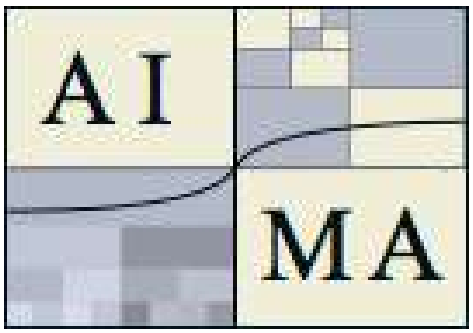
Universidad Complutense de Madrid.
Facultad de Ingeniería Informática
Inteligencia Artificial 1.



Práctica 2.
Toma de contacto con AIMA.

Grupo 5:
Frederick Ernesto Borges Noronha
Victor Manuel Cavero Garcia

PARTE I. REPRESENTACION DEL PROBLEMA.

Puzzle de Ocho:

Se ha definido una clase Ocho_Puzzle que hereda de la clase Problem de la librería de AIMA, donde se han definido las funciones __init__ (constructora), actions. result y h.

Como ejercicio de la práctica se han tenido que implementar actions y result. En actions se han definido los posibles movimientos que se pueden hacer:

```
        if pos_hueco not in (0,1,2):
            accs.append("Mover hueco arriba")

        ### EJERCICIO 1.1. COMPLETA LA DEFINICIÓN DE LOS OPERADORES.
        if pos_hueco not in (6,7,8):
            accs.append("Mover hueco abajo")

        if pos_hueco not in (2,5,8):
            accs.append("Mover hueco derecha")

        if pos_hueco not in (0,3,6):
            accs.append("Mover hueco izquierda")
```

Y en la función result se han definido los resultados de aplicar las operaciones anteriores (“Mover hueco arriba”, “Mover hueco abajo”, “Mover hueco derecha”, “Mover hueco izquierda”):

```
        if accion == "Mover hueco arriba":
            l[pos_hueco] = l[pos_hueco-3]
            l[pos_hueco-3] = 0

        ### EJERCICIO 1.2. COMPLETA LA DEFINICIÓN DE LOS OPERADORES.
        if accion == "Mover hueco abajo":
            l[pos_hueco] = l[pos_hueco+3]
            l[pos_hueco+3] = 0

        if accion == "Mover hueco izquierda":
            l[pos_hueco] = l[pos_hueco-1]
            l[pos_hueco-1] = 0

        if accion == "Mover hueco derecha":
            l[pos_hueco] = l[pos_hueco+1]
            l[pos_hueco+1] = 0
```

En el apartado 1.3 se debía comprobar la ejecución de algunas funciones una vez se definiera la clase, viendo así que el programa funciona correctamente y además es capaz de saber cuándo realizamos un movimiento que no es posible.

PARTE II. EXPERIMENTACIÓN CON LOS ALGORITMOS IMPLEMENTADOS.

Se utilizaron los algoritmos de búsqueda en anchura y en profundidad para buscar las soluciones al puzle de 8 para el siguiente ejemplo:

2	4	3
1	5	6
7	8	H

Para los algoritmos de búsqueda ciega podemos mirar la siguiente tabla comparativa de tiempo y longitud de la solución:

Algoritmo	Tiempo	Longitud de la solución
Búsqueda en anchura (Sin control de repetidos)	19.5 ms	8
Búsqueda en profundidad (Con control de repetidos)	40min 30s	28842
Búsqueda en anchura (Con control de repetidos)	2.91ms	8

Como podemos observar en la tabla anterior el tiempo de ejecución en el algoritmo de búsqueda ciega en profundidad es claramente el peor algoritmo cuando las posibilidades de movimientos son grandes, por otra parte, los tiempos para la búsqueda por anchura son mucho mejores. Si analizamos por que los tiempos son tan elevados en búsqueda por profundidad vemos que es porque encuentra una solución realizando 28842 movimientos más la búsqueda por profundidad consigue una solución con 8 movimientos.

También se deben definir heurísticas para trabajar con algoritmos como primero el mejor y A*, las cuales podemos definir como:

```
# Heuristicas para el 8 Puzzle

def linear(node):
    #goal = node.state.goal
    goal = (1,2,3,4,5,6,7,8,0)
    state = node.state

    count = 0
    for value in state:
        if (value != goal[state.index(value)]):
            count += 1

    return count

def manhattan(node):
    state = node.state
    goal = (1,2,3,4,5,6,7,8,0)

    mhd = 0
    for value in state:
        mhd += abs(value - goal[state.index(value)])

    return mhd

def sqrt_manhattan(node):
    state = node.state
    mhd = manhattan(node)
    return math.sqrt(mhd)

def max_heuristic(node):
    score1 = manhattan(node)
    score2 = linear(node)
    return max(score1, score2)
```

Para el ejercicio 4 calcularemos los tiempos de cada uno de los algoritmos de búsqueda con heurística:

Heurística	Coste Uniforme	Primero el Mejor	A*
Sin heurística	13 ms	-	13.7 ms
Linear	-	136 ms	385 µs
Manhattan	-	1.22 ms	388 µs
Sqrt Manhattan	-	1.17 ms	3.26 ms
Max heuristic	-	1.33 ms	423 µs

Como podemos observar, si realizamos una búsqueda de “Coste Uniforme” es mucho mejor que si realizamos una búsqueda “Primero el Mejor” con una heurística mala, sin embargo, si tenemos una heurística que se acerque a la realidad el algoritmo “Primero el Mejor” da una solución en un tiempo mucho más corto (para este ejemplo casi 13 veces mejor) que el algoritmo de “Coste Uniforme”. Por otra parte, también observamos que el algoritmo “A*” es mucho más rápido en encontrar soluciones si se le da una heurística correcta, ya que reduce el tiempo a menos de 1 ms, pero si se le da una heurística mala, o no se le da ninguna puede ser más lento que el “Primero el Mejor” o incluso, peor que el “Coste Uniforme”.

PARTE III. CALCULAR ESTADÍSTICAS SOBRE LA EJECUCIÓN DE LOS ALGORITMOS PARA RESOLUCIÓN DE PROBLEMAS.

Se ha definido una nueva clase “Problema_con_Analizados” que es capaz de saber si un estado tiene solución y también se ha añadido una nueva función “resuelve_ocho_puzzle” que, dado un estado inicial, un algoritmo de búsqueda y una heurística da una solución los problemas o indica si el problema no tiene solución. Para probar esta función se definen los siguientes problemas y E1, E2, E3 y E4 definidos de la siente forma:

E1

2	1	3
4	8	6
7	H	6

E2

1	H	3
4	8	6
7	2	5

E3

4	5	6
1	H	3
7	8	2

E4

1	2	3
H	5	6
4	7	8

Para los algoritmos de “Búsqueda en Anchura” y “Búsqueda en Profundidad” no se da ningún resultado de tiempos ya que al ser de búsqueda ciega pueden tardar mucho tiempo en explorar todos los nodos posibles antes de llegar a una solución, sin embargo, para todos los demás podemos observar que en general el algoritmo “A*” tiene mejores tiempos que los demás algoritmos, cumpliéndose así la teoría.

Algoritmo	E1	E2	E3	E4
Anchura	L=¿? T=¿? NA=¿?	L=¿? T=¿? NA=¿?	L=¿? T=¿? NA=¿?	L=¿? T=¿? NA=¿?
Profundidad	L=¿? T=¿? NA=¿?	L=¿? T=¿? NA=¿?	L=¿? T=¿? NA=¿?	L=¿? T=¿? NA=¿?
Coste Uniforme	L=17 T=1min 19s NA=14092	L=11 T=286 ms NA=870	L=20 T=16min 34s NA=48428	L=3 T=547 µs NA=4
Primero el mejor (Linear)	L=39 T=15 ms NA=173	L=33 T=4.42 ms NA=78	L=76 T=210 ms NA=748	L=3 T=435 µs NA=4
Primero el mejor (Manhattan)	L=83 T=229 ms NA=840	L=93 T=245 ms NA=852	L=140 T=17.2 s NA=7599	L=3 T=436 µs NA=5
A* (Linear)	L=17 T=271 ms NA=873	L=11 T=4.4 ms NA=77	L=20 T=3.62 s NA=3377	L=3 T=440 µs NA=4
A* (Manhattan)	L=21 T=1.88 s NA=2349	L=11 T=47.1 ms NA=336	L=20 T=16.9 s NA=7553	L=3 T=451 µs NA=5

Para dar solución al ejercicio 6 presente en el notebook, hemos definido una heurística más informada sobre el problema del puzzle de ocho la cual hemos llamado “h3”, esta heurística obtiene sumando a la distancia manhattan una componente que sirve para cuantificar la secuencialidad en las casillas del tablero, al recorrerlo en el sentido de las agujas del reloj (esta componente se incrementa en uno por cada elemento que rompe la secuencia). Podemos ver a continuación la definición de dicha heurística:

```
def h3(node):
    mhd = manhattan(node)
    sec = (0,1,2,5,8,7,6,3,0)
    goalsec = (1,2,3,6,0,8,7,4,1)
    h3 = mhd

    for x in range (1,8):
        if(node.state[sec[x]] in goalsec):
            if (goalsec[goalsec.index(node.state[sec[x]]) - 1] !=
node.state[sec[x-1]]): h3 +=1

        #sumo uno a la heuristica manhattan por cada elemento que
rompe la secuencialidad

    return h3
```

Con esta nueva heurística la tabla del apartado anterior quedaría tal que:

Algoritmo	E1	E2	E3	E4
Coste Uniforme	L=17 T=1min 19s NA=14092	L=11 T=286 ms NA=870	L=20 T=16min 34s NA=48428	L=3 T=547 µs NA=4
Primero el mejor (Linear)	L=39 T=15 ms NA=173	L=33 T=4.42 ms NA=78	L=76 T=210 ms NA=748	L=3 T=435 µs NA=4
Primero el mejor (h3)	L=41 T=26.4 ms NA=232	L=11 T=1 s NA=1796	L=34 T=3.25 s NA=3376	L=3 T=486 µs NA=5
Primero el mejor (Manhattan)	L=83 T=229 ms NA=840	L=93 T=245 ms NA=852	L=140 T=17.2 s NA=7599	L=3 T=436 µs NA=5
A* (Linear)	L=17 T=271 ms NA=873	L=11 T=4.4 ms NA=77	L=20 T=3.62 s NA=3377	L=3 T=440 µs NA=4
A* (Manhattan)	L=21 T=1.88 s NA=2349	L=11 T=47.1 ms NA=336	L=20 T=16.9 s NA=7553	L=3 T=451 µs NA=5
A* (h3)	L=17 T=95.6 ms NA=596	L=11 T=6.72 ms NA=117	L=26 T=5.3 s NA=5045	L=3 T=398 µs NA=6

Se han eliminado las filas de “Anchura” y “Profundidad” ya que no dan ningún ejemplo para comparar la información

En la tabla anterior podemos ver que la cantidad de nodos analizados es menor en la mayoría de los casos, además esto repercute directamente en su tiempo de ejecución.

PROBLEMA DE LOS MISIONEROS.

Para este ejercicio se nos pedía definir una heurística que fuese admisible y estudiar con ella las propiedades del algoritmo de búsqueda "A*". La heurística que hemos definido es dos veces la cantidad de personas a la izquierda menos la posición de la barca (siendo 1 si esta a la izquierda y 0 si esta a la derecha), quedando definida de la siguiente manera:

```
def h1(node):
    '''
        Devuelve 2 veces el número de personas a la izquierda y
        si la barca está a la izquierda la resta 1.
    '''
    goal = (0, 0, 1)
    misioneros_izq = node.state[0]
    canibales_izq = node.state[1]
    pos_barca = node.state[2]
    return 2 * (misioneros_izq + canibales_izq) - pos_barca
```

Una vez definida la heurística se comparan los tiempos de ejecución de un algoritmo de búsqueda ciega, como lo es la "Búsqueda en Anchura (sin control de repetidos)" y el algoritmo "A*" con la heurística definida, obteniendo como resultado:

Algoritmo	Tiempo	Longitud de la solución
Busqueda en Anchura (Sin control de repetidos)	104 ms	11
A* (h1)	187 µs	11

Como podemos observar en la tabla anterior, el algoritmo "A*" demuestra ser mucho más rápido que la "Búsqueda en Anchura", consiguiendo una solución con igual numero de movimientos (11 movimientos).