UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Documentación
Proyecto de Laboratorio

Jaime Camacho García

Esther Camacho Caro

Jorge Herrero Úbeda

Asignatura: Sistemas Inteligentes

Grupo de Trabajo: LAB-BC1-10

Titulación: Grado en Ingeniería Informática

Fecha: 2 de junio de 2022

Contenido

Objetivo del trabajo de laboratorio

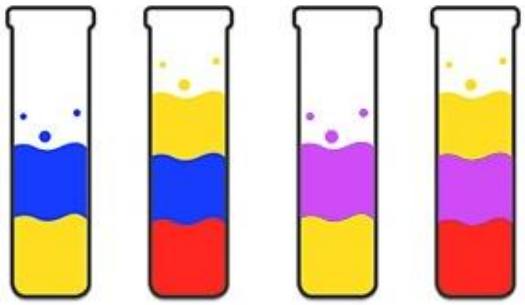
El objetivo general del laboratorio consiste en implementar un programa que simulo el problema de Water Sort Puzzle, empleando los algoritmos de búsqueda, tanto informada como no informada, vistos en la parte teórica de la asignatura:

- Búsqueda en Anchura
- Búsqueda en Profundidad
- Búsqueda en Coste Uniforme
- Búsqueda Voraz
- Búsqueda A*

El trabajo esta dividido en 4 tareas progresivas, siendo la última la que tendrá la funcionalidad completa.

El programa recibirá como entrada un fichero en formato JSON que contendrá el ID del problema, el tamaño de las botellas y su estado inicial.

El resultado de este será un archivo con el camino hacia la solución.



Tarea 1

Descripción

Los objetivos de esta tarea son definir una representación textual del estado a partir de un JSON y definir la acción y si es posible.

La función a desarrollar es un artefacto software que represente un estado con las siguientes funcionalidades:

Crear y construir el estado a partir de una representación con JSON, clonar el artefacto, implementar la funcionalidad ***ES_AccionPosible()*** que devuelva True o False si la acción es posible e implementar la funcionalidad ***Accion()*** que lleva a cabo la acción de mover el líquido y devuelve el nuevo estado si es posible.

Cómo se ha resuelto

Creamos la clase ***Estado(),*** que definirá un estado específico de las botellas y tendrá los siguientes métodos:

```
def __init__(self, listOfBottles, capacidad):
    self.listOfBottles = listOfBottles
    self.capacidad = capacidad

def amountBotella(self, botella):
    n=0
    for i in botella:
        n+=i[1]
    return n

def ES_AccionPosible(self, Botella_origen, Botella_destino, Cantidad):
    if (self.amountBotella(Botella_origen)<Cantidad or self.capacidad-self.amountBotella(Botella_destino)<Cantidad):
        return False
    else:
        return True
```

Creamos el constructor, un método que devuelva la cantidad de líquido en la botella para modularizar y facilitar el código y el método ***ES_AccionPosible()*** que comprueba si la acción de mover líquidos es posible en función de los requisitos especificados.

Creamos el método **Acción(),** que se encarga de pasar el líquido de una botella a la otra haciendo la comprobación pertinente con el método ***ES_AccionPosible().***

```
def Accion(self, Botella_origen, Botella_destino, Cantidad):
    if (self.ES_AccionPosible(Botella_origen, Botella_destino, Cantidad)):
        contador=0

        while contador<Cantidad:

            if Cantidad-contador >= Botella_origen[0][1]:

                contador+=Botella_origen[0][1]
                Botella_destino.insert(0, Botella_origen.pop(0))

            else:

                contador+=Botella_origen[0][1]-(Cantidad-contador)
                Botella_destino.insert(0, [Botella_origen[0][0], Botella_origen[0][1]-(Cantidad-contador)])
                Botella_origen[0][1]-=Cantidad-contador

        n=0
        while n+1<len(Botella_destino):
            if Botella_destino[n][0]==Botella_destino[n+1][0]:
                Botella_destino[n][1]+=Botella_destino[n+1][1]
                Botella_destino.pop(n+1)
            else:
                n+=1

        return self

    else:
        return None
```

Manual de usuario

Para probar la ejecución de la Tarea 1 hicimos una pequeña ejecución de comprobación.

El programa pide por pantalla una estructura JSON que sería el estado inicial, luego nos pide que indiquemos la botella 1, la botella 2 y la cantidad a traspasar. Entonces se ejecutará la acción y se mostrará por pantalla el nuevo estado.

Ejemplo de ejecución:

```
Introduzca estructura JSON:[[[0,7],[1,3]],[[0,2],[2,5],[1,3]],[[0,1],[2,4],[1,1],[2,1],[1,3]],[],[]]
Indique botella numero 1 (0-4): 1
Indique botella numero 2 (0-4): 3
Indique cantidad de liquido a mover: 7
[[[0, 7], [1, 3]], [[1, 3]], [[0, 1], [2, 4], [1, 1], [2, 1], [1, 3]], [[2, 5], [0, 2]], []]
```

Tarea 2

Descripción

El objetivo de esta tarea es definir, crear e importar y exportar un problema water puzzle, que viene definido por el espacio de estados, el estado inicial y la función objetivo.

Para definir el espacio de estados ya tenemos definido lo que es un estado de la tarea anterior, luego únicamente nos faltará definir la función sucesores de un estado.

Los sucesores(estado) son una lista ordenada que almacena los sucesores válidos de un estado. El sucesor es una tupla de tres elementos (*accion*:tupla, *nuevo estado*:estado, *costo*:float).

Como la lista de sucesores debe ser ordenada, el orden establecido será el marcado por su acción: primero los de id's <botellas_origen> más pequeña, a igual <botella_origen> los id's de la <botella_destino> más pequeña, y a igual <botella destino> los de cantidad más pequeña.

El estado inicial será el estado del que partimos, contará de un conjunto de botellas llenas con distintos tipos de líquidos y un conjunto de botellas vacías.

La función objetivo(estado) será verdad si y sólo si todas las botellas con líquido en <estado> están llenas con un único tipo de líquido.

La persistencia de un problema se plantea mediante un fichero de texto tipo json (id.json) con la siguiente estructura y elementos en un único string:

{ 'id':<String ID del problema> , 'BottleSize':<int>, 'InitState':<json state>}

Donde 'BottleSize' es el tamaño máximo que tendrán las botellas de los posibles estados del problema.

Cómo se ha resuelto

Creamos la clase Problema con su constructor y la función objetivo que determina si el estado cumple con el objetivo.

```
class Problema():

    def __init__(self, initState):
        self.initState = initState

    def objetivo(self, estado):
        cont = 0
        for i in range(len(estado.listOfBottles)):
            if estado.listOfBottles[i] != []:
                if estado.listOfBottles[i][0][1] == estado.capacidad or not estado.listOfBottles[i]:
                    cont+=1

            else:
                cont+=1

        if cont == len(estado.listOfBottles):
            return True
        else:
            return False
```

Creamos el método Sucesores que a partir de un estado crea los estados sucesores con los posibles movimientos de dicho estado.

```
def sucesores(estado):
    sucesores=[]

    for i in range(len(estado.listOfBottles)):
        for j in range(len(estado.listOfBottles)):
            if estado.listOfBottles[i] and i!=j and estado.ES_AccionPosible(estado.listOfBottles[i], estado.listOfBottles[j], estado.listOfBottles[i][0][1]) and (not estado.listOfBo

                estadoAux = copy.deepcopy(estado)

                #cant = copy.deepcopy(estadoAux.listOfBottles[i][0][1])
                cant = estado.listOfBottles[i][0][1]

                estadoAux.Accion(estadoAux.listOfBottles[i], estadoAux.listOfBottles[j], cant)

                #sucesores.append(((i, j, cant), copy.deepcopy(estadoAux), 1))
                sucesores.append(((i, j, cant), estadoAux, 1))

    return sucesores
```

Tarea 3

Descripción

Los objetivos serán implementar los artefactos para el árbol de búsqueda, implementar el algoritmo de búsqueda en el espacio de estados, resolver el problema con las estrategias siguientes: anchura, costo uniforme y profundidad acotada; obtener resultados idénticos a los de los ficheros de ejemplo.

Para la realización del árbol vamos a especificar dos artefactos software: Nodo del árbol de búsqueda y Frontera (colección ordenada de nodos hoja).

Especificaciones del artefacto software de nodo

La información mínima que debe tener el nodo es la siguiente: ID identificador único del nodo que comenzará con 0 e irá incrementándose conforme se vayan creando nuevos nodos, ESTADO acceso al estado del espacio de estados en dicho nodo, VALOR valor del nodo según la estrategia seleccionada de tipo REAL, PROFUNDIDAD o nivel del nodo (en el nodo raíz la profundidad es 0), COSTO costo acumulado por el camino hasta dicho nodo, HEURISTICA valor de la heurística (por ahora un float con un valor random), ACCIÓN movimiento que ha generado el nodo y PADRE un acceso al nodo padre.

La representación mediante una cadena de caracteres de un nodo será (para tareas de depuración, representación de la solución, etc.).

Especificaciones del artefacto software frontera:

La frontera tiene que ser una colección ordenada de artefactos nodos.

En la documentación se debe justificar numéricamente la elección de la estructura de datos asociada a la Frontera. Los parámetros que permitirán una comparación entre estructuras

serán: máximo número de nodos que se pueden almacenar, velocidad mínima de inserción, velocidad máxima y velocidad media.

El criterio de orden es: 1º Atributo Valor del nodo, 2º Identificador único del nodo.

Las operaciones a implementar son: insertar un nodo n (PUSH n) y obtener el primer nodo eliminándolo de la colección (POP n).

Las inserciones han de ser ordenadas para mantener siempre el artefacto ordenado.

Comprobaciones de la implementación del árbol de búsqueda:

Lectura de un problema (JSON).

Generación de un artefacto nodo a partir del estado inicial e inserción del mismo en la Frontera con un campo Valor generado aleatoriamente en un intervalo [0,1000]

Repetir hasta agotar los recursos del computador: Extracción del primer elemento de la frontera e inserción de sus sucesores en la Frontera con un campo Valor para cada nodo generado aleatoriamente en un intervalo [0,1000].

Comprobar que los nodos se insertan en la Frontera de manera ascendente según su campo Valor. Chequear si en las colisiones (dos nodos con el mismo campo Valor) se cumple el criterio de orden fijado.

Comprobar que se realiza una salida controlada cuando se agotan los recursos del computador.

Implementar el algoritmo de búsqueda en el Espacio de Estados

Tomar el pseudocódigo del algoritmo suministrado en el Tema 2. Se establecerá la poda de nodos del árbol mediante una estructura (visitados) que pueda almacenar un conjunto de estados y que cuente con los métodos: crear_vacio, insertar(estado) y la función pertenece(estado).
Aunque a nivel teórico sólo se ha estudiado poner un límite de profundidad (l) al árbol en la búsqueda en profundidad acotada, este se va a establecer para todas las estrategias y será igual a 1.000.000.
La solución será la lista de nodos del camino solución y se debe exportar a un fichero cuyo nombre será el id_estrategia.txt donde id se recupera del problema en el fichero JSON y estrategia es la seleccionada por el usuario.

Resolver el problema con las estrategias siguientes: Anchura, Costo Uniforme, Profundidad Acotada.

Estrategias: Al inicio de la ejecución del programa el usuario seleccionará la estrategia de búsqueda que se va a utilizar. Por ejemplo, si selecciona anchura esta decisión se almacenará

en una cadena de caracteres como 'BREADTH'. Para esta y el resto de las estrategias se especifica el valor de la cadena en la siguiente tabla.

Cómo se ha resuelto

Implementamos las clases Nodo Frontera y Visitados con sus respectivos constructores.

Para Nodo tenemos el método ***_lt_*** que compara el valor ***self*** de un nodo con otro para devolver el menor.

Para frontera tenemos el método ***insertar*** que inserta y ordena un nodo en la frontera y el método obtener que devuelve el último nodo.

Para Visitados tenemos el método ***insertar*** para insertar un nodo y el método pertenece que comprueba si un estado se encuentra dentro de Visitados.

```python
class Nodo():
    def __init__(self, id, costo, estado, padre, accion, profundidad, heuristica, valor):
        self.id = id
        self.costo = costo
        self.estado = estado
        self.padre = padre
        self.accion = accion
        self.profundidad = profundidad
        self.heuristica = heuristica
        self.valor = valor

    def __lt__(self, other):
        return self.valor < other.valor or (self.valor == other.valor and self.id < other.id)
```

```python
class Frontera():
    def __init__(self):
        self.nodosFrontera = []

    def insertar(self, nodo):
        bisect.insort(self.nodosFrontera, nodo)
        #self.nodosFrontera.append(nodo)
        #self.nodosFrontera.insert(0, nodo)
        #       self.nodosFrontera.sort(key = lambda x: (x.valor, x.id))

    def obtener(self):
        return self.nodosFrontera.pop(0)
```

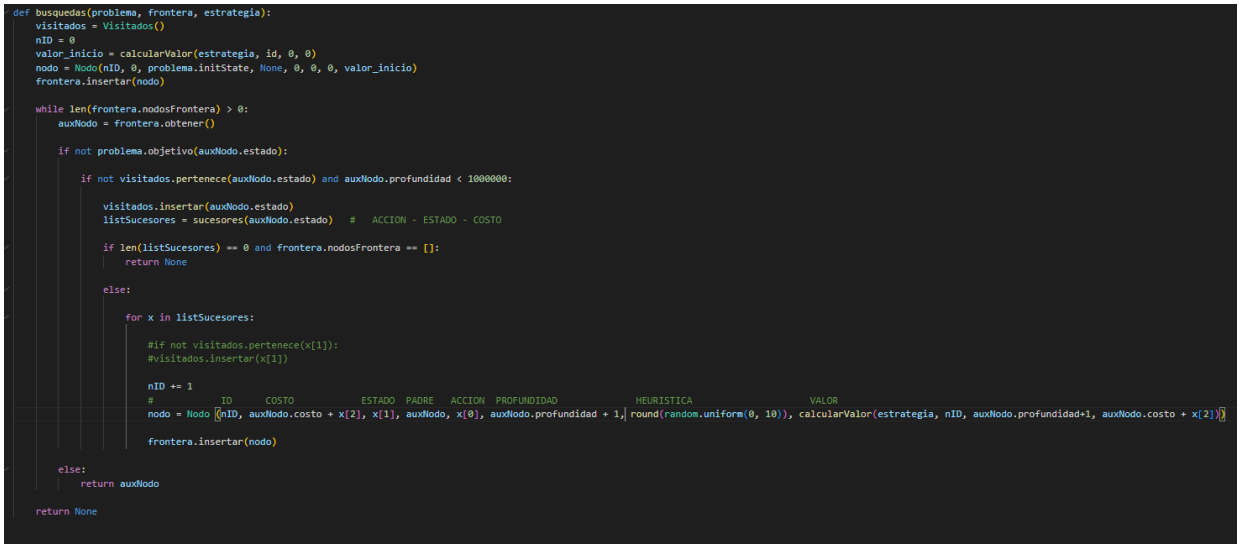
```python
class Visitados():
    def __init__(self):
        self.estadosVisitados = []

    def insertar(self, nodo):
        self.estadosVisitados.append(nodo)

    def pertenece(self, estado):
        for x in self.estadosVisitados:
            if x == estado:
                return True
        return False
```

La función ***búsquedas()*** es la encargada de ejecutar la búsqueda dentro del árbol.

Insertamos el primer nodo y comenzamos la búsqueda con un while. Para cada caso comprobamos si el nodo es Objetivo y si el nodo ya ha sido visitado o se ha excedido la profundidad máxima. Si no es el caso insertamos el estado en visitados, obtenemos los sucesores y si sucesores[] no está vacía, los recorremos para insertarlos en la frontera y seguir la búsqueda.



```
def busquedas(problema, frontera, estrategia):
    visitados = Visitados()
    nID = 0
    valor_inicio = calcularValor(estrategia, id, 0, 0)
    nodo = Nodo(nID, 0, problema.initState, None, 0, 0, 0, valor_inicio)
    frontera.insertar(nodo)

    while len(frontera.nodosFrontera) > 0:
        auxNodo = frontera.obtener()

        if not problema.objetivo(auxNodo.estado):

            if not visitados.pertenece(auxNodo.estado) and auxNodo.profundidad < 1000000:

                visitados.insertar(auxNodo.estado)
                listSucesores = sucesores(auxNodo.estado)   #   ACCION - ESTADO - COSTO

                if len(listSucesores) == 0 and frontera.nodosFrontera == []:
                    return None

                else:

                    for x in listSucesores:

                        #if not visitados.pertenece(x[1]):
                        #visitados.insertar(x[1])

                        nID += 1
                        #                  ID      COSTO            ESTADO  PADRE   ACCION  PROFUNDIDAD             HEURISTICA                  VALOR
                        nodo = Nodo (nID, auxNodo.costo + x[2], x[1], auxNodo, x[0], auxNodo.profundidad + 1, round(random.uniform(0, 10)), calcularValor(estrategia, nID, auxNodo.profundidad+1, auxNodo.costo + x[2]))

                        frontera.insertar(nodo)

        else:
            return auxNodo

    return None
```

La función ***calcularValor()*** calcula el valor de cada nodo en la función búsquedas dependiendo de la estrategia de búsqueda que se esté siguiendo.

```
def calcularValor(estrategia, id, prof, costo):
    if estrategia=="BREADTH":
        return prof
    elif estrategia=="DEPTH":
        return 1/(prof+1)
    elif estrategia=="UNIFORM":
        return costo
```

Dificultades encontradas

Una de las dificultades con la que nos hemos encontrado en esta tarea, junto con la tarea posterior, fue la representación de la solución al codificarla con md5.

Una vez completada la tarea, veíamos que su comportamiento era correcto en todos los sentidos, sin embargo, si comparábamos la "traducción" a md5 por parte del estado de botellas, esta codificación no correspondía a la proporcionada en las soluciones por parte de los profesores.

Es por ello por lo que durante el desarrollo de esta tarea hemos probado numerosas formas de codificar este dato, hasta que nos dimos cuenta del error, y es que al codificar en md5, este tiene en cuenta los espacios en blanco.

Una codificación en md5 de un estado representado con espacios:

[[[0, 2], [2, 1], [3, 1]], ...]

No es la misma para un estado representado sin estados, aunque a simple vista sean el mismo:

[[[0,2],[2,1],[3,1]],...]

Por ello, para evitar este error, hemos creado una pequeña función que elimina estos espacios, remplazándolos:

```
def eliminarEspacios(cad):
    return cad.replace(" ", "")
```

De esta manera, al codificar los estados, estos eran introducidos previamente a la función:

```
hashlib.md5(eliminarEspacios(str(nodo.estado.listOfBottles)).encode()).hexdigest()
```

Manual de usuario

El programa pregunta al usuario la estrategia a seguir entre las 3 opciones disponibles. Una vez introducida, el programa lleva a cabo la búsqueda y muestra por pantalla el camino de nodos hasta alcanzar a solución.

Ejemplo de ejecución:

```
ES/Practicas/LAB-BC1-10/Tarea3.py"º CURSO\INTELIGENTES\Practicas\LAB-BC1-10>
Estrategias:
 - Anchura
 - Profundidad Acotada
 - Coste uniforme

Seleccione la estrategia deseada: anchura
[0] [0, 40b5abda7ba5f5c191c7fe621c6e85b1, None, None, 0, 0, 0]
[1] [1, e123755010df6104e367f26122c1b1f9, 0, (0, 5, 2), 1, 2, 1]
[12] [2, 1771c37572ca1e241aa21f076fc6f51e, 1, (1, 6, 3), 2, 5, 2]
[81] [3, a217548173e6c437a782a6a83ce5ab34, 12, (0, 1, 1), 3, 0, 3]
[204] [4, e2f4017a60de1def4a2be648aebdaf4e, 81, (2, 1, 2), 4, 1, 4]
[403] [5, 7bb8b47111123d03c5fbc078bed2913e, 204, (0, 2, 1), 5, 10, 5]
[770] [6, 4c353ea4c178aa8e549e99ec5cbeb841, 403, (3, 5, 2), 6, 3, 6]
[1455] [7, 4b95af3057fa60d8fd426da111814ce9, 770, (3, 6, 1), 7, 8, 7]
[2677] [8, 8adba6e6b3195af40a6e2ef246837594, 1455, (4, 3, 2), 8, 9, 8]
[4941] [9, ff0542ab1caad893e6ddb728d2c37452, 2677, (2, 4, 2), 9, 5, 9]
[9411] [10, c449db352b22540f4e00d0a4f525191c, 4941, (2, 3, 1), 10, 8, 10]

Tiempo en resolver el problema (en segundos):
0:00:07.347999
PS C:\Users\Usuario\Desktop\UCLM\3º CURSO\INTELIGENTES\Practicas\LAB-BC1-10>
```

Tarea 4

Descripción

Los objetivos de la última tarea son definir una función heurística h(e) e implementar las estrategias de búsqueda Greedy y A*.

Para resolver el ejercicio hemos considerado como posible función heurística la proporcionada en el enunciado: **contabilizar el número de líquidos no mezclados y las botellas vacías que tenemos en un estado, y compararlo con el número de botellas totales del problema**.

Si en un estado esos dos números son iguales (***h(e)=0***) la función objetivo del mismo devolverá Verdad (True), en caso contrario su valor será Falso (False), y conforme más diferentes sean mayor será el coste de llegar a un estado que satisfaga la función objetivo.

Para evitar que un líquido quede repartido entre botellas; cuando contabilicemos el número de líquidos que hay en una botella añadiremos 1 si el tipo del primer liquido ya ha sido alguna vez el primero de alguna de las botellas ya contabilizadas.

Cómo se ha resuelto

Empezamos la tarea 4 ampliando la función **calcularValor()** para que también acepte las estrategias Greedy y A*.

```
def calcularValor(estrategia, id, prof, costo, heuristica):
    if estrategia=="BREADTH":
        return prof
    elif estrategia=="DEPTH":
        return 1/(prof+1)
    elif estrategia=="UNIFORM":
        return costo
    elif estrategia=="GREEDY":
        return heuristica
    elif estrategia=="A":
        return heuristica+costo
```

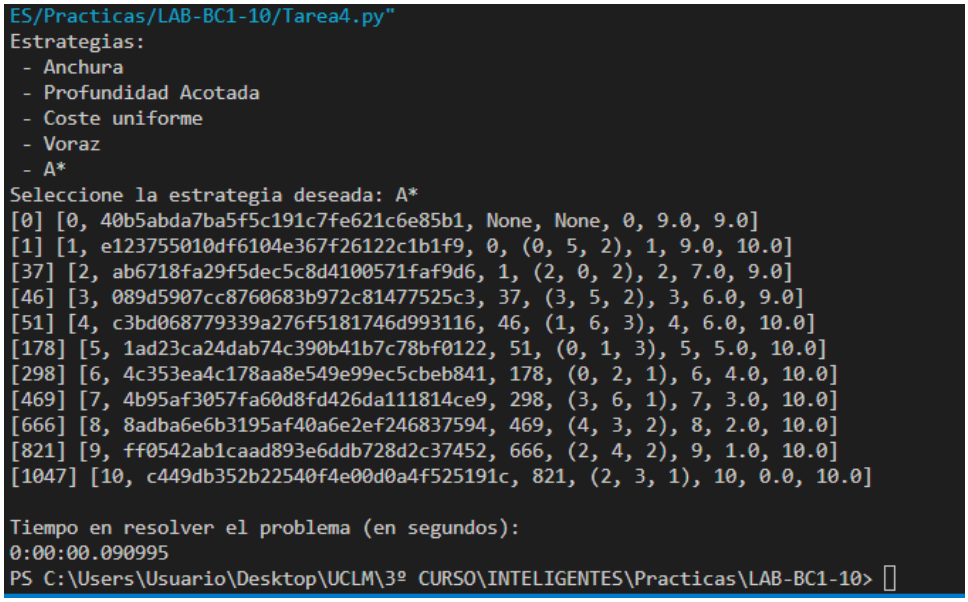
Creamos el método **heuristicaFunción()** que calcula la heurística del estado en base a la pedida en el enunciado.

```
def heuristicaFuncion(estado):
    tiposVistos = set()
    cant = 0
    h = 0.0
    for i in range(len(estado.listOfBottles)):
        botella = estado.listOfBottles[i]
        cant = len(botella)
        if cant == 0:
            h+=1
        else:
            if botella[0][0] in tiposVistos: # Para evitar liquido repartido, +1 si el primero ya ha sido visto
                h+=1
            else:
                tiposVistos.add(botella[0][0])
            h+=cant
    return h - len(estado.listOfBottles)
```

Manual de usuario

Al igual que en la tarea 3, se pide al usuario que introduzca la estrategia a seguir y cuando la obtiene, ejecuta la búsqueda y muestra el resultado por pantalla.

Ejemplo de ejecución:



Opinión personal

- **Jorge Herrero Úbeda:**

Mi opinión con respecto a la práctica es que hasta la tarea 2 era bastante trivial, luego a partir de la tarea 3 se complicaba a raíz de la búsqueda, pero dedicándole tiempo no ha sido tan complicado. El sistema a replicar, al ser un juego bastante común y visual, ha facilitado el planteamiento y el desarrollo del mismo.

- **Esther Camacho Caro:**

En cuanto al problema del laboratorio, al principio me costó asimilar los conceptos a la hora de trabajar y representar los diferentes estados de las botellas, pero ahora los comprendo mucho mejor. Las dos últimas tareas me han ayudado a reforzar también los tipos de búsqueda y como el resultado puede variar según la estrategia elegida.

En cuanto a la metodología, considero que el ir trabajando con tareas progresivas es mejor que una única tarea, ya que, si el objetivo final de este proyecto (Tarea 4), se nos hubiera pedido desde un inicio, tal vez nos habría costado más realizarlo, no sabiendo por dónde empezar.

- **Jaime Camacho García:**

Este proyecto de laboratorio me ha resultado muy interesante y entretenido, ya que el juego del water puzzle me ha parecido muy buen ejemplo para poder comprender todo lo referente a la asignatura de sistemas inteligentes, como son los estados, acciones, problemas o estrategias, a su vez, de por si la asignatura y lo que podemos hacer con los conocimientos adquiridos, también me han interesado mucho debido a los diferentes ámbitos en los que podemos usar estos conocimientos y habilidades.

Reparto del trabajo

Miembro del equipo	**Categoría**
Jaime Camacho García	(C) Como todo el mundo
Esther Camacho Caro	(C) Como todo el mundo
Jorge Herrero Úbeda	(C) Como todo el mundo

Hemos querido trabajar todos por igual, ayudándonos entre nosotros cuando alguno podía "atascarse" a la hora de resolver o entender algún problema o concepto.