), n.getCasilla()) and equalsCasillas(nodo.ge
            return True
    return False

def comparaIncluye(lista, nodo, mExplorados, nExplorados): #comprueba si la casilla est
    esta = False
    for n in lista:
        if equalsCasillas(n.getCasilla(), nodo.getCasilla()):
            if n.getG() > nodo.getG():
                lista.remove(n)
                lista.append(nodo)
            esta = True
    if not esta:
        lista.append(nodo)
        mExplorados[nodo.getCasilla().getFila()][nodo.getCasilla().getCol()] = nExplora
        nExplorados[0] += 1

def equalsNodos(n1, n2): #Si las casillas y las casillas de los padres son iguales, los
    if not equalsCasillas(n1.getCasilla(), n2.getCasilla()):
        return False
    if type(n1.getPadre()) == Nodo or type(n1.getPadre()) == NodoAjustado:
        if type(n2.getPadre()) == Nodo or type(n2.getPadre()) == NodoAjustado:
            if not equalsCasillas(n1.getPadre().getCasilla(), n2.getPadre().getCasilla(
                return False
        else:
            if not equalsCasillas(n1.getPadre().getCasilla(), n2.getPadre()):
                return False
    else:
        if type(n2.getPadre()) == Nodo or type(n2.getPadre()) == NodoAjustado:
            if not equalsCasillas(n1.getCasilla(), n2.getPadre().getCasilla()):
                return False
        else:
            if not equalsCasillas(n1.getPadre().getCasilla(), n2.getCasilla()):
                return False
    return True

def eliminarDeLista(l1, l2): #Devuelve una lista resultado de la diferencia entre la pr
    lf = []
    for n in l1:
        esta = False
        for m in l2:
            if (equalsCasillas(n.getCasilla(), m.getCasilla())):
                esta = True
        if esta == False:
```

```
            lf.append(n)
    return lf
```

In [ ]:

```
def aEstrella(mapi, origen, destino, camino, mExplorados, nExplorados):
    listaInterior = []
    listaFrontera = [Nodo(origen, origen, destino)] #inicializala lista con el nodo ori
    while len(listaFrontera) != 0:
        n = obtenerMejor(listaFrontera) # obtiene el nodo con menor coste esperado
        if equalsCasillas(n.getCasilla(), destino): #ha llegado a la meta
            return construirCamino(camino, n)
        else:
            listaFrontera.remove(n)
            listaInterior.append(n)

            for m in eliminarDeLista(listaHijos(n, mapi, destino, False), listaInterior
                comparaIncluye(listaFrontera, m, mExplorados, nExplorados)
    return -1
```

La diferencia entre la implementación del algoritmo con y sin ajuste de pesos en su implementación es únicamente que este va a utilizar nodos de tipo NodoAjustado, pues estos almacenan los datos de `f(n)` con el ajuste.

In [ ]:

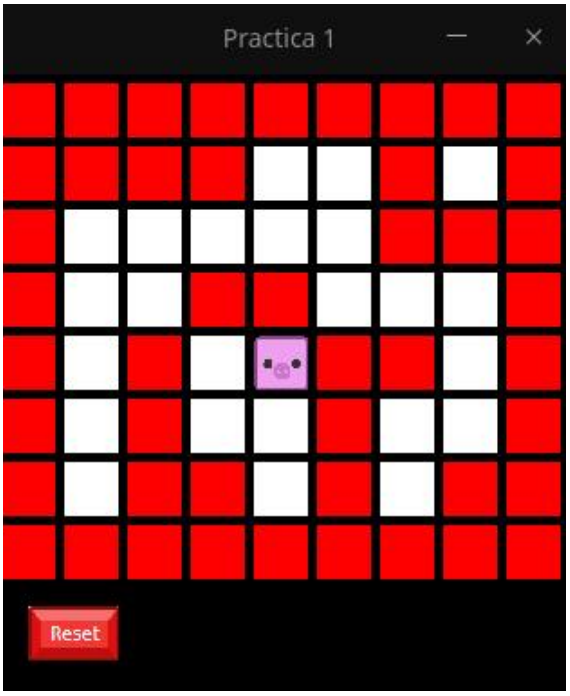
```
def aEstrellaAjustado(mapi, origen, destino, camino, mExplorados, nExplorados):
    listaInterior = []
    listaFrontera = [NodoAjustado(origen, origen, destino)] #inicializala lista con el
    while len(listaFrontera) != 0:
        n = obtenerMejor(listaFrontera) # obtiene el nodo con menor coste esperado
        if equalsCasillas(n.getCasilla(), destino): #ha llegado a la meta
            return construirCamino(camino, n)
        else:
            listaFrontera.remove(n)
            listaInterior.append(n)

            for m in eliminarDeLista(listaHijos(n, mapi, destino, True), listaInterior)
                comparaIncluye(listaFrontera, m, mExplorados, nExplorados)
    return -1
```

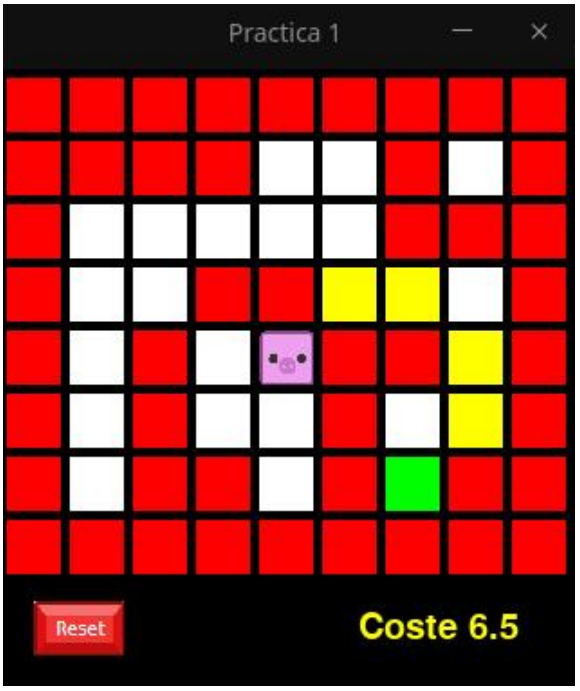
Otras diferencias implementadas con respecto al código de la plantilla son:

- Una variable `modoAjustado` booleana para controlar si llamar al algoritmo con o sin ajuste de pesos.
- Código que permite cronometrar la duración de cada ejecución del algoritmo y guardar estos datos, clasificados por mapa y método heurístico.
- Codigo que almacena en este archivo de datos el camino explorado y un mapa con el orden de exploración de los nodos. Además el número de nodos explorados

Traza detallada A*

Para realizar esta traza se ha utilizado el mapa del archivo `mapaChico.txt` . En este mapa, la ficha comienza en la casilla (4, 4). Se ha seleccionado como objetivo la casilla (6, 6).





La traza se ha realizado con la heurística euclidea.

El camino que sigue la ficha es (4, 4), (5, 3), (6, 3), (7, 4), (7, 5) y (6, 6). Como se muestra en la segunda imagen, el coste total es de 6,5, que viene de 3 movimientos diagonales y 2 adyacentes (3×1,5 + 2×1 = 6,5).

A continuación se muestran una matriz que indica el órden en que se han explorado los nodos:

In [ ]:
```
fich = open('../Fuente/datos-mapaChico.txt', 'r')
fila=-1
txt = ''
for i in fich:
  txt += i
data = txt.split('\n')
i = 0
while data[i] != 'Camino explorado:':
  i += 1
while data[i] != 'Nodos explorados: 14':
  print(data[i])
  i += 1
```

```
Camino explorado:
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1  9  8 -1 -1 -1
-1 -1  6 -1 -1  4  7 10 -1
-1 -1 -1  1  0 -1 -1 11 -1
-1 -1 -1  3  2 -1 13 12 -1
-1 -1 -1 -1  5 -1 14 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
```

Como se puede apreciar en el orden, se van explorando casillas en dirección hacia el objetivo. Analizando iteración a iteración el algoritmo tenemos lo siguiente (el formato de representación de nodos será ((x, y), f), estando f aproximada a 2 decimales):

Iteración	listaFrontera	listaInterior
0	[((4, 4), 2.82)]	[]
1	[((3, 4), 4.61), ((4, 5), 3.24), ((3, 5), 4.66), ((5, 3), 4.66)]	[((4, 4), 2.82)]
2	[((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66), ((4, 6), 4)]	[((4, 4), 2.82), ((4, 5), 3.24)]
3	[((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66),]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4)]

Iteración	listaFrontera	listaInterior
4	[((3, 5), 4.66), ((5, 3), 4.66), ((2, 3), 7.5)]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61)]
5	[((5, 3), 4.66), ((2, 3), 7.5)]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61), ((3, 5), 4.66)]
6	[((2, 3), 7.5), ((6, 3), 5.5), ((5, 2), 6.62), ((4, 2), 7.47)]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66)]
7	[((2, 3), 7.5), ((5, 2), 6.62), ((4, 2), 7.47), ((7, 3), 6.66), ((7, 4), 6.24)]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66), ((6, 3), 5.5)]
8	[((2, 3), 7.5), ((5, 2), 6.62), ((4, 2), 7.47), ((7, 3), 6.66), ((7, 5), 6.41), ((6, 5), 6.5)]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66), ((6, 3), 5.5), ((7, 4), 6.24)]
9	[((2, 3), 7.5), ((5, 2), 6.62), ((4, 2), 7.47), ((7, 3), 6.66), ((6, 5), 6.5), **((6, 6), 6.5)**]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66), ((6, 3), 5.5), ((7, 4), 6.24), ((7, 5), 6.41)]
10	[((2, 3), 7.5), ((5, 2), 6.62), ((4, 2), 7.47), ((7, 3), 6.66), **((6, 6), 6.5)**]	[((4, 4), 2.82), ((4, 5), 3.24), ((4, 6), 4), ((3, 4), 4.61), ((3, 5), 4.66), ((5, 3), 4.66), ((6, 3), 5.5), ((7, 4), 6.24), ((7, 5), 6.41), ((6, 5), 6.5)]

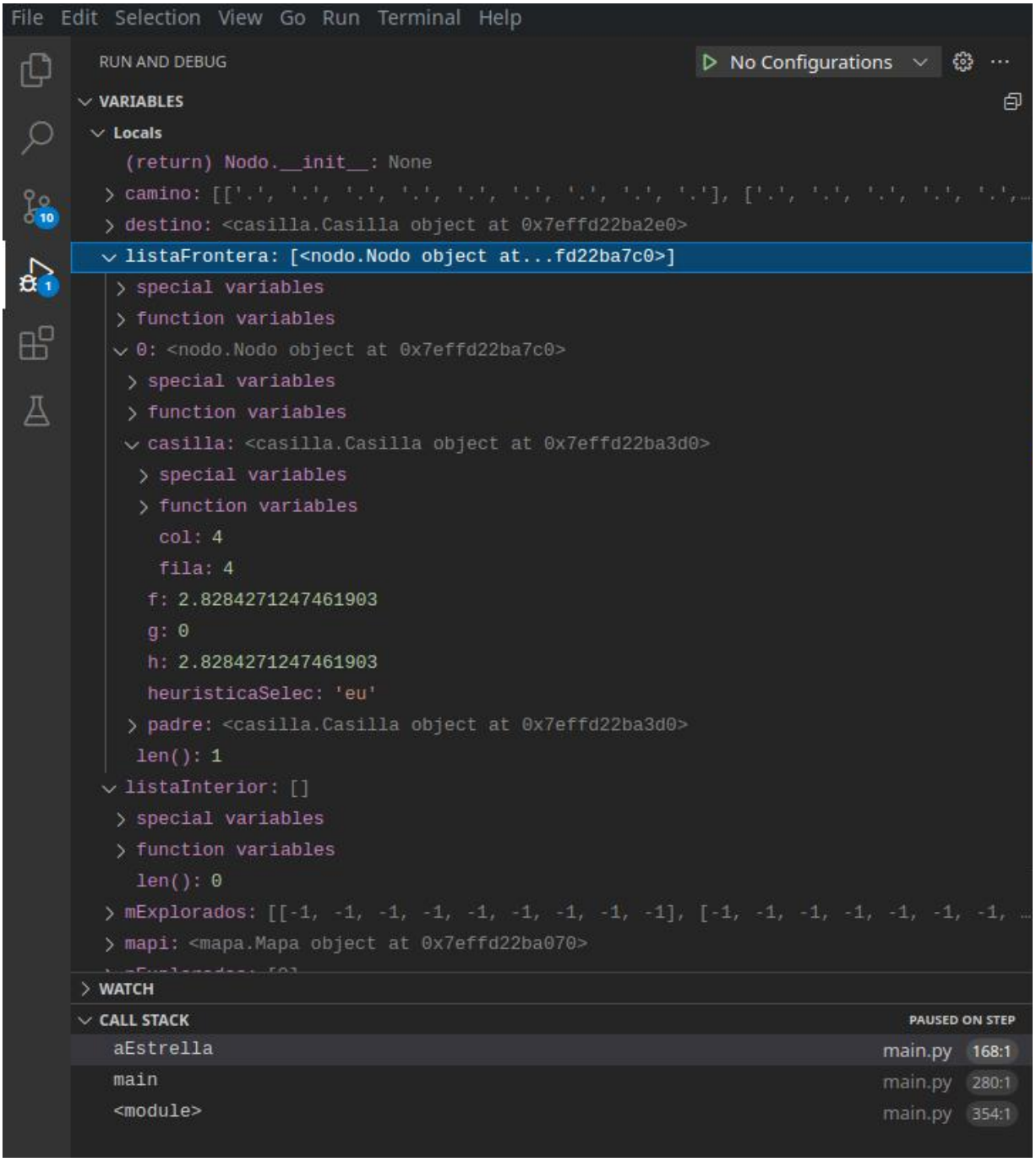
En cada iteración se extrae el nodo de la listaFrontera con menor f y se añade a listaInterior. Después se añaden las casillas adyacentes y diagonales de este nodo que sean válidas (que no sean rojas). Por ejemplo en la iteración 2 se ha añadido el nodo de la casilla (4, 6) a la listaFrontera, hijo del nodo de la casilla (4, 5), que previamente se ha extraído de la listaFrontera y ahora se halla en la listaInterior.

Esta tabla refleja los nodos y sus valores de f, pero es necesario explicar cual ha sido el cálculo realizado para obtener este valor f. Como ejemplo se utilizarán las casilla mencionada previamente, (4, 5) y (4, 6):

- El nodo de la casilla (4, 5) es hijo del nodo de la casilla origen (4, 4) y estas dos son casillas adyacentes, por lo que el valor de g es 1.
- La distancia euclidea se calcula de la siguiente forma: $d(A, B) = ((X_A - X_B)^2 + (Y_A - Y_B)^2)^{1/2}$. Esto es, la raiz de los cuadrados de las diferencias de coordenadas. Para el cálculo de h se ha utilizado esta distancia y por tanto el valor de h para la casilla (4, 5) es raíz de 5 (aproximado a 2 decimales 2.24).
- La f se calcula como la suma de g y h, por tanto para la casilla (4, 5) será 1 + 2.24 = 3.24. Como vemos en la tabla este resultado es así. -Repitiendo estos pasos para el nodo de la casilla (4, 6) tenemos que: g ≡ 1 + g(4,5) = 2; h ≡ raíz de 4 = 2; y por tanto f ≡ 2 + 2 = 4.

Se puede apreciar también, como el orden en que se van añadiendo nodos a la listaFrontera es el mismo que el que se indica en la matriz anterior.

Para obetener los datos de la tabla de iteraciones se ha realizado una traza paso a paso con el debugger de python de Visual Studio Code, que nos permite ver el valor de las variables locales en cada punto de ejecución. Los datos de las listas se han extraído al comienzo de cada iteración del bucle que examina si listaFrontera está vacía. En esta vista se aprecia el valor de las variables locales al comienzo de la primera iteración:



Traza con destino no alcanzable

Además se ha realizado otra traza en el mismo mapa. En esta, se ha seleccionado como casilla destino la (7, 1), que como se puede apreciar en la imagen del mapa, es inaccesible.

En este caso el algoritmo explora todos los nodos accesibles pero no logra encontrar el nodo de la casilla destino, por lo que finaliza lanzando un mensaje.

El orden de exploración de nodos por parte del algoritmo es el siguiente:

In [ ]:

```
while data[i] != 'Camino explorado:':
  i += 1
while data[i] != 'Nodos explorados: 23':
  print(data[i])
  i += 1
```

```
Camino explorado:
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 11 10 -1 -1 -1
-1 20 19 14  7  6 -1 -1 -1
-1 17 12 -1 -1  4  5  8 -1
-1 18 -1  1  0 -1 -1  9 -1
-1 22 -1  3  2 -1 16 15 -1
-1 23 -1 -1 13 -1 21 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
```