Desarrollo de un planificador para resolver el cubo de Rubik

GUIADO Y NAVEGACIÓN DE ROBOTS

Carlos Prados Sesmero | 70830147z | M18129

Contenido

Introducción

Con el objetivo de realizar un código en Python que sea capaz de resolver un cubo de Rubik, se ha planteado un algoritmo de búsqueda heurística para encontrar una solución del problema. Para ello se parte de una clase, localizada en un fichero, que permite simular movimientos del cubo, y da una codificación del estado de este.

Una vez comprendida la clase desarrollada por el profesor Miguel Hernando, se plantean las siguientes tareas:

- Desarrollar una clase en Python para la implementación de cada vértice del grafo.
- Desarrollar un grafo que describa los estados del cubo de Rubik dependiendo de la acción desarrollada.
- Determinar el algoritmo de búsqueda.
- Implementar el algoritmo de búsqueda heurística para recorrer los vértices del grafo desarrollado.
- Mostrar los resultados obtenidos.

Desarrollo de una clase para los vértices

Con el objetivo de que cada vértice del grafo sea un objeto dentro del programa, se ha desarrollado una clase en Python que tenga las características típicas de un vértice, en concreto de un vértice dentro de un grafo en forma de árbol. Esta clase debe incluir:

- Número de identificación.
- Hijos del vértice. Son los vértices inmediatamente inferiores.
- Padre del vértice. Es el vértice inmediatamente superior.
- Eje en el que se mueve el cubo de Rubik.
- Fila que se mueve.
- Numero de movimientos generados en dicha fila.
- Estado del vértice. Si se ha estudiado si el vértice es una solución.

El código de dicha clase se muestra a continuación, donde se encuentran las variables, la inicialización de un nuevo vértice y una función asociada para mostrar los datos del vértice asociado:

```
# Clase objeto de cada vértice del grafo
class Vertice:
    iden = 0
    hijos = deque()
    padre = 0
    eje = ''
    fila = 0
    num = 0
    estado = 0

    # Inicialización de cada vértice
    def __init__(self, ide_n, padre_n, eje_n, fila_n, num_n, hijos_n):
        self.iden = ide_n
        self.padre = padre_n
        self.eje = eje_n
        self.fila = fila_n
        self.num = num_n
        self.hijos = hijos_n

    # Muestra información del vértice asociado
    def muestra_Datos(self):
        print('\nId: ',self.iden)
        print("Padre: ", self.padre)
        print('Eje: ', self.eje)
        print("Fila: ", self.fila)
        print("Num: ", self.num)
```

Desarrollo del grafo

Se ha planteado un grafo en forma de árbol, en el cual todo nodo padre tenga tantos hijos como movimientos posibles a realizar sobre el cubo de Rubik. Los movimientos posibles son (mediante la función definida "rotate_90()"):

- Eje: Se puede mover el cubo de Rubik en los tres ejes del espacio ('x', 'y', 'z').
- Fila: Por cada eje se puede mover en una de las N filas existentes en el cubo.
- Numero: Por cada fila elegida podemos girar K veces (-1, 1 ó 2).

Es decir, en un cubo de Rubik de 3x3, se pueden realizar 27 operaciones distintas, en uno de 4x4, se pueden 36, etc. El grafo tiene la apariencia mostrada en la *Figura 1*.

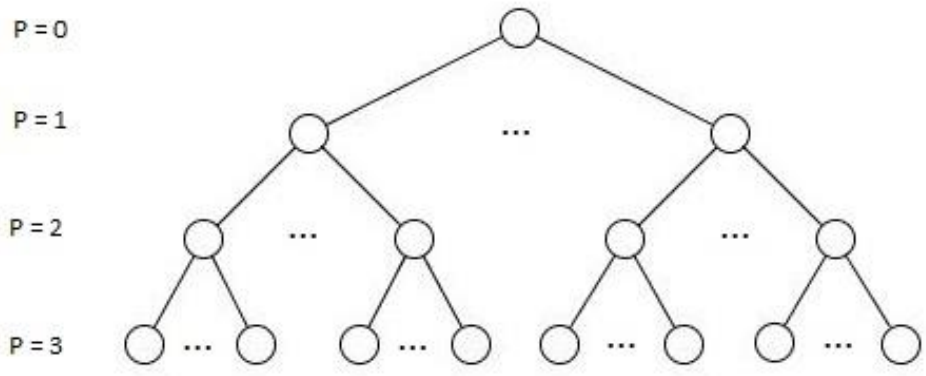

Figura 1. Estructura del grafo desarrollado para la resolución del problema del cubo de Rubik.

Por lo tanto, se va a desarrollar un grafo de una profundidad dada (P), dependiendo de la complejidad del cubo que queramos resolver. Es decir, si sobre el cubo de Rubik resuelto realizamos 3 movimientos, debemos generar un árbol de profundidad 3 (tomando el nodo inicial como profundidad 0).

La generación del grafo, por tanto, va a consistir en generar tantos vértices como sean necesarios para una determinada profundidad. Este número de vértices viene definido por una serie geométrica como la siguiente, siendo N el tamaño del cubo de Rubik:

$$b = 3 \cdot 3 \cdot N$$

$$vertices = \sum_{i=0}^{P} b^i$$

Cada uno de los vértices va a tener un valor de eje, fila y número. Además, el padre va a ser uno nuevo (consecutivos) cada 'b' vértices. La generación del grafo se refleja en el siguiente código:

```
# Variables para la generación del grafo
profun = 4
ide = 0
collection_eje = ['x','y','z']
collection_num = [-1, 1, 2]
num_nodos = 0

# Vertice inicial
hijos = deque()
x = Vertice(0,-1,'',-1,0,hijos)
grafo.append(x)

# Numero de nodos a crear
b = (3*3*N)
for i in range(0, profun+1):
    num_nodos += b**i

# Generación del grafo
padre = 0

while ide+1<num_nodos:
    for j in collection_eje:
        for k in range(0,3):
            for l in collection_num:
                ide+=1
                hijos = deque()
                x = Vertice(ide, padre, j, k, l, hijos)
                grafo.append(x)

    if ide==(b*padre + b):
        for m in range(ide-(b-1), ide+1):
            grafo[padre].hijos.append(m)
        padre = padre + 1


print("\nGrafo generado")
```

Determinación y desarrollo del algoritmo de búsqueda

Las posibilidades de empleabilidad de algoritmos de búsqueda se pueden reducir a dos. O bien una búsqueda en profundidad o en anchura, ya que el resto de los algoritmos son utilizados en grafos cuyas aristas tienen pesos.

Cuando tenemos un grafo en forma de árbol como el que se dispone, con gran cantidad de vértices por nivel, existe un gran dilema para elegir cual es el más conveniente. Nuestro grafo tiene gran cantidad de vértices por cada nivel y este valor aumenta exponencialmente cuando añadimos nuevos niveles.

Cuando utilizamos un algoritmo de búsqueda en anchura, todos los vértices se tienen que explorar, algo que es muy lento si la solución se encuentra en el último nivel. Sin embargo, cuando utilizamos un algoritmo de búsqueda en profundidad, nuestro algoritmo se puede perder en una ramificación sin encontrar nunca la solución.

En nuestro caso, las ramificaciones son homogéneas, es decir, no hay ramas más profundas que otras, ni tienen diferente número de vértices, por lo que se va a utilizar un algoritmo de búsqueda en profundidad.

A pesar de ello, se ha desarrollado también un algoritmo de búsqueda en anchura, ya que la modificación es muy sencilla y fácil de implementar. De esta manera podemos comparar el comportamiento de ambos algoritmos.

La implementación del algoritmo es la siguiente:

```
# Algoritmo de búsqueda
def Busqueda(profundidad):

    ##### Algoritmo de búsqueda (DFS)
    for i in range(0, len(grafo)):
        grafo[i].estado = 0

    grafo[0].estado = 1;

    # Creamos cola de exploración
    cola = deque()
    for i in range(0, len(grafo[0].hijos)):
        cola.append(grafo[0].hijos[i])

    exito=0

    # Se sigue explorando
    while ((len(cola)!=0) and (exito==0)):

        # Sacamos un vertice de la cola
        if profundidad==1:
            u = cola.pop()
        else:
            u = cola.popleft()
        grafo[u].estado = 1

        a = define_Cubo(N)

        # Algoritmo de comprobación
        exito = Comprueba(a, u, resuelto)

        if exito==1:
            print("\nSOLUCION ENCONTRADA")
            acciones = deque()
            acciones.append(u)
            a = define_Cubo(N)
```

```
            while grafo[u].padre != 0:
                u = grafo[u].padre
                acciones.append(u)

            # Muestro el primer avance en la resolución
            a.plot()

            # Muestro el resto de avances
            while len(acciones)!=0:
                accion = acciones.pop()
                # Información sobre el camino tomado
                grafo[accion].muestra_Datos()
                a.rotate_90(grafo[accion].eje, grafo[accion].fila, grafo[accion].num)
                a.plot()
            acciones.clear()

        else:
            for i in range(0, len(grafo[u].hijos)):
                # Encolo los nuevos nodos
                cola.append(grafo[u].hijos[i])
```

Lo que hace este algoritmo es generar una cola de búsqueda y, dependiendo del tipo de búsqueda, analizar los vértices correspondientes. El análisis cosiste en comprobar los movimientos previos que existen para llegar al vértice dado. Esto provoca que a medida que aumenta la profundidad el algoritmo se hace muy lento, ya que tiene que recalcular todos los movimientos previos en cada iteración. Una posible mejora consiste en almacenar el estado del cubo de Rubik de cada nodo.

Para cada vértice analizado se compara el estado del cubo actual con el cubo resuelto. Esto se hace gracias a una función llamada “Comprueba”:

```
# Algoritmo de identificación del exito de una serie de acciones en el cubo
def Comprueba(cubo, nodo, resultado):

    acciones = deque()
    acciones.append(nodo)
    while grafo[nodo].padre != 0:
        nodo = grafo[nodo].padre
        acciones.append(nodo)

    while len(acciones)!=0:
        accion = acciones.pop()
        cubo.rotate_90(grafo[accion].eje, grafo[accion].fila, grafo[accion].num)

        # Comprobar si se ha resuelto
        if(resultado==cubo.get_State()):
            return 1

    acciones.clear()
    return 0
```

Resultados obtenidos

En este apartado, se van a mostrar tres ejemplos de funcionamiento del programa desarrollado y el tiempo que ha empleado cada una de ellas en encontrar el resultado correcto.

El primer ejemplo es el mostrado en la *Figura 2*.

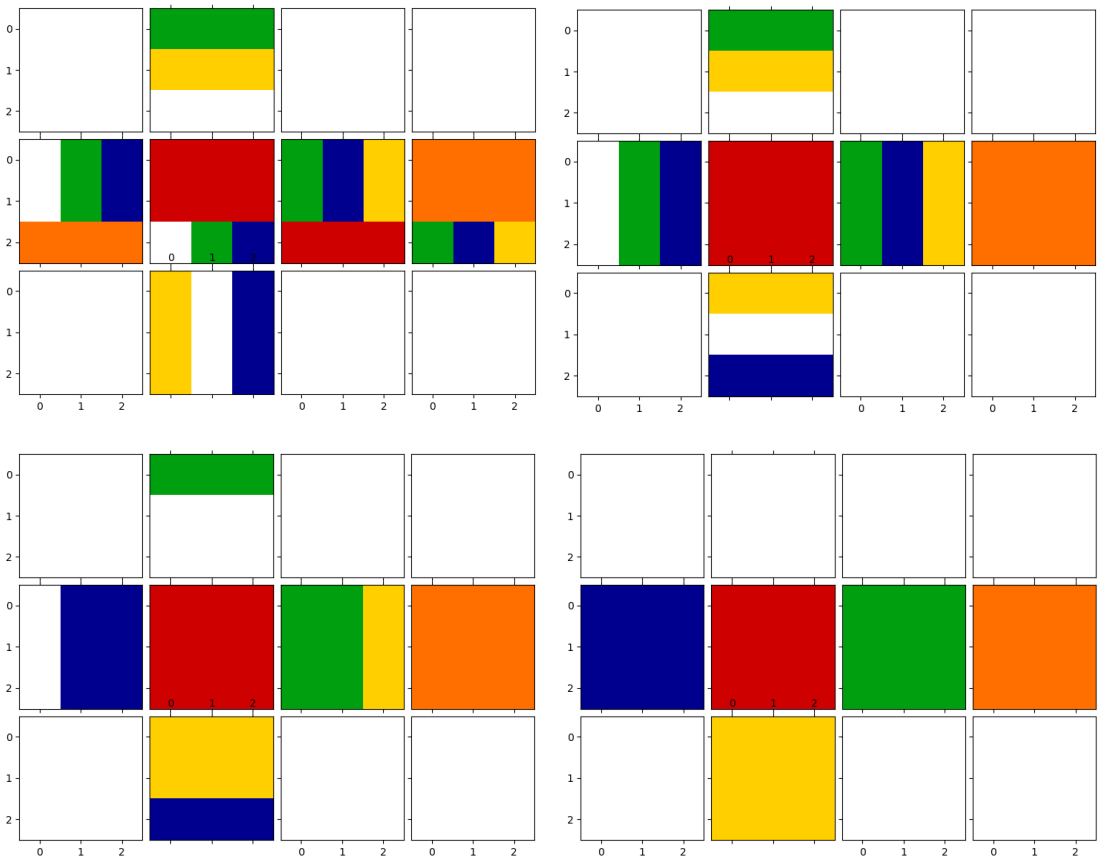

Figura 2. Ejemplo de resolución del cubo de Rubik en 3 movimientos

El camino para encontrar la solución se muestra en la *Figura 3*, donde se han explorado los vértices 0->25->681->18394.

```
prados@prados:~/Documentos/Guiado$ python cubo.py
SOLUCION ENCONTRADA

Id:  25
Padre:  0
Eje:  z
Fila:  2
Num:  -1

Id:  681
Padre:  25
Eje:  x
Fila:  1
Num:  2

Id:  18394
Padre:  681
Eje:  x
Fila:  2
Num:  -1
prados@prados:~/Documentos/Guiado$
```

Figura 3. Resultados del programa

Este ejemplo ha encontrado la solución en tres movimientos, ya que se han generado tres movimientos previos. El siguiente ejemplo va a consistir en cuatro movimientos previos, por lo que el resultado ha sido el que se muestra en la *Figura 4*.

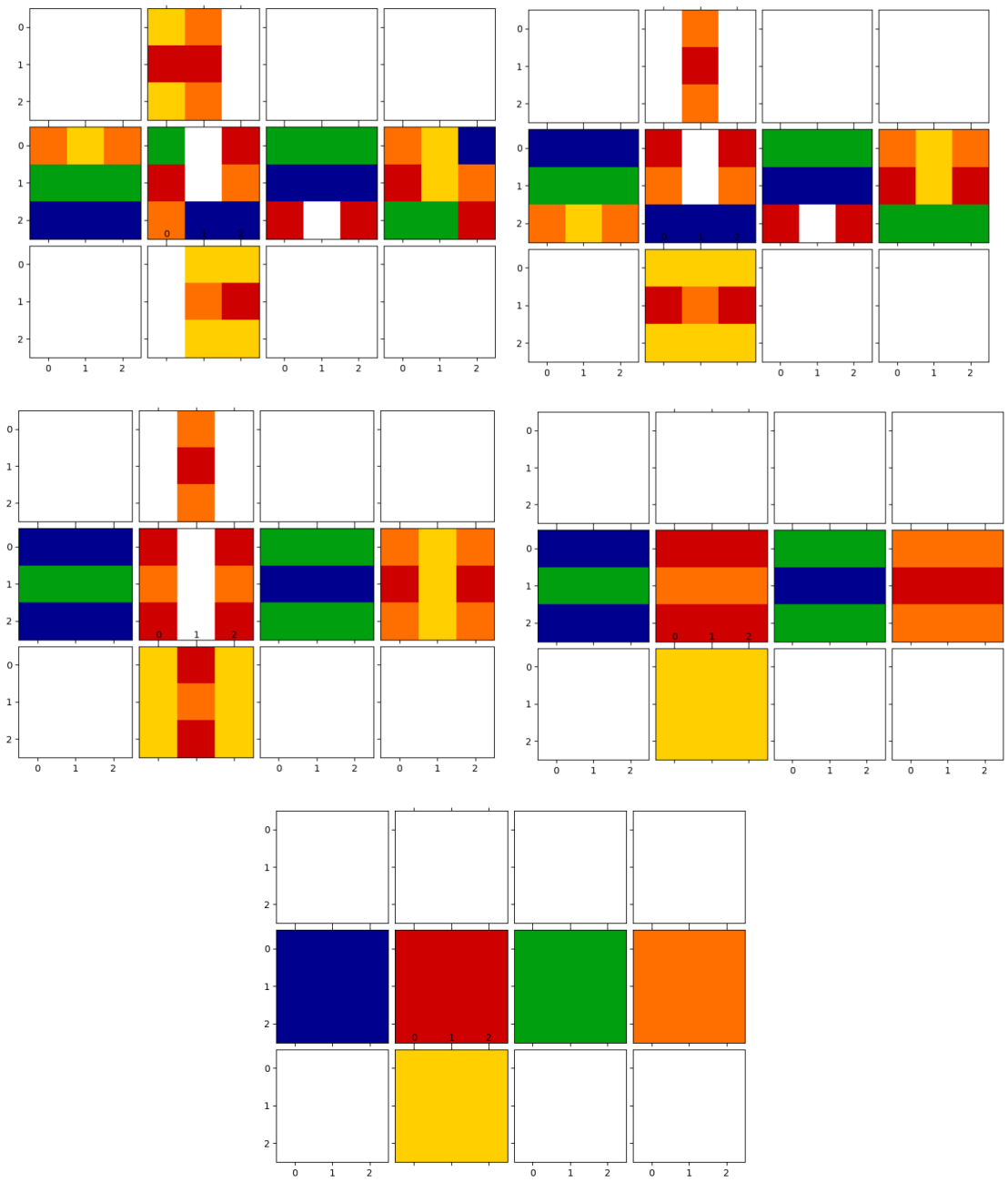

Figura 4. Ejemplo de resolución del cubo de Rubik en 4 movimientos.

En este caso, los vértices explorados han sido los mostrados en la *Figura 5*, es decir, el 0->18->511->13810->372894.

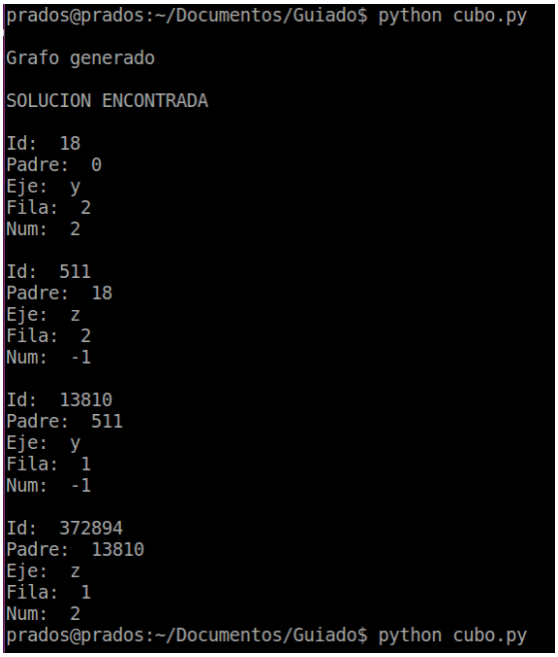

Figura 5. Resultados de la exploración.

Conclusiones

Cuando ejercemos tres movimientos sobre el cubo de Rubik, el resultado se alcanza al cabo de unos segundos, sin embargo, si aplicamos cuatro movimientos, el resultado se alcanza pasado un minuto aproximadamente.

Este tiempo es variante dependiendo del algoritmo de búsqueda utilizado. Las pruebas indican los siguientes resultados:

- Primero en anchura:
  - 3 movimientos. En este caso se tarda aproximadamente dos segundos en alcanzar el resultado.
  - 4 movimientos. En este caso, cerca de dos minutos.
- Primero en profundidad:
  - 3 movimientos. Se tarda entre uno y dos segundos.
  - 4 movimientos. Tarda entre uno y dos minutos.

Por lo que podemos concluir que el algoritmo de búsqueda en profundidad es más rápido, al igual que habíamos supuesto inicialmente.

En contrapartida, si la solución del grafo se encuentra en una capa inferior (C) a la profundidad máxima (P), la búsqueda en profundidad buscará soluciones más complejas de una manera más lenta. Es decir, en vez de generar únicamente 'C' movimientos, va a generar 'P' movimientos complementarios.