PRÁCTICA 1:
Búsqueda en espacio de estados

Algoritmos Básicos de Inteligencia Artificial

2023/2024

Tomás Rial Costa - 35602105Z
tomas.rial@alumnos.uvigo.es

Carlos Sánchez Alcalá-Zamora - 47436230H
carlos.sanchez@alumnos.uvigo.es

ÍNDICE

1. Descripción del problema

En esta práctica nos dedicaremos a desarrollar diferentes algoritmos con el objetivo de conseguir resolver un cubo de rubik, para ello, primero realizaremos unos cuantos movimientos aleatorios para así desordenarlo, escogidos estos por la semilla del DNI, y a continuación, en el "main", escogeremos el algoritmo que prefiramos, siendo algunos más rápidos que otros y llegando, incluso, a existir los que entren en un bucle infinito y por lo tanto no sean capaces de alcanzar la solución.

2. Descripción de las funciones heurísticas utilizadas

a. **Heurística Errores**:
   i. Compara cada sticker del cubo actual con su posición correcta en el cubo resuelto.
   ii. Por cada sticker incorrecto, se incrementa el contador de discrepancias.
   iii. Devuelve el número total de discrepancias como la estimación del costo restante para alcanzar el estado objetivo.

```
def Heuristica_Errores(cubo):
    terminado = Cubo()
    prediccion = 0
    for i in range(0, 6):
        for j in range(0, 9):
            if terminado.caras[i].casillas[j].color != cubo.caras[i].casillas[j].color:
                prediccion += 1
    return prediccion
```

No se tiene en cuenta la distancia entre las posiciones actuales y la correcta, simplemente se cuentan cuántos stickers están en la posición incorrecta en comparación con el estado objetivo.

b. Heurística de Manhattan:
   i. Se basa en calcular la distancia de cada sticker a su posición correcta en el estado objetivo.
   ii. Para cada sticker del cubo, se calcula la distancia de Manhattan entre su posición actual y su posición objetivo.
   iii. La distancia de Manhattan entre dos puntos se define como la suma de las diferencias absolutas en sus coordenadas vertical y horizontal.
   iv. Luego, la heurística devuelve la suma total de estas distancias de Manhattan como una estimación del costo restante para alcanzar el estado objetivo.

```python
def distancia_entre_posiciones(posicion_actual, posicion_correcta):
    return abs(posicion_actual[0] - posicion_correcta[0]) + abs(posicion_actual[1] - posicion_correcta[1])


def HeuristicaManhattan(cubo):
    cubo_final = Cubo()
    posiciones_objetivo = {}
    for i in range(6):
        for j in range(9):
            color_sticker = cubo_final.caras[i].casillas[j].color
            posiciones_objetivo[color_sticker] = (i, j)

    distancia_total = 0
    for i in range(6):
        for j in range(9):
            color_sticker = cubo.caras[i].casillas[j].color
            posicion_actual = (i, j)
            posicion_objetivo = posiciones_objetivo[color_sticker]
            distancia_total += Cubo.distancia_entre_posiciones(posicion_actual, posicion_objetivo)

    return distancia_total
```

3. Descripción de la implementación concreta realizada

a. Detalles de implementación de las heurísticas

Las heurísticas se almacenan al final del archivo Cubo.py y se las llama a ese archivo:

Cubo.”*Heurística a utilizar*”(“*cubo a analizar*”)

i. **Heurística Errores:**

Se guarda el cubo resuelto y se itera sobre Caras y Casillas del cubo actual incrementado una variable en 1 cada vez que el color de una casilla en el cubo actual no se corresponda al color en el cubo resuelto. Finalmente devuelve esa variable.

ii. **Heurística de Manhattan:**

Esta función crea un cubo objetivo resuelto y almacena las posiciones objetivo de cada sticker en un diccionario. Luego, itera sobre las casillas del cubo actual y calcula la distancia de Manhattan entre cada casilla y su posición objetivo tomando el valor absoluto de la diferencia entre la posición actual y correcta de fila y columna.

En cada iteración va sumando todas estas distancias para obtener la heurística de Manhattan total del cubo actual en comparación con el cubo objetivo.

b. Detalles de implementación de cada algoritmo de búsqueda

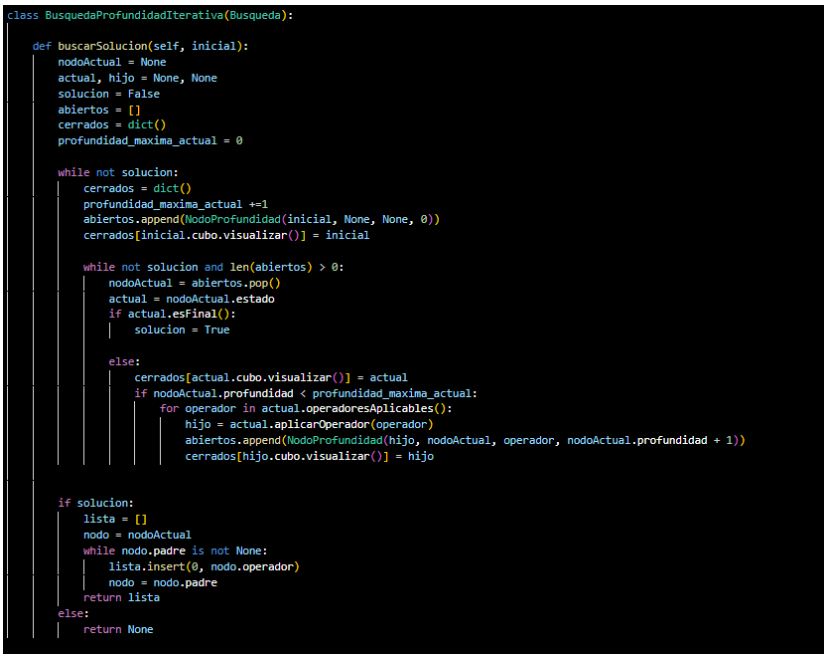
- **Búsqueda en Anchura:** Implementa la búsqueda en anchura, utilizando una lista de nodos abiertos y un diccionario de nodos cerrados para controlar la repetición de estados. Recorre todos los estados, nivel por nivel, hasta encontrar la solución.

```python
class BusquedaAnchura(Busqueda):

    #Implementa la búsqueda en anchura. Si encuentra solución recupera la lista de Operadores empleados almacenada en los atributos de los objetos NodoAnchura
    def buscarSolucion(self,inicial):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = dict()
        abiertos.append(NodoAnchura(inicial, None, None))
        cerrados[inicial.cubo.visualizar()]=inicial
        while not solucion and len(abiertos)>0:
            nodoActual = abiertos.pop(0)
            actual = nodoActual.estado
            if actual.esFinal():
                solucion = True
            else:
                #cerrados[actual.cubo.visualizar()] = nodoActual
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    if hijo.cubo.visualizar() not in cerrados.keys():
                        abiertos.append(NodoAnchura(hijo, nodoActual, operador))
                        cerrados[hijo.cubo.visualizar()]=hijo
        if solucion:
            lista = []
            nodo = nodoActual
            while nodo.padre != None: #Asciende hasta la raíz
                lista.insert(0, nodo.operador)
                nodo = nodo.padre
            return lista
        else:
            return None
```

- **Búsqueda en Profundidad:** Implementa la búsqueda en profundidad, al igual que en el caso anterior, se utiliza una lista de nodos abiertos y un diccionario de nodos cerrados para así evitar la sobrecarga de memoria con estados repetidos. En este caso se explora cada rama del árbol hasta encontrar una solución, provocando que esta no pueda llegar a hallarse, ya que el algoritmo se podría quedar en una rama infinita. En el código, la principal diferencia que encontramos con Anchura es que el nuevo estado se añade al principio de abiertos, con el *pop(-1)* y el *append()* del nodo.

```python
class BusquedaProfundidad(Busqueda):

    def buscarSolucion(self,inicial):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = dict()
        abiertos.append(NodoAnchura(inicial, None, None))
        cerrados[inicial.cubo.visualizar()]=inicial
        while not solucion and len(abiertos)>0:
            nodoActual = abiertos.pop(-1)
            actual = nodoActual.estado
            if actual.esFinal():
                solucion = True
            else:
                #cerrados[actual.cubo.visualizar()] = nodoActual
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    if hijo.cubo.visualizar() not in cerrados.keys() and hijo.cubo.visualizar() not in abiertos:
                        abiertos.append(NodoAnchura(hijo, nodoActual, operador))
                        cerrados[hijo.cubo.visualizar()]=hijo
        if solucion:
            lista = []
            nodo = nodoActual
            while nodo.padre != None: #Asciende hasta la raíz
                lista.insert(0, nodo.operador)
                nodo = nodo.padre
            return lista
        else:
            return None
```

- **Búsqueda en Profundidad Iterativa:** Funciona igual que la anterior, pero se añade un límite de profundidad para evitar las ramas infinitas ya nombradas, consiguiendo así obtener resultados. La cota es una variable que aumenta en cada repetición del primer bucle *while* y que define hasta qué profundidad puede ir el código gracias a una condición en el *else,* para poder entrar a aplicar operadores, especificando que la profundidad actual tiene que ser menor que la de la cota.



```
class BusquedaProfundidadIterativa(Busqueda):

    def buscarSolucion(self, inicial):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = dict()
        profundidad_maxima_actual = 0

        while not solucion:
            cerrados = dict()
            profundidad_maxima_actual +=1
            abiertos.append(NodoProfundidad(inicial, None, None, 0))
            cerrados[inicial.cubo.visualizar()] = inicial

            while not solucion and len(abiertos) > 0:
                nodoActual = abiertos.pop()
                actual = nodoActual.estado
                if actual.esFinal():
                    solucion = True

                else:
                    cerrados[actual.cubo.visualizar()] = actual
                    if nodoActual.profundidad < profundidad_maxima_actual:
                        for operador in actual.operadoresAplicables():
                            hijo = actual.aplicarOperador(operador)
                            abiertos.append(NodoProfundidad(hijo, nodoActual, operador, nodoActual.profundidad + 1))
                            cerrados[hijo.cubo.visualizar()] = hijo


        if solucion:
            lista = []
            nodo = nodoActual
            while nodo.padre is not None:
                lista.insert(0, nodo.operador)
                nodo = nodo.padre
            return lista
        else:
            return None
```

- **Búsqueda Voraz:** Este algoritmo implementa una búsqueda informada, en concreto la búsqueda voraz (Greedy Best-First Search). En una búsqueda voraz se expande el nodo que parece ser el más prometedor según la heurística utilizada. En este caso, se está utilizando la heurística de Manhattan para ordenar los nodos en la lista abierta antes de expandirlos. Esto significa que se selecciona el nodo que tiene la menor distancia heurística al objetivo, pero no necesariamente considera la profundidad o el costo total de llegar a ese nodo.

```
class BusquedaVoraz(Busqueda):

    def buscarSolucion(self, inicial):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = dict()
        abiertos.append(NodoAnchura(inicial, None, None))
        cerrados[inicial.cubo.visualizar()] = inicial
        while not solucion and len(abiertos) > 0:
            abiertos.sort(key=lambda x: Cubo.HeuristicaManhattan(x.estado.cubo))
            nodoActual = abiertos.pop(0)
            actual = nodoActual.estado
            if actual.esFinal():
                solucion = True
            else:
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    if hijo.cubo.visualizar() not in cerrados.keys():
                        abiertos.append(NodoAnchura(hijo, nodoActual, operador))
                        cerrados[hijo.cubo.visualizar()] = hijo
        if solucion:
            lista = []
            nodo = nodoActual
            while nodo.padre is not None:
                lista.insert(0, nodo.operador)
                nodo = nodo.padre
            return lista
        else:
            return None
```

- **Búsqueda A*:** El algoritmo A* es una técnica de búsqueda informada que combina la estrategia de búsqueda en anchura con la estrategia voraz. Utiliza una función de costo que combina el costo acumulado desde el estado inicial con una estimación heurística del costo restante hasta el estado objetivo. Esta función de costo, denotada como f = g + h, se utiliza para evaluar los nodos y guiar la búsqueda hacia la solución. A* selecciona en cada paso el nodo con el valor de f más bajo, lo que garantiza que se explore primero el camino más prometedor.

```python
class BusquedaAStar(Busqueda):

    def buscarSolucion(self, inicial):
        nodoActual = None
        actual, hijo = None, None
        solucion = False
        abiertos = []
        cerrados = dict()
        abiertos.append(NodoAStar(inicial, None, None, 0))  # Agrega el costo g(n) como último argumento
        cerrados[inicial.cubo.visualizar()] = NodoAStar(inicial, None, None, 0)  # Almacena un NodoAStar en cerrados
        while not solucion and len(abiertos) > 0:
            abiertos.sort(key=lambda x: x.costo + Cubo.HeuristicaManhattan(x.estado.cubo))  # Ordena por f(n) = g(n) + h(n)
            nodoActual = abiertos.pop(0)
            actual = nodoActual.estado
            if actual.esFinal():
                solucion = True
            else:
                for operador in actual.operadoresAplicables():
                    hijo = actual.aplicarOperador(operador)
                    g_nuevo = nodoActual.costo + 1  # Costo acumulado actualizado
                    h_estimada = Cubo.HeuristicaManhattan(hijo.cubo)  # Calcula la heurística h(n) para el nuevo estado hijo
                    if hijo.cubo.visualizar() not in cerrados.keys() or g_nuevo + h_estimada < cerrados[hijo.cubo.visualizar()].costo:
                        abiertos.append(NodoAStar(hijo, nodoActual, operador, g_nuevo))
                        cerrados[hijo.cubo.visualizar()] = NodoAStar(hijo, nodoActual, operador, g_nuevo)  # Almacena un NodoAStar en cerrados
        if solucion:
            lista = []
            nodo = nodoActual
            while nodo.padre != None:  # Asciende hasta la raíz
                lista.insert(0, nodo.operador)
                nodo = nodo.padre
            return lista
        else:
            return None
```

- **Búsqueda IDA***: Este algoritmo combina la estrategia de búsqueda en profundidad con un límite iterativo y la heurística de A* para guiar la exploración del espacio de búsqueda hacia la solución de manera eficiente. Utiliza una cota que se actualiza en cada iteración para controlar la profundidad de búsqueda y realiza una búsqueda exhaustiva hasta encontrar la solución o hasta que se agoten los nodos por explorar.

```python
class BusquedaIDAStar(Busqueda):
    def buscarSolucion(self, inicial):
        NUEVA_COTA = Cubo.HeuristicaManhattan(inicial.cubo)
        Solucion = False
        while not Solucion:
            COTA = NUEVA_COTA
            NUEVA_COTA = float('inf')
            ABIERTOS = []
            ABIERTOS.append(NodoAStar(inicial, None, None, 0))
            while ABIERTOS and not Solucion:
                ABIERTOS.sort(key=lambda x:Cubo.HeuristicaManhattan(x.estado.cubo))
                ACTUAL = ABIERTOS.pop(0)
                actual = ACTUAL.estado
                if actual.esFinal():
                    Solucion = True
                else:
                    for operador in actual.operadoresAplicables():
                        hijo = actual.aplicarOperador(operador)
                        g = ACTUAL.costo + 1
                        f = g + Cubo.HeuristicaManhattan(hijo.cubo)
                        if f <= COTA:
                            ABIERTOS.append(NodoAStar(hijo, ACTUAL, operador, g))
                        else:
                            NUEVA_COTA = min(NUEVA_COTA, f)
            if Solucion:
                lista = []
                nodo = ACTUAL
                while nodo.padre is not None:
                    lista.insert(0, nodo.operador)
                    nodo = nodo.padre
                return lista
            else:
                return None
```

c. Modificaciones realizadas sobre el código de partida

Además de las búsquedas y heurísticas explicadas anteriormente, se han añadido los nodos de profundidad y estrella para el correcto funcionamiento.

```python
class NodoProfundidad(Nodo):
    def __init__(self, estado, padre, operador, profundidad):
        super().__init__(estado, padre)
        self.operador = operador
        self.profundidad = profundidad

class NodoAStar:
    def __init__(self, estado, padre, operador, costo):
        self.estado = estado
        self.padre = padre
        self.operador = operador
        self.costo = costo
```

Durante el diseño añadimos una funcionalidad al visionado del cubo para poder ver bien los movimientos que hacía, añadiendo al lado del color de cada casilla su posición correcta.

```python
def stringFila1(self, cara):
    return self.etq_colores[cara.casillas[0].color] + str(cara.casillas[0].posicionCorrecta) +
```

```
CUBO FINAL:
    W0W1W2
    W7W8W3
    W6W5W4
Y0Y1Y2 O0O1O2 R0R1R2 G0G1G2
Y7Y8Y3 O7O8O3 R7R8R3 G7G8G3
Y6Y5Y4 O6O5O4 R6R5R4 G6G5G4

    B0B1B2
    B7B8B3
    B6B5B4
```

Finalmente, cuando ya hicimos todas las comprobaciones necesarias con este método, decidimos quitar esta funcionalidad debido a que obstaculiza un poco la limpieza del visionado del cubo.

4. Medidas/criterios usados en la evaluación.

a. Tiempo de búsqueda
b. Nodos explorados
c. Tamaño de la lista ABIERTOS
d. Número de movimientos hasta encontrar la solución

5. Experimentos realizados y resultados obtenidos.

Tener en cuenta que para hacer los experimentos hemos usado una semilla para que todos los algoritmos se ejecuten de una manera equitativa.
Nuestra semilla es 35.

Para cada criterio hemos realizado la ejecución de todos los posibles algoritmos con distintas heurísticas e incrementando el número de movimientos de desorden. Después de hacer los experimentos hemos removido sus funciones del código para una mayor limpieza.

a. Tiempo de búsqueda

Para la obtención de los tiempos de ejecución realizamos un *time.time()* al comienzo y final de la ejecución del *main* y los restamos entre sí.

```
inicio = time.time()
opsSolucion = problema.obtenerSolucion()
tiempo=time.time() - inicio
```

Tipo de Búsqueda	Número de movimientos de desorden				
	2	3	4	5	6
Anchura	0.00603	0.02411	0.72540	4.49138	38.58355
Profundidad	0.51992	Bucle infinito			
Profundidad iterativa	0.00199	0.01894	0.06183	2.79580	43.16757
Voraz Errores	0.00201	0.00301	0.00338	0.00411	-
Voraz Manhattan	0.00149	0.01253	-	-	-
A* Errores	0.00598	0.00357	0.00604	0.00598	-
A* Manhattan	0.00600	0.00300	0.0391	0.4286	-
IDA* Errores	0.00100	0.00267	0.00456	0.00410	-
IDA* Manhattan	0.01744	0.00257	-		

b. Nodos explorados

Para la obtención del número de nodos visitados hemos creado una variable en el código e incrementado su valor en 1 unidad antes de sacar un nodo utilizando el *pop()*.

```
num_nodos += 1
ACTUAL = ABIERTOS.pop(0)
```

Tipo de Búsqueda	Número de movimientos de desorden				
	2	**3**	**4**	**5**	**6**
Anchura	11	59	1343	7283	51371
Profundidad	755	Bucle infinito			
Profundidad iterativa	4	122	437	15518	219108
Voraz Errores	2	3	4	5	-
Voraz Manhattan	2	3	-		
A* Errores	2	3	4	5	-
A* Manhattan	2	3	21	22	-
IDA* Errores	2	3	4	5	-
IDA* Manhattan	2	3	-	-	-

c. Tamaño máximo de la lista ABIERTOS

Para la obtención del tamaño máximo de la lista abiertos hemos creado una variable y la hemos ido actualizando dándole el valor de *len(abiertos)* si es mayor al que ya tiene para cada abiertos.

```
if len(abiertos)>maximo_abiertos:
        maximo_abiertos=len(abiertos)
```

Tipo de Búsqueda	Número de movimientos de desorden				
	2	**3**	**4**	**5**	**6**
Anchura	111	567	11295	61023	430215

Profundidad	5283	Bucle infinito			
Profundidad iterativa	12	23	34	45	56
Voraz Errores	12	23	34	45	-
Voraz Manhattan	12	23	-		
A* Errores	2	3	34	45	-
A* Manhattan	12	23	125	136	-
IDA* Errores	1	3	19	36	-
IDA* Manhattan	1	9	-	-	-

d. Número de movimientos hasta encontrar la solución

Para la obtención del número de nodos visitados hemos creado una variable que se incrementa en 1 antes de aplicar un operador a actual para conseguir hijo en cada iteración.

```
for operador in actual.operadoresAplicables():
    num_movimientos+=1
    hijo = actual.aplicarOperador(operador)
```

Tipo de Búsqueda	**Número de movimientos de desorden**				
	2	**3**	**4**	**5**	**6**
Anchura	120	696	16104	87384	616440
Profundidad	9048	Bucle infinito			
Profundidad iterativa	12	132	456	15540	219132
Voraz Errores	12	24	36	48	-
Voraz Manhattan	12	24	-		
A* Errores	12	24	36	48	-
A* Manhattan	12	24	240	252	-

IDA* Errores	12	24	36	48	-
IDA* Manhattan	24	36	-	-	-

6. Conclusiones y problemas encontrados.

- **Eficiencia Temporal:** Los algoritmos informados, como Voraz Manhattan, A* e IDA*, tienden a tener tiempos de ejecución más bajos en comparación con los algoritmos no informados, como Anchura o Profundidad. Esto se debe a la capacidad de los algoritmos informados para dirigirse directamente hacia soluciones prometedoras en lugar de explorar todo el espacio de búsqueda.

  A partir de un número de movimientos de desorden un poco elevado, la mayoría de algoritmos no son capaces de encontrar una solución para un tiempo aceptable. La Profundidad Iterativa llega una solución incluso con 7 movimientos en un tiempo “aceptable”, pero nos pareció redundante ponerlo en la tabla de comparación ya que los otros algoritmos o no llegan por culpa de bucles infinitos (Profundidad), o lo hacen en un tiempo mucho más elevado.

- **Complejidad Espacial:** Los algoritmos informados exploran menos nodos en comparación con los algoritmos no informados. Esto sugiere que los algoritmos informados son más eficientes en términos de uso de memoria, ya que pueden priorizar los nodos más prometedores en función de la heurística utilizada.

- **Uso de Memoria:** Los algoritmos no informados, como la búsqueda en anchura y la búsqueda en profundidad, tienden a requerir más memoria, especialmente para un mayor número de movimientos de desorden. Por otro lado, los algoritmos informados tienden a mantener listas ABIERTOS más pequeñas debido a su capacidad para enfocarse en áreas prometedoras del espacio de búsqueda.

- **Efectividad en la Resolución del Problema:** Los algoritmos informados generalmente encuentran soluciones en menos movimientos en comparación con los algoritmos no informados. Sin embargo, es importante tener en cuenta qué algunos algoritmos pueden no encontrar soluciones para ciertos casos (por ejemplo, la búsqueda en Profundidad cayendo en un bucle infinito).

  En general, los algoritmos informados parecen ser más efectivos en términos de tiempo, espacio y efectividad en la resolución del problema en comparación con los algoritmos no informados (aunque en términos de efectividad/tiempo el mejor sería la profundidad iterativa ya que llega a más movimientos de desorden en tiempos buenos).

- **Problemas de ejecución:** El algoritmo de búsqueda en profundidad queda atrapado en un bucle infinito a partir de más de 2 iteraciones, por otro lado, la mayoría de algoritmos comienzan a tardar demasiado tiempo una vez se llega a las 6

iteraciones, esto puede ser debido a una cantidad demasiado elevada de nodos creados.

- **Problemas heurísticos**: La heurística de Manhattan, al ser más compleja que la heurística basada en “errores” (mala posición de los stickers), es posible que en algunos algoritmos (como es el caso del voraz), al aumentar los movimientos, le lleve un tiempo indefinido mientras que el de errores termina sin problemas.