Carmen Rosal Malvar – Juan Carlos Utrilla Martín

rosalmalvar9@gmail.com – utri1990@gmail.com

Descripción breve

Aquí se detallara todo el trabajo realizado sobre el proyecto Pentaminos

PENTAMINOS

Memoria del proyecto

Contenido

[1. Decisiones de diseño 2](#_Toc485537810)

[2. Desarrollo de la implementación 4](#_Toc485537811)

[2.1. mapa.py 4](#_Toc485537812)

[2.2. ficha.py 6](#_Toc485537813)

[2.3. fichas.py 7](#_Toc485537814)

[2.4. rellenarmapa.py 9](#_Toc485537815)

[2.5. principal.py 10](#_Toc485537816)

[2.6. problema\_espacio\_estados.py 16](#_Toc485537817)

[2.7. Búsqueda\_espacio\_estados.py 16](#_Toc485537818)

[3. Pruebas realizadas 17](#_Toc485537819)

[3.1. Algoritmo de búsqueda en profundidad 17](#_Toc485537820)

# Decisiones de diseño

La idea a desarrollar en este problema es un mapa al que, a medida que avance el algoritmo, irá colocando las fichas de una forma determinada hasta que el número de casillas desmarcadas sea menor o igual que 4. Además, se le da libertad al usuario para elegir el tamaño del mismo, aportando una mayor interacción por parte de este.

En este problema, un estado es un mapa con una serie de fichas colocadas. En cada estado se guarda la ficha que se acaba de colocar, el tipo, el lugar que ocupa dentro de su tipo y se actualizan las coordenadas con la ficha recién añadida. De este modo, el estado inicial es un mapa vacío y el final es aquel que tenga todos sus huecos rellenos menos 4 o menos.

Para representar la posición que ocupa cada coordenada dentro de un mapa se ha considerado como coordenada (0, 0) aquella casilla situada en la esquina superior-izquierda, tal y como se muestra en el siguiente ejemplo:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | (0,0) | (0,1) | (0,2) | | (1,0) | (1,1) | (1,2) | | (2,0) | (2,1) | (2,2) | | |  |  |  | | --- | --- | --- | | x |  |  | |  |  |  | |  |  |  | |

Si queremos representar de forma gráfica la ficha F cuyas coordenadas son (0,1), (0,2), (1,0), (1,1), (2,1) quedaría de la siguiente forma:

|  |  |  |
| --- | --- | --- |
|  | x | x |
| x | x |  |
|  | x |  |

Dentro de cada ficha podemos observar los siguientes datos:

* Entre comillas simples se encuentra el tipo de ficha (**'F'**)
* El siguiente número sirve para representar el lugar que ocupa dentro de ese tipo. Como se tiene que considerar tanto los giros y como las simetrías, ésta es la forma en la que se ha decidido diferenciar.
* Las coordenadas de las que se compone.

Se ha desarrollado una mejora en el algoritmo a la hora de determinar todas las acciones del problema, eliminando directamente aquellas acciones que nunca se puedan realizar para un mapa, en donde, por ejemplo, para una determinada posición no podamos colocar una ficha concreta.

Cuando se comenzó a modelar el funcionamiento de nuestro problema, se planteó la idea de tener dos acciones: ponerFicha y quitarFicha. Nos dimos cuenta de que no era necesario tener dos acciones ya que lo único que nos interesa es un conjunto de fichas colocadas en el mapa tomando como referencia unas coordenadas (x, y) y, por tanto, no tiene ningún sentido la acción quitarFicha. En su lugar, nos quedamos con la acción ponerFicha que posteriormente se renombró a RellenarMapa.

Otro detalle a destacar es la heurística usada. Se considera mejor aquellos casos que dejen menos huecos encerrados en el tablero, definiendo hueco encerrado como aquel que está vacío y tiene las casillas de su derecha, izquierda, arriba y abajo ocupada.

# Desarrollo de la implementación

## mapa.py

Este script contiene la clase Mapa. Esta clase será utilizada para definir el estado de nuestro problema, en nuestro caso, un mapa y contiene los siguientes métodos:

* **Constructor**: recibe como parámetros el número de filas y de columnas y crea el mapa (una matriz) y una lista para las fichas que se van colocando en el problema.

**def** \_\_init\_\_(self, filas, columnas):  
 self.map = [([0] \* columnas) **for** i **in** range(filas)]  
 self.filas = filas  
 self.columnas = columnas  
 self.fichasColocadas = []

* **numeroFilas**: método que devuelve el número de filas del mapa.

**def numeroFilas**(self):  
 **return** len(self.map[0])

* **numeroColumnas**: método que devuelve el número de columnas del mapa.

**def numeroColumnas**(self):  
 **return** len(self.map)

* **valorCasilla**: método que devuelve si una ficha está ocupada o no.

**def valorCasilla**(self, fila, columna):  
 **return** self.map[fila][columna]

* **marcarCasilla**: método que marca una casilla del mapa.

**def marcarCasilla**(self, fila, columna):  
 **if** self.map[fila][columna] == 1:  
 **raise** ValueError("ERROR, la casilla ya está marcada")  
  
 self.map[fila][columna] = 1

* **desmarcarCasilla**: método que desmarca una casilla del mapa.

**def desmarcarCasilla**(self, fila, columna):  
 **if** self.map[fila][columna] == 0:  
 **raise** ValueError("ERROR, la casilla ya está desmarcada")  
  
 self.map[fila][columna] = 0

* **anadirFicha**: método que añade una ficha en el mapa.

**def anadirFicha**(self, x,y, ficha):  
 # Comprobamos que las variables de entrada son del tipo que

# se espera y si podemos añadir la ficha...  
 **if** isinstance(ficha, fic.Ficha)

**and** isinstance(x, int)

**and** isinstance(y, int)

**and** self.verificarFichaEnMapa(x, y, ficha):  
 # ...añado la ficha y su posición en la lista

# fichasColocadas  
 self.fichasColocadas.append(((x,y),ficha))  
  
 # Por cada posición de la ficha seleccionada...  
 **for** tupla **in** ficha.listaPosiciones:  
 # ...marco la casilla correspondiente  
 self.marcarCasilla(x + tupla[0], y + tupla[1])  
 **else**:  
 print("No puedo añadir la ficha")

* **verificarFichaEnMapa**: método que verifica si podemos añadir una ficha en nuestro mapa comprobando si la ficha se sale de los límites del mapa o si las casillas donde se quiere colocar están ocupadas.

**def verificarFichaEnMapa**(self, x, y, ficha):  
  
 # Si no corresponden con las instancias...  
 **if** isinstance(ficha, fic.Ficha)

**and** isinstance(x, int)

**and** isinstance(y, int):  
  
 # Recorre cada una de las casillas de la ficha...  
 **for** tupla **in** ficha.listaPosiciones:  
  
 # ...comprobamos que no nos salimos del mapa ...  
  
 **try**:  
 **if** x + tupla[0] > self.numeroColumnas() - 1

**or**

y + tupla[1] > self.numeroFilas() - 1:  
 **return False** # ...no puedo incluir la ficha  
 # ...y si la casilla ya se encuentra marcada...  
 **elif** self.map[x + tupla[0]][y + tupla[1]] == 1:  
 **return False** # ...no puedo incluir la ficha  
 # Si salta este error...  
 **except** IndexError:  
 # ... es que estamos comprobando posiciones

# fuera del mapa (que no existe)  
 **return False  
 else**:  
 **return False** # En caso de que todas las posiciones estén desmarcadas,

# entonces puedo añadir la ficha.  
 **return True**

* **mostrarMapa**: método que muestra el mapa por pantalla.

**def mostrarMapa**(self):  
 **for** i **in** range(0, self.numeroFilas(), 1):  
 print("+------" \* self.numeroFilas() + "+")  
  
 **for** j **in** range(0, self.numeroColumnas(), 1):  
 **if** self.map[i][j] == 1:  
 **for** ficha **in** self.fichasColocadas.values():  
 **for** tupla **in** ficha.listaPosiciones:  
 **if** (tupla[0], tupla[1]) == (i, j):  
 valorCasilla = str(ficha.letraFicha)

+ str(ficha.numero)  
 **else**:  
 valorCasilla = " "  
  
 print("| " + valorCasilla + " ", end="")  
  
 print("|")  
  
 print("+------" \* self.numeroFilas() + "+")

* **numCasillasMarcadas**: método que calcula el número de casillas que están marcadas en el mapa.

**def numCasillasMarcadas**(self):  
 contador = 0 # Cuenta el nº casillas marcadas  
  
 **for** i **in** self.map:  
 **for** j **in** i:  
 **if** j == 1:  
 contador += 1  
  
 **return** contador

* **numCasillasDesmarcadas**: método que calcula el número de casillas que están desmarcadas en el mapa.

**def numCasillasDesmarcadas** (self):  
 **return** (self.numeroFilas() \* self.numeroColumnas())

– self.numCasillasMarcadas()

## ficha.py

Este script contiene la clase Ficha. Esta clase será utilizada para representar el tipo Ficha que tendrá cualquier ficha del problema. Contiene los siguientes métodos:

**Constructor**: crea una ficha del juego. Recibe como parámetros el tipo, el lugar que ocupa dentro de su tipo y sus coordenadas.

**def** \_\_init\_\_(self, letraFicha, numero, listaPosiciones):  
 self.letraFicha = letraFicha  
 self.numero = numero  
 self.listaPosiciones = listaPosiciones

* **mostrarFicha**: método que dibuja una ficha por pantalla. Este método nos ha permitido encontrar de forma más fácil e intuitiva ciertos errores y solucionarlos rápidamente.

**def mostrarFicha**(self):  
 **for** i **in** range(0, 5, 1):  
 **for** j **in** range(0, 5, 1):  
 **if** (i, j) **in** self.listaPosiciones:  
 print(" x ", end="")  
 **else**:  
 print(" ", end="")  
  
 print("")

## fichas.py

Este script contiene la clase auxiliar Fichas. En ella, definimos todas y cada una de las fichas, incluyendo giros y simetrías, que tendrá nuestro problema. Contiene los siguientes métodos:

* **Constructor**: define todas y cada una de las fichas. Se ha implementado en un diccionario de fichas para facilitar la búsqueda de fichas del conjunto con sus giros y simetrías.

**def** \_\_init\_\_(self):  
 self.dicFichas = {

# Ficha F

'F0': fic.Ficha('F',0, [(0,1),(0,2),(1,0),(1,1),(2,1)] ),  
 'F1': fic.Ficha('F',1, [(0,1),(1,0),(1,1),(1,2),(2,2)] ),  
 'F2': fic.Ficha('F',2, [(0,1),(1,1),(1,2),(2,0),(2,1)] ),  
 'F3': fic.Ficha('F',3, [(0,0),(1,0),(1,1),(1,2),(2,1)] ),  
 'F4': fic.Ficha('F',4, [(0,1),(1,0),(1,1),(2,1),(2,2)] ),  
 'F5': fic.Ficha('F',5, [(0,2),(1,0),(1,1),(1,2),(2,1)] ),  
 'F6': fic.Ficha('F',6, [(0,0),(0,1),(1,1),(1,2),(2,1)] ),  
 'F7': fic.Ficha('F',7, [(0,1),(1,0),(1,1),(1,2),(2,0)] ),

# Ficha I  
 'I0': fic.Ficha('I',0, [(0,0),(0,1),(0,2),(0,3),(0,4)] ),  
 'I1': fic.Ficha('I',1, [(0,0),(1,0),(2,0),(3,0),(4,0)] ),

# Ficha L  
 'L0': fic.Ficha('L',0, [(0,0),(1,0),(2,0),(3,0),(3,1)] ),  
 'L1': fic.Ficha('L',1, [(0,0),(0,1),(0,2),(0,3),(1,0)] ),  
 'L2': fic.Ficha('L',2, [(0,0),(0,1),(1,1),(2,1),(3,1)] ),  
 'L3': fic.Ficha('L',3, [(0,3),(1,0),(1,1),(1,2),(1,3)] ),  
 'L4': fic.Ficha('L',4, [(0,0),(0,1),(1,0),(2,0),(3,0)] ),  
 'L5': fic.Ficha('L',5, [(0,0),(1,0),(1,1),(1,2),(1,3)] ),  
 'L6': fic.Ficha('L',6, [(0,1),(1,1),(2,1),(3,0),(3,1)] ),  
 'L7': fic.Ficha('L',7, [(0,0),(0,1),(0,2),(0,3),(1,3)] ),

# Ficha N   
 'N0': fic.Ficha('N',0, [(0,1),(1,0),(1,1),(2,0),(3,0)] ),  
 'N1': fic.Ficha('N',1, [(0,0),(0,1),(0,2),(1,2),(1,3)] ),  
 'N2': fic.Ficha('N',2, [(0,1),(1,1),(2,0),(2,1),(3,0)] ),  
 'N3': fic.Ficha('N',3, [(0,0),(0,1),(1,1),(1,2),(1,3)] ),  
 'N4': fic.Ficha('N',4, [(0,0),(1,0),(2,0),(2,1),(3,1)] ),  
 'N5': fic.Ficha('N',5, [(0,2),(0,3),(1,0),(1,1),(1,2)] ),  
 'N6': fic.Ficha('N',6, [(0,0),(1,0),(1,1),(2,1),(3,1)] ),  
 'N7': fic.Ficha('N',7, [(0,1),(0,2),(0,3),(1,0),(1,1)] ),

# Ficha P  
 'P0': fic.Ficha('P',0, [(0,1),(1,0),(1,1),(2,0),(2,1)] ),  
 'P1': fic.Ficha('P',1, [(0,0),(0,1),(1,0),(1,1),(1,2)] ),  
 'P2': fic.Ficha('P',2, [(0,0),(0,1),(1,0),(1,1),(2,0)] ),  
 'P3': fic.Ficha('P',3, [(0,0),(0,1),(0,2),(1,1),(1,2)] ),  
 'P4': fic.Ficha('P',4, [(0,0),(0,1),(1,0),(1,1),(2,1)] ),  
 'P5': fic.Ficha('P',5, [(0,0),(0,1),(0,2),(1,0),(1,1)] ),  
 'P6': fic.Ficha('P',6, [(0,0),(1,0),(1,1),(2,0),(2,1)] ),  
 'P7': fic.Ficha('P',7, [(0,1),(0,2),(1,0),(1,1),(1,2)] ),

# Ficha T  
 'T0': fic.Ficha('T',0, [(0,0),(0,1),(0,2),(1,1),(2,1)] ),  
 'T1': fic.Ficha('T',1, [(0,2),(1,0),(1,1),(1,2),(2,2)] ),  
 'T2': fic.Ficha('T',2, [(0,1),(1,1),(2,0),(2,1),(2,2)] ),  
 'T3': fic.Ficha('T',3, [(0,0),(1,0),(1,1),(1,2),(2,0)] ),

# Ficha U

'U0': fic.Ficha('U',0, [(0,0),(0,2),(1,0),(1,1),(1,2)] ),  
 'U1': fic.Ficha('U',1, [(0,0),(0,1),(1,0),(2,0),(2,1)] ),  
 'U2': fic.Ficha('U',2, [(0,0),(0,1),(0,2),(1,0),(1,2)] ),  
 'U3': fic.Ficha('U',3, [(0,0),(0,1),(1,1),(2,0),(2,1)] ),

# Ficha V  
 'V0': fic.Ficha('V',0, [(0,0),(1,0),(2,0),(2,1),(2,2)] ),  
 'V1': fic.Ficha('V',1, [(0,0),(0,1),(0,2),(1,0),(2,0)] ),  
 'V2': fic.Ficha('V',2, [(0,0),(0,1),(0,2),(1,2),(2,2)] ),  
 'V3': fic.Ficha('V',3, [(0,2),(1,2),(2,0),(2,1),(2,2)] ),

# Ficha W

'W0': fic.Ficha('W',0, [(0,0),(1,0),(1,1),(2,1),(2,2)] ),  
 'W1': fic.Ficha('W',1, [(0,1),(0,2),(1,0),(1,1),(2,0)] ),  
 'W2': fic.Ficha('W',2, [(0,0),(0,1),(1,1),(1,2),(2,2)] ),  
 'W3': fic.Ficha('W',3, [(0,2),(1,1),(1,2),(2,0),(2,1)] ),

# Ficha X  
 'X0': fic.Ficha('X',0, [(0,1),(1,0),(1,1),(1,2),(2,1)] ),

# Ficha Y  
 'Y0': fic.Ficha('Y',0, [(0,1),(1,0),(1,1),(2,1),(3,1)] ),  
 'Y1': fic.Ficha('Y',1, [(0,2),(1,0),(1,1),(1,2),(1,3)] ),  
 'Y2': fic.Ficha('Y',2, [(0,0),(1,0),(2,0),(2,1),(3,0)] ),  
 'Y3': fic.Ficha('Y',3, [(0,0),(0,1),(0,2),(0,3),(1,1)] ),  
 'Y4': fic.Ficha('Y',4, [(0,1),(1,1),(2,0),(2,1),(3,1)] ),  
 'Y5': fic.Ficha('Y',5, [(0,0),(0,1),(0,2),(0,3),(1,2)] ),  
 'Y6': fic.Ficha('Y',6, [(0,0),(1,0),(1,1),(2,0),(3,0)] ),  
 'Y7': fic.Ficha('Y',7, [(0,1),(1,0),(1,1),(1,2),(1,3)] ),

# Ficha Z  
 'Z0': fic.Ficha('Z',0, [(0,0),(0,1),(1,1),(2,1),(2,2)] ),  
 'Z1': fic.Ficha('Z',1, [(0,2),(1,0),(1,1),(1,2),(2,0)] ),  
 'Z2': fic.Ficha('Z',2, [(0,1),(0,2),(1,1),(2,0),(2,1)] ),  
 'Z3': fic.Ficha('Z',3, [(0,0),(1,0),(1,1),(1,2),(2,2)] )  
 }

* **dibujarFichas**: método que dibuja las fichas en pantalla de un tipo concreto. Este método ha sido utilizado únicamente para verificar si las coordenadas de cada ficha, introducidas de forma manual, se han añadido correctamente.

**def dibujarFichas**(self, tipoFicha):  
 # Por cada una de las fichas del tipo indicado...  
 **for** ficha **in** self.dicFichas[tipoFicha]:  
 # ...iremos dibujandolo casilla a casilla si corresponde  
 **for** i **in** range(0, 5, 1):  
 **for** j **in** range(0, 5, 1):  
 **if** (i,j) **in** ficha:  
 print(" x ", end="")  
 **else**:  
 print(" ", end="")  
  
 print("")

* **listadoFichas**: método que lista por pantalla todas las fichas disponibles.

**def listadoFichas**(self):  
 **for** keys, values **in** self.dicFichas.items():  
 print(keys)  
 print(values)

* **cogerFicha**: método que se utiliza para coger una ficha de todas las disponibles. Este método tiene dos implementaciones.

**def cogerFicha**(self, ficha):  
 **return** self.dicFichas[ficha]  
  
**def cogerFicha**(self, ficha, numero):  
 **return** self.dicFichas[str(ficha)+str(numero)]

## rellenarmapa.py

Este script contiene la clase RellenarMapa que herede de la clase Acción. En ella, se define la única acción que tiene el problema y, como acción que es, se deben redefinir los métodos: es\_aplicable, aplicar y opcionalmente coste. Contiene los siguientes métodos:

* **Constructor**: recibe como parámetros las coordenadas x, y a partir de la cual se colocará la ficha y la ficha correspondiente.

**def** \_\_init\_\_(self, x, y, ficha):  
 nombre = 'Ficha {} en pos.({},{})'.format(ficha, x, y)  
 super().\_\_init\_\_(nombre)  
 self.x = x  
 self.y = y  
 self.ficha = ficha

* **es\_aplicable**: método redefinido que devuelve cuando la acción es aplicable.

**def es\_aplicable**(self, estado):  
 # El mapa tiene los huevos vacíos de una ficha a partir de

# las coordenadas (x, y) y no se sale del mapa  
 **return** estado.verificarFichaEnMapa(self.x, self.y,

self.ficha)

* **aplicar**: método redefinido que aplica la acción al estado, es decir, añade la ficha en el mapa en la posición (x, y).

**def aplicar**(self, estado):

# Copia el estado actual   
 nuevo\_estado = copy.deepcopy(estado)

# Definimos el nuevo estado sobre esa copia  
 nuevo\_estado.anadirFicha(self.x, self.y, self.ficha)   
  
 # El nuevo estado será el mapa actualizado  
 **return** nuevo\_estado

* **coste**: método redefinido que devuelve el coste que tiene aplicar una acción. El problema no exige que ciertas fichas tengan un coste asociado porque en nuestro caso, este coste se utiliza para comprobar que existen diferentes soluciones en nuestro problema.

**def coste**(self, estado):  
 **if** self.ficha.letraFicha == 'I':  
 **return** 2  
 **else**:  
 **return** 1

## principal.py

En este apartado se evalúa todo el código:

# ----------------------------- MODULOS -----------------------------  
**import** Modulos.Mapa **as** map  
**import** Modulos.Fichas **as** fic  
**import** Modulos.RellenarMapa **as** accion  
**import** Modulos.problema\_espacio\_estados **as** probee  
**import** Modulos.búsqueda\_espacio\_estados **as** busquee  
**import** sys, timeit, functools  
  
# --------------- CONSTANTES Y VARIABLES GLOBALES -------------------  
menu\_actions = {} # Diccionario de menus  
fichas = fic.Fichas() # Todas las fichas del problema  
numFilas = **None** # Nº de filas del mapa  
numColumnas = **None** # Nº de columnas del mapa  
  
problemaPentominos = **None** # Representa el problema  
acciones = [] # Representa las acciones del problema  
estadoInicial = **None**

# ------------------------ MÉTODOS AUXILIARES -----------------------  
# definicionProblema: método que define el Problema Pentonimos  
**def definicionProblema**():  
 # Necesitamos indicarle a Python que queremos utilizar una

# variable global  
 **global** numFilas, numColumnas  
  
 # Mientras filas no tenga un valor asignado...  
 **while not** numFilas:  
 **try**:  
 numFilas = int(input("Elija el nº de filas: "))  
 **except** ValueError:  
 print("Los datos introducidos no son correctos")  
  
 # Mientras columnas no tenga un valor asignado...  
 **while not** numColumnas:  
 **try**:  
 numColumnas = int(input("Elija el nº de columnas: "))  
 **except** ValueError:  
 print("Los datos introducidos no son correctos")  
  
 # Mejora: eliminamos aquellas acciones que ya sabemos con certeza  
 # que no se pueden aplicar. Nos referimos a aquellas

# acciones en las que partiendo de una coordenada (x,y),

# se sale del mapa  
 **for** keys, values **in** fichas.dicFichas.items():  
 **for** x **in** range(0, numColumnas, 1):  
 **for** y **in** range(0, numFilas, 1):  
 fueraLimites = **False  
  
 for** tupla **in** values.listaPosiciones:  
 **if** x + tupla[0] > numColumnas - 1

**or** y + tupla[1] > numFilas - 1:  
 fueraLimites = **True** # Si la acción no está fuera de los límites...  
 **if not** fueraLimites:  
 # ...creamos la acción  
 acciones.append(accion.RellenarMapa(x, y, values))  
  
 # Necesitamos indicarle a Python que queremos utilizar una

# variable global  
 **global** estadoInicial  
 **global** problemaPentominos  
  
 mapa = map.Mapa(numFilas, numColumnas)  
 estadoInicial = mapa  
  
 problemaPentominos = probee.ProblemaEspacioEstados(  
 acciones, estadoInicial, []  
 )

# h1: heurística que cuenta el número de huecos vacíos  
**def h1**(nodo):  
 estado = nodo.estado  
  
 huecosCerrados = 0  
  
 **for** i **in** range(0, estado.numeroFilas(), 1):  
 **for** j **in** range(0, estado.numeroColumnas(), 1):  
 # Si la casilla está desmarcada...  
 **if** estado.valorCasilla(i, j) == 0:  
 # Calculamos los índices...  
 arriba = j - 1  
 abajo = j + 1  
 derecha = i + 1  
 izquierda = i - 1  
  
 # Comprobamos si nos salimos del rango...  
 **if** (arriba < 0)

**or** (izquierda < 0)

**or** (abajo > estado.numeroFilas() - 1)

**or** (derecha > estado.numeroColumnas() - 1):  
 **if** (arriba < 0)

**and** (izquierda < 0):  
 # Comprobamos casillas: derecha, abajo  
 **if** estado.valorCasilla(derecha, j) == 1

**and** estado.valorCasilla(i, abajo) == 1:  
 huecosCerrados += 1  
 **elif** izquierda < 0

**and** (abajo > estado.numeroFilas() - 1):  
 # Comprobamos casillas: derecha, arriba  
 **if** estado.valorCasilla(derecha, j) == 1

**and** estado.valorCasilla(i, arriba) == 1:  
 huecosCerrados += 1  
  
 **elif** (abajo > estado.numeroFilas() - 1)

**and** (derecha > estado.numeroColumnas() - 1):  
 # Comprobamos casillas: arriba, izquierda  
 **if** estado.valorCasilla(i, arriba) == 1

**and** estado.valorCasilla(izquierda, j)== 1:  
 huecosCerrados += 1  
  
 **elif** arriba < 0

**and** (derecha > estado.numeroColumnas() - 1):  
 # Comprobamos casillas: abajo, izquierda  
 **if** estado.valorCasilla(i, abajo) == 1

**and** estado.valorCasilla(izquierda, j)== 1:  
 huecosCerrados += 1  
 **elif** arriba < 0:  
 # Comprobamos casillas: izquierda, derecha,

# abajo  
 **if** estado.valorCasilla(izquierda, j) == 1

**and** estado.valorCasilla(derecha, j) == 1 \  
 **and** estado.valorCasilla(i, abajo) == 1:  
 huecosCerrados += 1  
 **elif** izquierda < 0:  
 # Comprobamos casillas: derecha, arriba,

# abajo  
 **if** estado.valorCasilla(derecha, j) == 1

**and** estado.valorCasilla(i, arriba) == 1 \  
 **and** estado.valorCasilla(i, abajo) == 1:  
 huecosCerrados += 1  
 **elif** abajo > estado.numeroFilas() - 1:  
 # Comprobamos casillas: arriba, izquierda,

# derecha  
 **if** estado.valorCasilla(i, arriba) == 1

**and** estado.valorCasilla(izquierda, j) == 1\  
 **and** estado.valorCasilla(derecha, j) == 1:  
 huecosCerrados += 1  
  
 **elif** derecha > estado.numeroColumnas() - 1:  
 # Comprobamos casillas: izquierda, arriba,

# abajo  
 **if** estado.valorCasilla(izquierda, j) == 1

**and** estado.valorCasilla(i, arriba) == 1 \  
 **and** estado.valorCasilla(i, abajo) == 1:  
 huecosCerrados += 1

# Si no se sale del rango, comprobamos casillas:

# arriba, abajo, derecha, izquierda  
 **elif** estado.valorCasilla(derecha, j) == 1

**and** estado.valorCasilla(izquierda, j) == 1 \  
 **and** estado.valorCasilla(i, arriba) == 1

**and** estado.valorCasilla(i, abajo) == 1:  
 huecosCerrados += 1  
  
 **return** huecosCerrados  
  
# menuPrincipal: método que representa al menú principal por pantalla.  
**def menuPrincipal**():  
 # Definimos el problema  
 definicionProblema()  
  
 # Ejecuta el comando clear (limpia la consola)  
 print("\n" \* 100)  
  
 # Se muestran las opciones disponibles  
 print("+----------------- Bienvenido ------------------+")  
 print("| Por favor, elige el menu que quieres iniciar: |")  
 print("| 1. Ejecutar búsqueda informada |")  
 print("| 2. Ejecutar búsqueda no informada |")  
 print("+-----------------------------------------------+")  
 print("| 0. Quitar |")  
 print("+-----------------------------------------------+")  
  
 opcion = input(" >> ")  
 ejecutarMenu(opcion)  
 **return**

# ejecutarMenu: método que ejecuta cualquier menú de los disponibles  
**def ejecutarMenu**(opcion):  
 # Capturamos la elección  
 eleccion = opcion.lower() # Se pasa la opción a minúscula  
  
 # Si no se ha elegido nada...  
 **if** eleccion == '':  
 menu\_acciones['menuPrincipal']() # Se vuelve al menú principal  
 # En caso contrario...  
 **else**:  
 **try**:  
 menu\_acciones[eleccion]() # Se carga el menú elegido  
 **except** KeyError: # Si hemos elegido una acción que no existe  
 print ("Selección no valida, intentelo de nuevo.")  
 menu\_acciones['menuPrincipal']()  
 **return**# Menu 1: BUSQUEDA INFORMADA  
**def menuBusquedaNoInformada**():  
 print("\n" \* 100)  
 # Se muestran las opciones disponibles  
 print("+----------- BUSQUEDA NO INFORMADA -------------+")  
 print("| Elija el algoritmo que quieres ejecutar: |")  
 print("| 1. B. no Informada en profundidad |")  
 print("| 2. B. no Informada en anchura |")  
 print("+-----------------------------------------------+")  
 print("| 9. Volver al menú principal |")  
 print("| 0. Salir del programa |")  
 print("+-----------------------------------------------+")  
  
 eleccion = input(" >> ")  
  
 # Ejecutamos el algoritmo correspondiente  
 **if** eleccion.lower() == '1':  
 bProfundidad = busquee.BúsquedaEnProfundidad(detallado=**True**)  
 print("Tiempo de ejecución: %f" %

timeit.timeit(functools.partial(bProfundidad.buscar,

problemaPentominos), number=1))  
 # print("Tiempo de ejecución: %f" %

# timeit.repeat(functools.partial(bProfundidad.buscar,

# problemaPentominos), repeat=2, number=1))  
 input("Pulse una tecla para volver al menú principal")  
 **elif** eleccion.lower() == '2':  
 bAnchura = busquee.BúsquedaEnAnchura(detallado=**True**)  
 print("Tiempo de ejecución: %f" %

timeit.timeit(functools.partial(bAnchura.buscar,

problemaPentominos), number=1))  
 # print("Tiempo de ejecución: %f" %

# timeit.repeat(functools.partial(bAnchura.buscar,

# problemaPentominos), repeat=2, number=1))  
 input("Pulse una tecla para volver al menú principal")  
 **elif** eleccion.lower() == '9':  
 volver()  
 **elif** eleccion.lower() == '0':  
 salir()  
 **else**:  
 menu\_acciones['2']()  
  
 input("Pulse una tecla para volver al menú principal")  
 volver()  
 **return**# Menu 2: BUSQUEDA NO INFORMADA  
**def menuBusquedaInformada**():  
 print("\n" \* 100)  
 print("+------------- BUSQUEDA INFORMADA --------------+")  
 print("| Elija el algoritmo que quieres ejecutar: |")  
 print("| 1. B. Informada primero el mejor |")  
 print("| 2. B. Informada A estrella (A\*) |")  
 print("| 3. B. no Informada Óptima |")  
 print("+-----------------------------------------------+")  
 print("| 9. Volver al menú principal |")  
 print("| 0. Quitar |")  
 print("+-----------------------------------------------+")  
  
 eleccion = input(" >> ")  
  
 # Ejecutamos el algoritmo correspondiente  
 **if** eleccion.lower() == '1': # BUSQUEDA PRIMERO EL MEJOR  
 bPrimeroElMejor = busquee.BúsquedaPrimeroElMejor(h1,

detallado=**True**)  
 print("Tiempo de ejecución: %f" %

timeit.timeit(functools.partial(bPrimeroElMejor.buscar,

problemaPentominos), number=1))  
 # print("Tiempo de ejecución: %f" %

# timeit.repeat(functools.partial(bPrimeroElMejor.buscar,

# problemaPentominos), repeat=2, number=1))  
 input("Pulse una tecla para volver al menú principal")  
 **elif** eleccion.lower() == '2': # BUSQUEDA A ESTRELLA (A\*)  
 bAEstrella = busquee.BúsquedaAEstrella(h1, detallado=**True**)  
 print("Tiempo de ejecución: %f" %

timeit.timeit(functools.partial(bAEstrella.buscar,

problemaPentominos), number=1))  
 # print(("Tiempo de ejecución: %f" %

# timeit.repeat(functools.partial(bAEstrella.buscar,

# problemaPentominos), repeat=2, number=1))  
 input("Pulse una tecla para volver al menú principal")  
 **elif** eleccion.lower() == '3': # BUSUQEDA OPTIMA  
 bOptima = busquee.BúsquedaÓptima(detallado=**True**)  
 print("Tiempo de ejecución: %f" %

timeit.timeit(functools.partial(bOptima.buscar,

problemaPentominos), number=1))  
 # print(timeit.repeat("Tiempo de ejecución: %f" %

# functools.partial(bOptima.buscar, problemaPentominos),

# repeat=2, number=1))  
 input("Pulse una tecla para volver al menú principal")  
 **elif** eleccion.lower() == '9':  
 volver()  
 **elif** eleccion.lower() == '0':  
 salir()  
 **else**:  
 menu\_acciones['1']()  
  
 **return**# volver: método que vuelve al menu principal  
**def volver**():  
 menu\_acciones['menuPrincipal']()

# salir: método que finaliza la ejecución del programa  
**def salir**():  
 print("Desarrolladores:"  
 "\n\t· Juan Carlos Utrilla –

https://www.linkedin.com/in/juancarlosutrilla/"  
 "\n\t· Carmen Rosal Malvar –

https://www.linkedin.com/in/carmen-rosal-malvar-

7a91b7130/")  
 sys.exit()  
  
# ==========================  
# DEFINICIONES DE MENU  
# ==========================  
  
menu\_acciones = {  
 'menuPrincipal': menuPrincipal,  
 '1': menuBusquedaInformada,  
 '2': menuBusquedaNoInformada,  
 '9': volver,  
 '0': salir,  
}  
  
# Mostramos el menú de opciones  
menuPrincipal()

## problema\_espacio\_estados.py

Para definir un problema de espacio de estados Pentóminos, haremos uso de las clases de objetos proporcionadas por el script problema\_espacio\_estados.py. Este script es exactamente igual que el script facilitado en la práctica de clase excepto el método es\_estado\_final, que en lugar de verificar si el estado final coincide con alguna lista de posibles estados finales, debe cumplir la condición de que el número de casillas desmarcadas sea menor o igual que 4.

**def es\_estado\_final**(self, estado): # Devuelve si dicho estado es igual a alguno de los  
 **return** estado.numCasillasDesmarcadas() <= 4

## Búsqueda\_espacio\_estados.py

El script que implementa todos los algoritmos de búsqueda es búsqueda\_espacio\_estados.py. Coincide exactamente por el script facilitado en la práctica de clase. Aquí encontramos los siguientes algoritmos:

* **Búsqueda no informada**:
  + **Algoritmo de búsqueda en profundidad**
  + **Algoritmo de búsqueda en anchura**
* **Búsqueda informada**:
  + **Algoritmo de búsqueda primero el mejor**
  + **Algoritmo de búsqueda A estrella (A\*)**
  + **Algoritmo de búsqueda óptima**

# Pruebas realizadas

Para verificar que el algoritmo funciona de forma correcta, hemos probado cada uno de los algoritmos con unos valores de entrada determinados. Las características de los PCs en los que se han hecho las pruebas son los siguientes:

|  |  |
| --- | --- |
| Lenovo Y50-70 | |
| Sistema operativo | Windows 10 Home x64 |
| Procesador | Intel Core i7 – 4700HQ CPU @2.40 GHz |
| Memoria RAM | 12 GB |
| Tarjeta gráfica | Nvidia GeForce 860M |

## Algoritmo de búsqueda en profundidad

|  |  |  |
| --- | --- | --- |
| Nº filas | Nº columnas | Tiempos obtenidos |
| 4 | 4 | 0.089940, 0.091370, 0.215065, 0.229433 |
| 4 | 5 | No encuentra solución |
|  |  |  |