Comparativa de distintas representaciones para búsquedas Cubo Rubik

Pedro Viegas Peñalosa

17 Enero 2022

1 Conceptos Previos

Sabemos que un cubo de Rubik es un puzzle, en el que tenemos bloques que giran sobre un el centro de una cara, pudiendo estas piezas rotarse según la secuencia de movimientos realizada.

Para un cubo de 2×2 tendremos 8 piezas, teniendo cada una 3 posibles rotaciones. Esto nos deja con $8! \times 3^8 = 264.539.520$ posiciones. En primer lugar, vamos a desglosar los posibles movimientos que podemos realizar, con objetivo de poder componer un buen espacio de estados y sus posibles transiciones.

1.1 Notación

Para nombrar las caras del cubo, se usa una notación de tal forma que usamos la inicial del nombre en inglés de la cara. Estas serían:

- $Up \rightarrow U$
- $Down \rightarrow D$
- $Right \rightarrow R$
- $Left \rightarrow L$
- $Front \rightarrow F$
- $Back \rightarrow B$

Los posibles movimientos los he catalogado en:

- Elementales
- Rotaciones
- Secuencias

Iremos explicando cada uno de estos, y como los tendremos en cuenta en nuestra búsqueda según como estos funcionan.

2 Movimientos

2.1 Elementales

Son los giros de cada cara, individualmente. Se les identifica a cada uno, por la inicial del nombre de la cara, usando la notación. Estos giros son siempre en el sentido horario correspondiente si hacemos la traslación de esa cara a la $Front$. Los giros antihorarios se denotan como la inicial correspondiente a la cara, seguida de $'$, leyéndose como prima.

Como se puede deducir, los movimientos elementales primos, se pueden representar como 3 giros consecutivos horarios. Por este mismo motivo podríamos no tenerlos en cuenta en la generación de los estados hijos del actual, reduciendo el número de posibles transiciones, pero teniendo en cuenta estos movimientos al final del algoritmo, post-procesando el resultado de la búsqueda y convirtiendo 3 movimientos consecutivos de la misma cara como un movimiento primo.

2.2 Rotaciones

Por rotaciones denotamos a los giros completos del cubo, ya sea por su rotación del eje $x, yóz$, y estos giros se denotaran con esa misma notación, siendo en sentido horario y existiendo las rotaciones primas.

Teniendo en cuenta como funcionan las rotaciones, podemos darnos cuenta de que realmente una rotación podría ser considerada como el giro de las 3 capas que conforman 1 cara. (representar con una imagen para más claridad)

Por este mismo motivo, vamos a omitir también los giros de las posibles transiciones, con fin de evitar la gran cantidad de posibles transiciones en una búsqueda BFS. Si bien, para A* si podríamos tenerlas en cuenta. Trataremos las rotaciones como los movimientos Elementales primos, post-procesando el resultado del algoritmo y mostrando el número de movimientos que se hace en realidad, ya que las rotaciones no cuentan como un movimiento realmente. (en verdad si, pero explicar mejor si hace falta)

2.3 Secuencias

Las secuencias se definen como la sucesión de movimientos elementales, que dan un resultado concreto (ejemplo, rotar dos aristas, intercambiar piezas...) sin afectar al resto del estado del cubo. Intento probar que haciendo uso de estos en una búsqueda se consigue:

- No afectar a los cálculos heurísticos tanto, ya que al no empeorar el estado con los movimientos intermedios, dejaríamos que nuestro estado empeore internamente, consiguiendo que se llegue a un estado de resolución o cercano a esta.
- Reducir el tiempo de búsqueda de la solución en BFS.

No obstante, para poder hacer uso de las secuencias, tendremos que contemplar 4(una por lado de la cara)*6(número de caras) posibilidades de aplicar la secuencia en cada paso del BFS. Como se puede entender, más que solucionar el problema planteado, estaríamos empeorándolo. Para ello, podemos aplicar un truco, el cual sería añadir como posibles transiciones las 3 rotaciones del cubo, y 1 sola transición por secuencia. Con esto conseguiremos que se pueda aplicar la secuencia en cada una de sus posibles posiciones, ahorrándonos las innecesarias transiciones por cada iteración.

2.4 Resumen

Por tanto, nuestras posibles transiciones entre estados serán:

- Movimientos elementales: U, D, F, B, L, y R.
- Rotaciones: X, Y, Z
- Secuencias predefinidas. Se desglosarán en movimientos elementales al post-procesar el resultado final de la búsqueda.

Los movimientos primos no vamos a tenerlos en consideración en las transiciones posibles, pero si en el post-procesado del resultado obtenido con BFS. Siendo que 3 movimientos consecutivos iguales equivaldría a 1 movimiento primo.

3 Primeros pasos

En un principio definí los estados como dos listas, conteniendo la primera las piezas y la segunda sus orientaciones. La primera lista, de posiciones contendría el número de la pieza que ocupa la posición respectiva al índice de la lista. La segunda lista, indicaría según su índice la orientación (r).

$X : [[1, 2, 3, 4, 5, 6, 7, 8], [r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8]]$
$r \in [-1, 0, 1]$
Solo podríamos decir que una pieza está mal orientada, si esta pieza se encuentra en al menos 1 cara de 1 de sus colores. Pero, en un 2x2 al no existir centros tendríamos dos opciones para poder hacer uso de esta definición de estado.

- Imaginarnos centros de colores. Aunque no existan los centros que nos marquen donde irán los colores, podemos suponer que irán en determinada posición, construyendo la solución sabiendo en qué cara tendremos cada color.
- Determinar el color de una cara una vez haya dos colores o más iguales en la misma cara.

Continuaremos con la primera opción asumiendo las limitaciones y desventajas de la misma, ya que si por alguna casualidad el cubo en su estado no resuelto tuviera una cara o más resueltas, teniendo una solución trivial, tendríamos que resolverlo entero sin aprovechar de la ventajosa situación del estado.

Aun así, comentamos que haciendo uso de la segunda opción, obtendremos una mejora considerable, ya que podemos asegurar que siempre habrá 2 o más pegatinas iguales en la misma cara.

Demostración. RA

Suponiendo que hay un caso en el que no hay 2 o más pegatinas iguales en la misma cara, por tanto, tendríamos que realizar una coloración usando 6 colores, pero 4 veces máximo cada uno, de 6 K_4. Podemos darnos cuenta de que nos faltarían colores para poder hacer la coloración legal.

Por tanto, para construir la primera cara solo habría que rotar una pareja o única pieza. Aunque está bien que tengamos estos casos en cuenta, ya que el algoritmo de BFS podría encaminarse así, realmente no tendremos control sobre la ejecución del mismo, siendo estos casos posibles a implementar si siguiésemos algún tipo de problema de satisfacción de restricciones.

3.1 Definición de transiciones

$T : \{Mov.Elementales, Rotaciones\}$
$Mov.Elementales : \{R, L, B, F, U, D\}$

$R : 1- > 2- > 6- > 5- > 1$ Rotación de las piezas para el movimiento R
$L : 4- > 8- > 7- > 3- > 4$ " L
$B : 2- > 3- > 7- > 6- > 2$ " B
$F : 1- > 5- > 8- > 4- > 1$ " F
$U : 1- > 4- > 3- > 2- > 1$ " U
$D : 8- > 5- > 6- > 7- > 8$ " D

3.2 Problemas

Como no podemos decir que orientación tiene una pieza hasta no empezar a plasmar los colores de las caras, no estaríamos guardando coherentemente los estados de las piezas del cubo. Debemos pensar un nuevo espacio de estados al cual aplicar las transiciones anteriores (por índices), y en el que guardemos el estado de cada pieza.

4 Espacio de estado por cadenas

$S_final \in X : [$"bar","ban","bvn","bvr","yar","yan","yvn","yvr"$]$

De tal forma que la primera letra representará el color de las caras Up o Down, la segunda Right o Left y la última Front o Back. De esta forma podemos almacenar correctamente el estado de todas las piezas, ya que solo necesitamos 3 "pegatinas" que guardar por pieza, y que una pieza tenga pegatina Right excluye de que tenga pegatina Left (igual para Front y Back, o Up y Down).

Ahora solo debemos aplicar las transiciones de T antes definidas, pero con un matiz diferente, y es que si solo desplazásemos las piezas sin rotarlas, no estaríamos realizando correctamente los movimientos elementales.

Una vez definido nuestro espacio de estado X y las transiciones T, para comenzar nuestra búsqueda por BFS nos hará falta obtener cuando un estado es final.

Tal y como hemos definido nuestro estado, podemos darnos cuenta de que:

- Las primeras letras de las 4 primeras cadenas forman la cara Up.
- Las primeras letras de las 4 últimas cadenas forman la cara Down
- Las segundas letras de las dos primeras cadenas, la 5^o y 6^o forman la cara Right.

- Las segundas letras de las cadenas 3^{o}, 4^{o}, 7^{o} y 8^{o} forman la cara Left.
- Las terceras letras de las cadenas 1^{o}, 4^{o}, 5^{o} y 8^{o} forman la cara Front.
- Las terceras letras de las cadenas 2^{o}, 3^{o}, 6^{o} y 7^{o} forman la cara Back.

Solo tendríamos que hacer la comprobación de que todas las letras que formen cada cara sean iguales, para así poder decir que estamos en un estado final. Podemos darnos cuenta, que gracias a esta nueva definición de estados, no nos estamos limitando a que los colores de cada cara estén establecidos en determinada posición antes de realizar la búsqueda BFS. Si no que, cualquier estado que cumpla con la condición será final. Y como las posibles transiciones son movimientos físicamente posibles en el cubo, los estados resultantes siempre serán válidos.

5 Creación del programa

Comenzando la programación de este problema de búsquedas, nos damos cuenta de que programar a mano las transiciones de los movimientos (sean secuencias o Mov. Elementales) es muy tedioso y confuso en un principio. Es por ello, que una de las primeras funciones desarrolladas fue la diferenciaSecuencia(estadoInicial, estadoFinal).

Esta función recibe dos estados, y nos muestra los pasos dados para pasar del Inicial al Final. Estos pasos no se corresponden con movimientos elementales, pero nos sirven para obtener las permutaciones de una secuencia o de un movimiento elemental. También nos indica las rotaciones en las piezas realizadas, y que piezas se mantienen en la misma posición y rotación.

5.1 Clase Estado

La clase estada contiene todo lo necesario para nuestro espacio de estados definido. Cada estado tendrá: Valor, Transición realizada para llegar a este estado, Profundidad en el árbol, Camino seguido, y una variable que almacena si es final o no.

La profundidad la usaremos más adelante, a la hora de ejecutar el BFS. Y nos será muy útil para hacer una poda del árbol de búsqueda, ya que como se ha demostrado ([1]), podríamos establecer un límite en la profundidad en 20 (algo más en nuestro caso, ya que contemplamos los movimientos primos como 3 mov. consecutivos normales, por tanto 20 mov. primos serían 60 movimientos.) con la idea de que, si la solución tiene más de 20 movimientos, no nos interesará. Esto romperá ciclos largos en los que se podría bifurcar mucho nuestro árbol. Aunque es cierto que esto sería realmente bueno en una aplicación de DFS.

Camino, será el camino recorrido (transiciones realizadas) hasta llegar al estado actual.

Final, almacenará el resultado de aplicar el método antes comentado para saber si un estado es o no final. Lo usaremos para poner fin al BFS cuando lleguemos a un estado final.

Por otro lado, tendremos dos funciones parecidas, pero con un uso diferente:

- rota(movimiento): La cual aplica el movimiento a un cubo, modificando su valor, pero sin almacenar en Path y Depth. Este será usado a la hora de realizar el Scramble (desordenar el cubo), partiendo del cubo resuelto realizaremos los movimientos leídos por consola y mostraremos el cubo tras realizar los movimientos del Scramble.
- aplicaTransicion(movimiento): Aplica el movimiento a un cubo, pero devolviendo un nuevo elemento de la clase estado, concatenando su Path al movimiento realizado y aumentando su Depth. Esta será la utilizada en el BFS.

Esta clase también contiene la función encargada de mostrar por pantalla una vista simplificada en 2D del estado, mostrando los colores de cada pegatina. Ante la falta del color Naranja en los colores ASCII usados, este ha sido sustituido por el Magenta.

5.2 Clase rubik

Esta clase contiene toda la lógica de las rotaciones y movimientos elementales, así como la función diferenciaSecuencia() y la usada para el post-procesado del resultado del BFS. Así mismo, en esta clase han sido definidos los valores para la creación de estados tanto ya resueltos (solved_Yellow_Up, un estado resuelto con la cara Amarilla arriba, la roja al frente y la verde a la derecha), como estados con los diferentes movimientos elementales aplicados.

Todos estos han sido usados para probar la función diferenciaSecuencia() y así programar las transiciones pertinentes.

5.3 Clase BFS

Esta clase contiene el algoritmo de BFS en sí, como la lógica de ejecución del programa. Este es el archivo principal a ejecutar para probar el algoritmo.

6 Ejecución del Programa

Tras ejecutar el archivo BFS.py, se nos mostrará por consola la vista en 2D del cubo resuelto (estado solved_Yellow_Up) para poder orientarnos si tenemos algún cubo a mano para seguir la ejecución del programa con este. La cara de arriba será la correspondiente a la cara Back, justo debajo estará Up, y seguida Front. De izquierda a derecha veremos las caras Left, Up, Right y Down.

Con esta explicación de la vista 2D, continuemos esta guía a través de la ejecución del programa. Lo siguiente que aparecerá, es un mensaje pidiéndonos que introduzcamos un Scramble. Un Scramble es la secuencia de movimientos (en notación del Cubo de Rubik, por fidelidad a la comunidad de este) que se seguirá de Izquierda a Derecha.

Por ejemplo, el Scramble "URLBFD" realizará el movimiento U, R, L, B, F y D.

Una vez introducido el Scramble, se nos mostrará el estado después de desordenar el cubo, nuevamente en la vista 2D de consola. Se nos pedirá a continuación presionar Enter para iniciar la búsqueda BFS. Es recomendable no introducir Scarmbles muy largos (entre 6-7 es aceptable),

debido a lo rápido que crece el árbol de búsquedas, y el enorme tiempo de procesamiento que se llevaría.

7 Resultados

Para comprobar la ejecución el BFS para los distintos conjuntos de movimientos, se han creado 3 líneas de return en la función aplicaTransiciones de BFS.py. La primera realiza todos los movimientos elementales y rotaciones; la segunda dos Permutaciones o Secuencias, y las rotaciones; y la última todos los movimientos definidos.

En un primer momento, partimos de la hipótesis de que el hecho de añadir un par de secuencias predefinidas haría que la búsqueda fuese más corta, ya que estaría compactando numerosos movimientos en una sola transición de estado. Comprobando en la ejecución del BFS, comprobamos que no solo tiene una ejecución más lenta, sino que no nos llega a devolver ningún resultado, o no encuentra una solución para el número de profundidad máximo definido.

En cambio, la ejecución con movimientos elementales reducidos (solo R, U y F) y rotaciones sí que nos entrega un resultado más o menos rápido, dependiendo del Scramble introducido (a partir de Scrambles de 6 movimientos puede quedarse atascado buscando una solución).

Por último, si metemos todos los movimientos definidos, como es obvio no conseguiremos que se ejecute en un tiempo aceptable, debido a la gran amplitud del árbol que crece exponencialmente.

Podriamos entonces plantear una nueva ejecución, sin hacer uso de las rotaciones, que si bien agilizan el proceso de transiciones compactando otras y permitiendo la ejecución de algunos movimientos elementales primos en menos pasos (ya que de por sí, no están implementados, sino que son la ejecución consecutiva de 3 movimientos elementales iguales), también hace que se amplíe mucho en anchura el árbol de búsqueda. De esta nueva forma de aplicar las transiciones, sí que obtenemos un resultado para algunos Scrambles de hasta 5-6 movimientos, pero no para más.

8 Conclusiones

Partiendo de la premisa de obtener un resultado a la resolución de un cubo de Rubik 2x2 en un tiempo aceptable, hemos intentado aplicar distintas técnicas de transiciones que nos permitirían recorrer todo el espacio de estados, y que hagan que este pueda llegar a reducirse en anchura. Y es cierto, que para alguna de estas configuraciones hemos obtenido un resultado en tiempos más o menos aceptables, aunque sea para Scrambles muy cortitos.

La solución a este problema de juguete, no tiene comparación a la resolución de cubos competitivamente, ni aun siendo 2x2, ya que en las competiciones los Scrambles son de 20 movimientos(a partir de los cuales ya se considera que no se está desordenando más el cubo). Es, por tanto, que el problema de resolver un cubo 3x3 se nos queda muy grande con una busqueda BFS, en cuanto a capacidad de computación.

Concluimos con el aprendizaje y limitaciones de una búsqueda en profundidad cuando el espacio de estados es muy grande o, aún reduciendo este, cuando este crece de forma exponencial o similar.

9 Continuación del proyecto

Para concluir esta investigación, me gustaría comentar ciertos temas que por tiempo se han quedado en el tintero, pero que seguiré investigando a fin de obtener una forma eficiente de resolver ya no solo un 2x2, sino idealmente, una generalización de cualquier puzzle en 3 dimensiones similar a estos.

Una buena técnica a aplicar a un 2x2, podría ser un ACO. Como el espacio de estados permite alcanzar cualquier estado desde cualquier estado, y los movimientos son reversibles, es decir, las aristas no son dirigidas. Podríamos generar en memoria una sola vez el espacio de estados completo, realizando las optimizaciones necesarias como eliminar simetrías, etc. Con esto, solo tendríamos que poner a las hormigas a buscar desde el estado resuelto, hasta el estado desordenado inicial, devolviendo el camino al revés.

También sería interesante la búsqueda por Templado Simulado, a fin de obtener no una solución directamente, sino un estado en el cual si pueda ser eficiente un BFS con secuencias predefinidas.

Para finalizar, comentar que estas pruebas con BFS podrían ser realizadas con este mismo código, haciendo unas simples modificaciones. Y es que un 2x2 es como considerar solo las esquinas de un 3x3, por tanto, solo tendríamos que añadir la lógica de las aristas a los movimientos ya definidos, siendo estás más sencillas al solo tener 2 pegatinas por pieza. Es muy probable que, para la inmensidad del espacio de estados de un 3x3, las técnicas de optimización probadas en esta investigación sí que obtengan mejora de rendimiento en comparación a realizar todos los movimientos posibles de forma voraz.

References

[1] Ruwix, https://ruwix.com/the-rubiks-cube/gods-number/

[2] Proffessor, Erik Demaine, https://www.youtube.com/watch?v=qeqWGnEyFs0

[3] https://www.cubelelo.com/blogs/cubing/how-to-solve-rubiks-cube-in-20-moves

[4] No pude encontrar el nombre del autor, http://www.facweb.iitkgp.ac.in/ sourav/BFS.pdf

[5] MackHartley, Me fue de gran ayuda para inspirarme en nuevas formas de hacer y ver las cosas, https://github.com/MackHartley/2x2RubixSolver