PRACTICA DE LABORATORIO
SISTEMAS INTELIGENTES

CURSO 2020-2021

INTEGRANTES:

- Alejandro Ruiz Aranda
- Jesús Santiyán Reviriego

GRUPO DE TRABAJO:

- C1-8

REPOSITORIO DE TRABAJO

- **https://github.com/alexra99/SI_C1-8.git**

ÍNDICE

1. Introducción del problema y definición como un problema de búsqueda en el espacio de estados.

Definición del problema: Tenemos un tablero en el que hay dibujado un laberinto, este laberinto ha sido generado de forma aleatoria y todas sus casillas tienen conexiones. Además, como es lógico cuenta con una casilla de entrada y una casilla de salida.

La misión consiste en encontrar el camino más óptimo entre la entrada y la salida para poder establecerla como la mejor ruta de escape. Este camino se podrá realizar siguiendo distintas estrategias.

Teniendo en cuenta que hay diferentes tipos de casillas que pueden ser de tierra, agua, hierba y asfalto, según su tipo tendremos más o menos dificultades para atravesar y siempre deberíamos elegir la que más nos convenga siguiendo un criterio, este criterio consiste en elegir la casilla con menor coste.

Si varias casillas tuvieran el mismo coste habría que elegir la que tuviera un número de fila menor. Si coincidiera nos guiaremos por el número de columna y si todo esto no sirviera elegiremos la que tuviera un menor ID de casilla.

Variables: Las variables son las casillas que tenga el laberinto, representadas por su número de fila y de columna. **Dominio:** El dominio serían todas las casillas (FC) que tenga el laberinto (L). $\forall$ FC $\subseteq$ L
Estado: La casilla en la que nos encontramos, con su movimiento (M), número de fila (F) y de columna (C) y el coste del movimiento (MC). Un estado E sería:
E = (M,F,C,MC) | (M $\in$ FC), (F $\in$ FC), (C $\in$ FC) y (MC $\in$ FC)
Función sucesor:
Estado inicial: Casilla de entrada del laberinto
Sucesor: dada una casilla, su sucesores serán las casillas a la que pueda acceder desde la misma, es decir, no hay una pared que lo impida. Desde una casilla solo podrán hacer lo siguientes movimientos en el siguiente orden:

NORTE: decrementar columna
ESTE: incrementar fila
SUR: incrementar columna
OESTE: decrementar fila

Objetivo: Casilla de salida el laberinto

2. Tareas realizadas con la justificación de los criterios y opciones tomadas.

2.1. Tarea 1

A) **<u>Elección del lenguaje de programación:</u>**

Para la resolución de este problema hemos optado por usar el lenguaje de programación Python. Tras haber estado investigando y barajar otras opciones como Java, definitivamente escogimos Python por las siguientes razones

- **Sencillez**: se puede conseguir lo mismo que en otros lenguajes como Java en muchas menos líneas de código.
- **Documentación:** hemos encontrado mucha más documentación sobre este tema y ejemplos en los que apoyarnos escritos en Python.
- **Nuevas funciones:** nunca habíamos trabajado con ficheros json, sin embargo hemos encontrado una forma muy fácil y sencilla de trabajar con ellos usando Python.
- **Ampliar conocimientos:** hasta ahora, la mayoría de las prácticas de las demás asignaturas las habíamos realizado en Java, excepto en redes 2 dónde ya empezamos a dar los primeros en lenguaje Python y nos dio muy buenas sensaciones. Por ello queremos seguir ampliando conocimientos en este lenguaje y consideramos que el hecho de realizar esta práctica en Python nos ayudará a conseguirlo.

B) Generar json del laberinto:

Para generar el json hemos utilizado la librería json de Python y le hemos dado el formato que se pedía.

Pedimos la filas y las columnas por teclado, y para saber los vecinos de cada celda recorremos el tablero y en cada celda con el método *islinked()* vemos si tienen celda unidas en el norte, sur, este y oeste, esto indicará si hay vecino en esa coordenada o no.

```
def save_json(self):
    """Generar el json con el formato dado"""
    output = {
        "rows": 4,
        "cols": 4,
        "max_n": 4,
        "mov": [[-1, 0], [0, 1], [1, 0], [0, -1]],
        "id_mov": ["N", "E", "S", "O"],
        "cells": {}
    }
    for r in range(self.rows):
        for column in range(self.columns):
            key_cell = f'({r}, {column})'
            output["cells"][key_cell] = {
                "value": 0,
                "neighbors": [self.grid[r][column].isLinked(self.grid[r][column].cellNorth),
                              self.grid[r][column].isLinked(self.grid[r][column].cellEast),
                              self.grid[r][column].isLinked(self.grid[r][column].cellSouth),
                              self.grid[r][column].isLinked(self.grid[r][column].cellWest)]
            }

    with open(f'Lab_{self.rows}_{self.columns}.json', 'w') as outfile:
        dump(output, outfile)
```

C) Leer laberinto de un json:

Para leer el json en el constructor del *grid* se crea el *grid* correspondiente con los datos que se toman del json.. Para crear el resto del laberinto se van leyendo lo vecinos que pueda tener una celda y con el método link().

```
else:
    with open(filename) as fdata:
        jsondata = load(fdata)
    self.rows = jsondata['rows']
    self.columns = jsondata['cols']
    self.grid = self.prepare_grid()
    for cell in self.each_cell():
        key = f'{cell}'
        row, col = cell.row, cell.column
        cell.setValue(int(jsondata['cells'][key]['value']))
        cell.cellNorth = self[row - 1, col]
        cell.cellSouth = self[row + 1, col]
        cell.cellWest = self[row, col - 1]
        cell.cellEast = self[row, col + 1]

        if jsondata['cells'][key]['neighbors'][0]:
            cell.link(self[row - 1, col])
        if jsondata['cells'][key]['neighbors'][1]:
            cell.link(self[row, col + 1])
        if jsondata['cells'][key]['neighbors'][2]:
            cell.link(self[row + 1, col])
        if jsondata['cells'][key]['neighbors'][3]:
            cell.link(self[row, col - 1])
```

2.2. Tarea 2

A) Artefacto software nodo:

Para la creación del artefacto nodo hemos hecho una clase nodo donde se le definen todos sus atributos y se imprime en un string todos los atributos del nodo en el mismo orden.

```python
class Node:
    def __init__(self,ID,cost,state,father_id,action,depth,h,value):
        self.ID = int(ID)
        self.cost = int(cost)
        self.state = state
        self.father_id = int(father_id)
        self.action = str(action)
        self.depth = int(depth)
        self.h = int(h)
        self.value = float(value)

    def __str__(self):
        tex = f'[{self.ID}]({self.cost},{self.state},{self.father_id},{self.action},{self.depth},{self.h},{self.value})'
        return tex
```

B) Artefacto software frontera:

La frontera es una lista ordenada de artefactos nodos. También contiene una serie de métodos para trabajar con ella: sacar elementos, insertar, etc.

```python
class Frontier:
    def __init__(self):
        self.items=[]
    def is_empty(self):
        return self.items==[]

    def push(self, data):
        self.items.append(data)

    def insert(self, pos, data):
        self.items.insert(pos, data)

    def pop(self):
        return self.items.pop()

    def remove(self):
        self.items.remove()

    def get_value(self, pos):
        return self.items[pos].value

    def get_size(self):
        return len(self.items);

    def pop_order(self):
        return self.items.pop(0)

    def remove_by_pos(self, pos):
        for index in range(len(self.items)):
            if self.items[index].state.get_tuple() == pos:
                self.items.pop(index)
                break
```

En la frontera podemos destacar el método *push_frontier* que hace un push ordenado de nodos en la misma.
Con este método ordenamos los nodos que entran en la frontera y los ordenamos para que a la hora de sacar la solución ésta venga dada en el orden correcto.
Para la ordenación nos servimos de cuatro criterios:
el primero, sería mirar el campo "value". Si este campo es igual, el siguiente criterio a mirar sería el número de fila. A continuación, en el caso de coincidir, se miraría el número de columna.
Por último, si este también fuera el mismo, se ordenaría por menor ID.

```
def push_frontier(frontier, node):
    if (frontier.is_empty()):
        frontier.push(node)
    else:
        x = frontier.get_size() - 1
        pos = frontier.items[x].state.get_tuple()[0]
        while x >= 0:
            if ((frontier.get_value(x) < node.value)):
                frontier.insert(x+1,node)
                break
            elif(frontier.get_value(x) == node.value):
                if(frontier.items[x].state.get_tuple()[0] < node.state.get_tuple()[0]):
                    frontier.insert(x+1,node)
                    break
                else:
                    if(frontier.items[x].state.get_tuple()[0] == node.state.get_tuple()[0]):
                        if(frontier.items[x].state.get_tuple()[1] < node.state.get_tuple()[1] ):
                            frontier.insert(x+1,node)
                            break
                        elif(frontier.items[x].state.get_tuple()[1] == node.state.get_tuple()[1] ):
                            if (frontier.items(x).state.ID < node.ID):
                                frontier.insert(x+1,node)
                                break
            if (x==0):
                frontier.insert(0,node)
            x= x-1
```

Para sacar el elemento óptimo de la frontera, utilizamos el método *pop_order (definido en el archivo frontier.py)*, que saca el elemento 0 de la lista. En esta posición siempre tenemos dicho elemento.

2.3. Tarea 3

Función expansión: para expandir un nodo, a través de la función expandir primero, obtenemos los vecinos de un estado con la función *neighbors_format*. Una vez tenemos los vecinos, los metemos en una lista llamada sucesores, que está formada por los nodos vecinos. Procedemos a comprobar si fueron visitados o no. Para ellos utilizamos un diccionario que contiene todo las posiciones del laberinto inicializadas a false. Si un estado fue visitado se añade la frontera. Por el contrario si no lo fue, no se hace pues estaríamos tratando un estado que nunca ofrecerá una solución prometedora.

```
def neighbors_format(self):
    """ Return a list of all cells neighboring this cell
    """
    n = []
    if self.isLinked(self.cellNorth):
        n.append(('N', self.cellNorth))
    if self.isLinked(self.cellEast):
        n.append(('E', self.cellEast))
    if self.isLinked(self.cellSouth):
        n.append(('S', self.cellSouth))
    if self.isLinked(self.cellWest):
        n.append(('O', self.cellWest))
    return n
```

```
def expandir(current_node, g, visited, expanded, frontier, all_succesor, strategy, end):
    list_sucesores = get_sucessor_nodes(current_node, g, current_node.state.neighbors_format(), all_succesor, strategy, end)
    if strategy == 2:
        for sucesor in list_sucesores:
            if not visited[f"{sucesor.state}"]:
                push_frontier(frontier, sucesor)
                visited[f"{sucesor.state}"] = True
            elif not expanded[f'{sucesor.state}']:
                frontier.remove_by_pos(sucesor.state.get_tuple())
                push_frontier(frontier, sucesor)
                visited[f"{sucesor.state}"] = True
    else:
        for sucesor in list_sucesores:
            if not visited[f"{sucesor.state}"]:
                push_frontier(frontier, sucesor)
                visited[f"{sucesor.state}"] = True
```

Una vez obtenidos los sucesores procedemos a asignarle los valores necesarios en la función get_succesor_nodes. En esta función podemos destacar que en función de la estrategia que se elija el *value* se calculará de una forma u otra. (La estrategia las representamos con un número del 1 al 5 que se introduce por teclado)

```python
def get_sucessor_nodes(current_node, g, list_sucessors, all_succesor, strategy, end):
    list_nodes = []
    value = 0
    for sucessor in list_sucessors:
        all_succesor.append(sucessor)
        ID = len(all_succesor)
        cost = current_node.cost + g[sucessor[1].get_tuple()].value + 1
        state = sucessor[1]
        father_id = current_node.ID
        depth = current_node.depth + 1
        action = sucessor[0]
        h = abs(sucessor[1].row - end[0]) + abs(sucessor[1].column - end[1])
        node = Node(ID, cost, state, father_id, action, depth, h, value)
        if strategy == 1:
            value = depth
        elif strategy == 2:
                value = 1.0/(depth+1)
        elif strategy == 3:
            value = cost
        elif strategy == 4:
            value = abs(sucessor[1].row - end[0]) + abs(sucessor[1].column - end[1])
        elif strategy == 5:
            value = cost + h
        node.value = value
        #output = f'[{node.ID}]({node.cost},{node.state},{node.father_id},{node.action},{node.depth},{node.h},{Decimal(node.value)})'
        #print (f"* {output}")
        list_nodes.append(node)
    return list_nodes
```

3. Manual de usuario

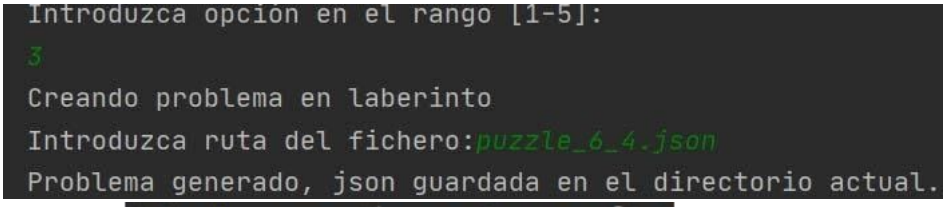
En cuanto iniciamos el programa nos aparece el menú principal indicando que introducimos una de las 5 opciones que puede manejar el programa siendo la número cinco una opción de salida con la que cerraremos el programa.

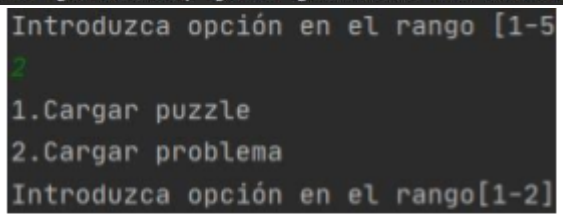
```
-----MAZE-SOLVER-----
1. Generar laberinto
2. Cargar laberinto
3. Generar Problema
4. Resolver Problema
5. Salir
Introduzca opción en el rango [1-5]:
```

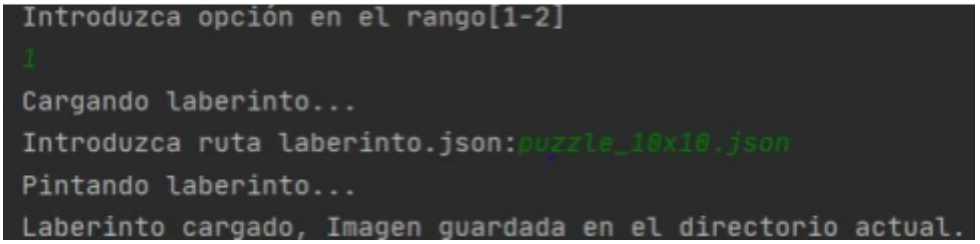
Si introducimos la primera opción podremos generar un laberinto que para ello nos pedirá el número de filas y de columnas. Una vez introducido este generará un fichero json con el laberinto y lo guardará en el directorio raíz.

```
Introduzca opción en el rango [1-5]:
1
Generando laberinto...
Introduzca el número de filas: 4
Introduzca el número de columnas: 6
```

Si elegimos la segunda opción nos aparecerá un submenú. La primera opción del submenú nos generará una imagen del laberinto que le pasemos desde un fichero json. La segunda opción nos cargará un problema de un determinado laberinto que ya esté creado.

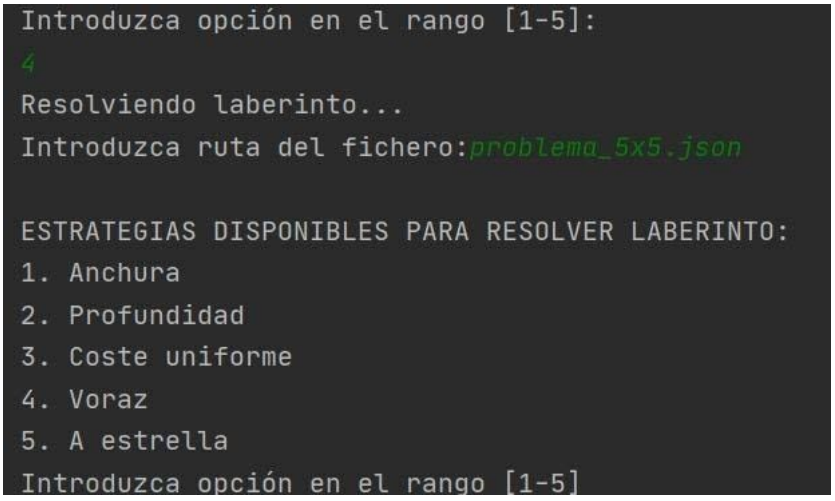






La tercera opción nos generará un problema a partir de un laberinto ya creado con anterioridad.

La cuarta y última opción es la que nos permite resolver un problema por las distintas estrategias especificadas. Una vez señalemos la opción se nos pedirá introducir el archivo del laberinto que queremos resolver. A continuación aparecerá un submenú que nos deja elegir entre la estrategia que queremos utilizar.



5. Valoración personal y del equipo respecto a la práctica.

Alejandro Ruiz Aranda:

Creo que ha sido una práctica compleja ya que no teníamos mucha idea de como hacerla y estábamos un poco perdidos pero al final creo que hemos conseguido aprender lo que se intenta enseñar en la asignatura. En cuanto al equipo creo que ambos hemos hecho un buen trabajo y le hemos dedicado bastantes horas, no creo que haya mucho que resaltar.

Jesús Santiyán Reviriego:

Es una práctica difícil y en la que en muchos momentos no sabíamos cómo avanzar, quizá no sea la mejor metodología la que se está usando, pero desde luego creo que ha sido efectiva ya que hemos podido aprender todo lo que se enseña en la asignatura tanto a la hora de plantear los problemas como en resolverlos. En cuanto al equipo, los dos hemos trabajado bastante pero quiero resaltar a mi compañero Alejandro que ha tenido más constancia y sabido seguir adelante cuando el trabajo se complicaba.