ESDI III

Universidad Panamericana Campus Aguascalientes

Cristian Aragón Salazar - 0250005

Rubik's Solver (Github)

Aún más información en el README.md de github

IDEA

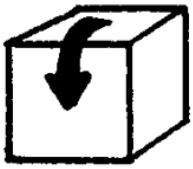
Pensamiento de la idea

1. Al principio tenía pensado utilizar una representación de una matriz de 3x3x6, pero al estar investigando me encontré con un paper en el que se hablaba de todas las representaciones que se pueden hacer con un cubo, al final la que más me convenció por entendimiento y facilidad fue la de un array de 54 caracteres, en el cual puse todos los stickers con la denominación de stickers, con las letras W, Y, O, R, G, y B que significan lo siguiente:
   W = White (Cara frontal)
   Y = Yellow (Cara trasera)
   O = Orange (Cara superior)
   R = Red (Cara inferior)
   G = Green (Cara izquierda)
   B = Blue (Cara derecha)
2. Cuando comencé a desarrollar los movimientos de los stickers desarrollé la siguiente estructura de movimientos:

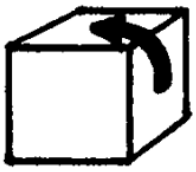
["x", "U", "L"],
["x", "U", "R"],
["x", "D", "L"],
["x", "D", "R"],
["y", "U", "L"],
["y", "U", "R"],
["y", "D", "L"],
["y", "D", "R"],
["z", "L", "U"],
["z", "L", "D"],
["z", "R", "U"],
["z", "R", "D"]

Donde [x, y, z] representan sobre qué eje se llevará a cabo el movimiento, [L, R] (Left y Right), en el caso de x, y significan la posición en la cual se llevará a cabo el movimiento, por ejemplo si es [x, L] será sobre el eje x en la columna de stickers que están en la izquierda, para el caso de z, estos significan hacia donde se llevará a cabo el movimiento, por ejemplo, si es [z, L] se girará sobre el eje z, hacia la izquierda, la fila de stickers se girará a la izquierda. Por último [U, D] (Up y Down), en el caso de x, y significan si el movimiento se hará hacia arriba o hacia abajo, para el caso de z se trata de si son la fila superior o la fila inferior. Las rotaciones las ilustro a continuación:

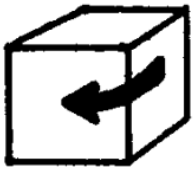
X



Y



Z



Problemas

1. Al ser una representación de 54 elementos, cada movimiento lo tenía que hacer entre cada elemento de forma lineal y para tenerlo más fácil seguí la siguiente representación de mi cubo de rubik:

Donde del 0 al 8 son mi cara frontal y de ahí se deducen las demás.

FUNCIONAMIENTO

rubiks_cube.py

Complejidad: Explicada a detalle más adelante.

La representación de 54 elementos la elegí gracias a su simplicidad como mencioné más arriba, ya que mediante un solo array de 54 elementos es simple y fácil de manejar en comparación con la representación de matriz, etc. Además de que ofrece un acceso eficiente a los stickers, ya que estos se almacenan en el único array y esto facilita el acceso y manipulación de ellos mediante índices.

En espacio lo que ocupa es el único array de 54 elementos, por lo que para almacenarlo se trata de un espacio constante de memoria.

En cuanto a complejidad:

- **Movimientos y rotaciones:** Implican intercambiar los stickers en el array, por lo que cada operación implica mover un número constante de stickers y eso implica tanto en espacio como tiempo **O(1)**, gracias al acceso aleatorio que ofrece el array.
- **Movimientos aleatorios:** Se lleva a cabo en la función ***random_shuffle***, en la cual se da un número de movimientos que se harán de forma aleatoria, y dado que cada movimiento tiene una complejidad de **O(1)**, la complejidad total será de **O(n)**, donde **n** es el número de movimientos.
- **Movimientos personalizados:** Se lleva a cabo en la función ***list_shuffle***, que permite hacer movimientos en base a una lista dada, por lo tanto, como cada movimiento tiene una complejidad de **O(1)**, la complejidad total será de **O(m)**, donde **m** es el número de movimientos en la lista.
- **Movimiento individual:** Se lleva a cabo en la función ***move***, que permite realizar un único movimiento, dado que el manejo de stickers, como lo vemos en el punto de movimientos y rotaciones, es constante, al hacer un único movimiento su complejidad será de **O(1)**.

heuristics.py

Complejidad: Análisis individual

- **count misplaced stickers (conteo de stickers mal ubicados):** En este caso, la heurística cuenta el número de stickers que están en una posición incorrecta, tomando en cuenta con el cubo resuelto, compara cada sticker del cubo con el correspondiente en el cubo objetivo y cuenta aquellos que no coinciden.

  La justifico ya que esta heurística va directamente cuantificando cuántos stickers están en la posición incorrecta, lo que proporciona una media de la distancia entre el estado actual y el estado objetivo del cubo.

  Tiene una complejidad **O(n)**, donde **n** es el número de stickers, ya que debe recorrer cada sticker del cubo y comparar con el cubo objetivo.
- **manhattan distance (distancia manhattan):** En esta heurística se calcula la distancia Manhattan entre cada sticker del cubo y su posición objetivo en el cubo resuelto. Se calcula como la suma de las diferencias de las coordenadas, filas y columnas, entre la posición actual y la posición objetivo de cada sticker.

  En este caso se ofrece una estimación de la distancia entre el estado actual y el estado resuelto, además de que es rápida de calcular.

La complejidad en este caso sería **O(n^2)**, donde **n** es el número de stickers del cubo, ya que para cada sticker del cubo se realiza una búsqueda lineal en el array de sticker del cubo objetivo para encontrar su posición, lo que lleva a una complejidad cuadrática.

- **count incorrect orientations (conteo de orientaciones incorrectas):** En este caso se cuenta el número de stickers que no están orientados correctamente en relación con el centro de su cara correspondiente, compara cada sticker con el sticker central de la cara y cuenta aquellos que no coinciden.

  En esta heurística uso el concepto de resolución de varios algoritmos que se centran en orientar primero los stickers antes de resolver su posición, además de que proporciona una medida de cuántos stickers necesitan ser orientados correctamente.

  Tiene una complejidad de **O(1)**, ya que realiza operaciones constantes para cada uno de los stickers de la cara, y no depende del tamaño del cubo.

rubiks_solver.py

Complejidad: Explicada en cada algoritmo

- **Breadth First Search:** Para este algoritmo, las transiciones se llevan a cabo con una cola, que se usa para almacenar los estados del cubo y las secuencias de movimientos que se llevaron a cabo para llegar a ese estado. En cada iteración se saca el primer elemento de la cola, se generan todos los estados posibles a partir de los movimientos y se agregan a la cola para explorarlos, repitiendo este proceso hasta llegar al estado resuelto.
  Los estados visitados los guardo en un conjunto para evitar pasar el mismo estado más de una vez.

  Tiene una complejidad en tiempo que en el peor de los casos cada estado se genera y expande una vez, lo que daría como resultado una complejidad de **O(b^d)**, donde **b** es el factor de ramificación y **d** la profundidad de la solución más corta.

  En espacio, su complejidad es alta, ya que en todos los estados generados se almacenan en la memoria hasta que se encuentra la solución, por lo tanto, la complejidad en espacio de igual manera será **O(b^d)**, donde **b** y **d** son los mencionados anteriormente.
- **Best First Search:** En este algoritmo uso una cola de prioridad, para almacenar los estados del cubo y las secuencias de los movimientos que se usaron para llegar a ese estado. La prioridad de cada estado es determinada mediante las heurísticas, que estiman qué tan cerca se está del estado objetivo. En cada iteración se extrae el estado

con la mayor prioridad y se generan todos los estados alcanzables a partir de él aplicando todos los movimientos posibles, para después agregar estos nuevos estados a la cola de prioridad.
Los estados visitados los almaceno un conjunto para evitar visitar un estado más de una vez.

En cuanto a complejidad en tiempo depende de la heurística, en el peor de los casos si la heurística no es buena, la búsqueda puede comportarse de manera similar a la de breadth first search. Sin embargo, si la heurística es buena, la complejidad en tiempo se reduce significativamente.

En espacio la complejidad es similar a la de breadth first search, ya que de igual forma se almacena todos los estados generados en la memoria, pero esta puede ser menor si la heurística permite un descarte efectivo de los estados menos prometedores.

- **A*:** Es similar a best first search, pero se utiliza una función de costo que combina el costo de llegar al estado actual, junto con la estimación heurística. En cada iteración se extrae el estado con el menor costo total y se generan todos los estados alcanzables a partir de él, agregándolos a la cola de prioridad para explorarlos posteriormente.
  De igual forma que los demás, los estados visitados los guardo en un conjunto para evitar su exploración más de una vez.

  Su complejidad en tiempo depende de la heurística como best first search, el mejor caso cuando la heurística es demasiado buena sería de **O(b^d)**, donde b es el factor de ramificación y d es la profundidad de la solución. En el peor caso puede ser tan mala como la de breadth first search.

  En espacio lo que pasa es que su complejidad es casi idéntica a la de best first search, ya que como él, se guardan los estados hasta hallar la solución óptima, pero puede ser menor si la heurística descarta de manera correcta y eficiente.
- **IDA*:** Es una variante de A*, que utiliza un enfoque de búsqueda limitada, y en cada iteración se realiza un búsqueda en profundidad con un límite de profundidad creciente hasta que se encuentra la solución. Durante la búsqueda se utiliza una función de costo que combina el costo real de llegar al estado actual, con la heurística. Decidí usar este algoritmo por su menor costo en memoria que los demás, ya que no ocupa almacenar todos los estados para funcionar adecuadamente.
  Los estados visitados de igual forma los guardo en un conjunto.

  En tiempo, su complejidad es similar a la de A*, pero tiende a ser más eficiente en

términos de memoria, ya que no necesita almacenar todos los estados generados en memoria, su complejidad depende de la heurística y del límite de profundidad alcanzado antes de encontrar la solución.

En espacio, su complejidad es menor a A*, ya que como menciono anteriormente, no se almacenan todos los estados generados en la memoria.

TABLA COMPARATIVA

	Tiempo en segundos											
Movimientos	**Breadth First Search**	**Best First Search**			**A***			**IDA***				
	None	**H1**	**H2**	**H3**	**H1**	**H2**	**H3**	**H1**	**H2**	**H3**		
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00		
2	0.09	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.04	**H1**	**count_misplaced_stickers**
3	0.99	0.00	0.01	0.00	0.00	None	0.01	0.00	None	0.04	**H2**	**manhattan_distance**
4	8.42	0.00	0.01	0.00	0.00		0.01	0.00		2.17	**H3**	**count_incorrect_orientations**
5	45.90	None	None	None	53.30		0.07	0.02		1.18		
6	None				7.41		0.03	None		36.90		
7					None		0.16			9.34		
8							2.73			0.10		
9							1.57			None		
10							83.91					
11							63.69					
12							None					
...												
20	None	600.32			510.67			552.13				